曦云® 系列通用计算 GPU

mcThrust API 参考

CSRD-23020-020-F3_V03　　2024-03-15

沐曦专有和三级保密信息　　本文档受 NDA 管控

飞腾信息 MetaX Confidential 2024-11-08 15:00:00

版权所有 © 沐曦集成电路（上海）有限公司

声明

版权所有 ©2023-2024 沐曦集成电路（上海）有限公司。保留所有权利。

本文档中呈现的信息属于沐曦集成电路（上海）有限公司和/或其附属公司（以下统称为“沐曦”），非经沐曦事先书面许可，任何实体或个人均不得获得本文档的副本，且无权以任何方式处理本文档，包括但不限于使用、复制、修改、合并、出版、发行、销售或传播本文档的部分或全部。

本文档内容仅供参考，不提供任何形式的、明示或暗示的保证，包括但不限于对适销性、适用于任何目的和/或不侵权的保证。在任何情况下，沐曦均不对因本文档引起的、由本文档造成的、或与之相关的任何索赔、损害或其他责任负责。

沐曦保留自行决定随时更改、修改、添加或删除本文档的部分或全部的权利。沐曦保留最终解释权。

沐曦、MetaX 和其他沐曦图标是沐曦的商标。本文档中提及的所有其他商标和商品名称均为其各自所有者的财产。

飞腾信息 MetaX Confidential
2024-11-18 15:00:00

更新记录

版本	日期	更新说明
V03	2024-03-15	新增曦云®系列 GPU 产品信息
V02	2023-12-29	描述性修改及格式修正
V01	2023-10-16	正式版本首次发布

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

目录

1 简介 **1**
1.1 安装 . . . 1
1.2 Hello thrust . . . 1

2 使用 Thrust API **3**
2.1 算法 . . . 3
2.1.1 复制（Copying） . . . 3
2.1.1.1 复制 . . . 3
2.1.1.2 聚集（Gathering） . . . 16
2.1.1.3 分散（Scattering） . . . 25
2.1.2 合并（Merging） . . . 34
2.1.2.1 thrust::merge . . . 36
2.1.2.2 thrust::merge . . . 38
2.1.2.3 thrust::merge . . . 39
2.1.2.4 thrust::merge . . . 40
2.1.2.5 thrust::merge_by_key . . . 42
2.1.2.6 thrust::merge_by_key . . . 44
2.1.2.7 thrust::merge_by_key . . . 45
2.1.2.8 thrust::merge_by_key . . . 47
2.1.3 前缀和（Prefix Sum） . . . 49
2.1.3.1 前缀和 . . . 49
2.1.3.2 分段前缀和（Segmented Prefix Sums） . . . 61
2.1.3.3 变换前缀和（Transformed Prefix Sum） . . . 83
2.1.4 归约（Reductions） . . . 89
2.1.4.1 归约 . . . 89
2.1.4.2 比较（Comparison） . . . 105
2.1.4.3 计数（Counting） . . . 110
2.1.4.4 最值（Extrema） . . . 114
2.1.4.5 逻辑（Logical） . . . 128
2.1.4.6 谓词（Predicate） . . . 134
2.1.4.7 变换归约（Transformed Reduction） . . . 146
2.1.5 重新排序（Reordering） . . . 154
2.1.5.1 分区（Partitioning） . . . 154
2.1.5.2 洗牌（Shuffling） . . . 179
2.1.5.3 流压缩（Stream Compaction） . . . 183
2.1.6 搜索（Searching） . . . 229
2.1.6.1 搜索 . . . 229
2.1.6.2 二分搜索（Binary Search） . . . 245
2.1.7 集合运算（Set Operation） . . . 287
2.1.7.1 thrust::set_difference . . . 296
2.1.7.2 thrust::set_difference . . . 297
2.1.7.3 thrust::set_difference . . . 299
2.1.7.4 thrust::set_difference . . . 300
2.1.7.5 thrust::set_intersection . . . 301
2.1.7.6 thrust::set_intersection . . . 303
2.1.7.7 thrust::set_intersection . . . 304

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

2.1.7.8 thrust::set_intersection . . . 306
2.1.7.9 thrust::set_symmetric_difference . . . 307
2.1.7.10 thrust::set_symmetric_difference . . . 309
2.1.7.11 thrust::set_symmetric_difference . . . 310
2.1.7.12 thrust::set_symmetric_difference . . . 312
2.1.7.13 thrust::set_union . . . 313
2.1.7.14 thrust::set_union . . . 315
2.1.7.15 thrust::set_union . . . 316
2.1.7.16 thrust::set_union . . . 318
2.1.7.17 thrust::set_difference_by_key . . . 319
2.1.7.18 thrust::set_difference_by_key . . . 321
2.1.7.19 thrust::set_difference_by_key . . . 323
2.1.7.20 thrust::set_difference_by_key . . . 325
2.1.7.21 thrust::set_intersection_by_key . . . 327
2.1.7.22 thrust::set_intersection_by_key . . . 329
2.1.7.23 thrust::set_intersection_by_key . . . 330
2.1.7.24 thrust::set_intersection_by_key . . . 332
2.1.7.25 thrust::set_symmetric_difference_by_key . . . 334
2.1.7.26 thrust::set_symmetric_difference_by_key . . . 336
2.1.7.27 thrust::set_symmetric_difference_by_key . . . 338
2.1.7.28 thrust::set_symmetric_difference_by_key . . . 340
2.1.7.29 thrust::set_union_by_key . . . 342
2.1.7.30 thrust::set_union_by_key . . . 344
2.1.7.31 thrust::set_union_by_key . . . 346
2.1.7.32 thrust::set_union_by_key . . . 348
2.1.8 排序（Sorting） . . . 349
2.1.8.1 thrust::sort . . . 352
2.1.8.2 thrust::sort . . . 353
2.1.8.3 thrust::sort . . . 354
2.1.8.4 thrust::sort . . . 354
2.1.8.5 thrust::stable_sort . . . 355
2.1.8.6 thrust::stable_sort . . . 356
2.1.8.7 thrust::stable_sort . . . 357
2.1.8.8 thrust::stable_sort . . . 358
2.1.8.9 thrust::sort_by_key . . . 359
2.1.8.10 thrust::sort_by_key . . . 360
2.1.8.11 thrust::sort_by_key . . . 361
2.1.8.12 thrust::sort_by_key . . . 362
2.1.8.13 thrust::stable_sort_by_key . . . 363
2.1.8.14 thrust::stable_sort_by_key . . . 364
2.1.8.15 thrust::stable_sort_by_key . . . 365
2.1.8.16 thrust::stable_sort_by_key . . . 366
2.1.9 变换（Transformation） . . . 367
2.1.9.1 变换 . . . 367
2.1.9.2 填充（Filling） . . . 399
2.1.9.3 修改（Modifying） . . . 408
2.1.9.4 替换（Replacing） . . . 413
2.2 容器类（Container Class） . . . 432
2.2.1 主机容器（Host Container） . . . 432
2.2.1.1 thrust::host_vector . . . 432
2.2.1.2 thrust::swap . . . 432
2.2.2 容器（Container） . . . 433
2.2.2.1 thrust::device_vector . . . 433
2.2.2.2 thrust::swap . . . 433
2.3 函数对象类（Function Objects Class） . . . 433
2.3.1 函数对象适配器（Function Object Adaptor） . . . 433
2.3.1.1 thrust::not1 . . . 434
2.3.1.2 thrust::not2 . . . 435

飞腾信息技术有限公司 FTIC Confidential 2024-11-18 15:00:00

2.3.1.3 thrust::make_zip_function . . . 435
2.3.2 占位符对象（Placeholder Object） . . . 435
2.3.3 预定义函数对象（Predefined Function Object） . . . 436
2.3.3.1 算术运算（Arithmetic Operation） . . . 436
2.3.3.2 位运算（Bitwise Operation） . . . 437
2.3.3.3 比较运算（Comparison Operation） . . . 438
2.3.3.4 通用标识运算（Generalized Identity Operation） . . . 439
2.3.3.5 逻辑运算（Logical Operation） . . . 440
2.4 迭代器（Iterator） . . . 441
2.4.1 thrust::advance . . . 442
2.4.2 thrust::distance . . . 442
2.4.3 高级迭代器（Fancy Iterator） . . . 443
2.4.3.1 thrust::make_constant_iterator . . . 446
2.4.3.2 thrust::make_constant_iterator . . . 446
2.4.3.3 thrust::make_counting_iterator . . . 447
2.4.3.4 thrust::make_discard_iterator . . . 447
2.4.3.5 thrust::make_permutation_iterator . . . 447
2.4.3.6 thrust::make_reverse_iterator . . . 448
2.4.3.7 thrust::make_transform_input_output_iterator . . . 448
2.4.3.8 thrust::make_transform_iterator . . . 448
2.4.3.9 thrust::make_transform_output_iterator . . . 449
2.4.3.10 thrust::make_zip_iterator . . . 449
2.4.3.11 thrust::make_zip_iterator . . . 449
2.4.4 迭代器标签（Iterator Tag） . . . 450
2.4.4.1 迭代器标签类（Iterator Tag Class） . . . 450
2.4.4.2 thrust::input_host_iterator_tag . . . 451
2.4.4.3 thrust::output_host_iterator_tag . . . 451
2.4.4.4 thrust::forward_host_iterator_tag . . . 452
2.4.4.5 thrust::bidirectional_host_iterator_tag . . . 452
2.4.4.6 thrust::random_access_host_iterator_tag . . . 452
2.5 内存管理（Memory Management） . . . 453
2.5.1 内存管理 . . . 453
2.5.1.1 thrust::device_reference . . . 455
2.5.1.2 thrust::device_ptr . . . 455
2.5.1.3 thrust::device_delete . . . 455
2.5.1.4 thrust::device_free . . . 456
2.5.1.5 thrust::device_malloc . . . 456
2.5.1.6 thrust::device_malloc . . . 457
2.5.1.7 thrust::device_new . . . 457
2.5.1.8 thrust::device_new . . . 458
2.5.1.9 thrust::device_new . . . 458
2.5.1.10 thrust::operator<< . . . 458
2.5.1.11 thrust::device_pointer_cast . . . 459
2.5.1.12 thrust::device_pointer_cast . . . 459
2.5.1.13 thrust::swap . . . 460
2.5.1.14 thrust::operator<< . . . 460
2.5.1.15 thrust::malloc . . . 460
2.5.1.16 thrust::malloc . . . 461
2.5.1.17 thrust::get_temporary_buffer . . . 462
2.5.1.18 thrust::free . . . 463
2.5.1.19 thrust::return_temporary_buffer . . . 463
2.5.1.20 thrust::raw_pointer_cast . . . 464
2.5.1.21 thrust::raw_reference_cast . . . 465
2.5.1.22 thrust::raw_reference_cast . . . 465
2.5.2 分配器（Allocator） . . . 466
2.5.2.1 thrust::mr::polymorphic_allocator . . . 467
2.5.2.2 thrust::mr::operator== . . . 467
2.5.2.3 thrust::mr::operator!= . . . 467

飞腾信息技术有限公司 FTXX Confidential 2024-11-18 15:00:00

2.5.3 内存资源（Memory Resources） . . . 467
2.5.3.1 thrust::system::cpp::memory_resource . . . 469
2.5.3.2 thrust::system::cpp::universal_memory_resource . . . 470
2.5.3.3 thrust::system::cpp::universal_host_pinned_memory_resource . . . 470
2.5.3.4 thrust::system::omp::memory_resource . . . 470
2.5.3.5 thrust::system::omp::universal_memory_resource . . . 470
2.5.3.6 thrust::system::omp::universal_host_pinned_memory_resource . . . 470
2.5.3.7 thrust::system::tbb::memory_resource . . . 470
2.5.3.8 thrust::system::tbb::universal_memory_resource . . . 470
2.5.3.9 thrust::system::tbb::universal_host_pinned_memory_resource . . . 470
2.5.3.10 thrust::universal_ptr . . . 471
2.5.3.11 thrust::mr::tls_disjoint_pool . . . 471
2.5.3.12 thrust::mr::operator== . . . 471
2.5.3.13 thrust::mr::operator!= . . . 472
2.5.3.14 thrust::mr::get_global_resource . . . 472
2.5.3.15 thrust::mr::tls_pool . . . 472
2.6 数值（Numeric） . . . 472
2.6.1 复数（Cmplex Number） . . . 472
2.6.1.1 thrust::complex::complex . . . 479
2.6.1.2 thrust::complex::complex . . . 480
2.6.1.3 thrust::complex::complex . . . 480
2.6.1.4 thrust::complex::complex . . . 480
2.6.1.5 thrust::complex::complex . . . 480
2.6.1.6 thrust::complex::complex . . . 480
2.6.1.7 thrust::complex::complex . . . 481
2.6.1.8 thrust::complex::operator= . . . 481
2.6.1.9 thrust::complex::operator= . . . 481
2.6.1.10 thrust::complex::operator= . . . 481
2.6.1.11 thrust::complex::operator= . . . 482
2.6.1.12 thrust::complex::operator= . . . 482
2.6.1.13 thrust::complex::operator+= . . . 482
2.6.1.14 thrust::complex::operator-= . . . 482
2.6.1.15 thrust::complex::operator*= . . . 483
2.6.1.16 thrust::complex::operator/= . . . 483
2.6.1.17 thrust::complex::operator+= . . . 483
2.6.1.18 thrust::complex::operator-= . . . 483
2.6.1.19 thrust::complex::operator*= . . . 484
2.6.1.20 thrust::complex::operator/= . . . 484
2.6.1.21 thrust::complex::real . . . 484
2.6.1.22 thrust::complex::imag . . . 484
2.6.1.23 thrust::complex::real . . . 484
2.6.1.24 thrust::complex::imag . . . 485
2.6.1.25 thrust::complex::real . . . 485
2.6.1.26 thrust::complex::imag . . . 485
2.6.1.27 thrust::complex::real . . . 485
2.6.1.28 thrust::complex::imag . . . 485
2.6.1.29 thrust::complex::complex< T > . . . 486
2.6.1.30 thrust::abs . . . 486
2.6.1.31 thrust::arg . . . 486
2.6.1.32 thrust::norm . . . 486
2.6.1.33 thrust::conj . . . 486
2.6.1.34 thrust::polar . . . 487
2.6.1.35 thrust::proj . . . 487
2.6.1.36 thrust::operator+ . . . 487
2.6.1.37 thrust::operator+ . . . 487
2.6.1.38 thrust::operator+ . . . 488
2.6.1.39 thrust::operator- . . . 488
2.6.1.40 thrust::operator- . . . 488

飞腾信息技术有限公司 Confidential 2024-12-13 15:00:00

2.6.1.41 thrust::operator- . . . 489
2.6.1.42 thrust::operator* . . . 489
2.6.1.43 thrust::operator* . . . 489
2.6.1.44 thrust::operator* . . . 490
2.6.1.45 thrust::operator/ . . . 490
2.6.1.46 thrust::operator/ . . . 490
2.6.1.47 thrust::operator/ . . . 491
2.6.1.48 thrust::operator+ . . . 491
2.6.1.49 thrust::operator- . . . 491
2.6.1.50 thrust::exp . . . 491
2.6.1.51 thrust::log . . . 492
2.6.1.52 thrust::log10 . . . 492
2.6.1.53 thrust::pow . . . 492
2.6.1.54 thrust::pow . . . 492
2.6.1.55 thrust::pow . . . 493
2.6.1.56 thrust::sqrt . . . 493
2.6.1.57 thrust::cos . . . 493
2.6.1.58 thrust::sin . . . 493
2.6.1.59 thrust::tan . . . 494
2.6.1.60 thrust::cosh . . . 494
2.6.1.61 thrust::sinh . . . 494
2.6.1.62 thrust::tanh . . . 494
2.6.1.63 thrust::acos . . . 494
2.6.1.64 thrust::asin . . . 495
2.6.1.65 thrust::atan . . . 495
2.6.1.66 thrust::acosh . . . 495
2.6.1.67 thrust::asinh . . . 495
2.6.1.68 thrust::atanh . . . 496
2.6.1.69 thrust::operator<< . . . 496
2.6.1.70 thrust::operator>> . . . 496
2.6.1.71 thrust::operator== . . . 497
2.6.1.72 thrust::operator== . . . 497
2.6.1.73 thrust::operator== . . . 497
2.6.1.74 thrust::operator== . . . 497
2.6.1.75 thrust::operator== . . . 498
2.6.1.76 thrust::operator!= . . . 498
2.6.1.77 thrust::operator!= . . . 498
2.6.1.78 thrust::operator!= . . . 499
2.6.1.79 thrust::operator!= . . . 499
2.6.1.80 thrust::operator!= . . . 499
2.7 并行执行策略（Parallel Execution Policy） . . . 499
2.7.1 thrust::host . . . 500
2.7.2 thrust::device . . . 501
2.8 随机数生成（Random Number Generation） . . . 502
2.8.1 带预定义参数的随机数引擎（Random Number Engines with Predefined Parameter） . . . 502
2.8.1.1 thrust::random::ranlux24 . . . 502
2.8.1.2 thrust::random::ranlux48 . . . 503
2.8.1.3 thrust::random::taus88 . . . 503
2.8.1.4 thrust::random::default_random_engine . . . 503
2.8.1.5 thrust::random::minstd_rand0 . . . 503
2.8.1.6 thrust::random::minstd_rand . . . 503
2.8.1.7 thrust::random::ranlux24_base . . . 504
2.8.1.8 thrust::random::ranlux48_base . . . 504
2.8.2 随机数分布类模板（Random Number Distributions Class Template） . . . 504
2.8.2.1 thrust::random::normal_distribution . . . 506
2.8.2.2 thrust::random::uniform_int_distribution . . . 506
2.8.2.3 thrust::random::uniform_real_distribution . . . 506

飞腾信息技术有限公司 FeiTeng Information Technology Co., Ltd. Confidential
2024-11-18 15:00:00

2.8.2.4 thrust::random::operator== . . . 506
2.8.2.5 thrust::random::operator!= . . . 506
2.8.2.6 thrust::random::operator<< . . . 507
2.8.2.7 thrust::random::operator>> . . . 507
2.8.2.8 thrust::random::operator== . . . 507
2.8.2.9 thrust::random::operator!= . . . 508
2.8.2.10 thrust::random::operator<< . . . 508
2.8.2.11 thrust::random::operator>> . . . 508
2.8.2.12 thrust::random::operator== . . . 509
2.8.2.13 thrust::random::operator!= . . . 509
2.8.2.14 thrust::random::operator<< . . . 509
2.8.2.15 thrust::random::operator>> . . . 510
2.8.3 随机数引擎适配器类模板（Random Number Engine Adaptor Class Template） 510
2.8.3.1 thrust::random::discard_block_engine . . . 512
2.8.3.2 thrust::random::xor_combine_engine . . . 512
2.8.3.3 thrust::random::operator== . . . 512
2.8.3.4 thrust::random::operator!= . . . 512
2.8.3.5 thrust::random::operator<< . . . 513
2.8.3.6 thrust::random::operator>> . . . 513
2.8.3.7 thrust::random::operator== . . . 513
2.8.3.8 thrust::random::operator!= . . . 514
2.8.3.9 thrust::random::operator<< . . . 514
2.8.3.10 thrust::random::operator>> . . . 514
2.8.4 随机数引擎类模板（Random Number Engine Adaptor Class Template） . . . 515
2.8.4.1 thrust::random::linear_congruential_engine . . . 517
2.8.4.2 thrust::random::linear_feedback_shift_engine . . . 518
2.8.4.3 thrust::random::subtract_with_carry_engine . . . 518
2.8.4.4 thrust::random::operator== . . . 518
2.8.4.5 thrust::random::operator!= . . . 518
2.8.4.6 thrust::random::operator<< . . . 519
2.8.4.7 thrust::random::operator>> . . . 519
2.8.4.8 thrust::random::operator== . . . 519
2.8.4.9 thrust::random::operator!= . . . 520
2.8.4.10 thrust::random::operator<< . . . 520
2.8.4.11 thrust::random::operator>> . . . 521
2.8.4.12 thrust::random::operator== . . . 521
2.8.4.13 thrust::random::operator!= . . . 521
2.8.4.14 thrust::random::operator<< . . . 522
2.8.4.15 thrust::random::operator>> . . . 522
2.9 系统（System） . . . 523
2.9.1 系统 . . . 523
2.9.1.1 thrust::system::cpp::pointer . . . 523
2.9.1.2 thrust::system::cpp::universal_pointer . . . 524
2.9.1.3 thrust::system::cpp::reference . . . 524
2.9.1.4 thrust::system::omp::pointer . . . 524
2.9.1.5 thrust::system::omp::universal_pointer . . . 525
2.9.1.6 thrust::system::omp::reference . . . 525
2.9.1.7 thrust::system::tbb::pointer . . . 525
2.9.1.8 thrust::system::tbb::universal_pointer . . . 526
2.9.1.9 thrust::system::tbb::reference . . . 526
2.9.2 系统诊断（System Diagnostics） . . . 526
2.9.2.1 thrust::system::is_error_code_enum . . . 527
2.9.2.2 thrust::system::is_error_condition_enum . . . 527
2.9.2.3 thrust::system::is_error_condition_enum< errc::errc_t > . . . 527
2.9.2.4 thrust::system::error_category . . . 527
2.9.2.5 thrust::system::error_code . . . 527
2.9.2.6 thrust::system::error_condition . . . 527
2.9.2.7 thrust::system::system_error . . . 528

飞腾信息技术 Confidential 2024-11-13 15:00:00

2.9.2.8 thrust::system::generic_category . . . 528
2.9.2.9 thrust::system::system_category . . . 528
2.9.2.10 thrust::system::make_error_code . . . 528
2.9.2.11 thrust::system::operator< . . . 528
2.9.2.12 thrust::system::operator<< . . . 529
2.9.2.13 thrust::system::make_error_condition . . . 529
2.9.2.14 thrust::system::operator< . . . 529
2.9.2.15 thrust::system::operator== . . . 529
2.9.2.16 thrust::system::operator== . . . 529
2.9.2.17 thrust::system::operator== . . . 529
2.9.2.18 thrust::system::operator== . . . 530
2.9.2.19 thrust::system::operator!= . . . 530
2.9.2.20 thrust::system::operator!= . . . 530
2.9.2.21 thrust::system::operator!= . . . 530
2.9.2.22 thrust::system::operator!= . . . 530
2.9.2.23 thrust::system::system_error::system_error . . . 531
2.9.2.24 thrust::system::system_error::system_error . . . 531
2.9.2.25 thrust::system::system_error::system_error . . . 531
2.9.2.26 thrust::system::system_error::system_error . . . 531
2.9.2.27 thrust::system::system_error::system_error . . . 532
2.9.2.28 thrust::system::system_error::system_error . . . 532
2.9.2.29 thrust::system::system_error::~system_error . . . 532
2.9.2.30 thrust::system::system_error::code . . . 532
2.9.2.31 thrust::system::system_error::what . . . 533
2.10 工具库（Utility） . . . 533
2.10.1 Pair . . . 533
2.10.1.1 thrust::pair . . . 534
2.10.1.2 thrust::tuple_element . . . 534
2.10.1.3 thrust::tuple_size . . . 534
2.10.1.4 thrust::operator== . . . 534
2.10.1.5 thrust::operator< . . . 535
2.10.1.6 thrust::operator!= . . . 535
2.10.1.7 thrust::operator> . . . 535
2.10.1.8 thrust::operator<= . . . 536
2.10.1.9 thrust::operator>= . . . 536
2.10.1.10thrust::swap . . . 537
2.10.1.11thrust::make_pair . . . 537
2.10.2 交换（Swap） . . . 537
2.10.2.1 thrust::swap . . . 538
2.10.3 元组（Tuple） . . . 538
2.10.3.1 thrust::tuple_element . . . 539
2.10.3.2 thrust::tuple_size . . . 539
2.10.3.3 thrust::tuple . . . 539
2.10.3.4 thrust::get . . . 539
2.10.3.5 thrust::get . . . 539
2.10.3.6 thrust::make_tuple . . . 540
2.10.3.7 thrust::make_tuple . . . 540
2.10.3.8 thrust::tie . . . 541
2.10.3.9 thrust::tie . . . 541
2.10.3.10thrust::swap . . . 541
2.10.4 Type Traits . . . 542
2.10.4.1 thrust::proclaim_contiguous_iterator . . . 542
2.10.4.2 thrust::conjunction_value . . . 542
2.10.4.3 thrust::disjunction_value . . . 542
2.10.4.4 thrust::negation_value . . . 542
2.10.4.5 thrust::remove_cvref . . . 542
2.10.4.6 thrust::voider . . . 542
2.10.4.7 thrust::integer_sequence . . . 543

飞腾信息技术有限公司 FeiTeng Information Technology Co., Inc. 2024-11-18 15:00:00 Confidential

2.10.4.8 thrust::index_sequence . . . 543
2.10.4.9 thrust::make_integer_sequence . . . 543
2.10.4.10thrust::make_index_sequence . . . 544
2.10.4.11thrust::make_reversed_integer_sequence . . . 544
2.10.4.12thrust::make_reversed_index_sequence . . . 544
2.10.4.13thrust::integer_sequence_push_front . . . 545
2.10.4.14thrust::integer_sequence_push_back . . . 545
2.10.4.15thrust::is_contiguous_iterator . . . 545
2.10.4.16thrust::is_execution_policy . . . 546
2.10.4.17thrust::is_operator_less_function_object . . . 546
2.10.4.18thrust::is_operator_greater_function_object . . . 546
2.10.4.19thrust::is_operator_less_or_greater_function_object . . . 546
2.10.4.20thrust::is_operator_plus_function_object . . . 547
2.10.4.21thrust::conjunction . . . 547
2.10.4.22thrust::disjunction . . . 547
2.10.4.23thrust::negation . . . 548
2.10.4.24thrust::remove_cvref_t . . . 548
2.10.4.25thrust::is_contiguous_iterator_v . . . 548
2.10.4.26thrust::is_execution_policy_v . . . 548
2.10.4.27thrust::is_operator_less_function_object_v . . . 549
2.10.4.28thrust::is_operator_greater_function_object_v . . . 549
2.10.4.29thrust::is_operator_less_or_greater_function_object_v . . . 549
2.10.4.30thrust::is_operator_plus_function_object_v . . . 549
2.10.4.31thrust::conjunction_v . . . 550
2.10.4.32thrust::disjunction_v . . . 550
2.10.4.33thrust::negation_v . . . 550
2.10.4.34thrust::conjunction_value_v . . . 550
2.10.4.35thrust::disjunction_value_v . . . 551
2.10.4.36thrust::negation_value_v . . . 551

飞腾信息 Mega Confidential 2024-11-18 15:00:00

1 简介

Thrust 是基于标准模板库（Standard Template Library，STL）的 MXMACA® C++ 模板库。Thrust 支持通过与 MXMACA C 完全互通的高级接口，以最少的编程工作量实现高性能并行应用程序。

Thrust 提供了丰富的数据并行原语集合，如扫描（scan），排序（sort），归约（reduce）等，这些原语可以组合在一起，以简洁、可读的源代码实现复杂的算法。通过用这些高级抽象描述计算，可以让 Thrust 自由、自动地选择最有效的实现。因此，Thrust 可用于 MXMACA 应用程序的快速原型设计（最重要的是程序员生产力），也可用于生产（稳健性和绝对性能至关重要）。本文档介绍了如何使用 Thrust 开发 MXMACA 应用程序。即使 C++ 或 MXMACA 经验有限，也适用于本教程。

1.1 安装

安装 MXMACA 工具包会将 Thrust 头文件复制到系统标准的 MXMACA include 目录。由于 Thrust 是头文件的模板库，因此无需进一步安装即可使用 Thrust。

```
# header location:
${MACA_PATH}/include/thrust
${MACA_PATH}/include/cub
```

1.2 Hello thrust

下面展示了如何在 CMake 项目中使用 Thrust。

CMakeLists.txt

```
project(example)
cmake_minimum_required(VERSION 3.16)

# maca toolkits path
set(MACA_PATH $ENV{MACA_PATH})

# thrust path in maca toolkits
set(thrust_src_dir ${MACA_PATH}/include)

set(example_target thrust.example)
set(example_src example.cpp)

enable_language(CXX)

add_compile_options( -x maca -std=c++17 -fPIC
                     --maca-device-lib-path=${MACA_PATH}/lib/
                     --maca-device-lib=maca_mathlib.bc
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
                    --maca-device-lib=maca_kernellib.bc
                    --maca-host-lib-path=${MACA_PATH}/lib/
                    --maca-host-lib=maca_mathlib_host.bc
)

add_executable(${example_target} "${example_src}")
target_include_directories(${example_target} PRIVATE ${thrust_src_dir})
```

example.cpp

```
// Example.Application Using C and Thrust: copy
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/copy.h>
#include <thrust/fill.h>
#include <thrust/sequence.h>
#include <iostream>
int main(void)
{
   // initialize all ten integers of a device_vector to 1
   thrust::device_vector<int> D(10, 1);

   // set the first seven elements of a vector to 9
   thrust::fill(D.begin(), D.begin() + 7, 9);

   // initialize a host_vector with the first five elements of D
   thrust::host_vector<int> H(D.begin(), D.begin() + 5);

   // set the elements of H to 0, 1, 2, 3, ...
   thrust::sequence(H.begin(), H.end());

   // copy all of H back to the beginning of D
   thrust::copy(H.begin(), H.end(), D.begin());

   // print D
   for(size_t i = 0; i < D.size(); i++)
     std::cout << "D[" << i << "] = " << D[i] << std::endl;

   return 0;
}
```

将 example.cpp 和 CMakeLists.txt 存储在一起，运行 build 命令：

```
export ${MACA_PATH}=your/maca/toolkits/path
export CXX=${MACA_PATH}/mxgpu_llvm/bin/mxcc

mkdir build
cmake -B build .
cmake --build build -j
```

thrust.example 二进制文件将出现在 **build** 目录中。

飞腾信息 MetaX Confidential 2021-11-18 15:00:00

2 使用 Thrust API

2.1 算法

2.1.1 复制（Copying）

2.1.1.1 复制

函数

```
template <typename DerivedPolicy,
  typename InputIterator,
  typename OutputIterator>
__host__ __device__ OutputIterator
thrust::copy(const thrust::detail::execution_policy_base< DerivedPolicy > &
 ↪ exec,
  InputIterator first,
  InputIterator last,
  OutputIterator result);

template <typename DerivedPolicy,
  typename InputIterator,
  typename Size,
  typename OutputIterator>
__host__ __device__ OutputIterator
thrust::copy_n(const thrust::detail::execution_policy_base< DerivedPolicy >
 ↪ & exec,
  InputIterator first,
  Size n,
  OutputIterator result);

template <typename InputIterator,
  typename OutputIterator>
OutputIterator
thrust::copy(InputIterator first,
  InputIterator last,
  OutputIterator result);

template <typename InputIterator,
  typename Size,
  typename OutputIterator>
OutputIterator
thrust::copy_n(InputIterator first,
  Size n,
  OutputIterator result);
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
template <typename DerivedPolicy,
  typename ForwardIterator1,
  typename ForwardIterator2>
__host__ __device__ ForwardIterator2
thrust::swap_ranges(const thrust::detail::execution_policy_base<
 ↪DerivedPolicy > & exec,
  ForwardIterator1 first1,
  ForwardIterator1 last1,
  ForwardIterator2 first2);

template <typename ForwardIterator1,
  typename ForwardIterator2>
ForwardIterator2
thrust::swap_ranges(ForwardIterator1 first1,
  ForwardIterator1 last1,
  ForwardIterator2 first2);

template <typename DerivedPolicy,
  typename InputIterator,
  typename ForwardIterator>
__host__ __device__ ForwardIterator
thrust::uninitialized_copy(const thrust::detail::execution_policy_base<
 ↪DerivedPolicy > & exec,
  InputIterator first,
  InputIterator last,
  ForwardIterator result);

template <typename InputIterator,
  typename ForwardIterator>
ForwardIterator
thrust::uninitialized_copy(InputIterator first,
  InputIterator last,
  ForwardIterator result);

template <typename DerivedPolicy,
  typename InputIterator,
  typename Size,
  typename ForwardIterator>
__host__ __device__ ForwardIterator
thrust::uninitialized_copy_n(const thrust::detail::execution_policy_base<
 ↪DerivedPolicy > & exec,
  InputIterator first,
  Size n,
  ForwardIterator result);

template <typename InputIterator,
  typename Size,
  typename ForwardIterator>
ForwardIterator
thrust::uninitialized_copy_n(InputIterator first,
  Size n,
  ForwardIterator result);
```

飞曦信息 MetaX Confidential 2024-11-18 15:00:00

thrust::copy

```
template <typename DerivedPolicy,
  typename InputIterator,
  typename OutputIterator>
__host__ __device__ OutputIterator
copy(const thrust::detail::execution_policy_base< DerivedPolicy > & exec,
  InputIterator first,
  InputIterator last,
  OutputIterator result);
```

copy 将元素从范围 [first, last) 复制到范围 [result, result + (last - first))。即，执行赋值 *result = *first, *(result + 1) = *(first + 1)，依此类推。总而言之，copy 对从 0 到 last - first 的每一个整数 n 执行赋值 *(result + n) = *(first + n)。与 std::copy 不同，copy 不保证操作顺序。因此，在源范围和目标范围重叠的情况下调用 copy 具有未定义的行为。

返回值是 result + (last - first)。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::device 并行化策略，使用 copy 将元素从一个范围复制到另一个范围：

```
#include <thrust/copy.h>
#include <thrust/device_vector.h>
#include <thrust/execution_policy.h>
...

thrust::device_vector<int> vec0(100);
thrust::device_vector<int> vec1(100);
...

thrust::copy(thrust::device, vec0.begin(), vec0.end(), vec1.begin());

// vec1 is now a copy of vec0
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **InputIterator**：必须是 Input Iterator 的模型，InputIterator 的 value_type 必须可转换为 OutputIterator 的 value_type
- **OutputIterator**：必须是 Output Iterator 的模型

函数参数

- **exec**：用于并行化的执行策略
- **first**：要复制序列的起始
- **last**：要复制序列的末尾
- **result**：目标序列

前提条件

result 可以等于 first ，但除此外，result 不应在范围 [first, last) 内

返回值

目标序列末尾

另请参阅

https://en.cppreference.com/w/cpp/algorithm/copy

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

thrust::copy_n

```
template <typename DerivedPolicy,
  typename InputIterator,
  typename Size,
  typename OutputIterator>
__host__ __device__ OutputIterator
copy_n(const thrust::detail::execution_policy_base< DerivedPolicy > & exec,
  InputIterator first,
  Size n,
  OutputIterator result);
```

copy_n 将元素从范围 [first, first + n) 复制到范围 [result, result + n)。即，执行赋值 *result = *first, *(result + 1) = *(first + 1)，依此类推。总而言之，copy 对从 0 到 n 的每一个整数 i 执行赋值 *(result + i) = *(first + i)。与 std::copy_n 不同，copy_n 不保证操作顺序。因此，在源范围和目标范围重叠的情况下调用 copy_n 具有未定义的行为。

返回值是 result + n。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::device 并行化策略，使用 copy 将元素从一个范围复制到另一个范围：

```
#include <thrust/copy.h>
#include <thrust/device_vector.h>
#include <thrust/execution_policy.h>
...
size_t n = 100;
thrust::device_vector<int> vec0(n);
thrust::device_vector<int> vec1(n);
...
thrust::copy_n(thrust::device, vec0.begin(), n, vec1.begin());

// vec1 is now a copy of vec0
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **InputIterator**：必须是 Input Iterator 的模型，InputIterator 的 value_type 必须可转换为 OutputIterator 的 value_type
- **Size**：整型类型
- **OutputIterator**：必须是 Output Iterator 的模型

函数参数

- **exec**：用于并行化的执行策略
- **first**：要复制范围的起始
- **n**：要复制的元素的数量
- **result**：目标范围起始

前提条件

result 可以等于 first，但除此外，result 不应在范围 [first, first + n) 内

返回值

目标范围末尾

另请参阅

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

- https://en.cppreference.com/w/cpp/algorithm/copy_n
- thrust::copy

thrust::copy

```
template <typename InputIterator,
  typename OutputIterator>
OutputIterator
copy(InputIterator first,
  InputIterator last,
  OutputIterator result);
```

copy 将元素从范围 [first, last) 复制到范围 [result, result + (last - first))。即，执行赋值 *result = *first, *(result + 1) = *(first + 1)，依此类推。总而言之，copy 对从 0 到 last - first 的每一个整数 n 执行赋值 *(result + n) = *(first + n)。与 std::copy 不同，copy 不保证操作顺序。因此，在源范围和目标范围重叠的情况下调用 copy 具有未定义的行为。

返回值是 result + (last - first)。

以下代码片段演示了如何使用 copy 从一个范围复制到另一个范围：

```
#include <thrust/copy.h>
#include <thrust/device_vector.h>
...

thrust::device_vector<int> vec0(100);
thrust::device_vector<int> vec1(100);
...

thrust::copy(vec0.begin(), vec0.end(),
vec1.begin());

// vec1 is now a copy of vec0
```

模板参数

- **InputIterator**：必须是 Input Iterator 的模型，InputIterator 的 value_type 必须可转换为 OutputIterator 的 value_type
- **OutputIterator**：必须是 Output Iterator 的模型

函数参数

- **first**：要复制序列的起始
- **last**：要复制序列的末尾
- **result**：目标序列

前提条件

result 可以等于 first ，但除此外，result 不应在范围 [first, last) 内

返回值

目标序列末尾

另请参阅

https://en.cppreference.com/w/cpp/algorithm/copy

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

thrust::copy_n

```
template <typename InputIterator,
  typename Size,
  typename OutputIterator>
OutputIterator
copy_n(InputIterator first,
  Size n,
  OutputIterator result);
```

copy_n 将元素从范围 [first, first + n) 复制到范围 [result, result + n)。即，执行赋值 *result = *first, *(result + 1) = *(first + 1)，依此类推。总而言之，copy 对从 0 到 n 的每一个整数 i 执行赋值 *(result + i) = *(first + i)。与 std::copy_n 不同，copy_n 不保证操作顺序。因此，在源范围和目标范围重叠的情况下调用 copy_n 具有未定义的行为。

返回值是 result + n。

以下代码片段演示了如何使用 copy 从一个范围复制到另一个范围：

```
#include <thrust/copy.h>
#include <thrust/device_vector.h>
...
size_t n = 100;
thrust::device_vector<int> vec0(n);
thrust::device_vector<int> vec1(n);
...
thrust::copy_n(vec0.begin(), n, vec1.begin());

// vec1 is now a copy of vec0
```

模板参数

- **InputIterator**：必须是 Input Iterator 的模型，InputIterator 的 value_type 必须可转换为 OutputIterator 的 value_type
- **Size**：整型类型
- **OutputIterator**：必须是 Output Iterator 的模型

函数参数

- **first**：要复制范围的起始
- **n**：要复制的元素的数量
- **result**：目标范围起始

前提条件

result 可以等于 first，但除此外，result 不应在范围 [first, first + n) 内

返回值

目标范围末尾

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/copy_n
- thrust::copy

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

thrust::swap_ranges

```
template <typename DerivedPolicy,
  typename ForwardIterator1,
  typename ForwardIterator2>
__host__ __device__ ForwardIterator2
swap_ranges(const thrust::detail::execution_policy_base< DerivedPolicy > &
↪exec,
  ForwardIterator1 first1,
  ForwardIterator1 last1,
  ForwardIterator2 first2);
```

swap_ranges 交换范围 [first1, last1) 和范围 [first2, first2 + (last1 - first1)) 内的每个元素。即，对于每个满足 0 <= n < (last1 - first1) 的整数 n ，它交换 *(first1 + n) 和 *(first2 + n) 的元素。

返回值是 first2 + (last1 - first1)。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::device 并行化执行策略，使用 swap_ranges 交换两个 thrust::device_vectors 的内容：

```
#include <thrust/swap.h>
#include <thrust/device_vector.h>
#include <thrust/execution_policy.h>
...
thrust::device_vector<int> v1(2), v2(2);
v1[0] = 1;
v1[1] = 2;
v2[0] = 3;
v2[1] = 4;

thrust::swap_ranges(thrust::device, v1.begin(), v1.end(), v2.begin());

// v1[0] == 3, v1[1] == 4, v2[0] == 1, v2[1] == 2
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **ForwardIterator1**：是 Forward Iterator 的模型，ForwardIterator1 的 value_type 必须可转换为 ForwardIterator2 的 value_type
- **ForwardIterator2**：是 Forward Iterator 的模型，ForwardIterator2 的 value_type 必须可转换为 ForwardIterator1 的 value_type

函数参数

- **exec**：用于并行化的执行策略
- **first1**：要交换的第一个序列的起始
- **last1**：要交换的第一个序列中最后一个元素的下一个位置
- **first2**：要交换的第二个序列的起始

前提条件

first1 可以等于 first2 ，但除此外，范围 [first1, last1) 不应与范围 [first2, first2 + (last1 - first1)) 重叠

返回值

指向第二个序列中需要交换的最后一个元素的下一个位置的迭代器

另请参阅

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

- https://en.cppreference.com/w/cpp/algorithm/swap_ranges
- Swap

thrust::swap_ranges

```
template <typename ForwardIterator1,
  typename ForwardIterator2>
ForwardIterator2
swap_ranges(ForwardIterator1 first1,
  ForwardIterator1 last1,
  ForwardIterator2 first2);
```

swap_ranges 交换范围 [first1, last1) 和范围 [first2, first2 + (last1 - first1)) 内的每个元素。即，对于每个满足 0 <= n < (last1 - first1) 的整数 n ，它交换 *(first1 + n) 和 *(first2 + n) 的元素。

返回值是 first2 + (last1 - first1)。

以下代码片段演示了如何使用 swap_ranges 交换两个 thrust::device_vectors 的内容：

```
#include <thrust/swap.h>
#include <thrust/device_vector.h>
...
thrust::device_vector<int> v1(2), v2(2);
v1[0] = 1;
v1[1] = 2;
v2[0] = 3;
v2[1] = 4;

thrust::swap_ranges(v1.begin(), v1.end(), v2.begin());

// v1[0] == 3, v1[1] == 4, v2[0] == 1, v2[1] == 2
```

模板参数

- **ForwardIterator1**：是 Forward Iterator 的模型，ForwardIterator1 的 value_type 必须可转换为 ForwardIterator2 的 value_type
- **ForwardIterator2**：是 Forward Iterator 的模型，ForwardIterator2 的 value_type 必须可转换为 ForwardIterator1 的 value_type

函数参数

- **first1**：要交换的第一个序列的起始
- **last1**：要交换的第一个序列中最后一个元素的下一个位置
- **first2**：要交换的第二个序列的起始

前提条件

first1 可以等于 first2 ，但除此外，范围 [first1, last1) 不应与范围 [first2, first2 + (last1 - first1)) 重叠

返回值

指向要交换的第二个序列中最后一个元素的下一个位置的迭代器

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/swap_ranges
- Swap

飞腾信息 MetaX Confidential 2025-11-18 15:00:00

thrust::uninitialized_copy

```
template <typename DerivedPolicy,
  typename InputIterator,
  typename ForwardIterator>
__host__ __device__ ForwardIterator
uninitialized_copy(const thrust::detail::execution_policy_base<
↪DerivedPolicy > & exec,
  InputIterator first,
  InputIterator last,
  ForwardIterator result);
```

在 thrust 中，函数 thrust::device_new 为对象分配内存，然后通过调用构造函数在该位置创建对象。然而，偶尔将这两个操作分开非常有用。如果范围 [result, result + (last - first)) 内的每个迭代器都指向未初始化的内存，则 uninitialized_copy 在该范围内创建 [first, last) 的副本。即，对于输入中的每个迭代器 i，uninitialized_copy 在输出范围内相应迭代器指向的位置，通过 ForwardIterator 的 value_type 的拷贝构造函数以 *i 为其参数创建 *i 的副本。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::device 并行化执行策略，使用 uninitialized_copy 初始化未初始化的内存范围：

```
#include <thrust/uninitialized_copy.h>
#include <thrust/device_malloc.h>
#include <thrust/device_vector.h>
#include <thrust/execution_policy.h>

struct Int
{
__host__ __device__
Int(int x) : val(x) {}
int val;
};
...
const int N = 137;

Int val(46);
thrust::device_vector<Int> input(N, val);
thrust::device_ptr<Int> array = thrust::device_malloc<Int>(N);
thrust::uninitialized_copy(thrust::device, input.begin(), input.end(),
↪array);

// Int x = array[i];
// x.val == 46 for all 0 <= i < N
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **InputIterator**：是 Input Iterator 的模型
- **ForwardIterator**：是 Forward Iterator 的模型，ForwardIterator 是可变的，并且 ForwardIterator 的 value_type 有一个采用 InputIterator 的 value_type 类型单参数的构造函数

函数参数

- **exec**：用于并行化的执行策略
- **first**：要从中复制的输入范围的第一个元素
- **last**：要从中复制的输入范围的最后一个元素

飞曦信息 MetaX Confidential 2024-11-18 15:00:00

- **result**：要复制到的输出范围的第一个元素

前提条件

first 可以等于 result ，但除此外，范围 [first, last) 不应与范围 [result, result + (last - first)) 重叠

返回值

指向输出范围最后一个元素的迭代器

另请参阅

- https://en.cppreference.com/w/cpp/memory/uninitialized_copy
- copy
- uninitialized_fill
- device_new
- device_malloc

thrust::uninitialized_copy

```
template <typename InputIterator,
  typename ForwardIterator>
ForwardIterator
uninitialized_copy(InputIterator first,
  InputIterator last,
  ForwardIterator result);
```

在 thrust 中，函数 thrust::device_new 为对象分配内存，然后通过调用构造函数在该位置创建对象。然而，偶尔将这两个操作分开非常有用。如果范围 [result, result + (last - first)) 内的每个迭代器都指向未初始化的内存，则 uninitialized_copy 在该范围内创建 [first, last) 的副本。即，对于输入中的每个迭代器 i ，uninitialized_copy 在输出范围内相应迭代器指向的位置，通过 ForwardIterator 的 value_type 的 拷贝构造函数以 *i 为其参数创建 *i 的副本。

以下代码片段演示了如何使用 uninitialized_copy 初始化未初始化的内存范围：

```
#include <thrust/uninitialized_copy.h>
#include <thrust/device_malloc.h>
#include <thrust/device_vector.h>

struct Int
{
__host__ __device__
Int(int x) : val(x) {}
int val;
};
...
const int N = 137;

Int val(46);
thrust::device_vector<Int> input(N, val);
thrust::device_ptr<Int> array = thrust::device_malloc<Int>(N);
thrust::uninitialized_copy(input.begin(), input.end(), array);

// Int x = array[i];
// x.val == 46 for all 0 <= i < N
```

模板参数

- **InputIterator**：是 Input Iterator 的模型

飞腾信息 MetaX Confidential 2024-11-18 17:00:00

- **ForwardIterator**：是 Forward Iterator 的模型，ForwardIterator 是可变的，并且 ForwardIterator 的 value_type 有一个采用 InputIterator 的 value_type 类型单参数的构造函数

函数参数

- **first**：要从中复制的输入范围的第一个元素
- **last**：要从中复制的输入范围的最后一个元素
- **result**：要复制到的输出范围的第一个元素

前提条件

first 可以等于 result ，但除此外，范围 [first, last) 不应与范围 [result, result + (last - first)) 重叠

返回值

指向输出范围最后一个元素的迭代器

另请参阅

- https://en.cppreference.com/w/cpp/memory/uninitialized_copy
- copy
- uninitialized_fill
- device_new
- device_malloc

thrust::uninitialized_copy_n

```
template <typename DerivedPolicy,
  typename InputIterator,
  typename Size,
  typename ForwardIterator>
__host__ __device__ ForwardIterator
uninitialized_copy_n(const thrust::detail::execution_policy_base<
↪DerivedPolicy > & exec,
  InputIterator first,
  Size n,
  ForwardIterator result);
```

在 thrust 中，函数 thrust::device_new 为对象分配内存，然后通过调用构造函数在该位置创建对象。然而，偶尔将这两个操作分开非常有用。如果范围 [result, result + n) 内的每个迭代器都指向未初始化的内存，则 uninitialized_copy_n 在该范围内创建 [first, first + n) 的副本。即，对于输入中的每个迭代器 i ，uninitialized_copy_n 在输出范围内相应迭代器指向的位置，通过 InputIterator 的 value_type 的 拷贝构造函数以 *i 为其参数创建 *i 的副本。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::device 并行化执行策略，使用 uninitialized_copy 初始化未初始化的内存范围：

```
#include <thrust/uninitialized_copy.h>
#include <thrust/device_malloc.h>
#include <thrust/device_vector.h>
#include <thrust/execution_policy.h>

struct Int
{
__host__ __device__
Int(int x) : val(x) {}
```

(下页继续)

飞腾信息 MetaX Confidential 2021-11-18 15:00:00

(续上页)

```
int val;
};
...
const int N = 137;

Int val(46);
thrust::device_vector<Int> input(N, val);
thrust::device_ptr<Int> array = thrust::device_malloc<Int>(N);
thrust::uninitialized_copy_n(thrust::device, input.begin(), N, array);

// Int x = array[i];
// x.val == 46 for all 0 <= i < N
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **InputIterator**：是 Input Iterator 的模型
- **Size**：整型类型
- **ForwardIterator**：是 Forward Iterator 的模型，ForwardIterator 是可变的，并且 ForwardIterator 的 value_type 有一个采用 InputIterator 的 value_type 类型单参数的构造函数

函数参数

- **exec**：用于并行化的执行策略
- **first**：要从中复制的输入范围的第一个元素
- **n**：要复制的元素的数量
- **result**：要复制到的输出范围的第一个元素

前提条件

first 可以等于 result ，但除此外，范围 [first, first + n) 不应与范围 [result, result + n) 重叠

返回值

指向输出范围最后一个元素的迭代器

另请参阅

- https://en.cppreference.com/w/cpp/memory/uninitialized_copy
- uninitialized_copy
- copy
- uninitialized_fill
- device_new
- device_malloc

thrust::uninitialized_copy_n

```
template <typename InputIterator,
  typename Size,
  typename ForwardIterator>
ForwardIterator
uninitialized_copy_n(InputIterator first,
  Size n,
  ForwardIterator result);
```

飞腾信息 Metax Confidential 2024-11-18 15:00:00

在 thrust 中，函数 thrust::device_new 为对象分配内存，然后通过调用构造函数在该位置创建对象。然而，偶尔将这两个操作分开非常有用。如果范围 [result, result + n) 内的每个迭代器都指向未初始化的内存，则 uninitialized_copy_n 在该范围内创建 [first, first + n) 的副本。即，对于输入中的每个迭代器 i，uninitialized_copy_n 在输出范围内相应迭代器指向的位置，通过 InputIterator 的 value_type 的 拷贝构造函数以 *i 为其参数创建 *i 的副本。

以下代码片段演示了如何使用 uninitialized_copy 初始化未初始化的内存范围：

```
#include <thrust/uninitialized_copy.h>
#include <thrust/device_malloc.h>
#include <thrust/device_vector.h>

struct Int
{
__host__ __device__
Int(int x) : val(x) {}
int val;
};
...
const int N = 137;

Int val(46);
thrust::device_vector<Int> input(N, val);
thrust::device_ptr<Int> array = thrust::device_malloc<Int>(N);
thrust::uninitialized_copy_n(input.begin(), N, array);

// Int x = array[i];
// x.val == 46 for all 0 <= i < N
```

模板参数

- **InputIterator**：是 Input Iterator 的模型
- **Size**：整型类型
- **ForwardIterator**：是 Forward Iterator 的模型，ForwardIterator 是可变的，并且 ForwardIterator 的 value_type 有一个采用 InputIterator 的 value_type 类型单参数的构造函数

函数参数

- **first**：要从中复制的输入范围的第一个元素
- **n**：要复制的元素的数量
- **result**：要复制到的输出范围的第一个元素

前提条件

first 可以等于 result，但除此外，范围 [first, first + n) 不应与范围 [result, result + n) 重叠

返回值

指向输出范围最后一个元素的迭代器

另请参阅

- https://en.cppreference.com/w/cpp/memory/uninitialized_copy
- uninitialized_copy
- copy
- uninitialized_fill
- device_new
- device_malloc

飞腾信息 MetaX Confidential 2024-11-19 15:00:00

2.1.1.2 聚集（Gathering）

函数

```
template <typename DerivedPolicy,
  typename InputIterator,
  typename RandomAccessIterator,
  typename OutputIterator>
__host__ __device__ OutputIterator
thrust::gather(const thrust::detail::execution_policy_base< DerivedPolicy >
↪ & exec,
  InputIterator map_first,
  InputIterator map_last,
  RandomAccessIterator input_first,
  OutputIterator result);

template <typename InputIterator,
  typename RandomAccessIterator,
  typename OutputIterator> OutputIterator
thrust::gather(InputIterator map_first,
  InputIterator map_last,
  RandomAccessIterator input_first,
  OutputIterator result);

template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2,
  typename RandomAccessIterator,
  typename OutputIterator>
__host__ __device__ OutputIterator
thrust::gather_if(const thrust::detail::execution_policy_base<
↪DerivedPolicy > & exec,
  InputIterator1 map_first,
  InputIterator1 map_last,
  InputIterator2 stencil,
  RandomAccessIterator input_first,
  OutputIterator result);

template <typename InputIterator1,
  typename InputIterator2,
  typename RandomAccessIterator,
  typename OutputIterator> OutputIterator
thrust::gather_if(InputIterator1 map_first,
  InputIterator1 map_last,
  InputIterator2 stencil,
  RandomAccessIterator input_first,
  OutputIterator result);

template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2,
  typename RandomAccessIterator,
  typename OutputIterator,
  typename Predicate>
__host__ __device__ OutputIterator
thrust::gather_if(const thrust::detail::execution_policy_base<
↪DerivedPolicy > & exec,
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
  InputIterator1 map_first,
  InputIterator1 map_last,
  InputIterator2 stencil,
  RandomAccessIterator input_first,
  OutputIterator result,
  Predicate pred);

template <typename InputIterator1,
  typename InputIterator2,
  typename RandomAccessIterator,
  typename OutputIterator,
  typename Predicate> OutputIterator
thrust::gather_if(InputIterator1 map_first,
  InputIterator1 map_last,
  InputIterator2 stencil,
  RandomAccessIterator input_first,
  OutputIterator result,
  Predicate pred);
```

thrust::gather

```
template <typename DerivedPolicy,
  typename InputIterator,
  typename RandomAccessIterator,
  typename OutputIterator>
__host__ __device__ OutputIterator
gather(const thrust::detail::execution_policy_base< DerivedPolicy > & exec,
  InputIterator map_first,
  InputIterator map_last,
  RandomAccessIterator input_first,
  OutputIterator result);
```

gather 根据映射将元素从源数组复制到目标范围。对于范围 [map_first, map_last) 内的每个迭代器 i，将值 input_first[*i] 赋给 *(result + (i - map_first))。RandomAccessIterator 必须允许随机访问。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::device 并行化执行策略，使用 gather 对一个范围内的元素进行重新排序：

```
#include <thrust/gather.h>
#include <thrust/device_vector.h>
#include <thrust/execution_policy.h>
...
// mark even indices with a 1; odd indices with a 0
int values[10] = {1, 0, 1, 0, 1, 0, 1, 0, 1, 0};
thrust::device_vector<int> d_values(values, values + 10);

// gather all even indices into the first half of the range
// and odd indices to the last half of the range
int map[10]    = {0, 2, 4, 6, 8, 1, 3, 5, 7, 9};
thrust::device_vector<int> d_map(map, map + 10);

thrust::device_vector<int> d_output(10);
thrust::gather(thrust::device,
```

(下页继续)

飞诺信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
d_map.begin(), d_map.end(),
d_values.begin(),
d_output.begin());
// d_output is now {1, 1, 1, 1, 1, 0, 0, 0, 0, 0}
```

备注

gather 是 thrust::scatter 的逆过程

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **InputIterator**：必须是 Input Iterator 的模型，InputIterator 的 value_type 必须可转换为 RandomAccessIterator 的 difference_type
- **RandomAccessIterator**：必须是 Random Access Iterator 的模型，RandomAccessIterator 的 value_type 必须可转换为 OutputIterator 的 value_type
- **OutputIterator**：必须是 Output Iterator 的模型

函数参数

- **exec**：用于并行化的执行策略
- **map_first**：聚集位置的范围起始
- **map_last**：聚集位置的范围末尾
- **input_first**：源范围起始
- **result**：目标范围起始

前提条件

- 范围 [map_first, map_last) 不应与范围 [result, result + (map_last - map_first)) 重叠
- 输入数据不应与范围 [result, result + (map_last - map_first)) 重叠

thrust::gather

```
template <typename InputIterator,
  typename RandomAccessIterator,
  typename OutputIterator> OutputIterator
gather(InputIterator map_first,
  InputIterator map_last,
  RandomAccessIterator input_first,
  OutputIterator result);
```

gather 根据映射将元素从源数组复制到目标范围。对于范围 [map_first, map_last) 内的每个迭代器 i，将值 input_first[*i] 赋给 *(result + (i - map_first))。RandomAccessIterator 必须允许随机访问。

以下代码片段演示了如何使用 gather 对一个范围内的元素进行重新排序：

```
#include <thrust/gather.h>
#include <thrust/device_vector.h>
...
// mark even indices with a 1; odd indices with a 0
int values[10] = {1, 0, 1, 0, 1, 0, 1, 0, 1, 0};
thrust::device_vector<int> d_values(values, values + 10);

// gather all even indices into the first half of the range
// and odd indices to the last half of the range
```

(下页继续)

飞曦信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
int map[10]   = {0, 2, 4, 6, 8, 1, 3, 5, 7, 9};
thrust::device_vector<int> d_map(map, map + 10);

thrust::device_vector<int> d_output(10);
thrust::gather(d_map.begin(), d_map.end(),
d_values.begin(),
d_output.begin());
// d_output is now {1, 1, 1, 1, 1, 0, 0, 0, 0, 0}
```

备注

gather 是 thrust::scatter 的逆过程

模板参数

- **InputIterator**：必须是 Input Iterator 的模型，InputIterator 的 value_type 必须可转换为 RandomAccessIterator 的 difference_type
- **RandomAccessIterator**：必须是 Random Access Iterator 的模型，RandomAccessIterator 的 value_type 必须可转换为 OutputIterator 的 value_type
- **OutputIterator**：必须是 Output Iterator 的模型

函数参数

- **map_first**：聚集位置的范围起始
- **map_last**：聚集位置的范围末尾
- **input_first**：源范围起始
- **result**：目标范围起始

前提条件

- 范围 [map_first, map_last) 不应与范围 [result, result + (map_last - map_first)) 重叠
- 输入数据不应与范围 [result, result + (map_last - map_first)) 重叠

thrust::gather_if

```
template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2,
  typename RandomAccessIterator,
  typename OutputIterator>
__host__ __device__ OutputIterator
gather_if(const thrust::detail::execution_policy_base< DerivedPolicy > &
↪exec,
  InputIterator1 map_first,
  InputIterator1 map_last,
  InputIterator2 stencil,
  RandomAccessIterator input_first,
  OutputIterator result);
```

gather_if 根据映射有条件地将元素从源数组复制到目标范围。对于范围 [map_first, map_last) 内使得 *(stencil + (i - map_first)) 的值为 true 的每个迭代器 i ，将值 input_first[*i] 赋给 *(result + (i - map_first))。RandomAccessIterator 必须允许随机访问。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::device 并行化执行策略，使用 gather_if 从输入范围内聚集已选择的值：

飞腾信息 MetaX Confidential 2021-11-18 15:00:00

```
#include <thrust/gather.h>
#include <thrust/device_vector.h>
#include <thrust/execution_policy.h>
...

int values[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
thrust::device_vector<int> d_values(values, values + 10);

// select elements at even-indexed locations
int stencil[10] = {1, 0, 1, 0, 1, 0, 1, 0, 1, 0};
thrust::device_vector<int> d_stencil(stencil, stencil + 10);

// map all even indices into the first half of the range
// and odd indices to the last half of the range
int map[10]   = {0, 2, 4, 6, 8, 1, 3, 5, 7, 9};
thrust::device_vector<int> d_map(map, map + 10);

thrust::device_vector<int> d_output(10, 7);
thrust::gather_if(thrust::device,
d_map.begin(), d_map.end(),
d_stencil.begin(),
d_values.begin(),
d_output.begin());
// d_output is now {0, 7, 4, 7, 8, 7, 3, 7, 7, 7}
```

备注

gather_if 是 scatter_if 的逆过程

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **InputIterator1**：必须是 Input Iterator 的模型，InputIterator1 的 value_type 必须可转换为 RandomAccessIterator 的 difference_type
- **InputIterator2**：必须是 Input Iterator 的模型，InputIterator2 的 value_type 必须可转换为 bool
- **RandomAccessIterator**：必须是 Random Access Iterator 的模型，RandomAccessIterator 的 value_type 必须可转换为 OutputIterator 的 value_type
- **OutputIterator**：必须是 Output Iterator 的模型

函数参数

- **exec**：用于并行化的执行策略
- **map_first**：聚集位置的范围起始
- **map_last**：聚集位置的范围末尾
- **stencil**：谓词（predicate）值范围的起始
- **input_first**：源范围起始
- **result**：目标范围起始

前提条件

- 范围 [map_first, map_last) 不应与范围 [result, result + (map_last - map_first)) 重叠
- 范围 [stencil, stencil + (map_last - map_first)) 不应与范围 [result, result + (map_last - map_first)) 重叠
- 输入数据不应与范围 [result, result + (map_last - map_first)) 重叠

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

thrust::gather_if

```
template <typename InputIterator1,
  typename InputIterator2,
  typename RandomAccessIterator,
  typename OutputIterator> OutputIterator
gather_if(InputIterator1 map_first,
  InputIterator1 map_last,
  InputIterator2 stencil,
  RandomAccessIterator input_first,
  OutputIterator result);
```

gather_if 根据映射有条件地将元素从源数组复制到目标范围。对于范围 [map_first, map_last) 内使得 *(stencil + (i - map_first)) 的值为 true 的每个迭代器 i，将值 input_first[*i] 赋给 *(result + (i - map_first))。RandomAccessIterator 必须允许随机访问。

以下代码片段演示了如何使用 gather_if 从输入范围内聚集已选择的值：

```
#include <thrust/gather.h>
#include <thrust/device_vector.h>
...

int values[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
thrust::device_vector<int> d_values(values, values + 10);

// select elements at even-indexed locations
int stencil[10] = {1, 0, 1, 0, 1, 0, 1, 0, 1, 0};
thrust::device_vector<int> d_stencil(stencil, stencil + 10);

// map all even indices into the first half of the range
// and odd indices to the last half of the range
int map[10]   = {0, 2, 4, 6, 8, 1, 3, 5, 7, 9};
thrust::device_vector<int> d_map(map, map + 10);

thrust::device_vector<int> d_output(10, 7);
thrust::gather_if(d_map.begin(), d_map.end(),
d_stencil.begin(),
d_values.begin(),
d_output.begin());
// d_output is now {0, 7, 4, 7, 8, 7, 3, 7, 7, 7}
```

备注

gather_if 是 scatter_if 的逆过程

模板参数

- **InputIterator1**：必须是 Input Iterator 的模型，InputIterator1 的 value_type 必须可转换为 RandomAccessIterator 的 difference_type
- **InputIterator2**：必须是 Input Iterator 的模型，InputIterator2 的 value_type 必须可转换为 bool
- **RandomAccessIterator**：必须是 Random Access Iterator 的模型，RandomAccessIterator 的 value_type 必须可转换为 OutputIterator 的 value_type
- **OutputIterator**：必须是 Output Iterator 的模型

函数参数

- **map_first**：聚集位置的范围起始
- **map_last**：聚集位置的范围末尾

沐曦信息 MetaX Confidential 2024-11-18 15:00:00

- **stencil**：谓词值范围的起始
- **input_first**：源范围起始
- **result**：目标范围起始

前提条件

- 范围 [map_first, map_last) 不应与范围 [result, result + (map_last - map_first)) 重叠
- 范围 [stencil, stencil + (map_last - map_first)) 不应与范围 [result, result + (map_last - map_first)) 重叠
- 输入数据不应与范围 [result, result + (map_last - map_first)) 重叠

thrust::gather_if

```
template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2,
  typename RandomAccessIterator,
  typename OutputIterator,
  typename Predicate>
__host__ __device__ OutputIterator
gather_if(const thrust::detail::execution_policy_base< DerivedPolicy > &
 ↪exec,
  InputIterator1 map_first,
  InputIterator1 map_last,
  InputIterator2 stencil,
  RandomAccessIterator input_first,
  OutputIterator result,
  Predicate pred);
```

gather_if 根据映射有条件地将元素从源数组复制到目标范围。对于范围 [map_first, map_last) 内使得 pred(*(stencil + (i - map_first))) 的值为 true 的每个迭代器 i ，将值 input_first[*i] 赋给 *(result + (i - map_first))。RandomAccessIterator 必须允许随机访问。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::device 并行化执行策略，使用 gather_if 基于任意选择函数从输入范围内聚集已选择的值：

```
#include <thrust/gather.h>
#include <thrust/device_vector.h>
#include <thrust/execution_policy.h>

struct is_even
{
__host__ __device__
bool operator()(const int x)
{
   return (x % 2) == 0;
}
};
...

int values[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
thrust::device_vector<int> d_values(values, values + 10);

// we will select an element when our stencil is even
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
int stencil[10] = {0, 3, 4, 1, 4, 1, 2, 7, 8, 9};
thrust::device_vector<int> d_stencil(stencil, stencil + 10);

// map all even indices into the first half of the range
// and odd indices to the last half of the range
int map[10]   = {0, 2, 4, 6, 8, 1, 3, 5, 7, 9};
thrust::device_vector<int> d_map(map, map + 10);

thrust::device_vector<int> d_output(10, 7);
thrust::gather_if(thrust::device,
d_map.begin(), d_map.end(),
d_stencil.begin(),
d_values.begin(),
d_output.begin(),
is_even());
// d_output is now {0, 7, 4, 7, 8, 7, 3, 7, 7, 7}
```

备注

gather_if 是 scatter_if 的逆过程

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **InputIterator1**：必须是 Input Iterator 的模型，InputIterator1 的 value_type 必须可转换为 RandomAccessIterator 的 difference_type
- **InputIterator2**：必须是 Input Iterator 的模型，InputIterator2 的 value_type 必须可转换为 Predicate 的 argument_type
- **RandomAccessIterator**：必须是 Random Access Iterator 的模型，RandomAccessIterator 的 value_type 必须可转换为 OutputIterator 的 value_type
- **OutputIterator**：必须是 Output Iterator 的模型
- **Predicate**：必须是 Predicate 的模型

函数参数

- **exec**：用于并行化的执行策略
- **map_first**：聚集位置的范围起始
- **map_last**：聚集位置的范围末尾
- **stencil**：谓词值范围的起始
- **input_first**：源范围起始
- **result**：目标范围起始
- **pred**：应用于模板值的谓词

前提条件

- 范围 [map_first, map_last) 不应与范围 [result, result + (map_last - map_first)) 重叠
- 范围 [stencil, stencil + (map_last - map_first)) 不应与范围 [result, result + (map_last - map_first)) 重叠
- 输入数据不应与范围 [result, result + (map_last - map_first)) 重叠

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

thrust::gather_if

```
template <typename InputIterator1,
  typename InputIterator2,
  typename RandomAccessIterator,
  typename OutputIterator,
  typename Predicate>
OutputIterator
gather_if(InputIterator1 map_first,
  InputIterator1 map_last,
  InputIterator2 stencil,
  RandomAccessIterator input_first,
  OutputIterator result,
  Predicate pred);
```

gather_if 根据映射有条件地将元素从源数组复制到目标范围。对于范围 [map_first, map_last) 内使得 pred(*(stencil + (i - map_first))) 的值为 true 的每个迭代器 i，将值 input_first[*i] 赋给 *(result + (i - map_first))。RandomAccessIterator 必须允许随机访问。

以下代码片段演示了如何使用 gather_if 基于任意选择函数从输入范围内聚集已选择的值：

```
#include <thrust/gather.h>
#include <thrust/device_vector.h>

struct is_even
{
__host__ __device__
bool operator()(const int x)
{
   return (x % 2) == 0;
}
};
...

int values[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
thrust::device_vector<int> d_values(values, values + 10);

// we will select an element when our stencil is even
int stencil[10] = {0, 3, 4, 1, 4, 1, 2, 7, 8, 9};
thrust::device_vector<int> d_stencil(stencil, stencil + 10);

// map all even indices into the first half of the range
// and odd indices to the last half of the range
int map[10]   = {0, 2, 4, 6, 8, 1, 3, 5, 7, 9};
thrust::device_vector<int> d_map(map, map + 10);

thrust::device_vector<int> d_output(10, 7);
thrust::gather_if(d_map.begin(), d_map.end(),
d_stencil.begin(),
d_values.begin(),
d_output.begin(),
is_even());
// d_output is now {0, 7, 4, 7, 8, 7, 3, 7, 7, 7}
```

备注

gather_if 是 scatter_if 的逆过程

模板参数

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

- **InputIterator1**：必须是 Input Iterator 的模型，InputIterator1 的 value_type 必须可转换为 RandomAccessIterator 的 difference_type
- **InputIterator2**：必须是 Input Iterator 的模型，InputIterator2 的 value_type 必须可转换为 Predicate 的 argument_type
- **RandomAccessIterator**：必须是 Random Access Iterator 的模型，RandomAccessIterator 的 value_type 必须可转换为 OutputIterator 的 value_type
- **OutputIterator**：必须是 Output Iterator 的模型
- **Predicate**：必须是 Predicate 的模型

函数参数

- **map_first**：聚集位置的范围起始
- **map_last**：聚集位置的范围末尾
- **stencil**：谓词值范围的起始
- **input_first**：源范围起始
- **result**：目标范围起始
- **pred**：应用于模板值的谓词

前提条件

- 范围 [map_first, map_last) 不应与范围 [result, result + (map_last - map_first)) 重叠
- 范围 [stencil, stencil + (map_last - map_first)) 不应与范围 [result, result + (map_last - map_first)) 重叠
- 输入数据不应与范围 [result, result + (map_last - map_first)) 重叠

2.1.1.3 分散（Scattering）

函数

```
template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2,
  typename RandomAccessIterator>
__host__ __device__ void
thrust::scatter(const thrust::detail::execution_policy_base< DerivedPolicy
↪> & exec,
  InputIterator1 first,
  InputIterator1 last,
  InputIterator2 map,
  RandomAccessIterator result);

template <typename InputIterator1,
  typename InputIterator2,
  typename RandomAccessIterator> void
thrust::scatter(InputIterator1 first,
  InputIterator1 last,
  InputIterator2 map,
  RandomAccessIterator result);

template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2,
  typename InputIterator3,
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
  typename RandomAccessIterator>
__host__ __device__ void
thrust::scatter_if(const thrust::detail::execution_policy_base<
 ↪DerivedPolicy > & exec,
  InputIterator1 first,
  InputIterator1 last,
  InputIterator2 map,
  InputIterator3 stencil,
  RandomAccessIterator output);

template <typename InputIterator1,
  typename InputIterator2,
  typename InputIterator3,
  typename RandomAccessIterator> void
thrust::scatter_if(InputIterator1 first,
  InputIterator1 last,
  InputIterator2 map,
  InputIterator3 stencil,
  RandomAccessIterator output);

template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2,
  typename InputIterator3,
  typename RandomAccessIterator,
  typename Predicate>
__host__ __device__ void
thrust::scatter_if(const thrust::detail::execution_policy_base<
 ↪DerivedPolicy > & exec,
  InputIterator1 first,
  InputIterator1 last,
  InputIterator2 map,
  InputIterator3 stencil,
  RandomAccessIterator output,
  Predicate pred);

template <typename InputIterator1,
  typename InputIterator2,
  typename InputIterator3,
  typename RandomAccessIterator,
  typename Predicate> void
thrust::scatter_if(InputIterator1 first,
  InputIterator1 last,
  InputIterator2 map,
  InputIterator3 stencil,
  RandomAccessIterator output,
  Predicate pred);
```

thrust::scatter

```
template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2,
  typename RandomAccessIterator>
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
__host__ __device__ void
scatter(const thrust::detail::execution_policy_base< DerivedPolicy > &
↪exec,
  InputIterator1 first,
  InputIterator1 last,
  InputIterator2 map,
  RandomAccessIterator result);
```

scatter 根据映射将元素从源范围复制到输出数组。对于范围 [first,last) 内的每个迭代器 i，将值 *i 赋给 output[*(map + (i - first))]。 输出迭代器必须允许随机访问。如果相同的索引在范围 [map, map + (last - first)) 内多次出现，则结果未定义。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::device 并行化执行策略，使用 scatter 对一个范围内的元素进行重新排序：

```
#include <thrust/scatter.h>
#include <thrust/device_vector.h>
#include <thrust/execution_policy.h>
...
// mark even indices with a 1; odd indices with a 0
int values[10] = {1, 0, 1, 0, 1, 0, 1, 0, 1, 0};
thrust::device_vector<int> d_values(values, values + 10);

// scatter all even indices into the first half of the
// range, and odd indices vice versa
int map[10]   = {0, 5, 1, 6, 2, 7, 3, 8, 4, 9};
thrust::device_vector<int> d_map(map, map + 10);

thrust::device_vector<int> d_output(10);
thrust::scatter(thrust::device,
d_values.begin(), d_values.end(),
d_map.begin(), d_output.begin());
// d_output is now {1, 1, 1, 1, 1, 0, 0, 0, 0, 0}
```

备注

scatter 是 thrust::gather 的逆过程

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **InputIterator1**：必须是 Input Iterator 的模型，InputIterator1 的 value_type 必须可转换为 RandomAccessIterator 的 value_type
- **InputIterator2**：必须是 Input Iterator 的模型，InputIterator2 的 value_type 必须可转换为 RandomAccessIterator 的 difference_type
- **RandomAccessIteratorr**：必须是 Random Access Iterator 的模型

函数参数

- **exec**：用于并行化的执行策略
- **first**：待分散的元素序列起始位置
- **last**：待分散的元素序列末尾位置
- **map**：输出序列的起始索引位置
- **result**：目标序列

前提条件

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

- 对于范围 [map,map + (last - first)) 内的所有迭代器 i ，迭代器 result + i 不应引用范围 [first,last) 内任意迭代器 j 引用的任何元素
- 对于范围 [map,map + (last - first)) 内的所有迭代器 i ，迭代器 result + i 不应引用范围 [map,map + (last - first)) 内任意迭代器 j 引用的任何元素
- 表达式 result[*i] 对范围 [map,map + (last - first)) 内的所有迭代器都有效

thrust::scatter

```
template <typename InputIterator1,
  typename InputIterator2,
  typename RandomAccessIterator> void
scatter(InputIterator1 first,
  InputIterator1 last,
  InputIterator2 map,
  RandomAccessIterator result);
```

scatter 根据映射将元素从源范围复制到输出数组。对于范围 [first,last) 内的每个迭代器 i ，将值 *i 赋给 output[*(map + (i - first))]。 输出迭代器必须允许随机访问。如果相同的索引在范围 [map, map + (last - first)) 内多次出现，则结果未定义。

以下代码片段演示了如何使用 scatter 对范围进行重新排序：

```
#include <thrust/scatter.h>
#include <thrust/device_vector.h>
...
// mark even indices with a 1; odd indices with a 0
int values[10] = {1, 0, 1, 0, 1, 0, 1, 0, 1, 0};
thrust::device_vector<int> d_values(values, values + 10);

// scatter all even indices into the first half of the
// range, and odd indices vice versa
int map[10]   = {0, 5, 1, 6, 2, 7, 3, 8, 4, 9};
thrust::device_vector<int> d_map(map, map + 10);

thrust::device_vector<int> d_output(10);
thrust::scatter(d_values.begin(), d_values.end(),
d_map.begin(), d_output.begin());
// d_output is now {1, 1, 1, 1, 1, 0, 0, 0, 0, 0}
```

备注

scatter 是 thrust::gather 的逆过程

模板参数

- **InputIterator1**：必须是 Input Iterator 的模型，InputIterator1 的 value_type 必须可转换为 RandomAccessIterator 的 value_type
- **InputIterator2**：必须是 Input Iterator 的模型，InputIterator2 的 value_type 必须可转换为 RandomAccessIterator 的 difference_type
- **RandomAccessIteratorr**：必须是 Random Access Iterator 的模型

函数参数

- **first**：待分散的元素序列起始位置
- **last**：待分散的元素序列末尾位置
- **map**：输出序列的起始索引位置

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

- **result**：目标序列

前提条件

- 对于范围 [map,map + (last - first)) 内的所有迭代器 i ，迭代器 result + i 不应引用范围 [first,last) 内任意迭代器 j 引用的任何元素
- 对于范围 [map,map + (last - first)) 内的所有迭代器 i ，迭代器 result + i 不应引用范围 [map,map + (last - first)) 内任意迭代器 j 引用的任何元素
- 表达式 result[*i] 对范围 [map,map + (last - first)) 内的所有迭代器都有效

thrust::scatter_if

```
template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2,
  typename InputIterator3,
  typename RandomAccessIterator>
__host__ __device__ void
scatter_if(const thrust::detail::execution_policy_base< DerivedPolicy > &
↪exec,
  InputIterator1 first,
  InputIterator1 last,
  InputIterator2 map,
  InputIterator3 stencil,
  RandomAccessIterator output);
```

scatter_if 根据映射有条件地将元素从源范围复制到输出数组。对于范围 [first, last) 内使得 *(stencil + (i - first)) 的值为 true 的每个迭代器 i ，将值 *i 赋给 output[*(map + (i - first))]。 输出迭代器必须允许随机访问。如果相同的索引在范围 [map, map + (last - first)) 内多次出现，则结果未定义。

算法的执行由 exec 确定并行化。

```
#include <thrust/scatter.h>
#include <thrust/execution_policy.h>
...
int V[8] = {10, 20, 30, 40, 50, 60, 70, 80};
int M[8] = {0, 5, 1, 6, 2, 7, 3, 4};
int S[8] = {1, 0, 1, 0, 1, 0, 1, 0};
int D[8] = {0, 0, 0, 0, 0, 0, 0, 0};

thrust::scatter_if(thrust::host, V, V + 8, M, S, D);

// D contains [10, 30, 50, 70, 0, 0, 0, 0];
```

备注

scatter_if 是 thrust::gather_if 的逆过程

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **InputIterator1**：必须是 Input Iterator 的模型，InputIterator1 的 value_type 必须可转换为 RandomAccessIterator 的 value_type
- **InputIterator2**：必须是 Input Iterator 的模型，InputIterator2 的 value_type 必须可转换为 RandomAccessIterator 的 difference_type
- **InputIterator3**：必须是 Input Iterator 的模型，InputIterator3 的 value_type 必须可转换为 bool
- **RandomAccessIteratorr**：必须是 Random Access Iterator 的模型

沐曦信息 MetaX Confidential 2024-11-18 15:00:00

函数参数

- **first**：待分散的元素序列起始位置
- **last**：待分散的元素序列末尾位置
- **map**：输出序列的起始索引位置
- **stencil**：谓词值的序列起始索引位置
- **output**：目标序列起始索引位置

前提条件

- 对于范围 [map,map + (last - first)) 内的所有迭代器 i ，迭代器 result + i 不应引用范围 [first,last) 内任意迭代器 j 引用的任何元素
- 对于范围 [map,map + (last - first)) 内的所有迭代器 i ，迭代器 result + i 不应引用范围 [map,map + (last - first)) 内任意迭代器 j 引用的任何元素
- 对于范围 [map,map + (last - first)) 内的所有迭代器 i ，迭代器 result + i 不应引用范围 [stencil,stencil + (last - first)) 内任意迭代器 j 引用的任何元素
- 表达式 result[*i] 对范围 [map,map + (last - first)) 内使得 *(stencil + i) != false 的所有迭代器 i 有效

thrust::scatter_if

```
template <typename InputIterator1,
  typename InputIterator2,
  typename InputIterator3,
  typename RandomAccessIterator> void
scatter_if(InputIterator1 first,
  InputIterator2 last,
  InputIterator3 map,
  InputIterator4 stencil,
  RandomAccessIterator output);
```

scatter_if 根据映射有条件地将元素从源范围复制到输出数组。对于范围 [first, last) 内使得 *(stencil + (i - first)) 的值为 true 的每个迭代器 i ，将值 *i 赋给 output[*(map + (i - first))]。 输出迭代器必须允许随机访问。如果相同的索引在范围 [map, map + (last - first)) 内多次出现，则结果未定义。

```
#include <thrust/scatter.h>
...
int V[8] = {10, 20, 30, 40, 50, 60, 70, 80};
int M[8] = {0, 5, 1, 6, 2, 7, 3, 4};
int S[8] = {1, 0, 1, 0, 1, 0, 1, 0};
int D[8] = {0, 0, 0, 0, 0, 0, 0, 0};

thrust::scatter_if(V, V + 8, M, S, D);

// D contains [10, 30, 50, 70, 0, 0, 0, 0];
```

备注

scatter_if 是 thrust::gather_if 的逆过程

模板参数

- **InputIterator1**：必须是 Input Iterator 的模型，InputIterator1 的 value_type 必须可转换为 RandomAccessIterator 的 value_type
- **InputIterator2**：必须是 Input Iterator 的模型，InputIterator2 的 value_type 必须可转换为 RandomAccessIterator 的 difference_type

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

- **InputIterator3**：必须是 Input Iterator 的模型，InputIterator3 的 value_type 必须可转换为 bool
- **RandomAccessIteratorr**：必须是 Random Access Iterator 的模型

函数参数

- **first**：待分散的元素序列起始位置
- **last**：待分散的元素序列末尾位置
- **map**：输出序列的起始索引位置
- **stencil**：谓词值的序列起始索引位置
- **output**：目标序列起始索引位置

前提条件

- 对于范围 [map,map + (last - first)) 内的所有迭代器 i，迭代器 result + i 不应引用范围 [first,last) 内任意迭代器 j 引用的任何元素
- 对于范围 [map,map + (last - first)) 内的所有迭代器 i，迭代器 result + i 不应引用范围 [map,map + (last - first)) 内任意迭代器 j 引用的任何元素
- 对于范围 [map,map + (last - first)) 内的所有迭代器 i，迭代器 result + i 不应引用范围 [stencil,stencil + (last - first)) 内任意迭代器 j 引用的任何元素
- 表达式 result[*i] 对范围 [map,map + (last - first)) 内使得 *(stencil + i) != false 的所有迭代器 i 有效

thrust::scatter_if

```
template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2,
  typename InputIterator3,
  typename RandomAccessIterator,
  typename Predicate>
__host__ __device__ void
scatter_if(const thrust::detail::execution_policy_base< DerivedPolicy > &
 ↪exec,
  InputIterator1 first,
  InputIterator1 last,
  InputIterator2 map,
  InputIterator3 stencil,
  RandomAccessIterator output,
  Predicate pred);
```

scatter_if 根据映射有条件地将元素从源范围复制到输出数组。对于范围 [first, last) 内使得 pred(*(stencil + (i - first))) 的值为 true 的个迭代器 i，则将值 *i 赋给 output[*(map + (i - first))]。输出迭代器必须允许随机访问。如果相同的索引在范围 [map, map + (last - first)) 内多次出现，则结果未定义。

算法的执行由 exec 确定并行化。

```
#include <thrust/scatter.h>
#include <thrust/execution_policy.h>

struct is_even
{
__host__ __device__
bool operator()(int x)
{
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
    return (x % 2) == 0;
}
};

...

int V[8] = {10, 20, 30, 40, 50, 60, 70, 80};
int M[8] = {0, 5, 1, 6, 2, 7, 3, 4};
int S[8] = {2, 1, 2, 1, 2, 1, 2, 1};
int D[8] = {0, 0, 0, 0, 0, 0, 0, 0};

is_even pred;
thrust::scatter_if(thrust::host, V, V + 8, M, S, D, pred);

// D contains [10, 30, 50, 70, 0, 0, 0, 0];
```

备注

scatter_if 是 thrust::gather_if 的逆过程

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **InputIterator1**：必须是 Input Iterator 的模型，InputIterator1 的 value_type 必须可转换为 RandomAccessIterator 的 value_type
- **InputIterator2**：必须是 Input Iterator 的模型，InputIterator2 的 value_type 必须可转换为 RandomAccessIterator 的 difference_type
- **InputIterator3**：必须是 Input Iterator 的模型，InputIterator3 的 value_type 必须可转换为 Predicate 的 argument_type
- **RandomAccessIteratorr**：必须是 Random Access Iterator 的模型
- **Predicate**：必须是 Predicate 的模型

函数参数

- **exec**：用于并行化的执行策略
- **first**：待分散的元素序列起始位置
- **last**：待分散的元素序列末尾位置
- **map**：输出序列的起始索引位置
- **stencil**：谓词值序列起始索引位置
- **output**：目标序列起始索引位置
- **pred**：应用于模板值的谓词

前提条件

- 对于范围 [map,map + (last - first)) 内的所有迭代器 i ，迭代器 result + i 不应引用范围 [first,last) 内任意迭代器 j 引用的任何元素
- 对于范围 [map,map + (last - first)) 内的所有迭代器 i ，迭代器 result + i 不应引用范围 [map,map + (last - first)) 内任意迭代器 j 引用的任何元素
- 对于范围 [map,map + (last - first)) 内的所有迭代器 i ，迭代器 result + i 不应引用范围 [stencil,stencil + (last - first)) 内任意迭代器 j 引用的任何元素
- 表达式 result[*i] 对范围 [map,map + (last - first)) 内使得 pred(*(stencil + i)) != false 的所有迭代器 i 有效

飞腾信息 MetaX Confidential 2025-11-18 15:00:00

thrust::scatter_if

```
template <typename InputIterator1,
  typename InputIterator2,
  typename InputIterator3,
  typename RandomAccessIterator,
  typename Predicate> void
scatter_if(InputIterator1 first,
  InputIterator1 last,
  InputIterator2 map,
  InputIterator3 stencil,
  RandomAccessIterator output,
  Predicate pred);
```

scatter_if 根据映射有条件地将元素从源范围复制到输出数组。对于范围 [first, last) 内使得 pred(*(stencil + (i - first))) 的值为 true 的个迭代器 i，则将值 *i 赋给 output[*(map + (i - first))]。输出迭代器必须允许随机访问。如果相同的索引在范围 [map, map + (last - first)) 内多次出现，则结果未定义。

```
#include <thrust/scatter.h>

struct is_even
{
__host__ __device__
bool operator()(int x)
{
   return (x % 2) == 0;
}
};

...

int V[8] = {10, 20, 30, 40, 50, 60, 70, 80};
int M[8] = {0, 5, 1, 6, 2, 7, 3, 4};
int S[8] = {2, 1, 2, 1, 2, 1, 2, 1};
int D[8] = {0, 0, 0, 0, 0, 0, 0, 0};

is_even pred;
thrust::scatter_if(V, V + 8, M, S, D, pred);

// D contains [10, 30, 50, 70, 0, 0, 0, 0];
```

备注

scatter_if 是 thrust::gather_if 的逆过程

模板参数

- **InputIterator1**：必须是 Input Iterator 的模型，InputIterator1 的 value_type 必须可转换为 RandomAccessIterator 的 value_type
- **InputIterator2**：必须是 Input Iterator 的模型，InputIterator2 的 value_type 必须可转换为 RandomAccessIterator 的 difference_type
- **InputIterator3**：必须是 Input Iterator 的模型，InputIterator3 的 value_type 必须可转换为 Predicate 的 argument_type
- **RandomAccessIteratorr**：必须是 Random Access Iterator 的模型
- **Predicate**：必须是 Predicate 的模型

函数参数

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

- **first**：待分散的元素序列起始位置
- **last**：待分散的元素序列末尾位置
- **map**：输出序列的起始索引位置
- **stencil**：谓词值的序列起始位置
- **output**：目标序列起始位置
- **pred**：应用于模板值的谓词

前提条件

- 对于范围 [map,map + (last - first)) 内的所有迭代器 i ，迭代器 result + i 不应引用范围 [first,last) 内任意迭代器 j 引用的任何元素
- 对于范围 [map,map + (last - first)) 内的所有迭代器 i ，迭代器 result + i 不应引用范围 [map,map + (last - first)) 内任意迭代器 j 引用的任何元素
- 对于范围 [map,map + (last - first)) 内的所有迭代器 i ，迭代器 result + i 不应引用范围 [stencil,stencil + (last - first)) 内任意迭代器 j 引用的任何元素
- 表达式 result[*i] 对范围 [map,map + (last - first)) 内使得 pred(*(stencil + i)) != false 的所有迭代器 i 有效

2.1.2 合并（Merging）

函数

```
template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator>
__host__ __device__ OutputIterator
thrust::merge(const thrust::detail::execution_policy_base< DerivedPolicy >
 ↪& exec,
  InputIterator1 first1,
  InputIterator1 last1,
  InputIterator2 first2,
  InputIterator2 last2,
  OutputIterator result);

template <typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator>
OutputIterator
thrust::merge(InputIterator1 first1,
  InputIterator1 last1,
  InputIterator2 first2,
  InputIterator2 last2,
  OutputIterator result);

template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator,
  typename StrictWeakCompare>
__host__ __device__ OutputIterator
thrust::merge(const thrust::detail::execution_policy_base< DerivedPolicy >
 ↪& exec,
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
  InputIterator1 first1,
  InputIterator1 last1,
  InputIterator2 first2,
  InputIterator2 last2,
  OutputIterator result,
  StrictWeakCompare comp);

template <typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator,
  typename StrictWeakCompare>
OutputIterator
thrust::merge(InputIterator1 first1,
  InputIterator1 last1,
  InputIterator2 first2,
  InputIterator2 last2,
  OutputIterator result,
  StrictWeakCompare comp);

template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2,
  typename InputIterator3,
  typename InputIterator4,
  typename OutputIterator1,
  typename OutputIterator2>
__host__ __device__ thrust::pair< OutputIterator1, OutputIterator2 >
thrust::merge_by_key(const thrust::detail::execution_policy_base<
 ↪DerivedPolicy > & exec,
  InputIterator1 keys_first1,
  InputIterator1 keys_last1,
  InputIterator2 keys_first2,
  InputIterator2 keys_last2,
  InputIterator3 values_first1,
  InputIterator4 values_first2,
  OutputIterator1 keys_result,
  OutputIterator2 values_result);

template <typename InputIterator1,
  typename InputIterator2,
  typename InputIterator3,
  typename InputIterator4,
  typename OutputIterator1,
  typename OutputIterator2> t
hrust::pair< OutputIterator1, OutputIterator2 >
thrust::merge_by_key(InputIterator1 keys_first1,
  InputIterator1 keys_last1,
  InputIterator2 keys_first2,
  InputIterator2 keys_last2,
  InputIterator3 values_first1,
  InputIterator4 values_first2,
  OutputIterator1 keys_result,
  OutputIterator2 values_result);

template <typename DerivedPolicy,
  typename InputIterator1,
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
  typename InputIterator2,
  typename InputIterator3,
  typename InputIterator4,
  typename OutputIterator1,
  typename OutputIterator2,
  typename Compare>
__host__ __device__ thrust::pair< OutputIterator1, OutputIterator2 >
thrust::merge_by_key(const thrust::detail::execution_policy_base<
↪DerivedPolicy > & exec,
  InputIterator1 keys_first1,
  InputIterator1 keys_last1,
  InputIterator2 keys_first2,
  InputIterator2 keys_last2,
  InputIterator3 values_first1,
  InputIterator4 values_first2,
  OutputIterator1 keys_result,
  OutputIterator2 values_result,
  Compare comp);

template <typename InputIterator1,
  typename InputIterator2,
  typename InputIterator3,
  typename InputIterator4,
  typename OutputIterator1,
  typename OutputIterator2,
  typename StrictWeakCompare>
thrust::pair< OutputIterator1, OutputIterator2 >

thrust::merge_by_key(InputIterator1 keys_first1,
  InputIterator1 keys_last1,
  InputIterator2 keys_first2,
  InputIterator2 keys_last2,
  InputIterator3 values_first1,
  InputIterator4 values_first2,
  OutputIterator1 keys_result,
  OutputIterator2 values_result,
  StrictWeakCompare comp);
```

2.1.2.1 thrust::merge

```
template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator>
__host__ __device__ OutputIterator
merge(const thrust::detail::execution_policy_base< DerivedPolicy > & exec,
  InputIterator1 first1,
  InputIterator1 last1,
  InputIterator2 first2,
  InputIterator2 last2,
  OutputIterator result);
```

merge 将两个有序范围 [first1, last1) 和 [first2, last2) 合并到一个有序范围内。即，将元素从 [first1, last1) 和 [first2, last2) 复制到 [result, result + (last1 - first1) + (last2 - first2)) 内，使得结果范围按升序排序。merge 是稳定的，指的是每个输入范围内元素的相对顺序保持不变，并且对于两个输入范围内的

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

等价元素，第一个范围内的元素先于第二个范围的元素。

返回值是 result + (last1 - first1) + (last2 - first2)。

此版本的 merge 使用 operator< 比较元素。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::host 并行化执行策略，使用 merge 实现两个有序整数集合的合并：

```
#include <thrust/merge.h>
#include <thrust/execution_policy.h>
...
int A1[6] = {1, 3, 5, 7, 9, 11};
int A2[7] = {1, 1, 2, 3, 5,  8, 13};

int result[13];

int *result_end =
thrust::merge(thrust::host,
A1, A1 + 6,
A2, A2 + 7,
result);
// result = {1, 1, 1, 2, 3, 3, 5, 5, 7, 8, 9, 11, 13}
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **InputIterator1**：是 Input Iterator 的模型。InputIterator1 和 InputIterator2 具有相同的 value_type，InputIterator1 的 value_type 是 LessThan Comparable 的模型，如 LessThan Comparable 要求中定义的，InputIterator1 的 value_type 上的排序是严格弱序，并且可转换为 OutputIterator 的 value_types 集合中的类型
- **InputIterator2**：是 Input Iterator 的模型。InputIterator2 和 InputIterator1 具有相同的 value_type，InputIterator2 的 value_type 是 LessThan Comparable 的模型，如 LessThan Comparable 要求中定义的，InputIterator2 的 value_type 上的排序是严格弱序，并且可转换为 OutputIterator 的 value_types 集合中的类型
- **OutputIterator**：是 Output Iterator 的模型

函数参数

- **exec**：用于并行化的执行策略
- **first1**：第一个输入范围的起始
- **last1**：第一个输入范围的末尾
- **first2**：第二个输入范围的起始
- **last2**：第二个输入范围的末尾
- **result**：合并输出的起始

前提条件

- 范围 [first1, last1) 和 [first2, last2) 应按照 operator< 进行排序
- 结果范围不应与任何一个输入范围重叠

返回值

输出范围的末尾

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/merge

沐曦 MetaX Confidential 2024-11-16 15:00:00

- set_union
- sort
- is_sorted

2.1.2.2 thrust::merge

```
template <typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator>
OutputIterator
merge(InputIterator1 first1,
  InputIterator1 last1,
  InputIterator2 first2,
  InputIterator2 last2,
  OutputIterator result);
```

merge 将两个有序范围 [first1, last1) 和 [first2, last2) 合并到一个有序范围内。即，将元素从 [first1, last1) 和 [first2, last2) 复制到 [result, result + (last1 - first1) + (last2 - first2)) 内，使得结果范围按升序排序。merge 是稳定的，指的是每个输入范围内元素的相对顺序保持不变，并且对于两个输入范围内的等价元素，第一个范围内的元素先于第二个范围的元素。

返回值是 result + (last1 - first1) + (last2 - first2)。

此版本的 merge 使用 operator< 比较元素。

以下代码片段演示了如何使用 merge 实现两个有序整数集合的合并：

```
#include <thrust/merge.h>
...
int A1[6] = {1, 3, 5, 7, 9, 11};
int A2[7] = {1, 1, 2, 3, 5,  8, 13};

int result[13];

int *result_end = thrust::merge(A1, A1 + 6, A2, A2 + 7, result);
// result = {1, 1, 1, 2, 3, 3, 5, 5, 7, 8, 9, 11, 13}
```

模板参数

- **InputIterator1**：是 Input Iterator 的模型。InputIterator1 和 InputIterator2 具有相同的 value_type，InputIterator1 的 value_type 是 LessThan Comparable 的模型，如 LessThan Comparable 要求中定义的，InputIterator1 的 value_type 上的排序是严格弱序，并且可转换为 OutputIterator 的 value_types 集合中的类型
- **InputIterator2**：是 Input Iterator 的模型。InputIterator2 和 InputIterator1 具有相同的 value_type，InputIterator2 的 value_type 是 LessThan Comparable 的模型，如 LessThan Comparable 要求中定义的，InputIterator2 的 value_type 上的排序是严格弱序，并且可转换为 OutputIterator 的 value_types 集合中的类型
- **OutputIterator**：是 Output Iterator 的模型

函数参数

- **first1**：第一个输入范围的起始
- **last1**-：第一个输入范围的末尾
- **first2**：第二个输入范围的起始
- **last2**：第二个输入范围的末尾
- **result**：合并输出的起始

飞腾信息 Metax Confidential 2024-11-18 15:00:00

前提条件

- 范围 [first1, last1) 和 [first2, last2) 应按照 operator< 进行排序
- 结果范围不应与任何一个输入范围重叠

返回值

输出范围的末尾

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/merge
- set_union
- sort
- is_sorted

2.1.2.3 thrust::merge

```
template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator,
  typename StrictWeakCompare>
__host__ __device__ OutputIterator
merge(const thrust::detail::execution_policy_base< DerivedPolicy > & exec,
  InputIterator1 first1,
  InputIterator1 last1,
  InputIterator2 first2,
  InputIterator2 last2,
  OutputIterator result,
  StrictWeakCompare comp);
```

merge 将两个有序范围 [first1, last1) 和 [first2, last2) 合并到一个有序范围内。即，将元素从 [first1, last1) 和 [first2, last2) 复制到 [result, result + (last1 - first1) + (last2 - first2)) 内，使得结果范围按升序排序。merge 是稳定的，指的是每个输入范围内元素的相对顺序保持不变，并且对于两个输入范围内的等价元素，第一个范围内的元素先于第二个范围的元素。

返回值是 result + (last1 - first1) + (last2 - first2)。

此版本的 merge 使用函数对象 comp 比较元素。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::host 并行化执行策略，使用 merge 实现两组降序整数集合的合并：

```
#include <thrust/merge.h>
#include <thrust/functional.h>
#include <thrust/execution_policy.h>
...
int A1[6] = {11, 9, 7, 5, 3, 1};
int A2[7] = {13, 8, 5, 3, 2, 1, 1};

int result[13];

int *result_end = thrust::merge(thrust::host,
A1, A1 + 6,
A2, A2 + 7,
result,
```

(下页继续)

MetaX Confidential 2024-11-8 15:00:00

(续上页)

```
thrust::greater<int>());
// result = {13, 11, 9, 8, 7, 5, 5, 3, 3, 2, 1, 1, 1}
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **InputIterator1**：是 Input Iterator 的模型，InputIterator1 的 value_type 可转换为 StrictWeakCompare 的 first_argument_type 和 OutputIterator 的 value_types 集合中的类型
- **InputIterator2**：是 Input Iterator 的模型，InputIterator2 的 value_type 可转换为 StrictWeakCompare 的 second_argument_type 和 OutputIterator 的 value_types 集合中的类型
- **OutputIterator**：是 Output Iterator 的模型
- **StrictWeakCompare**：是 Strict Weak Ordering 的模型

函数参数

- **exec**：用于并行化的执行策略
- **first1**：第一个输入范围的起始
- **last1**-：第一个输入范围的末尾
- **first2**：第二个输入范围的起始
- **last2**：第二个输入范围的末尾
- **result**：合并输出的起始
- **comp**：比较运算符

前提条件

- 范围 [first1, last1) 和 [first2, last2) 应按照 comp 进行排序
- 结果范围不应与任何一个输入范围重叠

返回值

输出范围的末尾

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/merge
- sort
- is_sorted

2.1.2.4 thrust::merge

```
template <typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator,
  typename StrictWeakCompare>
OutputIterator
merge(InputIterator1 first1,
  InputIterator1 last1,
  InputIterator2 first2,
  InputIterator2 last2,
  OutputIterator result,
  StrictWeakCompare comp);
```

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

merge 将两个有序范围 [first1, last1) 和 [first2, last2) 合并到一个有序范围内。即，将元素从 [first1, last1) 和 [first2, last2) 复制到 [result, result + (last1 - first1) + (last2 - first2)) 内，使得结果范围按升序排序。merge 是稳定的，指的是每个输入范围内元素的相对顺序保持不变，并且对于两个输入范围内的等价元素，第一个范围内的元素先于第二个范围的元素。

返回值是 result + (last1 - first1) + (last2 - first2)。

此版本的 merge 使用函数对象 comp 比较元素。

以下代码片段演示了如何使用 merge 实现两组降序整数集合的合并：

```
#include <thrust/merge.h>
#include <thrust/functional.h>
...
int A1[6] = {11, 9, 7, 5, 3, 1};
int A2[7] = {13, 8, 5, 3, 2, 1, 1};

int result[13];

int *result_end = thrust::merge(A1, A1 + 6, A2, A2 + 7, result,
↪thrust::greater<int>());
// result = {13, 11, 9, 8, 7, 5, 5, 3, 3, 2, 1, 1, 1}
```

模板参数

- **InputIterator1**：是 Input Iterator 的模型，InputIterator1 的 value_type 可转换为 StrictWeakCompare 的 first_argument_type 和 OutputIterator 的 value_types 集合中的类型
- **InputIterator2**：是 Input Iterator 的模型，InputIterator2 的 value_type 可转换为 StrictWeakCompare 的 second_argument_type 和 OutputIterator 的 value_types 集合中的类型
- **OutputIterator**：是 Output Iterator 的模型
- **StrictWeakCompare**：是 Strict Weak Ordering 的模型

函数参数

- **first1**：第一个输入范围的起始
- **last1**-：第一个输入范围的末尾
- **first2**：第二个输入范围的起始
- **last2**：第二个输入范围的末尾
- **result**：合并输出的起始
- **comp**：比较运算符

前提条件

- 范围 [first1, last1) 和 [first2, last2) 应按照 comp 进行排序
- 结果范围不应与任何一个输入范围重叠

返回值

输出范围的末尾

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/merge
- sort
- is_sorted

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

2.1.2.5 thrust::merge_by_key

```
template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2,
  typename InputIterator3,
  typename InputIterator4,
  typename OutputIterator1,
  typename OutputIterator2>
__host__ __device__ thrust::pair< OutputIterator1, OutputIterator2 >
merge_by_key(const thrust::detail::execution_policy_base< DerivedPolicy > &
 ↪ exec,
  InputIterator1 keys_first1,
  InputIterator1 keys_last1,
  InputIterator2 keys_first2,
  InputIterator2 keys_last2,
  InputIterator3 values_first1,
  InputIterator4 values_first2,
  OutputIterator1 keys_result,
  OutputIterator2 values_result);
```

merge_by_key 执行键值（key-value）合并。即，merge_by_key 将元素从 [keys_first1, keys_last1) 和 [keys_first2, keys_last2) 复制到范围 [keys_result, keys_result + (keys_last1 - keys_first1) + (keys_last2 - keys_first2)) 内，使得结果范围按键（key）升序排序。

同时，merge_by_key 将两个关联范围 [values_first1 + (keys_last1 - keys_first1)) 和 [values_first2 + (keys_last2 - keys_first2)) 的元素复制到范围 [values_result, values_result + (keys_last1 - keys_first1) + (keys_last2 - keys_first2)) 内，使得结果范围按每个输入元素关联键隐含的升序排列。

merge_by_key 是稳定的，指的是每个输入范围内元素的相对顺序保持不变，并且对于所有输入键范围内的等价元素，第一个范围内的元素先于第二个范围的元素。

返回值是 (keys_result + (keys_last1 - keys_first1) + (keys_last2 - keys_first2)) 和 (values_result + (keys_last1 - keys_first1) + (keys_last2 - keys_first2))。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::host 并行化执行策略，使用 merge_by_key 实现两组升序整数集合的合并：

```
#include <thrust/merge.h>
#include <thrust/functional.h>
#include <thrust/execution_policy.h>
...
int A_keys[6] = {1, 3, 5, 7, 9, 11};
int A_vals[6] = {0, 0, 0, 0, 0, 0};

int B_keys[7] = {1, 1, 2, 3, 5, 8, 13};
int B_vals[7] = {1, 1, 1, 1, 1, 1, 1};

int keys_result[13];
int vals_result[13];

thrust::pair<int*,int*> end =
thrust::merge_by_key(thrust::host,
A_keys, A_keys + 6,
B_keys, B_keys + 7,
A_vals, B_vals,
keys_result, vals_result);
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:10:00

(续上页)

```
// keys_result = {1, 1, 1, 2, 3, 3, 5, 5, 7, 8, 9, 11, 13}
// vals_result = {0, 1, 1, 1, 0, 1, 0, 1, 0, 1, 0,  0,  1}
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **InputIterator1**：是 Input Iterator 的模型。InputIterator1 和 InputIterator2 具有相同的 value_type，InputIterator1 的 value_type 是 LessThan Comparable 的模型，如 LessThan Comparable 要求中定义的，InputIterator1 的 value_type 上的排序是严格弱序，并且可转换为 OutputIterator 的 value_types 集合中的类型
- **InputIterator2**：是 Input Iterator 的模型。InputIterator2 和 InputIterator1 具有相同的 value_type，InputIterator2 的 value_type 是 LessThan Comparable 的模型，如 LessThan Comparable 要求中定义的，InputIterator2 的 value_type 上的排序是严格弱序，并且可转换为 OutputIterator 的 value_types 集合中的类型
- **InputIterator3**：是 Input Iterator 的模型，InputIterator3 的 value_type 可转换为 OutputIterator2 的 value_types 集合中的类型
- **InputIterator4**：是 Input Iterator 的模型，InputIterator4 的 value_type 可转换为 OutputIterator2 的 value_types 集合中的类型
- **OutputIterator1**：是 Output Iterator 的模型
- **OutputIterator2**：是 Output Iterator 的模型

函数参数

- **exec**：用于并行化的执行策略
- **keys_first1**：第一个键输入范围的起始
- **keys_last1**：第一个键输入范围的末尾
- **keys_first2**：第二个键输入范围的起始
- **keys_last2**：第二个键输入范围的末尾
- **values_first1**：第一个值输入范围的起始
- **values_first2**：第二个值输入范围的起始
- **keys_result**：键合并输出范围的起始
- **values_result**：值合并输出范围的起始

前提条件

- 范围 [keys_first1, keys_last1) 和 [keys_first2, keys_last2) 应按照 operator< 进行排序
- 结果范围不应与任何一个输入范围重叠

返回值

p pair，其中 p.first 是键输出范围的末尾，p.second 是值输出范围的末尾

另请参阅

- merge
- sort_by_key
- is_sorted

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

2.1.2.6 thrust::merge_by_key

```
template <typename InputIterator1,
  typename InputIterator2,
  typename InputIterator3,
  typename InputIterator4,
  typename OutputIterator1,
  typename OutputIterator2>
thrust::pair< OutputIterator1, OutputIterator2 >
merge_by_key(InputIterator1 keys_first1,
  InputIterator1 keys_last1,
  InputIterator2 keys_first2,
  InputIterator2 keys_last2,
  InputIterator3 values_first1,
  InputIterator4 values_first2,
  OutputIterator1 keys_result,
  OutputIterator2 values_result);
```

merge_by_key 执行键值合并。即，merge_by_key 将元素从 [keys_first1, keys_last1) 和 [keys_first2, keys_last2) 复制到范围 [keys_result, keys_result + (keys_last1 - keys_first1) + (keys_last2 - keys_first2)) 内，使得结果范围按键升序排序。

同时，merge_by_key 将两个关联范围 [values_first1 + (keys_last1 - keys_first1)) 和 [values_first2 + (keys_last2 - keys_first2)) 的元素复制到范围 [values_result, values_result + (keys_last1 - keys_first1) + (keys_last2 - keys_first2)) 内，使得结果范围按每个输入元素关联键隐含的升序排列。

merge_by_key 是稳定的，指的是每个输入范围内元素的相对顺序保持不变，并且对于所有输入键范围内的等价元素，第一个范围内的元素先于第二个范围的元素。

返回值是 (keys_result + (keys_last1 - keys_first1) + (keys_last2 - keys_first2)) 和 (values_result + (keys_last1 - keys_first1) + (keys_last2 - keys_first2))。

以下代码片段演示了如何使用 merge_by_key 实现两组升序整数集合的合并：

```
#include <thrust/merge.h>
#include <thrust/functional.h>
...
int A_keys[6] = {1, 3, 5, 7, 9, 11};
int A_vals[6] = {0, 0, 0, 0, 0, 0};

int B_keys[7] = {1, 1, 2, 3, 5, 8, 13};
int B_vals[7] = {1, 1, 1, 1, 1, 1, 1};

int keys_result[13];
int vals_result[13];

thrust::pair<int*,int*> end = thrust::merge_by_key(A_keys, A_keys + 6, B_
↪keys, B_keys + 7, A_vals, B_vals, keys_result, vals_result);

// keys_result = {1, 1, 1, 2, 3, 3, 5, 5, 7, 8, 9, 11, 13}
// vals_result = {0, 1, 1, 1, 0, 1, 0, 1, 0, 1, 0,  0,  1}
```

模板参数

- **InputIterator1**：是 Input Iterator 的模型。InputIterator1 和 InputIterator2 具有相同的 value_type，InputIterator1 的 value_type 是 LessThan Comparable 的模型，如 LessThan Comparable 要求中定义的，InputIterator1 的 value_type 上的排序是严格弱序，并且可转换为 OutputIterator 的 value_types 集合中的类型
- **InputIterator2**：是 Input Iterator 的模型。InputIterator2 和 InputIterator1 具有相同的 value_type，InputIterator2 的 value_type 是 LessThan Comparable 的模型，如 LessThan Comparable 要求

飞腾信息技术有限公司 MetaX Confidential 2024-11-18 15:09:00

中定义的，InputIterator2 的 value_type 上的排序是严格弱序，并且可转换为 OutputIterator 的 value_types 集合中的类型

- **InputIterator3**：是 Input Iterator 的模型，InputIterator3 的 value_type 可转换为 OutputIterator2 的 value_types 集合中的类型
- **InputIterator4**：是 Input Iterator 的模型，InputIterator4 的 value_type 可转换为 OutputIterator2 的 value_types 集合中的类型
- **OutputIterator1**：是 Output Iterator 的模型
- **OutputIterator2**：是 Output Iterator 的模型

函数参数

- **keys_first1**：第一个键输入范围的起始
- **keys_last1**：第一个键输入范围的末尾
- **keys_first2**：第二个键输入范围的起始
- **keys_last2**：第二个键输入范围的末尾
- **values_first1**：第一个值输入范围的起始
- **values_first2**：第二个值输入范围的起始
- **keys_result**：键合并输出范围的起始
- **values_result**：值合并输出范围的起始

前提条件

- 范围 [keys_first1, keys_last1) 和 [keys_first2, keys_last2) 应按照 operator< 进行排序
- 结果范围不应与任何一个输入范围重叠

返回值

p pair ，其中 p.first 是键输出范围的末尾，p.second 是值输出范围的末尾

另请参阅

- merge
- sort_by_key
- is_sorted

2.1.2.7 thrust::merge_by_key

```
template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2,
  typename InputIterator3,
  typename InputIterator4,
  typename OutputIterator1,
  typename OutputIterator2,
  typename Compare>
__host__ __device__ thrust::pair< OutputIterator1, OutputIterator2 >
merge_by_key(const thrust::detail::execution_policy_base< DerivedPolicy > &
 ↪ exec,
  InputIterator1 keys_first1,
  InputIterator1 keys_last1,
  InputIterator2 keys_first2,
  InputIterator2 keys_last2,
  InputIterator3 values_first1,
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
  InputIterator4 values_first2,
  OutputIterator1 keys_result,
  OutputIterator2 values_result,
  Compare comp);
```

merge_by_key 执行键值合并。即，merge_by_key 将元素从 [keys_first1, keys_last1) 和 [keys_first2, keys_last2) 复制到范围 [keys_result, keys_result + (keys_last1 - keys_first1) + (keys_last2 - keys_first2)) 内，使得结果范围按键升序排序。

同时，merge_by_key 将两个关联范围 [values_first1 + (keys_last1 - keys_first1)) 和 [values_first2 + (keys_last2 - keys_first2)) 的元素复制到范围 [values_result, values_result + (keys_last1 - keys_first1) + (keys_last2 - keys_first2)) 内，使得结果范围按每个输入元素关联键隐含的升序排列。

merge_by_key 是稳定的，指的是每个输入范围内元素的相对顺序保持不变，并且对于所有输入键范围内的等价元素，第一个范围内的元素先于第二个范围的元素。

返回值是 (keys_result + (keys_last1 - keys_first1) + (keys_last2 - keys_first2)) 和 (values_result + (keys_last1 - keys_first1) + (keys_last2 - keys_first2))。

此版本的 merge_by_key 使用函数对象 comp 比较键元素。

用 exec 并行化算法的执行。

以下代码片段演示了如何通过 thrust::host 并行化执行策略，使用 merge_by_key 实现两组降序整数集合的合并：

```
#include <thrust/merge.h>
#include <thrust/functional.h>
#include <thrust/execution_policy.h>
...
int A_keys[6] = {11, 9, 7, 5, 3, 1};
int A_vals[6] = { 0, 0, 0, 0, 0, 0};

int B_keys[7] = {13, 8, 5, 3, 2, 1, 1};
int B_vals[7] = { 1, 1, 1, 1, 1, 1, 1};

int keys_result[13];
int vals_result[13];

thrust::pair<int*,int*> end =
thrust::merge_by_key(thrust::host,
A_keys, A_keys + 6,
B_keys, B_keys + 7,
A_vals, B_vals,
keys_result, vals_result,
thrust::greater<int>());

// keys_result = {13, 11, 9, 8, 7, 5, 5, 3, 3, 2, 1, 1, 1}
// vals_result = { 1,  0, 0, 1, 0, 0, 1, 0, 1, 1, 0, 1, 1}
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **InputIterator1**：是 Input Iterator 的模型，InputIterator1 的 value_type 可转换为 StrictWeakCompare 的 first_argument_type 和 OutputIterator1 的 value_types 集合中的类型
- **InputIterator2**：是 Input Iterator 的模型，InputIterator2 的 value_type 可转换为 StrictWeakCompare 的 second_argument_type 和 OutputIterator1 的 value_types 集合中的类型
- **InputIterator3**：是 Input Iterator 的模型，InputIterator3 的 value_type 可转换为 OutputIterator2 的 value_types 集合中的类型

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

- **InputIterator4**：是 Input Iterator 的模型，InputIterator4 的 value_type 可转换为 OutputIterator2 的 value_types 集合中的类型
- **OutputIterator1**：是 Output Iterator 的模型
- **OutputIterator2**：是 Output Iterator 的模型
- **StrictWeakCompare**：是 Strict Weak Ordering 的模型

函数参数

- **exec**：用于并行化的执行策略
- **keys_first1**：第一个键输入范围的起始
- **keys_last1**：第一个键输入范围的末尾
- **keys_first2**：第二个键输入范围的起始
- **keys_last2**：第二个键输入范围的末尾
- **values_first1**：第一个值输入范围的起始
- **values_first2**：第二个值输入范围的起始
- **keys_result**：键合并输出范围的起始
- **values_result**：值合并输出范围的起始
- **comp**：比较运算符

前提条件

- 范围 [keys_first1, keys_last1) 和 [keys_first2, keys_last2) 应按照 comp 进行排序
- 结果范围不应与任何一个输入范围重叠

返回值

p pair ，其中 p.first 是键输出范围的末尾，p.second 是值输出范围的末尾

另请参阅

- merge
- sort_by_key
- is_sorted

2.1.2.8 thrust::merge_by_key

```
template <typename InputIterator1,
  typename InputIterator2,
  typename InputIterator3,
  typename InputIterator4,
  typename OutputIterator1,
  typename OutputIterator2,
  typename StrictWeakCompare> thrust::pair< OutputIterator1,
↪OutputIterator2 >
merge_by_key(InputIterator1 keys_first1,
  InputIterator1 keys_last1,
  InputIterator2 keys_first2,
  InputIterator2 keys_last2,
  InputIterator3 values_first1,
  InputIterator4 values_first2,
  OutputIterator1 keys_result,
  OutputIterator2 values_result,
  StrictWeakCompare comp);
```

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

merge_by_key 执行键值合并。即，merge_by_key 将元素从 [keys_first1, keys_last1) 和 [keys_first2, keys_last2) 复制到范围 [keys_result, keys_result + (keys_last1 - keys_first1) + (keys_last2 - keys_first2)) 内，使得结果范围按键升序排序。

同时，merge_by_key 将两个关联范围 [values_first1 + (keys_last1 - keys_first1)) 和 [values_first2 + (keys_last2 - keys_first2)) 的元素复制到范围 [values_result, values_result + (keys_last1 - keys_first1) + (keys_last2 - keys_first2)) 内，使得结果范围按每个输入元素关联键隐含的升序排列。

merge_by_key 是稳定的，指的是每个输入范围内元素的相对顺序保持不变，并且对于所有输入键范围内的等价元素，第一个范围内的元素先于第二个范围的元素。

返回值是 (keys_result + (keys_last1 - keys_first1) + (keys_last2 - keys_first2)) 和 (values_result + (keys_last1 - keys_first1) + (keys_last2 - keys_first2))。

此版本的 merge_by_key 使用函数对象 comp 比较键元素。

以下代码片段演示了如何使用 merge_by_key 实现两组降序整数集合的合并：

```
#include <thrust/merge.h>
#include <thrust/functional.h>
...
int A_keys[6] = {11, 9, 7, 5, 3, 1};
int A_vals[6] = { 0, 0, 0, 0, 0, 0};

int B_keys[7] = {13, 8, 5, 3, 2, 1, 1};
int B_vals[7] = { 1, 1, 1, 1, 1, 1, 1};

int keys_result[13];
int vals_result[13];

thrust::pair<int*,int*> end = thrust::merge_by_key(A_keys, A_keys + 6, B_
↪keys, B_keys + 7, A_vals, B_vals, keys_result, vals_result,
↪thrust::greater<int>());

// keys_result = {13, 11, 9, 8, 7, 5, 5, 3, 3, 2, 1, 1, 1}
// vals_result = { 1,  0, 0, 1, 0, 0, 1, 0, 1, 1, 0, 1, 1}
```

模板参数

- **InputIterator1**：是 Input Iterator 的模型，InputIterator1 的 value_type 可转换为 StrictWeakCompare 的 first_argument_type 和 OutputIterator1 的 value_types 集合中的类型
- **InputIterator2**：是 Input Iterator 的模型，InputIterator2 的 value_type 可转换为 StrictWeakCompare 的 second_argument_type 和 OutputIterator1 的 value_types 集合中的类型
- **InputIterator3**：是 Input Iterator 的模型，InputIterator3 的 value_type 可转换为 OutputIterator2 的 value_types 集合中的类型
- **InputIterator4**：是 Input Iterator 的模型，InputIterator4 的 value_type 可转换为 OutputIterator2 的 value_types 集合中的类型
- **OutputIterator1**：是 Output Iterator 的模型
- **OutputIterator2**：是 Output Iterator 的模型
- **StrictWeakCompare**：是 Strict Weak Ordering 的模型

函数参数

- **keys_first1**：第一个键输入范围的起始
- **keys_last1**：第一个键输入范围的末尾
- **keys_first2**：第二个键输入范围的起始
- **keys_last2**：第二个键输入范围的末尾

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

- **values_first1**：第一个值输入范围的起始
- **values_first2**：第二个值输入范围的起始
- **keys_result**：键合并输出范围的起始
- **values_result**：值合并输出范围的起始
- **comp**：比较运算符

前提条件

- 范围 [keys_first1, keys_last1) 和 [keys_first2, keys_last2) 应按照 comp 进行排序
- 结果范围不应与任何一个输入范围重叠

返回值

p pair，其中 p.first 是键输出范围的末尾，p.second 是值输出范围的末尾

另请参阅

- merge
- sort_by_key
- is_sorted

2.1.3 前缀和（Prefix Sum）

2.1.3.1 前缀和

函数

```
template <typename DerivedPolicy,
  typename InputIterator,
  typename OutputIterator>
__host__ __device__ OutputIterator
thrust::inclusive_scan(const thrust::detail::execution_policy_base<
↪DerivedPolicy > & exec,
  InputIterator first,
  InputIterator last,
  OutputIterator result);

template <typename InputIterator,
  typename OutputIterator>
OutputIterator
thrust::inclusive_scan(InputIterator first,
  InputIterator last,
  OutputIterator result);

template <typename DerivedPolicy,
  typename InputIterator,
  typename OutputIterator,
  typename AssociativeOperator>

__host__ __device__ OutputIterator
thrust::inclusive_scan(const thrust::detail::execution_policy_base<
↪DerivedPolicy > & exec,
  InputIterator first,
  InputIterator last,
  OutputIterator result,
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
  AssociativeOperator binary_op);

template <typename InputIterator,
  typename OutputIterator,
  typename AssociativeOperator>
OutputIterator
thrust::inclusive_scan(InputIterator first,
  InputIterator last,
  OutputIterator result,
  AssociativeOperator binary_op);

template <typename DerivedPolicy,
  typename InputIterator,
  typename OutputIterator>
__host__ __device__ OutputIterator
thrust::exclusive_scan(const thrust::detail::execution_policy_base<
↪DerivedPolicy > & exec,
  InputIterator first,
  InputIterator last,
  OutputIterator result);

template <typename InputIterator,
  typename OutputIterator>
OutputIterator
thrust::exclusive_scan(InputIterator first,
  InputIterator last,
  OutputIterator result);

template <typename DerivedPolicy,
  typename InputIterator,
  typename OutputIterator,
  typename T>
__host__ __device__ OutputIterator
thrust::exclusive_scan(const thrust::detail::execution_policy_base<
↪DerivedPolicy > & exec,
  InputIterator first,
  InputIterator last,
  OutputIterator result,
  T init);

template <typename InputIterator,
  typename OutputIterator,
  typename T>
OutputIterator
thrust::exclusive_scan(InputIterator first,
  InputIterator last,
  OutputIterator result,
  T init);

template <typename DerivedPolicy,
  typename InputIterator,
  typename OutputIterator,
  typename T,
  typename AssociativeOperator>
__host__ __device__ OutputIterator
thrust::exclusive_scan(const thrust::detail::execution_policy_base<
↪DerivedPolicy > & exec,
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
  InputIterator first,
  InputIterator last,
  OutputIterator result,
  T init,
  AssociativeOperator binary_op);

template <typename InputIterator,
  typename OutputIterator,
  typename T,
  typename AssociativeOperator>
OutputIterator
thrust::exclusive_scan(InputIterator first,
  InputIterator last,
  OutputIterator result,
  T init,
  AssociativeOperator binary_op);
```

thrust::inclusive_scan

```
template <typename DerivedPolicy,
  typename InputIterator,
  typename OutputIterator>
__host__ __device__ OutputIterator
inclusive_scan(const thrust::detail::execution_policy_base< DerivedPolicy >
↪ & exec,
  InputIterator first,
  InputIterator last,
  OutputIterator result);
```

inclusive_scan 计算 inclusive 前缀和运算。inclusive 指的是每个结果在部分和中包含相应的输入操作数。具体而言，将 *first 赋给 *result，将 *first 和 *(first + 1) 之和赋给 *(result + 1)，依此类推。此版本的 inclusive_scan 假设加号为具有结合律的运算符。

当输入和输出序列相同时，就地（in-place）执行扫描。

inclusive_scan 类似于 STL 中的 std::partial_sum。这两个函数之间的主要区别是 std::partial_sum 保证串行求和顺序，而 inclusive_scan 需要二元运算的结合律来并行化前缀和。

对于不满足结合律的运算符的结果是不确定的（例如，浮点类型的加法）。对于不满足结合律的运算符的结果可能因运行而异。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::host 并行化执行策略，使用 inclusive_scan 计算就地前缀和：

```
#include <thrust/scan.h>
#include <thrust/execution_policy.h>
...

int data[6] = {1, 0, 2, 2, 1, 3};

thrust::inclusive_scan(thrust::host, data, data + 6, data); // in-place
↪scan

// data is now {1, 1, 3, 5, 6, 9}
```

模板参数

飞腾信息 MetaX Confidential 2024-11-15 15:00:00

- **DerivedPolicy**：派生执行策略的名称
- **InputIterator**：是 Input Iterator 的模型，InputIterator 的 value_type 可转换为 OutputIterator 的 value_type
- **OutputIterator**：是 Output Iterator 的模型，如果 x 和 y 是 OutputIterator 的 value_type 的对象，则定义 x + y；如果 T 是 OutputIterator 的 value_type ，则定义 T(0)

函数参数

- **exec**：用于并行化的执行策略
- **first**：输入序列的起始
- **last**：输入序列的末尾
- **result**：输出序列的起始

前提条件

first 可以等于 result ，但除此外，范围 [first, last) 不应与范围 [result, result + (last - first)) 重叠

返回值

输出序列的末尾

另请参阅

https://en.cppreference.com/w/cpp/algorithm/partial_sum

thrust::inclusive_scan

```
template <typename InputIterator,
  typename OutputIterator>
OutputIterator
inclusive_scan(InputIterator first,
  InputIterator last,
  OutputIterator result);
```

inclusive_scan 计算 inclusive 前缀和运算。inclusive 指的是每个结果在部分和中包含相应的输入操作数。具体而言，将 *first 赋给 *result ，将 *first 和 *(first + 1) 之和赋给 *(result + 1) ，依此类推。此版本的 inclusive_scan 假设加号为具有结合律的运算符。

当输入和输出序列相同时，就地（in-place）执行扫描。

inclusive_scan 类似于 STL 中的 std::partial_sum。这两个函数之间的主要区别是 std::partial_sum 保证串行求和顺序，而 inclusive_scan 需要二元运算的结合律来并行化前缀和。

对于不满足结合律的运算符的结果是不确定的（例如，浮点类型的加法）。对于不满足结合律的运算符的结果可能因运行而异。

以下代码片段演示了如何使用 inclusive_scan。

```
#include <thrust/scan.h>

int data[6] = {1, 0, 2, 2, 1, 3};

thrust::inclusive_scan(data, data + 6, data); // in-place scan

// data is now {1, 1, 3, 5, 6, 9}
```

模板参数

- **InputIterator**：是 Input Iterator 的模型，InputIterator 的 value_type 可转换为 OutputIterator 的 value_type

沐曦信息 MetaX Confidential 2024-11-8 15:00:00

- **OutputIterator**：是 Output Iterator 的模型，如果 x 和 y 是 OutputIterator 的 value_type 的对象，则定义 x + y；如果 T 是 OutputIterator 的 value_type，则定义 T(0)

函数参数

- **first**：输入序列的起始
- **last**：输入序列的末尾
- **result**：输出序列的起始

前提条件

first 可以等于 result，但除此外，范围 [first, last) 不应与范围 [result, result + (last - first)) 重叠

返回值

输出序列的末尾

另请参阅

https://en.cppreference.com/w/cpp/algorithm/partial_sum

thrust::inclusive_scan

```
template <typename DerivedPolicy,
  typename InputIterator,
  typename OutputIterator,
  typename AssociativeOperator>
__host__ __device__ OutputIterator
thrust::inclusive_scan(const thrust::detail::execution_policy_base<
↪DerivedPolicy > & exec,
  InputIterator first,
  InputIterator last,
  OutputIterator result,
  AssociativeOperator binary_op);
```

inclusive_scan 计算 inclusive 前缀和运算。inclusive 指的是每个结果在部分和中包含相应的输入操作数。当输入和输出序列相同时，就地执行扫描。

inclusive_scan 类似于 STL 中的 std::partial_sum。这两个函数之间的主要区别是 std::partial_sum 保证串行求和顺序，而 inclusive_scan 需要二元运算的结合律来并行化前缀和。

对于不满足结合律的运算符的结果是不确定的（例如，浮点类型的加法）。对于不满足结合律的运算符的结果可能因运行而异。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::host 并行化执行策略，使用 inclusive_scan 计算就地前缀和：

```
int data[10] = {-5, 0, 2, -3, 2, 4, 0, -1, 2, 8};

thrust::maximum<int> binary_op;

thrust::inclusive_scan(thrust::host, data, data + 10, data, binary_op); //
↪in-place scan

// data is now {-5, 0, 2, 2, 2, 4, 4, 4, 4, 8}
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **InputIterator**：是 Input Iterator 的模型，InputIterator 的 value_type 可转换为 OutputIterator 的 value_type

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

- **OutputIterator**：是 Output Iterator 的模型，OutputIterator 的 value_type 可转换为 Associative-Operator 的 first_argument_type 和 second_argument_type
- **AssociativeOperator**：是 Binary Function 的模型，AssociativeOperator 的 result_type 可转换为 OutputIterator 的 value_type

函数参数

- **exec**：用于并行化的执行策略
- **first**：输入序列的起始
- **last**：输入序列的末尾
- **result**：输出序列的起始
- **binary_op**：具有结合律的运算符，用于求和

前提条件

first 可以等于 result ，但除此外，范围 [first, last) 不应与范围 [result, result + (last - first)) 重叠

返回值

输出序列的末尾

另请参阅

https://en.cppreference.com/w/cpp/algorithm/partial_sum

thrust::inclusive_scan

```
template <typename InputIterator,
  typename OutputIterator,
  typename AssociativeOperator>
OutputIterator
thrust::inclusive_scan(InputIterator first,
  InputIterator last,
  OutputIterator result,
  AssociativeOperator binary_op);
```

inclusive_scan 计算 inclusive 前缀和运算。inclusive 指的是每个结果在部分和中包含相应的输入操作数。当输入和输出序列相同时，就地执行扫描。

inclusive_scan 类似于 STL 中的 std::partial_sum。这两个函数之间的主要区别是 std::partial_sum 保证串行求和顺序，而 inclusive_scan 需要二元运算的结合律来并行化前缀和。

对于不满足结合律的运算符的结果是不确定的（例如，浮点类型的加法）。对于不满足结合律的运算符的结果可能因运行而异。

以下代码片段演示了如何使用 inclusive_scan。

```
int data[10] = {-5, 0, 2, -3, 2, 4, 0, -1, 2, 8};

thrust::maximum<int> binary_op;

thrust::inclusive_scan(data, data + 10, data, binary_op); // in-place scan

// data is now {-5, 0, 2, 2, 2, 4, 4, 4, 4, 8}
```

模板参数

- **InputIterator**：是 Input Iterator 的模型，InputIterator 的 value_type 可转换为 OutputIterator 的 value_type

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

- **OutputIterator**：是 Output Iterator 的模型，OutputIterator 的 value_type 可转换为 AssociativeOperator 的 first_argument_type 和 second_argument_type
- **AssociativeOperator**：是 Binary Function 的模型，AssociativeOperator 的 result_type 可转换为 OutputIterator 的 value_type

函数参数

- **first**：输入序列的起始
- **last**：输入序列的末尾
- **result**：输出序列的起始
- **binary_op**：具有结合律的运算符，用于求和

前提条件

first 可以等于 result ，但除此外，范围 [first, last) 不应与范围 [result, result + (last - first)) 重叠

返回值

输出序列的末尾

另请参阅

https://en.cppreference.com/w/cpp/algorithm/partial_sum

thrust::exclusive_scan

```
template <typename DerivedPolicy,
  typename InputIterator,
  typename OutputIterator>
__host__ __device__ OutputIterator
thrust::exclusive_scan(const thrust::detail::execution_policy_base<
↪DerivedPolicy > & exec,
  InputIterator first,
  InputIterator last,
  OutputIterator result);
```

exclusive_scan 计算 exclusive 前缀和运算。exclusive 指的是每个结果在部分和中不包含相应的输入操作数。具体而言，将 0 赋给 *result ，将 0 和 *first 之和赋给 *(result + 1) ，依此类推。此版本的 exclusive_scan 假设加号为具有结合律的运算符，0 为初始值。当输入和输出序列相同时，就地执行扫描。

对于不满足结合律的运算符的结果是不确定的（例如，浮点类型的加法）。对于不满足结合律的运算符的结果可能因运行而异。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::host 并行化执行策略，使用 exclusive_scan 计算就地前缀和：

```
#include <thrust/scan.h>
#include <thrust/execution_policy.h>
...

int data[6] = {1, 0, 2, 2, 1, 3};

thrust::exclusive_scan(thrust::host, data, data + 6, data); // in-place
↪scan

// data is now {0, 1, 1, 3, 5, 6}
```

模板参数

沐曦信息 MetaX Confidential 2024-11-18 15:00:00

- **DerivedPolicy**：派生执行策略的名称
- **InputIterator**：是 Input Iterator 的模型，InputIterator 的 value_type 可转换为 OutputIterator 的 value_type
- **OutputIterator**：是 Output Iterator 的模型，如果 x 和 y 是 OutputIterator 的 value_type 的对象，则定义 x + y ；如果 T 是 OutputIterator 的 value_type ，则定义 T(0)

函数参数

- **exec**：用于并行化的执行策略
- **first**：输入序列的起始
- **last**：输入序列的末尾
- **result**：输出序列的起始

前提条件

first 可以等于 result ，但除此外，范围 [first, last) 不应与范围 [result, result + (last - first)) 重叠

返回值

输出序列的末尾

另请参阅

https://en.cppreference.com/w/cpp/algorithm/partial_sum

thrust::exclusive_scan

```
template <typename InputIterator,
  typename OutputIterator>
OutputIterator
thrust::exclusive_scan(InputIterator first,
  InputIterator last,
  OutputIterator result);
```

exclusive_scan 计算 exclusive 前缀和运算。exclusive 指的是每个结果在部分和中不包含相应的输入操作数。具体而言，将 0 赋给 *result ，将 0 和 *first 之和赋给 *(result + 1) ，依此类推。此版本的 exclusive_scan 假设加号为具有结合律的运算符，0 为初始值。当输入和输出序列相同时，就地执行扫描。

对于不满足结合律的运算符的结果是不确定的（例如，浮点类型的加法）。对于不满足结合律的运算符的结果可能因运行而异。

以下代码片段演示了如何使用 exclusive_scan。

```
#include <thrust/scan.h>

int data[6] = {1, 0, 2, 2, 1, 3};

thrust::exclusive_scan(data, data + 6, data); // in-place scan

// data is now {0, 1, 1, 3, 5, 6}
```

模板参数

- **InputIterator**：是 Input Iterator 的模型，InputIterator 的 value_type 可转换为 OutputIterator 的 value_type
- **OutputIterator**：是 Output Iterator 的模型，如果 x 和 y 是 OutputIterator 的 value_type 的对象，则定义 x + y ；如果 T 是 OutputIterator 的 value_type ，则定义 T(0)

函数参数

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

- **first**：输入序列的起始
- **last**：输入序列的末尾
- **result**：输出序列的起始

前提条件

first 可以等于 result ，但除此外，范围 [first, last) 不应与范围 [result, result + (last - first)) 重叠

返回值

输出序列的末尾

另请参阅

https://en.cppreference.com/w/cpp/algorithm/partial_sum

thrust::exclusive_scan

```
template <typename DerivedPolicy,
  typename InputIterator,
  typename OutputIterator,
  typename T>
__host__ __device__ OutputIterator
thrust::exclusive_scan(const thrust::detail::execution_policy_base<
↪DerivedPolicy > & exec,
  InputIterator first,
  InputIterator last,
  OutputIterator result,
  T init);
```

exclusive_scan 计算 exclusive 前缀和运算。exclusive 指的是每个结果在部分和中不包含相应的输入操作数。具体而言，将 init 赋给 *result ，将 init 和 *first 之和赋给 *(result + 1) ，依此类推。此版本的 exclusive_scan 假设加号为具有结合律的运算符，但需要一个初始值 init。当输入和输出序列相同时，就地执行扫描。

对于不满足结合律的运算符的结果是不确定的（例如，浮点类型的加法）。对于不满足结合律的运算符的结果可能因运行而异。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::host 并行化执行策略，使用 exclusive_scan 计算就地前缀和：

```
#include <thrust/scan.h>
#include <thrust/execution_policy.h>

int data[6] = {1, 0, 2, 2, 1, 3};

thrust::exclusive_scan(thrust::host, data, data + 6, data, 4); // in-place
↪scan

// data is now {4, 5, 5, 7, 9, 10}
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **InputIterator**：是 Input Iterator 的模型，InputIterator 的 value_type 可转换为 OutputIterator 的 value_type
- **OutputIterator**：是 Output Iterator 的模型，如果 x 和 y 是 OutputIterator 的 value_type 的对象，则定义 x + y

飞腾信息 MetaX Confidential 2024-11-18 5:00:00

- **T**：可转换为 OutputIterator 的 value_type

函数参数

- **exec**：用于并行化的执行策略
- **first**：输入序列的起始
- **last**：输入序列的末尾
- **result**：输出序列的起始
- **init**：初始值

前提条件

first 可以等于 result ，但除此外，范围 [first, last) 不应与范围 [result, result + (last - first)) 重叠

返回值

输出序列的末尾

另请参阅

https://en.cppreference.com/w/cpp/algorithm/partial_sum

thrust::exclusive_scan

```
template <typename InputIterator,
  typename OutputIterator,
  typename T>
OutputIterator
thrust::exclusive_scan(InputIterator first,
  InputIterator last,
  OutputIterator result,
  T init);
```

exclusive_scan 计算 exclusive 前缀和运算。exclusive 指的是每个结果在部分和中不包含相应的输入操作数。具体而言，将 init 赋给 *result ，将 init 和 *first 之和赋给 *(result + 1) ，依此类推。此版本的 exclusive_scan 假设加号为具有结合律的运算符，但需要一个初始值 init。当输入和输出序列相同时，就地执行扫描。

对于不满足结合律的运算符的结果是不确定的（例如，浮点类型的加法）。对于不满足结合律的运算符的结果可能因运行而异。

以下代码片段演示了如何使用 exclusive_scan。

```
#include <thrust/scan.h>

int data[6] = {1, 0, 2, 2, 1, 3};

thrust::exclusive_scan(data, data + 6, data, 4); // in-place scan

// data is now {4, 5, 5, 7, 9, 10}
```

模板参数

- **InputIterator**：是 Input Iterator 的模型，InputIterator 的 value_type 可转换为 OutputIterator 的 value_type
- **OutputIterator**：是 Output Iterator 的模型，如果 x 和 y 是 OutputIterator 的 value_type 的对象，则定义 x + y
- **T**：可转换为 OutputIterator 的 value_type

函数参数

沐曦信息 MetaX Confidential 2024-11-18 15:00:00

- **first**：输入序列的起始
- **last**：输入序列的末尾
- **result**：输出序列的起始
- **init**：初始值

前提条件

first 可以等于 result ，但除此外，范围 [first, last) 不应与范围 [result, result + (last - first)) 重叠

返回值

输出序列的末尾

另请参阅

https://en.cppreference.com/w/cpp/algorithm/partial_sum

thrust::exclusive_scan

```
template <typename DerivedPolicy,
  typename InputIterator,
  typename OutputIterator,
  typename T,
  typename AssociativeOperator>
__host__ __device__ OutputIterator
thrust::exclusive_scan(const thrust::detail::execution_policy_base<
↪DerivedPolicy > & exec,
  InputIterator first,
  InputIterator last,
  OutputIterator result,
  T init,
  AssociativeOperator binary_op);
```

exclusive_scan 计算 exclusive 前缀和运算。exclusive 指的是每个结果在部分和中不包含相应的输入操作数。具体而言，将 init 赋给 *result ，将值 binary_op(init, *first) 赋给 *(result + 1) ，依此类推。此版本的函数需要具有结合律的运算符和初始值 init。当输入和输出序列相同时，就地执行扫描。

对于不满足结合律的运算符的结果是不确定的（例如，浮点类型的加法）。对于不满足结合律的运算符的结果可能因运行而异。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::host 并行化执行策略，使用 exclusive_scan 计算就地前缀和：

```
#include <thrust/scan.h>
#include <thrust/functional.h>
#include <thrust/execution_policy.h>
...

int data[10] = {-5, 0, 2, -3, 2, 4, 0, -1, 2, 8};

thrust::maximum<int> binary_op;

thrust::exclusive_scan(thrust::host, data, data + 10, data, 1, binary_op);
↪// in-place scan

// data is now {1, 1, 1, 2, 2, 2, 4, 4, 4, 4 }
```

模板参数

飞腾信息 MetaX Confidential 2024-11-12 15:00:00

- **DerivedPolicy**：派生执行策略的名称
- **InputIterator**：是 Input Iterator 的模型，InputIterator 的 value_type 可转换为 OutputIterator 的 value_type
- **OutputIterator**：是 Output Iterator 的模型，OutputIterator 的 value_type 可转换为 Associative-Operator 的 first_argument_type 和 second_argument_type
- **T**：可转换为 OutputIterator 的 value_type
- **AssociativeOperator**：是 Binary Function 的模型，AssociativeOperator 的 result_type 可转换为 OutputIterator 的 value_type

函数参数

- **exec**：用于并行化的执行策略
- **first**：输入序列的起始
- **last**：输入序列的末尾
- **result**：输出序列的起始
- **init**：初始值
- **binary_op**：具有结合律的运算符，用于求和

前提条件

first 可以等于 result ，但除此外，范围 [first, last) 不应与范围 [result, result + (last - first)) 重叠

返回值

输出序列的末尾

另请参阅

https://en.cppreference.com/w/cpp/algorithm/partial_sum

thrust::exclusive_scan

```
template <typename InputIterator,
  typename OutputIterator,
  typename T,
  typename AssociativeOperator>
OutputIterator
thrust::exclusive_scan(InputIterator first,
  InputIterator last,
  OutputIterator result,
  T init,
  AssociativeOperator binary_op);
```

exclusive_scan 计算 exclusive 前缀和运算。exclusive 指的是每个结果在部分和中不包含相应的输入操作数。具体而言，将 init 赋给 *result ，将值 binary_op(init, *first) 赋给 *(result + 1) ，依此类推。此版本的函数需要具有结合律的运算符和初始值 init。当输入和输出序列相同时，就地执行扫描。

对于不满足结合律的运算符的结果是不确定的（例如，浮点类型的加法）。对于不满足结合律的运算符的结果可能因运行而异。

以下代码片段演示了如何使用 exclusive_scan。

```
#include <thrust/scan.h>
#include <thrust/functional.h>

int data[10] = {-5, 0, 2, -3, 2, 4, 0, -1, 2, 8};
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
thrust::maximum<int> binary_op;

thrust::exclusive_scan(data, data + 10, data, 1, binary_op); // in-place
↪scan

// data is now {1, 1, 1, 2, 2, 2, 4, 4, 4, 4 }
```

模板参数

- **InputIterator**：是 Input Iterator 的模型，InputIterator 的 value_type 可转换为 OutputIterator 的 value_type
- **OutputIterator**：是 Output Iterator 的模型，OutputIterator 的 value_type 可转换为 AssociativeOperator 的 first_argument_type 和 second_argument_type
- **T**：可转换为 OutputIterator 的 value_type
- **AssociativeOperator**：是 Binary Function 的模型，AssociativeOperator 的 result_type 可转换为 OutputIterator 的 value_type

函数参数

- **first**：输入序列的起始
- **last**：输入序列的末尾
- **result**：输出序列的起始
- **init**：初始值
- **binary_op**：具有结合律的运算符，用于求和

前提条件

first 可以等于 result，但除此外，范围 [first, last) 不应与范围 [result, result + (last - first)) 重叠

返回值

输出序列的末尾

另请参阅

https://en.cppreference.com/w/cpp/algorithm/partial_sum

2.1.3.2 分段前缀和（Segmented Prefix Sums）

函数

```
template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator>
__host__ __device__ OutputIterator
thrust::inclusive_scan_by_key(const thrust::detail::execution_policy_base<
↪DerivedPolicy > & exec,
  InputIterator1 first1,
  InputIterator1 last1,
  InputIterator2 first2,
  OutputIterator result);

template <typename InputIterator1,
  typename InputIterator2,
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
  typename OutputIterator>
OutputIterator
thrust::inclusive_scan_by_key(InputIterator1 first1,
  InputIterator1 last1,
  InputIterator2 first2,
  OutputIterator result);

template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator,
  typename BinaryPredicate>
__host__ __device__ OutputIterator
thrust::inclusive_scan_by_key(const thrust::detail::execution_policy_base<
↪DerivedPolicy > & exec,
  InputIterator1 first1,
  InputIterator1 last1,
  InputIterator2 first2,
  OutputIterator result,
  BinaryPredicate binary_pred);

template <typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator,
  typename BinaryPredicate>
OutputIterator
thrust::inclusive_scan_by_key(InputIterator1 first1,
  InputIterator1 last1,
  InputIterator2 first2,
  OutputIterator result,
  BinaryPredicate binary_pred);

template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator,
  typename BinaryPredicate,
  typename AssociativeOperator>
__host__ __device__ OutputIterator
thrust::inclusive_scan_by_key(const thrust::detail::execution_policy_base<
↪DerivedPolicy > & exec,
  InputIterator1 first1,
  InputIterator1 last1,
  InputIterator2 first2,
  OutputIterator result,
  BinaryPredicate binary_pred,
  AssociativeOperator binary_op);

template <typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator,
  typename BinaryPredicate,
  typename AssociativeOperator>
OutputIterator
thrust::inclusive_scan_by_key(InputIterator1 first1,
  InputIterator1 last1,
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
  InputIterator2 first2,
  OutputIterator result,
  BinaryPredicate binary_pred,
  AssociativeOperator binary_op);

template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator>
__host__ __device__ OutputIterator
thrust::exclusive_scan_by_key(const thrust::detail::execution_policy_base<
↪DerivedPolicy > & exec,
  InputIterator1 first1,
  InputIterator1 last1,
  InputIterator2 first2,
  OutputIterator result);
template <typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator> OutputIterator
thrust::exclusive_scan_by_key(InputIterator1 first1,
  InputIterator1 last1,
  InputIterator2 first2,
  OutputIterator result);

template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator,
  typename T>
__host__ __device__ OutputIterator
thrust::exclusive_scan_by_key(const thrust::detail::execution_policy_base<
↪DerivedPolicy > & exec,
  InputIterator1 first1,
  InputIterator1 last1,
  InputIterator2 first2,
  OutputIterator result,
  T init);

template <typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator,
  typename T>
OutputIterator
thrust::exclusive_scan_by_key(InputIterator1 first1,
  InputIterator1 last1,
  InputIterator2 first2,
  OutputIterator result,
  T init);

template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator,
  typename T,
  typename BinaryPredicate>
__host__ __device__ OutputIterator
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
thrust::exclusive_scan_by_key(const thrust::detail::execution_policy_base<
↪DerivedPolicy > & exec,
  InputIterator1 first1,
  InputIterator1 last1,
  InputIterator2 first2,
  OutputIterator result,
  T init,
  BinaryPredicate binary_pred);

template <typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator,
  typename T,
  typename BinaryPredicate>
OutputIterator
thrust::exclusive_scan_by_key(InputIterator1 first1,
  InputIterator1 last1,
  InputIterator2 first2,
  OutputIterator result,
  T init,
  BinaryPredicate binary_pred);

template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator,
  typename T,
  typename BinaryPredicate,
  typename AssociativeOperator>
__host__ __device__ OutputIterator
thrust::exclusive_scan_by_key(const thrust::detail::execution_policy_base<
↪DerivedPolicy > & exec,
  InputIterator1 first1,
  InputIterator1 last1,
  InputIterator2 first2,
  OutputIterator result,
  T init,
  BinaryPredicate binary_pred,
  AssociativeOperator binary_op);

template <typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator,
  typename T,
  typename BinaryPredicate,
  typename AssociativeOperator>
OutputIterator
thrust::exclusive_scan_by_key(InputIterator1 first1,
  InputIterator1 last1,
  InputIterator2 first2,
  OutputIterator result,
  T init,
  BinaryPredicate binary_pred,
  AssociativeOperator binary_op);
```

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

thrust::inclusive_scan_by_key

```
template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator>
__host__ __device__ OutputIterator
inclusive_scan_by_key(const thrust::detail::execution_policy_base<
 ↪DerivedPolicy > & exec,
  InputIterator1 first1,
  InputIterator1 last1,
  InputIterator2 first2,
  OutputIterator result);
```

inclusive_scan_by_key 计算 inclusive 键值或 segmented 前缀和运算。inclusive 指的是每个结果在部分和中包含相应的输入操作数。segmented 指的是部分和被分成不同的段。即，在每个段内计算单独的 inclusive 扫描运算。使用示例，请参见以下代码示例。

此版本的 inclusive_scan_by_key 假设 equal_to 为用于比较相邻键的二元谓词（binary predicate)。具体而言，如果 *i == *(i+1)，则范围 [first1, last1) 内的连续迭代器 i 和 i+1 属于同一段；否则属于不同的段。

此版本的 inclusive_scan_by_key 假设 plus 为具有结合律的运算符，用于执行前缀和。当输入和输出序列相同时，就地执行扫描。

对于不满足结合律的运算符的结果是不确定的（例如，浮点类型的加法)。对于不满足结合律的运算符的结果可能因运行而异。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::host 并行化执行策略，使用 inclusive_scan_by_key：

```
#include <thrust/scan.h>
#include <thrust/execution_policy.h>
...

int data[10] = {1, 1, 1, 1, 1, 1, 1, 1, 1, 1};
int keys[10] = {0, 0, 0, 1, 1, 2, 3, 3, 3, 3};

thrust::inclusive_scan_by_key(thrust::host, keys, keys + 10, data, data); /
 ↪/ in-place scan

// data is now {1, 2, 3, 1, 2, 1, 1, 2, 3, 4};
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **InputIterator1**：是 Input Iterator 的模型
- **InputIterator2**：是 Input Iterator 的模型，InputIterator2 的 value_type 可转换为 OutputIterator 的 value_type
- **OutputIterator**：是 Output Iterator 的模型，如果 x 和 y 是 OutputIterator 的 value_type 的对象，则定义 binary_op(x,y)

函数参数

- **exec**：用于并行化的执行策略
- **first1**：键序列的起始
- **last1**：键序列的末尾
- **first2**：输入值的序列起始

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

- **result**：输出值的序列起始

前提条件

- first1 可以等于 result，但除此外，范围 [first1, last1) 不应与范围 [result, result + (last1 - first1)) 重叠
- first2 可以等于 result，但除此外，范围 [first2, first2 + (last1 - first1) 不应与范围 [result, result + (last1 - first1)) 重叠

返回值

输出序列的末尾

另请参阅

- inclusive_scan
- exclusive_scan_by_key

thrust::inclusive_scan_by_key

```
template <typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator>
OutputIterator
inclusive_scan_by_key(InputIterator1 first1,
  InputIterator1 last1,
  InputIterator2 first2,
  OutputIterator result);
```

inclusive_scan_by_key 计算 inclusive 键值或 segmented 前缀和运算。inclusive 指的是每个结果在部分和中包含相应的输入操作数。segmented 指的是部分和被分成不同的段。即，在每个段内计算单独的 inclusive 扫描运算。使用示例，请参见以下代码示例。

此版本的 inclusive_scan_by_key 假设 equal_to 为用于比较相邻键的二元谓词。具体而言，如果 *i == *(i+1)，则范围 [first1, last1) 内的连续迭代器 i 和 i+1 属于同一段；否则属于不同的段。

此版本的 inclusive_scan_by_key 假设 plus 为具有结合律的运算符，用于执行前缀和。当输入和输出序列相同时，就地执行扫描。

对于不满足结合律的运算符的结果是不确定的（例如，浮点类型的加法）。对于不满足结合律的运算符的结果可能因运行而异。

以下代码片段演示了如何使用 inclusive_scan_by_key。

```
#include <thrust/scan.h>

int data[10] = {1, 1, 1, 1, 1, 1, 1, 1, 1, 1};
int keys[10] = {0, 0, 0, 1, 1, 2, 3, 3, 3, 3};

thrust::inclusive_scan_by_key(keys, keys + 10, data, data); // in-place
↪scan

// data is now {1, 2, 3, 1, 2, 1, 1, 2, 3, 4};
```

模板参数

- **InputIterator1**：是 Input Iterator 的模型
- **InputIterator2**：是 Input Iterator 的模型，InputIterator2 的 value_type 可转换为 OutputIterator 的 value_type
- **OutputIterator**：是 Output Iterator 的模型，如果 x 和 y 是 OutputIterator 的 value_type 的对象，则定义 binary_op(x,y)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

函数参数

- **first1**：键序列的起始
- **last1**：键序列的末尾
- **first2**：输入值的序列起始
- **result**：输出值的序列起始

前提条件

- first1 可以等于 result，但除此外，范围 [first1, last1) 不应与范围 [result, result + (last1 - first1)) 重叠
- first2 可以等于 result，但除此外，范围 [first2, first2 + (last1 - first1) 不应与范围 [result, result + (last1 - first1)) 重叠

返回值

输出序列的末尾

另请参阅

- inclusive_scan
- exclusive_scan_by_key

thrust::inclusive_scan_by_key

```
template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator,
  typename BinaryPredicate>
__host__ __device__ OutputIterator
inclusive_scan_by_key(const thrust::detail::execution_policy_base<
 ↪DerivedPolicy > & exec,
  InputIterator1 first1,
  InputIterator1 last1,
  InputIterator2 first2,
  OutputIterator result,
  BinaryPredicate binary_pred);
```

inclusive_scan_by_key 计算 inclusive 键值或 segmented 前缀和运算。inclusive 指的是每个结果在部分和中包含相应的输入操作数。segmented 指的是部分和被分成不同的段。即，在每个段内计算单独的 inclusive 扫描运算。使用示例，请参见以下代码示例。

此版本的 inclusive_scan_by_key 使用二元谓词 pred 比较相邻键。具体而言，如果 binary_pred(*i, *(i+1)) 为 true，则范围 [first1, last1) 内的连续迭代器 i 和 i+1 属于同一段；否则属于不同的段。

此版本的 inclusive_scan_by_key 假设 plus 为具有结合律的运算符，用于执行前缀和。当输入和输出序列相同时，就地执行扫描。

对于不满足结合律的运算符的结果是不确定的（例如，浮点类型的加法）。对于不满足结合律的运算符的结果可能因运行而异。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::host 并行化执行策略，使用 inclusive_scan_by_key：

```
#include <thrust/scan.h>
#include <thrust/functional.h>
#include <thrust/execution_policy.h>
...
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
int data[10] = {1, 1, 1, 1, 1, 1, 1, 1, 1, 1};
int keys[10] = {0, 0, 0, 1, 1, 2, 3, 3, 3, 3};

thrust::equal_to<int> binary_pred;

thrust::inclusive_scan_by_key(thrust::host, keys, keys + 10, data, data,
↪binary_pred); // in-place scan

// data is now {1, 2, 3, 1, 2, 1, 1, 2, 3, 4};
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **InputIterator1**：是 Input Iterator 的模型
- **InputIterator2**：是 Input Iterator 的模型，InputIterator2 的 value_type 可转换为 OutputIterator 的 value_type
- **OutputIterator**：是 Output Iterator 的模型，如果 x 和 y 是 OutputIterator 的 value_type 的对象，则定义 binary_op(x,y)
- **BinaryPredicate**：是 Binary Predicate 的模型

函数参数

- **exec**：用于并行化的执行策略
- **first1**：键序列的起始
- **last1**：键序列的末尾
- **first2**：输入值的序列起始
- **result**：输出值的序列起始
- **binary_pred**：用于确定键相等性的二元谓词

前提条件

- first1 可以等于 result，但除此外，范围 [first1, last1) 不应与范围 [result, result + (last1 - first1)) 重叠
- first2 可以等于 result，但除此外，范围 [first2, first2 + (last1 - first1) 不应与范围 [result, result + (last1 - first1)) 重叠

返回值

输出序列的末尾

另请参阅

- inclusive_scan
- exclusive_scan_by_key

thrust::inclusive_scan_by_key

```
template <typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator,
  typename BinaryPredicate>
OutputIterator
inclusive_scan_by_key(InputIterator1 first1,
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
  InputIterator1 last1,
  InputIterator2 first2,
  OutputIterator result,
  BinaryPredicate binary_pred);
```

inclusive_scan_by_key 计算 inclusive 键值或 segmented 前缀和运算。inclusive 指的是每个结果在部分和中包含相应的输入操作数。segmented 指的是部分和被分成不同的段。即，在每个段内计算单独的 inclusive 扫描运算。使用示例，请参见以下代码示例。

此版本的 inclusive_scan_by_key 使用二元谓词 pred 比较相邻键。具体而言，如果 binary_pred(*i, *(i+1)) 为 true，则范围 [first1, last1) 内的连续迭代器 i 和 i+1 属于同一段；否则属于不同的段。

此版本的 inclusive_scan_by_key 假设 plus 为具有结合律的运算符，用于执行前缀和。当输入和输出序列相同时，就地执行扫描。

对于不满足结合律的运算符的结果是不确定的（例如，浮点类型的加法）。对于不满足结合律的运算符的结果可能因运行而异。

以下代码片段演示了如何使用 inclusive_scan_by_key。

```
#include <thrust/scan.h>
#include <thrust/functional.h>

int data[10] = {1, 1, 1, 1, 1, 1, 1, 1, 1, 1};
int keys[10] = {0, 0, 0, 1, 1, 2, 3, 3, 3, 3};

thrust::equal_to<int> binary_pred;

thrust::inclusive_scan_by_key(keys, keys + 10, data, data, binary_pred); //
↪ in-place scan

// data is now {1, 2, 3, 1, 2, 1, 1, 2, 3, 4};
```

模板参数

- **InputIterator1**：是 Input Iterator 的模型
- **InputIterator2**：是 Input Iterator 的模型，InputIterator2 的 value_type 可转换为 OutputIterator 的 value_type
- **OutputIterator**：是 Output Iterator 的模型，如果 x 和 y 是 OutputIterator 的 value_type 的对象，则定义 binary_op(x,y)
- **BinaryPredicate**：是 Binary Predicate 的模型

函数参数

- **first1**：键序列的起始
- **last1**：键序列的末尾
- **first2**：输入值的序列起始
- **result**：输出值的序列起始
- **binary_pred**：用于确定键相等性的二元谓词

前提条件

- first1 可以等于 result ，但除此外，范围 [first1, last1) 不应与范围 [result, result + (last1 - first1)) 重叠
- first2 可以等于 result ，但除此外，范围 [first2, first2 + (last1 - first1) 不应与范围 [result, result + (last1 - first1)) 重叠

返回值

输出序列的末尾

飞猪信息 MetaX Confidential 2024-11-18 15:00:00

另请参阅

- inclusive_scan
- exclusive_scan_by_key

thrust::inclusive_scan_by_key

```
template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator,
  typename BinaryPredicate,
  typename AssociativeOperator>
__host__ __device__ OutputIterator
inclusive_scan_by_key(const thrust::detail::execution_policy_base<
 ↪DerivedPolicy > & exec,
  InputIterator1 first1,
  InputIterator1 last1,
  InputIterator2 first2,
  OutputIterator result,
  BinaryPredicate binary_pred,
  AssociativeOperator binary_op);
```

inclusive_scan_by_key 计算 inclusive 键值或 segmented 前缀和运算。inclusive 指的是每个结果在部分和中包含相应的输入操作数。segmented 指的是部分和被分成不同的段。

即，在每个段内计算单独的 inclusive 扫描运算。使用示例，请参见以下代码示例。

此版本的 inclusive_scan_by_key 使用二元谓词 pred 比较相邻键。具体而言，如果 binary_pred(*i, *(i+1)) 为 true，则范围 [first1, last1) 内的连续迭代器 i 和 i+1 属于同一段；否则属于不同的段。

此版本的 inclusive_scan_by_key 使用具有结合律的运算符 binary_op 执行前缀和运算。当输入和输出序列相同时，就地执行扫描。

对于不满足结合律的运算符的结果是不确定的（例如，浮点类型的加法）。对于不满足结合律的运算符的结果可能因运行而异。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::host 并行化执行策略，使用 inclusive_scan_by_key：

```
#include <thrust/scan.h>
#include <thrust/functional.h>
#include <thrust/execution_policy.h>
...

int data[10] = {1, 1, 1, 1, 1, 1, 1, 1, 1, 1};
int keys[10] = {0, 0, 0, 1, 1, 2, 3, 3, 3, 3};

thrust::equal_to<int> binary_pred;
thrust::plus<int>     binary_op;

thrust::inclusive_scan_by_key(thrust::host, keys, keys + 10, data, data,
 ↪binary_pred, binary_op); // in-place scan

// data is now {1, 2, 3, 1, 2, 1, 1, 2, 3, 4};
```

模板参数

- **DerivedPolicy**：派生执行策略的名称

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

- **InputIterator1**：是 Input Iterator 的模型
- **InputIterator2**：是 Input Iterator 的模型，InputIterator2 的 value_type 可转换为 OutputIterator 的 value_type
- **OutputIterator**：是 Output Iterator 的模型，如果 x 和 y 是 OutputIterator 的 value_type 的对象，则定义 binary_op(x,y)
- **BinaryPredicate**：是 Binary Predicate 的模型
- **AssociativeOperator**：是 Binary Function 的模型，AssociativeOperator 的 result_type 可转换为 OutputIterator 的 value_type

函数参数

- **exec**：用于并行化的执行策略
- **first1**：键序列的起始
- **last1**：键序列的末尾
- **first2**：输入值的序列起始
- **result**：输出值的序列起始
- **binary_pred**：用于确定键相等性的二元谓词
- **binary_op**：具有结合律的运算符，用于求和

前提条件

- first1 可以等于 result ，但除此外，范围 [first1, last1) 不应与范围 [result, result + (last1 - first1)) 重叠
- first2 可以等于 result ，但除此外，范围 [first2, first2 + (last1 - first1) 不应与范围 [result, result + (last1 - first1)) 重叠

返回值

输出序列的末尾

另请参阅

- inclusive_scan
- exclusive_scan_by_key

thrust::inclusive_scan_by_key

```
template <typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator,
  typename BinaryPredicate,
  typename AssociativeOperator>
OutputIterator
inclusive_scan_by_key(InputIterator1 first1,
  InputIterator1 last1,
  InputIterator2 first2,
  OutputIterator result,
  BinaryPredicate binary_pred,
  AssociativeOperator binary_op);
```

inclusive_scan_by_key 计算 inclusive 键值或 segmented 前缀和运算。inclusive 指的是每个结果在部分和中包含相应的输入操作数。segmented 指的是部分和被分成不同的段。即，在每个段内计算单独的 inclusive 扫描运算。使用示例，请参见以下代码示例。

此版本的 inclusive_scan_by_key 使用二元谓词 pred 比较相邻键。具体而言，如果 binary_pred(*i, *(i+1)) 为 true，则范围 [first1, last1) 内的连续迭代器 i 和 i+1 属于同一段；否则属于不同的段。

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

对于不满足结合律的运算符的结果是不确定的（例如，浮点类型的加法）。对于不满足结合律的运算符的结果可能因运行而异。

此版本的 inclusive_scan_by_key 使用具有结合律的运算符 binary_op 执行前缀和运算。当输入和输出序列相同时，就地执行扫描。

以下代码片段演示了如何使用 inclusive_scan_by_key。

```
#include <thrust/scan.h>
#include <thrust/functional.h>

int data[10] = {1, 1, 1, 1, 1, 1, 1, 1, 1, 1};
int keys[10] = {0, 0, 0, 1, 1, 2, 3, 3, 3, 3};

thrust::equal_to<int> binary_pred;
thrust::plus<int>     binary_op;

thrust::inclusive_scan_by_key(keys, keys + 10, data, data, binary_pred,
↪binary_op); // in-place scan

// data is now {1, 2, 3, 1, 2, 1, 1, 2, 3, 4};
```

模板参数

- **InputIterator1**：是 Input Iterator 的模型
- **InputIterator2**：是 Input Iterator 的模型，InputIterator2 的 value_type 可转换为 OutputIterator 的 value_type
- **OutputIterator**：是 Output Iterator 的模型，如果 x 和 y 是 OutputIterator 的 value_type 的对象，则定义 binary_op(x,y)
- **BinaryPredicate**：是 Binary Predicate 的模型
- **AssociativeOperator**：是 Binary Function 的模型，AssociativeOperator 的 result_type 可转换为 OutputIterator 的 value_type

函数参数

- **first1**：键序列的起始
- **last1**：键序列的末尾
- **first2**：输入值的序列起始
- **result**：输出值的序列起始
- **binary_pred**：用于确定键相等性的二元谓词
- **binary_op**：具有结合律的运算符，用于求和

前提条件

- first1 可以等于 result，但除此外，范围 [first1, last1) 不应与范围 [result, result + (last1 - first1)) 重叠
- first2 可以等于 result，但除此外，范围 [first2, first2 + (last1 - first1) 不应与范围 [result, result + (last1 - first1)) 重叠

返回值

输出序列的末尾

另请参阅

- inclusive_scan
- exclusive_scan_by_key

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

thrust::exclusive_scan_by_key

```
template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator>
__host__ __device__ OutputIterator
exclusive_scan_by_key(const thrust::detail::execution_policy_base<
 ↪DerivedPolicy > & exec,
  InputIterator1 first1,
  InputIterator1 last1,
  InputIterator2 first2,
  OutputIterator result);
```

exclusive_scan_by_key 计算 exclusive 分段前缀和。

此版本的 exclusive_scan_by_key 使用值 0 初始化 exclusive 扫描运算。

此版本的 exclusive_scan_by_key 假设 plus 为具有结合律的运算符，用于执行前缀和。当输入和输出序列相同时，就地执行扫描。

此版本的 exclusive_scan_by_key 假设 equal_to 为用于比较相邻键的二元谓词。具体而言，如果 *i == *(i+1)，则范围 [first1, last1 内的连续迭代器 i 和 i+1 属于同一段；否则属于不同的段。

对于不满足结合律的运算符的结果是不确定的（例如，浮点类型的加法）。对于不满足结合律的运算符的结果可能因运行而异。

更多详细信息，请参见 exclusive_scan_by_key 的最常见形式。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::host 并行化执行策略，使用 exclusive_scan_by_key：

```
#include <thrust/scan.h>
#include <thrust/execution_policy.h>
...

int keys[10] = {0, 0, 0, 1, 1, 2, 3, 3, 3, 3};
int vals[10] = {1, 1, 1, 1, 1, 1, 1, 1, 1, 1};

thrust::exclusive_scan_by_key(thrust::host, key, key + 10, vals, vals); //
 ↪in-place scan

// vals is now {0, 1, 2, 0, 1, 0, 0, 1, 2, 3};
```

函数参数

- **exec**：用于并行化的执行策略
- **first1**：键序列的起始
- **last1**：键序列的末尾
- **first2**：输入值的序列起始
- **result**：输出值的序列起始

前提条件

- first1 可以等于 result，但除此外，范围 [first1, last1) 不应与范围 [result, result + (last1 - first1)) 重叠
- first2 可以等于 result，但除此外，范围 [first2, first2 + (last1 - first1) 不应与范围 [result, result + (last1 - first1)) 重叠

另请参阅

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

exclusive_scan

thrust::exclusive_scan_by_key

```
template <typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator>
OutputIterator
exclusive_scan_by_key(InputIterator1 first1,
  InputIterator1 last1,
  InputIterator2 first2,
  OutputIterator result);
```

exclusive_scan_by_key 计算 exclusive 分段前缀和。

此版本的 exclusive_scan_by_key 使用值 0 初始化 exclusive 扫描运算。

此版本的 exclusive_scan_by_key 假设 plus 为具有结合律的运算符，用于执行前缀和。当输入和输出序列相同时，就地执行扫描。

此版本的 exclusive_scan_by_key 假设 equal_to 为用于比较相邻键的二元谓词。具体而言，如果 *i == *(i+1)，则范围 [first1, last1) 内的连续迭代器 i 和 i+1 属于同一段；否则属于不同的段。

对于不满足结合律的运算符的结果是不确定的（例如，浮点类型的加法）。对于不满足结合律的运算符的结果可能因运行而异。

更多详细信息，请参见 exclusive_scan_by_key 的最常见形式。

以下代码片段演示了如何使用 exclusive_scan_by_key。.

```
#include <thrust/scan.h>

int keys[10] = {0, 0, 0, 1, 1, 2, 3, 3, 3, 3};
int vals[10] = {1, 1, 1, 1, 1, 1, 1, 1, 1, 1};

thrust::exclusive_scan_by_key(key, key + 10, vals, vals); // in-place scan

// vals is now {0, 1, 2, 0, 1, 0, 0, 1, 2, 3};
```

函数参数

- **first1**：键序列的起始
- **last1**：键序列的末尾
- **first2**：输入值的序列起始
- **result**：输出值的序列起始

前提条件

- first1 可以等于 result，但除此外，范围 [first1, last1) 不应与范围 [result, result + (last1 - first1)) 重叠
- first2 可以等于 result，但除此外，范围 [first2, first2 + (last1 - first1) 不应与范围 [result, result + (last1 - first1)) 重叠

另请参阅

exclusive_scan

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

thrust::exclusive_scan_by_key

```
template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator,
  typename T>
__host__ __device__ OutputIterator
exclusive_scan_by_key(const thrust::detail::execution_policy_base<
↪DerivedPolicy > & exec,
  InputIterator1 first1,
  InputIterator1 last1,
  InputIterator2 first2,
  OutputIterator result,
  T init);
```

exclusive_scan_by_key 计算 excexclusive 键值或 segmented 前缀和运算。exclusive 指的是每个结果在部分和中不包含相应的输入操作数。segmented 指的是部分和被分成不同的段。即，在每个段内计算单独的 exclusive 扫描运算。使用示例，请参见以下代码示例。

此版本的 exclusive_scan_by_key 使用值 init 初始化 exclusive 扫描运算。

对于不满足结合律的运算符的结果是不确定的（例如，浮点类型的加法）。对于不满足结合律的运算符的结果可能因运行而异。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::host 并行化执行策略，使用 exclusive_scan_by_key：

```
#include <thrust/scan.h>
#include <thrust/functional.h>
#include <thrust/execution_policy.h>
...

int keys[10] = {0, 0, 0, 1, 1, 2, 3, 3, 3, 3};
int vals[10] = {1, 1, 1, 1, 1, 1, 1, 1, 1, 1};

int init = 5;

thrust::exclusive_scan_by_key(thrust::host, key, key + 10, vals, vals,
↪init); // in-place scan

// vals is now {5, 6, 7, 5, 6, 5, 5, 6, 7, 8};
```

函数参数

- **exec**：用于并行化的执行策略
- **first1**：键序列的起始
- **last1**：键序列的末尾
- **first2**：输入值的序列起始
- **result**：输出值的序列起始
- **init**：exclusive 和的初始值

前提条件

- first1 可以等于 result ，但除此外，范围 [first1, last1) 不应与范围 [result, result + (last1 - first1)) 重叠
- first2 可以等于 result ，但除此外，范围 [first2, first2 + (last1 - first1) 不应与范围 [result, result + (last1 - first1)) 重叠

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

返回值

输出序列的末尾

另请参阅

- exclusive_scan
- inclusive_scan_by_key

thrust::exclusive_scan_by_key

```
template <typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator,
  typename T>
OutputIterator
exclusive_scan_by_key(InputIterator1 first1,
  InputIterator1 last1,
  InputIterator2 first2,
  OutputIterator result,
  T init);
```

exclusive_scan_by_key 计算 exclusive 键值或 segmented 前缀和运算。exclusive 指的是每个结果在部分和中不包含相应的输入操作数。segmented 指的是部分和被分成不同的段。即，在每个段内计算单独的 exclusive 扫描运算。使用示例，请参见以下代码示例。

此版本的 exclusive_scan_by_key 使用值 init 初始化 exclusive 扫描运算。

对于不满足结合律的运算符的结果是不确定的（例如，浮点类型的加法）。对于不满足结合律的运算符的结果可能因运行而异。

以下代码片段演示了如何使用 exclusive_scan_by_key。

```
#include <thrust/scan.h>
#include <thrust/functional.h>

int keys[10] = {0, 0, 0, 1, 1, 2, 3, 3, 3, 3};
int vals[10] = {1, 1, 1, 1, 1, 1, 1, 1, 1, 1};

int init = 5;

thrust::exclusive_scan_by_key(key, key + 10, vals, vals, init); // in-
↪place scan

// vals is now {5, 6, 7, 5, 6, 5, 5, 6, 7, 8};
```

函数参数

- **first1**：键序列的起始
- **last1**：键序列的末尾
- **first2**：输入值的序列起始
- **result**：输出值的序列起始
- **init**：exclusive 和的初始值

前提条件

- first1 可以等于 result ，但除此外，范围 [first1, last1) 不应与范围 [result, result + (last1 - first1)) 重叠

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

- first2 可以等于 result ，但除此外，范围 [first2, first2 + (last1 - first1) 不应与范围 [result, result + (last1 - first1)) 重叠

返回值

输出序列的末尾

另请参阅

- exclusive_scan
- inclusive_scan_by_key

thrust::exclusive_scan_by_key

```
template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator,
  typename T,
  typename BinaryPredicate>
 __host__ __device__
 OutputIterator exclusive_scan_by_key(const thrust::detail::execution_
↪policy_base< DerivedPolicy > & exec,
   InputIterator1 first1,
   InputIterator1 last1,
    InputIterator2 first2,
    OutputIterator result,
    T init,
    BinaryPredicate binary_pred);
```

exclusive_scan_by_key 计算 exclusive 键值或 segmented 前缀和运算。exclusive 指的是每个结果在部分和中不包含相应的输入操作数。segmented 指的是部分和被分成不同的段。即，在每个段内计算单独的 exclusive 扫描运算。使用示例，请参见以下代码示例。

此版本的 exclusive_scan_by_key 使用值 init 初始化 exclusive 扫描运算。

此版本的 exclusive_scan_by_key 使用二元谓词 binary_pred 比较相邻键。具体而言，如果 binary_pred(*i, *(i+1)) 为 true，则范围 [first1, last1) 内的连续迭代器 i 和 i+1 属于同一段；否则属于不同的段。

对于不满足结合律的运算符的结果是不确定的（例如，浮点类型的加法）。对于不满足结合律的运算符的结果可能因运行而异。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::host 并行化执行策略，使用 exclusive_scan_by_key：

```
#include <thrust/scan.h>
#include <thrust/functional.h>
#include <thrust/execution_policy.h>
...

int keys[10] = {0, 0, 0, 1, 1, 2, 3, 3, 3, 3};
int vals[10] = {1, 1, 1, 1, 1, 1, 1, 1, 1, 1};

int init = 5;

thrust::equal_to<int> binary_pred;

thrust::exclusive_scan_by_key(thrust::host, key, key + 10, vals, vals,
↪init, binary_pred); // in-place scan
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
// vals is now {5, 6, 7, 5, 6, 5, 5, 6, 7, 8};
```

函数参数

- **exec**：用于并行化的执行策略
- **first1**：键序列的起始
- **last1**：键序列的末尾
- **first2**：输入值的序列起始
- **result**：输出值的序列起始
- **init**：exclusive 和的初始值
- **binary_pred**：用于确定键相等性的二元谓词

前提条件

- first1 可以等于 result ，但除此外，范围 [first1, last1) 不应与范围 [result, result + (last1 - first1)) 重叠
- first2 可以等于 result ，但除此外，范围 [first2, first2 + (last1 - first1) 不应与范围 [result, result + (last1 - first1)) 重叠

返回值

输出序列的末尾

另请参阅

- exclusive_scan
- inclusive_scan_by_key

thrust::exclusive_scan_by_key

```
template <typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator,
  typename T,
  typename BinaryPredicate> OutputIterator exclusive_scan_by_
↪key(InputIterator1 first1,
  InputIterator1 last1,
  InputIterator2 first2,
  OutputIterator result,
  T init,
  BinaryPredicate binary_pred);
```

exclusive_scan_by_key 计算 exclusive 键值或 segmented 前缀和运算。exclusive 指的是每个结果在部分和中不包含相应的输入操作数。segmented 指的是部分和被分成不同的段。即，在每个段内计算单独的 exclusive 扫描运算。使用示例，请参见以下代码示例。

此版本的 exclusive_scan_by_key 使用值 init 初始化 exclusive 扫描运算。

此版本的 exclusive_scan_by_key 使用二元谓词 binary_pred 比较相邻键。具体而言，如果 binary_pred(*i, *(i+1)) 为 true，则范围 [first1, last1) 内的连续迭代器 i 和 i+1 属于同一段；否则属于不同的段。

对于不满足结合律的运算符的结果是不确定的（例如，浮点类型的加法）。对于不满足结合律的运算符的结果可能因运行而异。

以下代码片段演示了如何使用 exclusive_scan_by_key。.

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

```
#include <thrust/scan.h>
#include <thrust/functional.h>

int keys[10] = {0, 0, 0, 1, 1, 2, 3, 3, 3, 3};
int vals[10] = {1, 1, 1, 1, 1, 1, 1, 1, 1, 1};

int init = 5;

thrust::equal_to<int> binary_pred;

thrust::exclusive_scan_by_key(key, key + 10, vals, vals, init, binary_
↪pred); // in-place scan

// vals is now {5, 6, 7, 5, 6, 5, 5, 6, 7, 8};
```

函数参数

- **first1**：键序列的起始
- **last1**：键序列的末尾
- **first2**：输入值的序列起始
- **result**：输出值的序列起始
- **init**：exclusive 和的初始值
- **binary_pred**：用于确定键相等性的二元谓词

前提条件

- first1 可以等于 result，但除此外，范围 [first1, last1) 不应与范围 [result, result + (last1 - first1)) 重叠
- first2 可以等于 result，但除此外，范围 [first2, first2 + (last1 - first1) 不应与范围 [result, result + (last1 - first1)) 重叠

返回值

输出序列的末尾

另请参阅

- exclusive_scan
- inclusive_scan_by_key

thrust::exclusive_scan_by_key

```
template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator,
  typename T,
  typename BinaryPredicate,
  typename AssociativeOperator>
__host__ __device__ OutputIterator
exclusive_scan_by_key(const thrust::detail::execution_policy_base<
↪DerivedPolicy > & exec,
  InputIterator1 first1,
  InputIterator1 last1,
  InputIterator2 first2,
  OutputIterator result,
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
  T init,
  BinaryPredicate binary_pred,
  AssociativeOperator binary_op);
```

exclusive_scan_by_key 计算 exclusive 键值或 segmented 前缀和运算。exclusive 指的是每个结果在部分和中不包含相应的输入操作数。segmented 指的是部分和被分成不同的段。即，在每个段内计算单独的 exclusive 扫描运算。使用示例，请参见以下代码示例。

此版本的 exclusive_scan_by_key 使用值 init 初始化 exclusive 扫描运算。

此版本的 exclusive_scan_by_key 使用二元谓词 binary_pred 比较相邻键。具体而言，如果 binary_pred(*i, *(i+1)) 为 true，则范围 [first1, last1) 内的连续迭代器 i 和 i+1 属于同一段；否则属于不同的段。

此版本的 exclusive_scan_by_key 使用具有结合律的运算符 binary_op 执行前缀和运算。当输入和输出序列相同时，就地执行扫描。

对于不满足结合律的运算符的结果是不确定的（例如，浮点类型的加法）。对于不满足结合律的运算符的结果可能因运行而异。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::host 并行化执行策略，使用 exclusive_scan_by_key：

```
#include <thrust/scan.h>
#include <thrust/functional.h>
#include <thrust/execution_policy.h>
...

int keys[10] = {0, 0, 0, 1, 1, 2, 3, 3, 3, 3};
int vals[10] = {1, 1, 1, 1, 1, 1, 1, 1, 1, 1};

int init = 5;

thrust::equal_to<int> binary_pred;
thrust::plus<int>     binary_op;

thrust::exclusive_scan_by_key(thrust::host, key, key + 10, vals, vals,
↪init, binary_pred, binary_op); // in-place scan

// vals is now {5, 6, 7, 5, 6, 5, 5, 6, 7, 8};
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **InputIterator1**：是 Input Iterator 的模型
- **InputIterator2**：是 Input Iterator 的模型，InputIterator2 的 value_type 可转换为 OutputIterator 的 value_type
- **OutputIterator**：是 Output Iterator 的模型，如果 x 和 y 是 OutputIterator 的 value_type 的对象，则定义 binary_op(x,y)
- **T**：可转换为 OutputIterator 的 value_type
- **BinaryPredicate**：是 Binary Predicate 的模型
- **AssociativeOperator**：是 Binary Function 的模型，AssociativeOperator 的 result_type 可转换为 OutputIterator 的 value_type

函数参数

- **exec**：用于并行化的执行策略

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

- **first1**：键序列的起始
- **last1**：键序列的末尾
- **first2**：输入值的序列起始
- **result**：输出值的序列起始
- **init**：exclusive 和的初始值
- **binary_pred**：用于确定键相等性的二元谓词
- **binary_op**：具有结合律的运算符，用于求和

前提条件

- first1 可以等于 result，但除此外，范围 [first1, last1) 不应与范围 [result, result + (last1 - first1)) 重叠
- first2 可以等于 result，但除此外，范围 [first2, first2 + (last1 - first1) 不应与范围 [result, result + (last1 - first1)) 重叠

返回值

输出序列的末尾

另请参阅

- exclusive_scan
- inclusive_scan_by_key

thrust::exclusive_scan_by_key

```
template <typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator,
  typename T,
  typename BinaryPredicate,
  typename AssociativeOperator>
OutputIterator exclusive_scan_by_key(InputIterator1 first1,
  InputIterator1 last1,
  InputIterator2 first2,
  OutputIterator result,
  T init,
  BinaryPredicate binary_pred,
  AssociativeOperator binary_op);
```

exclusive_scan_by_key 计算 exclusive 键值或 segmented 前缀和运算。exclusive 指的是每个结果在部分和中不包含相应的输入操作数。segmented 指的是部分和被分成不同的段。即，在每个段内计算单独的 exclusive 扫描运算。使用示例，请参见以下代码示例。

此版本的 exclusive_scan_by_key 使用值 init 初始化 exclusive 扫描运算。

此版本的 exclusive_scan_by_key 使用二元谓词 binary_pred 比较相邻键。具体而言，如果 binary_pred(*i, *(i+1)) 为 true，则范围 [first1, last1) 内的连续迭代器 i 和 i+1 属于同一段；否则属于不同的段。

此版本的 exclusive_scan_by_key 使用具有结合律的运算符 binary_op 执行前缀和运算。当输入和输出序列相同时，就地执行扫描。

对于不满足结合律的运算符的结果是不确定的（例如，浮点类型的加法）。对于不满足结合律的运算符的结果可能因运行而异。

以下代码片段演示了如何使用 exclusive_scan_by_key。

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

```
#include <thrust/scan.h>
#include <thrust/functional.h>

int keys[10] = {0, 0, 0, 1, 1, 2, 3, 3, 3, 3};
int vals[10] = {1, 1, 1, 1, 1, 1, 1, 1, 1, 1};

int init = 5;

thrust::equal_to<int> binary_pred;
thrust::plus<int>     binary_op;

thrust::exclusive_scan_by_key(key, key + 10, vals, vals, init, binary_pred,
↪ binary_op); // in-place scan

// vals is now {5, 6, 7, 5, 6, 5, 5, 6, 7, 8};
```

模板参数

- **InputIterator1**：是 Input Iterator 的模型
- **InputIterator2**：是 Input Iterator 的模型，InputIterator2 的 value_type 可转换为 OutputIterator 的 value_type
- **OutputIterator**：是 Output Iterator 的模型，如果 x 和 y 是 OutputIterator 的 value_type 的对象，则定义 binary_op(x,y)
- **T**：可转换为 OutputIterator 的 value_type
- **BinaryPredicate**：是 Binary Predicate 的模型
- **AssociativeOperator**：是 Binary Function 的模型，AssociativeOperator 的 result_type 可转换为 OutputIterator 的 value_type

函数参数

- **first1**：键序列的起始
- **last1**：键序列的末尾
- **first2**：输入值的序列起始
- **result**：输出值的序列起始
- **init**：exclusive 和的初始值
- **binary_pred**：用于确定键相等性的二元谓词
- **binary_op**：具有结合律的运算符，用于求和

前提条件

- first1 可以等于 result，但除此外，范围 [first1, last1) 不应与范围 [result, result + (last1 - first1)) 重叠
- first2 可以等于 result，但除此外，范围 [first2, first2 + (last1 - first1) 不应与范围 [result, result + (last1 - first1)) 重叠

返回值

输出序列的末尾

另请参阅

- exclusive_scan
- inclusive_scan_by_key

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

2.1.3.3 变换前缀和（Transformed Prefix Sum）

函数

```
template <typename DerivedPolicy,
  typename InputIterator,
  typename OutputIterator,
  typename UnaryFunction,
  typename AssociativeOperator>
__host__ __device__ OutputIterator
thrust::transform_inclusive_scan(const thrust::detail::execution_policy_
↪base< DerivedPolicy > & exec,
  InputIterator first,
  InputIterator last,
  OutputIterator result,
  UnaryFunction unary_op,
  AssociativeOperator binary_op);

template <typename InputIterator,
  typename OutputIterator,
  typename UnaryFunction,
  typename AssociativeOperator>
OutputIterator
thrust::transform_inclusive_scan(InputIterator first,
  InputIterator last,
  OutputIterator result,
  UnaryFunction unary_op,
  AssociativeOperator binary_op);

template <typename DerivedPolicy,
  typename InputIterator,
  typename OutputIterator,
  typename UnaryFunction,
  typename T,
  typename AssociativeOperator>
__host__ __device__ OutputIterator
thrust::transform_exclusive_scan(const thrust::detail::execution_policy_
↪base< DerivedPolicy > & exec,
  InputIterator first,
  InputIterator last,
  OutputIterator result,
  UnaryFunction unary_op,
  T init,
  AssociativeOperator binary_op);

template <typename InputIterator,
  typename OutputIterator,
  typename UnaryFunction,
  typename T,
  typename AssociativeOperator>
OutputIterator
thrust::transform_exclusive_scan(InputIterator first,
  InputIterator last,
  OutputIterator result,
  UnaryFunction unary_op,
  T init,
  AssociativeOperator binary_op);
```

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

thrust::transform_inclusive_scan

```
template <typename DerivedPolicy,
  typename InputIterator,
  typename OutputIterator,
  typename UnaryFunction,
  typename AssociativeOperator>
__host__ __device__ OutputIterator
transform_inclusive_scan(const thrust::detail::execution_policy_base<
↪DerivedPolicy > & exec,
  InputIterator first,
  InputIterator last,
  OutputIterator result,
  UnaryFunction unary_op,
  AssociativeOperator binary_op);
```

transform_inclusive_scan 融合了 transform 和 inclusive_scan 运算。transform_inclusive_scan 相当于执行由 unary_op 定义的变换，使序列成为临时序列，然后对变换后的序列执行 inclusive_scan 运算。多数情况下，将这两个运算融合在一起更高效，因为需要的内存读取和写入更少了。在 transform_inclusive_scan 中，将 unary_op(*first) 赋给 *result，将 binary_op(unary_op(*first), unary_op(*(first + 1))) 的结果 赋给 *(result + 1)，依此类推。允许就地执行变换扫描操作。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::host 并行化执行策略，使用 transform_inclusive_scan：

```
#include <thrust/transform_scan.h>
#include <thrust/execution_policy.h>
...

int data[6] = {1, 0, 2, 2, 1, 3};

thrust::negate<int> unary_op;
thrust::plus<int> binary_op;

thrust::transform_inclusive_scan(thrust::host, data, data + 6, data, unary_
↪op, binary_op); // in-place scan

// data is now {-1, -1, -3, -5, -6, -9}
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **InputIterator**：是 Input Iterator 的模型，InputIterator 的 value_type 可转换为 unary_op 的 input type
- **OutputIterator**：是 Output Iterator 的模型
- **UnaryFunction**：是 Unary Function 的模型，且接受 InputIterator 的 value_type 的输入，Unary-Function 的 result_type 可转换为 OutputIterator 的 value_type
- **AssociativeOperator**：是 Binary Function 的模型，AssociativeOperator 的 result_type 可转换为 OutputIterator 的 value_type

函数参数

- **exec**：用于并行化的执行策略
- **first**：输入序列的起始
- **last**：输入序列的末尾
- **result**：输出序列的起始

飞腾信息 MetaX Confidential 2024-11-18 15:00:03

- **unary_op**：用于变换输入序列的函数
- **binary_op**：具有结合律的运算符，用于变换值求和

前提条件

first 可以等于 result ，但除此外，范围 [first, last) 不应与范围 [result, result + (last - first)) 重叠

返回值

输出序列的末尾

另请参阅

- transform
- inclusive_scan

thrust::transform_inclusive_scan

```
template <typename InputIterator,
  typename OutputIterator,
  typename UnaryFunction,
  typename AssociativeOperator> OutputIterator
transform_inclusive_scan(InputIterator first,
  InputIterator last,
  OutputIterator result,
  UnaryFunction unary_op,
  AssociativeOperator binary_op);
```

transform_inclusive_scan 融合了 transform 和 inclusive_scan 运算。transform_inclusive_scan 相当于执行由 unary_op 定义的变换，使序列成为临时序列，然后对变换后的序列执行 inclusive_scan 运算。多数情况下，将这两个运算融合在一起更高效，因为需要的内存读取和写入更少了。在 transform_inclusive_scan 中，将 unary_op(*first) 赋给 *result ，将 binary_op(unary_op(*first), unary_op(*(first + 1))) 的结果 赋给 *(result + 1) ，依此类推。允许就地执行变换扫描操作。

以下代码片段演示了如何使用 transform_inclusive_scan。

```
1#include <thrust/transform_scan.h>

int data[6] = {1, 0, 2, 2, 1, 3};

thrust::negate<int> unary_op;
thrust::plus<int> binary_op;

thrust::transform_inclusive_scan(data, data + 6, data, unary_op, binary_
↪op); // in-place scan

// data is now {-1, -1, -3, -5, -6, -9}
```

模板参数

- **InputIterator**：是 Input Iterator 的模型，InputIterator 的 value_type 可转换为 unary_op 的 input type
- **OutputIterator**：是 Output Iterator 的模型
- **UnaryFunction**：是 Unary Function 的模型，且接受 InputIterator 的 value_type 的输入，Unary-Function 的 result_type 可转换为 OutputIterator 的 value_type
- **AssociativeOperator**：是 Binary Function 的模型，AssociativeOperator 的 result_type 可转换为 OutputIterator 的 value_type

函数参数

飞腾信息 MetaX Confidential 2024-11-13 00:00:00

- **first**：输入序列的起始
- **last**：输入序列的末尾
- **result**：输出序列的起始
- **unary_op**：用于变换输入序列的函数
- **binary_op**：具有结合律的运算符，用于变换值求和

前提条件

first 可以等于 result ，但除此外，范围 [first, last) 不应与范围 [result, result + (last - first)) 重叠

返回值

输出序列的末尾

另请参阅

- transform
- inclusive_scan

thrust::transform_exclusive_scan

```
template <typename DerivedPolicy,
  typename InputIterator,
  typename OutputIterator,
  typename UnaryFunction,
  typename T,
  typename AssociativeOperator>
__host__ __device__ OutputIterator
transform_exclusive_scan(const thrust::detail::execution_policy_base<
↪DerivedPolicy > & exec,
  InputIterator first,
  InputIterator last,
  OutputIterator result,
  UnaryFunction unary_op,
  T init,
  AssociativeOperator binary_op);
```

transform_exclusive_scan 融合了 transform 和 exclusive_scan 运算。transform_exclusive_scan 相当于执行由 unary_op 定义的变换，使序列成为临时序列，然后对变换后的序列执行 exclusive_scan 运算。多数情况下，将这两个运算融合在一起更高效，因为需要的内存读取和写入更少了。在 transform_exclusive_scan 中，将 init 赋给 *result ，将 binary_op(init, unary_op(*first)) 的结果 赋给 *(result + 1) ，依此类推。允许就地执行变换扫描操作。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::host 并行化执行策略，使用 transform_exclusive_scan：

```
#include <thrust/transform_scan.h>
#include <thrust/execution_policy.h>
...

int data[6] = {1, 0, 2, 2, 1, 3};

thrust::negate<int> unary_op;
thrust::plus<int> binary_op;

thrust::transform_exclusive_scan(thrust::host, data, data + 6, data, unary_
↪op, 4, binary_op); // in-place scan
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
// data is now {4, 3, 3, 1, -1, -2}
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **InputIterator**：是 Input Iterator 的模型，InputIterator 的 value_type 可转换为 unary_op 的 input type
- **OutputIterator**：是 Output Iterator 的模型
- **UnaryFunction**：是 Unary Function 的模型，且接受 InputIterator 的 value_type 的输入，UnaryFunction 的 result_type 可转换为 OutputIterator 的 value_type
- **T**：可转换为 OutputIterator 的 value_type
- **AssociativeOperator**：是 Binary Function 的模型，AssociativeOperator 的 result_type 可转换为 OutputIterator 的 value_type

函数参数

- **exec**：用于并行化的执行策略
- **first**：输入序列的起始
- **last**：输入序列的末尾
- **result**：输出序列的起始
- **unary_op**：用于变换输入序列的函数
- **init**：exclusive_scan 的初始值
- **binary_op**：具有结合律的运算符，用于变换值求和

前提条件

first 可以等于 result ，但除此外，范围 [first, last) 不应与范围 [result, result + (last - first)) 重叠

返回值

输出序列的末尾

另请参阅

- transform
- exclusive_scan

thrust::transform_exclusive_scan

```
template <typename InputIterator,
  typename OutputIterator,
  typename UnaryFunction,
  typename T,
  typename AssociativeOperator> OutputIterator transform_exclusive_
↪scan(InputIterator first,
  InputIterator last,
  OutputIterator result,
  UnaryFunction unary_op,
  T init,
  AssociativeOperator binary_op);
```

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

transform_exclusive_scan 融合了 transform 和 exclusive_scan 运算。transform_exclusive_scan 相当于执行由 unary_op 定义的变换，使序列成为临时序列，然后对变换后的序列执行 exclusive_scan 运算。多数情况下，将这两个运算融合在一起更高效，因为需要的内存读取和写入更少了。在 transform_exclusive_scan 中，将 init 赋给 *result ，将 binary_op(init, unary_op(*first)) 的结果 赋给 *(result + 1) ，依此类推。允许就地执行变换扫描操作。

以下代码片段演示了如何使用 transform_exclusive_scan。

```
#include <thrust/transform_scan.h>

int data[6] = {1, 0, 2, 2, 1, 3};

thrust::negate<int> unary_op;
thrust::plus<int> binary_op;

thrust::transform_exclusive_scan(data, data + 6, data, unary_op, 4, binary_
↪op); // in-place scan

// data is now {4, 3, 3, 1, -1, -2}
```

模板参数

- **InputIterator**：是 Input Iterator 的模型，InputIterator 的 value_type 可转换为 unary_op 的 input type
- **OutputIterator**：是 Output Iterator 的模型
- **UnaryFunction**：是 Unary Function 的模型，且接受 InputIterator 的 value_type 的输入，UnaryFunction 的 result_type 可转换为 OutputIterator 的 value_type
- **T**：可转换为 OutputIterator 的 value_type
- **AssociativeOperator**：是 Binary Function 的模型，AssociativeOperator 的 result_type 可转换为 OutputIterator 的 value_type

函数参数

- **first**：输入序列的起始
- **last**：输入序列的末尾
- **result**：输出序列的起始
- **unary_op**：用于变换输入序列的函数
- **init**：exclusive_scan 的初始值
- **binary_op**：具有结合律的运算符，用于变换值求和

前提条件

first 可以等于 result ，但除此外，范围 [first, last) 不应与范围 [result, result + (last - first)) 重叠

返回值

输出序列的末尾

另请参阅

- transform
- exclusive_scan

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

2.1.4 归约（Reductions）

2.1.4.1 归约

函数

```
template <typename DerivedPolicy,
  typename InputIterator>
__host__ __device__ thrust::iterator_traits< InputIterator >::value_type
thrust::reduce(const thrust::detail::execution_policy_base< DerivedPolicy >
↪ & exec,
  InputIterator first,
  InputIterator last);

template <typename InputIterator>
thrust::iterator_traits< InputIterator >::value_type
thrust::reduce(InputIterator first,
  InputIterator last);

template <typename DerivedPolicy,
  typename InputIterator,
  typename T>
__host__ __device__ T
thrust::reduce(const thrust::detail::execution_policy_base< DerivedPolicy >
↪ & exec,
  InputIterator first,
  InputIterator last,
  T init);

template <typename InputIterator,
  typename T>
T
thrust::reduce(InputIterator first,
  InputIterator last,
  T init);

template <typename DerivedPolicy,
  typename InputIterator,
  typename T,
  typename BinaryFunction>
__host__ __device__
T
thrust::reduce(const thrust::detail::execution_policy_base< DerivedPolicy >
↪ & exec,
  InputIterator first,
  InputIterator last,
  T init,
  BinaryFunction binary_op);

template <typename InputIterator,
  typename T,
  typename BinaryFunction>
T
thrust::reduce(InputIterator first,
  InputIterator last,
  T init,
  BinaryFunction binary_op);
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator1,
  typename OutputIterator2>
__host__ __device__ thrust::pair< OutputIterator1, OutputIterator2 >
thrust::reduce_by_key(const thrust::detail::execution_policy_base<
↪DerivedPolicy > & exec,
  InputIterator1 keys_first,
  InputIterator1 keys_last,
  InputIterator2 values_first,
  OutputIterator1 keys_output,
  OutputIterator2 values_output);

template <typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator1,
  typename OutputIterator2>
thrust::pair< OutputIterator1, OutputIterator2 >
thrust::reduce_by_key(InputIterator1 keys_first,
  InputIterator1 keys_last,
  InputIterator2 values_first,
  OutputIterator1 keys_output,
  OutputIterator2 values_output);

template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator1,
  typename OutputIterator2,
  typename BinaryPredicate>
__host__ __device__ thrust::pair< OutputIterator1, OutputIterator2 >
thrust::reduce_by_key(const thrust::detail::execution_policy_base<
↪DerivedPolicy > & exec,
  InputIterator1 keys_first,
  InputIterator1 keys_last,
  InputIterator2 values_first,
  OutputIterator1 keys_output,
  OutputIterator2 values_output,
  BinaryPredicate binary_pred);

template <typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator1,
  typename OutputIterator2,
  typename BinaryPredicate>
thrust::pair< OutputIterator1, OutputIterator2 >
thrust::reduce_by_key(InputIterator1 keys_first,
  InputIterator1 keys_last,
  InputIterator2 values_first,
  OutputIterator1 keys_output,
  OutputIterator2 values_output,
  BinaryPredicate binary_pred);

template <typename DerivedPolicy,
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-12 15:00:00

(续上页)

```
  typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator1,
  typename OutputIterator2,
  typename BinaryPredicate,
  typename BinaryFunction>
__host__ __device__ thrust::pair< OutputIterator1, OutputIterator2 >
thrust::reduce_by_key(const thrust::detail::execution_policy_base<
↪DerivedPolicy > & exec,
  InputIterator1 keys_first,
  InputIterator1 keys_last,
  InputIterator2 values_first,
  OutputIterator1 keys_output,
  OutputIterator2 values_output,
  BinaryPredicate binary_pred,
  BinaryFunction binary_op);

template <typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator1,
  typename OutputIterator2,
  typename BinaryPredicate,
  typename BinaryFunction>
thrust::pair< OutputIterator1, OutputIterator2 >
thrust::reduce_by_key(InputIterator1 keys_first,
  InputIterator1 keys_last,
  InputIterator2 values_first,
  OutputIterator1 keys_output,
  OutputIterator2 values_output,
  BinaryPredicate binary_pred,
  BinaryFunction binary_op);
```

thrust::reduce

```
template <typename DerivedPolicy,
  typename InputIterator>
__host__ __device__ thrust::iterator_traits< InputIterator >::value_type
reduce(const thrust::detail::execution_policy_base< DerivedPolicy > & exec,
  InputIterator first,
  InputIterator last);
```

reduce 是求和的泛化：它计算范围 [first, last) 内所有元素的和（或其他二元运算）。此版本的 reduce 将 0 作为归约的初始值。reduce 类似于 C++ 标准模板库的 std::accumulate。这两个函数之间的主要区别是 std::accumulate 保证求和顺序，而 reduce 需要二元运算的结合律来并行化归约。

请注意，reduce 还假设二元归约运算符（在这种情况下为运算符 +）具有交换律。如果归约运算符不具有交换律，那么不应使用 thrust::reduce。相反，可以使用 inclusive_scan （不需要具有交换律）并选择输出数组的最后一个元素。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::host 并行化执行策略，使用 reduce 计算整数序列的和：

```
#include <thrust/reduce.h>
#include <thrust/execution_policy.h>
...
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
int data[6] = {1, 0, 2, 2, 1, 3};
int result = thrust::reduce(thrust::host, data, data + 6);

// result == 9
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **InputIterator**：是 Input Iterator 的模型，如果 x 和 y 是 InputIterator 的 value_type 的对象，则定义 x + y ，且可转换为 InputIterator 的 value_type；如果 T 是 InputIterator 的 value_type ，则定义 T(0)

函数参数

- **exec**：用于并行化的执行策略
- **first**：序列起始
- **last**：序列末尾

返回值

归约的结果

另请参阅

https://en.cppreference.com/w/cpp/algorithm/accumulate

thrust::reduce

```
template <typename InputIterator>
thrust::iterator_traits< InputIterator >::value_type
reduce(InputIterator first,
  InputIterator last);
```

reduce 是求和的泛化：它计算范围 [first, last) 内所有元素的和（或其他二元运算）。此版本的 reduce 将 0 作为归约的初始值。reduce 类似于 C++ 标准模板库的 std::accumulate。这两个函数之间的主要区别是 std::accumulate 保证求和顺序，而 reduce 需要二元运算的结合律来并行化归约。

请注意，reduce 还假设二元归约运算符（在这种情况下为运算符 +）具有交换律。如果归约运算符不具有交换律，那么不应使用 thrust::reduce。相反，可以使用 inclusive_scan （不需要具有交换律）并选择输出数组的最后一个元素。

以下代码片段演示了如何使用 reduce 计算整数序列的和：

```
#include <thrust/reduce.h>
...
int data[6] = {1, 0, 2, 2, 1, 3};
int result = thrust::reduce(data, data + 6);

// result == 9
```

模板参数

InputIterator：是 Input Iterator 的模型，如果 x 和 y 是 InputIterator 的 value_type 的对象，则定义 x + y ，且可转换为 InputIterator 的 value_type；如果 T 是 InputIterator 的 value_type ，则定义 T(0)

函数参数

- **first**：序列起始
- **last**：序列末尾

沐曦信息 MetaX Confidential 2024-11-18 15:00:00

返回值

归约的结果

另请参阅

https://en.cppreference.com/w/cpp/algorithm/accumulate

thrust::reduce

```
template <typename DerivedPolicy,
  typename InputIterator,
  typename T>
 __host__ __device__ T
 reduce(const thrust::detail::execution_policy_base< DerivedPolicy > &
↪exec,
 InputIterator first,
 InputIterator last,
 T init);
```

reduce 是求和的泛化：它计算范围 [first, last) 内所有元素的和（或其他二元运算）。此版本的 reduce 将 init 作为归约的初始值。reduce 类似于 C++ 标准模板库的 std::accumulate。这两个函数之间的主要区别是 std::accumulate 保证求和顺序，而 reduce 需要二元运算的结合律来并行化归约。

请注意，reduce 还假设二元归约运算符（在这种情况下为运算符 +）具有交换律。如果归约运算符不具有交换律，那么不应使用 thrust::reduce。相反，可以使用 inclusive_scan （不需要具有交换律）并选择输出数组的最后一个元素。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::host 并行化执行策略，使用 reduce 计算包含初始值的整数序列的和：

```
#include <thrust/reduce.h>
#include <thrust/execution_policy.h>
...
int data[6] = {1, 0, 2, 2, 1, 3};
int result = thrust::reduce(thrust::host, data, data + 6, 1);

// result == 10
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **InputIterator**：是 Input Iterator 的模型，如果 x 和 y 是 InputIterator 的 value_type 的对象，则定义 x + y ，且可转换为 T
- **T**：可转换为 InputIterator 的 value_type

函数参数

- **exec**：用于并行化的执行策略
- **first**：输入序列的起始
- **last**：输入序列的末尾
- **init**：初始值

返回值

归约的结果

另请参阅

飞腾信息 MoFax Confidential 2024-11-18 15:00:00

https://en.cppreference.com/w/cpp/algorithm/accumulate

thrust::reduce

```
template <typename InputIterator,
  typename T>
T
reduce(InputIterator first,
  InputIterator last,
  T init);
```

reduce 是求和的泛化：它计算范围 [first, last) 内所有元素的和（或其他二元运算）。此版本的 reduce 将 init 作为归约的初始值。reduce 类似于 C++ 标准模板库的 std::accumulate。这两个函数之间的主要区别是 std::accumulate 保证求和顺序，而 reduce 需要二元运算的结合律来并行化归约。

请注意，reduce 还假设二元归约运算符（在这种情况下为运算符 +）具有交换律。如果归约运算符不具有交换律，那么不应使用 thrust::reduce。相反，可以使用 inclusive_scan （不需要具有交换律）并选择输出数组的最后一个元素。

以下代码片段演示了如何使用 reduce 计算包含初始值的整数序列的和：

```
#include <thrust/reduce.h>
...
int data[6] = {1, 0, 2, 2, 1, 3};
int result = thrust::reduce(data, data + 6, 1);

// result == 10
```

模板参数

- **InputIterator**：是 Input Iterator 的模型，如果 x 和 y 是 InputIterator 的 value_type 的对象，则定义 x + y ，且可转换为 T
- **T**：可转换为 InputIterator 的 value_type

函数参数

- **first**：输入序列的起始
- **last**：输入序列的末尾
- **init**：初始值

返回值

归约的结果

另请参阅

https://en.cppreference.com/w/cpp/algorithm/accumulate

thrust::reduce

```
template <typename DerivedPolicy,
  typename InputIterator,
  typename T,
  typename BinaryFunction>
__host__ __device__ T
reduce(const thrust::detail::execution_policy_base< DerivedPolicy > & exec,
  InputIterator first,
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
  InputIterator last,
  T init,
  BinaryFunction binary_op);
```

reduce 是求和的泛化：它计算范围 [first, last) 内所有元素的和(或其他二元运算)。此版本的 reduce 将 init 作为归约的初始值、binary_op 作为求和的二元函数。reduce 类似于 C++ 标准模板库的 std::accumulate。这两个函数之间的主要区别是 std::accumulate 保证求和顺序，而 reduce 需要 binary_op 的结合律来并行化归约。

请注意，reduce 还假设二元归约运算符具有交换律，在这种情况下为 binary_op。如果归约运算符不具有交换律，那么不应使用 thrust::reduce。相反，可以使用 inclusive_scan （不需要具有交换律）并选择输出数组的最后一个元素。

```
#include <thrust/reduce.h>
#include <thrust/functional.h>
#include <thrust/execution_policy.h>
...
int data[6] = {1, 0, 2, 2, 1, 3};
int result = thrust::reduce(thrust::host,
   data, data + 6,
   -1,
   thrust::maximum<int>());
// result == 3
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **InputIterator**：是 Input Iterator 的模型，InputIterator 的 value_type 可转换为 T
- **T**：是 Assignable 的模型，且可转换为 BinaryFunction 的 first_argument_type 和 second_argument_type
- **BinaryFunction**：是 Binary Function 的模型，BinaryFunction 的 result_type 可转换为 OutputType

函数参数

- **exec**：用于并行化的执行策略
- **first**：输入序列的起始
- **last**：输入序列的末尾
- **init**：初始值
- **binary_op**：用于求和的二元函数

返回值

归约的结果

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/accumulate
- transform_reduce

thrust::reduce

```
template <typename InputIterator,
  typename T,
  typename BinaryFunction>
T
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
reduce(InputIterator first,
  InputIterator last,
  T init,
  BinaryFunction binary_op);
```

reduce 是求和的泛化：它计算范围 [first, last) 内所有元素的和（或其他二元运算）。此版本的 reduce 将 init 作为归约的初始值、binary_op 作为求和的二元函数。reduce 类似于 C++ 标准模板库的 std::accumulate。这两个函数之间的主要区别是 std::accumulate 保证求和顺序，而 reduce 需要 binary_op 的结合律来并行化归约。

请注意，reduce 还假设二元归约运算符具有交换律，在这种情况下为 binary_op。如果归约运算符不具有交换律，那么不应使用 thrust::reduce。相反，可以使用 inclusive_scan（不需要具有交换律）并选择输出数组的最后一个元素。

以下代码片段演示了如何使用 reduce 计算整数序列的最大值：

```
#include <thrust/reduce.h>
#include <thrust/functional.h>
...
int data[6] = {1, 0, 2, 2, 1, 3};
int result = thrust::reduce(data, data + 6,
    -1,
    thrust::maximum<int>());
// result == 3
```

模板参数

- **InputIterator**：是 Input Iterator 的模型，InputIterator 的 value_type 可转换为 T
- **T**：是 Assignable 的模型，且可转换为 BinaryFunction 的 first_argument_type 和 second_argument_type
- **BinaryFunction**：是 Binary Function 的模型，BinaryFunction 的 result_type 可转换为 OutputType

函数参数

- **first**：输入序列的起始
- **last**：输入序列的末尾
- **init**：初始值
- **binary_op**：用于求和的二元函数

返回值

归约的结果

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/accumulate
- transform_reduce

thrust::reduce_by_key

```
template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator1,
  typename OutputIterator2>
__host__ __device__ thrust::pair< OutputIterator1, OutputIterator2 >
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
reduce_by_key(const thrust::detail::execution_policy_base< DerivedPolicy >
↪& exec,
  InputIterator1 keys_first,
  InputIterator1 keys_last,
  InputIterator2 values_first,
  OutputIterator1 keys_output,
  OutputIterator2 values_output);
```

reduce_by_key 是 reduce 到键值对的泛化。对于范围 [keys_first, keys_last) 内每一组相等的连续键，reduce_by_key 将该组中的第一个元素复制到 keys_output。使用 plus 对范围内对应的值进行归约，并将结果复制到 values_output。

此版本的 reduce_by_key 使用函数对象 equal_to 测试相等性，并使用 plus 对具有相等键的值进行归约。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::host 并行化执行策略，使用 reduce_by_key 压缩键值对序列，并用相等的键求和：

```
#include <thrust/reduce.h>
#include <thrust/execution_policy.h>
...
const int N = 7;
int A[N] = {1, 3, 3, 3, 2, 2, 1}; // input keys
int B[N] = {9, 8, 7, 6, 5, 4, 3}; // input values
int C[N];    // output keys
int D[N];    // output values

thrust::pair<int*,int*> new_end;
new_end = thrust::reduce_by_key(thrust::host, A, A + N, B, C, D);

// The first four keys in C are now {1, 3, 2, 1} and new_end.first - C is
↪4.
// The first four values in D are now {9, 21, 9, 3} and new_end.second - D
↪is 4.
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **InputIterator1**：是 Input Iterator 的模型
- **InputIterator2**：是 Input Iterator 的模型
- **OutputIterator1**：是 Output Iterator 的模型，InputIterator1 的 value_type 可转换为 OutputIterator1 的 value_type
- **OutputIterator2**：是 Output Iterator 的模型，InputIterator2 的 value_type 可转换为 OutputIterator2 的 value_type

函数参数

- **exec**：用于并行化的执行策略
- **keys_first**：输入键范围的起始
- **keys_last**：输入键范围的末尾
- **values_first**：输入值范围的起始
- **keys_output**：输出键范围的起始
- **values_output**：输出值范围的起始

飞腾信息 MetaX confidential 2024-11-18 15:00:00

前提条件

输入范围不应与任何一个输出范围重叠

返回值

范围 [keys_output, keys_output_last) 和 [values_output, values_output_last) 末尾的一对迭代器

另请参阅

- reduce
- unique_copy
- unique_by_key
- unique_by_key_copy

thrust::reduce_by_key

```
template <typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator1,
  typename OutputIterator2>
thrust::pair< OutputIterator1, OutputIterator2 >
reduce_by_key(InputIterator1 keys_first,
  InputIterator1 keys_last,
  InputIterator2 values_first,
  OutputIterator1 keys_output,
  OutputIterator2 values_output);
```

reduce_by_key 是 reduce 到键值对的泛化。对于范围 [keys_first, keys_last) 内每一组相等的连续键，reduce_by_key 将该组中的第一个元素复制到 keys_output。使用 plus 对范围内对应的值进行归约，并将结果复制到 values_output。

此版本的 reduce_by_key 使用函数对象 equal_to 测试相等性，并使用 plus 对具有相等键的值进行归约。

以下代码片段演示了如何使用 reduce_by_key 压缩键值对序列，并用相等的键求和：

```
#include <thrust/reduce.h>
...
const int N = 7;
int A[N] = {1, 3, 3, 3, 2, 2, 1}; // input keys
int B[N] = {9, 8, 7, 6, 5, 4, 3}; // input values
int C[N];    // output keys
int D[N];    // output values

thrust::pair<int*,int*> new_end;
new_end = thrust::reduce_by_key(A, A + N, B, C, D);

// The first four keys in C are now {1, 3, 2, 1} and new_end.first - C is
↪4.
// The first four values in D are now {9, 21, 9, 3} and new_end.second - D
↪is 4.
```

模板参数

- **InputIterator1**：是 Input Iterator 的模型
- **InputIterator2**：是 Input Iterator 的模型

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

- **OutputIterator1**：是 Output Iterator 的模型，InputIterator1 的 value_type 可转换为 OutputIterator1 的 value_type
- **OutputIterator2**：是 Output Iterator 的模型，InputIterator2 的 value_type 可转换为 OutputIterator2 的 value_type

函数参数

- **keys_first**：输入键范围的起始
- **keys_last**：输入键范围的末尾
- **values_first**：输入值范围的起始
- **keys_output**：输出键范围的起始
- **values_output**：输出值范围的起始

前提条件

输入范围不应与任何一个输出范围重叠

返回值

范围 [keys_output, keys_output_last) 和 [values_output, values_output_last) 末尾的一对迭代器

另请参阅

- reduce
- unique_copy
- unique_by_key
- unique_by_key_copy

thrust::reduce_by_key

```
template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator1,
  typename OutputIterator2,
  typename BinaryPredicate>
__host__ __device__ thrust::pair< OutputIterator1, OutputIterator2 >
reduce_by_key(const thrust::detail::execution_policy_base< DerivedPolicy >
↪& exec,
  InputIterator1 keys_first,
  InputIterator1 keys_last,
  InputIterator2 values_first,
  OutputIterator1 keys_output,
  OutputIterator2 values_output,
  BinaryPredicate binary_pred);
```

reduce_by_key 是 reduce 到键值对的泛化。对于范围 [keys_first, keys_last) 内每一组相等的连续键，reduce_by_key 将该组中的第一个元素复制到 keys_output。使用 plus 对范围内对应的值进行归约，并将结果复制到 values_output。

此版本的 reduce_by_key 使用函数对象 binary_pred 测试相等性，并使用 plus 对具有相等键的值进行归约。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::host 并行化执行策略，使用 reduce_by_key 压缩键值对序列，并用相等的键求和：

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

```
#include <thrust/reduce.h>
#include <thrust/execution_policy.h>
...
const int N = 7;
int A[N] = {1, 3, 3, 3, 2, 2, 1}; // input keys
int B[N] = {9, 8, 7, 6, 5, 4, 3}; // input values
int C[N];    // output keys
int D[N];    // output values

thrust::pair<int*,int*> new_end;
thrust::equal_to<int> binary_pred;
new_end = thrust::reduce_by_key(thrust::host, A, A + N, B, C, D, binary_
↪pred);

// The first four keys in C are now {1, 3, 2, 1} and new_end.first - C is
↪4.
// The first four values in D are now {9, 21, 9, 3} and new_end.second - D
↪is 4.
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **InputIterator1**：是 Input Iterator 的模型
- **InputIterator2**：是 Input Iterator 的模型
- **OutputIterator1**：是 Output Iterator 的模型，InputIterator1 的 value_type 可转换为 OutputIterator1 的 value_type
- **OutputIterator2**：是 Output Iterator 的模型，InputIterator2 的 value_type 可转换为 OutputIterator2 的 value_type
- **BinaryPredicate**：是 Binary Predicate 的模型

函数参数

- **exec**：用于并行化的执行策略
- **keys_first**：输入键范围的起始
- **keys_last**：输入键范围的末尾
- **values_first**：输入值范围的起始
- **keys_output**：输出键范围的起始
- **values_output**：输出值范围的起始
- **binary_pred**：用于确定相等性的二元谓词

前提条件

输入范围不应与任何一个输出范围重叠

返回值

范围 [keys_output, keys_output_last) 和 [values_output, values_output_last) 末尾的一对迭代器

另请参阅

- reduce
- unique_copy
- unique_by_key
- unique_by_key_copy

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

thrust::reduce_by_key

```
template <typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator1,
  typename OutputIterator2,
  typename BinaryPredicate>
thrust::pair< OutputIterator1, OutputIterator2 >
reduce_by_key(InputIterator1 keys_first,
  InputIterator1 keys_last,
  InputIterator2 values_first,
  OutputIterator1 keys_output,
  OutputIterator2 values_output,
  BinaryPredicate binary_pred);
```

reduce_by_key 是 reduce 到键值对的泛化。对于范围 [keys_first, keys_last) 内每一组相等的连续键，reduce_by_key 将该组中的第一个元素复制到 keys_output。使用 plus 对范围内对应的值进行归约，并将结果复制到 values_output。

此版本的 reduce_by_key 使用函数对象 binary_pred 测试相等性，并使用 plus 对具有相等键的值进行归约。

以下代码片段演示了如何使用 reduce_by_key 压缩键值对序列，并用相等的键求和：

```
#include <thrust/reduce.h>
...
const int N = 7;
int A[N] = {1, 3, 3, 3, 2, 2, 1}; // input keys
int B[N] = {9, 8, 7, 6, 5, 4, 3}; // input values
int C[N];     // output keys
int D[N];     // output values

thrust::pair<int*,int*> new_end;
thrust::equal_to<int> binary_pred;
new_end = thrust::reduce_by_key(A, A + N, B, C, D, binary_pred);

// The first four keys in C are now {1, 3, 2, 1} and new_end.first - C is
↪4.
// The first four values in D are now {9, 21, 9, 3} and new_end.second - D
↪is 4.
```

模板参数

- **InputIterator1**：是 Input Iterator 的模型
- **InputIterator2**：是 Input Iterator 的模型
- **OutputIterator1**：是 Output Iterator 的模型，InputIterator1 的 value_type 可转换为 OutputIterator1 的 value_type
- **OutputIterator2**：是 Output Iterator 的模型，InputIterator2 的 value_type 可转换为 OutputIterator2 的 value_type
- **BinaryPredicate**：是 Binary Predicate 的模型

函数参数

- **keys_first**：输入键范围的起始
- **keys_last**：输入键范围的末尾
- **values_first**：输入值范围的起始
- **keys_output**：输出键范围的起始

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

- **values_output**：输出值范围的起始
- **binary_pred**：用于确定相等性的二元谓词

前提条件

输入范围不应与任何一个输出范围重叠

返回值

范围 [keys_output, keys_output_last) 和 [values_output, values_output_last) 末尾的一对迭代器

另请参阅

- reduce
- unique_copy
- unique_by_key
- unique_by_key_copy

thrust::reduce_by_key

```
template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator1,
  typename OutputIterator2,
  typename BinaryPredicate,
  typename BinaryFunction>
__host__ __device__ thrust::pair< OutputIterator1, OutputIterator2 >
reduce_by_key(const thrust::detail::execution_policy_base< DerivedPolicy >
↪& exec,
  InputIterator1 keys_first,
  InputIterator1 keys_last,
  InputIterator2 values_first,
  OutputIterator1 keys_output,
  OutputIterator2 values_output,
  BinaryPredicate binary_pred,
  BinaryFunction binary_op);
```

reduce_by_key 是 reduce 到键值对的泛化。对于范围 [keys_first, keys_last) 内每一组相等的连续键，reduce_by_key 将该组中的第一个元素复制到 keys_output。使用 BinaryFunction binary_op 对范围内的值进行归约，将结果复制到 values_output 具体而言，如果连续的键迭代器 i 和 (i + 1) 使得 binary_pred(*i, *(i+1)) 为 true ，相应的值会被归约为具有 binary_op 的单个值。

此版本的 reduce_by_key 使用函数对象 binary_pred 测试相等性，并使用 binary_op 对具有相等键的值进行归约。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::host 并行化执行策略，使用 reduce_by_key 压缩键值对序列，并用相等的键求和：

```
#include <thrust/reduce.h>
#include <thrust/execution_policy.h>
...
const int N = 7;
int A[N] = {1, 3, 3, 3, 2, 2, 1}; // input keys
int B[N] = {9, 8, 7, 6, 5, 4, 3}; // input values
int C[N];    // output keys
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
int D[N];    // output values

thrust::pair<int*,int*> new_end;
thrust::equal_to<int> binary_pred;
thrust::plus<int> binary_op;
new_end = thrust::reduce_by_key(thrust::host, A, A + N, B, C, D, binary_
↪pred, binary_op);

// The first four keys in C are now {1, 3, 2, 1} and new_end.first - C is
↪4.
// The first four values in D are now {9, 21, 9, 3} and new_end.second - D
↪is 4.
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **InputIterator1**：是 Input Iterator 的模型
- **InputIterator2**：是 Input Iterator 的模型
- **OutputIterator1**：是 Output Iterator 的模型，InputIterator1 的 value_type 可转换为 OutputIterator1 的 value_type
- **OutputIterator2**：是 Output Iterator 的模型，InputIterator2 的 value_type 可转换为 OutputIterator2 的 value_type
- **BinaryPredicate**：是 Binary Predicate 的模型
- **BinaryFunction**：是 Binary Function 的模型，BinaryFunction 的 result_type 可转换为 OutputIterator2 的 value_type

函数参数

- **exec**：用于并行化的执行策略
- **keys_first**：输入键范围的起始
- **keys_last**：输入键范围的末尾
- **values_first**：输入值范围的起始
- **keys_output**：输出键范围的起始
- **values_output**：输出值范围的起始
- **binary_pred**：用于确定相等性的二元谓词
- **binary_op**：用于累加值的二元函数

前提条件

输入范围不应与任何一个输出范围重叠

返回值

范围 [keys_output, keys_output_last) 和 [values_output, values_output_last) 末尾的一对迭代器

另请参阅

- reduce
- unique_copy
- unique_by_key
- unique_by_key_copy

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

thrust::reduce_by_key

```
template <typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator1,
  typename OutputIterator2,
  typename BinaryPredicate,
  typename BinaryFunction>
thrust::pair< OutputIterator1, OutputIterator2 >
reduce_by_key(InputIterator1 keys_first,
  InputIterator1 keys_last,
  InputIterator2 values_first,
  OutputIterator1 keys_output,
  OutputIterator2 values_output,
  BinaryPredicate binary_pred,
  BinaryFunction binary_op);
```

reduce_by_key 是 reduce 到键值对的泛化。对于范围 [keys_first, keys_last) 内每一组相等的连续键，reduce_by_key 将该组中的第一个元素复制到 keys_output。使用 BinaryFunction binary_op 对范围内的值进行归约，将结果复制到 values_output 具体而言，如果连续的键迭代器 i 和 (i + 1) 使得 binary_pred(*i, *(i+1)) 为 true ，相应的值会被归约为具有 binary_op 的单个值。

此版本的 reduce_by_key 使用函数对象 binary_pred 测试相等性，并使用 binary_op 对具有相等键的值进行归约。

以下代码片段演示了如何使用 reduce_by_key 压缩键值对序列，并用相等的键求和：

```
#include <thrust/reduce.h>
...
const int N = 7;
int A[N] = {1, 3, 3, 3, 2, 2, 1}; // input keys
int B[N] = {9, 8, 7, 6, 5, 4, 3}; // input values
int C[N];     // output keys
int D[N];     // output values

thrust::pair<int*,int*> new_end;
thrust::equal_to<int> binary_pred;
thrust::plus<int> binary_op;
new_end = thrust::reduce_by_key(A, A + N, B, C, D, binary_pred, binary_op);

// The first four keys in C are now {1, 3, 2, 1} and new_end.first - C is
↪4.
// The first four values in D are now {9, 21, 9, 3} and new_end.second - D
↪is 4.
```

模板参数

- **InputIterator1**：是 Input Iterator 的模型
- **InputIterator2**：是 Input Iterator 的模型
- **OutputIterator1**：是 Output Iterator 的模型，InputIterator1 的 value_type 可转换为 OutputIterator1 的 value_type
- **OutputIterator2**：是 Output Iterator 的模型，InputIterator2 的 value_type 可转换为 OutputIterator2 的 value_type
- **BinaryPredicate**：是 Binary Predicate 的模型
- **BinaryFunction**：是 Binary Function 的模型，BinaryFunction 的 result_type 可转换为 OutputIterator2 的 value_type

飞猪信息 Metax Confidential 2024-11-18 15:00:00

函数参数

- **keys_first**：输入键范围的起始
- **keys_last**：输入键范围的末尾
- **values_first**：输入值范围的起始
- **keys_output**：输出键范围的起始
- **values_output**：输出值范围的起始
- **binary_pred**：用于确定相等性的二元谓词
- **binary_op**：用于累加值的二元函数

前提条件

输入范围不应与任何一个输出范围重叠

返回值

范围 [keys_output, keys_output_last) 和 [values_output, values_output_last) 末尾的一对迭代器

另请参阅

- reduce
- unique_copy
- unique_by_key
- unique_by_key_copy

2.1.4.2 比较（Comparison）

函数

```
template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2>
__host__ __device__ bool
thrust::equal(const thrust::detail::execution_policy_base< DerivedPolicy >
 ↪& exec,
  InputIterator1 first1,
  InputIterator1 last1,
  InputIterator2 first2);

template <typename InputIterator1,
  typename InputIterator2>
bool
thrust::equal(InputIterator1 first1,
  InputIterator1 last1,
  InputIterator2 first2);

template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2,
  typename BinaryPredicate>
__host__ __device__ bool
thrust::equal(const thrust::detail::execution_policy_base< DerivedPolicy >
 ↪& exec,
  InputIterator1 first1,
  InputIterator1 last1,
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
  InputIterator2 first2,
  BinaryPredicate binary_pred);

template <typename InputIterator1,
  typename InputIterator2,
  typename BinaryPredicate>
bool
thrust::equal(InputIterator1 first1,
  InputIterator1 last1,
  InputIterator2 first2,
  BinaryPredicate binary_pred);
```

thrust::equal

```
template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2>
__host__ __device__ bool
equal(const thrust::detail::execution_policy_base< DerivedPolicy > & exec,
  InputIterator1 first1,
  InputIterator1 last1,
  InputIterator2 first2);
```

如果逐个比较元素时，两个范围 [first1, last1) 和 [first2, first2 + (last1 - first1)) 相等，equal 返回 true；否则返回 false。

当且仅当对于 [first1, last1) 中的每个迭代器 i ，*i == *(first2 + (i - first1)) ，此版本的 equal 返回 true。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::host 执行策略，使用 equal 测试两个范围的相等性：

```
#include <thrust/equal.h>
#include <thrust/execution_policy.h>
...
int A1[7] = {3, 1, 4, 1, 5, 9, 3};
int A2[7] = {3, 1, 4, 2, 8, 5, 7};
...
bool result = thrust::equal(thrust::host, A1, A1 + 7, A2);

// result == false
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **InputIterator1**：是 Input Iterator 的模型，InputIterator1 的 value_type 是 Equality Comparable 的模型，并且 InputIterator1 的 value_type 和 InputIterator2 的 value_type 可以进行相等性比较
- **InputIterator2**：是 Input Iterator 的模型，InputIterator2 的 value_type 是 Equality Comparable 的模型，并且 InputIterator2 的 value_type 和 InputIterator1 的 value_type 可以进行相等性比较

函数参数

- **exec**：用于并行化的执行策略
- **first1**：第一个序列的起始
- **last1**：第一个序列的末尾
- **first2**：第二个序列的起始

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

返回值

如果序列相等，返回 true ，否则返回 false

另请参阅

https://en.cppreference.com/w/cpp/algorithm/equal

thrust::equal

```
template <typename InputIterator1,
  typename InputIterator2>
bool
equal(InputIterator1 first1,
  InputIterator1 last1,
  InputIterator2 first2);
```

如果逐个比较元素时，两个范围 [first1, last1) 和 [first2, first2 + (last1 - first1)) 相等，equal 返回 true ；否则返回 false。

当且仅当对于 [first1, last1) 中的每个迭代器 i ，*i == *(first2 + (i - first1)) ，此版本的 equal 返回 true。

以下代码片段演示了如何使用 equal 测试两个范围的相等性：

```
#include <thrust/equal.h>
...
int A1[7] = {3, 1, 4, 1, 5, 9, 3};
int A2[7] = {3, 1, 4, 2, 8, 5, 7};
...
bool result = thrust::equal(A1, A1 + 7, A2);

// result == false
```

模板参数

- **InputIterator1**：是 Input Iterator 的模型，InputIterator1 的 value_type 是 Equality Comparable 的模型，并且 InputIterator1 的 value_type 和 InputIterator2 的 value_type 可以进行相等性比较
- **InputIterator2**：是 Input Iterator 的模型，InputIterator2 的 value_type 是 Equality Comparable 的模型，并且 InputIterator2 的 value_type 和 InputIterator1 的 value_type 可以进行相等性比较

函数参数

- **first1**：第一个序列的起始
- **last1**：第一个序列的末尾
- **first2**：第二个序列的起始

返回值

如果序列相等，返回 true ，否则返回 false

另请参阅

https://en.cppreference.com/w/cpp/algorithm/equal

thrust::equal

```
template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2,
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
  typename BinaryPredicate>
__host__ __device__ bool
equal(const thrust::detail::execution_policy_base< DerivedPolicy > & exec,
  InputIterator1 first1,
  InputIterator1 last1,
  InputIterator2 first2,
  BinaryPredicate binary_pred);
```

如果逐个比较元素时，两个范围 [first1, last1) 和 [first2, first2 + (last1 - first1)) 相等，equal 返回 true；否则返回 false。

当且仅当对于 [first1, last1) 中的每个迭代器 i，binary_pred(*i, *(first2 + (i - first1))) 为 true，此版本的 equal 返回 true。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::host 执行策略，使用 equal 比较两个范围内的元素模 2 取余的结果是否相等：

```
#include <thrust/equal.h>
#include <thrust/execution_policy.h>
...

struct compare_modulo_two
{
__host__ __device__
bool operator()(int x, int y) const
{
   return (x % 2) == (y % 2);
}
};
...
int x[6] = {0, 2, 4, 6, 8, 10};
int y[6] = {1, 3, 5, 7, 9, 11};

bool result = thrust::equal(x, x + 6, y, compare_modulo_two());

// result is false
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **InputIterator1**：是 Input Iterator 的模型，InputIterator1 的 value_type 可转换为 BinaryPredicate 的 first_argument_type
- **InputIterator2**：是 Input Iterator 的模型，InputIterator2 的 value_type 可转换为 BinaryPredicate 的 second_argument_type
- **BinaryPredicate**：是 Binary Predicate 的模型

函数参数

- **exec**：用于并行化的执行策略
- **first1**：第一个序列的起始
- **last1**：第一个序列的末尾
- **first2**：第二个序列的起始
- **binary_pred**：用于测试元素相等性的二元谓词

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

返回值

如果序列相等，返回 true ，否则返回 false

另请参阅

https://en.cppreference.com/w/cpp/algorithm/equal

thrust::equal

```
template <typename InputIterator1,
  typename InputIterator2,
  typename BinaryPredicate>
bool
equal(InputIterator1 first1,
  InputIterator1 last1,
  InputIterator2 first2,
  BinaryPredicate binary_pred);
```

如果逐个比较元素时，两个范围 [first1, last1) 和 [first2, first2 + (last1 - first1)) 相等，equal 返回 true ；否则返回 false。

当且仅当对于 [first1, last1) 中的每个迭代器 i ，binary_pred(*i, *(first2 + (i - first1))) 为 true ，此版本的 equal 返回 true。

以下代码片段演示了如何使用 equal 比较 modulo 2 两个范围内的元素：

```
#include <thrust/equal.h>

struct compare_modulo_two
{
__host__ __device__
bool operator()(int x, int y) const
{
   return (x % 2) == (y % 2);
}
};
...
int x[6] = {0, 2, 4, 6, 8, 10};
int y[6] = {1, 3, 5, 7, 9, 11};

bool result = thrust::equal(x, x + 5, y, compare_modulo_two());

// result is true
```

模板参数

- **InputIterator1**：是 Input Iterator 的模型，InputIterator1 的 value_type 可转换为 BinaryPredicate 的 first_argument_type
- **InputIterator2**：是 Input Iterator 的模型，InputIterator2 的 value_type 可转换为 BinaryPredicate 的 second_argument_type
- **BinaryPredicate**：是 Binary Predicate 的模型

函数参数

- **first1**：第一个序列的起始
- **last1**：第一个序列的末尾
- **first2**：第二个序列的起始

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

- **binary_pred**：用于测试元素相等性的二元谓词

返回值

如果序列相等，返回 true ，否则返回 false

另请参阅

https://en.cppreference.com/w/cpp/algorithm/equal

2.1.4.3 计数（Counting）

函数

```
template <typename DerivedPolicy,
  typename InputIterator,
  typename EqualityComparable>
__host__ __device__ thrust::iterator_traits< InputIterator >::difference_
↪type
thrust::count(const thrust::detail::execution_policy_base< DerivedPolicy >
↪& exec,
  InputIterator first,
  InputIterator last,
  const EqualityComparable & value);

template <typename InputIterator,
  typename EqualityComparable>
thrust::iterator_traits< InputIterator >::difference_type
thrust::count(InputIterator first,
  InputIterator last,
  const EqualityComparable & value);

template <typename DerivedPolicy,
  typename InputIterator,
  typename Predicate>
__host__ __device__ thrust::iterator_traits< InputIterator >::difference_
↪type
thrust::count_if(const thrust::detail::execution_policy_base<
↪DerivedPolicy > & exec,
  InputIterator first,
  InputIterator last,
  Predicate pred);

template <typename InputIterator,
  typename Predicate>
thrust::iterator_traits< InputIterator >::difference_type
thrust::count_if(InputIterator first,
  InputIterator last,
  Predicate pred);
```

thrust::count

```
template <typename DerivedPolicy,
  typename InputIterator,
  typename EqualityComparable>
__host__ __device__ thrust::iterator_traits< InputIterator >::difference_
↪type
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
count(const thrust::detail::execution_policy_base< DerivedPolicy > & exec,
  InputIterator first,
  InputIterator last,
  const EqualityComparable & value);
```

count 查找 [first,last) 内等于 value 的元素数量。具体而言，count 返回 [first, last) 内满足 *i == value 的迭代器 i 的数量。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::device 执行策略，使用 count 计算感兴趣值范围内的实例数量：

```
#include <thrust/count.h>
#include <thrust/device_vector.h>
#include <thrust/execution_policy.h>
...
// put 3 1s in a device_vector
thrust::device_vector<int> vec(5,0);
vec[1] = 1;
vec[3] = 1;
vec[4] = 1;

// count the 1s
int result = thrust::count(thrust::device, vec.begin(), vec.end(), 1);
// result == 3
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **InputIterator**：必须是 Input Iterator 的模型，InputIterator 的 value_type 必须是 Equality Comparable 的模型
- **EqualityComparable**：必须是 Equality Comparable 的模型，并且可以和 InputIterator 的 value_type 进行相等性比较

函数参数

- **exec**：用于并行化的执行策略
- **first**：序列起始
- **last**：序列末尾
- **value**：待计数的值

返回值

等于 value 的元素数量

另请参阅

https://en.cppreference.com/w/cpp/algorithm/count

thrust::count

```
template <typename InputIterator,
  typename EqualityComparable> thrust::iterator_traits< InputIterator >
↪::difference_type
count(InputIterator first,
  InputIterator last,
  const EqualityComparable & value);
```

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

count 查找 [first,last) 内等于 value 的元素数量。具体而言，count 返回 [first, last) 内满足 *i == value 的迭代器 i 的数量。

以下代码片段演示了如何使用 count 计算感兴趣值范围内的实例数量：

```
#include <thrust/count.h>
#include <thrust/device_vector.h>
...
// put 3 1s in a device_vector
thrust::device_vector<int> vec(5,0);
vec[1] = 1;
vec[3] = 1;
vec[4] = 1;

// count the 1s
int result = thrust::count(vec.begin(), vec.end(), 1);
// result == 3
```

模板参数

- **InputIterator**：必须是 Input Iterator 的模型，InputIterator 的 value_type 必须是 Equality Comparable 的模型
- **EqualityComparable**：必须是 Equality Comparable 的模型，并且可以和 InputIterator 的 value_type 进行相等性比较

函数参数

- **first**：序列起始
- **last**：序列末尾
- **value**：待计数的值

返回值

等于 value 的元素数量

另请参阅

https://en.cppreference.com/w/cpp/algorithm/count

thrust::count_if

```
template <typename DerivedPolicy,
  typename InputIterator,
  typename Predicate>
__host__ __device__ thrust::iterator_traits< InputIterator >::difference_
↪type
count_if(const thrust::detail::execution_policy_base< DerivedPolicy > &
↪exec,
  InputIterator first,
  InputIterator last,
  Predicate pred);
```

count_if 查找 [first,last) 内其谓词为 true 的元素数量。具体而言，count_if 返回 [first, last) 内满足 pred(*i) == true 的迭代器 i 的数量。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::device 执行策略，使用 count 计算一个范围内的奇数数量：

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

```
#include <thrust/count.h>
#include <thrust/device_vector.h>
#include <thrust/execution_policy.h>
...
struct is_odd
{
__host__ __device__
bool operator()(int &x)
{
   return x & 1;
}
};
...
// fill a device_vector with even & odd numbers
thrust::device_vector<int> vec(5);
vec[0] = 0;
vec[1] = 1;
vec[2] = 2;
vec[3] = 3;
vec[4] = 4;

// count the odd elements in vec
int result = thrust::count_if(thrust::device, vec.begin(), vec.end(), is_
↪odd());
// result == 2
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **InputIterator**：必须是 Input Iterator 的模型，InputIterator 的 value_type 必须可转换为 Predicate 的 argument_type
- **Predicate**：必须是 Predicate 的模型

函数参数

- **exec**：用于并行化的执行策略
- **first**：序列起始
- **last**：序列末尾
- **pred**：谓词

返回值

pred 为 true 的元素数量

另请参阅

https://en.cppreference.com/w/cpp/algorithm/count

thrust::count_if

```
template <typename InputIterator,
  typename Predicate>
thrust::iterator_traits< InputIterator >::difference_type
count_if(InputIterator first,
  InputIterator last,
  Predicate pred);
```

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

count_if 查找 [first,last) 内其谓词为 true 的元素数量。具体而言，count_if 返回 [first, last) 内满足 pred(*i) == true 的迭代器 i 的数量。

以下代码片段演示了如何使用 count 计算一个范围内的奇数数量：

```
#include <thrust/count.h>
#include <thrust/device_vector.h>
...
struct is_odd
{
__host__ __device__
bool operator()(int &x)
{
   return x & 1;
}
};
...
// fill a device_vector with even & odd numbers
thrust::device_vector<int> vec(5);
vec[0] = 0;
vec[1] = 1;
vec[2] = 2;
vec[3] = 3;
vec[4] = 4;

// count the odd elements in vec
int result = thrust::count_if(vec.begin(), vec.end(), is_odd());
// result == 2
```

模板参数

- **InputIterator**：必须是 Input Iterator 的模型，InputIterator 的 value_type 必须可转换为 Predicate 的 argument_type
- **Predicate**：必须是 Predicate 的模型

函数参数

- **first**：序列起始
- **last**：序列末尾
- **pred**：谓词

返回值

pred 为 true 的元素数量

另请参阅

https://en.cppreference.com/w/cpp/algorithm/count

2.1.4.4 最值（Extrema）

函数

```
template <typename DerivedPolicy,
  typename ForwardIterator>
__host__ __device__ ForwardIterator
thrust::min_element(const thrust::detail::execution_policy_base<
↪DerivedPolicy > & exec,
  ForwardIterator first,
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
  ForwardIterator last);

template <typename ForwardIterator>
ForwardIterator
thrust::min_element(ForwardIterator first,
  ForwardIterator last);

template <typename DerivedPolicy,
  typename ForwardIterator,
  typename BinaryPredicate>
__host__ __device__ ForwardIterator
thrust::min_element(const thrust::detail::execution_policy_base<
 ↪DerivedPolicy > & exec,
  ForwardIterator first,
  ForwardIterator last,
  BinaryPredicate comp);

template <typename ForwardIterator,
  typename BinaryPredicate>
ForwardIterator
thrust::min_element(ForwardIterator first,
  ForwardIterator last,
  BinaryPredicate comp);

template <typename DerivedPolicy,
  typename ForwardIterator>
__host__ __device__ ForwardIterator
thrust::max_element(const thrust::detail::execution_policy_base<
 ↪DerivedPolicy > & exec,
  ForwardIterator first,
  ForwardIterator last);

template <typename ForwardIterator>
ForwardIterator
thrust::max_element(ForwardIterator first,
  ForwardIterator last);

template <typename DerivedPolicy,
  typename ForwardIterator,
  typename BinaryPredicate>
__host__ __device__
ForwardIterator
thrust::max_element(const thrust::detail::execution_policy_base<
 ↪DerivedPolicy > & exec,
  ForwardIterator first,
  ForwardIterator last,
  BinaryPredicate comp);

template <typename ForwardIterator,
  typename BinaryPredicate>
ForwardIterator
thrust::max_element(ForwardIterator first,
  ForwardIterator last,
  BinaryPredicate comp);

template <typename DerivedPolicy,
```

(下页继续)

飞燕信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
  typename ForwardIterator>
__host__ __device__ thrust::pair< ForwardIterator, ForwardIterator >
thrust::minmax_element(const thrust::detail::execution_policy_base<
↪DerivedPolicy > & exec,
  ForwardIterator first,
  ForwardIterator last);

template <typename ForwardIterator>
thrust::pair< ForwardIterator, ForwardIterator >
thrust::minmax_element(ForwardIterator first,
  ForwardIterator last);

template <typename DerivedPolicy,
  typename ForwardIterator,
  typename BinaryPredicate>
__host__ __device__ thrust::pair< ForwardIterator, ForwardIterator >
thrust::minmax_element(const thrust::detail::execution_policy_base<
↪DerivedPolicy > & exec,
  ForwardIterator first,
  ForwardIterator last,
  BinaryPredicate comp);

template <typename ForwardIterator,
  typename BinaryPredicate>
thrust::pair< ForwardIterator, ForwardIterator >
thrust::minmax_element(ForwardIterator first,
  ForwardIterator last,
  BinaryPredicate comp);
```

thrust::min_element

```
template <typename DerivedPolicy,
  typename ForwardIterator>
__host__ __device__ ForwardIterator
min_element(const thrust::detail::execution_policy_base< DerivedPolicy > &
↪exec,
  ForwardIterator first,
  ForwardIterator last);
```

min_element 查找范围 [first, last) 内最小的元素。返回 [first, last) 内第一个迭代器 i，使得 [first, last) 内没有迭代器指向小于 *i 的值。当且仅当 [first, last) 为空范围，返回值为 last。

min_element 的两个版本在定义一个元素是否小于另一个元素方面有所不同。此版本使用 operator< 比较对象。具体而言，此版本的 min_element 返回 [first, last) 内的第一个迭代器 i，因此，对于 [first, last) 内第一个迭代器 j，*j < *i 为 false。

算法的执行由 exec 确定并行化。

```
#include <thrust/extrema.h>
#include <thrust/execution_policy.h>
...
int data[6] = {1, 0, 2, 2, 1, 3};
int *result = thrust::min_element(thrust::host, data, data + 6);

// result is data + 1
// *result is 0
```

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

模板参数

ForwardIterator：是 Forward Iterator 的模型，ForwardIterator 的 value_type 是 LessThan Comparable 的模型

函数参数

- **exec**：用于并行化的执行策略
- **first**：序列起始
- **last**：序列末尾

返回值

如果 [first, last) 不是空范围，返回指向范围内最小元素的迭代器；否则，返回 last

另请参阅

https://en.cppreference.com/w/cpp/algorithm/min_element

thrust::min_element

```
template <typename ForwardIterator>
ForwardIterator
min_element(ForwardIterator first,
  ForwardIterator last);
```

min_element 查找范围 [first, last) 内最小的元素。返回 [first, last) 内第一个迭代器 i ，使得 [first, last) 内没有迭代器指向小于 *i 的值。当且仅当 [first, last) 为空范围，返回值为 last。

min_element 的两个版本在定义一个元素是否小于另一个元素方面有所不同。此版本使用 operator< 比较对象。具体而言，此版本的 min_element 返回 [first, last) 内的第一个迭代器 i ，因此，对于 [first, last) 内第一个迭代器 j ，*j < *i 为 false。

```
#include <thrust/extrema.h>
...
int data[6] = {1, 0, 2, 2, 1, 3};
int *result = thrust::min_element(data, data + 6);

// result is data + 1
// *result is 0
```

模板参数

ForwardIterator：是 Forward Iterator 的模型，ForwardIterator 的 value_type 是 LessThan Comparable 的模型

函数参数

- **first**：序列起始
- **last**：序列末尾

返回值

如果 [first, last) 不是空范围，返回指向范围内最小元素的迭代器；否则，返回 last

另请参阅

https://en.cppreference.com/w/cpp/algorithm/min_element

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

thrust::min_element

```
template <typename DerivedPolicy,
  typename ForwardIterator,
  typename BinaryPredicate>
__host__ __device__ ForwardIterator
min_element(const thrust::detail::execution_policy_base< DerivedPolicy > &
↪exec,
  ForwardIterator first,
  ForwardIterator last,
  BinaryPredicate comp);
```

min_element 查找范围 [first, last) 内最小的元素。返回 [first, last) 内第一个迭代器 i，使得 [first, last) 内没有迭代器指向小于 *i 的值。当且仅当 [first, last) 为空范围，返回值为 last。

min_element 的两个版本在定义一个元素是否小于另一个元素方面有所不同。此版本使用函数对象 comp 比较对象。具体而言，此版本的 min_element 返回 [first, last) 内的第一个迭代器 i，因此，对于 [first, last) 内第一个迭代器 j，comp(*j, *i) 为 false。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::host 并行化执行策略，使用 min_element 查找键值对集合中的最小元素：

```
#include <thrust/extrema.h>
#include <thrust/execution_policy.h>
...

struct key_value
{
int key;
int value;
};

struct compare_key_value
{
__host__ __device__
bool operator()(key_value lhs, key_value rhs)
{
  return lhs.key < rhs.key;
}
};

...
key_value data[4] = { {4,5}, {0,7}, {2,3}, {6,1} };

key_value *smallest = thrust::min_element(thrust::host, data, data + 4,
↪compare_key_value());

// smallest == data + 1
// *smallest == {0,7}
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **ForwardIterator**：是 Forward Iterator 的模型，ForwardIterator 的 value_type 可转换为 comp 的 first_argument_type 和 second_argument_type
- **BinaryPredicate**：是 Binary Predicate 的模型

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

函数参数

- **exec**：用于并行化的执行策略
- **first**：序列起始
- **last**：序列末尾
- **comp**：用于比较的二元谓词

返回值

如果 [first, last) 不是空范围，返回指向范围内最小元素的迭代器；否则，返回 last

另请参阅

https://en.cppreference.com/w/cpp/algorithm/min_element

thrust::min_element

```
template <typename ForwardIterator,
  typename BinaryPredicate>
ForwardIterator
min_element(ForwardIterator first,
  ForwardIterator last,
  BinaryPredicate comp);
```

min_element 查找范围 [first, last) 内最小的元素。返回 [first, last) 内第一个迭代器 i，使得 [first, last) 内没有迭代器指向小于 *i 的值。当且仅当 [first, last) 为空范围，返回值为 last。

min_element 的两个版本在定义一个元素是否小于另一个元素方面有所不同。此版本使用函数对象 comp 比较对象。具体而言，此版本的 min_element 返回 [first, last) 内的第一个迭代器 i，因此，对于 [first, last) 内第一个迭代器 j，comp(*j, *i) 为 false。

以下代码片段演示了如何使用 min_element 查找键值对集合中的最小元素：

```
#include <thrust/extrema.h>

struct key_value
{
int key;
int value;
};

struct compare_key_value
{
__host__ __device__
bool operator()(key_value lhs, key_value rhs)
{
   return lhs.key < rhs.key;
}
};

...
key_value data[4] = { {4,5}, {0,7}, {2,3}, {6,1} };

key_value *smallest = thrust::min_element(data, data + 4, compare_key_
↪value());

// smallest == data + 1
// *smallest == {0,7}
```

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

模板参数

- **ForwardIterator**：是 Forward Iterator 的模型，ForwardIterator 的 value_type 可转换为 comp 的 first_argument_type 和 second_argument_type
- **BinaryPredicate**：是 Binary Predicate 的模型

函数参数

- **first**：序列起始
- **last**：序列末尾
- **comp**：用于比较的二元谓词

返回值

如果 [first, last) 不是空范围，返回指向范围内最小元素的迭代器；否则，返回 last

另请参阅

https://en.cppreference.com/w/cpp/algorithm/min_element

thrust::max_element

```
template <typename DerivedPolicy,
  typename ForwardIterator>
__host__ __device__ ForwardIterator
max_element(const thrust::detail::execution_policy_base< DerivedPolicy > &
↪exec,
  ForwardIterator first,
  ForwardIterator last);
```

max_element 查找范围 [first, last) 内最大的元素。返回 [first, last) 内第一个迭代器 i，使得 [first, last) 内没有迭代器指向大于 *i 的值。当且仅当 [first, last) 为空范围，返回值为 last。

max_element 的两个版本不同之处在于它们如何定义一个元素是否大于另一个元素。此版本使用 operator< 比较对象。具体而言，此版本的 max_element 返回 [first, last) 内的第一个迭代器 i，因此，对于 [first, last) 内第一个迭代器 j，*i < *j 为 false。

算法的执行由 exec 确定并行化。

```
#include <thrust/extrema.h>
#include <thrust/execution_policy.h>
...
int data[6] = {1, 0, 2, 2, 1, 3};
int *result = thrust::max_element(thrust::host, data, data + 6);

// *result == 3
```

模板参数

- **DerivedPolicy**：Thrust 后端系统
- **ForwardIterator**：是 Forward Iterator 的模型，ForwardIterator 的 value_type 是 LessThan Comparable 的模型

函数参数

- **exec**：用于并行化的执行策略
- **first**：序列起始
- **last**：序列末尾

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

返回值

如果 [first, last) 不是空范围，返回指向范围内最大元素的迭代器；否则，返回 last

另请参阅

https://en.cppreference.com/w/cpp/algorithm/max_element

thrust::max_element

```
template <typename ForwardIterator>
ForwardIterator
max_element(ForwardIterator first,
  ForwardIterator last);
```

max_element 查找范围 [first, last) 内最大的元素。返回 [first, last) 内第一个迭代器 i，使得 [first, last) 内没有迭代器指向大于 *i 的值。当且仅当 [first, last) 为空范围，返回值为 last。

max_element 的两个版本不同之处在于它们如何定义一个元素是否大于另一个元素。此版本使用 operator< 比较对象。具体而言，此版本的 max_element 返回 [first, last) 内的第一个迭代器 i，因此，对于 [first, last) 内第一个迭代器 j，*i < *j 为 false。

```
#include <thrust/extrema.h>
...
int data[6] = {1, 0, 2, 2, 1, 3};
int *result = thrust::max_element(data, data + 6);

// *result == 3
```

模板参数

ForwardIterator：是 Forward Iterator 的模型，ForwardIterator 的 value_type 是 LessThan Comparable 的模型

函数参数

- **first**：序列起始
- **last**：序列末尾

返回值

如果 [first, last) 不是空范围，返回指向范围内最大元素的迭代器；否则，返回 last

另请参阅

https://en.cppreference.com/w/cpp/algorithm/max_element

thrust::max_element

```
template <typename DerivedPolicy,
  typename ForwardIterator,
  typename BinaryPredicate>
__host__ __device__ ForwardIterator
max_element(const thrust::detail::execution_policy_base< DerivedPolicy > &
↪exec,
  ForwardIterator first,
  ForwardIterator last,
  BinaryPredicate comp);
```

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

max_element 查找范围 [first, last) 内最大的元素。返回 [first, last) 内第一个迭代器 i ，使得 [first, last) 内没有迭代器指向大于 *i 的值。当且仅当 [first, last) 为空范围，返回值为 last。

max_element 的两个版本在定义一个元素是否小于另一个元素方面有所不同。此版本使用函数对象 comp 比较对象。具体而言，此版本的 max_element 返回 [first, last) 内的第一个迭代器 i ，因此，对于 [first, last) 内第一个迭代器 j ，comp(*i, *j) 为 false。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::host 并行化执行策略，使用 max_element 查找键值对集合中的最大元素：

```
#include <thrust/extrema.h>
#include <thrust/execution_policy.h>
...

struct key_value
{
int key;
int value;
};

struct compare_key_value
{
__host__ __device__
bool operator()(key_value lhs, key_value rhs)
{
   return lhs.key < rhs.key;
}
};

...
key_value data[4] = { {4,5}, {0,7}, {2,3}, {6,1} };

key_value *largest = thrust::max_element(thrust::host, data, data + 4,
↪compare_key_value());

// largest == data + 3
// *largest == {6,1}
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **ForwardIterator**：是 Forward Iterator 的模型，ForwardIterator 的 value_type 可转换为 comp 的 first_argument_type 和 second_argument_type
- **BinaryPredicate**：是 Binary Predicate 的模型

函数参数

- **exec**：用于并行化的执行策略
- **first**：序列起始
- **last**：序列末尾
- **comp**：用于比较的二元谓词

返回值

如果 [first, last) 不是空范围，返回指向范围内最大元素的迭代器；否则，返回 last

另请参阅

https://en.cppreference.com/w/cpp/algorithm/max_element

飞腾信息 MetaX Confident 2024-11-18 15:00:00

thrust::max_element

```
template <typename ForwardIterator,
  typename BinaryPredicate>
ForwardIterator
max_element(ForwardIterator first,
  ForwardIterator last,
  BinaryPredicate comp);
```

max_element 查找范围 [first, last) 内最大的元素。返回 [first, last) 内第一个迭代器 i，使得 [first, last) 内没有迭代器指向大于 *i 的值。当且仅当 [first, last) 为空范围，返回值为 last。

max_element 的两个版本在定义一个元素是否小于另一个元素方面有所不同。此版本使用函数对象 comp 比较对象。具体而言，此版本的 max_element 返回 [first, last) 内的第一个迭代器 i，因此，对于 [first, last) 内第一个迭代器 j，comp(*i, *j) 为 false。

以下代码片段演示了如何使用 max_element 查找键值对集合中的最大元素：

```
#include <thrust/extrema.h>

struct key_value
{
int key;
int value;
};

struct compare_key_value
{
__host__ __device__
bool operator()(key_value lhs, key_value rhs)
{
   return lhs.key < rhs.key;
}
};

...
key_value data[4] = { {4,5}, {0,7}, {2,3}, {6,1} };

key_value *largest = thrust::max_element(data, data + 4, compare_key_
↪value());

// largest == data + 3
// *largest == {6,1}
```

模板参数

- **ForwardIterator**：是 Forward Iterator 的模型，ForwardIterator 的 value_type 可转换为 comp 的 first_argument_type 和 second_argument_type
- **BinaryPredicate**：是 Binary Predicate 的模型

函数参数

- **first**：序列起始
- **last**：序列末尾
- **comp**：用于比较的二元谓词

返回值

如果 [first, last) 不是空范围，返回指向范围内最大元素的迭代器；否则，返回 last

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

另请参阅

https://en.cppreference.com/w/cpp/algorithm/max_element

thrust::minmax_element

```
template <typename DerivedPolicy,
  typename ForwardIterator>
__host__ __device__ thrust::pair< ForwardIterator, ForwardIterator >
minmax_element(const thrust::detail::execution_policy_base< DerivedPolicy >
 ↪ & exec,
  ForwardIterator first,
  ForwardIterator last);
```

minmax_element 查找范围 [first, last) 内最小和最大的元素。返回一对迭代器 (imin, imax)，其中 imin 与 min_element 返回的迭代器相同；imax 与 max_element 返回的迭代器相同。此函数可能比单独调用 min_element 和 max_element 更高效。

算法的执行由 exec 确定并行化。

```
#include <thrust/extrema.h>
#include <thrust/execution_policy.h>
...
int data[6] = {1, 0, 2, 2, 1, 3};
thrust::pair<int *, int *> result = thrust::minmax_element(thrust::host,
 ↪data, data + 6);

// result.first is data + 1
// result.second is data + 5
// *result.first is 0
// *result.second is 3
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **ForwardIterator**：是 Forward Iterator 的模型，ForwardIterator 的 value_type 是 LessThan Comparable 的模型

函数参数

- **exec**：用于并行化的执行策略
- **first**：序列起始
- **last**：序列末尾

返回值

如果 [first, last) 不是空范围，返回指向范围内最小和最大元素的一对迭代器；否则，返回 last

另请参阅

- min_element
- max_element
- http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1840.pdf

飞诺信息 MetaX Confidential 2024-11-18 15:00:00

thrust::minmax_element

```
template <typename ForwardIterator>
thrust::pair< ForwardIterator, ForwardIterator >
minmax_element(ForwardIterator first,
  ForwardIterator last);
```

minmax_element 查找范围 [first, last) 内最小和最大的元素。返回一对迭代器 (imin, imax)，其中 imin 与 min_element 返回的迭代器相同；imax 与 max_element 返回的迭代器相同。此函数可能比单独调用 min_element 和 max_element 更高效。

```
#include <thrust/extrema.h>
...
int data[6] = {1, 0, 2, 2, 1, 3};
thrust::pair<int *, int *> result = thrust::minmax_element(data, data + 6);

// result.first is data + 1
// result.second is data + 5
// *result.first is 0
// *result.second is 3
```

模板参数

ForwardIterator：是 Forward Iterator 的模型，ForwardIterator 的 value_type 是 LessThan Comparable 的模型

函数参数

- **first**：序列起始
- **last**：序列末尾

返回值

如果 [first, last) 不是空范围，返回指向范围内最小和最大元素的一对迭代器；否则，返回 last

另请参阅

- min_element
- max_element
- http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1840.pdf

thrust::minmax_element

```
template <typename DerivedPolicy,
  typename ForwardIterator,
  typename BinaryPredicate>
__host__ __device__ thrust::pair< ForwardIterator, ForwardIterator >
minmax_element(const thrust::detail::execution_policy_base< DerivedPolicy >
↪ & exec,
  ForwardIterator first,
  ForwardIterator last,
  BinaryPredicate comp);
```

minmax_element 查找范围 [first, last) 内最小和最大的元素。返回一对迭代器 (imin, imax)，其中 imin 与 min_element 返回的迭代器相同；imax 与 max_element 返回的迭代器相同。此函数可能比单独调用 min_element 和 max_element 更高效。

算法的执行由 exec 确定并行化。

飞曦信息 MetaX Confidential 2024-11-18 15:00:00

以下代码片段演示了如何通过 thrust::host 并行化执行策略，使用 minmax_element 查找键值对集合中的最小和最大元素：

```
#include <thrust/extrema.h>
#include <thrust/pair.h>
#include <thrust/execution_policy.h>
...

struct key_value
{
int key;
int value;
};

struct compare_key_value
{
__host__ __device__
bool operator()(key_value lhs, key_value rhs)
{
   return lhs.key < rhs.key;
}
};

...
key_value data[4] = { {4,5}, {0,7}, {2,3}, {6,1} };

thrust::pair<key_value*,key_value*> extrema = thrust::minmax_
↪element(thrust::host, data, data + 4, compare_key_value());

// extrema.first  == data + 1
// *extrema.first  == {0,7}
// extrema.second  == data + 3
// *extrema.second == {6,1}
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **ForwardIterator**：是 Forward Iterator 的模型，ForwardIterator 的 value_type 可转换为 comp 的 first_argument_type 和 second_argument_type
- **BinaryPredicate**：是 Binary Predicate 的模型

函数参数

- **exec**：用于并行化的执行策略
- **first**：序列起始
- **last**：序列末尾
- **comp**：用于比较的二元谓词

返回值

如果 [first, last) 不是空范围，返回指向范围内最小和最大元素的一对迭代器；否则，返回 last

另请参阅

- min_element
- max_element
- http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1840.pdf

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

thrust::minmax_element

```
template <typename ForwardIterator,
  typename BinaryPredicate>
thrust::pair< ForwardIterator, ForwardIterator >
minmax_element(ForwardIterator first,
  ForwardIterator last,
  BinaryPredicate comp);
```

minmax_element 查找范围 [first, last) 内最小和最大的元素。返回一对迭代器 (imin, imax)，其中 imin 与 min_element 返回的迭代器相同；imax 与 max_element 返回的迭代器相同。此函数可能比单独调用 min_element 和 max_element 更高效。

以下代码片段演示了如何使用 minmax_element 查找键值对集合中的最小和最大元素：

```
#include <thrust/extrema.h>
#include <thrust/pair.h>

struct key_value
{
int key;
int value;
};

struct compare_key_value
{
__host__ __device__
bool operator()(key_value lhs, key_value rhs)
{
   return lhs.key < rhs.key;
}
};

...
key_value data[4] = { {4,5}, {0,7}, {2,3}, {6,1} };

thrust::pair<key_value*,key_value*> extrema = thrust::minmax_element(data,
↪data + 4, compare_key_value());

// extrema.first   == data + 1
// *extrema.first  == {0,7}
// extrema.second  == data + 3
// *extrema.second == {6,1}
```

模板参数

- **ForwardIterator**：是 Forward Iterator 的模型，ForwardIterator 的 value_type 可转换为 comp 的 first_argument_type 和 second_argument_type
- **BinaryPredicate**：是 Binary Predicate 的模型

函数参数

- **first**：序列起始
- **last**：序列末尾
- **comp**：用于比较的二元谓词

返回值

如果 [first, last) 不是空范围，返回指向范围内最小和最大元素的一对迭代器；否则，返回 last

飞猪信息 MetaX Confidential 2024-11-18 15:00:00

另请参阅

- min_element
- max_element
- http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1840.pdf

2.1.4.5 逻辑（Logical）

逻辑

```
template <typename DerivedPolicy,
  typename InputIterator,
  typename Predicate>
__host__ __device__ bool
thrust::all_of(const thrust::detail::execution_policy_base< DerivedPolicy >
 ↪ & exec,
  InputIterator first,
  InputIterator last,
 Predicate pred);

template <typename InputIterator,
  typename Predicate>
bool
thrust::all_of(InputIterator first,
  InputIterator last,
  Predicate pred);

template <typename DerivedPolicy,
  typename InputIterator,
  typename Predicate>
__host__ __device__ bool
thrust::any_of(const thrust::detail::execution_policy_base< DerivedPolicy >
 ↪ & exec,
  InputIterator first,
  InputIterator last,
  Predicate pred);

template <typename InputIterator,
  typename Predicate>
bool
thrust::any_of(InputIterator first,
  InputIterator last,
  Predicate pred);

template <typename DerivedPolicy,
  typename InputIterator,
  typename Predicate>
__host__ __device__ bool
thrust::none_of(const thrust::detail::execution_policy_base< DerivedPolicy
 ↪> & exec,
  InputIterator first,
  InputIterator last,
  Predicate pred);

template <typename InputIterator,
  typename Predicate>
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
bool
thrust::none_of(InputIterator first,
  InputIterator last,
  Predicate pred);
```

thrust::all_of

```
template <typename DerivedPolicy,
  typename InputIterator,
  typename Predicate>
__host__ __device__ bool
all_of(const thrust::detail::execution_policy_base< DerivedPolicy > & exec,
  InputIterator first,
  InputIterator last,
  Predicate pred);
```

all_of 确定范围内所有元素是否满足一个谓词。具体而言，如果对于范围 [first, last) 内每个迭代器 i，pred(*i) 为 true，则 all_of 返回 true；否则返回 false。

算法的执行由 exec 确定并行化。

```
#include <thrust/logical.h>
#include <thrust/functional.h>
#include <thrust/execution_policy.h>
...
bool A[3] = {true, true, false};

thrust::all_of(thrust::host, A, A + 2, thrust::identity<bool>()); //
↪returns true
thrust::all_of(thrust::host, A, A + 3, thrust::identity<bool>()); //
↪returns false

// empty range
thrust::all_of(thrust::host, A, A, thrust::identity<bool>()); // returns
↪false
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **InputIterator**：是 Input Iterator 的模型
- **Predicate**：必须是 Predicate 的模型

函数参数

- **exec**：用于并行化的执行策略
- **first**：序列起始
- **last**：序列末尾
- **pred**：用于测试范围内元素的谓词

返回值

如果所有元素满足谓词，则返回 true；否则返回 false

另请参阅

- any_of

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

- none_of
- transform_reduce

thrust::all_of

```
template <typename InputIterator,
  typename Predicate>
bool
all_of(InputIterator first,
  InputIterator last,
  Predicate pred);
```

all_of 确定范围内所有元素是否满足一个谓词。具体而言，如果对于范围 [first, last) 内每个迭代器 i，pred(*i) 为 true，则 all_of 返回 true；否则返回 false。

```
#include <thrust/logical.h>
#include <thrust/functional.h>
...
bool A[3] = {true, true, false};

thrust::all_of(A, A + 2, thrust::identity<bool>()); // returns true
thrust::all_of(A, A + 3, thrust::identity<bool>()); // returns false

// empty range
thrust::all_of(A, A, thrust::identity<bool>()); // returns false
```

模板参数

- **InputIterator**：是 Input Iterator 的模型
- **Predicate**：必须是 Predicate 的模型

函数参数

- **first**：序列起始
- **last**：序列末尾
- **pred**：用于测试范围内元素的谓词

返回值

如果所有元素满足谓词，则返回 true；否则返回 false

另请参阅

- any_of
- none_of
- transform_reduce

thrust::any_of

```
template <typename DerivedPolicy,
  typename InputIterator,
  typename Predicate>
__host__ __device__ bool
any_of(const thrust::detail::execution_policy_base< DerivedPolicy > & exec,
  InputIterator first,
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
  InputIterator last,
  Predicate pred);
```

any_of 确定范围内是否有元素满足一个谓词。具体而言，如果对于范围 [first, last) 内任意一个迭代器 i，pred(*i) 为 true，则 any_of 返回 true；否则返回 false。

算法的执行由 exec 确定并行化。

```
#include <thrust/logical.h>
#include <thrust/functional.h>
#include <thrust/execution_policy.h>
...
bool A[3] = {true, true, false};

thrust::any_of(thrust::host, A, A + 2, thrust::identity<bool>()); //
↪returns true
thrust::any_of(thrust::host, A, A + 3, thrust::identity<bool>()); //
↪returns true

thrust::any_of(thrust::host, A + 2, A + 3, thrust::identity<bool>()); //
↪returns false

// empty range
thrust::any_of(thrust::host, A, A, thrust::identity<bool>()); // returns
↪false
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **InputIterator**：是 Input Iterator 的模型
- **Predicate**：必须是 Predicate 的模型

函数参数

- **exec**：用于并行化的执行策略
- **first**：序列起始
- **last**：序列末尾
- **pred**：用于测试范围内元素的谓词

返回值

如果有元素满足谓词，则返回 true；否则返回 false

另请参阅

- all_of
- none_of
- transform_reduce

thrust::any_of

```
template <typename InputIterator,
  typename Predicate>
bool
any_of(InputIterator first,
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
  InputIterator last,
  Predicate pred);
```

any_of 确定范围内是否有元素满足一个谓词。具体而言，如果对于范围 [first, last) 内任意一个迭代器 i，pred(*i) 为 true，则 any_of 返回 true；否则返回 false。

```
#include <thrust/logical.h>
#include <thrust/functional.h>
...
bool A[3] = {true, true, false};

thrust::any_of(A, A + 2, thrust::identity<bool>()); // returns true
thrust::any_of(A, A + 3, thrust::identity<bool>()); // returns true

thrust::any_of(A + 2, A + 3, thrust::identity<bool>()); // returns false

// empty range
thrust::any_of(A, A, thrust::identity<bool>()); // returns false
```

模板参数

- **InputIterator**：是 Input Iterator 的模型
- **Predicate**：必须是 Predicate 的模型

函数参数

- **first**：序列起始
- **last**：序列末尾
- **pred**：用于测试范围内元素的谓词

返回值

如果有元素满足谓词，则返回 true；否则返回 false

另请参阅

- all_of
- none_of
- transform_reduce

thrust::none_of

```
template <typename DerivedPolicy,
  typename InputIterator,
  typename Predicate>
__host__ __device__ bool
none_of(const thrust::detail::execution_policy_base< DerivedPolicy > &
↪exec,
  InputIterator first,
  InputIterator last,
  Predicate pred);
```

none_of 确定范围内元素是否都不满足一个谓词。具体而言，如果对于范围 [first, last) 内的迭代器 i，pred(*i) 不为 true，则 none_of 返回 true；否则返回 false。

算法的执行由 exec 确定并行化。

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

```
#include <thrust/logical.h>
#include <thrust/functional.h>
#include <thrust/execution_policy.h>
...
bool A[3] = {true, true, false};

thrust::none_of(thrust::host, A, A + 2, thrust::identity<bool>()); //
↪returns false
thrust::none_of(thrust::host, A, A + 3, thrust::identity<bool>()); //
↪returns false

thrust::none_of(thrust::host, A + 2, A + 3, thrust::identity<bool>()); //
↪returns true

// empty range
thrust::none_of(thrust::host, A, A, thrust::identity<bool>()); // returns
↪true
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **InputIterator**：是 Input Iterator 的模型
- **Predicate**：必须是 Predicate 的模型

函数参数

- **exec**：用于并行化的执行策略
- **first**：序列起始
- **last**：序列末尾
- **pred**：用于测试范围内元素的谓词

返回值

如果没有元素满足谓词，则返回 true；否则返回 false

另请参阅

- all_of
- any_of
- transform_reduce

thrust::none_of

```
template <typename InputIterator,
  typename Predicate>
bool
none_of(InputIterator first,
  InputIterator last,
  Predicate pred);
```

none_of 确定范围内元素是否都不满足一个谓词。具体而言，如果对于范围 [first, last) 内的迭代器 i，pred(*i) 不为 true，则 none_of 返回 true；否则返回 false。

```
#include <thrust/logical.h>
#include <thrust/functional.h>
```

(下页继续)

飞腾信息 MetaX confidential 2024-11-18 15:00:00

(续上页)

```
...
bool A[3] = {true, true, false};

thrust::none_of(A, A + 2, thrust::identity<bool>()); // returns false
thrust::none_of(A, A + 3, thrust::identity<bool>()); // returns false

thrust::none_of(A + 2, A + 3, thrust::identity<bool>()); // returns true

// empty range
thrust::none_of(A, A, thrust::identity<bool>()); // returns true
```

模板参数

- **InputIterator**：是 Input Iterator 的模型
- **Predicate**：必须是 Predicate 的模型

函数参数

- **first**：序列起始
- **last**：序列末尾
- **pred**：用于测试范围内元素的谓词

返回值

如果没有元素满足谓词，则返回 true；否则返回 false

另请参阅

- all_of
- any_of
- transform_reduce

2.1.4.6 谓词（Predicate）

函数

```
template <typename DerivedPolicy,
  typename InputIterator,
  typename Predicate>
__host__ __device__ bool
thrust::is_partitioned(const thrust::detail::execution_policy_base<
 ↪DerivedPolicy > & exec,
  InputIterator first,
  InputIterator last,
  Predicate pred);

template <typename InputIterator,
  typename Predicate>
bool
thrust::is_partitioned(InputIterator first,
  InputIterator last,
  Predicate pred);

template <typename DerivedPolicy,
  typename ForwardIterator>
__host__ __device__ bool
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
thrust::is_sorted(const thrust::detail::execution_policy_base<
 ↪DerivedPolicy > & exec,
  ForwardIterator first,
  ForwardIterator last);

template <typename ForwardIterator>
bool
thrust::is_sorted(ForwardIterator first,
  ForwardIterator last);

template <typename DerivedPolicy,
  typename ForwardIterator,
  typename Compare>
__host__ __device__ bool
thrust::is_sorted(const thrust::detail::execution_policy_base<
 ↪DerivedPolicy > & exec,
  ForwardIterator first,
  ForwardIterator last,
  Compare comp);

template <typename ForwardIterator,
  typename Compare>
bool
thrust::is_sorted(ForwardIterator first,
  ForwardIterator last,
  Compare comp);

template <typename DerivedPolicy,
  typename ForwardIterator>
__host__ __device__ ForwardIterator
thrust::is_sorted_until(const thrust::detail::execution_policy_base<
 ↪DerivedPolicy > & exec,
  ForwardIterator first,
  ForwardIterator last);

template <typename ForwardIterator>
ForwardIterator
thrust::is_sorted_until(ForwardIterator first,
  ForwardIterator last);

template <typename DerivedPolicy,
  typename ForwardIterator,
  typename Compare>
__host__ __device__ ForwardIterator
thrust::is_sorted_until(const thrust::detail::execution_policy_base<
 ↪DerivedPolicy > & exec,
  ForwardIterator first,
  ForwardIterator last,
  Compare comp);

template <typename ForwardIterator,
  typename Compare>
ForwardIterator
thrust::is_sorted_until(ForwardIterator first,
  ForwardIterator last,
  Compare comp);
```

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

thrust::is_partitioned

```
template <typename DerivedPolicy,
  typename InputIterator,
  typename Predicate>
__host__ __device__ bool
is_partitioned(const thrust::detail::execution_policy_base< DerivedPolicy >
↪ & exec,
  InputIterator first,
  InputIterator last,
  Predicate pred);
```

如果给定范围是根据谓词划分的，is_partitioned 返回 true；否则返回 false。

具体而言，如果 [first, last) 为空或 [first, last) 根据 pred 分区，则 is_partitioned 为 true ，即满足 pred 的所有元素都先于不满足 pred 的元素。

算法的执行由 exec 确定并行化。

```
#include <thrust/partition.h>
#include <thrust/execution_policy.h>

struct is_even
{
__host__ __device__
bool operator()(const int &x)
{
   return (x % 2) == 0;
}
};

...

int A[] = {2, 4, 6, 8, 10, 1, 3, 5, 7, 9};
int B[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

thrust::is_partitioned(thrust::host, A, A + 10, is_even()); // returns true
thrust::is_partitioned(thrust::host, B, B + 10, is_even()); // returns
↪false
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **InputIterator**：是 Input Iterator 的模型，InputIterator 的 value_type 可转换为 Predicate 的 argument_type
- **Predicate**：是 Predicate 的模型

函数参数

- **exec**：用于并行化的执行策略
- **first**：待检查的范围的起始
- **last**：待检查的范围的末尾
- **pred**：决定范围 [first, last) 内每个元素属于哪个分区的函数对象

返回值

如果 [first, last) 被 pred 分区或 [first, last) 为空，则返回 true；否则，返回 false

另请参阅

飞谱信息 MetaX Confidential 2024-11-18 15:00:00

partition

thrust::is_partitioned

```
template <typename InputIterator,
  typename Predicate>
bool
is_partitioned(InputIterator first,
  InputIterator last,
  Predicate pred);
```

如果给定范围是根据谓词划分的，is_partitioned 返回 true；否则返回 false。具体而言，如果 [first, last) 为空或 [first, last) 根据 pred 分区，则 is_partitioned 为 true ，即满足 pred 的所有元素都先于不满足 pred 的元素。

```
#include <thrust/partition.h>

struct is_even
{
__host__ __device__
bool operator()(const int &x)
{
   return (x % 2) == 0;
}
};

...

int A[] = {2, 4, 6, 8, 10, 1, 3, 5, 7, 9};
int B[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

thrust::is_partitioned(A, A + 10, is_even()); // returns true
thrust::is_partitioned(B, B + 10, is_even()); // returns false
```

模板参数

- **InputIterator**：是 Input Iterator 的模型，InputIterator 的 value_type 可转换为 Predicate 的 argument_type
- **Predicate**：是 Predicate 的模型

函数参数

- **first**：待检查的范围的起始
- **last**：待检查的范围的末尾
- **pred**：决定范围 [first, last) 内每个元素属于哪个分区的函数对象

返回值

如果 [first, last) 被 pred 分区或 [first, last) 为空，则返回 true；否则，返回 false

另请参阅

partition

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

thrust::is_sorted

```
template <typename DerivedPolicy,
  typename ForwardIterator>
__host__ __device__ bool
is_sorted(const thrust::detail::execution_policy_base< DerivedPolicy > &
↪exec,
  ForwardIterator first,
  ForwardIterator last);
```

如果范围 [first, last) 为升序，is_sorted 返回 true；否则，返回 false。

具体而言，如果对于范围 [first, last - 1) 内的某个迭代器 i ，表达式 *(i + 1) < *i 为 true ，则此版本的 is_sorted 返回 false。

算法的执行由 exec 确定并行化。

下面的代码片段演示了如何通过 thrust::device 并行化执行策略，使用 is_sorted 测试 device_vector 的内容是否按升序存储：

```
#include <thrust/sort.h>
#include <thrust/device_vector.h>
#include <thrust/sort.h>
#include <thrust/execution_policy.h>
...
thrust::device_vector<int> v(6);
v[0] = 1;
v[1] = 4;
v[2] = 2;
v[3] = 8;
v[4] = 5;
v[5] = 7;

bool result = thrust::is_sorted(thrust::device, v.begin(), v.end());

// result == false

thrust::sort(v.begin(), v.end());
result = thrust::is_sorted(thrust::device, v.begin(), v.end());

// result == true
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **ForwardIterator**：是 Forward Iterator 的模型，ForwardIterator 的 value_type 是 LessThan Comparable 的模型，且如在 LessThan Comparable 要求中定义的，ForwardIterator 的 value_type 对象上的排序是 严格弱序

函数参数

- **exec**：用于并行化的执行策略
- **first**：序列起始
- **last**：序列末尾

返回值

如果序列是有序的，返回 true；否则，返回 false

另请参阅

飞猪信息 MetaX Confidential 2024-11-18 15:00:00

- https://en.cppreference.com/w/cpp/algorithm/is_sorted
- is_sorted_until
- sort
- stable_sort
- less<T>

thrust::is_sorted

```
template <typename ForwardIterator> bool is_sorted(ForwardIterator first,
↪  ForwardIterator last);
```

如果范围 [first, last) 为升序，is_sorted 返回 true；否则，返回 false。

具体而言，如果对于范围 [first, last - 1) 内的某个迭代器 i ，表达式 *(i + 1) < *i 为 true ，则此版本的 is_sorted 返回 false。

下面的代码片段演示了如何使用 is_sorted 测试 device_vector 的内容是否按升序排序：

```
#include <thrust/sort.h>
#include <thrust/device_vector.h>
#include <thrust/sort.h>
...
thrust::device_vector<int> v(6);
v[0] = 1;
v[1] = 4;
v[2] = 2;
v[3] = 8;
v[4] = 5;
v[5] = 7;

bool result = thrust::is_sorted(v.begin(), v.end());

// result == false

thrust::sort(v.begin(), v.end());
result = thrust::is_sorted(v.begin(), v.end());

// result == true
```

模板参数

ForwardIterator：是 Forward Iterator 的模型，ForwardIterator 的 value_type 是 LessThan Comparable 的模型，且如在 LessThan Comparable 要求中定义的，ForwardIterator 的 value_type 对象上的排序是 严格弱序

函数参数

- **first**：序列起始
- **last**：序列末尾

返回值

如果序列是有序的，返回 true；否则，返回 false

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/is_sorted
- is_sorted_until

飞曦信息 MetaX Confidential 2024-11-18 15:00:00

- sort
- stable_sort
- less<T>

thrust::is_sorted

```
template <typename DerivedPolicy,
  typename ForwardIterator,
  typename Compare>
__host__ __device__ bool
is_sorted(const thrust::detail::execution_policy_base< DerivedPolicy > &
↪exec,
  ForwardIterator first,
  ForwardIterator last,
  Compare comp);
```

如果根据用户定义的比较运算按升序对范围 [first, last) 进行排序，is_sorted 返回 true；否则，返回 false。

具体而言，如果对于范围 [first, last - 1) 内的某个迭代器 i ，表达式 comp(*(i + 1), *i) 为 true ，则此版本的 is_sorted 返回 false。

算法的执行由 exec 确定并行化。

下面的代码片段演示了如何通过 thrust::device 并行化执行策略，使用 is_sorted 测试 device_vector 的内容是否按降序排序：

```
#include <thrust/sort.h>
#include <thrust/functional.h>
#include <thrust/device_vector.h>
#include <thrust/execution_policy.h>
...
thrust::device_vector<int> v(6);
v[0] = 1;
v[1] = 4;
v[2] = 2;
v[3] = 8;
v[4] = 5;
v[5] = 7;

thrust::greater<int> comp;
bool result = thrust::is_sorted(thrust::device, v.begin(), v.end(), comp);

// result == false

thrust::sort(v.begin(), v.end(), comp);
result = thrust::is_sorted(thrust::device, v.begin(), v.end(), comp);

// result == true
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **ForwardIterator**：是 Forward Iterator 的模型，ForwardIterator 的 value_type 可转换为 StrictWeakOrdering 的 first_argument_type 和 second_argument_type
- **Compare**：是 Strict Weak Ordering 的模型

沐曦信息 MetaX Confidential 2024-11-18 15:00:00

函数参数

- **exec**：用于并行化的执行策略
- **first**：序列起始
- **last**：序列末尾
- **comp**：比较运算符

返回值

如果序列是按照 comp 排序的，返回 true；否则，返回 false

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/is_sorted
- sort
- stable_sort
- less<T>

thrust::is_sorted

```
template <typename ForwardIterator,
  typename Compare>
bool
is_sorted(ForwardIterator first,
  ForwardIterator last,
  Compare comp);
```

如果根据用户定义的比较运算按升序对范围 [first, last) 进行排序，is_sorted 返回 true；否则，返回 false。

具体而言，如果对于范围 [first, last - 1) 内的某个迭代器 i ，表达式 comp(*(i + 1), *i) 为 true ，则此版本的 is_sorted 返回 false。

下面的代码片段演示了如何使用 is_sorted 测试 device_vector 的内容是否按降序排序：

```
#include <thrust/sort.h>
#include <thrust/functional.h>
#include <thrust/device_vector.h>
...
thrust::device_vector<int> v(6);
v[0] = 1;
v[1] = 4;
v[2] = 2;
v[3] = 8;
v[4] = 5;
v[5] = 7;

thrust::greater<int> comp;
bool result = thrust::is_sorted(v.begin(), v.end(), comp);

// result == false

thrust::sort(v.begin(), v.end(), comp);
result = thrust::is_sorted(v.begin(), v.end(), comp);

// result == true
```

飞谱信息 MetaX Confidential 2024-11-18 15:00:00

模板参数

- **ForwardIterator**：是 Forward Iterator 的模型，ForwardIterator 的 value_type 可转换为 StrictWeakOrdering 的 first_argument_type 和 second_argument_type
- **Compare**：是 Strict Weak Ordering 的模型

函数参数

- **first**：序列起始
- **last**：序列末尾
- **comp**：比较运算符

返回值

如果序列是按照 comp 排序的，返回 true；否则，返回 false

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/is_sorted
- sort
- stable_sort
- less<T>

thrust::is_sorted_until

```
template <typename DerivedPolicy,
  typename ForwardIterator>
__host__ __device__ ForwardIterator
is_sorted_until(const thrust::detail::execution_policy_base< DerivedPolicy
↪> & exec,
  ForwardIterator first,
  ForwardIterator last);
```

此版本的 is_sorted_until 返回 [first,last] 内使得 [first,last) 根据 operator< 排序的最后一个迭代器 i。如果 distance(first,last) < 2 ，is_sorted_until 则返回 last。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::host 并行化执行策略，使用 is_sorted_until 查找数组中数据未排序的第一个位置：

```
#include <thrust/sort.h>
#include <thrust/execution_policy.h>

...

int A[8] = {0, 1, 2, 3, 0, 1, 2, 3};

int * B = thrust::is_sorted_until(thrust::host, A, A + 8);

// B - A is 4
// [A, B) is sorted
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **ForwardIterator**：是 Forward Iterator 的模型，ForwardIterator 的 value_type 是 LessThan Comparable 的模型

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

函数参数

- **exec**：用于并行化的执行策略
- **first**：感兴趣范围的起始
- **last**：感兴趣范围的末尾

返回值

输入范围内使其排序的最后一个迭代器

另请参阅

- is_sorted
- sort
- sort_by_key
- stable_sort
- stable_sort_by_key

thrust::is_sorted_until

```
template <typename ForwardIterator>
ForwardIterator
is_sorted_until(ForwardIterator first,
  ForwardIterator last);
```

此版本的 is_sorted_until 返回 [first,last] 内使得 [first,last) 根据 operator< 排序的最后一个迭代器 i。如果 distance(first,last) < 2 ，is_sorted_until 则返回 last。

以下代码片段演示了如何使用 is_sorted_until 查找数组中数据未排序的第一个位置：

```
#include <thrust/sort.h>

...

int A[8] = {0, 1, 2, 3, 0, 1, 2, 3};

int * B = thrust::is_sorted_until(A, A + 8);

// B - A is 4
// [A, B) is sorted
```

模板参数

ForwardIterator：是 Forward Iterator 的模型，ForwardIterator 的 value_type 是 LessThan Comparable 的模型

函数参数

- **first**：感兴趣范围的起始
- **last**：感兴趣范围的末尾

返回值

输入范围内使其排序的最后一个迭代器

另请参阅

- is_sorted
- sort

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

- sort_by_key
- stable_sort
- stable_sort_by_key

thrust::is_sorted_until

```
template <typename DerivedPolicy,
  typename ForwardIterator,
  typename Compare>
__host__ __device__ ForwardIterator
is_sorted_until(const thrust::detail::execution_policy_base< DerivedPolicy
↪> & exec,
  ForwardIterator first,
  ForwardIterator last,
  Compare comp);
```

此版本的 is_sorted_until 返回 [first,last] 内使得 [first,last) 根据函数对象 comp 排序的最后一个迭代器 i。如果 distance(first,last) < 2 ，is_sorted_until 则返回 last。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::host 并行化执行策略，使用 is_sorted_until 查找数组中数据未按降序排序的第一个位置：

```
#include <thrust/sort.h>
#include <thrust/functional.h>
#include <thrust/execution_policy.h>

...

int A[8] = {3, 2, 1, 0, 3, 2, 1, 0};

thrust::greater<int> comp;
int * B = thrust::is_sorted_until(thrust::host, A, A + 8, comp);

// B - A is 4
// [A, B) is sorted in descending order
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **ForwardIterator**：是 Forward Iterator 的模型，且 ForwardIterator 的 value_type 可转换为 Compare 的 argument_type
- **Compare**：是 Strict Weak Ordering 的模型

函数参数

- **exec**：用于并行化的执行策略
- **first**：感兴趣范围的起始
- **last**：感兴趣范围的末尾
- **comp**：用于比较的函数对象

返回值

输入范围内使其排序的最后一个迭代器

另请参阅

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

- is_sorted
- sort
- sort_by_key
- stable_sort
- stable_sort_by_key

thrust::is_sorted_until

```
template <typename ForwardIterator,
  typename Compare>
ForwardIterator
is_sorted_until(ForwardIterator first,
  ForwardIterator last,
  Compare comp);
```

此版本的 is_sorted_until 返回 [first,last] 内使得 [first,last) 根据函数对象 comp 排序的最后一个迭代器 i。如果 distance(first,last) < 2 ，is_sorted_until 则返回 last。

以下代码片段演示了如何使用 is_sorted_until 查找数组中数据未按降序排列的第一个位置：

```
#include <thrust/sort.h>
#include <thrust/functional.h>

...

int A[8] = {3, 2, 1, 0, 3, 2, 1, 0};

thrust::greater<int> comp;
int * B = thrust::is_sorted_until(A, A + 8, comp);

// B - A is 4
// [A, B) is sorted in descending order
```

模板参数

- **ForwardIterator**：是 Forward Iterator 的模型，且 ForwardIterator 的 value_type 可转换为 Compare 的 argument_type
- **Compare**：是 Strict Weak Ordering 的模型

函数参数

- **first**：感兴趣范围的起始
- **last**：感兴趣范围的末尾
- **comp**：用于比较的函数对象

返回值

输入范围内使其排序的最后一个迭代器

另请参阅

- is_sorted
- sort
- sort_by_key
- stable_sort

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

- stable_sort_by_key

2.1.4.7 变换归约（Transformed Reduction）

函数

```
template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2,
  typename OutputType>
__host__ __device__ OutputType
thrust::inner_product(const thrust::detail::execution_policy_base<
 ↪DerivedPolicy > & exec,
  InputIterator1 first1,
  InputIterator1 last1,
  InputIterator2 first2,
  OutputType init);

template <typename InputIterator1,
  typename InputIterator2,
  typename OutputType>
OutputType
thrust::inner_product(InputIterator1 first1,
  InputIterator1 last1,
  InputIterator2 first2,
  OutputType init);

template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2,
  typename OutputType,
  typename BinaryFunction1,
  typename BinaryFunction2>
__host__ __device__ OutputType
thrust::inner_product(const thrust::detail::execution_policy_base<
 ↪DerivedPolicy > & exec,
  InputIterator1 first1,
  InputIterator1 last1,
  InputIterator2 first2,
  OutputType init,
  BinaryFunction1 binary_op1,
  BinaryFunction2 binary_op2);

template <typename InputIterator1,
  typename InputIterator2,
  typename OutputType,
  typename BinaryFunction1,
  typename BinaryFunction2>
OutputType
thrust::inner_product(InputIterator1 first1,
  InputIterator1 last1,
  InputIterator2 first2,
  OutputType init,
  BinaryFunction1 binary_op1,
  BinaryFunction2 binary_op2);
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
template <typename DerivedPolicy,
  typename InputIterator,
  typename UnaryFunction,
  typename OutputType,
  typename BinaryFunction>
__host__ __device__ OutputType
thrust::transform_reduce(const thrust::detail::execution_policy_base<
 ↪DerivedPolicy > & exec,
  InputIterator first,
  InputIterator last,
  UnaryFunction unary_op,
  OutputType init,
  BinaryFunction binary_op);

template <typename InputIterator,
  typename UnaryFunction,
  typename OutputType,
  typename BinaryFunction>
OutputType
thrust::transform_reduce(InputIterator first,
  InputIterator last,
  UnaryFunction unary_op,
  OutputType init,
  BinaryFunction binary_op);
```

thrust::inner_product

```
template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2,
  typename OutputType>
__host__ __device__ OutputType
inner_product(const thrust::detail::execution_policy_base< DerivedPolicy >
 ↪& exec,
  InputIterator1 first1,
  InputIterator1 last1,
  InputIterator2 first2,
  OutputType init);
```

inner_product 计算范围 [first1, last1) 和 [first2, first2 + (last1 - first1)) 的内积。

具体而言，此版本的 inner_product 计算 init + (*first1 * *first2) + (*(first1+1) * *(first2+1)) + …的和。

算法的执行由 exec 确定并行化。

下面的代码片段演示了如何通过 thrust::host 并行化执行策略，用 inner_product 计算两个向量的点积：

```
#include <thrust/inner_product.h>
#include <thrust/execution_policy.h>
...
float vec1[3] = {1.0f, 2.0f, 5.0f};
float vec2[3] = {4.0f, 1.0f, 5.0f};

float result = thrust::inner_product(thrust::host, vec1, vec1 + 3, vec2, 0.
 ↪0f);
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
// result == 31.0f
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **InputIterator1**：是 Input Iterator 的模型
- **InputIterator2**：是 Input Iterator 的模型
- **OutputType**：是 Assignable 的模型，如果 x 是 OutputType 的类型对象，y 是 InputIterator1 的 value_type 的对象，z 是 InputIterator2 的 value_type 的对象，则定义 x + y * z，且可转换为 OutputType

函数参数

- **exec**：用于并行化的执行策略
- **first1**：第一个序列的起始
- **last1**：第一个序列的末尾
- **first2**：第二个序列的起始
- **init**：结果的初始值

返回值

序列 [first1, last1) 和 [first2, last2) 加上 init 的内积

另请参阅

https://en.cppreference.com/w/cpp/algorithm/inner_product

thrust::inner_product

```
template <typename InputIterator1,
  typename InputIterator2,
  typename OutputType>
OutputType
inner_product(InputIterator1 first1,
  InputIterator1 last1,
  InputIterator2 first2,
  OutputType init);
```

inner_product 计算范围 [first1, last1) 和 [first2, first2 + (last1 - first1)) 的内积。

具体而言，此版本的 inner_product 计算 init + (*first1 * *first2) + (*(first1+1) * *(first2+1)) + ⋯的和。

与 C++ 标准模板库函数 std::inner_product 不同，此函数不保证执行顺序。

下面的代码片段演示了如何使用 inner_product 计算两个向量的点积：

```
#include <thrust/inner_product.h>
...
float vec1[3] = {1.0f, 2.0f, 5.0f};
float vec2[3] = {4.0f, 1.0f, 5.0f};

float result = thrust::inner_product(vec1, vec1 + 3, vec2, 0.0f);

// result == 31.0f
```

模板参数

- **InputIterator1**：是 Input Iterator 的模型

飞书信息 MetaX Confidential 2024-11-18 15:00:00

- **InputIterator2**：是 Input Iterator 的模型
- **OutputType**：是 Assignable 的模型，如果 x 是 OutputType 的类型对象，y 是 InputIterator1 的 value_type 的对象，z 是 InputIterator2 的 value_type 的对象，则定义 x + y * z，且可转换为 OutputType

函数参数

- **first1**：第一个序列的起始
- **last1**：第一个序列的末尾
- **first2**：第二个序列的起始
- **init**：结果的初始值

返回值

序列 [first1, last1) 和 [first2, last2) 加上 init 的内积

另请参阅

https://en.cppreference.com/w/cpp/algorithm/inner_product

thrust::inner_product

```
template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2,
  typename OutputType,
  typename BinaryFunction1,
  typename BinaryFunction2>
__host__ __device__ OutputType
inner_product(const thrust::detail::execution_policy_base< DerivedPolicy >
↪& exec,
  InputIterator1 first1,
  InputIterator1 last1,
  InputIterator2 first2,
  OutputType init,
  BinaryFunction1 binary_op1,
  BinaryFunction2 binary_op2);
```

inner_product 计算范围 [first1, last1) 和 [first2, first2 + (last1 - first1)) 的内积。

此版本的 inner_product 与第一个版本相同，只不过使用的是两个用户提供的函数对象，而不是 operator+ 和 operator*。

具体而言，此版本的 inner_product 计算 binary_op1(init, binary_op2(*first1, *first2)), …的和。

算法的执行由 exec 确定并行化。

```
#include <thrust/inner_product.h>
#include <thrust/execution_policy.h>
...
float vec1[3] = {1.0f, 2.0f, 5.0f};
float vec2[3] = {4.0f, 1.0f, 5.0f};

float init = 0.0f;
thrust::plus<float>       binary_op1;
thrust::multiplies<float> binary_op2;

float result = thrust::inner_product(thrust::host, vec1, vec1 + 3, vec2,
↪init, binary_op1, binary_op2);
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
// result == 31.0f
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **InputIterator1**：是 Input Iterator 的模型，InputIterator1 的 value_type 可转换为 BinaryFunction2 的 first_argument_type
- **InputIterator2**：是 Input Iterator 的模型，InputIterator2 的 value_type 可转换为 BinaryFunction2 的 second_argument_type
- **OutputType**：是 Assignable 的模型，OutputType 可转换为 BinaryFunction1 的 first_argument_type
- **BinaryFunction1**：是 Binary Function 的模型，BinaryFunction1 的 return_type 可转换为 OutputType
- **BinaryFunction2**：是 Binary Function 的模型，BinaryFunction2 的 return_type 可转换为 BinaryFunction1 的 second_argument_type

函数参数

- **exec**：用于并行化的执行策略
- **first1**：第一个序列的起始
- **last1**：第一个序列的末尾
- **first2**：第二个序列的起始
- **init**：结果的初始值
- **binary_op1**：广义相加运算
- **binary_op2**：广义相乘运算

返回值

序列 [first1, last1) 和 [first2, last2) 的内积

另请参阅

https://en.cppreference.com/w/cpp/algorithm/inner_product

thrust::inner_product

```
template <typename InputIterator1,
  typename InputIterator2,
  typename OutputType,
  typename BinaryFunction1,
  typename BinaryFunction2>
OutputType
inner_product(InputIterator1 first1,
  InputIterator1 last1,
  InputIterator2 first2,
  OutputType init,
  BinaryFunction1 binary_op1,
  BinaryFunction2 binary_op2);
```

inner_product 计算范围 [first1, last1) 和 [first2, first2 + (last1 - first1)) 的内积。

此版本的 inner_product 与第一个版本相同，只不过使用的是两个用户提供的函数对象，而不是 operator+ 和 operator*。

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

具体而言，此版本的 inner_product 计算 binary_op1(init, binary_op2(*first1, *first2)), …的和。

与 C++ 标准模板库函数 std::inner_product 不同，此函数不保证执行顺序。

```
#include <thrust/inner_product.h>
...
float vec1[3] = {1.0f, 2.0f, 5.0f};
float vec2[3] = {4.0f, 1.0f, 5.0f};

float init = 0.0f;
thrust::plus<float>       binary_op1;
thrust::multiplies<float> binary_op2;

float result = thrust::inner_product(vec1, vec1 + 3, vec2, init, binary_
↪op1, binary_op2);

// result == 31.0f
```

模板参数

- **InputIterator1**：是 Input Iterator 的模型，InputIterator1 的 value_type 可转换为 BinaryFunction2 的 first_argument_type
- **InputIterator2**：是 Input Iterator 的模型，InputIterator2 的 value_type 可转换为 BinaryFunction2 的 second_argument_type
- **OutputType**：是 Assignable 的模型，OutputType 可转换为 BinaryFunction1 的 first_argument_type
- **BinaryFunction1**：是 Binary Function 的模型，BinaryFunction1 的 return_type 可转换为 OutputType
- **BinaryFunction2**：是 Binary Function 的模型，BinaryFunction2 的 return_type 可转换为 BinaryFunction1 的 second_argument_type

函数参数

- **first1**：第一个序列的起始
- **last1**：第一个序列的末尾
- **first2**：第二个序列的起始
- **init**：结果的初始值
- **binary_op1**：广义相加运算
- **binary_op2**：广义相乘运算

返回值

序列 [first1, last1) 和 [first2, last2) 的内积

另请参阅

https://en.cppreference.com/w/cpp/algorithm/inner_product

thrust::transform_reduce

```
template <typename DerivedPolicy,
  typename InputIterator,
  typename UnaryFunction,
  typename OutputType,
  typename BinaryFunction>
```

(下页继续)

飞谱信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
__host__ __device__ OutputType
transform_reduce(const thrust::detail::execution_policy_base<
↪DerivedPolicy > & exec,
  InputIterator first,
  InputIterator last,
  UnaryFunction unary_op,
  OutputType init,
  BinaryFunction binary_op);
```

transform_reduce 融合了 transform 和 reduce 运算。transform_reduce 相当于执行由 unary_op 定义的变换，使序列成为临时序列，然后对变换后的序列执行 reduce 运算。多数情况下，将这两个运算融合在一起更高效，因为需要的内存读取和写入更少了。

transform_reduce 根据 unary_op 对序列 [first, last) 的变换执行归约。具体而言，将 unary_op 应用于序列的每个元素，然后使用初始值 init 将结果归约为具有 binary_op 的单个值。请注意，不对初始值 init 应用变换 unary_op。不指定归约的顺序，所以 binary_op 必须具有交换律和结合律。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::host 并行化执行策略，使用 transform_reduce 计算一个范围内元素绝对值的最大值：

```
#include <thrust/transform_reduce.h>
#include <thrust/functional.h>
#include <thrust/execution_policy.h>

template<typename T>
struct absolute_value : public unary_function<T,T>
{
__host__ __device__ T operator()(const T &x) const
{
   return x < T(0) ? -x : x;
}
};

...

int data[6] = {-1, 0, -2, -2, 1, -3};
int result = thrust::transform_reduce(thrust::host,
data, data + 6,
absolute_value<int>(),
0,
thrust::maximum<int>());
// result == 3
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **InputIterator**：是 Input Iterator 的模型，InputIterator 的 value_type 可转换为 UnaryFunction 的 argument_type
- **UnaryFunction**：是 Unary Function 的模型，UnaryFunction 的 result_type 可转换为 OutputType
- **OutputType**：是 Assignable 的模型，且可转换为 BinaryFunction 的 first_argument_type 和 second_argument_type
- **BinaryFunction**：是 Binary Function 的模型，BinaryFunction 的 result_type 可转换为 OutputType

函数参数

- **exec**：用于并行化的执行策略

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

- **first**：序列起始
- **last**：序列末尾
- **unary_op**：应用于输入序列每个元素的函数
- **init**：结果被初始化为此值
- **binary_op**：归约运算

返回值

变换归约的结果

另请参阅

- transform
- reduce

thrust::transform_reduce

```
template <typename InputIterator,
  typename UnaryFunction,
  typename OutputType,
  typename BinaryFunction>
OutputType
transform_reduce(InputIterator first,
  InputIterator last,
  UnaryFunction unary_op,
  OutputType init,
  BinaryFunction binary_op);
```

transform_reduce 融合了 transform 和 reduce 运算。transform_reduce 相当于执行由 unary_op 定义的变换，使序列成为临时序列，然后对变换后的序列执行 reduce 运算。多数情况下，将这两个运算融合在一起更高效，因为需要的内存读取和写入更少了。

transform_reduce 根据 unary_op 对序列 [first, last) 的变换执行归约。具体而言，将 unary_op 应用于序列的每个元素，然后使用初始值 init 将结果归约为具有 binary_op 的单个值。请注意，不对初始值 init 应用变换 unary_op。不指定归约的顺序，所以 binary_op 必须具有交换律和结合律。

以下代码片段演示了如何使用 transform_reduce 计算一个范围内元素绝对值的最大值：

```
#include <thrust/transform_reduce.h>
#include <thrust/functional.h>

template<typename T>
struct absolute_value : public unary_function<T,T>
{
__host__ __device__ T operator()(const T &x) const
{
   return x < T(0) ? -x : x;
}
};

...

int data[6] = {-1, 0, -2, -2, 1, -3};
int result = thrust::transform_reduce(data, data + 6,
absolute_value<int>(),
0,
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
thrust::maximum<int>());
// result == 3
```

模板参数

- **InputIterator**：是 Input Iterator 的模型，InputIterator 的 value_type 可转换为 UnaryFunction 的 argument_type
- **UnaryFunction**：是 Unary Function 的模型，UnaryFunction 的 result_type 可转换为 OutputType
- **OutputType**：是 Assignable 的模型，且可转换为 BinaryFunction 的 first_argument_type 和 second_argument_type
- **BinaryFunction**：是 Binary Function 的模型，BinaryFunction 的 result_type 可转换为 OutputType

函数参数

- **first**：序列起始
- **last**：序列末尾
- **unary_op**：应用于输入序列每个元素的函数
- **init**：结果被初始化为此值
- **binary_op**：归约运算

返回值

变换归约的结果

另请参阅

- transform
- reduce

2.1.5 重新排序（Reordering）

2.1.5.1 分区（Partitioning）

函数

```
template <typename DerivedPolicy,
  typename ForwardIterator,
  typename Predicate>
__host__ __device__ ForwardIterator
thrust::partition(const thrust::detail::execution_policy_base<
↪DerivedPolicy > & exec,
 ForwardIterator first,
  ForwardIterator last,
  Predicate pred);

template <typename ForwardIterator,
  typename Predicate>
ForwardIterator
thrust::partition(ForwardIterator first,
  ForwardIterator last,
  Predicate pred);

template <typename DerivedPolicy,
  typename ForwardIterator,
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
  typename InputIterator,
  typename Predicate>
__host__ __device__ ForwardIterator
thrust::partition(const thrust::detail::execution_policy_base<
↪DerivedPolicy > & exec,
  ForwardIterator first,
  ForwardIterator last,
  InputIterator stencil,
  Predicate pred);

template <typename ForwardIterator,
  typename InputIterator,
  typename Predicate>
ForwardIterator
thrust::partition(ForwardIterator first,
  ForwardIterator last,
  InputIterator stencil,
  Predicate pred);

template <typename DerivedPolicy,
  typename InputIterator,
  typename OutputIterator1,
  typename OutputIterator2,
  typename Predicate>
__host__ __device__ thrust::pair< OutputIterator1, OutputIterator2 >
thrust::partition_copy(const thrust::detail::execution_policy_base<
↪DerivedPolicy > & exec,
  InputIterator first,
  InputIterator last,
  OutputIterator1 out_true,
  OutputIterator2 out_false,
  Predicate pred);

template <typename InputIterator,
  typename OutputIterator1,
  typename OutputIterator2,
  typename Predicate>
thrust::pair< OutputIterator1, OutputIterator2 >
thrust::partition_copy(InputIterator first,
  InputIterator last,
  OutputIterator1 out_true,
  OutputIterator2 out_false,
  Predicate pred);

template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator1,
  typename OutputIterator2,
  typename Predicate>
__host__ __device__ thrust::pair< OutputIterator1, OutputIterator2 >
thrust::partition_copy(const thrust::detail::execution_policy_base<
↪DerivedPolicy > & exec,
  InputIterator1 first,
  InputIterator1 last,
  InputIterator2 stencil,
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
  OutputIterator1 out_true,
  OutputIterator2 out_false,
  Predicate pred);

template <typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator1,
  typename OutputIterator2,
  typename Predicate>
thrust::pair< OutputIterator1, OutputIterator2 >
thrust::partition_copy(InputIterator1 first,
  InputIterator1 last,
  InputIterator2 stencil,
  OutputIterator1 out_true,
  OutputIterator2 out_false,
  Predicate pred);

template <typename DerivedPolicy,
  typename ForwardIterator,
  typename Predicate>
__host__ __device__ ForwardIterator
thrust::stable_partition(const thrust::detail::execution_policy_base<
↪DerivedPolicy > & exec,
  ForwardIterator first,
  ForwardIterator last,
  Predicate pred);

template <typename ForwardIterator,
  typename Predicate>
ForwardIterator
thrust::stable_partition(ForwardIterator first,
  ForwardIterator last,
  Predicate pred);

template <typename DerivedPolicy,
  typename ForwardIterator,
  typename InputIterator,
  typename Predicate>
__host__ __device__ ForwardIterator
thrust::stable_partition(const thrust::detail::execution_policy_base<
↪DerivedPolicy > & exec,
  ForwardIterator first,
  ForwardIterator last,
  InputIterator stencil,
  Predicate pred);

template <typename ForwardIterator,
  typename InputIterator,
  typename Predicate>
ForwardIterator
thrust::stable_partition(ForwardIterator first,
  ForwardIterator last,
  InputIterator stencil,
  Predicate pred);

template <typename DerivedPolicy,
```

(下页继续)

飞谱信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
  typename InputIterator,
  typename OutputIterator1,
  typename OutputIterator2,
  typename Predicate>
__host__ __device__ thrust::pair< OutputIterator1, OutputIterator2 >
thrust::stable_partition_copy(const thrust::detail::execution_policy_base<
↪DerivedPolicy > & exec,
  InputIterator first,
  InputIterator last,
  OutputIterator1 out_true,
  OutputIterator2 out_false,
  Predicate pred);

template <typename InputIterator,
  typename OutputIterator1,
  typename OutputIterator2,
  typename Predicate>
thrust::pair< OutputIterator1, OutputIterator2 >
thrust::stable_partition_copy(InputIterator first,
  InputIterator last,
  OutputIterator1 out_true,
  OutputIterator2 out_false,
  Predicate pred);

template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator1,
  typename OutputIterator2,
  typename Predicate>
__host__ __device__ thrust::pair< OutputIterator1, OutputIterator2 >
thrust::stable_partition_copy(const thrust::detail::execution_policy_base<
↪DerivedPolicy > & exec,
  InputIterator1 first,
  InputIterator1 last,
  InputIterator2 stencil,
  OutputIterator1 out_true,
  OutputIterator2 out_false,
  Predicate pred);

template <typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator1,
  typename OutputIterator2,
  typename Predicate>
thrust::pair< OutputIterator1, OutputIterator2 >
thrust::stable_partition_copy(InputIterator1 first,
  InputIterator1 last,
  InputIterator2 stencil,
  OutputIterator1 out_true,
  OutputIterator2 out_false,
  Predicate pred);
```

沐曦集成电路（上海）有限公司 MetaX Confidential 2024-11-18 15:00:00

thrust::partition

```
template <typename DerivedPolicy,
  typename ForwardIterator,
  typename Predicate>
__host__ __device__ ForwardIterator
partition(const thrust::detail::execution_policy_base< DerivedPolicy > &
↪exec,
  ForwardIterator first,
  ForwardIterator last,
  Predicate pred);
```

partition 基于函数对象 pred 对元素 [first, last) 重新排序，使得满足 pred 的所有元素先于不满足 pred 的元素。后置条件是，对于范围 [first, last) 内的某些迭代器 middle，范围 [first,middle) 内的每个迭代器 i 使 pred(*i) 为 true；范围 [middle, last) 内的每一个迭代器 i 使其为 false。partition 的返回值是 middle。

请注意，两个重新排序的序列中元素的相对顺序不一定与原始序列中的顺序相同。另一种算法 stable_partition 确实保证了保持相对顺序。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::host 并行化执行策略，使用 partition 对序列重新排序，使偶数先于奇数：

```
#include <thrust/partition.h>
#include <thrust/execution_policy.h>
...
struct is_even
{
__host__ __device__
bool operator()(const int &x)
{
   return (x % 2) == 0;
}
};
...
int A[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
const int N = sizeof(A)/sizeof(int);
thrust::partition(thrust::host,
A, A + N,
is_even());
// A is now {2, 4, 6, 8, 10, 1, 3, 5, 7, 9}
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **ForwardIterator**：是 Forward Iterator 的模型，ForwardIterator 的 value_type 可转换为 Predicate 的 argument_type。ForwardIterator 是可变的
- **Predicate**：是 Predicate 的模型

函数参数

- **exec**：用于并行化的执行策略
- **first**：待重新排序的序列的起始
- **last**：待重新排序的序列的末尾
- **pred**：决定序列 [first, last) 内每个元素属于哪个分区的函数对象

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

返回值

指向第二分区第一个元素的迭代器，即不满足 pred 的元素的序列

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/partition
- stable_partition
- partition_copy

thrust::partition

```
template <typename ForwardIterator,
  typename Predicate>
ForwardIterator
partition(ForwardIterator first,
  ForwardIterator last,
  Predicate pred);
```

partition 基于函数对象 pred 对元素 [first, last) 重新排序，使得满足 pred 的所有元素先于不满足 pred 的元素。后置条件是，对于范围 [first, last) 内的某些迭代器 middle，范围 [first,middle) 内的每个迭代器 i 使 pred(*i) 为 true；范围 [middle, last) 内的每一个迭代器 i 使其为 false。partition 的返回值是 middle。

请注意，两个重新排序的序列中元素的相对顺序不一定与原始序列中的顺序相同。另一种算法 stable_partition 确实保证了保持相对顺序。

以下代码片段演示了如何使用 partition 对序列重新排序，使偶数先于奇数：

```
#include <thrust/partition.h>
...
struct is_even
{
__host__ __device__
bool operator()(const int &x)
{
  return (x % 2) == 0;
}
};
...
int A[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
const int N = sizeof(A)/sizeof(int);
thrust::partition(A, A + N,
is_even());
// A is now {2, 4, 6, 8, 10, 1, 3, 5, 7, 9}
```

模板参数

- **ForwardIterator**：是 Forward Iterator 的模型，ForwardIterator 的 value_type 可转换为 Predicate 的 argument_type。ForwardIterator 是可变的
- **Predicate**：是 Predicate 的模型

函数参数

- **first**：待重新排序的序列的起始
- **last**：待重新排序的序列的末尾
- **pred**：决定序列 [first, last) 内每个元素属于哪个分区的函数对象

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

返回值

指向第二分区第一个元素的迭代器，即不满足 pred 的元素的序列

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/partition
- stable_partition
- partition_copy

thrust::partition

```
template <typename DerivedPolicy,
  typename ForwardIterator,
  typename InputIterator,
  typename Predicate>
__host__ __device__ ForwardIterator
partition(const thrust::detail::execution_policy_base< DerivedPolicy > &
↪exec,
  ForwardIterator first,
  ForwardIterator last,
  InputIterator stencil,
  Predicate pred);
```

partition 基于应用于模板范围 [stencil, stencil + (last - first)) 的函数对象 pred 对元素 [first, last) 重新排序，使得对应模板元素满足 pred 的所有元素先于其对应模板元素不满足 pred 的所有元素。后置条件是，对于范围 [first, last) 内的某些迭代器 middle，范围 [stencil,stencil + (middle - first)) 内的每个迭代器 stencil_i 使 pred(*stencil_i) 为 true；范围 [stencil + (middle - first), stencil + (last - first)) 内的每一个迭代器 stencil_i 使其为 false。stable_partition 的返回值是 middle。

请注意，两个重新排序的序列中元素的相对顺序不一定与原始序列中的顺序相同。另一种算法 stable_partition 确实保证了保持相对顺序。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::host 并行化执行策略，使用 partition 对序列重新排序，使偶数先于奇数：

```
#include <thrust/partition.h>
#include <thrust/execution_policy.h>
...
struct is_even
{
__host__ __device__
bool operator()(const int &x)
{
   return (x % 2) == 0;
}
};
...
int A[] = {0, 1, 0, 1, 0, 1, 0, 1, 0,  1};
int S[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
const int N = sizeof(A)/sizeof(int);
thrust::partition(thrust::host, A, A + N, S, is_even());
// A is now {1, 1, 1, 1, 1, 0, 0, 0, 0, 0}
// S is unmodified
```

模板参数

飞腾信息 MetaX Confidential 2024-11-18 15:30:00

- **DerivedPolicy**：派生执行策略的名称
- **ForwardIterator**：是 Forward Iterator 的模型，且 ForwardIterator 是可变的
- **InputIterator**：是 Input Iterator 的模型，InputIterator 的 value_type 可转换为 Predicate 的 argument_type
- **Predicate**：是 Predicate 的模型

函数参数

- **exec**：用于并行化的执行策略
- **first**：待重新排序的序列的起始
- **last**：待重新排序的序列的末尾
- **stencil**：模版序列的起始
- **pred**：决定序列 [first, last) 内每个元素属于哪个分区的函数对象

前提条件

范围 [first,last) 不应与范围 [stencil, stencil + (last - first)) 重叠

返回值

指向第二分区第一个元素的迭代器，即其模版元素不满足 pred 的元素的序列

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/partition
- stable_partition
- partition_copy

thrust::partition

```
template <typename ForwardIterator,
  typename InputIterator,
  typename Predicate>
ForwardIterator
partition(ForwardIterator first,
  ForwardIterator last,
  InputIterator stencil,
  Predicate pred);
```

partition 基于应用于模板范围 [stencil, stencil + (last - first)) 的函数对象 pred 对元素 [first, last) 重新排序，使得对应模板元素满足 pred 的所有元素先于其对应模板元素不满足 pred 的所有元素。后置条件是，对于范围 [first, last) 内的某些迭代器 middle，范围 [stencil,stencil + (middle - first)) 内的每个迭代器 stencil_i 使 pred(*stencil_i) 为 true；范围 [stencil + (middle - first), stencil + (last - first)) 内的每一个迭代器 stencil_i 使其为 false。stable_partition 的返回值是 middle。

请注意，两个重新排序的序列中元素的相对顺序不一定与原始序列中的顺序相同。另一种算法 stable_partition 确实保证了保持相对顺序。

以下代码片段演示了如何使用 partition 对序列重新排序，使偶数先于奇数：

```
#include <thrust/partition.h>
...
struct is_even
{
__host__ __device__
bool operator()(const int &x)
```

(下页继续)

飞腾信息 MetaX Confidential 2023-11-18 15:00:00

(续上页)

```
{
   return (x % 2) == 0;
}
};
...
int A[] = {0, 1, 0, 1, 0, 1, 0, 1, 0,  1};
int S[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
const int N = sizeof(A)/sizeof(int);
thrust::partition(A, A + N, S, is_even());
// A is now {1, 1, 1, 1, 1, 0, 0, 0, 0, 0}
// S is unmodified
```

模板参数

- **ForwardIterator**：是 Forward Iterator 的模型，且 ForwardIterator 是可变的
- **InputIterator**：是 Input Iterator 的模型，InputIterator 的 value_type 可转换为 Predicate 的 argument_type
- **Predicate**：是 Predicate 的模型

函数参数

- **first**：待重新排序的序列的起始
- **last**：待重新排序的序列的末尾
- **stencil**：模版序列的起始
- **pred**：决定序列 [first, last) 内每个元素属于哪个分区的函数对象

前提条件

范围 [first,last) 不应与范围 [stencil, stencil + (last - first)) 重叠

返回值

指向第二分区第一个元素的迭代器，即其模版元素不满足 pred 的元素的序列

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/partition
- stable_partition
- partition_copy

thrust::partition_copy

```
template <typename DerivedPolicy,
  typename InputIterator,
  typename OutputIterator1,
  typename OutputIterator2,
  typename Predicate>
__host__ __device__ thrust::pair< OutputIterator1, OutputIterator2 >
partition_copy(const thrust::detail::execution_policy_base< DerivedPolicy >
 ↪ & exec,
  InputIterator first,
  InputIterator last,
  OutputIterator1 out_true,
  OutputIterator2 out_false,
  Predicate pred);
```

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

partition_copy 与 partition 的不同之处仅在于重新排序的序列被写入不同的输出序列，而不是就地写入。

partition_copy 基于函数对象 pred 复制 [first, last) 内的元素。 将所有满足 pred 的元素复制到从 out_true 开始的范围，将所有不满足的元素复制到从 out_false 开始的范围。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::host 并行化执行策略，使用 partition_copy 将一个序列分成偶数和奇数的两个输出序列：

```
#include <thrust/partition.h>
#include <thrust/execution_policy.h>
...
struct is_even
{
__host__ __device__
bool operator()(const int &x)
{
   return (x % 2) == 0;
}
};
...
int A[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int result[10];
const int N = sizeof(A)/sizeof(int);
int *evens = result;
int *odds  = result + 5;
thrust::partition_copy(thrust::host, A, A + N, evens, odds, is_even());
// A remains {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
// result is now {2, 4, 6, 8, 10, 1, 3, 5, 7, 9}
// evens points to {2, 4, 6, 8, 10}
// odds points to {1, 3, 5, 7, 9}
```

备注

两个重新排序的序列中元素的相对顺序不一定与原始序列中的顺序相同。另一种算法 stable_partition_copy 确实保证了保持相对顺序。

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **InputIterator**：是 Input Iterator 的模型，InputIterator 的 value_type 可转换为 Predicate 的 argument_type ，以及 OutputIterator1 和 OutputIterator2 的 value_types
- **OutputIterator1**：是 Output Iterator 的模型
- **OutputIterator2**：是 Output Iterator 的模型
- **Predicate**：是 Predicate 的模型

函数参数

- **exec**：用于并行化的执行策略
- **first**：待重新排序的序列的起始
- **last**：待重新排序的序列的末尾
- **out_true**：满足 pred 的元素的目标结果序列
- **out_false**：不满足 pred 的元素的目标结果序列
- **pred**：决定序列 [first, last) 内每个元素属于哪个分区的函数对象

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

前提条件

输入范围不应与任何一个输出范围重叠

返回值

p pair ，其中 p.first 是从 out_true 开始的输出范围的末尾，p.second 是从 out_false 开始的输出范围的末尾

另请参阅

- http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2569.pdf
- stable_partition_copy
- partition

thrust::partition_copy

```
template <typename InputIterator,
  typename OutputIterator1,
  typename OutputIterator2,
  typename Predicate>
thrust::pair< OutputIterator1, OutputIterator2 >
partition_copy(InputIterator first,
  InputIterator last,
  OutputIterator1 out_true,
  OutputIterator2 out_false,
  Predicate pred);
```

partition_copy 与 partition 的不同之处仅在于重新排序的序列被写入不同的输出序列，而不是就地写入。

partition_copy 基于函数对象 pred 复制 [first, last) 内的元素。 将所有满足 pred 的元素复制到从 out_true 开始的范围，将所有不满足的元素复制到从 out_false 开始的范围。

以下代码片段演示了如何使用 partition_copy 将一个序列分成偶数和奇数的两个输出序列：

```
#include <thrust/partition.h>
...
struct is_even
{
__host__ __device__
bool operator()(const int &x)
{
   return (x % 2) == 0;
}
};
...
int A[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int result[10];
const int N = sizeof(A)/sizeof(int);
int *evens = result;
int *odds  = result + 5;
thrust::partition_copy(A, A + N, evens, odds, is_even());
// A remains {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
// result is now {2, 4, 6, 8, 10, 1, 3, 5, 7, 9}
// evens points to {2, 4, 6, 8, 10}
// odds points to {1, 3, 5, 7, 9}
```

备注

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

两个重新排序的序列中元素的相对顺序不一定与原始序列中的顺序相同。另一种算法 stable_partition_copy 确实保证了保持相对顺序。

模板参数

- **InputIterator**：是 Input Iterator 的模型，InputIterator 的 value_type 可转换为 Predicate 的 argument_type ，以及 OutputIterator1 和 OutputIterator2 的 value_types
- **OutputIterator1**：是 Output Iterator 的模型
- **OutputIterator2**：是 Output Iterator 的模型
- **Predicate**：是 Predicate 的模型

函数参数

- **first**：待重新排序的序列的起始
- **last**：待重新排序的序列的末尾
- **out_true**：满足 pred 的元素的目标结果序列
- **out_false**：不满足 pred 的元素的目标结果序列
- **pred**：决定序列 [first, last) 内每个元素属于哪个分区的函数对象

前提条件

输入范围不应与任何一个输出范围重叠

返回值

p pair ，其中 p.first 是从 out_true 开始的输出范围的末尾，p.second 是从 out_false 开始的输出范围的末尾

另请参阅

- http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2569.pdf
- stable_partition_copy
- partition

thrust::partition_copy

```
template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator1,
  typename OutputIterator2,
  typename Predicate>
__host__ __device__ thrust::pair< OutputIterator1, OutputIterator2 >
partition_copy(const thrust::detail::execution_policy_base< DerivedPolicy >
↪ & exec,
  InputIterator1 first,
  InputIterator1 last,
  InputIterator2 stencil,
  OutputIterator1 out_true,
  OutputIterator2 out_false,
  Predicate pred);
```

partition_copy 与 partition 的不同之处仅在于重新排序的序列被写入不同的输出序列，而不是就地写入。

partition_copy 基于应用于模板元素范围的函数对象 pred 复制 [first, last) 内的元素。 将其相应模板元素满足 pred 的所有元素复制到从 out_true 开始的范围，将其模板元素不满足 pred 的所有元素复制到从 out_false 开始的范围。

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::host 并行化执行策略，使用 partition_copy 将一个序列分成偶数和奇数的两个输出序列：

```
#include <thrust/partition.h>
#include <thrust/functional.h>
#include <thrust/execution_policy.h>
...
int A[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int S[] = {0, 1, 0, 1, 0, 1, 0, 1, 0,  1};
int result[10];
const int N = sizeof(A)/sizeof(int);
int *evens = result;
int *odds  = result + 5;
thrust::stable_partition_copy(thrust::host, A, A + N, S, evens, odds,
↪thrust::identity<int>());
// A remains {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
// S remains {0, 1, 0, 1, 0, 1, 0, 1, 0,  1}
// result is now {2, 4, 6, 8, 10, 1, 3, 5, 7, 9}
// evens points to {2, 4, 6, 8, 10}
// odds points to {1, 3, 5, 7, 9}
```

备注

两个重新排序的序列中元素的相对顺序不一定与原始序列中的顺序相同。另一种算法 stable_partition_copy 确实保证了保持相对顺序。

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **InputIterator1**：是 Input Iterator 的模型，InputIterator 的 value_type 可转换为 OutputIterator1 和 OutputIterator2 的 value_types
- **InputIterator2**：是 Input Iterator 的模型，InputIterator2 的 value_type 可转换为 Predicate 的 argument_type
- **OutputIterator1**：是 Output Iterator 的模型
- **OutputIterator2**：是 Output Iterator 的模型
- **Predicate**：是 Predicate 的模型

函数参数

- **exec**：用于并行化的执行策略
- **first**：待重新排序的序列的起始
- **last**：待重新排序的序列的末尾
- **stencil**：模版序列的起始
- **out_true**：满足 pred 的元素的目标结果序列
- **out_false**：不满足 pred 的元素的目标结果序列
- **pred**：决定序列 [first, last) 内每个元素属于哪个分区的函数对象

前提条件

输入范围不应与任何一个输出范围重叠

返回值

p pair ，其中 p.first 是从 out_true 开始的输出范围的末尾，p.second 是从 out_false 开始的输出范围的末尾

飞曦信息 MetaX Confidential 2024-11-18 15:00:00

另请参阅

- http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2569.pdf
- stable_partition_copy
- partition

thrust::partition_copy

```
template <typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator1,
  typename OutputIterator2,
  typename Predicate>
thrust::pair< OutputIterator1, OutputIterator2 >
partition_copy(InputIterator1 first,
  InputIterator1 last,
  InputIterator2 stencil,
  OutputIterator1 out_true,
  OutputIterator2 out_false,
  Predicate pred);
```

partition_copy 与 partition 的不同之处仅在于重新排序的序列被写入不同的输出序列，而不是就地写入。

partition_copy 基于应用于模板元素范围的函数对象 pred 复制 [first, last) 内的元素。 将其相应模板元素满足 pred 的所有元素复制到从 out_true 开始的范围，将其模板元素不满足 pred 的所有元素复制到从 out_false 开始的范围。

以下代码片段演示了如何使用 partition_copy 将一个序列分成偶数和奇数的两个输出序列：

```
#include <thrust/partition.h>
#include <thrust/functional.h>
...
int A[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int S[] = {0, 1, 0, 1, 0, 1, 0, 1, 0,  1};
int result[10];
const int N = sizeof(A)/sizeof(int);
int *evens = result;
int *odds  = result + 5;
thrust::stable_partition_copy(A, A + N, S, evens, odds, thrust::identity
↪<int>());
// A remains {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
// S remains {0, 1, 0, 1, 0, 1, 0, 1, 0,  1}
// result is now {2, 4, 6, 8, 10, 1, 3, 5, 7, 9}
// evens points to {2, 4, 6, 8, 10}
// odds points to {1, 3, 5, 7, 9}
```

备注

两个重新排序的序列中元素的相对顺序不一定与原始序列中的顺序相同。另一种算法 stable_partition_copy 确实保证了保持相对顺序。

模板参数

- **InputIterator1**：是 Input Iterator 的模型，InputIterator 的 value_type 可转换为 OutputIterator1 和 OutputIterator2 的 value_types
- **InputIterator2**：是 Input Iterator 的模型，InputIterator2 的 value_type 可转换为 Predicate 的 argument_type

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

- **OutputIterator1**：是 Output Iterator 的模型
- **OutputIterator2**：是 Output Iterator 的模型
- **Predicate**：是 Predicate 的模型

函数参数

- **first**：待重新排序的序列的起始
- **last**：待重新排序的序列的末尾
- **stencil**：模版序列的起始
- **out_true**：满足 pred 的元素的目标结果序列
- **out_false**：不满足 pred 的元素的目标结果序列
- **pred**：决定序列 [first, last) 内每个元素属于哪个分区的函数对象

前提条件

输入范围不应与任何一个输出范围重叠

返回值

p pair，其中 p.first 是从 out_true 开始的输出范围的末尾，p.second 是从 out_false 开始的输出范围的末尾

另请参阅

- http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2569.pdf
- stable_partition_copy
- partition

thrust::stable_partition

```
template <typename DerivedPolicy,
  typename ForwardIterator,
  typename Predicate>
__host__ __device__ ForwardIterator
stable_partition(const thrust::detail::execution_policy_base<
↪DerivedPolicy > & exec,
  ForwardIterator first,
  ForwardIterator last,
  Predicate pred);
```

stable_partition 很像 partition ，它基于函数对象 pred 对元素 [first, last) 重新排序，使得满足 pred 的所有元素先于不满足 pred 的元素。后置条件是，对于范围 [first, last) 内的某些迭代器 middle ，范围 [first,middle) 内的每个迭代器 i 使 pred(*i) 为 true；范围 [middle, last) 内的每一个迭代器 i 使其为 false。stable_partition 的返回值是 middle。

stable_partition 与 partition 的不同之处在于 stable_partition 会保持相对顺序。即，如果 x 和 y 是 [first, last) 内的元素，stencil_x 和 stencil_y 是 [stencil, stencil + (last - first)) 内相应位置的模版元素，满足 pred(stencil_x) == pred(stencil_y) ，并且如果 x 先于 y ，那么在 stable_partition 之后，x 先与 y 仍然为 true 。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::host 并行化执行策略，使用 stable_partition 对序列重新排序，使偶数先于奇数：

飞腾信息 MetaX Confidential 2021-11-18 15:00:00

```
#include <thrust/partition.h>
#include <thrust/execution_policy.h>
...
struct is_even
{
__host__ __device__
bool operator()(const int &x)
{
   return (x % 2) == 0;
}
};
...
int A[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
const int N = sizeof(A)/sizeof(int);
thrust::stable_partition(thrust::host,
   A, A + N,
   is_even());
// A is now {2, 4, 6, 8, 10, 1, 3, 5, 7, 9}
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **ForwardIterator**：是 Forward Iterator 的模型，ForwardIterator 的 value_type 可转换为 Predicate 的 argument_type。ForwardIterator 是可变的
- **Predicate**：是 Predicate 的模型

函数参数

- **exec**：用于并行化的执行策略
- **first**：待重新排序的序列的第一个元素
- **last1**：待重新排序的序列中最末元素之后的一个位置
- **pred**：决定序列 [first, last) 内每个元素属于哪个分区的函数对象

返回值

指向第二分区第一个元素的迭代器，即不满足 pred 的元素序列

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/stable_partition
- partition
- stable_partition_copy

thrust::stable_partition

```
template <typename ForwardIterator,
  typename Predicate>
ForwardIterator
stable_partition(ForwardIterator first,
  ForwardIterator last,
  Predicate pred);
```

stable_partition 很像 partition ，它基于函数对象 pred 对元素 [first, last) 重新排序，使得满足 pred 的所有元素先于不满足 pred 的元素。后置条件是，对于范围 [first, last) 内的某些迭代器 middle，范围 [first,middle) 内的每个迭代器 i 使 pred(*i) 为 true；范围 [middle, last) 内的每一个迭代器 i 使其为 false。stable_partition 的返回值是 middle。

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

stable_partition 与 partition 的不同之处在于 stable_partition 会保持相对顺序。即，如果 x 和 y 是 [first, last) 内的元素，stencil_x 和 stencil_y 是 [stencil, stencil + (last - first)) 内相应位置的模版元素，满足 pred(stencil_x) == pred(stencil_y)，并且如果 x 先于 y，那么在 stable_partition 之后，x 先与 y 仍然为 true。

以下代码片段演示了如何使用 stable_partition 对序列重新排序，使偶数先于奇数：

```
#include <thrust/partition.h>
...
struct is_even
{
__host__ __device__
bool operator()(const int &x)
{
   return (x % 2) == 0;
}
};
...
int A[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
const int N = sizeof(A)/sizeof(int);
thrust::stable_partition(A, A + N,
   is_even());
// A is now {2, 4, 6, 8, 10, 1, 3, 5, 7, 9}
```

模板参数

- **ForwardIterator**：是 Forward Iterator 的模型，ForwardIterator 的 value_type 可转换为 Predicate 的 argument_type。ForwardIterator 是可变的
- **Predicate**：是 Predicate 的模型

函数参数

- **first**：待重新排序的序列第一个元素
- **last1**：待重新排序的序列中最末元素之后的一个位置
- **pred**：决定序列 [first, last) 内每个元素属于哪个分区的函数对象

返回值

指向第二分区第一个元素的迭代器，即不满足 pred 的元素序列

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/stable_partition
- partition
- stable_partition_copy

thrust::stable_partition

```
template <typename DerivedPolicy,
  typename ForwardIterator,
  typename InputIterator,
  typename Predicate>
__host__ __device__ ForwardIterator
stable_partition(const thrust::detail::execution_policy_base<
↪DerivedPolicy > & exec,
  ForwardIterator first,
  ForwardIterator last,
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
  InputIterator stencil,
  Predicate pred);
```

stable_partition 很像 partition ，它基于应用于模板范围 [stencil, stencil + (last - first)) 的函数对象 pred 对元素 [first, last) 重新排序，使得对应模板元素满足 pred 的所有元素先于其对应模板元素不满足 pred 的所有元素。后置条件是，对于范围 [first, last) 内的某些迭代器 middle，范围 [stencil,stencil + (middle - first)) 内的每个迭代器 stencil_i 使 pred(*stencil_i) 为 true；范围 [stencil + (middle - first), stencil + (last - first)) 内的每一个迭代器 stencil_i 使其为 false。stable_partition 的返回值是 middle。

stable_partition 与 partition 的不同之处在于 stable_partition 会保持相对顺序。即，如果 x 和 y 是 [first, last) 内的元素，满足 pred(x) == pred(y)，并且如果 x 先于 y，那么在 stable_partition 之后，x 先于 y 仍然为 true 。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::host 并行化执行策略，使用 stable_partition 对序列重新排序，使偶数先于奇数：

```
#include <thrust/partition.h>
#include <thrust/execution_policy.h>
...
struct is_even
{
__host__ __device__
bool operator()(const int &x)
{
   return (x % 2) == 0;
}
};
...
int A[] = {0, 1, 0, 1, 0, 1, 0, 1, 0,  1};
int S[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
const int N = sizeof(A)/sizeof(int);
thrust::stable_partition(thrust::host, A, A + N, S, is_even());
// A is now {1, 1, 1, 1, 1, 0, 0, 0, 0, 0}
// S is unmodified
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **ForwardIterator**：是 Forward Iterator 的模型，且 ForwardIterator 是可变的
- **InputIterator**：是 Input Iterator 的模型，InputIterator 的 value_type 可转换为 Predicate 的 argument_type
- **Predicate**：是 Predicate 的模型

函数参数

- **exec**：用于并行化的执行策略
- **first**：待重新排序的序列第一个元素
- **last1**：待重新排序的序列中最末元素之后的一个位置
- **stencil**：模版序列的起始
- **pred**：决定序列 [first, last) 内每个元素属于哪个分区的函数对象

前提条件

范围 [first, last) 不应与范围 [stencil, stencil + (last - first)) 重叠

返回值

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

指向第二分区第一个元素的迭代器，即其模版元素不满足 pred 的元素的序列

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/stable_partition
- partition
- stable_partition_copy

thrust::stable_partition

```
template <typename ForwardIterator,
  typename InputIterator,
  typename Predicate>
ForwardIterator
stable_partition(ForwardIterator first,
  ForwardIterator last,
  InputIterator stencil,
  Predicate pred);
```

stable_partition 很像 partition ，它基于应用于模板范围 [stencil, stencil + (last - first)) 的函数对象 pred 对元素 [first, last) 重新排序，使得对应模板元素满足 pred 的所有元素先于其对应模板元素不满足 pred 的所有元素。后置条件是，对于范围 [first, last) 内的某些迭代器 middle，范围 [stencil,stencil + (middle - first)) 内的每个迭代器 stencil_i 使 pred(*stencil_i) 为 true；范围 [stencil + (middle - first), stencil + (last - first)) 内的每一个迭代器 stencil_i 使其为 false。stable_partition 的返回值是 middle。

stable_partition 与 partition 的不同之处在于 stable_partition 会保持相对顺序。即，如果 x 和 y 是 [first, last) 内的元素，满足 pred(x) == pred(y)，并且如果 x 先于 y，那么在 stable_partition 之后，x 先于 y 仍然为 true 。

以下代码片段演示了如何使用 stable_partition 对序列重新排序，使偶数先于奇数：

```
#include <thrust/partition.h>
...
struct is_even
{
__host__ __device__
bool operator()(const int &x)
{
  return (x % 2) == 0;
}
};
...
int A[] = {0, 1, 0, 1, 0, 1, 0, 1, 0,  1};
int S[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
const int N = sizeof(A)/sizeof(int);
thrust::stable_partition(A, A + N, S, is_even());
// A is now {1, 1, 1, 1, 1, 0, 0, 0, 0, 0}
// S is unmodified
```

模板参数

- **ForwardIterator**：是 Forward Iterator 的模型，且 ForwardIterator 是可变的
- **InputIterator**：是 Input Iterator 的模型，InputIterator 的 value_type 可转换为 Predicate 的 argument_type
- **Predicate**：是 Predicate 的模型

函数参数

飞谱信息 Privacy Confidential 2024-11-18 15:06:20

- **first**：待重新排序的序列第一个元素
- **last1**：待重新排序的序列中最末元素之后的一个位置
- **stencil**：模版序列的起始
- **pred**：决定序列 [first, last) 内每个元素属于哪个分区的函数对象

前提条件

范围 [first, last) 不应与范围 [stencil, stencil + (last - first)) 重叠

返回值

指向第二分区第一个元素的迭代器，即其模版元素不满足 pred 的元素的序列

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/stable_partition
- partition
- stable_partition_copy

thrust::stable_partition_copy

```
template <typename DerivedPolicy,
  typename InputIterator,
  typename OutputIterator1,
  typename OutputIterator2,
  typename Predicate>
__host__ __device__ thrust::pair< OutputIterator1, OutputIterator2 >
stable_partition_copy(const thrust::detail::execution_policy_base<
 ↪DerivedPolicy > & exec,
  InputIterator first,
  InputIterator last,
  OutputIterator1 out_true,
  OutputIterator2 out_false,
  Predicate pred);
```

stable_partition_copy 与 stable_partition 的不同之处仅在于重新排序的序列被写入不同的输出序列，而不是就地写入。

stable_partition_copy 基于函数对象 pred 复制 [first, last) 内的元素。 将所有满足 pred 的元素复制到从 out_true 开始的范围，将所有不满足的元素复制到从 out_false 开始的范围。

stable_partition_copy 与 partition_copy 的不同之处在于 stable_partition_copy 会保持相对顺序。即，如果 x 和 y 是 [first, last) 内的元素，满足 pred(x) == pred(y) ，并且如果 x 先于 y ，那么在 stable_partition_copy 之后，在输出中 x 先于 y 仍然为 true 。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::host 并行化执行策略，使用 stable_partition_copy 对序列重新排序，使偶数先于奇数：

```
#include <thrust/partition.h>
#include <thrust/execution_policy.h>
...
struct is_even
{
__host__ __device__
bool operator()(const int &x)
{
```

(下页继续)

(续上页)

```
    return (x % 2) == 0;
}
};
...
int A[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int result[10];
const int N = sizeof(A)/sizeof(int);
int *evens = result;
int *odds  = result + 5;
thrust::stable_partition_copy(thrust::host, A, A + N, evens, odds, is_
↪even());
// A remains {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
// result is now {2, 4, 6, 8, 10, 1, 3, 5, 7, 9}
// evens points to {2, 4, 6, 8, 10}
// odds points to {1, 3, 5, 7, 9}
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **InputIterator**：是 Input Iterator 的模型，InputIterator 的 value_type 可转换为 Predicate 的 argument_type ，以及 OutputIterator1 和 OutputIterator2 的 value_types
- **OutputIterator1**：是 Output Iterator 的模型
- **OutputIterator2**：是 Output Iterator 的模型
- **Predicate**：是 Predicate 的模型

函数参数

- **exec**：用于并行化的执行策略
- **first**：待重新排序的序列第一个元素
- **last1**：待重新排序的序列中最末元素之后的一个位置
- **out_true**：满足 pred 的元素的目标结果序列
- **out_false**：不满足 pred 的元素的目标结果序列
- **pred**：决定序列 [first, last) 内每个元素属于哪个分区的函数对象

前提条件

输入范围不应与任何一个输出范围重叠

返回值

p pair ，其中 p.first 是从 out_true 开始的输出范围的末尾，p.second 是从 out_false 开始的输出范围的末尾

另请参阅

- http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2569.pdf
- partition_copy
- stable_partition

thrust::stable_partition_copy

飞曦信息 MetaX Confidential 2024-11-18 15:00:00

```
template <typename InputIterator,
  typename OutputIterator1,
  typename OutputIterator2,
  typename Predicate>
thrust::pair< OutputIterator1, OutputIterator2 >
stable_partition_copy(InputIterator first,
  InputIterator last,
  OutputIterator1 out_true,
  OutputIterator2 out_false,
  Predicate pred);
```

stable_partition_copy 与 stable_partition 的不同之处仅在于重新排序的序列被写入不同的输出序列，而不是就地写入。

stable_partition_copy 基于函数对象 pred 复制 [first, last) 内的元素。 将所有满足 pred 的元素复制到从 out_true 开始的范围，将所有不满足的元素复制到从 out_false 开始的范围。

stable_partition_copy 与 partition_copy 的不同之处在于 stable_partition_copy 会保持相对顺序。即，如果 x 和 y 是 [first, last) 内的元素，满足 pred(x) == pred(y)，并且如果 x 先于 y ，那么在 stable_partition_copy 之后，在输出中 x 先于 y 仍然为 true 。

以下代码片段演示了如何使用 stable_partition_copy 对序列重新排序，使偶数先于奇数：

```
#include <thrust/partition.h>
...
struct is_even
{
__host__ __device__
bool operator()(const int &x)
{
   return (x % 2) == 0;
}
};
...
int A[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int result[10];
const int N = sizeof(A)/sizeof(int);
int *evens = result;
int *odds  = result + 5;
thrust::stable_partition_copy(A, A + N, evens, odds, is_even());
// A remains {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
// result is now {2, 4, 6, 8, 10, 1, 3, 5, 7, 9}
// evens points to {2, 4, 6, 8, 10}
// odds points to {1, 3, 5, 7, 9}
```

模板参数

- **InputIterator**：是 Input Iterator 的模型，InputIterator 的 value_type 可转换为 Predicate 的 argument_type ，以及 OutputIterator1 和 OutputIterator2 的 value_types
- **OutputIterator1**：是 Output Iterator 的模型
- **OutputIterator2**：是 Output Iterator 的模型
- **Predicate**：是 Predicate 的模型

函数参数

- **first**：待重新排序的序列第一个元素
- **last1**：待重新排序的序列中最末元素之后的一个位置
- **out_true**：满足 pred 的元素的目标结果序列

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

- **out_false**：不满足 pred 的元素的目标结果序列
- **pred**：决定序列 [first, last) 内每个元素属于哪个分区的函数对象

前提条件

输入范围不应与任何一个输出范围重叠

返回值

p pair，其中 p.first 是从 out_true 开始的输出范围的末尾，p.second 是从 out_false 开始的输出范围的末尾

另请参阅

- http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2569.pdf
- partition_copy
- stable_partition

thrust::stable_partition_copy

```
template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator1,
  typename OutputIterator2,
  typename Predicate>
__host__ __device__ thrust::pair< OutputIterator1, OutputIterator2 >
stable_partition_copy(const thrust::detail::execution_policy_base<
 ↪DerivedPolicy > & exec,
  InputIterator1 first,
  InputIterator1 last,
  InputIterator2 stencil,
  OutputIterator1 out_true,
  OutputIterator2 out_false,
  Predicate pred);
```

stable_partition_copy 与 stable_partition 的不同之处仅在于重新排序的序列被写入不同的输出序列，而不是就地写入。

stable_partition_copy 基于应用于模板元素范围的函数对象 pred 复制 [first, last) 内的元素。 将其相应模板元素满足 pred 的所有元素复制到从 out_true 开始的范围，将其模板元素不满足 pred 的所有元素复制到从 out_false 开始的范围。

stable_partition_copy 与 partition_copy 的不同之处在于 stable_partition_copy 会保持相对顺序。即，如果 x 和 y 是 [first, last) 内的元素，满足 pred(x) == pred(y)，并且如果 x 先于 y，那么在 stable_partition_copy 之后，在输出中 x 先于 y 仍然为 true。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::host 并行化执行策略，使用 stable_partition_copy 对序列重新排序，使偶数先于奇数：

```
#include <thrust/partition.h>
#include <thrust/functional.h>
#include <thrust/execution_policy.h>
...
int A[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int S[] = {0, 1, 0, 1, 0, 1, 0, 1, 0,  1};
int result[10];
```

(下页继续)

飞猪信息 MetaX Confidential 2024-1-18 15:00:00

(续上页)

```
const int N = sizeof(A)/sizeof(int);
int *evens = result;
int *odds  = result + 5;
thrust::stable_partition_copy(thrust::host, A, A + N, S, evens, odds,
↪thrust::identity<int>());
// A remains {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
// S remains {0, 1, 0, 1, 0, 1, 0, 1, 0,  1}
// result is now {2, 4, 6, 8, 10, 1, 3, 5, 7, 9}
// evens points to {2, 4, 6, 8, 10}
// odds points to {1, 3, 5, 7, 9}
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **InputIterator1**：是 Input Iterator 的模型，InputIterator 的 value_type 可转换为 OutputIterator1 和 OutputIterator2 的 value_types
- **InputIterator2**：是 Input Iterator 的模型，InputIterator2 的 value_type 可转换为 Predicate 的 argument_type
- **OutputIterator1**：是 Output Iterator 的模型
- **OutputIterator2**：是 Output Iterator 的模型
- **Predicate**：是 Predicate 的模型

函数参数

- **exec**：用于并行化的执行策略
- **first**：待重新排序的序列第一个元素
- **last1**：待重新排序的序列中最末元素之后的一个位置
- **stencil**：模版序列的起始
- **out_true**：满足 pred 的元素的目标结果序列
- **out_false**：不满足 pred 的元素的目标结果序列
- **pred**：决定序列 [first, last) 内每个元素属于哪个分区的函数对象

前提条件

输入范围不应与任何一个输出范围重叠

返回值

p pair ，其中 p.first 是从 out_true 开始的输出范围的末尾，p.second 是从 out_false 开始的输出范围的末尾

另请参阅

- http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2569.pdf
- partition_copy
- stable_partition

thrust::stable_partition_copy

```
template <typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator1,
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
  typename OutputIterator2,
  typename Predicate>
thrust::pair< OutputIterator1, OutputIterator2 >
stable_partition_copy(InputIterator1 first,
  InputIterator1 last,
  InputIterator2 stencil,
  OutputIterator1 out_true,
  OutputIterator2 out_false,
  Predicate pred);
```

stable_partition_copy 与 stable_partition 的不同之处仅在于重新排序的序列被写入不同的输出序列，而不是就地写入。

stable_partition_copy 基于应用于模板元素范围的函数对象 pred 复制 [first, last) 内的元素。 将其相应模板元素满足 pred 的所有元素复制到从 out_true 开始的范围，将其模板元素不满足 pred 的所有元素复制到从 out_false 开始的范围。

stable_partition_copy 与 partition_copy 的不同之处在于 stable_partition_copy 会保持相对顺序。即，如果 x 和 y 是 [first, last) 内的元素，满足 pred(x) == pred(y)，并且如果 x 先于 y ，那么在 stable_partition_copy 之后，在输出中 x 先于 y 仍然为 true 。

以下代码片段演示了如何使用 stable_partition_copy 对序列重新排序，使偶数先于奇数：

```
#include <thrust/partition.h>
#include <thrust/functional.h>
...
int A[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int S[] = {0, 1, 0, 1, 0, 1, 0, 1, 0,  1};
int result[10];
const int N = sizeof(A)/sizeof(int);
int *evens = result;
int *odds  = result + 5;
thrust::stable_partition_copy(A, A + N, S, evens, odds, thrust::identity
↪<int>());
// A remains {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
// S remains {0, 1, 0, 1, 0, 1, 0, 1, 0,  1}
// result is now {2, 4, 6, 8, 10, 1, 3, 5, 7, 9}
// evens points to {2, 4, 6, 8, 10}
// odds points to {1, 3, 5, 7, 9}
```

模板参数

- **InputIterator1**：是 Input Iterator 的模型，InputIterator 的 value_type 可转换为 OutputIterator1 和 OutputIterator2 的 value_types
- **InputIterator2**：是 Input Iterator 的模型，InputIterator2 的 value_type 可转换为 Predicate 的 argument_type
- **OutputIterator1**：是 Output Iterator 的模型
- **OutputIterator2**：是 Output Iterator 的模型
- **Predicate**：是 Predicate 的模型

函数参数

- **first**：待重新排序的序列第一个元素
- **last1**：待重新排序的序列中最末元素之后的一个位置
- **stencil**：模版序列的起始
- **out_true**：满足 pred 的元素的目标结果序列

飞腾信息 MetaX confidential 2024-11-18 15:00:00

- **out_false**：不满足 pred 的元素的目标结果序列
- **pred**：决定序列 [first, last) 内每个元素属于哪个分区的函数对象

前提条件

输入范围不应与任何一个输出范围重叠

返回值

p pair，其中 p.first 是从 out_true 开始的输出范围的末尾，p.second 是从 out_false 开始的输出范围的末尾

另请参阅

- http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2569.pdf
- partition_copy
- stable_partition

2.1.5.2 洗牌（Shuffling）

洗牌

```
template <typename DerivedPolicy,
  typename RandomIterator,
  typename URBG>
__host__ __device__ void
thrust::shuffle(const thrust::detail::execution_policy_base< DerivedPolicy
↪> & exec,
  RandomIterator first,
  RandomIterator last,
  URBG && g);

template <typename RandomIterator,
  typename URBG>
__host__ __device__ void
thrust::shuffle(RandomIterator first,
  RandomIterator last,
  URBG && g);

template <typename DerivedPolicy,
  typename RandomIterator,
  typename OutputIterator,
  typename URBG>
__host__ __device__ void
thrust::shuffle_copy(const thrust::detail::execution_policy_base<
↪DerivedPolicy > & exec,
  RandomIterator first,
  RandomIterator last,
  OutputIterator result,
  URBG && g);

template <typename RandomIterator,
  typename OutputIterator,
  typename URBG>
__host__ __device__ void
thrust::shuffle_copy(RandomIterator first,
  RandomIterator last,
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
  OutputIterator result,
  URBG && g);
```

thrust::shuffle

```
template <typename DerivedPolicy,
  typename RandomIterator,
  typename URBG>
__host__ __device__ void
shuffle(const thrust::detail::execution_policy_base< DerivedPolicy > &
↪exec,
  RandomIterator first,
  RandomIterator last,
  URBG && g);
```

shuffle 通过由随机引擎 g 定义的均匀伪随机排列（uniform pseudorandom permutation）来重新排序元素 [first, last)。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::host 并行化执行策略，使用 shuffle 创建随机排列：

```
#include <thrust/shuffle.h>
#include <thrust/random.h>
#include <thrust/execution_policy.h>
int A[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
const int N = sizeof(A)/sizeof(int);
thrust::default_random_engine g;
thrust::shuffle(thrust::host, A, A + N, g);
// A is now {6, 5, 8, 7, 2, 1, 4, 3, 10, 9}
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **RandomIterator**：随机访问迭代器
- **URBG**：均匀随机位生成器

函数参数

- **exec**：用于并行化的执行策略
- **first**：待洗牌的序列起始
- **last**：待洗牌的序列末尾
- **g**：一个均匀随机位生成器

另请参阅

shuffle_copy

thrust::shuffle

```
template <typename RandomIterator,
  typename URBG>
__host__ __device__ void
shuffle(RandomIterator first,
```

(下页继续)

飞曦信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
  RandomIterator last,
  URBG && g);
```

shuffle 通过由随机引擎 g 定义的均匀伪随机排列来重新排序元素 [first, last)。

以下代码片段演示了如何使用 shuffle 创建随机排列：

```
#include <thrust/shuffle.h>
#include <thrust/random.h>
int A[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
const int N = sizeof(A)/sizeof(int);
thrust::default_random_engine g;
thrust::shuffle(A, A + N, g);
// A is now {6, 5, 8, 7, 2, 1, 4, 3, 10, 9}
```

模板参数

- **RandomIterator**：随机访问迭代器
- **URBG**：均匀随机位生成器

函数参数

- **first**：待洗牌的序列起始
- **last**：待洗牌的序列末尾
- **g**：一个均匀随机位生成器

另请参阅

shuffle_copy

thrust::shuffle_copy

```
template <typename DerivedPolicy,
  typename RandomIterator,
  typename OutputIterator,
  typename URBG>
__host__ __device__ void
shuffle_copy(const thrust::detail::execution_policy_base< DerivedPolicy > &
↪ exec,
  RandomIterator first,
  RandomIterator last,
  OutputIterator result,
  URBG && g);
```

shuffle_copy 与 shuffle 的不同之处仅在于重新排序的序列被写入不同的输出序列，而不是就地写入。shuffle_copy 通过由随机引擎 g 定义的均匀伪随机排列来重新排序元素 [first, last)。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何使用 shuffle_copy 创建随机排列：

```
#include <thrust/shuffle.h>
#include <thrust/random.h>
#include <thrust/execution_policy.h>
int A[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int result[10];
const int N = sizeof(A)/sizeof(int);
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
thrust::default_random_engine g;
thrust::shuffle_copy(thrust::host, A, A + N, result, g);
// result is now {6, 5, 8, 7, 2, 1, 4, 3, 10, 9}
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **RandomIterator**：随机访问迭代器
- **OutputIterator**：是 Output Iterator 的模型
- **URBG**：均匀随机位生成器

函数参数

- **exec**：用于并行化的执行策略
- **first**：待洗牌的序列起始
- **last**：待洗牌的序列末尾
- **result**：已洗牌的目标序列
- **g**：一个均匀随机位生成器

另请参阅

shuffle

thrust::shuffle_copy

```
template <typename RandomIterator,
  typename OutputIterator,
  typename URBG>
__host__ __device__ void
shuffle_copy(RandomIterator first,
  RandomIterator last,
  OutputIterator result,
  URBG && g);
```

shuffle_copy 与 shuffle 的不同之处仅在于重新排序的序列被写入不同的输出序列，而不是就地写入。shuffle_copy 通过由随机引擎 g 定义的均匀伪随机排列来重新排序元素 [first, last)。

以下代码片段演示了如何使用 shuffle_copy 创建随机排列：

```
#include <thrust/shuffle.h>
#include <thrust/random.h>
int A[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int result[10];
const int N = sizeof(A)/sizeof(int);
thrust::default_random_engine g;
thrust::shuffle_copy(A, A + N, result, g);
// result is now {6, 5, 8, 7, 2, 1, 4, 3, 10, 9}
```

模板参数

- **RandomIterator**：随机访问迭代器
- **OutputIterator**：是 Output Iterator 的模型
- **URBG**：均匀随机位生成器

函数参数

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

- **first**：待洗牌的序列起始
- **last**：待洗牌的序列末尾
- **result**：已洗牌的目标序列
- **g**：一个均匀随机位生成器

另请参阅

shuffle

2.1.5.3 流压缩（Stream Compaction）

流压缩

```
template <typename DerivedPolicy,
  typename InputIterator,
  typename OutputIterator,
  typename Predicate>
__host__ __device__ OutputIterator
thrust::copy_if(const thrust::detail::execution_policy_base< DerivedPolicy
↪> & exec,
  InputIterator first,
  InputIterator last,
  OutputIterator result,
  Predicate pred);

template <typename InputIterator,
  typename OutputIterator,
  typename Predicate>
OutputIterator
thrust::copy_if(InputIterator first,
  InputIterator last,
  OutputIterator result,
  Predicate pred);

template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator,
  typename Predicate>
__host__ __device__ OutputIterator
thrust::copy_if(const thrust::detail::execution_policy_base< DerivedPolicy
↪> & exec,
  InputIterator1 first,
  InputIterator1 last,
  InputIterator2 stencil,
  OutputIterator result,
   Predicate pred);

 template <typename InputIterator1,
   typename InputIterator2,
   typename OutputIterator,
   typename Predicate>
 OutputIterator
 thrust::copy_if(InputIterator1 first,
   InputIterator1 last,
   InputIterator2 stencil,
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
  OutputIterator result,
  Predicate pred);

template <typename DerivedPolicy,
  typename ForwardIterator,
  typename T>
__host__ __device__ ForwardIterator
thrust::remove(const thrust::detail::execution_policy_base< DerivedPolicy
↪> & exec,
  ForwardIterator first,
  ForwardIterator last,
  const T & value);

template <typename ForwardIterator,
   typename T>
 ForwardIterator
 thrust::remove(ForwardIterator first,
   ForwardIterator last,
   const T & value);

 template <typename DerivedPolicy,
   typename InputIterator,
   typename OutputIterator,
   typename T>
 __host__ __device__ OutputIterator
 thrust::remove_copy(const thrust::detail::execution_policy_base<
↪DerivedPolicy > & exec,
   InputIterator first,
   InputIterator last,
   OutputIterator result,
   const T & value);

 template <typename InputIterator,
   typename OutputIterator,
   typename T>
 OutputIterator
 thrust::remove_copy(InputIterator first,
   InputIterator last,
   OutputIterator result,
   const T & value);

 template <typename DerivedPolicy,
   typename ForwardIterator,
   typename Predicate>
 __host__ __device__ ForwardIterator
 thrust::remove_if(const thrust::detail::execution_policy_base<
↪DerivedPolicy > & exec,
   ForwardIterator first,
   ForwardIterator last,
   Predicate pred);

 template <typename ForwardIterator,
   typename Predicate>
 ForwardIterator
 thrust::remove_if(ForwardIterator first,
   ForwardIterator last,
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
    Predicate pred);

  template <typename DerivedPolicy,
    typename InputIterator,
    typename OutputIterator,
    typename Predicate>
  __host__ __device__ OutputIterator
  thrust::remove_copy_if(const thrust::detail::execution_policy_base<
↪DerivedPolicy > & exec,
    InputIterator first,
    InputIterator last,
    OutputIterator result,
    Predicate pred);

  template <typename InputIterator,
    typename OutputIterator,
    typename Predicate>
  OutputIterator
  thrust::remove_copy_if(InputIterator first,
    InputIterator last,
    OutputIterator result,
    Predicate pred);

  template <typename DerivedPolicy,
    typename ForwardIterator,
    typename InputIterator,
    typename Predicate>
  __host__ __device__ ForwardIterator
  thrust::remove_if(const thrust::detail::execution_policy_base<
↪DerivedPolicy > & exec,
    ForwardIterator first,
    ForwardIterator last,
    InputIterator stencil,
    Predicate pred);

  template <typename ForwardIterator,
    typename InputIterator,
    typename Predicate>
  ForwardIterator
  thrust::remove_if(ForwardIterator first,
    ForwardIterator last,
    InputIterator stencil,
    Predicate pred);

  template <typename DerivedPolicy,
    typename InputIterator1,
    typename InputIterator2,
    typename OutputIterator,
    typename Predicate>
  __host__ __device__ OutputIterator
  thrust::remove_copy_if(const thrust::detail::execution_policy_base<
↪DerivedPolicy > & exec,
    InputIterator1 first,
    InputIterator1 last,
    InputIterator2 stencil,
    OutputIterator result,
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
    Predicate pred);

  template <typename InputIterator1,
    typename InputIterator2,
    typename OutputIterator,
    typename Predicate>
OutputIterator
thrust::remove_copy_if(InputIterator1 first,
  InputIterator1 last,
  InputIterator2 stencil,
  OutputIterator result,
  Predicate pred);

template <typename DerivedPolicy,
  typename ForwardIterator>
__host__ __device__ ForwardIterator
thrust::unique(const thrust::detail::execution_policy_base< DerivedPolicy >
 ↪ & exec,
  ForwardIterator first,
  ForwardIterator last);

template <typename ForwardIterator>
ForwardIterator thrust::unique(ForwardIterator first,
  ForwardIterator last);

template <typename DerivedPolicy,
  typename ForwardIterator,
  typename BinaryPredicate>
__host__ __device__ ForwardIterator
thrust::unique(const thrust::detail::execution_policy_base< DerivedPolicy >
 ↪ & exec,
  ForwardIterator first,
  ForwardIterator last,
  BinaryPredicate binary_pred);

template <typename ForwardIterator,
  typename BinaryPredicate>
ForwardIterator
thrust::unique(ForwardIterator first,
  ForwardIterator last,
  BinaryPredicate binary_pred);

template <typename DerivedPolicy,
  typename InputIterator,
  typename OutputIterator>
__host__ __device__ OutputIterator
thrust::unique_copy(const thrust::detail::execution_policy_base<
 ↪DerivedPolicy > & exec,
  InputIterator first,
  InputIterator last,
  OutputIterator result);

template <typename InputIterator,
  typename OutputIterator>
OutputIterator
thrust::unique_copy(InputIterator first,
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
  InputIterator last,
  OutputIterator result);

template <typename DerivedPolicy,
  typename InputIterator,
  typename OutputIterator,
  typename BinaryPredicate>
__host__ __device__ OutputIterator
thrust::unique_copy(const thrust::detail::execution_policy_base<
↪DerivedPolicy > & exec,
  InputIterator first,
  InputIterator last,
  OutputIterator result,
  BinaryPredicate binary_pred);

template <typename InputIterator,
  typename OutputIterator,
  typename BinaryPredicate>
OutputIterator
thrust::unique_copy(InputIterator first,
  InputIterator last,
  OutputIterator result,
  BinaryPredicate binary_pred);

template <typename DerivedPolicy,
  typename ForwardIterator1,
  typename ForwardIterator2>
__host__ __device__ thrust::pair< ForwardIterator1, ForwardIterator2 >
thrust::unique_by_key(const thrust::detail::execution_policy_base<
↪DerivedPolicy > & exec,
  ForwardIterator1 keys_first,
  ForwardIterator1 keys_last,
  ForwardIterator2 values_first);

template <typename ForwardIterator1,
  typename ForwardIterator2>
thrust::pair< ForwardIterator1, ForwardIterator2 >
thrust::unique_by_key(ForwardIterator1 keys_first,
  ForwardIterator1 keys_last,
  ForwardIterator2 values_first);

template <typename DerivedPolicy,
  typename ForwardIterator1,
  typename ForwardIterator2,
  typename BinaryPredicate>
__host__ __device__ thrust::pair< ForwardIterator1, ForwardIterator2 >
thrust::unique_by_key(const thrust::detail::execution_policy_base<
↪DerivedPolicy > & exec,
  ForwardIterator1 keys_first,
  ForwardIterator1 keys_last,
  ForwardIterator2 values_first,
  BinaryPredicate binary_pred);

template <typename ForwardIterator1,
  typename ForwardIterator2,
  typename BinaryPredicate>
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
thrust::pair< ForwardIterator1, ForwardIterator2 >
thrust::unique_by_key(ForwardIterator1 keys_first,
  ForwardIterator1 keys_last,
  ForwardIterator2 values_first,
  BinaryPredicate binary_pred);

template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator1,
  typename OutputIterator2>
__host__ __device__ thrust::pair< OutputIterator1, OutputIterator2 >
thrust::unique_by_key_copy(const thrust::detail::execution_policy_base<
 ↪DerivedPolicy > & exec,
  InputIterator1 keys_first,
  InputIterator1 keys_last,
  InputIterator2 values_first,
  OutputIterator1 keys_result,
  OutputIterator2 values_result);

template <typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator1,
  typename OutputIterator2>
thrust::pair< OutputIterator1, OutputIterator2 >
thrust::unique_by_key_copy(InputIterator1 keys_first,
  InputIterator1 keys_last,
  InputIterator2 values_first,
  OutputIterator1 keys_result,
  OutputIterator2 values_result);

template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator1,
  typename OutputIterator2,
  typename BinaryPredicate>
__host__ __device__ thrust::pair< OutputIterator1, OutputIterator2 >
thrust::unique_by_key_copy(const thrust::detail::execution_policy_base<
 ↪DerivedPolicy > & exec,
  InputIterator1 keys_first,
  InputIterator1 keys_last,
  InputIterator2 values_first,
  OutputIterator1 keys_result,
  OutputIterator2 values_result,
  BinaryPredicate binary_pred);

template <typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator1,
  typename OutputIterator2,
  typename BinaryPredicate>
thrust::pair< OutputIterator1, OutputIterator2 >
thrust::unique_by_key_copy(InputIterator1 keys_first,
  InputIterator1 keys_last,
  InputIterator2 values_first,
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
  OutputIterator1 keys_result,
  OutputIterator2 values_result,
  BinaryPredicate binary_pred);

template <typename DerivedPolicy,
  typename ForwardIterator,
  typename BinaryPredicate>
__host__ __device__ thrust::iterator_traits< ForwardIterator >::difference_
↪type
thrust::unique_count(const thrust::detail::execution_policy_base<
↪DerivedPolicy > & exec,
  ForwardIterator first,
  ForwardIterator last,
  BinaryPredicate binary_pred);

template <typename DerivedPolicy,
  typename ForwardIterator>
__host__ __device__ thrust::iterator_traits< ForwardIterator >::difference_
↪type
thrust::unique_count(const thrust::detail::execution_policy_base<
↪DerivedPolicy > & exec,
  ForwardIterator first,
  ForwardIterator last);

template <typename ForwardIterator,
  typename BinaryPredicate>
__host__ __device__ thrust::iterator_traits< ForwardIterator >::difference_
↪type
thrust::unique_count(ForwardIterator first,
  ForwardIterator last,
  BinaryPredicate binary_pred);

template <typename ForwardIterator>
__host__ __device__ thrust::iterator_traits< ForwardIterator >::difference_
↪type
thrust::unique_count(ForwardIterator first,
  ForwardIterator last);
```

thrust::copy_if

```
template <typename DerivedPolicy,
  typename InputIterator,
  typename OutputIterator,
  typename Predicate>
__host__ __device__ OutputIterator
copy_if(const thrust::detail::execution_policy_base< DerivedPolicy > &
↪exec,
  InputIterator first,
  InputIterator last,
  OutputIterator result,
  Predicate pred);
```

此版本的 copy_if 将范围 [first,last) 内除了使得 pred 为 false 之外的其他元素复制到从 result 开始的范围。copy_if 是稳定的，指的是复制的元素的相对顺序不变。

飞谱信息 MetaX Confidential 2024-11-18 15:00:00

具体而言，对于满足 0 <= n < last-first 的每个整数 n ，copy_if 执行赋值 *result = *(first+n)，并且如果 pred(*(first+n)) ，result 前进一个位置。否则，不会发生赋值，result 也不会提前。

算法的执行由 system 确定并行化。

以下代码片段演示了如何通过 thrust::host 并行化执行策略，使用 copy_if 执行流压缩以将偶数复制到输出范围：

```
#include <thrust/copy.h>
#include <thrust/execution_policy.h>
...
struct is_even
{
__host__ __device__
bool operator()(const int x)
{
   return (x % 2) == 0;
}
};
...
const int N = 6;
int V[N] = {-2, 0, -1, 0, 1, 2};
int result[4];

thrust::copy_if(thrust::host, V, V + N, result, is_even());

// V remains {-2, 0, -1, 0, 1, 2}
// result is now {-2, 0, 0, 2}
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **InputIterator**：是 Input Iterator 的模型，InputIterator 的 value_type 可转换为 Predicate 的 argument_type
- **OutputIterator**：是 Output Iterator 的模型
- **Predicate**：是 Predicate 的模型

函数参数

- **exec**：用于并行化的执行策略
- **first**：待从中复制的序列的起始
- **last**：待从中复制的序列的末尾
- **result**：待复制到的序列的起始
- **pred**：对范围 [first, last) 内每个值进行测试的谓词

前提条件

范围 [first, last) 不应与范围 [result, result + (last - first)) 重叠

返回值

result + n，其中 n 等于 pred 在范围 [first, last) 内计算为 true 的次数

另请参阅

remove_copy_if

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

thrust::copy_if

```
template <typename InputIterator,
  typename OutputIterator,
  typename Predicate>
OutputIterator
copy_if(InputIterator first,
  InputIterator last,
  OutputIterator result,
  Predicate pred);
```

此版本的 copy_if 将范围 [first,last) 内除了使得 pred 为 false 之外的其他元素复制到从 result 开始的范围。copy_if 是稳定的，指的是复制的元素的相对顺序不变。

具体而言，对于满足 0 <= n < last-first 的每个整数 n ，copy_if 执行赋值 *result = *(first+n)，并且如果 pred(*(first+n)) ，result 前进一个位置。否则，不会发生赋值，result 也不会提前。

以下代码片段演示了如何使用 copy_if 执行流压缩以将偶数复制到输出范围：

```
#include <thrust/copy.h>
...
struct is_even
{
__host__ __device__
bool operator()(const int x)
{
   return (x % 2) == 0;
}
};
...
const int N = 6;
int V[N] = {-2, 0, -1, 0, 1, 2};
int result[4];

thrust::copy_if(V, V + N, result, is_even());

// V remains {-2, 0, -1, 0, 1, 2}
// result is now {-2, 0, 0, 2}
```

模板参数

- **InputIterator**：是 Input Iterator 的模型，InputIterator 的 value_type 可转换为 Predicate 的 argument_type
- **OutputIterator**：是 Output Iterator 的模型
- **Predicate**：是 Predicate 的模型

函数参数

- **first**：待从中复制的序列的起始
- **last**：待从中复制的序列的末尾
- **result**：待复制到的序列的起始
- **pred**：对范围 [first, last) 内每个值进行测试的谓词

前提条件

范围 [first, last) 不应与范围 [result, result + (last - first)) 重叠

返回值

result + n，其中 n 等于 pred 在范围 [first, last) 内计算为 true 的次数

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

另请参阅

remove_copy_if

thrust::copy_if

```
template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator,
  typename Predicate>
__host__ __device__ OutputIterator
copy_if(const thrust::detail::execution_policy_base< DerivedPolicy > &
↪exec,
  InputIterator1 first,
  InputIterator1 last,
  InputIterator2 stencil,
  OutputIterator result,
  Predicate pred);
```

此版本的 copy_if 将范围 [first,last) 内除了其相应模版元素使得 pred 为 false 之外的其他元素复制到从 result 开始的范围。copy_if 是稳定的，指的是复制的元素的相对顺序不变。

具体而言，对于满足 0 <= n < last-first 的每个整数 n ，copy_if 执行赋值 *result = *(first+n)，并且如果 pred(*(stencil+n)) ，result 前进一个位置。否则，不会发生赋值，result 也不会提前。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::host 并行化执行策略，当对应的模版元素为偶数时，使用 copy_if 执行流压缩以将数字复制到输出范围：

```
#include <thrust/copy.h>
#include <thrust/execution_policy.h>
...
struct is_even
{
__host__ __device__
bool operator()(const int x)
{
  return (x % 2) == 0;
}
};
...
int N = 6;
int data[N]    = { 0, 1,  2, 3, 4, 5};
int stencil[N] = {-2, 0, -1, 0, 1, 2};
int result[4];

thrust::copy_if(thrust::host, data, data + N, stencil, result, is_even());

// data remains    = { 0, 1,  2, 3, 4, 5};
// stencil remains = {-2, 0, -1, 0, 1, 2};
// result is now     { 0, 1,  3, 5}
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **InputIterator1**：是 Input Iterator 的模型

飞诺信息 MetaX Confidential 2024-11-18 15:00:00

- **InputIterator2**：是 Input Iterator 的模型，InputIterator2 的 value_type 可转换为 Predicate 的 argument_type
- **OutputIterator**：是 Output Iterator 的模型
- **Predicate**：是 Predicate 的模型

函数参数

- **exec**：用于并行化的执行策略
- **first**：待从中复制的序列的起始
- **last**：待从中复制的序列的末尾
- **stencil**：模版序列的起始
- **result**：待复制到的序列的起始
- **pred**：对范围 [stencil, stencil + (last-first)) 内每个值进行测试的谓词

前提条件

- 范围 [first, last) 不应与范围 [result, result + (last - first)) 重叠
- 范围 [stencil, stencil + (last - first)) 不应与范围 [result, result + (last - first)) 重叠

返回值

result + n，其中 n 等于 pred 在范围 [stencil, stencil + (last-first)) 内计算为 true 的次数

另请参阅

remove_copy_if

thrust::copy_if

```
template <typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator,
  typename Predicate>
OutputIterator
copy_if(InputIterator1 first,
  InputIterator1 last,
  InputIterator2 stencil,
  OutputIterator result,
  Predicate pred);
```

此版本的 copy_if 将范围 [first,last) 内除了其相应模版元素使得 pred 为 false 之外的其他元素复制到从 result 开始的范围。copy_if 是稳定的，指的是复制的元素的相对顺序不变。

具体而言，对于满足 0 <= n < last-first 的每个整数 n ，copy_if 执行赋值 *result = *(first+n)，并且如果 pred(*(stencil+n)) ，result 前进一个位置。否则，不会发生赋值，result 也不会提前。

以下代码片段演示了当对应的模版元素为偶数时，使用 copy_if 执行流压缩以将数字复制到输出范围：

```
#include <thrust/copy.h>
...
struct is_even
{
__host__ __device__
bool operator()(const int x)
{
   return (x % 2) == 0;
}
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
};
...
int N = 6;
int data[N]    = { 0, 1,  2, 3, 4, 5};
int stencil[N] = {-2, 0, -1, 0, 1, 2};
int result[4];

thrust::copy_if(data, data + N, stencil, result, is_even());

// data remains    = { 0, 1,  2, 3, 4, 5};
// stencil remains = {-2, 0, -1, 0, 1, 2};
// result is now     { 0, 1,  3, 5}
```

模板参数

- **InputIterator1**：是 Input Iterator 的模型
- **InputIterator2**：是 Input Iterator 的模型，InputIterator2 的 value_type 可转换为 Predicate 的 argument_type
- **OutputIterator**：是 Output Iterator 的模型
- **Predicate**：是 Predicate 的模型

函数参数

- **first**：待从中复制的序列的起始
- **last**：待从中复制的序列的末尾
- **stencil**：模版序列的起始
- **result**：待复制到的序列的起始
- **pred**：对范围 [stencil, stencil + (last-first)) 内每个值进行测试的谓词

前提条件

- 范围 [first, last) 不应与范围 [result, result + (last - first)) 重叠
- 范围 [stencil, stencil + (last - first)) 不应与范围 [result, result + (last - first)) 重叠

返回值

result + n，其中 n 等于 pred 在范围 [stencil, stencil + (last-first)) 内计算为 true 的次数

另请参阅

remove_copy_if

thrust::remove

```
template <typename DerivedPolicy,
  typename ForwardIterator,
  typename T>
__host__ __device__ ForwardIterator
remove(const thrust::detail::execution_policy_base< DerivedPolicy > & exec,
  ForwardIterator first,
  ForwardIterator last,
  const T & value);
```

remove 从范围 [first, last) 内删除所有等于 value 的值。即，remove 返回迭代器 new_last ，使得范围 [first, new_last) 不包含等于 value 的元素。范围 [new_first,last) 内的迭代器仍然是可解引用的，但它们所指向的元素是未指定的。remove 是稳定的，指的是不等于 value 的元素的相对顺序保持不变。

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::host 并行化执行策略，使用 remove 从范围内删除感兴趣的数字：

```
#include <thrust/remove.h>
#include <thrust/execution_policy.h>
...
const int N = 6;
int A[N] = {3, 1, 4, 1, 5, 9};
int *new_end = thrust::remove(A, A + N, 1);
// The first four values of A are now {3, 4, 5, 9}
// Values beyond new_end are unspecified
```

备注

removal 含义有些微妙。remove 不会销毁任何迭代器，也不会更改 first 和 last 之间的距离。（它不可能做任何类似的事情。） 所以，比如，如果 V 是一个 device_vector ，remove(V.begin(), V.end(), 0) 不会改变 V.size()：V 将包含与以前一样多的元素。

remove 返回一个迭代器，该迭代器在从结果范围内删除元素后指向结果范围的末尾；由此可见，该迭代器之后的元素不在感兴趣范围内，可能会被丢弃。如果要从 Sequence 中删除元素，可以简单地擦除它们。即，从 Sequence 中删除元素的合理方法是 S.erase(remove(S.begin(), S.end(), x), S.end())。

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **ForwardIterator**：是 Forward Iterator 的模型，且 ForwardIterator 是可变的
- **T**：是 Equality Comparable 的模型，且类型 T 的对象可以和 ForwardIterator 的 value_type 的对象进行相等性比较

函数参数

- **exec**：用于并行化的执行策略
- **first**：感兴趣范围的起始
- **last**：感兴趣范围的末尾
- **value**：待从范围 [first, last) 内删除的值。等于 value 的元素将从序列中删除

返回值

ForwardIterator ，指向不等于 value 的元素结果范围的末尾

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/remove
- remove_if
- remove_copy
- remove_copy_if

thrust::remove

```
template <typename ForwardIterator,
  typename T>
ForwardIterator
remove(ForwardIterator first,
  ForwardIterator last,
  const T & value);
```

飞腾信息 MetaX confidential 2024-11-18 15:00:00

remove 从范围 [first, last) 内删除所有等于 value 的值。即，remove 返回迭代器 new_last ，使得范围 [first, new_last) 不包含等于 value 的元素。范围 [new_first,last) 内的迭代器仍然是可解引用的，但它们所指向的元素是未指定的。remove 是稳定的，指的是不等于 value 的元素的相对顺序保持不变。

以下代码片段演示了如何使用 remove 从范围内删除感兴趣的数字：

```
#include <thrust/remove.h>
...
const int N = 6;
int A[N] = {3, 1, 4, 1, 5, 9};
int *new_end = thrust::remove(A, A + N, 1);
// The first four values of A are now {3, 4, 5, 9}
// Values beyond new_end are unspecified
```

备注

removal 含义有些微妙。remove 不会销毁任何迭代器，也不会更改 first 和 last 之间的距离。(它不可能做任何类似的事情。) 所以，比如，如果 V 是一个 device_vector ，remove(V.begin(), V.end(), 0) 不会改变 V.size()：V 将包含与以前一样多的元素。

remove 返回一个迭代器，该迭代器在从结果范围内删除元素后指向结果范围的末尾；由此可见，该迭代器之后的元素不在感兴趣范围内，可能会被丢弃。如果要从 Sequence 中删除元素，可以简单地擦除它们。即，从 Sequence 中删除元素的合理方法是 S.erase(remove(S.begin(), S.end(), x), S.end())。

模板参数

- **ForwardIterator**：是 Forward Iterator 的模型，且 ForwardIterator 是可变的
- **T**：是 Equality Comparable 的模型，且类型 T 的对象可以和 ForwardIterator 的 value_type 的对象进行相等性比较

函数参数

- **first**：感兴趣范围的起始
- **last**：感兴趣范围的末尾
- **value**：待从范围 [first, last) 内删除的值。等于 value 的元素将从序列中删除

返回值

ForwardIterator ，指向不等于 value 的元素结果范围的末尾

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/remove
- remove_if
- remove_copy
- remove_copy_if

thrust::remove_copy

```
template <typename DerivedPolicy,
  typename InputIterator,
  typename OutputIterator,
  typename T>
__host__ __device__ OutputIterator
remove_copy(const thrust::detail::execution_policy_base< DerivedPolicy > &
↪exec,
  InputIterator first,
  InputIterator last,
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
  OutputIterator result,
  const T & value);
```

remove_copy 将不等于 value 的元素从范围 [first, last) 复制到从 result 开始的范围。返回值是结果范围的末尾。此运算是稳定的，指的是复制的元素的相对顺序与范围 [first, last) 内的顺序相同。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::host 并行化执行策略，使用 remove_copy 将数字序列复制到输出范围，同时忽略感兴趣的值：

```
#include <thrust/remove.h>
#include <thrust/execution_policy.h>
...
const int N = 6;
int V[N] = {-2, 0, -1, 0, 1, 2};
int result[N-2];
thrust::remove_copy(thrust::host, V, V + N, result, 0);
// V remains {-2, 0, -1, 0, 1, 2}
// result is now {-2, -1, 1, 2}
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **InputIterator**：是 Input Iterator 的模型，InputIterator 的 value_type 可转换为 OutputIterator 的 value_types 集合中的类型
- **OutputIterator**：是 Output Iterator 的模型
- **T**：是 Equality Comparable 的模型，且类型 T 的对象可以和 InputIterator 的 value_type 的对象进行相等性比较

函数参数

- **exec**：用于并行化的执行策略
- **first**：感兴趣范围的起始
- **last**：感兴趣范围的末尾
- **result**：结果范围被复制到从此位置开始的序列中
- **value**：待从范围 内忽略的值

前提条件

范围 [first, last) 不应与范围 [result, result + (last - first)) 重叠

返回值

OutputIterator ，指向不等于 value 的元素结果范围的末尾

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/remove_copy
- remove
- remove_if
- remove_copy_if

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

thrust::remove_copy

```
template <typename InputIterator,
  typename OutputIterator,
  typename T>
OutputIterator
remove_copy(InputIterator first,
  InputIterator last,
  OutputIterator result,
  const T & value);
```

remove_copy 将不等于 value 的元素从范围 [first, last) 复制到从 result 开始的范围。返回值是结果范围的末尾。此运算是稳定的，指的是复制的元素的相对顺序与范围 [first, last) 内的顺序相同。

以下代码片段演示了如何使用 remove_copy 将数字序列复制到输出范围，同时忽略感兴趣的值：

```
#include <thrust/remove.h>
...
const int N = 6;
int V[N] = {-2, 0, -1, 0, 1, 2};
int result[N-2];
thrust::remove_copy(V, V + N, result, 0);
// V remains {-2, 0, -1, 0, 1, 2}
// result is now {-2, -1, 1, 2}
```

模板参数

- **InputIterator**：是 Input Iterator 的模型，InputIterator 的 value_type 可转换为 OutputIterator 的 value_types 集合中的类型
- **OutputIterator**：是 Output Iterator 的模型
- **T**：是 Equality Comparable 的模型，且类型 T 的对象可以和 InputIterator 的 value_type 的对象进行相等性比较

函数参数

- **first**：感兴趣范围的起始
- **last**：感兴趣范围的末尾
- **result**：结果范围被复制到从此位置开始的序列中
- **value**：待从范围 内忽略的值

前提条件

范围 [first, last) 不应与范围 [result, result + (last - first)) 重叠

返回值

OutputIterator ，指向不等于 value 的元素结果范围的末尾

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/remove_copy
- remove
- remove_if
- remove_copy_if

沐曦信息 MetaX Confidential 2024-11-18 15:00:00

thrust::remove_if

```
template <typename DerivedPolicy,
  typename ForwardIterator,
  typename Predicate>
__host__ __device__ ForwardIterator
remove_if(const thrust::detail::execution_policy_base< DerivedPolicy > &
↪exec,
  ForwardIterator first,
  ForwardIterator last,
  Predicate pred);
```

remove_if 从范围 [first, last) 内删除使得 pred(x) 为 true 的每个元素 x。即，remove_if 返回迭代器 new_last ，使得范围 [first,new_last) 不包含满足 pred 为 true 的元素。范围 [new_last,last) 内的迭代器仍然是可解引用的，但它们所指向的元素是未指定的。remove_if 是稳定的，指的是未被删除的元素的相对顺序不变。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::host 并行化执行策略，使用 remove_if 从整数数组中删除所有偶数：

```
#include <thrust/remove.h>
#include <thrust/execution_policy.h>
...
struct is_even
{
__host__ __device__
bool operator()(const int x)
{
   return (x % 2) == 0;
}
};
...
const int N = 6;
int A[N] = {1, 4, 2, 8, 5, 7};
int *new_end = thrust::remove_if(thrust::host, A, A + N, is_even());
// The first three values of A are now {1, 5, 7}
// Values beyond new_end are unspecified
```

备注

removal 含义有些微妙。remove_if 不会销毁任何迭代器，也不会更改 first 和 last 之间的距离。（它不可能做任何类似的事情。） 所以，比如，如果 V 是一个 device_vector ，remove_if(V.begin(), V.end(), pred) 不会改变 V.size()：V 将包含与以前一样多的元素。

remove_if 返回一个迭代器，该迭代器在从结果范围内删除元素后指向结果范围的末尾；由此可见，该迭代器之后的元素不在感兴趣范围内，可能会被丢弃。如果要从 Sequence 中删除元素，可以简单地擦除它们。即，从 Sequence 中删除元素的合理方法是 S.erase(remove_if(S.begin(), S.end(), pred), S.end())。

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **ForwardIterator**：是 Forward Iterator 的模型，ForwardIterator 是可变的，并且 ForwardIterator 的 value_type 可转换为 Predicate 的 argument_type
- **Predicate**：是 Predicate 的模型

函数参数

- **exec**：用于并行化的执行策略

飞谱信息 MetaX Confidential 2021-11-18 15:00:00

- **first**：感兴趣范围的起始
- **last**：感兴趣范围的末尾
- **pred**：计算范围 [first,last) 内每个元素的值的谓词。将 pred 计算为 true 的元素从序列中删除

返回值

ForwardIterator ，指向 pred 计算为 true 的元素结果范围的末尾

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/remove
- remove
- remove_copy
- remove_copy_if

thrust::remove_if

```
template <typename ForwardIterator,
  typename Predicate>
ForwardIterator
remove_if(ForwardIterator first,
  ForwardIterator last,
  Predicate pred);
```

remove_if 从范围 [first, last) 内删除使得 pred(x) 为 true 的每个元素 x。即，remove_if 返回迭代器 new_last ，使得范围 [first,new_last) 不包含满足 pred 为 true 的元素。范围 [new_last,last) 内的迭代器仍然是可解引用的，但它们所指向的元素是未指定的。remove_if 是稳定的，指的是未被删除的元素的相对顺序不变。

以下代码片段演示了如何使用 remove_if 从整数数组中删除所有偶数：

```
#include <thrust/remove.h>
...
struct is_even
{
__host__ __device__
bool operator()(const int x)
{
  return (x % 2) == 0;
}
};
...
const int N = 6;
int A[N] = {1, 4, 2, 8, 5, 7};
int *new_end = thrust::remove_if(A, A + N, is_even());
// The first three values of A are now {1, 5, 7}
// Values beyond new_end are unspecified
```

备注

removal 含义有些微妙。remove_if 不会销毁任何迭代器，也不会更改 first 和 last 之间的距离。（它不可能做任何类似的事情。） 所以，比如，如果 V 是一个 device_vector ，remove_if(V.begin(), V.end(), pred) 不会改变 V.size()：V 将包含与以前一样多的元素。

remove_if 返回一个迭代器，该迭代器在从结果范围内删除元素后指向结果范围的末尾；由此可见，该迭代器之后的元素不在感兴趣范围内，可能会被丢弃。如果要从 Sequence 中删除元素，可以简单地擦除它们。即，从 Sequence 中删除元素的合理方法是 S.erase(remove_if(S.begin(), S.end(), pred), S.end())。

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

模板参数

- **ForwardIterator**：是 Forward Iterator 的模型，ForwardIterator 是可变的，并且 ForwardIterator 的 value_type 可转换为 Predicate 的 argument_type
- **Predicate**：是 Predicate 的模型

函数参数

- **first**：感兴趣范围的起始
- **last**：感兴趣范围的末尾
- **pred**：计算范围 [first,last) 内每个元素的值的谓词。将 pred 计算为 true 的元素从序列中删除

返回值

ForwardIterator ，指向 pred 计算为 true 的元素结果范围的末尾

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/remove
- remove
- remove_copy
- remove_copy_if

thrust::remove_copy_if

```
template <typename DerivedPolicy,
  typename InputIterator,
  typename OutputIterator,
  typename Predicate>
__host__ __device__ OutputIterator
remove_copy_if(const thrust::detail::execution_policy_base< DerivedPolicy >
↪ & exec,
  InputIterator first,
  InputIterator last,
  OutputIterator result,
  Predicate pred);
```

remove_copy_if 将范围 [first,last) 内除了使得 pred 为 true 之外的其他元素复制到从 result 开始的范围。返回值是结果范围的末尾。此运算是稳定的，指的是复制的元素的相对顺序与范围 [first,last) 内的顺序相同。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::host 并行化执行策略，使用 remove_copy_if 将数字序列复制到输出范围，同时忽略偶数：

```
#include <thrust/remove.h>
#include <thrust/execution_policy.h>
...
struct is_even
{
__host__ __device__
bool operator()(const int x)
{
   return (x % 2) == 0;
}
};
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
...
const int N = 6;
int V[N] = {-2, 0, -1, 0, 1, 2};
int result[2];
thrust::remove_copy_if(thrust::host, V, V + N, result, is_even());
// V remains {-2, 0, -1, 0, 1, 2}
// result is now {-1, 1}
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **InputIterator**：是 Input Iterator 的模型，InputIterator 的 value_type 可转换为 OutputIterator 的 value_types 集合中的类型和 Predicate 的 argument_type
- **OutputIterator**：是 Output Iterator 的模型
- **Predicate**：是 Predicate 的模型

函数参数

- **exec**：用于并行化的执行策略
- **first**：感兴趣范围的起始
- **last**：感兴趣范围的末尾
- **result**：结果范围被复制到从此位置开始的序列中
- **pred**：计算范围 [first,last) 内每个元素的值的谓词。不会将 pred 计算为 false 的元素复制到结果序列中

前提条件

范围 [first, last) 不应与范围 [result, result + (last - first)) 重叠

返回值

指向结果范围末尾的 OutputIterator

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/remove_copy
- remove
- remove_copy
- remove_if

thrust::remove_copy_if

```
template <typename InputIterator,
  typename OutputIterator,
  typename Predicate>
OutputIterator
remove_copy_if(InputIterator first,
  InputIterator last,
  OutputIterator result,
  Predicate pred);
```

remove_copy_if 将范围 [first,last) 内除了使得 pred 为 true 之外的其他元素复制到从 result 开始的范围。返回值是结果范围的末尾。此运算是稳定的，指的是复制的元素的相对顺序与范围 [first,last) 内的顺序相同。

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

以下代码片段演示了如何使用 remove_copy_if 将数字序列复制到输出范围，同时忽略偶数：

```
#include <thrust/remove.h>
...
struct is_even
{
__host__ __device__
bool operator()(const int x)
{
   return (x % 2) == 0;
}
};
...
const int N = 6;
int V[N] = {-2, 0, -1, 0, 1, 2};
int result[2];
thrust::remove_copy_if(V, V + N, result, is_even());
// V remains {-2, 0, -1, 0, 1, 2}
// result is now {-1, 1}
```

模板参数

- **InputIterator**：是 Input Iterator 的模型，InputIterator 的 value_type 可转换为 OutputIterator 的 value_types 集合中的类型和 Predicate 的 argument_type
- **OutputIterator**：是 Output Iterator 的模型
- **Predicate**：是 Predicate 的模型

函数参数

- **first**：感兴趣范围的起始
- **last**：感兴趣范围的末尾
- **result**：结果范围被复制到从此位置开始的序列中
- **pred**：计算范围 [first,last) 内每个元素的值的谓词。不会将 pred 计算为 false 的元素复制到结果序列中

前提条件

范围 [first, last) 不应与范围 [result, result + (last - first)) 重叠

返回值

指向结果范围末尾的 OutputIterator

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/remove_copy
- remove
- remove_copy
- remove_if

thrust::remove_if

```
template <typename DerivedPolicy,
  typename ForwardIterator,
  typename InputIterator,
  typename Predicate>
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
__host__ __device__ ForwardIterator
remove_if(const thrust::detail::execution_policy_base< DerivedPolicy > &
↪exec,
  ForwardIterator first,
  ForwardIterator last,
  InputIterator stencil,
  Predicate pred);
```

remove_if 从范围 [first, last) 内删除使得 pred(x) 为 true 的每个元素 x。即，remove_if 返回迭代器 new_last ，使得范围 [first, new_last) 不包含对应模板值的 pred 为 true 的元素。范围 [new_last,last) 内的迭代器仍然是可解引用的，但它们所指向的元素是未指定的。remove_if 是稳定的，指的是未被删除的元素的相对顺序不变。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::host 并行化执行策略，使用 remove_if 从整数数组中删除特定元素：

```
#include <thrust/remove.h>
#include <thrust/execution_policy.h>
...
const int N = 6;
int A[N] = {1, 4, 2, 8, 5, 7};
int S[N] = {0, 1, 1, 1, 0, 0};

int *new_end = thrust::remove_if(thrust::host, A, A + N, S,
↪thrust::identity<int>());
// The first three values of A are now {1, 5, 7}
// Values beyond new_end are unspecified
```

备注

范围 [first, last) 不可与范围 [stencil, stencil + (last - first)) 重叠

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **ForwardIterator**：是 Forward Iterator 的模型，且 ForwardIterator 是可变的
- **InputIterator**：是 Input Iterator 的模型，InputIterator 的 value_type 可转换为 Predicate 的 argument_type
- **Predicate**：是 Predicate 的模型

函数参数

- **exec**：用于并行化的执行策略
- **first**：感兴趣范围的起始
- **last**：感兴趣范围的末尾
- **stencil**：模版序列的起始
- **pred**：计算范围 [stencil, stencil + (last - first)) 内每个元素的值的谓词。将 pred 计算为 true 的元素从序列 [first, last) 中删除

前提条件

- 范围 [first, last) 不应与范围 [result, result + (last - first)) 重叠
- 范围 [stencil, stencil + (last - first)) 不应与范围 [result, result + (last - first)) 重叠

返回值

ForwardIterator ，指向 pred 计算为 true 的元素结果范围的末尾

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/remove
- remove
- remove_copy
- remove_copy_if

thrust::remove_if

```
template <typename ForwardIterator,
  typename InputIterator,
  typename Predicate>
ForwardIterator
remove_if(ForwardIterator first,
  ForwardIterator last,
  InputIterator stencil,
  Predicate pred);
```

remove_if 从范围 [first, last) 内删除使得 pred(x) 为 true 的每个元素 x。即，remove_if 返回迭代器 new_last ，使得范围 [first, new_last) 不包含对应模板值的 pred 为 true 的元素。范围 [new_last,last) 内的迭代器仍然是可解引用的，但它们所指向的元素是未指定的。remove_if 是稳定的，指的是未被删除的元素的相对顺序不变。

以下代码片段演示了如何使用 remove_if 从整数数组中删除特定元素：

```
#include <thrust/remove.h>
...
const int N = 6;
int A[N] = {1, 4, 2, 8, 5, 7};
int S[N] = {0, 1, 1, 1, 0, 0};

int *new_end = thrust::remove_if(A, A + N, S, thrust::identity<int>());
// The first three values of A are now {1, 5, 7}
// Values beyond new_end are unspecified
```

备注

范围 [first, last) 不可与范围 [stencil, stencil + (last - first)) 重叠

模板参数

- **ForwardIterator**：是 Forward Iterator 的模型，且 ForwardIterator 是可变的
- **InputIterator**：是 Input Iterator 的模型，InputIterator 的 value_type 可转换为 Predicate 的 argument_type
- **Predicate**：是 Predicate 的模型

函数参数

- **first**：感兴趣范围的起始
- **last**：感兴趣范围的末尾
- **stencil**：模版序列的起始
- **pred**：计算范围 [stencil, stencil + (last - first)) 内每个元素的值的谓词。将 pred 计算为 true 的元素从序列 [first, last) 中删除

前提条件

- 范围 [first, last) 不应与范围 [result, result + (last - first)) 重叠

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

- 范围 [stencil, stencil + (last - first)) 不应与范围 [result, result + (last - first)) 重叠

返回值

ForwardIterator ，指向 pred 计算为 true 的元素结果范围的末尾

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/remove
- remove
- remove_copy
- remove_copy_if

thrust::remove_copy_if

```
template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator,
  typename Predicate>
__host__ __device__ OutputIterator
remove_copy_if(const thrust::detail::execution_policy_base< DerivedPolicy >
↪ & exec,
  InputIterator1 first,
  InputIterator1 last,
  InputIterator2 stencil,
  OutputIterator result,
  Predicate pred);
```

remove_copy_if 将范围 [first,last) 内除了使得对应模板值的 pred 为 true 之外的其他元素复制到从 result 开始的范围。返回值是结果范围的末尾。此运算是稳定的，指的是复制的元素的相对顺序与范围 [first,last) 内的顺序相同。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::host 并行化执行策略，使用 remove_copy_if 将数字序列复制到输出范围，同时忽略特定元素：

```
#include <thrust/remove.h>
#include <thrust/execution_policy.h>
...
const int N = 6;
int V[N] = {-2, 0, -1, 0, 1, 2};
int S[N] = { 1, 1,  0, 1, 0, 1};
int result[2];
thrust::remove_copy_if(thrust::host, V, V + N, S, result, thrust::identity
↪<int>());
// V remains {-2, 0, -1, 0, 1, 2}
// result is now {-1, 1}
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **InputIterator1**：是 Input Iterator 的模型，InputIterator1 的 value_type 可转换为 OutputIterator 的 value_types 集合中的类型
- **InputIterator2**：是 Input Iterator 的模型，InputIterator2 的 value_type 可转换为 Predicate 的 argument_type

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

- **OutputIterator**：是 Output Iterator 的模型
- **Predicate**：是 Predicate 的模型

函数参数

- **exec**：用于并行化的执行策略
- **first**：感兴趣范围的起始
- **last**：感兴趣范围的末尾
- **stencil**：模版序列的起始
- **result**：结果范围被复制到从此位置开始的序列中
- **pred**：计算范围 [first,last) 内每个元素的值的谓词。不会将 pred 计算为 false 的元素复制到结果序列中

前提条件

范围 [stencil, stencil + (last - first)) 不应与范围 [result, result + (last - first)) 重叠

返回值

指向结果范围末尾的 OutputIterator

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/remove_copy
- remove
- remove_copy
- remove_if
- copy_if

thrust::remove_copy_if

```
template <typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator,
  typename Predicate>
OutputIterator
remove_copy_if(InputIterator1 first,
  InputIterator1 last,
  InputIterator2 stencil,
  OutputIterator result,
  Predicate pred);
```

remove_copy_if 将范围 [first,last) 内除了使得对应模板值的 pred 为 true 之外的其他元素复制到从 result 开始的范围。返回值是结果范围的末尾。此运算是稳定的，指的是复制的元素的相对顺序与范围 [first,last) 内的顺序相同。

以下代码片段演示了如何使用 remove_copy_if 将数字序列复制到输出范围，同时忽略特定元素：

```
#include <thrust/remove.h>
...
const int N = 6;
int V[N] = {-2, 0, -1, 0, 1, 2};
int S[N] = { 1, 1,  0, 1, 0, 1};
int result[2];
thrust::remove_copy_if(V, V + N, S, result, thrust::identity<int>());
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
// V remains {-2, 0, -1, 0, 1, 2}
// result is now {-1, 1}
```

模板参数

- **InputIterator1**：是 Input Iterator 的模型，InputIterator1 的 value_type 可转换为 OutputIterator 的 value_types 集合中的类型
- **InputIterator2**：是 Input Iterator 的模型，InputIterator2 的 value_type 可转换为 Predicate 的 argument_type
- **OutputIterator**：是 Output Iterator 的模型
- **Predicate**：是 Predicate 的模型

函数参数

- **first**：感兴趣范围的起始
- **last**：感兴趣范围的末尾
- **stencil**：模版序列的起始
- **result**：结果范围被复制到从此位置开始的序列中
- **pred**：计算范围 [first,last) 内每个元素的值的谓词。不会将 pred 计算为 false 的元素复制到结果序列中

前提条件

范围 [stencil, stencil + (last - first)) 不应与范围 [result, result + (last - first)) 重叠

返回值

指向结果范围末尾的 OutputIterator

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/remove_copy
- remove
- remove_copy
- remove_if
- copy_if

thrust::unique

```
template <typename DerivedPolicy,
  typename ForwardIterator>
__host__ __device__ ForwardIterator
unique(const thrust::detail::execution_policy_base< DerivedPolicy > & exec,
  ForwardIterator first,
  ForwardIterator last);
```

对于范围 [first, last) 内每组具有相同值的连续元素，unique 删除每组除第一个元素以外的所有元素。返回值是迭代器 new_last ，使得范围 [first, new_last) 内任意两个连续元素都不相等。范围 [new_last, last) 内的迭代器仍然是可解引用的，但它们所指向的元素是未指定的。unique 是稳定的，指的是未被删除的元素的相对顺序不变。

此版本的 unique 使用 operator== 测试相等性。

算法的执行由 exec 确定并行化。

飞曦信息 MetaX Confidential 2024-11-18 15:00:00

以下代码片段演示了如何通过 thrust::host 并行化执行策略，使用 unique 压缩数字序列以删除连续的重复项：

```
#include <thrust/unique.h>
#include <thrust/execution_policy.h>
...
const int N = 7;
int A[N] = {1, 3, 3, 3, 2, 2, 1};
int *new_end = thrust::unique(thrust::host, A, A + N);
// The first four values of A are now {1, 3, 2, 1}
// Values beyond new_end are unspecified.
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **ForwardIterator**：是 Forward Iterator 的模型，ForwardIterator 是可变的，且 ForwardIterator 的 value_type 是 Equality Comparable 的模型

函数参数

- **exec**：用于并行化的执行策略
- **first**：输入范围的起始
- **last**：输入范围的末尾

返回值

唯一范围 [first, new_last) 的末尾

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/unique
- unique_copy

thrust::unique

```
template <typename ForwardIterator>
ForwardIterator
unique(ForwardIterator first,
  ForwardIterator last);
```

对于范围 [first, last) 内每组具有相同值的连续元素，unique 删除每组除第一个元素以外的所有元素。返回值是迭代器 new_last ，使得范围 [first, new_last) 内任意两个连续元素都不相等。范围 [new_last, last) 内的迭代器仍然是可解引用的，但它们所指向的元素是未指定的。unique 是稳定的，指的是未被删除的元素的相对顺序不变。

此版本的 unique 使用 operator== 测试相等性。

以下代码片段演示了如何使用 unique 压缩数字序列以删除连续的重复项：

```
#include <thrust/unique.h>
...
const int N = 7;
int A[N] = {1, 3, 3, 3, 2, 2, 1};
int *new_end = thrust::unique(A, A + N);
// The first four values of A are now {1, 3, 2, 1}
// Values beyond new_end are unspecified.
```

模板参数

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

ForwardIterator：是 Forward Iterator 的模型，ForwardIterator 是可变的，且 ForwardIterator 的 value_type 是 Equality Comparable 的模型

函数参数

- **first**：输入范围的起始
- **last**：输入范围的末尾

返回值

唯一范围 [first, new_last) 的末尾

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/unique
- unique_copy

thrust::unique

```
template <typename DerivedPolicy,
  typename ForwardIterator,
  typename BinaryPredicate>
__host__ __device__ ForwardIterator
unique(const thrust::detail::execution_policy_base< DerivedPolicy > & exec,
  ForwardIterator first,
  ForwardIterator last,
  BinaryPredicate binary_pred);
```

对于范围 [first, last) 内每组具有相同值的连续元素，unique 删除每组除第一个元素以外的所有元素。返回值是迭代器 new_last，使得范围 [first, new_last) 内任意两个连续元素都不相等。范围 [new_last, last) 内的迭代器仍然是可解引用的，但它们所指向的元素是未指定的。unique 是稳定的，指的是未被删除的元素的相对顺序不变。

此版本的 unique 使用函数对象 binary_pred 测试相等性。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::host 并行化执行策略，使用 unique 压缩数字序列以删除连续的重复项：

```
#include <thrust/unique.h>
#include <thrust/execution_policy.h>
...
const int N = 7;
int A[N] = {1, 3, 3, 3, 2, 2, 1};
int *new_end = thrust::unique(thrust::host, A, A + N, thrust::equal_to<int>
↪());
// The first four values of A are now {1, 3, 2, 1}
// Values beyond new_end are unspecified.
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **ForwardIterator**：是 Forward Iterator 的模型，ForwardIterator‘是可变的，并且 ‘ForwardIterator 的 value_type 可转换为 BinaryPredicate 的 first_argument_type 和 second_argument_type
- **BinaryPredicate**：是 Binary Predicate 的模型

函数参数

- **exec**：用于并行化的执行策略

飞书信息 MetaX Confidential 2024-11-18 15:00.00

- **first**：输入范围的起始
- **last**：输入范围的末尾
- **binary_pred**：用于确定相等性的二元谓词

返回值

唯一范围 [first, new_last) 的末尾

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/unique
- unique_copy

thrust::unique

```
template <typename ForwardIterator,
typename BinaryPredicate>
ForwardIterator
unique(ForwardIterator first,
  ForwardIterator last,
  BinaryPredicate binary_pred);
```

对于范围 [first, last) 内每组具有相同值的连续元素，unique 删除每组除第一个元素以外的所有元素。返回值是迭代器 new_last ，使得范围 [first, new_last) 内任意两个连续元素都不相等。范围 [new_last, last) 内的迭代器仍然是可解引用的，但它们所指向的元素是未指定的。unique 是稳定的，指的是未被删除的元素的相对顺序不变。

此版本的 unique 使用函数对象 binary_pred 测试相等性。

以下代码片段演示了如何使用 unique 压缩数字序列以删除连续的重复项：

```
#include <thrust/unique.h>
...
const int N = 7;
int A[N] = {1, 3, 3, 3, 2, 2, 1};
int *new_end = thrust::unique(A, A + N, thrust::equal_to<int>());
// The first four values of A are now {1, 3, 2, 1}
// Values beyond new_end are unspecified.
```

模板参数

- **ForwardIterator**：是 Forward Iterator 的模型，ForwardIterator ‘是可变的，并且 ‘ForwardIterator 的 value_type 可转换为 BinaryPredicate 的 first_argument_type 和 second_argument_type
- **BinaryPredicate**：是 Binary Predicate 的模型

函数参数

- **first**：输入范围的起始
- **last**：输入范围的末尾
- **binary_pred**：用于确定相等性的二元谓词

返回值

唯一范围 [first, new_last) 的末尾

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/unique
- unique_copy

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

thrust::unique_copy

```
template <typename DerivedPolicy,
  typename InputIterator,
  typename OutputIterator>
__host__ __device__ OutputIterator
unique_copy(const thrust::detail::execution_policy_base< DerivedPolicy > &
↪exec,
  InputIterator first,
  InputIterator last,
  OutputIterator result);
```

unique_copy 将范围 [first, last) 内的元素复制到从 result 开始的范围，只不过仅复制每组连续重复元素的第一个元素。返回值是结果范围的的末尾。

unique_copy 有两个不同版本的原因在于，它对于一组连续重复的元素，有两种不同的定义。在第一个版本中，测试简单的相等性：如果对于范围内的每个迭代器 i，要么 i == f，要么 *i == *(i-1)，则范围 [f, l) 内元素是重复的。在第二个版本中，测试任意 BinaryPredicate binary_pred：如果对于该范围内的每个迭代器 i，要么 i == f，要么 binary_pred(*i, *(i-1)) 为 true，则 [f, l) 内的元素是重复的。

此版本的 unique_copy 使用 operator== 测试相等性。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::host 并行化执行策略，使用 unique_copy 压缩数字序列以删除连续的重复项：

```
#include <thrust/unique.h>
#include <thrust/execution_policy.h>
...
const int N = 7;
int A[N] = {1, 3, 3, 3, 2, 2, 1};
int B[N];
int *result_end = thrust::unique_copy(thrust::host, A, A + N, B);
// The first four values of B are now {1, 3, 2, 1} and (result_end - B) is
↪4
// Values beyond result_end are unspecified
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **InputIterator**：是 Input Iterator 的模型，InputIterator 的 value_type 是 Equality Comparable 的模型
- **OutputIterator**：是 Output Iterator 的模型，InputIterator 的 value_type 可转换为 OutputIterator 的 value_type

函数参数

- **exec**：用于并行化的执行策略
- **first**：输入范围的起始
- **last**：输入范围的末尾
- **result**：输出范围的起始

前提条件

范围 [first,last) 不应与范围 [result, result + (last - first)) 重叠

返回值

唯一范围 [result, result_end) 的末尾

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

另请参阅

- unique
- https://en.cppreference.com/w/cpp/algorithm/unique_copy

thrust::unique_copy

```
template <typename InputIterator,
  typename OutputIterator>
OutputIterator
unique_copy(InputIterator first,
  InputIterator last,
  OutputIterator result);
```

unique_copy 将范围 [first, last) 内的元素复制到从 result 开始的范围，只不过仅复制每组连续重复元素的第一个元素。返回值是结果范围的的末尾。

unique_copy 有两个不同版本的原因在于，它对于一组连续重复的元素，有两种不同的定义。在第一个版本中，测试简单的相等性：如果对于范围内的每个迭代器 i ，要么 i == f ，要么 *i == *(i-1) ，则范围 [f, l) 内元素是重复的。在第二个版本中，测试任意 BinaryPredicate binary_pred：如果对于该范围内的每个迭代器 i ，要么 i == f ，要么 binary_pred(*i, *(i-1)) 为 true ，则 [f, l) 内的元素是重复的。

此版本的 unique_copy 使用 operator== 测试相等性。

以下代码片段演示了如何使用 unique_copy 压缩数字序列以删除连续的重复项：

```
#include <thrust/unique.h>
...
const int N = 7;
int A[N] = {1, 3, 3, 3, 2, 2, 1};
int B[N];
int *result_end = thrust::unique_copy(A, A + N, B);
// The first four values of B are now {1, 3, 2, 1} and (result_end - B) is
↪4
// Values beyond result_end are unspecified
```

模板参数

- **InputIterator**：是 Input Iterator 的模型，InputIterator 的 value_type 是 Equality Comparable 的模型
- **OutputIterator**：是 Output Iterator 的模型，InputIterator 的 value_type 可转换为 OutputIterator 的 value_type

函数参数

- **first**：输入范围的起始
- **last**：输入范围的末尾
- **result**：输出范围的起始

前提条件

范围 [first,last) 不应与范围 [result, result + (last - first)) 重叠

返回值

唯一范围 [result, result_end) 的末尾

另请参阅

- unique
- https://en.cppreference.com/w/cpp/algorithm/unique_copy

飞腾信息 MetaX confidential 2024-11-18 15:00:03

thrust::unique_copy

```
template <typename DerivedPolicy,
  typename InputIterator,
  typename OutputIterator,
  typename BinaryPredicate>
__host__ __device__ OutputIterator
unique_copy(const thrust::detail::execution_policy_base< DerivedPolicy > &
↪exec,
  InputIterator first,
  InputIterator last,
  OutputIterator result,
  BinaryPredicate binary_pred);
```

unique_copy 将范围 [first, last) 内的元素复制到从 result 开始的范围，只不过仅复制每组连续重复元素的第一个元素。返回值是结果范围的的末尾。

此版本的 unique_copy 使用函数对象 binary_pred 测试相等性。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::host 并行化执行策略，使用 unique_copy 压缩数字序列以删除连续的重复项：

```
#include <thrust/unique.h>
#include <thrust/execution_policy.h>
...
const int N = 7;
int A[N] = {1, 3, 3, 3, 2, 2, 1};
int B[N];
int *result_end = thrust::unique_copy(thrust::host, A, A + N, B,
↪thrust::equal_to<int>());
// The first four values of B are now {1, 3, 2, 1} and (result_end - B) is
↪4
// Values beyond result_end are unspecified.
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **InputIterator**：是 Input Iterator 的模型，InputIterator 的 value_type 是 Equality Comparable 的模型
- **OutputIterator**：是 Output Iterator 的模型，InputIterator 的 value_type 可转换为 OutputIterator 的 value_type
- **BinaryPredicate**：是 Binary Predicate 的模型

函数参数

- **exec**：用于并行化的执行策略
- **first**：输入范围的起始
- **last**：输入范围的末尾
- **result**：输出范围的起始
- **binary_pred**：用于确定相等性的二元谓词

前提条件

范围 [first,last) 不应与范围 [result, result + (last - first)) 重叠

返回值

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

唯一范围 [result, result_end) 的末尾

另请参阅

- unique
- https://en.cppreference.com/w/cpp/algorithm/unique_copy

thrust::unique_copy

```
template <typename InputIterator,
  typename OutputIterator,
  typename BinaryPredicate>
OutputIterator
unique_copy(InputIterator first,
  InputIterator last,
  OutputIterator result,
  BinaryPredicate binary_pred);
```

unique_copy 将范围 [first, last) 内的元素复制到从 result 开始的范围，只不过仅复制每组连续重复元素的第一个元素。返回值是结果范围的的末尾。

此版本的 unique_copy 使用函数对象 binary_pred 测试相等性。

以下代码片段演示了如何使用 unique_copy 压缩数字序列以删除连续的重复项：

```
#include <thrust/unique.h>
...
const int N = 7;
int A[N] = {1, 3, 3, 3, 2, 2, 1};
int B[N];
int *result_end = thrust::unique_copy(A, A + N, B, thrust::equal_to<int>
↪());
// The first four values of B are now {1, 3, 2, 1} and (result_end - B) is
↪4
// Values beyond result_end are unspecified.
```

模板参数

- **InputIterator**：是 Input Iterator 的模型，InputIterator 的 value_type 是 Equality Comparable 的模型
- **OutputIterator**：是 Output Iterator 的模型，InputIterator 的 value_type 可转换为 OutputIterator 的 value_type
- **BinaryPredicate**：是 Binary Predicate 的模型

函数参数

- **first**：输入范围的起始
- **last**：输入范围的末尾
- **result**：输出范围的起始
- **binary_pred**：用于确定相等性的二元谓词

前提条件

范围 [first,last) 不应与范围 [result, result + (last - first)) 重叠

返回值

唯一范围 [result, result_end) 的末尾

另请参阅

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

- unique
- https://en.cppreference.com/w/cpp/algorithm/unique_copy

thrust::unique_by_key

```
template <typename DerivedPolicy,
  typename ForwardIterator1,
  typename ForwardIterator2>
__host__ __device__ thrust::pair< ForwardIterator1, ForwardIterator2 >
unique_by_key(const thrust::detail::execution_policy_base< DerivedPolicy >
 ↪& exec,
  ForwardIterator1 keys_first,
  ForwardIterator1 keys_last,
  ForwardIterator2 values_first);
```

unique_by_key 是 unique 到键值对的泛化。对于范围 [keys_first, keys_last) 内每一组相等的连续键，unique_by_key 删除组中除第一个元素以外的所有元素。同样地，范围 [values_first, values_first + (keys_last - keys_first)) 内的相应值也将删除。

返回值是一对迭代器 (new_keys_last,new_values_last)，使得范围 [keys_first, new_keys_last) 内不存在两个相等的连续元素。

此版本的 unique_by_key 使用 operator== 测试相等性，并使用 project1st 对具有相等键的值进行归约。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::host 并行化执行策略，使用 unique_by_key 压缩键值对序列以删除连续的重复项：

```
#include <thrust/unique.h>
#include <thrust/execution_policy.h>
...
const int N = 7;
int A[N] = {1, 3, 3, 3, 2, 2, 1}; // keys
int B[N] = {9, 8, 7, 6, 5, 4, 3}; // values

thrust::pair<int*,int*> new_end;
new_end = thrust::unique_by_key(thrust::host, A, A + N, B);

// The first four keys in A are now {1, 3, 2, 1} and new_end.first - A is
 ↪4.
// The first four values in B are now {9, 8, 5, 3} and new_end.second - B
 ↪is 4.
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **ForwardIterator1**：是 Forward Iterator 的模型，ForwardIterator1 是可变动，且 ForwardIterator 的 value_type 是 Equality Comparable 的模型
- **ForwardIterator2**：是 Forward Iterator 的模型，且 ForwardIterator2 是可变的

函数参数

- **exec**：用于并行化的执行策略
- **keys_first**：键范围的起始
- **keys_last**：键范围的末尾
- **values_first**：值范围的起始

飞谱信息 MetaX confidential 2024-11-18 15:00:00

前提条件

范围 [keys_first, keys_last) 不应与范围 [values_first, values_first + (keys_last - keys_first)) 重叠

返回值

范围 [key_first, keys_new_last) 和 [values_first, values_new_last) 末尾的一对迭代器

另请参阅

- unique
- unique_by_key_copy
- reduce_by_key

thrust::unique_by_key

```
template <typename ForwardIterator1,
typename ForwardIterator2>
thrust::pair< ForwardIterator1, ForwardIterator2 >
unique_by_key(ForwardIterator1 keys_first,
  ForwardIterator1 keys_last,
  ForwardIterator2 values_first);
```

unique_by_key 是 unique 到键值对的泛化。对于范围 [keys_first, keys_last) 内每一组相等的连续键，unique_by_key 删除组中除第一个元素以外的所有元素。 同样地，范围 [values_first, values_first + (keys_last - keys_first)) 内的相应值也将删除。

返回值是一对迭代器 (new_keys_last,new_values_last) ，使得范围 [keys_first, new_keys_last) 内不存在两个相等的连续元素。

此版本的 unique_by_key 使用 operator== 测试相等性，并使用 project1st 对具有相等键的值进行归约。

以下代码片段演示了如何使用 unique_by_key 压缩键值对数字序列以删除连续的重复项：

```
#include <thrust/unique.h>
...
const int N = 7;
int A[N] = {1, 3, 3, 3, 2, 2, 1}; // keys
int B[N] = {9, 8, 7, 6, 5, 4, 3}; // values

thrust::pair<int*,int*> new_end;
new_end = thrust::unique_by_key(A, A + N, B);

// The first four keys in A are now {1, 3, 2, 1} and new_end.first - A is
↪4.
// The first four values in B are now {9, 8, 5, 3} and new_end.second - B
↪is 4.
```

模板参数

- **ForwardIterator1**：是 Forward Iterator 的模型，ForwardIterator1 是可变动，且 ForwardIterator 的 value_type 是 Equality Comparable 的模型
- **ForwardIterator**：是 Forward Iterator 的模型，且 ForwardIterator2 是可变的

函数参数

- **keys_first**：键范围的起始
- **keys_last**：键范围的末尾
- **values_first**：值范围的起始

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

前提条件

范围 [keys_first, keys_last) 不应与范围 [values_first, values_first + (keys_last - keys_first)) 重叠

返回值

范围 [key_first, keys_new_last) 和 [values_first, values_new_last) 末尾的一对迭代器

另请参阅

- unique
- unique_by_key_copy
- reduce_by_key

thrust::unique_by_key

```
template <typename DerivedPolicy,
  typename ForwardIterator1,
  typename ForwardIterator2,
  typename BinaryPredicate>
__host__ __device__ thrust::pair< ForwardIterator1, ForwardIterator2 >
unique_by_key(const thrust::detail::execution_policy_base< DerivedPolicy >
↪& exec,
  ForwardIterator1 keys_first,
  ForwardIterator1 keys_last,
  ForwardIterator2 values_first,
  BinaryPredicate binary_pred);
```

unique_by_key 是 unique 到键值对的泛化。对于范围 [keys_first, keys_last) 内每一组相等的连续键，unique_by_key 删除组中除第一个元素以外的所有元素。 同样地，范围 [values_first, values_first + (keys_last - keys_first)) 内的相应值也将删除。

此版本的 unique_by_key 使用函数对象 binary_pred 测试相等性，并使用 project1st 对具有相等键的值进行归约。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::host 并行化执行策略，使用 unique_by_key 压缩键值对序列以删除连续的重复项：

```
#include <thrust/unique.h>
#include <thrust/execution_policy.h>
...
const int N = 7;
int A[N] = {1, 3, 3, 3, 2, 2, 1}; // keys
int B[N] = {9, 8, 7, 6, 5, 4, 3}; // values

thrust::pair<int*,int*> new_end;
thrust::equal_to<int> binary_pred;
new_end = thrust::unique_by_key(thrust::host, keys, keys + N, values,
↪binary_pred);

// The first four keys in A are now {1, 3, 2, 1} and new_end.first - A is
↪4.
// The first four values in B are now {9, 8, 5, 3} and new_end.second - B
↪is 4.
```

模板参数

- **DerivedPolicy**：派生执行策略的名称

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

- **ForwardIterator1**：是 Forward Iterator 的模型，ForwardIterator1 是可变动，且 ForwardIterator 的 value_type 是 Equality Comparable 的模型
- **ForwardIterator**：是 Forward Iterator 的模型，且 ForwardIterator2 是可变的
- **BinaryPredicate**：是 Binary Predicate 的模型

函数参数

- **exec**：用于并行化的执行策略
- **keys_first**：键范围的起始
- **keys_last**：键范围的末尾
- **values_first**：值范围的起始
- **binary_pred**：用于确定相等性的二元谓词

前提条件

范围 [keys_first, keys_last) 不应与范围 [values_first, values_first + (keys_last - keys_first)) 重叠

返回值

唯一范围 [first, new_last) 的末尾

另请参阅

- unique
- unique_by_key_copy
- reduce_by_key

thrust::unique_by_key

```
template <typename ForwardIterator1,
  typename ForwardIterator2,
  typename BinaryPredicate>
thrust::pair< ForwardIterator1, ForwardIterator2 >
unique_by_key(ForwardIterator1 keys_first,
  ForwardIterator1 keys_last,
  ForwardIterator2 values_first,
  BinaryPredicate binary_pred);
```

unique_by_key 是 unique 到键值对的泛化。对于范围 [keys_first, keys_last) 内每一组相等的连续键，unique_by_key 删除组中除第一个元素以外的所有元素。 同样地，范围 [values_first, values_first + (keys_last - keys_first)) 内的相应值也将删除。

此版本的 unique_by_key 使用函数对象 binary_pred 测试相等性，并使用 project1st 对具有相等键的值进行归约。

以下代码片段演示了如何使用 unique_by_key 压缩键值对数字序列以删除连续的重复项：

```
#include <thrust/unique.h>
...
const int N = 7;
int A[N] = {1, 3, 3, 3, 2, 2, 1}; // keys
int B[N] = {9, 8, 7, 6, 5, 4, 3}; // values

thrust::pair<int*,int*> new_end;
thrust::equal_to<int> binary_pred;
new_end = thrust::unique_by_key(keys, keys + N, values, binary_pred);
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
// The first four keys in A are now {1, 3, 2, 1} and new_end.first - A is
↪4.
// The first four values in B are now {9, 8, 5, 3} and new_end.second - B
↪is 4.
```

模板参数

- **ForwardIterator1**：是 Forward Iterator 的模型，ForwardIterator1 是可变动，且 ForwardIterator 的 value_type 是 Equality Comparable 的模型
- **ForwardIterator**：是 Forward Iterator 的模型，且 ForwardIterator2 是可变的
- **BinaryPredicate**：是 Binary Predicate 的模型

函数参数

- **keys_first**：键范围的起始
- **keys_last**：键范围的末尾
- **values_first**：值范围的起始
- **binary_pred**：用于确定相等性的二元谓词

前提条件

范围 [keys_first, keys_last) 不应与范围 [values_first, values_first + (keys_last - keys_first)) 重叠

返回值

唯一范围 [first, new_last) 的末尾

另请参阅

- unique
- unique_by_key_copy
- reduce_by_key

thrust::unique_by_key_copy

```
template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator1,
  typename OutputIterator2>
__host__ __device__ thrust::pair< OutputIterator1, OutputIterator2 >
unique_by_key_copy(const thrust::detail::execution_policy_base<
↪DerivedPolicy > & exec,
  InputIterator1 keys_first,
  InputIterator1 keys_last,
  InputIterator2 values_first,
  OutputIterator1 keys_result,
  OutputIterator2 values_result);
```

unique_by_key_copy 是 unique_copy 到键值对的泛化。对于范围 [keys_first, keys_last) 内每一组相等的连续键，unique_by_key_copy 将组内第一个元素复制到从 keys_result 开始的范围，并且将范围 [values_first, values_first + (keys_last - keys_first)) 内的相应值复制到从 values_result 开始的范围。

此版本的 unique_by_key_copy 使用 operator== 测试相等性，并使用 project1st 对具有相等键的值进行归约。

算法的执行由 exec 确定并行化。

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

以下代码片段演示了如何通过 thrust::host 并行化执行策略，使用 unique_by_key_copy 压缩键值对序列以删除建相同的重复项：

```
#include <thrust/unique.h>
#include <thrust/execution_policy.h>
...
const int N = 7;
int A[N] = {1, 3, 3, 3, 2, 2, 1}; // input keys
int B[N] = {9, 8, 7, 6, 5, 4, 3}; // input values
int C[N];    // output keys
int D[N];    // output values

thrust::pair<int*,int*> new_end;
new_end = thrust::unique_by_key_copy(thrust::host, A, A + N, B, C, D);

// The first four keys in C are now {1, 3, 2, 1} and new_end.first - C is
↪4.
// The first four values in D are now {9, 8, 5, 3} and new_end.second - D
↪is 4.
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **InputIterator1**：是 Input Iterator 的模型
- **InputIterator2**：是 Input Iterator 的模型
- **OutputIterator1**：是 Output Iterator 的模型，InputIterator1 的 value_type 可转换为 OutputIterator1 的 value_type
- **OutputIterator2**：是 Output Iterator 的模型，InputIterator2 的 value_type 可转换为 OutputIterator2 的 value_type

函数参数

- **exec**：用于并行化的执行策略
- **keys_first**：输入键范围的起始
- **keys_last**：输入键范围的末尾
- **values_first**：输入值范围的起始
- **keys_result**：输出键范围的起始
- **values_result**：输出值范围的起始

前提条件

输入范围不应与任何一个输出范围重叠

返回值

范围 [keys_result, keys_result_last) 和 [values_result, values_result_last) 末尾的一对迭代器

另请参阅

- unique_copy
- unique_by_key
- reduce_by_key

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

thrust::unique_by_key_copy

```
template <typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator1,
  typename OutputIterator2>
thrust::pair< OutputIterator1, OutputIterator2 >
unique_by_key_copy(InputIterator1 keys_first,
  InputIterator1 keys_last,
  InputIterator2 values_first,
  OutputIterator1 keys_result,
  OutputIterator2 values_result);
```

unique_by_key_copy 是 unique_copy 到键值对的泛化。对于范围 [keys_first, keys_last) 内每一组相等的连续键，unique_by_key_copy 将组内第一个元素复制到从 keys_result 开始的范围，并且将范围 [values_first, values_first + (keys_last - keys_first)) 内的相应值复制到从 values_result 开始的范围。

此版本的 unique_by_key_copy 使用 operator== 测试相等性，并使用 project1st 对具有相等键的值进行归约。

以下代码片段演示了如何使用 unique_by_key_copy 压缩键值对序列和相等的键：

```
#include <thrust/unique.h>
...
const int N = 7;
int A[N] = {1, 3, 3, 3, 2, 2, 1}; // input keys
int B[N] = {9, 8, 7, 6, 5, 4, 3}; // input values
int C[N];    // output keys
int D[N];    // output values

thrust::pair<int*,int*> new_end;
new_end = thrust::unique_by_key_copy(A, A + N, B, C, D);

// The first four keys in C are now {1, 3, 2, 1} and new_end.first - C is
↪4.
// The first four values in D are now {9, 8, 5, 3} and new_end.second - D
↪is 4.
```

模板参数

- **InputIterator1**：是 Input Iterator 的模型
- **InputIterator2**：是 Input Iterator 的模型
- **OutputIterator1**：是 Output Iterator 的模型，InputIterator1 的 value_type 可转换为 OutputIterator1 的 value_type
- **OutputIterator2**：是 Output Iterator 的模型，InputIterator2 的 value_type 可转换为 OutputIterator2 的 value_type

函数参数

- **keys_first**：输入键范围的起始
- **keys_last**：输入键范围的末尾
- **values_first**：输入值范围的起始
- **keys_result**：输出键范围的起始
- **values_result**：输出值范围的起始

前提条件

输入范围不应与任何一个输出范围重叠

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

返回值

范围 [keys_result, keys_result_last) 和 [values_result, values_result_last) 末尾的一对迭代器

另请参阅

- unique_copy
- unique_by_key
- reduce_by_key

thrust::unique_by_key_copy

```
template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator1,
  typename OutputIterator2,
  typename BinaryPredicate>
__host__ __device__ thrust::pair< OutputIterator1, OutputIterator2 >
unique_by_key_copy(const thrust::detail::execution_policy_base<
 ↪DerivedPolicy > & exec,
  InputIterator1 keys_first,
  InputIterator1 keys_last,
  InputIterator2 values_first,
  OutputIterator1 keys_result,
  OutputIterator2 values_result,
  BinaryPredicate binary_pred);
```

unique_by_key_copy 是 unique_copy 到键值对的泛化。对于范围 [keys_first, keys_last) 内每一组相等的连续键，unique_by_key_copy 将组内第一个元素复制到从 keys_result 开始的范围，并且将范围 [values_first, values_first + (keys_last - keys_first)) 内的相应值复制到从 values_result 开始的范围。

此版本的 unique_by_key_copy 使用函数对象 binary_pred 测试相等性，并使用 project1st 对具有相等键的值进行归约。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::host 并行化执行策略，使用 unique_by_key_copy 压缩键值对序列和相等的键：

```
#include <thrust/unique.h>
#include <thrust/execution_policy.h>
...
const int N = 7;
int A[N] = {1, 3, 3, 3, 2, 2, 1}; // input keys
int B[N] = {9, 8, 7, 6, 5, 4, 3}; // input values
int C[N];     // output keys
int D[N];     // output values

thrust::pair<int*,int*> new_end;
thrust::equal_to<int> binary_pred;
new_end = thrust::unique_by_key_copy(thrust::host, A, A + N, B, C, D,
 ↪binary_pred);

// The first four keys in C are now {1, 3, 2, 1} and new_end.first - C is
 ↪4.
// The first four values in D are now {9, 8, 5, 3} and new_end.second - D
 ↪is 4.
```

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **InputIterator1**：是 Input Iterator 的模型
- **InputIterator2**：是 Input Iterator 的模型
- **OutputIterator1**：是 Output Iterator 的模型，InputIterator1 的 value_type 可转换为 OutputIterator1 的 value_type
- **OutputIterator2**：是 Output Iterator 的模型，InputIterator2 的 value_type 可转换为 OutputIterator2 的 value_type
- **BinaryPredicate**：是 Binary Predicate 的模型

函数参数

- **exec**：用于并行化的执行策略
- **keys_first**：输入键范围的起始
- **keys_last**：输入键范围的末尾
- **values_first**：输入值范围的起始
- **keys_result**：输出键范围的起始
- **values_result**：输出值范围的起始
- **binary_pred**：用于确定相等性的二元谓词

前提条件

输入范围不应与任何一个输出范围重叠

返回值

范围 [keys_result, keys_result_last) 和 [values_result, values_result_last) 末尾的一对迭代器

另请参阅

- unique_copy
- unique_by_key
- reduce_by_key

thrust::unique_by_key_copy

```
template <typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator1,
  typename OutputIterator2,
  typename BinaryPredicate>
thrust::pair< OutputIterator1, OutputIterator2 >
unique_by_key_copy(InputIterator1 keys_first,
  InputIterator1 keys_last,
  InputIterator2 values_first,
  OutputIterator1 keys_result,
  OutputIterator2 values_result,
  BinaryPredicate binary_pred);
```

unique_by_key_copy 是 unique_copy 到键值对的泛化。对于相等范围 [keys_first, keys_last) 内的每一组连续键，unique_by_key_copy 将组内第一个元素复制到从 keys_result 开始的范围，并且将范围 [values_first, values_first + (keys_last - keys_first)) 内的相应值复制到从 values_result 开始的范围。

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

此版本的 unique_by_key_copy 使用函数对象 binary_pred 测试相等性，并使用 project1st 对具有相等键的值进行归约。

以下代码片段演示了如何使用 unique_by_key_copy 压缩键值对序列和相等的键：

```
#include <thrust/unique.h>
...
const int N = 7;
int A[N] = {1, 3, 3, 3, 2, 2, 1}; // input keys
int B[N] = {9, 8, 7, 6, 5, 4, 3}; // input values
int C[N];    // output keys
int D[N];    // output values

thrust::pair<int*,int*> new_end;
thrust::equal_to<int> binary_pred;
new_end = thrust::unique_by_key_copy(A, A + N, B, C, D, binary_pred);

// The first four keys in C are now {1, 3, 2, 1} and new_end.first - C is
↪4.
// The first four values in D are now {9, 8, 5, 3} and new_end.second - D
↪is 4.
```

模板参数

- **InputIterator1**：是 Input Iterator 的模型
- **InputIterator2**：是 Input Iterator 的模型
- **OutputIterator1**：是 Output Iterator 的模型，InputIterator1 的 value_type 可转换为 OutputIterator1 的 value_type
- **OutputIterator2**：是 Output Iterator 的模型，InputIterator2 的 value_type 可转换为 OutputIterator2 的 value_type
- **BinaryPredicate**：是 Binary Predicate 的模型

函数参数

- **keys_first**：输入键范围的起始
- **keys_last**：输入键范围的末尾
- **values_first**：输入值范围的起始
- **keys_result**：输出键范围的起始
- **values_result**：输出值范围的起始
- **binary_pred**：用于确定相等性的二元谓词

前提条件

输入范围不应与任何一个输出范围重叠

返回值

范围 [keys_result, keys_result_last) 和 [values_result, values_result_last) 末尾的一对迭代器

另请参阅

- unique_copy
- unique_by_key
- reduce_by_key

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

thrust::unique_count

```
template <typename DerivedPolicy,
  typename ForwardIterator,
  typename BinaryPredicate>
__host__ __device__ thrust::iterator_traits< ForwardIterator >::difference_
↪type
unique_count(const thrust::detail::execution_policy_base< DerivedPolicy > &
↪ exec,
  ForwardIterator first,
  ForwardIterator last,
  BinaryPredicate binary_pred);
```

unique_count 计算范围 [first, last) 内具有相同值的相等元素的运行次数。此版本的 unique_count 使用函数对象 binary_pred 测试相等性。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::host 并行化执行策略，使用 unique_count 确定相等元素的运行次数：

```
#include <thrust/unique.h>
#include <thrust/execution_policy.h>
...
const int N = 7;
int A[N] = {1, 3, 3, 3, 2, 2, 1};
int count = thrust::unique_count(thrust::host, A, A + N, thrust::equal_to
↪<int>());
// count is now 4
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **ForwardIterator**：是 Forward Iterator 的模型，ForwardIterator 的 value_type 可转换为 BinaryPredicate 的 first_argument_type 和 second_argument_type
- **BinaryPredicate**：是 Binary Predicate 的模型

函数参数

- **exec**：用于并行化的执行策略
- **first**：输入范围的起始
- **last**：输入范围的末尾
- **binary_pred**：用于确定相等性的二元谓词

返回值

[first, new_last) 内相等元素的运行次数

另请参阅

- unique_copy
- unique_by_key_copy
- reduce_by_key_copy

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

thrust::unique_count

```
template <typename DerivedPolicy,
  typename ForwardIterator>
__host__ __device__ thrust::iterator_traits< ForwardIterator >::difference_
↪type
unique_count(const thrust::detail::execution_policy_base< DerivedPolicy > &
↪ exec,
  ForwardIterator first,
  ForwardIterator last);
```

unique_count 计算范围 [first, last) 内具有相同值的相等元素的运行次数。

此版本的 unique_count 使用 operator== 测试相等性。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::host 并行化执行策略，使用 unique_count 确定相等元素的运行次数：

```
#include <thrust/unique.h>
#include <thrust/execution_policy.h>
...
const int N = 7;
int A[N] = {1, 3, 3, 3, 2, 2, 1};
int count = thrust::unique_count(thrust::host, A, A + N);
// count is now 4
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **ForwardIterator**：是 Forward Iterator 的模型，ForwardIterator 的 value_type 可转换为 BinaryPredicate 的 first_argument_type 和 second_argument_type
- **BinaryPredicate**：是 Binary Predicate 的模型

函数参数

- **exec**：用于并行化的执行策略
- **first**：输入范围的起始
- **last**：输入范围的末尾
- **binary_pred**：用于确定相等性的二元谓词

返回值

[first, new_last) 内相等元素的运行次数

另请参阅

- unique_copy
- unique_by_key_copy
- reduce_by_key_copy

thrust::unique_count

```
template <typename ForwardIterator,
  typename BinaryPredicate>
__host__ __device__ thrust::iterator_traits< ForwardIterator >::difference_
↪type
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
unique_count(ForwardIterator first,
  ForwardIterator last,
  BinaryPredicate binary_pred);
```

unique_count 计算范围 [first, last) 内相等元素的运行次数。

此版本的 unique_count 使用函数对象 binary_pred 测试相等性。

以下代码片段演示了如何使用 unique_count 确定相等元素的运行次数：

```
#include <thrust/unique.h>
#include <thrust/execution_policy.h>
...
const int N = 7;
int A[N] = {1, 3, 3, 3, 2, 2, 1};
int count = thrust::unique_count(A, A + N, thrust::equal_to<int>());
// count is now 4
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **ForwardIterator**：是 Forward Iterator 的模型，ForwardIterator 的 value_type 可转换为 BinaryPredicate 的 first_argument_type 和 second_argument_type
- **BinaryPredicate**：是 Binary Predicate 的模型

函数参数

- **exec**：用于并行化的执行策略
- **first**：输入范围的起始
- **last**：输入范围的末尾
- **binary_pred**：用于确定相等性的二元谓词

返回值

[first, new_last) 内相等元素的运行次数

另请参阅

- unique_copy
- unique_by_key_copy
- reduce_by_key_copy

thrust::unique_count

```
template <typename ForwardIterator>
__host__ __device__ thrust::iterator_traits< ForwardIterator >::difference_
↪type
unique_count(ForwardIterator first,
  ForwardIterator last);
```

unique_count 计算范围 [first, last) 内具有相同值的相等元素的运行次数。

此版本的 unique_count 使用 operator== 测试相等性。

以下代码片段演示了如何使用 unique_count 确定相等元素的运行次数：

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

```
#include <thrust/unique.h>
#include <thrust/execution_policy.h>
...
const int N = 7;
int A[N] = {1, 3, 3, 3, 2, 2, 1};
int count = thrust::unique_count(thrust::host, A, A + N);
// count is now 4
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **ForwardIterator**：是 Forward Iterator 的模型，ForwardIterator 的 value_type 可转换为 BinaryPredicate 的 first_argument_type 和 second_argument_type
- **BinaryPredicate**：是 Binary Predicate 的模型

函数参数

- **exec**：用于并行化的执行策略
- **first**：输入范围的起始
- **last**：输入范围的末尾
- **binary_pred**：用于确定相等性的二元谓词

返回值

[first, new_last) 内相等元素的运行次数

另请参阅

- unique_copy
- unique_by_key_copy
- reduce_by_key_copy

2.1.6 搜索（Searching）

2.1.6.1 搜索

函数

```
template <typename DerivedPolicy,
  typename InputIterator,
  typename T>
__host__ __device__ InputIterator
thrust::find(const thrust::detail::execution_policy_base< DerivedPolicy > &
↪ exec,
  InputIterator first,
  InputIterator last,
  const T & value);

template <typename InputIterator,
  typename T>
InputIterator
thrust::find(InputIterator first,
  InputIterator last,
  const T & value);
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
template <typename DerivedPolicy,
  typename InputIterator,
  typename Predicate>
__host__ __device__ InputIterator
thrust::find_if(const thrust::detail::execution_policy_base< DerivedPolicy
↪> & exec,
  InputIterator first,
  InputIterator last,
  Predicate pred);

template <typename InputIterator,
  typename Predicate>
InputIterator t
hrust::find_if(InputIterator first,
  InputIterator last,
  Predicate pred);

template <typename DerivedPolicy,
  typename InputIterator,
  typename Predicate>
__host__ __device__ InputIterator
thrust::find_if_not(const thrust::detail::execution_policy_base<
↪DerivedPolicy > & exec,
  InputIterator first,
  InputIterator last,
  Predicate pred);

template <typename InputIterator,
  typename Predicate>
InputIterator
thrust::find_if_not(InputIterator first,
  InputIterator last,
  Predicate pred);

template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2>
__host__ __device__ thrust::pair< InputIterator1, InputIterator2 >
thrust::mismatch(const thrust::detail::execution_policy_base<
↪DerivedPolicy > & exec,
  InputIterator1 first1,
  InputIterator1 last1,
  InputIterator2 first2);

template <typename InputIterator1,
  typename InputIterator2>
thrust::pair< InputIterator1, InputIterator2 >
thrust::mismatch(InputIterator1 first1,
  InputIterator1 last1,
  InputIterator2 first2);
template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2,
  typename BinaryPredicate>
__host__ __device__ thrust::pair< InputIterator1, InputIterator2 >
thrust::mismatch(const thrust::detail::execution_policy_base<
↪DerivedPolicy > & exec,
```

(下页继续)

飞诺信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
  InputIterator1 first1,
  InputIterator1 last1,
  InputIterator2 first2,
  BinaryPredicate pred);

 template <typename InputIterator1,
   typename InputIterator2,
   typename BinaryPredicate>
 thrust::pair< InputIterator1, InputIterator2 >
 thrust::mismatch(InputIterator1 first1,
   InputIterator1 last1,
   InputIterator2 first2,
   BinaryPredicate pred);

 template <typename DerivedPolicy,
   typename ForwardIterator,
   typename Predicate>
 __host__ __device__ ForwardIterator
 thrust::partition_point(const thrust::detail::execution_policy_base<
↪DerivedPolicy > & exec,
   ForwardIterator first,
   ForwardIterator last,
   Predicate pred);

 template <typename ForwardIterator,
   typename Predicate>
 ForwardIterator
 thrust::partition_point(ForwardIterator first,
   ForwardIterator last,
   Predicate pred);
```

thrust::find

```
template <typename DerivedPolicy,
  typename InputIterator,
  typename T>
__host__ __device__ InputIterator
find(const thrust::detail::execution_policy_base< DerivedPolicy > & exec,
  InputIterator first,
  InputIterator last,
  const T & value);
```

find 返回范围 [first, last) 内使得 *i == value 的第一个迭代器 i；没有这种迭代器时返回 last。

算法的执行由 exec 确定并行化。

```
#include <thrust/find.h>
#include <thrust/device_vector.h>
#include <thrust/execution_policy.h>
...
thrust::device_vector<int> input(4);

input[0] = 0;
input[1] = 5;
input[2] = 3;
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
input[3] = 7;

thrust::device_vector<int>::iterator iter;

iter = thrust::find(thrust::device, input.begin(), input.end(), 3); //
↪returns input.first() + 2
iter = thrust::find(thrust::device, input.begin(), input.end(), 5); //
↪returns input.first() + 1
iter = thrust::find(thrust::device, input.begin(), input.end(), 9); //
↪returns input.end()
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **InputIterator**：是 Input Iterator 的模型，InputIterator 的 value_type 等价于类型 T
- **T**：是 EqualityComparable 的模型

函数参数

- **exec**：用于并行化的执行策略
- **first**：待搜索的序列的起始
- **last**：待搜索的序列的末尾
- **value**：要查找的值

返回值

返回使得 *i == value 的第一个迭代器 i；否则，返回 last

另请参阅

- find_if
- mismatch

thrust::find

```
template <typename InputIterator,
  typename T>
InputIterator
find(InputIterator first,
  InputIterator last,
  const T & value);
```

find 返回范围 [first, last) 内使得 *i == value 的第一个迭代器 i；没有这种迭代器时返回 last。

```
#include <thrust/find.h>
#include <thrust/device_vector.h>
...
thrust::device_vector<int> input(4);

input[0] = 0;
input[1] = 5;
input[2] = 3;
input[3] = 7;

thrust::device_vector<int>::iterator iter;
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
iter = thrust::find(input.begin(), input.end(), 3); // returns input.
↪first() + 2
iter = thrust::find(input.begin(), input.end(), 5); // returns input.
↪first() + 1
iter = thrust::find(input.begin(), input.end(), 9); // returns input.end()
```

模板参数

- **InputIterator**：是 Input Iterator 的模型，InputIterator 的 value_type 等价于类型 T
- **T**：是 EqualityComparable 的模型

函数参数

- **first**：待搜索的序列的起始
- **last**：待搜索的序列的末尾
- **value**：要查找的值

返回值

返回使得 *i == value 的第一个迭代器 i；否则，返回 last

另请参阅

- find_if
- mismatch

thrust::find_if

```
template <typename DerivedPolicy,
  typename InputIterator,
  typename Predicate>
__host__ __device__ InputIterator
find_if(const thrust::detail::execution_policy_base< DerivedPolicy > &
↪exec,
  InputIterator first,
  InputIterator last,
  Predicate pred);
```

find_if 返回范围 [first, last) 内使得 pred(*i) 为 true 的第一个迭代器 i；没有这种迭代器时返回 last。

算法的执行由 exec 确定并行化。

```
#include <thrust/find.h>
#include <thrust/device_vector.h>
#include <thrust/execution_policy.h>
...

struct greater_than_four
{
__host__ __device__
bool operator()(int x)
{
return x > 4;
}
};
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
struct greater_than_ten
{
__host__ __device__
bool operator()(int x)
{
return x > 10;
}
};

...
thrust::device_vector<int> input(4);

input[0] = 0;
input[1] = 5;
input[2] = 3;
input[3] = 7;

thrust::device_vector<int>::iterator iter;

iter = thrust::find_if(thrust::device, input.begin(), input.end(), greater_
↪than_four()); // returns input.first() + 1

iter = thrust::find_if(thrust::device, input.begin(), input.end(), greater_
↪than_ten());  // returns input.end()
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **InputIterator**：是 Input Iterator 的模型
- **Predicate**：是 Predicate 的模型

函数参数

- **exec**：用于并行化的执行策略
- **first**：待搜索的序列的起始
- **last**：待搜索的序列的末尾
- **pred**：用于测试范围内元素的谓词

返回值

返回使得 pred(*i) 为 true 的第一个迭代器 i；否则，返回 last

另请参阅

- find
- find_if_not
- mismatch

thrust::find_if

```
template <typename InputIterator,
  typename Predicate>
InputIterator
find_if(InputIterator first,
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
  InputIterator last,
  Predicate pred);
```

find_if 返回范围 [first, last) 内使得 pred(*i) 为 true 的第一个迭代器 i；没有这种迭代器时返回 last。

```
#include <thrust/find.h>
#include <thrust/device_vector.h>

struct greater_than_four
{
__host__ __device__
bool operator()(int x)
{
return x > 4;
}
};

struct greater_than_ten
{
__host__ __device__
bool operator()(int x)
{
return x > 10;
}
};

...
thrust::device_vector<int> input(4);

input[0] = 0;
input[1] = 5;
input[2] = 3;
input[3] = 7;

thrust::device_vector<int>::iterator iter;

iter = thrust::find_if(input.begin(), input.end(), greater_than_four()); //
↪ returns input.first() + 1

iter = thrust::find_if(input.begin(), input.end(), greater_than_ten());  //
↪ returns input.end()
```

模板参数

- **InputIterator**：是 Input Iterator 的模型
- **Predicate**：是 Predicate 的模型

函数参数

- **first**：待搜索的序列的起始
- **last**：待搜索的序列的末尾
- **pred**：用于测试范围内元素的谓词

返回值

返回使得 pred(*i) 为 true 的第一个迭代器 i；否则，返回 last

另请参阅

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

- find
- find_if_not
- mismatch

thrust::find_if_not

```
template <typename DerivedPolicy,
  typename InputIterator,
  typename Predicate>
__host__ __device__ InputIterator
find_if_not(const thrust::detail::execution_policy_base< DerivedPolicy > &
↪exec,
  InputIterator first,
  InputIterator last,
  Predicate pred);
```

find_if_not 返回范围 [first, last) 内使得 pred(*i) 为 false 的第一个迭代器 i；没有这种迭代器时返回 last。算法的执行由 exec 确定并行化。

```
#include <thrust/find.h>
#include <thrust/device_vector.h>
#include <thrust/execution_policy.h>
...

struct greater_than_four
{
__host__ __device__
bool operator()(int x)
{
return x > 4;
}
};

struct greater_than_ten
{
__host__ __device__
bool operator()(int x)
{
return x > 10;
}
};

...
thrust::device_vector<int> input(4);

input[0] = 0;
input[1] = 5;
input[2] = 3;
input[3] = 7;

thrust::device_vector<int>::iterator iter;

iter = thrust::find_if_not(thrust::device, input.begin(), input.end(),
↪greater_than_four()); // returns input.first()
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
iter = thrust::find_if_not(thrust::device, input.begin(), input.end(),
↪greater_than_ten());  // returns input.first()
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **InputIterator**：是 Input Iterator 的模型
- **Predicate**：是 Predicate 的模型

函数参数

- **exec**：用于并行化的执行策略
- **first**：待搜索的序列的起始
- **last**：待搜索的序列的末尾
- **pred**：用于测试范围内元素的谓词

返回值

返回使得 pred(*i) 为 false 的第一个迭代器 i；否则，返回 last

另请参阅

- find
- find_if
- mismatch

thrust::find_if_not

```
template <typename InputIterator,
  typename Predicate>
InputIterator
find_if_not(InputIterator first,
  InputIterator last,
  Predicate pred);
```

find_if_not 返回范围 [first, last) 内使得 pred(*i) 为 false 的第一个迭代器 i；没有这种迭代器时返回 last。

```
#include <thrust/find.h>
#include <thrust/device_vector.h>

struct greater_than_four
{
__host__ __device__
bool operator()(int x)
{
return x > 4;
}
};

struct greater_than_ten
{
__host__ __device__
bool operator()(int x)
{
return x > 10;
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
}
};

...
thrust::device_vector<int> input(4);

input[0] = 0;
input[1] = 5;
input[2] = 3;
input[3] = 7;

thrust::device_vector<int>::iterator iter;

iter = thrust::find_if_not(input.begin(), input.end(), greater_than_
↪four()); // returns input.first()

iter = thrust::find_if_not(input.begin(), input.end(), greater_than_ten());
↪  // returns input.first()
```

模板参数

- **InputIterator**：是 Input Iterator 的模型
- **Predicate**：是 Predicate 的模型

函数参数

- **first**：待搜索的序列的起始
- **last**：待搜索的序列的末尾
- **pred**：用于测试范围内元素的谓词

返回值

返回使得 pred(*i) 为 false 的第一个迭代器 i；否则，返回 last

另请参阅

- find
- find_if
- mismatch

thrust::mismatch

```
template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2>
__host__ __device__ thrust::pair< InputIterator1, InputIterator2 >
mismatch(const thrust::detail::execution_policy_base< DerivedPolicy > &
↪exec,
  InputIterator1 first1,
  InputIterator1 last1,
  InputIterator2 first2);
```

mismatch 查找两个范围 [first1, last1) 和 [first2, first2 + (last1 - first1)) 不同的第一个位置。mismatch 的两个版本使用不同的测试来判断元素是否不同。

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

此版本的 mismatch 查找 [first1, last1) 内使得 *i == *(first2 + (i - first1)) 为 false 的第一个迭代器 i。返回值为一个 pair：第一个元素是 i，第二个元素是 *(first2 + (i - first1))。如果不存在这样的迭代器 i，则返回的 pair 为：第一个元素是 last1，第二个元素是 *(first2 + (last1 - first1))。

算法的执行由 exec 确定并行化。

```
#include <thrust/mismatch.h>
#include <thrust/device_vector.h>
#include <thrust/execution_policy.h>
...
thrust::device_vector<int> vec1(4);
thrust::device_vector<int> vec2(4);

vec1[0] = 0;  vec2[0] = 0;
vec1[1] = 5;  vec2[1] = 5;
vec1[2] = 3;  vec2[2] = 8;
vec1[3] = 7;  vec2[3] = 7;

typedef thrust::device_vector<int>::iterator Iterator;
thrust::pair<Iterator,Iterator> result;

result = thrust::mismatch(thrust::device, vec1.begin(), vec1.end(), vec2.
↪begin());

// result.first  is vec1.begin() + 2
// result.second is vec2.begin() + 2
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **InputIterator1**：是 Input Iterator 的模型，InputIterator1 的 value_type 等价于 InputIterator2 的 value_type
- **InputIterator2**：是 Input Iterator 的模型

函数参数

- **exec**：用于并行化的执行策略
- **first1**：第一个序列的起始
- **last1**：第一个序列的末尾
- **first2**：第二个序列的起始

返回值

序列出现不同的第一个位置

另请参阅

- find
- find_if

thrust::mismatch

```
template <typename InputIterator1,
  typename InputIterator2>
thrust::pair< InputIterator1, InputIterator2 >
mismatch(InputIterator1 first1,
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
  InputIterator1 last1,
  InputIterator2 first2);
```

mismatch 查找两个范围 [first1, last1) 和 [first2, first2 + (last1 - first1)) 不同的第一个位置。mismatch 的两个版本使用不同的测试来判断元素是否不同。

此版本的 mismatch 查找 [first1, last1) 内使得 *i == *(first2 + (i - first1)) 为 false 的第一个迭代器 i。返回值为一个 pair：第一个元素是 i，第二个元素是 *(first2 + (i - first1))。如果不存在这样的迭代器 i，则返回的 pair 为：第一个元素是 last1，第二个元素是 *(first2 + (last1 - first1))。

```
#include <thrust/mismatch.h>
#include <thrust/device_vector.h>
...
thrust::device_vector<int> vec1(4);
thrust::device_vector<int> vec2(4);

vec1[0] = 0;  vec2[0] = 0;
vec1[1] = 5;  vec2[1] = 5;
vec1[2] = 3;  vec2[2] = 8;
vec1[3] = 7;  vec2[3] = 7;

typedef thrust::device_vector<int>::iterator Iterator;
thrust::pair<Iterator,Iterator> result;

result = thrust::mismatch(vec1.begin(), vec1.end(), vec2.begin());

// result.first  is vec1.begin() + 2
// result.second is vec2.begin() + 2
```

模板参数

- **InputIterator1**：是 Input Iterator 的模型，InputIterator1 的 value_type 等价于 InputIterator2 的 value_type
- **InputIterator2**：是 Input Iterator 的模型

函数参数

- **first1**：第一个序列的起始
- **last1**：第一个序列的末尾
- **first2**：第二个序列的起始

返回值

序列出现不同的第一个位置

另请参阅

- find
- find_if

thrust::mismatch

```
template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2,
  typename BinaryPredicate>
__host__ __device__ thrust::pair< InputIterator1, InputIterator2 >
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
mismatch(const thrust::detail::execution_policy_base< DerivedPolicy > &
 ↪exec,
  InputIterator1 first1,
  InputIterator1 last1,
  InputIterator2 first2,
  BinaryPredicate pred);
```

mismatch 查找两个范围 [first1, last1) 和 [first2, first2 + (last1 - first1)) 不同的第一个位置。mismatch 的两个版本使用不同的测试来判断元素是否不同。

此版本的 mismatch 查找 [first1, last1) 内使得 pred(*i, *(first2 + (i - first1)) 为 false 的第一个迭代器 i。返回值为一个 pair：第一个元素是 i，第二个元素是 *(first2 + (i - first1))。如果不存在这样的迭代器 i，则返回的 pair 为：第一个元素是 last1，第二个元素是 *(first2 + (last1 - first1))。

算法的执行由 exec 确定并行化。

```
#include <thrust/mismatch.h>
#include <thrust/device_vector.h>
#include <thrust/execution_policy.h>
...
thrust::device_vector<int> vec1(4);
thrust::device_vector<int> vec2(4);

vec1[0] = 0;  vec2[0] = 0;
vec1[1] = 5;  vec2[1] = 5;
vec1[2] = 3;  vec2[2] = 8;
vec1[3] = 7;  vec2[3] = 7;

typedef thrust::device_vector<int>::iterator Iterator;
thrust::pair<Iterator,Iterator> result;

result = thrust::mismatch(thrust::device, vec1.begin(), vec1.end(), vec2.
 ↪begin(), thrust::equal_to<int>());

// result.first  is vec1.begin() + 2
// result.second is vec2.begin() + 2
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **InputIterator1**：是 Input Iterator 的模型
- **InputIterator2**：是 Input Iterator 的模型
- **Predicate**：是 Input Iterator 的模型

函数参数

- **exec**：用于并行化的执行策略
- **first1**：第一个序列的起始
- **last1**：第一个序列的末尾
- **first2**：第二个序列的起始
- **pred**：比较元素的二元谓词

返回值

序列出现不同的第一个位置

另请参阅

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

- find
- find_if

thrust::mismatch

```
template <typename InputIterator1,
  typename InputIterator2,
  typename BinaryPredicate>
thrust::pair< InputIterator1, InputIterator2 >
mismatch(InputIterator1 first1,
  InputIterator1 last1,
  InputIterator2 first2,
  BinaryPredicate pred);
```

mismatch 查找两个范围 [first1, last1) 和 [first2, first2 + (last1 - first1)) 不同的第一个位置。mismatch 的两个版本使用不同的测试来判断元素是否不同。

此版本的 mismatch 查找 [first1, last1) 内使得 pred(*i, *(first2 + (i - first1)) 为 false 的第一个迭代器 i。返回值为一个 pair：第一个元素是 i，第二个元素是 *(first2 + (i - first1))。如果不存在这样的迭代器 i，则返回的 pair 为：第一个元素是 last1，第二个元素是 *(first2 + (last1 - first1))。

```
#include <thrust/mismatch.h>
#include <thrust/device_vector.h>
...
thrust::device_vector<int> vec1(4);
thrust::device_vector<int> vec2(4);

vec1[0] = 0;  vec2[0] = 0;
vec1[1] = 5;  vec2[1] = 5;
vec1[2] = 3;  vec2[2] = 8;
vec1[3] = 7;  vec2[3] = 7;

typedef thrust::device_vector<int>::iterator Iterator;
thrust::pair<Iterator,Iterator> result;

result = thrust::mismatch(vec1.begin(), vec1.end(), vec2.begin(),
↪thrust::equal_to<int>());

// result.first  is vec1.begin() + 2
// result.second is vec2.begin() + 2
```

模板参数

- **InputIterator1**：是 Input Iterator 的模型
- **InputIterator2**：是 Input Iterator 的模型
- **Predicate**：是 Input Iterator 的模型

函数参数

- **first1**：第一个序列的起始
- **last1**：第一个序列的末尾
- **first2**：第二个序列的起始
- **pred**：比较元素的二元谓词

返回值

序列出现不同的第一个位置

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

另请参阅

- find
- find_if

thrust::partition_point

```
template <typename DerivedPolicy,
  typename ForwardIterator,
  typename Predicate>
__host__ __device__ ForwardIterator
partition_point(const thrust::detail::execution_policy_base< DerivedPolicy
 ↪> & exec,
  ForwardIterator first,
  ForwardIterator last,
  Predicate pred);
```

partition_point 返回指向分区范围内 true 分区末尾的迭代器。partition_point 要求输入范围 [first,last) 是这样一个分区：所有满足 pred 的元素都应先于不满足 pred 的元素。

算法的执行由 exec 确定并行化。

```
#include <thrust/partition.h>
#include <thrust/execution_policy.h>

struct is_even
{
__host__ __device__
bool operator()(const int &x)
{
return (x % 2) == 0;
}
};

...

int A[] = {2, 4, 6, 8, 10, 1, 3, 5, 7, 9};
int * B = thrust::partition_point(thrust::host, A, A + 10, is_even());
// B - A is 5
// [A, B) contains only even values
```

备注

partition_point 与 find_if_not 尽管类似，但并不冗余。partition_point 的前提条件为更快的实现提供了机会。

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **ForwardIterator**：是 Forward Iterator 的模型，ForwardIterator 的 value_type 可转换为 Predicate 的 argument_type
- **Predicate**：是 Predicate 的模型

函数参数

- **exec**：用于并行化的执行策略
- **first**：待检查的范围的起始
- **last**：待检查的范围的末尾

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

- **pred**：决定范围 [first, last) 内每个元素属于哪个分区的函数对象

前提条件

范围 [first, last) 应根据 pred 划分

返回值

使得 all_of(first, mid, pred) 和 none_of(mid, last, pred) 都为 true 的迭代器 mid

另请参阅

- partition
- find_if_not

thrust::partition_point

```
template <typename ForwardIterator,
  typename Predicate>
ForwardIterator
partition_point(ForwardIterator first,
  ForwardIterator last,
  Predicate pred);
```

partition_point 返回指向分区范围内 true 分区末尾的迭代器。partition_point 要求输入范围 [first,last) 是这样一个分区：所有满足 pred 的元素都应先于不满足 pred 的元素。

```
#include <thrust/partition.h>

struct is_even
{
__host__ __device__
bool operator()(const int &x)
{
return (x % 2) == 0;
}
};

...

int A[] = {2, 4, 6, 8, 10, 1, 3, 5, 7, 9};
int * B = thrust::partition_point(A, A + 10, is_even());
// B - A is 5
// [A, B) contains only even values
```

备注

partition_point 与 find_if_not 尽管类似，但并不冗余。partition_point 的前提条件为更快的实现提供了机会。

模板参数

- **ForwardIterator**：是 Forward Iterator 的模型，并且 ForwardIterator 的 value_type 可转换为 Predicate 的 argument_type
- **Predicate**：是 Predicate 的模型

函数参数

- **first**：待检查的范围的起始
- **last**：待检查的范围的末尾

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

- **pred**：决定范围 [first, last) 内每个元素属于哪个分区的函数对象

前提条件

范围 [first, last) 应根据 pred 划分

返回值

使得 all_of(first, mid, pred) 和 none_of(mid, last, pred) 都为 true 的迭代器 mid

另请参阅

- partition
- find_if_not

2.1.6.2 二分搜索（Binary Search）

函数

```
template <typename DerivedPolicy,
  typename ForwardIterator,
  typename LessThanComparable>
__host__ __device__ ForwardIterator
thrust::lower_bound(const thrust::detail::execution_policy_base<
↪DerivedPolicy > & exec,
  ForwardIterator first,
  ForwardIterator last,
  const LessThanComparable & value);

template <class ForwardIterator,
  class LessThanComparable>
ForwardIterator
thrust::lower_bound(ForwardIterator first,
  ForwardIterator last,
  const LessThanComparable & value);

template <typename DerivedPolicy,
  typename ForwardIterator,
  typename T,
  typename StrictWeakOrdering>
__host__ __device__ ForwardIterator
thrust::lower_bound(const thrust::detail::execution_policy_base<
↪DerivedPolicy > & exec,
  ForwardIterator first,
  ForwardIterator last,
  const T & value,
  StrictWeakOrdering comp);

template <class ForwardIterator,
  class T,
  class StrictWeakOrdering>
ForwardIterator
thrust::lower_bound(ForwardIterator first,
  ForwardIterator last,
  const T & value,
  StrictWeakOrdering comp);

template <typename DerivedPolicy,
  typename ForwardIterator,
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
  typename LessThanComparable>
__host__ __device__ ForwardIterator
thrust::upper_bound(const thrust::detail::execution_policy_base<
 ↪DerivedPolicy > & exec,
  ForwardIterator first,
  ForwardIterator last,
  const LessThanComparable & value);

template <class ForwardIterator,
  class LessThanComparable>
ForwardIterator
thrust::upper_bound(ForwardIterator first,
  ForwardIterator last,
  const LessThanComparable & value);

template <typename DerivedPolicy,
  typename ForwardIterator,
  typename T,
  typename StrictWeakOrdering>
__host__ __device__ ForwardIterator
thrust::upper_bound(const thrust::detail::execution_policy_base<
 ↪DerivedPolicy > & exec,
  ForwardIterator first,
  ForwardIterator last,
  const T & value,
  StrictWeakOrdering comp);

template <class ForwardIterator,
  class T,
  class StrictWeakOrdering>
ForwardIterator
thrust::upper_bound(ForwardIterator first,
  ForwardIterator last,
  const T & value,
  StrictWeakOrdering comp);

template <typename DerivedPolicy,
  typename ForwardIterator,
  typename LessThanComparable>
__host__ __device__ bool
thrust::binary_search(const thrust::detail::execution_policy_base<
 ↪DerivedPolicy > & exec,
  ForwardIterator first,
  ForwardIterator last,
  const LessThanComparable & value);

template <class ForwardIterator,
  class LessThanComparable>
bool
thrust::binary_search(ForwardIterator first,
  ForwardIterator last,
  const LessThanComparable & value);

template <typename DerivedPolicy,
  typename ForwardIterator,
  typename T,
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
  typename StrictWeakOrdering>
__host__ __device__ bool
thrust::binary_search(const thrust::detail::execution_policy_base<
 ↪DerivedPolicy > & exec,
  ForwardIterator first,
  ForwardIterator last,
  const T & value,
  StrictWeakOrdering comp);

template <class ForwardIterator,
  class T,
  class StrictWeakOrdering>
bool
thrust::binary_search(ForwardIterator first,
  ForwardIterator last,
  const T & value,
  StrictWeakOrdering comp);

template <typename DerivedPolicy,
  typename ForwardIterator,
  typename LessThanComparable>
__host__ __device__ thrust::pair< ForwardIterator, ForwardIterator >
thrust::equal_range(const thrust::detail::execution_policy_base<
 ↪DerivedPolicy > & exec,
  ForwardIterator first,
  ForwardIterator last,
  const LessThanComparable & value);

template <class ForwardIterator,
  class LessThanComparable>
thrust::pair< ForwardIterator, ForwardIterator >
thrust::equal_range(ForwardIterator first,
  ForwardIterator last,
  const LessThanComparable & value);

template <typename DerivedPolicy,
  typename ForwardIterator,
  typename T,
  typename StrictWeakOrdering>
__host__ __device__ thrust::pair< ForwardIterator, ForwardIterator >
thrust::equal_range(const thrust::detail::execution_policy_base<
 ↪DerivedPolicy > & exec,
  ForwardIterator first,
  ForwardIterator last,
  const T & value,
  StrictWeakOrdering comp);

template <class ForwardIterator,
  class T,
  class StrictWeakOrdering>
thrust::pair< ForwardIterator, ForwardIterator >
thrust::equal_range(ForwardIterator first,
  ForwardIterator last,
  const T & value,
  StrictWeakOrdering comp);
```

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

thrust::lower_bound

```
template <typename DerivedPolicy,
  typename ForwardIterator,
  typename LessThanComparable>
__host__ __device__ ForwardIterator lower_bound(const
↪thrust::detail::execution_policy_base< DerivedPolicy > & exec,
  ForwardIterator first,
  ForwardIterator last,
  const LessThanComparable & value);
```

lower_bound 是二分搜索的一个版本：它试图在有序范围 [first, last) 内查找元素值。具体而言，它返回可以在不打乱顺序的情况下插入值的第一个位置。 此版本的 lower_bound 使用 operator< 进行比较，返回 [first, last) 内最远的迭代器 i ，使得 [first, i) 内的每一个迭代器 j ，满足 *j < value。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::device 并行化执行策略，使用 lower_bound 搜索有序范围内的值：

```
#include <thrust/binary_search.h>
#include <thrust/device_vector.h>
#include <thrust/execution_policy.h>
...
thrust::device_vector<int> input(5);

input[0] = 0;
input[1] = 2;
input[2] = 5;
input[3] = 7;
input[4] = 8;

thrust::lower_bound(thrust::device, input.begin(), input.end(), 0); //
↪returns input.begin()
thrust::lower_bound(thrust::device, input.begin(), input.end(), 1); //
↪returns input.begin() + 1
thrust::lower_bound(thrust::device, input.begin(), input.end(), 2); //
↪returns input.begin() + 1
thrust::lower_bound(thrust::device, input.begin(), input.end(), 3); //
↪returns input.begin() + 2
thrust::lower_bound(thrust::device, input.begin(), input.end(), 8); //
↪returns input.begin() + 4
thrust::lower_bound(thrust::device, input.begin(), input.end(), 9); //
↪returns input.end()
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **ForwardIterator**：是 Forward Iterator 的模型
- **LessThanComparable**：是 LessThanComparable 的模型

函数参数

- **exec**：用于并行化的执行策略
- **first**：有序序列的起始
- **last**：有序序列的末尾
- **value**：要查找的值

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

返回值

使得 *i < value 的最远的迭代器 i

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/lower_bound
- upper_bound
- equal_range
- Binary Search

thrust::lower_bound

```
template <class ForwardIterator,
  class LessThanComparable>
ForwardIterator
lower_bound(ForwardIterator first,
  ForwardIterator last,
  const LessThanComparable & value);
```

lower_bound 是二分搜索的一个版本：它试图在有序范围 [first, last) 内查找元素值。具体而言，它返回可以在不打乱顺序的情况下插入值的第一个位置。 此版本的 lower_bound 使用 operator< 进行比较，返回 [first, last) 内最远的迭代器 i ，使得 [first, i) 内的每一个迭代器 j ，满足 *j < value。

以下代码片段演示了如何使用 lower_bound 搜索有序范围内的值：

```
#include <thrust/binary_search.h>
#include <thrust/device_vector.h>
...
thrust::device_vector<int> input(5);

input[0] = 0;
input[1] = 2;
input[2] = 5;
input[3] = 7;
input[4] = 8;

thrust::lower_bound(input.begin(), input.end(), 0); // returns input.
↪begin()
thrust::lower_bound(input.begin(), input.end(), 1); // returns input.
↪begin() + 1
thrust::lower_bound(input.begin(), input.end(), 2); // returns input.
↪begin() + 1
thrust::lower_bound(input.begin(), input.end(), 3); // returns input.
↪begin() + 2
thrust::lower_bound(input.begin(), input.end(), 8); // returns input.
↪begin() + 4
thrust::lower_bound(input.begin(), input.end(), 9); // returns input.end()
```

模板参数

- **ForwardIterator**：是 Forward Iterator 的模型
- **LessThanComparable**：是 LessThanComparable 的模型

函数参数

- **first**：有序序列的起始
- **last**：有序序列的末尾

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

- **value**：要查找的值

返回值

使得 *i < value 的最远的迭代器 i

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/lower_bound
- upper_bound
- equal_range
- Binary Search

thrust::lower_bound

```
template <typename DerivedPolicy,
  typename ForwardIterator,
  typename T,
  typename StrictWeakOrdering>
__host__ __device__ ForwardIterator
lower_bound(const thrust::detail::execution_policy_base< DerivedPolicy > &
↪exec,
  ForwardIterator first,
  ForwardIterator last,
  const T & value,
  StrictWeakOrdering comp);
```

lower_bound 是二分搜索的一个版本：它试图在有序范围 [first, last) 内查找元素值。具体而言，它返回可以在不打乱顺序的情况下插入值的第一个位置。 此版本的 lower_bound 使用函数对象 comp 进行比较，返回 [first, last) 内最远的迭代器 i ，使得 [first, i) 内的每一个迭代器 j ，满足 comp(*j, value) 为 true。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::device 并行化执行策略，使用 lower_bound 搜索有序范围内的值：

```
#include <thrust/binary_search.h>
#include <thrust/device_vector.h>
#include <thrust/functional.h>
#include <thrust/execution_policy.h>
...
thrust::device_vector<int> input(5);

input[0] = 0;
input[1] = 2;
input[2] = 5;
input[3] = 7;
input[4] = 8;

thrust::lower_bound(input.begin(), input.end(), 0, thrust::less<int>()); //
↪ returns input.begin()
thrust::lower_bound(input.begin(), input.end(), 1, thrust::less<int>()); //
↪ returns input.begin() + 1
thrust::lower_bound(input.begin(), input.end(), 2, thrust::less<int>()); //
↪ returns input.begin() + 1
thrust::lower_bound(input.begin(), input.end(), 3, thrust::less<int>()); //
↪ returns input.begin() + 2
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 17:00:00

(续上页)

```
thrust::lower_bound(input.begin(), input.end(), 8, thrust::less<int>()); //
↪ returns input.begin() + 4
thrust::lower_bound(input.begin(), input.end(), 9, thrust::less<int>()); //
↪ returns input.end()
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **ForwardIterator**：是 Forward Iterator 的模型
- **T**：与 ForwardIterator 的 value_type 类似
- **StrictWeakOrdering**：是 Strict Weak Ordering 的模型

函数参数

- **exec**：用于并行化的执行策略
- **first**：有序序列的起始
- **last**：有序序列的末尾
- **value**：要查找的值
- **comp**：比较运算符

返回值

使得 comp(*i, value) 为 true 的最远的迭代器 i

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/lower_bound
- upper_bound
- equal_range
- Binary Search

thrust::lower_bound

```
template <class ForwardIterator,
  class T,
  class StrictWeakOrdering>
ForwardIterator
lower_bound(ForwardIterator first,
  ForwardIterator last,
  const T & value,
  StrictWeakOrdering comp);
```

lower_bound 是二分搜索的一个版本：它试图在有序范围 [first, last) 内查找元素值。具体而言，它返回可以在不打乱顺序的情况下插入值的第一个位置。 此版本的 lower_bound 使用函数对象 comp 进行比较，返回 [first, last) 内最远的迭代器 i ，使得 [first, i) 内的每一个迭代器 j ，满足 comp(*j, value) 为 true。

以下代码片段演示了如何使用 lower_bound 搜索有序范围内的值：

```
#include <thrust/binary_search.h>
#include <thrust/device_vector.h>
#include <thrust/functional.h>
...
thrust::device_vector<int> input(5);
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
input[0] = 0;
input[1] = 2;
input[2] = 5;
input[3] = 7;
input[4] = 8;

thrust::lower_bound(input.begin(), input.end(), 0, thrust::less<int>()); //
↪ returns input.begin()
thrust::lower_bound(input.begin(), input.end(), 1, thrust::less<int>()); //
↪ returns input.begin() + 1
thrust::lower_bound(input.begin(), input.end(), 2, thrust::less<int>()); //
↪ returns input.begin() + 1
thrust::lower_bound(input.begin(), input.end(), 3, thrust::less<int>()); //
↪ returns input.begin() + 2
thrust::lower_bound(input.begin(), input.end(), 8, thrust::less<int>()); //
↪ returns input.begin() + 4
thrust::lower_bound(input.begin(), input.end(), 9, thrust::less<int>()); //
↪ returns input.end()
```

模板参数

- **ForwardIterator**：是 Forward Iterator 的模型
- **T**：与 ForwardIterator 的 value_type 类似
- **StrictWeakOrdering**：是 Strict Weak Ordering 的模型

函数参数

- **first**：有序序列的起始
- **last**：有序序列的末尾
- **value**：要查找的值
- **comp**：比较运算符

返回值

使得 comp(*i, value) 为 true 的最远的迭代器 i

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/lower_bound
- upper_bound
- equal_range
- Binary Search

thrust::upper_bound

```
template <typename DerivedPolicy,
  typename ForwardIterator,
  typename LessThanComparable>
__host__ __device__ ForwardIterator
upper_bound(const thrust::detail::execution_policy_base< DerivedPolicy > &
↪exec,
  ForwardIterator first,
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
  ForwardIterator last,
  const LessThanComparable & value);
```

upper_bound 是二分搜索的一个版本：它试图在有序范围 [first, last) 内查找元素值。具体而言，它返回可以在不打乱顺序的情况下插入值的最后一个位置。 此版本的 upper_bound 使用 operator< 进行比较，返回 [first, last) 内最远的迭代器 i ，使得 [first, i) 内的每一个迭代器 j ，满足 value < *j 为 false。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::device 并行化执行策略，使用 upper_bound 搜索有序范围内的值：

```
#include <thrust/binary_search.h>
#include <thrust/device_vector.h>
#include <thrust/execution_policy.h>
...
thrust::device_vector<int> input(5);

input[0] = 0;
input[1] = 2;
input[2] = 5;
input[3] = 7;
input[4] = 8;

thrust::upper_bound(thrust::device, input.begin(), input.end(), 0); //
↪returns input.begin() + 1
thrust::upper_bound(thrust::device, input.begin(), input.end(), 1); //
↪returns input.begin() + 1
thrust::upper_bound(thrust::device, input.begin(), input.end(), 2); //
↪returns input.begin() + 2
thrust::upper_bound(thrust::device, input.begin(), input.end(), 3); //
↪returns input.begin() + 2
thrust::upper_bound(thrust::device, input.begin(), input.end(), 8); //
↪returns input.end()
thrust::upper_bound(thrust::device, input.begin(), input.end(), 9); //
↪returns input.end()
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **ForwardIterator**：是 Forward Iterator 的模型
- **LessThanComparable**：是 LessThanComparable 的模型

函数参数

- **exec**：用于并行化的执行策略
- **first**：有序序列的起始
- **last**：有序序列的末尾
- **value**：要查找的值

返回值

使得 value < *i 为 false 的最远的迭代器 i

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/upper_bound
- lower_bound

沐曦 MetaX Confidential 2024-11-18 15:00:00

- equal_range
- Binary Search

thrust::upper_bound

```
template <class ForwardIterator,
  class LessThanComparable>
ForwardIterator
upper_bound(ForwardIterator first,
  ForwardIterator last,
  const LessThanComparable & value);
```

upper_bound 是二分搜索的一个版本：它试图在有序范围 [first, last) 内查找元素值。具体而言，它返回可以在不打乱顺序的情况下插入值的最后一个位置。 此版本的 upper_bound 使用 operator< 进行比较，返回 [first, last) 内最远的迭代器 i ，使得 [first, i) 内的每一个迭代器 j ，满足 value < *j 为 false。

以下代码片段演示了如何使用 upper_bound 搜索有序范围内的值：

```
#include <thrust/binary_search.h>
#include <thrust/device_vector.h>
...
thrust::device_vector<int> input(5);

input[0] = 0;
input[1] = 2;
input[2] = 5;
input[3] = 7;
input[4] = 8;

thrust::upper_bound(input.begin(), input.end(), 0); // returns input.
↪begin() + 1
thrust::upper_bound(input.begin(), input.end(), 1); // returns input.
↪begin() + 1
thrust::upper_bound(input.begin(), input.end(), 2); // returns input.
↪begin() + 2
thrust::upper_bound(input.begin(), input.end(), 3); // returns input.
↪begin() + 2
thrust::upper_bound(input.begin(), input.end(), 8); // returns input.end()
thrust::upper_bound(input.begin(), input.end(), 9); // returns input.end()
```

模板参数

- **ForwardIterator**：是 Forward Iterator 的模型
- **LessThanComparable**：是 LessThanComparable 的模型

函数参数

- **first**：有序序列的起始
- **last**：有序序列的末尾
- **value**：要查找的值

返回值

使得 value < *i 为 false 的最远的迭代器 i

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/upper_bound

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

- lower_bound
- equal_range
- Binary Search

thrust::upper_bound

```
template <typename DerivedPolicy,
  typename ForwardIterator,
  typename T,
  typename StrictWeakOrdering>
__host__ __device__ ForwardIterator
upper_bound(const thrust::detail::execution_policy_base< DerivedPolicy > &
↪exec,
  ForwardIterator first,
  ForwardIterator last,
  const T & value,
  StrictWeakOrdering comp);
```

upper_bound 是二分搜索的一个版本：它试图在有序范围 [first, last) 内查找元素值。具体而言，它返回可以在不打乱顺序的情况下插入值的最后一个位置。 此版本的 upper_bound 使用函数对象 comp 进行比较，返回 [first, last) 内最远的迭代器 i ，使得 [first, i) 内的每一个迭代器 j ，满足 comp(value, *j) 为 false。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::device 并行化执行策略，使用 upper_bound 搜索有序范围内的值：

```
#include <thrust/binary_search.h>
#include <thrust/device_vector.h>
#include <thrust/functional.h>
#include <thrust/execution_policy.h>
...
thrust::device_vector<int> input(5);

input[0] = 0;
input[1] = 2;
input[2] = 5;
input[3] = 7;
input[4] = 8;

thrust::upper_bound(thrust::device, input.begin(), input.end(), 0,
↪thrust::less<int>()); // returns input.begin() + 1
thrust::upper_bound(thrust::device, input.begin(), input.end(), 1,
↪thrust::less<int>()); // returns input.begin() + 1
thrust::upper_bound(thrust::device, input.begin(), input.end(), 2,
↪thrust::less<int>()); // returns input.begin() + 2
thrust::upper_bound(thrust::device, input.begin(), input.end(), 3,
↪thrust::less<int>()); // returns input.begin() + 2
thrust::upper_bound(thrust::device, input.begin(), input.end(), 8,
↪thrust::less<int>()); // returns input.end()
thrust::upper_bound(thrust::device, input.begin(), input.end(), 9,
↪thrust::less<int>()); // returns input.end()
```

模板参数

- **DerivedPolicy**：派生执行策略的名称

- **ForwardIterator**：是 Forward Iterator 的模型
- **T**：与 ForwardIterator 的 value_type 类似
- **StrictWeakOrdering**：是 Strict Weak Ordering 的模型

函数参数

- **exec**：用于并行化的执行策略
- **first**：有序序列的起始
- **last**：有序序列的末尾
- **value**：要查找的值
- **comp**：比较运算符

返回值

使得 comp(value, *i) 为 false 的最远的迭代器 i

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/upper_bound
- lower_bound
- equal_range
- Binary Search

thrust::upper_bound

```
template <class ForwardIterator,
  class T,
  class StrictWeakOrdering>
ForwardIterator
upper_bound(ForwardIterator first,
  ForwardIterator last,
  const T & value,
  StrictWeakOrdering comp);
```

upper_bound 是二分搜索的一个版本：它试图在有序范围 [first, last) 内查找元素值。具体而言，它返回可以在不打乱顺序的情况下插入值的最后一个位置。 此版本的 upper_bound 使用函数对象 comp 进行比较，返回 [first, last) 内最远的迭代器 i ，使得 [first, i) 内的每一个迭代器 j ，满足 comp(value, *j) 为 false。

以下代码片段演示了如何使用 upper_bound 搜索有序范围内的值：

```
#include <thrust/binary_search.h>
#include <thrust/device_vector.h>
#include <thrust/functional.h>
...
thrust::device_vector<int> input(5);

input[0] = 0;
input[1] = 2;
input[2] = 5;
input[3] = 7;
input[4] = 8;

thrust::upper_bound(input.begin(), input.end(), 0, thrust::less<int>()); //
↪ returns input.begin() + 1
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
thrust::upper_bound(input.begin(), input.end(), 1, thrust::less<int>()); //
↪ returns input.begin() + 1
thrust::upper_bound(input.begin(), input.end(), 2, thrust::less<int>()); //
↪ returns input.begin() + 2
thrust::upper_bound(input.begin(), input.end(), 3, thrust::less<int>()); //
↪ returns input.begin() + 2
thrust::upper_bound(input.begin(), input.end(), 8, thrust::less<int>()); //
↪ returns input.end()
thrust::upper_bound(input.begin(), input.end(), 9, thrust::less<int>()); //
↪ returns input.end()
```

模板参数

- **ForwardIterator**：是 Forward Iterator 的模型
- **T**：与 ForwardIterator 的 value_type 类似
- **StrictWeakOrdering**：是 Strict Weak Ordering 的模型

函数参数

- **first**：有序序列的起始
- **last**：有序序列的末尾
- **value**：要查找的值
- **comp**：比较运算符

返回值

使得 comp(value, *i) 为 false 的最远的迭代器 i

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/upper_bound
- lower_bound
- equal_range
- Binary Search

thrust::binary_search

```
template <typename DerivedPolicy,
  typename ForwardIterator,
  typename LessThanComparable>
__host__ __device__ bool
binary_search(const thrust::detail::execution_policy_base< DerivedPolicy >
↪& exec,
  ForwardIterator first,
  ForwardIterator last,
  const LessThanComparable & value);
```

binary_search 是二分搜索的一个版本：它试图在有序范围 [first, last) 内查找元素值。如果在 [first, last) 中存在与 value 等价的元素，则返回 true；如果不存在这样的元素，那么返回 false。具体而言，当且仅当 [first, last) 内存在使得 *i < value 和 value < *i 都为 false 的迭代器 i，此版本返回 true。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::device 并行化执行策略，使用 binary_search 搜索有序范围内的值：

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

```
#include <thrust/binary_search.h>
#include <thrust/device_vector.h>
#include <thrust/execution_policy.h>
...
thrust::device_vector<int> input(5);

input[0] = 0;
input[1] = 2;
input[2] = 5;
input[3] = 7;
input[4] = 8;

thrust::binary_search(thrust::device, input.begin(), input.end(), 0); //
↪returns true
thrust::binary_search(thrust::device, input.begin(), input.end(), 1); //
↪returns false
thrust::binary_search(thrust::device, input.begin(), input.end(), 2); //
↪returns true
thrust::binary_search(thrust::device, input.begin(), input.end(), 3); //
↪returns false
thrust::binary_search(thrust::device, input.begin(), input.end(), 8); //
↪returns true
thrust::binary_search(thrust::device, input.begin(), input.end(), 9); //
↪returns false
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **ForwardIterator**：是 Forward Iterator 的模型
- **LessThanComparable**：是 LessThanComparable 的模型

函数参数

- **exec**：用于并行化的执行策略
- **first**：有序序列的起始
- **last**：有序序列的末尾
- **value**：要查找的值

返回值

如果 [first, last) 内存在等价元素，返回 true；否则，返回 false

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/binary_search
- lower_bound
- upper_bound
- equal_range

thrust::binary_search

```
template <class ForwardIterator,
  class LessThanComparable>
bool
binary_search(ForwardIterator first,
```

(下页继续)

飞曦信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
  ForwardIterator last,
  const LessThanComparable & value);
```

binary_search 是二分搜索的一个版本：它试图在有序范围 [first, last) 内查找元素值。如果在 [first, last) 中存在与 value 等价的元素，则返回 true；如果不存在这样的元素，那么返回 false。具体而言，当且仅当 [first, last) 内存在使得 *i < value 和 value < *i 都为 false 的迭代器 i，此版本返回 true。

以下代码片段演示了如何使用 binary_search 搜索有序范围内的值：

```
#include <thrust/binary_search.h>
#include <thrust/device_vector.h>
...
thrust::device_vector<int> input(5);

input[0] = 0;
input[1] = 2;
input[2] = 5;
input[3] = 7;
input[4] = 8;

thrust::binary_search(input.begin(), input.end(), 0); // returns true
thrust::binary_search(input.begin(), input.end(), 1); // returns false
thrust::binary_search(input.begin(), input.end(), 2); // returns true
thrust::binary_search(input.begin(), input.end(), 3); // returns false
thrust::binary_search(input.begin(), input.end(), 8); // returns true
thrust::binary_search(input.begin(), input.end(), 9); // returns false
```

模板参数

- **ForwardIterator**：是 Forward Iterator 的模型
- **LessThanComparable**：是 LessThanComparable 的模型

函数参数

- **first**：有序序列的起始
- **last**：有序序列的末尾
- **value**：要查找的值

返回值

如果 [first, last) 内存在等价元素，返回 true；否则，返回 false

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/binary_search
- lower_bound
- upper_bound
- equal_range

thrust::binary_search

```
template <typename DerivedPolicy,
  typename ForwardIterator,
  typename T,
  typename StrictWeakOrdering>
__host__ __device__ bool
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
binary_search(const thrust::detail::execution_policy_base< DerivedPolicy >
↪& exec,
  ForwardIterator first,
  ForwardIterator last,
  const T & value,
  StrictWeakOrdering comp);
```

binary_search 是二分搜索的一个版本：它试图在有序范围 [first, last) 内查找元素值。如果在 [first, last) 中存在与 value 等价的元素，则返回 true；如果不存在这样的元素，那么返回 false。具体而言，当且仅当 [first, last) 内存在使得 comp(*i, value) 和 comp(value, *i) 都为 false 的迭代器 i，此版本返回 true。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::device 并行化执行策略，使用 binary_search 搜索有序范围内的值：

```
#include <thrust/binary_search.h>
#include <thrust/device_vector.h>
#include <thrust/functional.h>
#include <thrust/execution_policy.h>
...
thrust::device_vector<int> input(5);

input[0] = 0;
input[1] = 2;
input[2] = 5;
input[3] = 7;
input[4] = 8;

thrust::binary_search(thrust::device, input.begin(), input.end(), 0,
↪thrust::less<int>()); // returns true
thrust::binary_search(thrust::device, input.begin(), input.end(), 1,
↪thrust::less<int>()); // returns false
thrust::binary_search(thrust::device, input.begin(), input.end(), 2,
↪thrust::less<int>()); // returns true
thrust::binary_search(thrust::device, input.begin(), input.end(), 3,
↪thrust::less<int>()); // returns false
thrust::binary_search(thrust::device, input.begin(), input.end(), 8,
↪thrust::less<int>()); // returns true
thrust::binary_search(thrust::device, input.begin(), input.end(), 9,
↪thrust::less<int>()); // returns false
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **ForwardIterator**：是 Forward Iterator 的模型
- **T**：与 ForwardIterator 的 value_type 类似
- **StrictWeakOrdering**：是 Strict Weak Ordering 的模型

函数参数

- **exec**：用于并行化的执行策略
- **first**：有序序列的起始
- **last**：有序序列的末尾
- **value**：要查找的值
- **comp**：比较运算符

飞腾信息 MetaX Confidential 2024-11-19 15:00:00

返回值

如果 [first, last) 内存在等价元素，返回 true；否则，返回 false

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/binary_search
- lower_bound
- upper_bound
- equal_range

thrust::binary_search

```
template <class ForwardIterator,
  class T,
  class StrictWeakOrdering>
bool
binary_search(ForwardIterator first,
  ForwardIterator last,
  const T & value,
  StrictWeakOrdering comp);
```

binary_search 是二分搜索的一个版本：它试图在有序范围 [first, last) 内查找元素值。如果在 [first, last) 中存在与 value 等价的元素，则返回 true；如果不存在这样的元素，那么返回 false。具体而言，当且仅当 [first, last) 内存在使得 comp(*i, value) 和 comp(value, *i) 都为 false 的迭代器 i，此版本返回 true。

以下代码片段演示了如何使用 binary_search 搜索有序范围内的值：

```
#include <thrust/binary_search.h>
#include <thrust/device_vector.h>
#include <thrust/functional.h>
...
thrust::device_vector<int> input(5);

input[0] = 0;
input[1] = 2;
input[2] = 5;
input[3] = 7;
input[4] = 8;

thrust::binary_search(input.begin(), input.end(), 0, thrust::less<int>());
↪// returns true
thrust::binary_search(input.begin(), input.end(), 1, thrust::less<int>());
↪// returns false
thrust::binary_search(input.begin(), input.end(), 2, thrust::less<int>());
↪// returns true
thrust::binary_search(input.begin(), input.end(), 3, thrust::less<int>());
↪// returns false
thrust::binary_search(input.begin(), input.end(), 8, thrust::less<int>());
↪// returns true
thrust::binary_search(input.begin(), input.end(), 9, thrust::less<int>());
↪// returns false
```

模板参数

- **ForwardIterator**：是 Forward Iterator 的模型
- **T**：与 ForwardIterator 的 value_type 类似

沐曦信息 MetaX Confidential 2024-11-18 15:00:00

- **StrictWeakOrdering**：是 Strict Weak Ordering 的模型

函数参数

- **first**：有序序列的起始
- **last**：有序序列的末尾
- **value**：要查找的值
- **comp**：比较运算符

返回值

如果 [first, last) 内存在等价元素，返回 true；否则，返回 false

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/binary_search
- lower_bound
- upper_bound
- equal_range

thrust::equal_range

```
template <typename DerivedPolicy,
  typename ForwardIterator,
  typename LessThanComparable>
__host__ __device__ thrust::pair< ForwardIterator, ForwardIterator >
equal_range(const thrust::detail::execution_policy_base< DerivedPolicy > &
↪exec,
  ForwardIterator first,
  ForwardIterator last,
  const LessThanComparable & value);
```

equal_range 是二分搜索的一个版本：它试图在有序范围 [first, last) 内查找元素值。equal_range 的返回值本质上是 lower_bound 和 upper_bound 返回值的组合：它返回一对迭代器 i 和 j，其中 i 是在不打乱顺序的情况下插入值的第一个位置，j 是在不打乱顺序的情况下插入值的最后一个位置。由此可见，范围 [i, j) 内的每个元素都等价于 value，并且 [i, j) 是 [first, last) 内具有此属性的最大的子范围。

此版本的 equal_range 返回一对迭代器 [i, j)。其中 i 是 [first, last) 内，使得 [first, i) 内的每一个迭代器 k 满足 *k < value，最远的迭代器。j 是 [first, last) 内，使得 [first, j) 内的每一个迭代器 k 满足 value < *k 为 false，最远的迭代器。对于 [i, j) 中的每个迭代器 k，value < *k 和 *k < value 都不为 true。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::device 并行化执行策略，使用 equal_range 搜索有序范围内的值：

```
#include <thrust/binary_search.h>
#include <thrust/device_vector.h>
#include <thrust/execution_policy.h>
...
thrust::device_vector<int> input(5);

input[0] = 0;
input[1] = 2;
input[2] = 5;
input[3] = 7;
input[4] = 8;
```

(下页继续)

(续上页)

```
thrust::equal_range(thrust::device, input.begin(), input.end(), 0); //
↪returns [input.begin(), input.begin() + 1)
thrust::equal_range(thrust::device, input.begin(), input.end(), 1); //
↪returns [input.begin() + 1, input.begin() + 1)
thrust::equal_range(thrust::device, input.begin(), input.end(), 2); //
↪returns [input.begin() + 1, input.begin() + 2)
thrust::equal_range(thrust::device, input.begin(), input.end(), 3); //
↪returns [input.begin() + 2, input.begin() + 2)
thrust::equal_range(thrust::device, input.begin(), input.end(), 8); //
↪returns [input.begin() + 4, input.end)
thrust::equal_range(thrust::device, input.begin(), input.end(), 9); //
↪returns [input.end(), input.end)
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **ForwardIterator**：是 Forward Iterator 的模型
- **LessThanComparable**：是 LessThanComparable 的模型

函数参数

- **exec**：用于并行化的执行策略
- **first**：有序序列的起始
- **last**：有序序列的末尾
- **value**：要查找的值

返回值

定义等价元素范围的一对迭代器 [i, j)

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/equal_range
- lower_bound
- upper_bound
- Binary Search

thrust::equal_range

```
template <class ForwardIterator,
  class LessThanComparable>
thrust::pair< ForwardIterator, ForwardIterator >
equal_range(ForwardIterator first,
  ForwardIterator last,
  const LessThanComparable & value);
```

equal_range 是二分搜索的一个版本：它试图在有序范围 [first, last) 内查找元素值。equal_range 的返回值本质上是 lower_bound 和 upper_bound 返回值的组合：它返回一对迭代器 i 和 j，其中 i 是在不打乱顺序的情况下插入值的第一个位置，j 是在不打乱顺序的情况下插入值的最后一个位置。由此可见，范围 [i, j) 内的每个元素都等价于 value，并且 [i, j) 是 [first, last) 内具有此属性的最大的子范围。

此版本的 equal_range 返回一对迭代器 [i, j)。其中 i 是 [first, last) 内，使得 [first, i) 内的每一个迭代器 k 满足 *k < value，最远的迭代器。j 是 [first, last) 内，使得 [first, j) 内的每一个迭代器 k 满足 value < *k 为 false，最远的迭代器。对于 [i, j) 中的每个迭代器 k，value < *k 和 *k < value 都不为 true。

以下代码片段演示了如何使用 equal_range 搜索有序范围内的值：

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

```
#include <thrust/binary_search.h>
#include <thrust/device_vector.h>
...
thrust::device_vector<int> input(5);

input[0] = 0;
input[1] = 2;
input[2] = 5;
input[3] = 7;
input[4] = 8;

thrust::equal_range(input.begin(), input.end(), 0); // returns [input.
↪begin(), input.begin() + 1)
thrust::equal_range(input.begin(), input.end(), 1); // returns [input.
↪begin() + 1, input.begin() + 1)
thrust::equal_range(input.begin(), input.end(), 2); // returns [input.
↪begin() + 1, input.begin() + 2)
thrust::equal_range(input.begin(), input.end(), 3); // returns [input.
↪begin() + 2, input.begin() + 2)
thrust::equal_range(input.begin(), input.end(), 8); // returns [input.
↪begin() + 4, input.end)
thrust::equal_range(input.begin(), input.end(), 9); // returns [input.
↪end(), input.end)
```

模板参数

- **ForwardIterator**：是 Forward Iterator 的模型
- **LessThanComparable**：是 LessThanComparable 的模型

函数参数

- **first**：有序序列的起始
- **last**：有序序列的末尾
- **value**：要查找的值

返回值

定义等价元素范围的一对迭代器 [i, j)

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/equal_range
- lower_bound
- upper_bound
- Binary Search

thrust::equal_range

```
template <typename DerivedPolicy,
  typename ForwardIterator,
  typename T,
  typename StrictWeakOrdering>
__host__ __device__ thrust::pair< ForwardIterator, ForwardIterator >
equal_range(const thrust::detail::execution_policy_base< DerivedPolicy > &
↪exec,
  ForwardIterator first,
```

(下页继续)

飞腾信息 MetaX confidential 2024-11-18 15:00:00

(续上页)

```
  ForwardIterator last,
  const T & value,
  StrictWeakOrdering comp);
```

equal_range 是二分搜索的一个版本：它试图在有序范围 [first, last) 内查找元素值。equal_range 的返回值本质上是 lower_bound 和 upper_bound 返回值的组合：它返回一对迭代器 i 和 j，其中 i 是在不打乱顺序的情况下插入值的第一个位置，j 是在不打乱顺序的情况下插入值的最后一个位置。由此可见，范围 [i, j) 内的每个元素都等价于 value，并且 [i, j) 是 [first, last) 内具有此属性的最大的子范围。

此版本的 equal_range 返回一对迭代器 [i, j)。i 是 [first, last) 内，使得 [first, i) 内的每一个迭代器 k 满足 comp(*k, value) 为 true，最远的迭代器。j 是 [first, last) 内，使得 [first, last) 内的每一个迭代器 k 满足 comp(value, *k) 为 false，最远的迭代器。对于 [i, j) 中的每个迭代器 k，comp(value, *k) 和 comp(*k, value) 都不为 true。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::device 并行化执行策略，使用 equal_range 搜索有序范围内的值：

```
#include <thrust/binary_search.h>
#include <thrust/device_vector.h>
#include <thrust/functional.h>
#include <thrust/execution_policy.h>
...
thrust::device_vector<int> input(5);

input[0] = 0;
input[1] = 2;
input[2] = 5;
input[3] = 7;
input[4] = 8;

thrust::equal_range(thrust::device, input.begin(), input.end(), 0,
 ↪thrust::less<int>()); // returns [input.begin(), input.begin() + 1)
thrust::equal_range(thrust::device, input.begin(), input.end(), 1,
 ↪thrust::less<int>()); // returns [input.begin() + 1, input.begin() + 1)
thrust::equal_range(thrust::device, input.begin(), input.end(), 2,
 ↪thrust::less<int>()); // returns [input.begin() + 1, input.begin() + 2)
thrust::equal_range(thrust::device, input.begin(), input.end(), 3,
 ↪thrust::less<int>()); // returns [input.begin() + 2, input.begin() + 2)
thrust::equal_range(thrust::device, input.begin(), input.end(), 8,
 ↪thrust::less<int>()); // returns [input.begin() + 4, input.end)
thrust::equal_range(thrust::device, input.begin(), input.end(), 9,
 ↪thrust::less<int>()); // returns [input.end(), input.end)
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **ForwardIterator**：是 Forward Iterator 的模型
- **T**：与 ForwardIterator 的 value_type 类似
- **StrictWeakOrdering**：是 Strict Weak Ordering 的模型

函数参数

- **exec**：用于并行化的执行策略
- **first**：有序序列的起始
- **last**：有序序列的末尾
- **value**：要查找的值

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

- **comp**：比较运算符

返回值

定义等价元素范围的一对迭代器 [i, j)

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/equal_range
- lower_bound
- upper_bound
- Binary Search

thrust::equal_range

```
template <class ForwardIterator,
  class T,
  class StrictWeakOrdering>
thrust::pair< ForwardIterator, ForwardIterator >
equal_range(ForwardIterator first,
  ForwardIterator last,
  const T & value,
  StrictWeakOrdering comp);
```

equal_range 是二分搜索的一个版本：它试图在有序范围 [first, last) 内查找元素值。equal_range 的返回值本质上是 lower_bound 和 upper_bound 返回值的组合：它返回一对迭代器 i 和 j，其中 i 是在不打乱顺序的情况下插入值的第一个位置，j 是在不打乱顺序的情况下插入值的最后一个位置。由此可见，范围 [i, j) 内的每个元素都等价于 value，并且 [i, j) 是 [first, last) 内具有此属性的最大的子范围。

此版本的 equal_range 返回一对迭代器 [i, j)。i 是 [first, last) 内，使得 [first, i) 内的每一个迭代器 k 满足 comp(*k, value) 为 true，最远的迭代器。j 是 [first, last) 内，使得 [first, last) 内的每一个迭代器 k 满足 comp(value, *k) 为 false，最远的迭代器。对于 [i, j) 中的每个迭代器 k，comp(value, *k) 和 comp(*k, value) 都不为 true。

以下代码片段演示了如何使用 equal_range 搜索有序范围内的值：

```
#include <thrust/binary_search.h>
#include <thrust/device_vector.h>
#include <thrust/functional.h>
...
thrust::device_vector<int> input(5);

input[0] = 0;
input[1] = 2;
input[2] = 5;
input[3] = 7;
input[4] = 8;

thrust::equal_range(input.begin(), input.end(), 0, thrust::less<int>()); //
↪ returns [input.begin(), input.begin() + 1)
thrust::equal_range(input.begin(), input.end(), 1, thrust::less<int>()); //
↪ returns [input.begin() + 1, input.begin() + 1)
thrust::equal_range(input.begin(), input.end(), 2, thrust::less<int>()); //
↪ returns [input.begin() + 1, input.begin() + 2)
thrust::equal_range(input.begin(), input.end(), 3, thrust::less<int>()); //
↪ returns [input.begin() + 2, input.begin() + 2)
thrust::equal_range(input.begin(), input.end(), 8, thrust::less<int>()); //
↪ returns [input.begin() + 4, input.end)
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:50:00

(续上页)

```
thrust::equal_range(input.begin(), input.end(), 9, thrust::less<int>()); //
↪ returns [input.end(), input.end)
```

模板参数

- **ForwardIterator**：是 Forward Iterator 的模型
- **T**：与 ForwardIterator 的 value_type 类似
- **StrictWeakOrdering**：是 Strict Weak Ordering 的模型

函数参数

- **first**：有序序列的起始
- **last**：有序序列的末尾
- **value**：要查找的值
- **comp**：比较运算符

返回值

定义等价元素范围的一对迭代器 [i, j)

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/equal_range
- lower_bound
- upper_bound
- Binary Search

向量化搜索（Vectorized Search）

```
template <typename DerivedPolicy,
  typename ForwardIterator,
  typename InputIterator,
  typename OutputIterator>
__host__ __device__ OutputIterator
thrust::lower_bound(const thrust::detail::execution_policy_base<
↪DerivedPolicy > & exec,
  ForwardIterator first,
  ForwardIterator last,
  InputIterator values_first,
  InputIterator values_last,
  OutputIterator result);

template <class ForwardIterator,
  class InputIterator,
  class OutputIterator>
OutputIterator
thrust::lower_bound(ForwardIterator first,
  ForwardIterator last,
  InputIterator values_first,
  InputIterator values_last,
  OutputIterator result);

template <typename DerivedPolicy,
  typename ForwardIterator,
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
  typename InputIterator,
  typename OutputIterator,
  typename StrictWeakOrdering>
__host__ __device__ OutputIterator
thrust::lower_bound(const thrust::detail::execution_policy_base<
 ↪DerivedPolicy > & exec,
  ForwardIterator first,
  ForwardIterator last,
  InputIterator values_first,
  InputIterator values_last,
  OutputIterator result,
  StrictWeakOrdering comp);

template <class ForwardIterator,
  class InputIterator,
  class OutputIterator,
  class StrictWeakOrdering>
OutputIterator
thrust::lower_bound(ForwardIterator first,
  ForwardIterator last,
  InputIterator values_first,
  InputIterator values_last,
  OutputIterator result,
  StrictWeakOrdering comp);

template <typename DerivedPolicy,
  typename ForwardIterator,
  typename InputIterator,
  typename OutputIterator>
__host__ __device__ OutputIterator
thrust::upper_bound(const thrust::detail::execution_policy_base<
 ↪DerivedPolicy > & exec,
  ForwardIterator first,
  ForwardIterator last,
  InputIterator values_first,
  InputIterator values_last,
  OutputIterator result);

template <class ForwardIterator,
  class InputIterator,
  class OutputIterator>
OutputIterator
thrust::upper_bound(ForwardIterator first,
  ForwardIterator last,
  InputIterator values_first,
  InputIterator values_last,
  OutputIterator result);

template <typename DerivedPolicy,
  typename ForwardIterator,
  typename InputIterator,
  typename OutputIterator,
  typename StrictWeakOrdering>
__host__ __device__ OutputIterator
thrust::upper_bound(const thrust::detail::execution_policy_base<
 ↪DerivedPolicy > & exec,
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
  ForwardIterator first,
  ForwardIterator last,
  InputIterator values_first,
  InputIterator values_last,
  OutputIterator result,
  StrictWeakOrdering comp);

template <class ForwardIterator,
  class InputIterator,
  class OutputIterator,
  class StrictWeakOrdering>
OutputIterator
thrust::upper_bound(ForwardIterator first,
  ForwardIterator last,
  InputIterator values_first,
  InputIterator values_last,
  OutputIterator result,
  StrictWeakOrdering comp);

template <typename DerivedPolicy,
  typename ForwardIterator,
  typename InputIterator,
  typename OutputIterator>
__host__ __device__ OutputIterator
thrust::binary_search(const thrust::detail::execution_policy_base<
 ↪DerivedPolicy > & exec,
  ForwardIterator first,
  ForwardIterator last,
  InputIterator values_first,
  InputIterator values_last,
  OutputIterator result);

template <class ForwardIterator,
  class InputIterator,
  class OutputIterator>
OutputIterator
thrust::binary_search(ForwardIterator first,
  ForwardIterator last,
  InputIterator values_first,
  InputIterator values_last,
  OutputIterator result);

template <typename DerivedPolicy,
  typename ForwardIterator,
  typename InputIterator,
  typename OutputIterator,
  typename StrictWeakOrdering>
 __host__ __device__ OutputIterator
 thrust::binary_search(const thrust::detail::execution_policy_base<
 ↪DerivedPolicy > & exec,
   ForwardIterator first,
   ForwardIterator last,
   InputIterator values_first,
   InputIterator values_last,
   OutputIterator result,
   StrictWeakOrdering comp);
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
template <class ForwardIterator,
  class InputIterator,
  class OutputIterator,
  class StrictWeakOrdering>
OutputIterator
thrust::binary_search(ForwardIterator first,
  ForwardIterator last,
  InputIterator values_first,
  InputIterator values_last,
  OutputIterator result,
  StrictWeakOrdering comp);
```

thrust::lower_bound

```
template <typename DerivedPolicy,
  typename ForwardIterator,
  typename InputIterator,
  typename OutputIterator>
__host__ __device__ OutputIterator
lower_bound(const thrust::detail::execution_policy_base< DerivedPolicy > &
↪exec,
  ForwardIterator first,
  ForwardIterator last,
  InputIterator values_first,
  InputIterator values_last,
  OutputIterator result);
```

lower_bound 是二分搜索的一个向量化版本：对于 [values_first, values_last) 范围内的每个迭代器 v，它试图在有序范围 [first, last) 内查找值 *v。具体而言，它返回在不打乱顺序的情况下插入值的第一个位置的索引。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::device 并行化执行策略，使用 lower_bound 在有序范围内搜索多个值：

```
#include <thrust/binary_search.h>
#include <thrust/device_vector.h>
#include <thrust/execution_policy.h>
...
thrust::device_vector<int> input(5);

input[0] = 0;
input[1] = 2;
input[2] = 5;
input[3] = 7;
input[4] = 8;

thrust::device_vector<int> values(6);
values[0] = 0;
values[1] = 1;
values[2] = 2;
values[3] = 3;
values[4] = 8;
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
values[5] = 9;

thrust::device_vector<unsigned int> output(6);

thrust::lower_bound(thrust::device,
input.begin(), input.end(),
values.begin(), values.end(),
output.begin());

// output is now [0, 1, 1, 2, 4, 5]
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **ForwardIterator**：是 Forward Iterator 的模型
- **InputIterator**：是 Input Iterator 的模型，InputIterator 的 value_type 为 LessThanComparable
- **OutputIterator**：是 Output Iterator 的模型，ForwardIterator 的 difference_type 可转换为 OutputIterator 的 value_type

函数参数

- **exec**：用于并行化的执行策略
- **first**：有序序列的起始
- **last**：有序序列的末尾
- **values_first**：搜索值序列的起始
- **values_last**：搜索值序列的末尾
- **result**：输出序列的起始

前提条件

范围 [first,last) 不应与范围 [result, result + (last - first)) 重叠

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/lower_bound
- upper_bound
- equal_range
- Binary Search

thrust::lower_bound

```
template <class ForwardIterator,
  class InputIterator,
  class OutputIterator>
OutputIterator
lower_bound(ForwardIterator first,
  ForwardIterator last,
  InputIterator values_first,
  InputIterator values_last,
  OutputIterator result);
```

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

lower_bound 是二分搜索的一个向量化版本：对于 [values_first, values_last) 范围内的每个迭代器 v，它试图在有序范围 [first, last) 内查找值 *v。具体而言，它返回在不打乱顺序的情况下插入值的第一个位置的索引。

以下代码片段演示了如何使用 lower_bound 在有序范围内搜索多个值：

```
#include <thrust/binary_search.h>
#include <thrust/device_vector.h>
...
thrust::device_vector<int> input(5);

input[0] = 0;
input[1] = 2;
input[2] = 5;
input[3] = 7;
input[4] = 8;

thrust::device_vector<int> values(6);
values[0] = 0;
values[1] = 1;
values[2] = 2;
values[3] = 3;
values[4] = 8;
values[5] = 9;

thrust::device_vector<unsigned int> output(6);

thrust::lower_bound(input.begin(), input.end(),
values.begin(), values.end(),
output.begin());

// output is now [0, 1, 1, 2, 4, 5]
```

模板参数

- **ForwardIterator**：是 Forward Iterator 的模型
- **InputIterator**：是 Input Iterator 的模型，InputIterator 的 value_type 为 LessThanComparable
- **OutputIterator**：是 Output Iterator 的模型，ForwardIterator 的 difference_type 可转换为 OutputIterator 的 value_type

函数参数

- **first**：有序序列的起始
- **last**：有序序列的末尾
- **values_first**：搜索值序列的起始
- **values_last**：搜索值序列的末尾
- **result**：输出序列的起始

前提条件

范围 [first,last) 不应与范围 [result, result + (last - first)) 重叠

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/lower_bound
- upper_bound
- equal_range
- Binary Search

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

thrust::lower_bound

```
template <typename DerivedPolicy,
  typename ForwardIterator,
  typename InputIterator,
 typename OutputIterator,
  typename StrictWeakOrdering>
__host__ __device__ OutputIterator
lower_bound(const thrust::detail::execution_policy_base< DerivedPolicy > &
↪exec,
  ForwardIterator first,
  ForwardIterator last,
  InputIterator values_first,
  InputIterator values_last,
  OutputIterator result,
  StrictWeakOrdering comp);
```

lower_bound 是二分搜索的一个向量化版本：对于 [values_first, values_last) 范围内的每个迭代器 v，它试图在有序范围 [first, last) 内查找值 *v。具体而言，它返回在不打乱顺序的情况下插入值的第一个位置的索引。此版本的 lower_bound 使用函数对象 comp 进行比较。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何使用 lower_bound 在有序范围内搜索多个值：

```
#include <thrust/binary_search.h>
#include <thrust/device_vector.h>
#include <thrust/functional.h>
#include <thrust/execution_policy.h>
...
thrust::device_vector<int> input(5);

input[0] = 0;
input[1] = 2;
input[2] = 5;
input[3] = 7;
input[4] = 8;

thrust::device_vector<int> values(6);
values[0] = 0;
values[1] = 1;
values[2] = 2;
values[3] = 3;
values[4] = 8;
values[5] = 9;

thrust::device_vector<unsigned int> output(6);

thrust::lower_bound(input.begin(), input.end(),
values.begin(), values.end(),
output.begin(),
thrust::less<int>());

// output is now [0, 1, 1, 2, 4, 5]
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **ForwardIterator**：是 Forward Iterator 的模型

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

- **InputIterator**：是 Input Iterator 的模型，InputIterator 的 value_type 类似 ForwardIterator 的 value_type
- **OutputIterator**：是 Output Iterator 的模型，ForwardIterator 的 difference_type 可转换为 OutputIterator 的 value_type
- **StrictWeakOrdering**：是 Strict Weak Ordering 的模型

函数参数

- **exec**：用于并行化的执行策略
- **first**：有序序列的起始
- **last**：有序序列的末尾
- **values_first**：搜索值序列的起始
- **values_last**：搜索值序列的末尾
- **result**：输出序列的起始
- **comp**：比较运算符

前提条件

范围 [first,last) 不应与范围 [result, result + (last - first)) 重叠

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/lower_bound
- upper_bound
- equal_range
- Binary Search

thrust::lower_bound

```
template <class ForwardIterator,
  class InputIterator,
  class OutputIterator,
  class StrictWeakOrdering>
OutputIterator
lower_bound(ForwardIterator first,
  ForwardIterator last,
  InputIterator values_first,
  InputIterator values_last,
  OutputIterator result,
  StrictWeakOrdering comp);
```

lower_bound 是二分搜索的一个向量化版本：对于 [values_first, values_last) 范围内的每个迭代器 v，它试图在有序范围 [first, last) 内查找值 *v。具体而言，它返回在不打乱顺序的情况下插入值的第一个位置的索引。此版本的 lower_bound 使用函数对象 comp 进行比较。

以下代码片段演示了如何使用 lower_bound 在有序范围内搜索多个值：

```
#include <thrust/binary_search.h>
#include <thrust/device_vector.h>
#include <thrust/functional.h>
...
thrust::device_vector<int> input(5);

input[0] = 0;
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
input[1] = 2;
input[2] = 5;
input[3] = 7;
input[4] = 8;

thrust::device_vector<int> values(6);
values[0] = 0;
values[1] = 1;
values[2] = 2;
values[3] = 3;
values[4] = 8;
values[5] = 9;

thrust::device_vector<unsigned int> output(6);

thrust::lower_bound(input.begin(), input.end(),
values.begin(), values.end(),
output.begin(),
thrust::less<int>());

// output is now [0, 1, 1, 2, 4, 5]
```

模板参数

- **ForwardIterator**：是 Forward Iterator 的模型
- **InputIterator**：是 Input Iterator 的模型，InputIterator 的 value_type 类似 ForwardIterator 的 value_type
- **OutputIterator**：是 Output Iterator 的模型，ForwardIterator 的 difference_type 可转换为 OutputIterator 的 value_type
- **StrictWeakOrdering**：是 Strict Weak Ordering 的模型

函数参数

- **first**：有序序列的起始
- **last**：有序序列的末尾
- **values_first**：搜索值序列的起始
- **values_last**：搜索值序列的末尾
- **result**：输出序列的起始
- **comp**：比较运算符

前提条件

范围 [first,last) 不应与范围 [result, result + (last - first)) 重叠

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/lower_bound
- upper_bound
- equal_range
- Binary Search

飞曦信息 MetaX Confidential 2024-11-18 15:00:00

thrust::upper_bound

```
template <typename DerivedPolicy,
  typename ForwardIterator,
  typename InputIterator,
  typename OutputIterator>
__host__ __device__ OutputIterator
upper_bound(const thrust::detail::execution_policy_base< DerivedPolicy > &
↪exec,
  ForwardIterator first,
  ForwardIterator last,
  InputIterator values_first,
  InputIterator values_last,
  OutputIterator result);
```

upper_bound 是二分搜索的一个向量化版本：对于 [values_first, values_last) 范围内的每个迭代器 v，它试图在有序范围 [first, last) 内查找值 *v。具体而言，它返回在不打乱顺序的情况下插入值的最后一个位置的索引。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::device 并行化执行策略，使用 upper_bound 在有序范围内搜索多个值：

```
#include <thrust/binary_search.h>
#include <thrust/device_vector.h>
#include <thrust/execution_policy.h>
...
thrust::device_vector<int> input(5);

input[0] = 0;
input[1] = 2;
input[2] = 5;
input[3] = 7;
input[4] = 8;

thrust::device_vector<int> values(6);
values[0] = 0;
values[1] = 1;
values[2] = 2;
values[3] = 3;
values[4] = 8;
values[5] = 9;

thrust::device_vector<unsigned int> output(6);

thrust::upper_bound(thrust::device,
input.begin(), input.end(),
values.begin(), values.end(),
output.begin());

// output is now [1, 1, 2, 2, 5, 5]
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **ForwardIterator**：是 Forward Iterator 的模型
- **InputIterator**：是 Input Iterator 的模型，InputIterator 的 value_type 为 LessThanComparable

飞谱信息 Metax Confidential 2024-11-18 15:00:00

- **OutputIterator**：是 Output Iterator 的模型，ForwardIterator 的 difference_type 可转换为 OutputIterator 的 value_type

函数参数

- **exec**：用于并行化的执行策略
- **first**：有序序列的起始
- **last**：有序序列的末尾
- **values_first**：搜索值序列的起始
- **values_last**：搜索值序列的末尾
- **result**：输出序列的起始

前提条件

范围 [first,last) 不应与范围 [result, result + (last - first)) 重叠

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/upper_bound
- upper_bound
- equal_range
- Binary Search

thrust::upper_bound

```
template <class ForwardIterator,
  class InputIterator,
  class OutputIterator>
OutputIterator
upper_bound(ForwardIterator first,
  ForwardIterator last,
  InputIterator values_first,
  InputIterator values_last,
  OutputIterator result);
```

upper_bound 是二分搜索的一个向量化版本：对于 [values_first, values_last) 范围内的每个迭代器 v，它试图在有序范围 [first, last) 内查找值 *v。具体而言，它返回在不打乱顺序的情况下插入值的最后一个位置的索引。

以下代码片段演示了如何使用 upper_bound 在有序范围内搜索多个值：

```
#include <thrust/binary_search.h>
#include <thrust/device_vector.h>
...
thrust::device_vector<int> input(5);

input[0] = 0;
input[1] = 2;
input[2] = 5;
input[3] = 7;
input[4] = 8;

thrust::device_vector<int> values(6);
values[0] = 0;
values[1] = 1;
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
values[2] = 2;
values[3] = 3;
values[4] = 8;
values[5] = 9;

thrust::device_vector<unsigned int> output(6);

thrust::upper_bound(input.begin(), input.end(),
values.begin(), values.end(),
output.begin());

// output is now [1, 1, 2, 2, 5, 5]
```

模板参数

- **ForwardIterator**：是 Forward Iterator 的模型
- **InputIterator**：是 Input Iterator 的模型，InputIterator 的 value_type 为 LessThanComparable
- **OutputIterator**：是 Output Iterator 的模型，ForwardIterator 的 difference_type 可转换为 OutputIterator 的 value_type

函数参数

- **first**：有序序列的起始
- **last**：有序序列的末尾
- **values_first**：搜索值序列的起始
- **values_last**：搜索值序列的末尾
- **result**：输出序列的起始

前提条件

范围 [first,last) 不应与范围 [result, result + (last - first)) 重叠

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/upper_bound
- upper_bound
- equal_range
- Binary Search

thrust::upper_bound

```
template <typename DerivedPolicy,
  typename ForwardIterator,
  typename InputIterator,
  typename OutputIterator,
  typename StrictWeakOrdering>
__host__ __device__ OutputIterator
upper_bound(const thrust::detail::execution_policy_base< DerivedPolicy > &
↪exec,
  ForwardIterator first,
  ForwardIterator last,
  InputIterator values_first,
  InputIterator values_last,
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
  OutputIterator result,
  StrictWeakOrdering comp);
```

upper_bound 是二分搜索的一个向量化版本：对于 [values_first, values_last) 范围内的每个迭代器 v，它试图在有序范围 [first, last) 内查找值 *v。具体而言，它返回在不打乱顺序的情况下插入值的第一个位置的索引。此版本的 upper_bound 使用函数对象 comp 进行比较。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::device 并行化执行策略，使用 upper_bound 在有序范围内搜索多个值：

```
#include <thrust/binary_search.h>
#include <thrust/device_vector.h>
#include <thrust/functional.h>
#include <thrust/execution_policy.h>
...
thrust::device_vector<int> input(5);

input[0] = 0;
input[1] = 2;
input[2] = 5;
input[3] = 7;
input[4] = 8;

thrust::device_vector<int> values(6);
values[0] = 0;
values[1] = 1;
values[2] = 2;
values[3] = 3;
values[4] = 8;
values[5] = 9;

thrust::device_vector<unsigned int> output(6);

thrust::upper_bound(thrust::device,
input.begin(), input.end(),
values.begin(), values.end(),
output.begin(),
thrust::less<int>());

// output is now [1, 1, 2, 2, 5, 5]
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **ForwardIterator**：是 Forward Iterator 的模型
- **InputIterator**：是 Input Iterator 的模型，InputIterator 的 value_type 类似 ForwardIterator 的 value_type
- **OutputIterator**：是 Output Iterator 的模型，ForwardIterator 的 difference_type 可转换为 OutputIterator 的 value_type
- **StrictWeakOrdering**：是 Strict Weak Ordering 的模型

函数参数

- **exec**：用于并行化的执行策略
- **first**：有序序列的起始

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

- **last**：有序序列的末尾
- **values_first**：搜索值序列的起始
- **values_last**：搜索值序列的末尾
- **result**：输出序列的起始
- **comp**：比较运算符

前提条件

范围 [first,last) 不应与范围 [result, result + (last - first)) 重叠

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/upper_bound
- lower_bound
- equal_range
- Binary Search

thrust::upper_bound

```
template <class ForwardIterator,
  class InputIterator,
  class OutputIterator,
  class StrictWeakOrdering>
OutputIterator
upper_bound(ForwardIterator first,
  ForwardIterator last,
  InputIterator values_first,
  InputIterator values_last,
  OutputIterator result,
  StrictWeakOrdering comp);
```

upper_bound 是二分搜索的一个向量化版本：对于 [values_first, values_last) 范围内的每个迭代器 v，它试图在有序范围 [first, last) 内查找值 *v。具体而言，它返回在不打乱顺序的情况下插入值的第一个位置的索引。

此版本的 upper_bound 使用函数对象 comp 进行比较。

以下代码片段演示了如何使用 upper_bound 在有序范围内搜索多个值：

```
#include <thrust/binary_search.h>
#include <thrust/device_vector.h>
#include <thrust/functional.h>
...
thrust::device_vector<int> input(5);

input[0] = 0;
input[1] = 2;
input[2] = 5;
input[3] = 7;
input[4] = 8;

thrust::device_vector<int> values(6);
values[0] = 0;
values[1] = 1;
values[2] = 2;
```

(下页继续)

沐曦信息 MetaX Confidential 2024-11-8 15:00:00

(续上页)

```
values[3] = 3;
values[4] = 8;
values[5] = 9;

thrust::device_vector<unsigned int> output(6);

thrust::upper_bound(input.begin(), input.end(),
values.begin(), values.end(),
output.begin(),
thrust::less<int>());

// output is now [1, 1, 2, 2, 5, 5]
```

模板参数

- **ForwardIterator**：是 Forward Iterator 的模型
- **InputIterator**：是 Input Iterator 的模型，InputIterator 的 value_type 类似 ForwardIterator 的 value_type
- **OutputIterator**：是 Output Iterator 的模型，ForwardIterator 的 difference_type 可转换为 OutputIterator 的 value_type
- **StrictWeakOrdering**：是 Strict Weak Ordering 的模型

函数参数

- **first**：有序序列的起始
- **last**：有序序列的末尾
- **values_first**：搜索值序列的起始
- **values_last**：搜索值序列的末尾
- **result**：输出序列的起始
- **comp**：比较运算符

前提条件

范围 [first,last) 不应与范围 [result, result + (last - first)) 重叠

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/upper_bound
- lower_bound
- equal_range
- Binary Search

thrust::binary_search

```
template <typename DerivedPolicy,
  typename ForwardIterator,
  typename InputIterator,
  typename OutputIterator>
__host__ __device__ OutputIterator
binary_search(const thrust::detail::execution_policy_base< DerivedPolicy >
↪& exec,
  ForwardIterator first,
```

(下页继续)

飞曦信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
  ForwardIterator last,
  InputIterator values_first,
  InputIterator values_last,
  OutputIterator result);
```

binary_search 是二分搜索的一个向量化版本：对于 [values_first, values_last) 范围内的每个迭代器 v，它试图在有序范围 [first, last) 内查找值 *v。如果在 [first, last) 中存在与 value 等价的元素，则返回 true；如果不存在这样的元素，那么返回 false。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::device 并行化执行策略，使用 binary_search 在有序范围内搜索多个值：

```
#include <thrust/binary_search.h>
#include <thrust/device_vector.h>
#include <thrust/execution_policy.h>
...
thrust::device_vector<int> input(5);

input[0] = 0;
input[1] = 2;
input[2] = 5;
input[3] = 7;
input[4] = 8;

thrust::device_vector<int> values(6);
values[0] = 0;
values[1] = 1;
values[2] = 2;
values[3] = 3;
values[4] = 8;
values[5] = 9;

thrust::device_vector<bool> output(6);

thrust::binary_search(thrust::device,
input.begin(), input.end(),
values.begin(), values.end(),
output.begin());

// output is now [true, false, true, false, true, false]
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **ForwardIterator**：是 Forward Iterator 的模型
- **InputIterator**：是 Input Iterator 的模型，InputIterator 的 value_type 为 LessThanComparable
- **OutputIterator**：是 Output Iterator 的模型，bool 可转换为 OutputIterator 的 value_type

函数参数

- **exec**：用于并行化的执行策略
- **first**：有序序列的起始
- **last**：有序序列的末尾
- **values_first**：搜索值序列的起始

飞猪信息 MetaX Confidential 2024-11-18 15:00:00

- **values_last**：搜索值序列的末尾
- **result**：输出序列的起始

前提条件

范围 [first,last) 不应与范围 [result, result + (last - first)) 重叠

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/binary_search
- lower_bound
- upper_bound
- equal_range

thrust::binary_search

```
template <class ForwardIterator,
  class InputIterator,
  class OutputIterator>
OutputIterator
binary_search(ForwardIterator first,
  ForwardIterator last,
  InputIterator values_first,
  InputIterator values_last,
  OutputIterator result);
```

binary_search 是二分搜索的一个向量化版本：对于 [values_first, values_last) 范围内的每个迭代器 v，它试图在有序范围 [first, last) 内查找值 *v。如果在 [first, last) 中存在与 value 等价的元素，则返回 true；如果不存在这样的元素，那么返回 false。

以下代码片段演示了如何使用 binary_search 在有序范围内搜索多个值：

```
#include <thrust/binary_search.h>
#include <thrust/device_vector.h>
...
thrust::device_vector<int> input(5);

input[0] = 0;
input[1] = 2;
input[2] = 5;
input[3] = 7;
input[4] = 8;

thrust::device_vector<int> values(6);
values[0] = 0;
values[1] = 1;
values[2] = 2;
values[3] = 3;
values[4] = 8;
values[5] = 9;

thrust::device_vector<bool> output(6);

thrust::binary_search(input.begin(), input.end(),
values.begin(), values.end(),
output.begin());
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
// output is now [true, false, true, false, true, false]
```

模板参数

- **ForwardIterator**：是 Forward Iterator 的模型
- **InputIterator**：是 Input Iterator 的模型，InputIterator 的 value_type 为 LessThanComparable
- **OutputIterator**：是 Output Iterator 的模型，bool 可转换为 OutputIterator 的 value_type

函数参数

- **first**：有序序列的起始
- **last**：有序序列的末尾
- **values_first**：搜索值序列的起始
- **values_last**：搜索值序列的末尾
- **result**：输出序列的起始

前提条件

范围 [first,last) 不应与范围 [result, result + (last - first)) 重叠

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/binary_search
- lower_bound
- upper_bound
- equal_range

thrust::binary_search

```
template <typename DerivedPolicy,
  typename ForwardIterator,
  typename InputIterator,
  typename OutputIterator,
  typename StrictWeakOrdering>
__host__ __device__ OutputIterator binary_search(const
↪thrust::detail::execution_policy_base< DerivedPolicy > & exec,
  ForwardIterator first,
  ForwardIterator last,
  InputIterator values_first,
  InputIterator values_last,
  OutputIterator result,
  StrictWeakOrdering comp);
```

binary_search 是二分搜索的一个向量化版本：对于 [values_first, values_last) 范围内的每个迭代器 v，它试图在有序范围 [first, last) 内查找值 *v。如果在 [first, last) 中存在与 value 等价的元素，则返回 true；如果不存在这样的元素，那么返回 false。此版本的 binary_search 使用函数对象 comp 进行比较。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::device 并行化执行策略，使用 binary_search 在有序范围内搜索多个值：

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

```
#include <thrust/binary_search.h>
#include <thrust/device_vector.h>
#include <thrust/functional.h>
#include <thrust/execution_policy.h>
...
thrust::device_vector<int> input(5);

input[0] = 0;
input[1] = 2;
input[2] = 5;
input[3] = 7;
input[4] = 8;

thrust::device_vector<int> values(6);
values[0] = 0;
values[1] = 1;
values[2] = 2;
values[3] = 3;
values[4] = 8;
values[5] = 9;

thrust::device_vector<bool> output(6);

thrust::binary_search(thrust::device,
input.begin(), input.end(),
values.begin(), values.end(),
output.begin(),
thrust::less<T>());

// output is now [true, false, true, false, true, false]
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **ForwardIterator**：是 Forward Iterator 的模型
- **InputIterator**：是 Input Iterator 的模型，InputIterator 的 value_type 为 LessThanComparable
- **OutputIterator**：是 Output Iterator 的模型，bool 可转换为 OutputIterator 的 value_type
- **StrictWeakOrdering**：是 Strict Weak Ordering 的模型

函数参数

- **exec**：用于并行化的执行策略
- **first**：有序序列的起始
- **last**：有序序列的末尾
- **values_first**：搜索值序列的起始
- **values_last**：搜索值序列的末尾
- **result**：输出序列的起始
- **comp**：比较运算符

前提条件

范围 [first,last) 不应与范围 [result, result + (last - first)) 重叠

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/binary_search

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

- lower_bound
- upper_bound
- equal_range

thrust::binary_search

```
template <class ForwardIterator,
  class InputIterator,
  class OutputIterator,
  class StrictWeakOrdering>
OutputIterator
binary_search(ForwardIterator first,
  ForwardIterator last,
  InputIterator values_first,
  InputIterator values_last,
  OutputIterator result,
  StrictWeakOrdering comp);
```

binary_search 是二分搜索的一个向量化版本：对于 [values_first, values_last) 范围内的每个迭代器 v，它试图在有序范围 [first, last) 内查找值 *v。如果在 [first, last) 中存在与 value 等价的元素，则返回 true；如果不存在这样的元素，那么返回 false。此版本的 binary_search 使用函数对象 comp 进行比较。

以下代码片段演示了如何使用 binary_search 在有序范围内搜索多个值：

```
#include <thrust/binary_search.h>
#include <thrust/device_vector.h>
#include <thrust/functional.h>
...
thrust::device_vector<int> input(5);

input[0] = 0;
input[1] = 2;
input[2] = 5;
input[3] = 7;
input[4] = 8;

thrust::device_vector<int> values(6);
values[0] = 0;
values[1] = 1;
values[2] = 2;
values[3] = 3;
values[4] = 8;
values[5] = 9;

thrust::device_vector<bool> output(6);

thrust::binary_search(input.begin(), input.end(),
values.begin(), values.end(),
output.begin(),
thrust::less<T>());

// output is now [true, false, true, false, true, false]
```

模板参数

- **ForwardIterator**：是 Forward Iterator 的模型

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

- **InputIterator**：是 Input Iterator 的模型，InputIterator 的 value_type 为 LessThanComparable
- **OutputIterator**：是 Output Iterator 的模型，bool 可转换为 OutputIterator 的 value_type
- **StrictWeakOrdering**：是 Strict Weak Ordering 的模型

函数参数

- **first**：有序序列的起始
- **last**：有序序列的末尾
- **values_first**：搜索值序列的起始
- **values_last**：搜索值序列的末尾
- **result**：输出序列的起始
- **comp**：比较运算符

前提条件

范围 [first,last) 不应与范围 [result, result + (last - first)) 重叠

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/binary_search
- lower_bound
- upper_bound
- equal_range

2.1.7 集合运算（Set Operation）

函数

```
template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator>
__host__ __device__ OutputIterator
thrust::set_difference(const thrust::detail::execution_policy_base<
↪DerivedPolicy > & exec,
  InputIterator1 first1,
  InputIterator1 last1,
  InputIterator2 first2,
  InputIterator2 last2,
  OutputIterator result);

template <typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator>
OutputIterator
thrust::set_difference(InputIterator1 first1,
  InputIterator1 last1,
  InputIterator2 first2,
  InputIterator2 last2,
  OutputIterator result);

template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2,
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
  typename OutputIterator,
  typename StrictWeakCompare>
__host__ __device__ OutputIterator
thrust::set_difference(const thrust::detail::execution_policy_base<
 ↪DerivedPolicy > & exec,
  InputIterator1 first1,
  InputIterator1 last1,
  InputIterator2 first2,
  InputIterator2 last2,
  OutputIterator result,
  StrictWeakCompare comp);

template <typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator,
  typename StrictWeakCompare>
OutputIterator
thrust::set_difference(InputIterator1 first1,
  InputIterator1 last1,
  InputIterator2 first2,
  InputIterator2 last2,
  OutputIterator result,
  StrictWeakCompare comp);

template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator>
__host__ __device__ OutputIterator
thrust::set_intersection(const thrust::detail::execution_policy_base<
 ↪DerivedPolicy > & exec,
  InputIterator1 first1,
  InputIterator1 last1,
  InputIterator2 first2,
  InputIterator2 last2,
  OutputIterator result);

template <typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator>
OutputIterator
thrust::set_intersection(InputIterator1 first1,
  InputIterator1 last1,
  InputIterator2 first2,
  InputIterator2 last2,
  OutputIterator result);

template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator,
  typename StrictWeakCompare>
__host__ __device__ OutputIterator
thrust::set_intersection(const thrust::detail::execution_policy_base<
 ↪DerivedPolicy > & exec,
  InputIterator1 first1,
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
  InputIterator1 last1,
  InputIterator2 first2,
  InputIterator2 last2,
  OutputIterator result,
  StrictWeakCompare comp);

template <typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator,
  typename StrictWeakCompare>
OutputIterator
thrust::set_intersection(InputIterator1 first1,
  InputIterator1 last1,
  InputIterator2 first2,
  InputIterator2 last2,
  OutputIterator result,
  StrictWeakCompare comp);

template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator>
__host__ __device__ OutputIterator
thrust::set_symmetric_difference(const thrust::detail::execution_policy_
 ↪base< DerivedPolicy > & exec,
  InputIterator1 first1,
  InputIterator1 last1,
  InputIterator2 first2,
  InputIterator2 last2,
  OutputIterator result);

template <typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator>
OutputIterator
thrust::set_symmetric_difference(InputIterator1 first1,
  InputIterator1 last1,
  InputIterator2 first2,
  InputIterator2 last2,
  OutputIterator result);

template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator,
  typename StrictWeakCompare>
__host__ __device__ OutputIterator
thrust::set_symmetric_difference(const thrust::detail::execution_policy_
 ↪base< DerivedPolicy > & exec,
  InputIterator1 first1,
  InputIterator1 last1,
  InputIterator2 first2,
  InputIterator2 last2,
  OutputIterator result,
  StrictWeakCompare comp);
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
template <typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator,
  typename StrictWeakCompare>
OutputIterator
thrust::set_symmetric_difference(InputIterator1 first1,
  InputIterator1 last1,
  InputIterator2 first2,
  InputIterator2 last2,
  OutputIterator result,
  StrictWeakCompare comp);

template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator>
__host__ __device__ OutputIterator
thrust::set_union(const thrust::detail::execution_policy_base<
 ↪DerivedPolicy > & exec,
  InputIterator1 first1,
  InputIterator1 last1,
  InputIterator2 first2,
  InputIterator2 last2,
  OutputIterator result);

template <typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator>
OutputIterator
thrust::set_union(InputIterator1 first1,
  InputIterator1 last1,
  InputIterator2 first2,
  InputIterator2 last2,
  OutputIterator result);

template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator,
  typename StrictWeakCompare>
__host__ __device__ OutputIterator
thrust::set_union(const thrust::detail::execution_policy_base<
 ↪DerivedPolicy > & exec,
  InputIterator1 first1,
  InputIterator1 last1,
  InputIterator2 first2,
  InputIterator2 last2,
  OutputIterator result,
  StrictWeakCompare comp);

template <typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator,
  typename StrictWeakCompare>
OutputIterator
thrust::set_union(InputIterator1 first1,
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
  InputIterator1 last1,
  InputIterator2 first2,
  InputIterator2 last2,
  OutputIterator result,
  StrictWeakCompare comp);

template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2,
  typename InputIterator3,
  typename InputIterator4,
  typename OutputIterator1,
  typename OutputIterator2>
__host__ __device__ thrust::pair< OutputIterator1, OutputIterator2 >
thrust::set_difference_by_key(const thrust::detail::execution_policy_base<
 ↪DerivedPolicy > & exec,
  InputIterator1 keys_first1,
  InputIterator1 keys_last1,
  InputIterator2 keys_first2,
  InputIterator2 keys_last2,
  InputIterator3 values_first1,
  InputIterator4 values_first2,
  OutputIterator1 keys_result,
  OutputIterator2 values_result);

template <typename InputIterator1,
  typename InputIterator2,
  typename InputIterator3,
  typename InputIterator4,
  typename OutputIterator1,
  typename OutputIterator2>
thrust::pair< OutputIterator1, OutputIterator2 >
thrust::set_difference_by_key(InputIterator1 keys_first1,
  InputIterator1 keys_last1,
  InputIterator2 keys_first2,
  InputIterator2 keys_last2,
  InputIterator3 values_first1,
  InputIterator4 values_first2,
  OutputIterator1 keys_result,
  OutputIterator2 values_result);

template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2,
  typename InputIterator3,
  typename InputIterator4,
  typename OutputIterator1,
  typename OutputIterator2,
  typename StrictWeakCompare>
__host__ __device__ thrust::pair< OutputIterator1, OutputIterator2 >
thrust::set_difference_by_key(const thrust::detail::execution_policy_base<
 ↪DerivedPolicy > & exec,
  InputIterator1 keys_first1,
  InputIterator1 keys_last1,
  InputIterator2 keys_first2,
  InputIterator2 keys_last2,
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
  InputIterator3 values_first1,
  InputIterator4 values_first2,
  OutputIterator1 keys_result,
  OutputIterator2 values_result,
  StrictWeakCompare comp);

template <typename InputIterator1,
  typename InputIterator2,
  typename InputIterator3,
  typename InputIterator4,
  typename OutputIterator1,
  typename OutputIterator2,
  typename StrictWeakCompare>
thrust::pair< OutputIterator1, OutputIterator2 >
thrust::set_difference_by_key(InputIterator1 keys_first1,
  InputIterator1 keys_last1,
  InputIterator2 keys_first2,
  InputIterator2 keys_last2,
  InputIterator3 values_first1,
  InputIterator4 values_first2,
  OutputIterator1 keys_result,
  OutputIterator2 values_result,
  StrictWeakCompare comp);

template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2,
  typename InputIterator3,
  typename OutputIterator1,
  typename OutputIterator2>
__host__ __device__ thrust::pair< OutputIterator1, OutputIterator2 >
thrust::set_intersection_by_key(const thrust::detail::execution_policy_base
↪< DerivedPolicy > & exec,
  InputIterator1 keys_first1,
  InputIterator1 keys_last1,
  InputIterator2 keys_first2,
  InputIterator2 keys_last2,
  InputIterator3 values_first1,
  OutputIterator1 keys_result,
  OutputIterator2 values_result);

template <typename InputIterator1,
  typename InputIterator2,
  typename InputIterator3,
  typename OutputIterator1,
  typename OutputIterator2>
thrust::pair< OutputIterator1, OutputIterator2 >
thrust::set_intersection_by_key(InputIterator1 keys_first1,
  InputIterator1 keys_last1,
  InputIterator2 keys_first2,
  InputIterator2 keys_last2,
  InputIterator3 values_first1,
  OutputIterator1 keys_result,
  OutputIterator2 values_result);

template <typename DerivedPolicy,
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
  typename InputIterator1,
  typename InputIterator2,
  typename InputIterator3,
  typename OutputIterator1,
  typename OutputIterator2,
  typename StrictWeakCompare>
__host__ __device__ thrust::pair< OutputIterator1, OutputIterator2 >
thrust::set_intersection_by_key(const thrust::detail::execution_policy_base
↪< DerivedPolicy > & exec,
  InputIterator1 keys_first1,
  InputIterator1 keys_last1,
  InputIterator2 keys_first2,
  InputIterator2 keys_last2,
  InputIterator3 values_first1,
  OutputIterator1 keys_result,
  OutputIterator2 values_result,
  StrictWeakCompare comp);

template <typename InputIterator1,
  typename InputIterator2,
  typename InputIterator3,
  typename OutputIterator1,
  typename OutputIterator2,
  typename StrictWeakCompare>
thrust::pair< OutputIterator1, OutputIterator2 >
thrust::set_intersection_by_key(InputIterator1 keys_first1,
  InputIterator1 keys_last1,
  InputIterator2 keys_first2,
  InputIterator2 keys_last2,
  InputIterator3 values_first1,
  OutputIterator1 keys_result,
  OutputIterator2 values_result,
  StrictWeakCompare comp);

template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2,
  typename InputIterator3,
  typename InputIterator4,
  typename OutputIterator1,
  typename OutputIterator2>
__host__ __device__ thrust::pair< OutputIterator1, OutputIterator2 >
thrust::set_symmetric_difference_by_key(const thrust::detail::execution_
↪policy_base< DerivedPolicy > & exec,
  InputIterator1 keys_first1,
  InputIterator1 keys_last1,
  InputIterator2 keys_first2,
  InputIterator2 keys_last2,
  InputIterator3 values_first1,
  InputIterator4 values_first2,
  OutputIterator1 keys_result,
  OutputIterator2 values_result);

template <typename InputIterator1,
  typename InputIterator2,
  typename InputIterator3,
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
  typename InputIterator4,
  typename OutputIterator1,
  typename OutputIterator2>
thrust::pair< OutputIterator1, OutputIterator2 >
thrust::set_symmetric_difference_by_key(InputIterator1 keys_first1,
  InputIterator1 keys_last1,
  InputIterator2 keys_first2,
  InputIterator2 keys_last2,
  InputIterator3 values_first1,
  InputIterator4 values_first2,
  OutputIterator1 keys_result,
  OutputIterator2 values_result);

template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2,
  typename InputIterator3,
  typename InputIterator4,
  typename OutputIterator1,
  typename OutputIterator2,
  typename StrictWeakCompare>
__host__ __device__ thrust::pair< OutputIterator1, OutputIterator2 >
thrust::set_symmetric_difference_by_key(const thrust::detail::execution_
↪policy_base< DerivedPolicy > & exec,
  InputIterator1 keys_first1,
  InputIterator1 keys_last1,
  InputIterator2 keys_first2,
  InputIterator2 keys_last2,
  InputIterator3 values_first1,
  InputIterator4 values_first2,
  OutputIterator1 keys_result,
  OutputIterator2 values_result,
  StrictWeakCompare comp);

template <typename InputIterator1,
  typename InputIterator2,
  typename InputIterator3,
  typename InputIterator4,
  typename OutputIterator1,
  typename OutputIterator2,
  typename StrictWeakCompare>
thrust::pair< OutputIterator1, OutputIterator2 >
thrust::set_symmetric_difference_by_key(InputIterator1 keys_first1,
  InputIterator1 keys_last1,
  InputIterator2 keys_first2,
  InputIterator2 keys_last2,
  InputIterator3 values_first1,
  InputIterator4 values_first2,
  OutputIterator1 keys_result,
  OutputIterator2 values_result,
  StrictWeakCompare comp);

template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2,
  typename InputIterator3,
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
  typename InputIterator4,
  typename OutputIterator1,
  typename OutputIterator2>
__host__ __device__ thrust::pair< OutputIterator1, OutputIterator2 >
thrust::set_union_by_key(const thrust::detail::execution_policy_base<
↪DerivedPolicy > & exec,
  InputIterator1 keys_first1,
  InputIterator1 keys_last1,
  InputIterator2 keys_first2,
  InputIterator2 keys_last2,
  InputIterator3 values_first1,
  InputIterator4 values_first2,
  OutputIterator1 keys_result,
  OutputIterator2 values_result);

template <typename InputIterator1,
  typename InputIterator2,
  typename InputIterator3,
  typename InputIterator4,
  typename OutputIterator1,
  typename OutputIterator2>
thrust::pair< OutputIterator1, OutputIterator2 >
thrust::set_union_by_key(InputIterator1 keys_first1,
  InputIterator1 keys_last1,
  InputIterator2 keys_first2,
  InputIterator2 keys_last2,
  InputIterator3 values_first1,
  InputIterator4 values_first2,
  OutputIterator1 keys_result,
  OutputIterator2 values_result);

template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2,
  typename InputIterator3,
  typename InputIterator4,
  typename OutputIterator1,
  typename OutputIterator2,
  typename StrictWeakCompare>
__host__ __device__ thrust::pair< OutputIterator1, OutputIterator2 >
thrust::set_union_by_key(const thrust::detail::execution_policy_base<
↪DerivedPolicy > & exec,
  InputIterator1 keys_first1,
  InputIterator1 keys_last1,
  InputIterator2 keys_first2,
  InputIterator2 keys_last2,
  InputIterator3 values_first1,
  InputIterator4 values_first2,
  OutputIterator1 keys_result,
  OutputIterator2 values_result,
  StrictWeakCompare comp);

template <typename InputIterator1,
  typename InputIterator2,
  typename InputIterator3,
  typename InputIterator4,
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
  typename OutputIterator1,
  typename OutputIterator2,
  typename StrictWeakCompare>
thrust::pair< OutputIterator1, OutputIterator2 >
thrust::set_union_by_key(InputIterator1 keys_first1,
  InputIterator1 keys_last1,
  InputIterator2 keys_first2,
  InputIterator2 keys_last2,
  InputIterator3 values_first1,
  InputIterator4 values_first2,
  OutputIterator1 keys_result,
  OutputIterator2 values_result,
  StrictWeakCompare comp);
```

2.1.7.1 thrust::set_difference

```
template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator>
__host__ __device__ OutputIterator
set_difference(const thrust::detail::execution_policy_base< DerivedPolicy >
↪ & exec,
  InputIterator1 first1,
  InputIterator1 last1,
  InputIterator2 first2,
  InputIterator2 last2,
  OutputIterator result);
```

set_difference 构造一个有序范围，是有序范围 [first1, last1) 和 [first2, last2) 的差集。返回值是输出范围的末尾。

在最简单的情况下，set_difference 执行集合的差运算：输出范围由所有属于 [first1, last1) 而不属于 [first2, last1) 的元素副本组成。一般情况更为复杂，因为输入范围内可能包含重复元素。总而言之，如果 [first1, last1) 包含 m 个彼此等价的元素，且 [first2, last2) 包含 n 个与它们等价的元素，则 [first1, last1) 内最后 max(m-n,0) 个元素会被复制到输出范围。

此版本的 set_difference 使用 operator< 比较元素。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::host 并行化执行策略，使用 set_difference 计算两组升序整数集合的差集：

```
#include <thrust/set_operations.h>
#include <thrust/execution_policy.h>
...
int A1[7] = {0, 1, 3, 4, 5, 6, 9};
int A2[5] = {1, 3, 5, 7, 9};

int result[3];

int *result_end = thrust::set_difference(thrust::host, A1, A1 + 7, A2, A2
↪+ 5, result);
// result is now {0, 4, 6}
```

模板参数

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

- **DerivedPolicy**：派生执行策略的名称
- **InputIterator1**：是 Input Iterator 的模型。InputIterator1 和 InputIterator2 具有相同的 value_type，InputIterator1 的 value_type 是 LessThan Comparable 的模型，如 LessThan Comparable 要求中定义的，InputIterator1 的 value_type 上的排序是严格弱序，并且可转换为 OutputIterator 的 value_types 集合中的类型
- **InputIterator2**：是 Input Iterator 的模型。InputIterator2 和 InputIterator1 具有相同的 value_type，InputIterator2 的 value_type 是 LessThan Comparable 的模型，如 LessThan Comparable 要求中定义的，InputIterator2 的 value_type 上的排序是严格弱序，并且可转换为 OutputIterator 的 value_types 集合中的类型
- **OutputIterator**：是 Output Iterator 的模型

函数参数

- **exec**：用于并行化的执行策略
- **first1**：第一个输入范围的起始
- **last1**-：第一个输入范围的末尾
- **first2**：第二个输入范围的起始
- **last2**：第二个输入范围的末尾
- **result**：输出范围的起始

前提条件

- 范围 [first1, last1) 和 [first2, last2) 应按照 operator< 进行排序
- 结果范围不应与任何一个输入范围重叠

返回值

输出范围的末尾

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/set_difference
- includes
- set_union
- set_intersection
- set_symmetric_difference
- sort
- is_sorted

2.1.7.2 thrust::set_difference

```
template <typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator>
OutputIterator
set_difference(InputIterator1 first1,
  InputIterator1 last1,
  InputIterator2 first2,
  InputIterator2 last2,
  OutputIterator result);
```

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

set_difference 构造一个有序范围，是有序范围 [first1, last1) 和 [first2, last2) 的差集。返回值是输出范围的末尾。

在最简单的情况下，set_difference 执行集合的差运算：输出范围由所有属于 [first1, last1) 而不属于 [first2, last1) 的元素副本组成。一般情况更为复杂，因为输入范围内可能包含重复元素。总而言之，如果 [first1, last1) 包含 m 个彼此等价的元素，且 [first2, last2) 包含 n 个与它们等价的元素，则 [first1, last1) 内最后 max(m-n,0) 个元素会被复制到输出范围。

此版本的 set_difference 使用 operator< 比较元素。

以下代码片段演示了如何使用 set_difference 计算两组升序排序整数集合的差集：

```
#include <thrust/set_operations.h>
...
int A1[7] = {0, 1, 3, 4, 5, 6, 9};
int A2[5] = {1, 3, 5, 7, 9};

int result[3];

int *result_end = thrust::set_difference(A1, A1 + 7, A2, A2 + 5, result);
// result is now {0, 4, 6}
```

模板参数

- **InputIterator1**：是 Input Iterator 的模型。InputIterator1 和 InputIterator2 具有相同的 value_type，InputIterator1 的 value_type 是 LessThan Comparable 的模型，如 LessThan Comparable 要求中定义的，InputIterator1 的 value_type 上的排序是严格弱序，并且可转换为 OutputIterator 的 value_types 集合中的类型
- **InputIterator2**：是 Input Iterator 的模型。InputIterator2 和 InputIterator1 具有相同的 value_type，InputIterator2 的 value_type 是 LessThan Comparable 的模型，如 LessThan Comparable 要求中定义的，InputIterator2 的 value_type 上的排序是严格弱序，并且可转换为 OutputIterator 的 value_types 集合中的类型
- **OutputIterator**：是 Output Iterator 的模型

函数参数

- **first1**：第一个输入范围的起始
- **last1**-：第一个输入范围的末尾
- **first2**：第二个输入范围的起始
- **last2**：第二个输入范围的末尾
- **result**：输出范围的起始

前提条件

- 范围 [first1, last1) 和 [first2, last2) 应按照 operator< 进行排序
- 结果范围不应与任何一个输入范围重叠

返回值

输出范围的末尾

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/set_difference
- includes
- set_union
- set_intersection
- set_symmetric_difference

飞腾信息 MetaX Confident 2024-11-18 15:00:00

- sort
- is_sorted

2.1.7.3 thrust::set_difference

```
template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator,
  typename StrictWeakCompare>
__host__ __device__ OutputIterator
set_difference(const thrust::detail::execution_policy_base< DerivedPolicy >
 ↪ & exec,
  InputIterator1 first1,
  InputIterator1 last1,
  InputIterator2 first2,
  InputIterator2 last2,
  OutputIterator result,
  StrictWeakCompare comp);
```

set_difference 构造一个有序范围，是有序范围 [first1, last1) 和 [first2, last2) 的差集。返回值是输出范围的末尾。

在最简单的情况下，set_difference 执行集合的差运算：输出范围由所有属于 [first1, last1) 而不属于 [first2, last1) 的元素副本组成。一般情况更为复杂，因为输入范围内可能包含重复元素。总而言之，如果 [first1, last1) 包含 m 个彼此等价的元素，且 [first2, last2) 包含 n 个与它们等价的元素，则 [first1, last1) 内最后 max(m-n,0) 个元素会被复制到输出范围。

此版本的 set_difference 使用函数对象 comp 比较元素。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::host 并行化执行策略，使用 set_difference 计算两组降序整数集合的差集：

```
#include <thrust/set_operations.h>
#include <thrust/functional.h>
#include <thrust/execution_policy.h>
...
int A1[7] = {9, 6, 5, 4, 3, 1, 0};
int A2[5] = {9, 7, 5, 3, 1};

int result[3];

int *result_end = thrust::set_difference(thrust::host, A1, A1 + 7, A2, A2
 ↪+ 5, result, thrust::greater<int>());
// result is now {6, 4, 0}
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **InputIterator1**：是 Input Iterator 的模型，InputIterator1 的 value_type 可转换为 StrictWeakCompare 的 first_argument_type 以及 OutputIterator 的 value_types 集合中的类型
- **InputIterator2**：是 Input Iterator 的模型，InputIterator2 的 value_type 可转换为 StrictWeakCompare 的 second_argument_type 以及 OutputIterator 的 value_types 集合中的类型
- **OutputIterator**：是 Output Iterator 的模型
- **StrictWeakCompare**：是 Strict Weak Ordering 的模型

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

函数参数

- **exec**：用于并行化的执行策略
- **first1**：第一个输入范围的起始
- **last1**-：第一个输入范围的末尾
- **first2**：第二个输入范围的起始
- **last2**：第二个输入范围的末尾
- **result**：输出范围的起始
- **comp**：比较运算符

前提条件

- 范围 [first1, last1) 和 [first2, last2) 应按照 comp 进行排序
- 结果范围不应与任何一个输入范围重叠

返回值

输出范围的末尾

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/set_difference
- includes
- set_union
- set_intersection
- set_symmetric_difference
- sort
- is_sorted

2.1.7.4 thrust::set_difference

```
template <typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator,
  typename StrictWeakCompare>
OutputIterator
set_difference(InputIterator1 first1,
  InputIterator1 last1,
  InputIterator2 first2,
  InputIterator2 last2,
  OutputIterator result,
  StrictWeakCompare comp);
```

set_difference 构造一个有序范围，是有序范围 [first1, last1) 和 [first2, last2) 的差集。返回值是输出范围的末尾。

在最简单的情况下，set_difference 执行集合的差运算：输出范围由所有属于 [first1, last1) 而不属于 [first2, last1) 的元素副本组成。一般情况更为复杂，因为输入范围内可能包含重复元素。总而言之，如果 [first1, last1) 包含 m 个彼此等价的元素，且 [first2, last2) 包含 n 个与它们等价的元素，则 [first1, last1) 内最后 max(m-n,0) 个元素会被复制到输出范围。

此版本的 set_difference 使用函数对象 comp 比较元素。

以下代码片段演示了如何使用 set_difference 计算两组降序整数集合的差集：

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

```
#include <thrust/set_operations.h>
#include <thrust/functional.h>
...
int A1[7] = {9, 6, 5, 4, 3, 1, 0};
int A2[5] = {9, 7, 5, 3, 1};

int result[3];

int *result_end = thrust::set_difference(A1, A1 + 7, A2, A2 + 5, result,
↪thrust::greater<int>());
// result is now {6, 4, 0}
```

模板参数

- **InputIterator1**：是 Input Iterator 的模型，InputIterator1 的 value_type 可转换为 StrictWeakCompare 的 first_argument_type 以及 OutputIterator 的 value_types 集合中的类型
- **InputIterator2**：是 Input Iterator 的模型，InputIterator2 的 value_type 可转换为 StrictWeakCompare 的 second_argument_type 以及 OutputIterator 的 value_types 集合中的类型
- **OutputIterator**：是 Output Iterator 的模型
- **StrictWeakCompare**：是 Strict Weak Ordering 的模型

函数参数

- **first1**：第一个输入范围的起始
- **last1**-：第一个输入范围的末尾
- **first2**：第二个输入范围的起始
- **last2**：第二个输入范围的末尾
- **result**：输出范围的起始
- **comp**：比较运算符

前提条件

- 范围 [first1, last1) 和 [first2, last2) 应按照 comp 进行排序
- 结果范围不应与任何一个输入范围重叠

返回值

输出范围的末尾

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/set_difference
- includes
- set_union
- set_intersection
- set_symmetric_difference
- sort
- is_sorted

2.1.7.5 thrust::set_intersection

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

```
template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator>
__host__ __device__ OutputIterator
set_intersection(const thrust::detail::execution_policy_base<
 ↪DerivedPolicy > & exec,
  InputIterator1 first1,
  InputIterator1 last1,
  InputIterator2 first2,
  InputIterator2 last2,
  OutputIterator result);
```

set_intersection 构造一个有序范围，是有序范围 [first1, last1) 和 [first2, last2) 的交集。返回值是输出范围的末尾。

在最简单的情况下，set_intersection 执行集合的交运算：输出范围由属于 [first1, last1) 和 [first2, last2) 的元素副本组成。一般情况更为复杂，因为输入范围内可能包含重复元素。总而言之，如果一个值在 [first1, last1) 内出现 m 次，在 [first2, last2) 内出现 n 次（其中 m 可能为零），则它在输出范围中会出现 min(m,n) 次。set_intersection 是稳定的，指的是元素都是从第一个范围而不是第二个范围复制的，并且输出范围中元素的相对顺序与第一个输入范围中的相同。

此版本的 set_intersection 使用 operator< 比较对象。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::host 并行化执行策略，使用 set_intersection 计算两组升序整数集合的交集：

```
#include <thrust/set_operations.h>
#include <thrust/execution_policy.h>
...
int A1[6] = {1, 3, 5, 7, 9, 11};
int A2[7] = {1, 1, 2, 3, 5,  8, 13};

int result[7];

int *result_end = thrust::set_intersection(thrust::host, A1, A1 + 6, A2,
 ↪A2 + 7, result);
// result is now {1, 3, 5}
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **InputIterator1**：是 Input Iterator 的模型。InputIterator1 和 InputIterator2 具有相同的 value_type，InputIterator1 的 value_type 是 LessThan Comparable 的模型，如 LessThan Comparable 要求中定义的，InputIterator1 的 value_type 上的排序是严格弱序，并且可转换为 OutputIterator 的 value_types 集合中的类型
- **InputIterator2**：是 Input Iterator 的模型。InputIterator2 和 InputIterator1 具有相同的 value_type，InputIterator2 的 value_type 是 LessThan Comparable 的模型，如 LessThan Comparable 要求中定义的，InputIterator2 的 value_type 上的排序是严格弱序，并且可转换为 OutputIterator 的 value_types 集合中的类型
- **OutputIterator**：是 Output Iterator 的模型

函数参数

- **exec**：用于并行化的执行策略
- **first1**：第一个输入范围的起始
- **last1**-：第一个输入范围的末尾

飞谱信息 MetaX Confidential 2024-11-18 15:00:00

- **first2**：第二个输入范围的起始
- **last2**：第二个输入范围的末尾
- **result**：输出范围的起始

前提条件

- 范围 [first1, last1) 和 [first2, last2) 应按照 operator< 进行排序
- 结果范围不应与任何一个输入范围重叠

返回值

输出范围的末尾

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/set_intersection
- includes
- set_union
- set_intersection
- set_symmetric_difference
- sort
- is_sorted

2.1.7.6 thrust::set_intersection

```
template <typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator>
OutputIterator
set_intersection(InputIterator1 first1,
  InputIterator1 last1,
  InputIterator2 first2,
  InputIterator2 last2,
  OutputIterator result);
```

set_intersection 构造一个有序范围，是有序范围 [first1, last1) 和 [first2, last2) 的交集。返回值是输出范围的末尾。

在最简单的情况下，set_intersection 执行集合的交运算：输出范围由属于 [first1, last1) 和 [first2, last2) 的元素副本组成。一般情况更为复杂，因为输入范围内可能包含重复元素。总而言之，如果一个值在 [first1, last1) 内出现 m 次，在 [first2, last2) 内出现 n 次（其中 m 可能为零），则它在输出范围中会出现 min(m,n) 次。set_intersection 是稳定的，指的是元素都是从第一个范围而不是第二个范围复制的，并且输出范围中元素的相对顺序与第一个输入范围中的相同。

此版本的 set_intersection 使用 operator< 比较对象。

以下代码片段演示了如何使用 set_intersection 计算两组升序整数集合的交集：

```
#include <thrust/set_operations.h>
...
int A1[6] = {1, 3, 5, 7, 9, 11};
int A2[7] = {1, 1, 2, 3, 5,  8, 13};

int result[7];

int *result_end = thrust::set_intersection(A1, A1 + 6, A2, A2 + 7, result);
// result is now {1, 3, 5}
```

飞腾信息 MetaX Confidential 2025-11-18 15:00:00

模板参数

- **InputIterator1**：是 Input Iterator 的模型。InputIterator1 和 InputIterator2 具有相同的 value_type，InputIterator1 的 value_type 是 LessThan Comparable 的模型，如 LessThan Comparable 要求中定义的，InputIterator1 的 value_type 上的排序是严格弱序，并且可转换为 OutputIterator 的 value_types 集合中的类型
- **InputIterator2**：是 Input Iterator 的模型。InputIterator2 和 InputIterator1 具有相同的 value_type，InputIterator2 的 value_type 是 LessThan Comparable 的模型，如 LessThan Comparable 要求中定义的，InputIterator2 的 value_type 上的排序是严格弱序，并且可转换为 OutputIterator 的 value_types 集合中的类型
- **OutputIterator**：是 Output Iterator 的模型

函数参数

- **first1**：第一个输入范围的起始
- **last1**-：第一个输入范围的末尾
- **first2**：第二个输入范围的起始
- **last2**：第二个输入范围的末尾
- **result**：输出范围的起始

前提条件

- 范围 [first1, last1) 和 [first2, last2) 应按照 operator< 进行排序
- 结果范围不应与任何一个输入范围重叠

返回值

输出范围的末尾

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/set_intersection
- includes
- set_union
- set_intersection
- set_symmetric_difference
- sort
- is_sorted

2.1.7.7 thrust::set_intersection

```
template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator,
  typename StrictWeakCompare>
__host__ __device__ OutputIterator
set_intersection(const thrust::detail::execution_policy_base<
↪DerivedPolicy > & exec,
  InputIterator1 first1,
  InputIterator1 last1,
  InputIterator2 first2,
  InputIterator2 last2,
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
  OutputIterator result,
  StrictWeakCompare comp);
```

set_intersection 构造一个有序范围，是有序范围 [first1, last1) 和 [first2, last2) 的交集。返回值是输出范围的末尾。

在最简单的情况下，set_intersection 执行集合的交运算：输出范围由属于 [first1, last1) 和 [first2, last2) 的元素副本组成。一般情况更为复杂，因为输入范围内可能包含重复元素。总而言之，如果一个值在 [first1, last1) 内出现 m 次，在 [first2, last2) 内出现 n 次（其中 m 可能为零），则它在输出范围中会出现 min(m,n) 次。set_intersection 是稳定的，指的是元素都是从第一个范围而不是第二个范围复制的，并且输出范围中元素的相对顺序与第一个输入范围中的相同。

此版本的 set_intersection 使用函数对象 comp 比较元素。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::host 并行化执行策略，使用 set_intersection 计算两组降序整数集合的交集：

```
#include <thrust/set_operations.h>
#include <thrust/execution_policy.h>
...
int A1[6] = {11, 9, 7, 5, 3, 1};
int A2[7] = {13, 8, 5, 3, 2,  1, 1};

int result[3];

int *result_end = thrust::set_intersection(thrust::host, A1, A1 + 6, A2,
↪A2 + 7, result, thrust::greater<int>());
// result is now {5, 3, 1}
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **InputIterator1**：是 Input Iterator 的模型。InputIterator1 和 InputIterator2 具有相同的 value_type，InputIterator1 的 value_type 是 LessThan Comparable 的模型，如 LessThan Comparable 要求中定义的，InputIterator1 的 value_type 上的排序是严格弱序，并且可转换为 OutputIterator 的 value_types 集合中的类型
- **InputIterator2**：是 Input Iterator 的模型。InputIterator2 和 InputIterator1 具有相同的 value_type，InputIterator2 的 value_type 是 LessThan Comparable 的模型，如 LessThan Comparable 要求中定义的，InputIterator2 的 value_type 上的排序是严格弱序，并且可转换为 OutputIterator 的 value_types 集合中的类型
- **OutputIterator**：是 Output Iterator 的模型

函数参数

- **exec**：用于并行化的执行策略
- **first1**：第一个输入范围的起始
- **last1**-：第一个输入范围的末尾
- **first2**：第二个输入范围的起始
- **last2**：第二个输入范围的末尾
- **result**：输出范围的起始
- **comp**：比较运算符

前提条件

- 范围 [first1, last1) 和 [first2, last2) 应按照 comp 进行排序

飞腾信息 MetaX Confidential 2024-11-19 15:00:00

- 结果范围不应与任何一个输入范围重叠

返回值

输出范围的末尾

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/set_intersection
- includes
- set_union
- set_intersection
- set_symmetric_difference
- sort
- is_sorted

2.1.7.8 thrust::set_intersection

```
template <typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator,
  typename StrictWeakCompare>
OutputIterator
set_intersection(InputIterator1 first1,
  InputIterator1 last1,
  InputIterator2 first2,
  InputIterator2 last2,
  OutputIterator result,
  StrictWeakCompare comp);
```

set_intersection 构造一个有序范围，是有序范围 [first1, last1) 和 [first2, last2) 的交集。返回值是输出范围的末尾。

在最简单的情况下，set_intersection 执行集合的交运算：输出范围由属于 [first1, last1) 和 [first2, last2) 的元素副本组成。一般情况更为复杂，因为输入范围内可能包含重复元素。总而言之，如果一个值在 [first1, last1) 内出现 m 次，在 [first2, last2) 内出现 n 次（其中 m 可能为零），则它在输出范围中会出现 min(m,n) 次。set_intersection 是稳定的，指的是元素都是从第一个范围而不是第二个范围复制的，并且输出范围中元素的相对顺序与第一个输入范围中的相同。

此版本的 set_intersection 使用函数对象 comp 比较元素。

以下代码片段演示了如何使用 set_intersection 计算两组降序整数集合的交集：

```
#include <thrust/set_operations.h>
...
int A1[6] = {11, 9, 7, 5, 3, 1};
int A2[7] = {13, 8, 5, 3, 2,  1, 1};

int result[3];

int *result_end = thrust::set_intersection(A1, A1 + 6, A2, A2 + 7, result,
↪thrust::greater<int>());
// result is now {5, 3, 1}
```

模板参数

- **InputIterator1**：是 Input Iterator 的模型。InputIterator1 和 InputIterator2 具有相同的 value_type，InputIterator1 的 value_type 是 LessThan Comparable 的模型，如 LessThan Comparable 要求

沐曦 MetaX Confidential 2024-[illegible]-18 15:00:00

中定义的，InputIterator1 的 value_type 上的排序是严格弱序，并且可转换为 OutputIterator 的 value_types 集合中的类型

- **InputIterator2**：是 Input Iterator 的模型。InputIterator2 和 InputIterator1 具有相同的 value_type，InputIterator2 的 value_type 是 LessThan Comparable 的模型，如 LessThan Comparable 要求中定义的，InputIterator2 的 value_type 上的排序是严格弱序，并且可转换为 OutputIterator 的 value_types 集合中的类型
- **OutputIterator**：是 Output Iterator 的模型

函数参数

- **first1**：第一个输入范围的起始
- **last1**-：第一个输入范围的末尾
- **first2**：第二个输入范围的起始
- **last2**：第二个输入范围的末尾
- **result**：输出范围的起始
- **comp**：比较运算符

前提条件

- 范围 [first1, last1) 和 [first2, last2) 应按照 comp 进行排序
- 结果范围不应与任何一个输入范围重叠

返回值

输出范围的末尾

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/set_intersection
- includes
- set_union
- set_intersection
- set_symmetric_difference
- sort
- is_sorted

2.1.7.9 thrust::set_symmetric_difference

```
template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator>
__host__ __device__ OutputIterator
set_symmetric_difference(const thrust::detail::execution_policy_base<
↪DerivedPolicy > & exec,
  InputIterator1 first1,
  InputIterator1 last1,
  InputIterator2 first2,
  InputIterator2 last2,
  OutputIterator result);
```

set_symmetric_difference 构造一个有序范围，是有序范围 [first1, last1) 和 [first2, last2) 的对称差集。返回值是输出范围的末尾。

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

在最简单的情况下，set_symmetric_difference 实现以下集合运算：构造两个集合 A - B 和 B - A 的并集，其中 A 和 B 是两个输入范围。即，输出范围由属于 [first1, last1) 但不属于 [first2, last2) 的元素副本，和属于 [first2, last2) 但不属于 [first1, last1) 的元素副本组成。一般情况更为复杂，因为输入范围内可能包含重复元素。总而言之，如果 [first1, last1) 包含 m 个彼此等价的元素，[first2, last1) 包含 n 个彼此等价的元素，则 |m - n| 个这样的元素应被复制到输出范围：如果 m > n ，则复制 [first1, last1) 内最后 m - n 个元素；如果 m < n ，则复制 [first2, last2) 中最后 n - m 个元素。

此版本的 set_union 使用 operator< 比较元素。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::host 并行化执行策略，使用 set_symmetric_difference 计算两组升序整数集合的对称差集：

```
#include <thrust/set_operations.h>
#include <thrust/execution_policy.h>
...
int A1[7] = {0, 1, 2, 2, 4, 6, 7};
int A2[5] = {1, 1, 2, 5, 8};

int result[6];

int *result_end = thrust::set_symmetric_difference(thrust::host, A1, A1 +
↪7, A2, A2 + 5, result);
// result = {0, 4, 5, 6, 7, 8}
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **InputIterator1**：是 Input Iterator 的模型。InputIterator1 和 InputIterator2 具有相同的 value_type ，InputIterator1 的 value_type 是 LessThan Comparable 的模型，如 LessThan Comparable 要求中定义的，InputIterator1 的 value_type 上的排序是严格弱序，并且可转换为 OutputIterator 的 value_types 集合中的类型
- **InputIterator2**：是 Input Iterator 的模型。InputIterator2 和 InputIterator1 具有相同的 value_type ，InputIterator2 的 value_type 是 LessThan Comparable 的模型，如 LessThan Comparable 要求中定义的，InputIterator2 的 value_type 上的排序是严格弱序，并且可转换为 OutputIterator 的 value_types 集合中的类型
- **OutputIterator**：是 Output Iterator 的模型

函数参数

- **exec**：用于并行化的执行策略
- **first1**：第一个输入范围的起始
- **last1**：第一个输入范围的末尾
- **first2**：第二个输入范围的起始
- **last2**：第二个输入范围的末尾
- **result**：输出范围的起始

前提条件

- 范围 [first1, last1) 和 [first2, last2) 应按照 operator< 进行排序
- 结果范围不应与任何一个输入范围重叠

返回值

输出范围的末尾

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/set_symmetric_difference

沐曦 MetaX Confidential 2024-11-18 15:00:00

- merge
- includes
- set_difference
- set_union
- set_intersection
- sort
- is_sorted

2.1.7.10 thrust::set_symmetric_difference

```
template <typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator>
OutputIterator
set_symmetric_difference(InputIterator1 first1,
  InputIterator1 last1,
  InputIterator2 first2,
  InputIterator2 last2,
  OutputIterator result);
```

set_symmetric_difference 构造一个有序范围，是有序范围 [first1, last1) 和 [first2, last2) 的对称差集。返回值是输出范围的末尾。

在最简单的情况下，set_symmetric_difference 实现以下集合运算：构造两个集合 A - B 和 B - A 的并集，其中 A 和 B 是两个输入范围。即，输出范围由属于 [first1, last1) 但不属于 [first2, last1) 的元素副本，和属于 [first2, last2) 但不属于 [first1, last1) 的元素副本组成。一般情况更为复杂，因为输入范围内可能包含重复元素。总而言之，如果 [first1, last1) 包含 m 个彼此等价的元素，[first2, last1) 包含 n 个彼此等价的元素，则 |m - n| 个这样的元素应被复制到输出范围：如果 m > n ，则复制 [first1, last1) 内最后 m - n 个元素；如果 m < n ，则复制 [first2, last2) 中最后 n - m 个元素。

此版本的 set_union 使用 operator< 比较元素。

以下代码片段演示了如何使用 set_symmetric_difference 计算两组升序整数集合的对称差集：

```
#include <thrust/set_operations.h>
...
int A1[7] = {0, 1, 2, 2, 4, 6, 7};
int A2[5] = {1, 1, 2, 5, 8};

int result[6];

int *result_end = thrust::set_symmetric_difference(A1, A1 + 7, A2, A2 + 5,
↪result);
// result = {0, 4, 5, 6, 7, 8}
```

模板参数

- **InputIterator1**：是 Input Iterator 的模型。InputIterator1 和 InputIterator2 具有相同的 value_type ，InputIterator1 的 value_type 是 LessThan Comparable 的模型，如 LessThan Comparable 要求中定义的，InputIterator1 的 value_type 上的排序是严格弱序，并且可转换为 OutputIterator 的 value_types 集合中的类型
- **InputIterator2**：是 Input Iterator 的模型。InputIterator2 和 InputIterator1 具有相同的 value_type ，InputIterator2 的 value_type 是 LessThan Comparable 的模型，如 LessThan Comparable 要求中定义的，InputIterator2 的 value_type 上的排序是严格弱序，并且可转换为 OutputIterator 的 value_types 集合中的类型

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

- **OutputIterator**：是 Output Iterator 的模型

函数参数

- **first1**：第一个输入范围的起始
- **last1**-：第一个输入范围的末尾
- **first2**：第二个输入范围的起始
- **last2**：第二个输入范围的末尾
- **result**：输出范围的起始

前提条件

- 范围 [first1, last1) 和 [first2, last2) 应按照 operator< 进行排序
- 结果范围不应与任何一个输入范围重叠

返回值

输出范围的末尾

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/set_symmetric_difference
- merge
- includes
- set_difference
- set_union
- set_intersection
- sort
- is_sorted

2.1.7.11 thrust::set_symmetric_difference

```
template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator,
  typename StrictWeakCompare>
__host__ __device__ OutputIterator
set_symmetric_difference(const thrust::detail::execution_policy_base<
 ↪DerivedPolicy > & exec,
  InputIterator1 first1,
  InputIterator1 last1,
  InputIterator2 first2,
  InputIterator2 last2,
  OutputIterator result,
  StrictWeakCompare comp);
```

set_symmetric_difference 构造一个有序范围，是有序范围 [first1, last1) 和 [first2, last2) 的对称差集。返回值是输出范围的末尾。

在最简单的情况下，set_symmetric_difference 实现以下集合运算：构造两个集合 A - B 和 B - A 的并集，其中 A 和 B 是两个输入范围。即，输出范围由属于 [first1, last1) 但不属于 [first2, last1) 的元素副本，和属于 [first2, last2) 但不属于 [first1, last1) 的元素副本组成。一般情况更为复杂，因为输入范围内可能包含重复元素。总而言之，如果 [first1, last1) 包含 m 个彼此等价的元素，[first2, last1) 包含 n 个彼

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

此等价的元素，则 |m - n| 个这样的元素应被复制到输出范围：如果 m > n ，则复制 [first1, last1) 内最后 m - n 个元素；如果 m < n ，则复制 [first2, last2) 中最后 n - m 个元素。

此版本的 set_union 使用函数对象 comp 比较元素。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::host 并行化执行策略，使用 set_symmetric_difference 计算两组降序整数集合的对称差集：

```
#include <thrust/set_operations.h>
#include <thrust/execution_policy.h>
...
int A1[7] = {7, 6, 4, 2, 2, 1, 0};
int A2[5] = {8, 5, 2, 1, 1};

int result[6];

int *result_end = thrust::set_symmetric_difference(thrust::host, A1, A1 +
↪7, A2, A2 + 5, result);
// result = {8, 7, 6, 5, 4, 0}
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **InputIterator1**：是 Input Iterator 的模型。InputIterator1 和 InputIterator2 具有相同的 value_type ，InputIterator1 的 value_type 是 LessThan Comparable 的模型，如 LessThan Comparable 要求中定义的，InputIterator1 的 value_type 上的排序是严格弱序，并且可转换为 OutputIterator 的 value_types 集合中的类型
- **InputIterator2**：是 Input Iterator 的模型。InputIterator2 和 InputIterator1 具有相同的 value_type ，InputIterator2 的 value_type 是 LessThan Comparable 的模型，如 LessThan Comparable 要求中定义的，InputIterator2 的 value_type 上的排序是严格弱序，并且可转换为 OutputIterator 的 value_types 集合中的类型
- **OutputIterator**：是 Output Iterator 的模型

函数参数

- **exec**：用于并行化的执行策略
- **first1**：第一个输入范围的起始
- **last1**-：第一个输入范围的末尾
- **first2**：第二个输入范围的起始
- **last2**：第二个输入范围的末尾
- **result**：输出范围的起始
- **comp**：比较运算符

前提条件

- 范围 [first1, last1) 和 [first2, last2) 应按照 comp 进行排序
- 结果范围不应与任何一个输入范围重叠

返回值

输出范围的末尾

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/set_symmetric_difference
- merge

飞腾信息 MetaX Confidential 2024-11-18 15:07:00

- includes
- set_difference
- set_union
- set_intersection
- sort
- is_sorted

2.1.7.12 thrust::set_symmetric_difference

```
template <typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator,
  typename StrictWeakCompare>
OutputIterator
set_symmetric_difference(InputIterator1 first1,
  InputIterator1 last1,
  InputIterator2 first2,
  InputIterator2 last2,
  OutputIterator result,
  StrictWeakCompare comp);
```

set_symmetric_difference 构造一个有序范围，是有序范围 [first1, last1) 和 [first2, last2) 的对称差集。返回值是输出范围的末尾。

在最简单的情况下，set_symmetric_difference 实现以下集合运算：构造两个集合 A - B 和 B - A 的并集，其中 A 和 B 是两个输入范围。即，输出范围由属于 [first1, last1) 但不属于 [first2, last1) 的元素副本，和属于 [first2, last2) 但不属于 [first1, last1) 的元素副本组成。一般情况更为复杂，因为输入范围内可能包含重复元素。总而言之，如果 [first1, last1) 包含 m 个彼此等价的元素，[first2, last1) 包含 n 个彼此等价的元素，则 |m - n| 个这样的元素应被复制到输出范围：如果 m > n ，则复制 [first1, last1) 内最后 m - n 个元素；如果 m < n ，则复制 [first2, last2) 中最后 n - m 个元素。

此版本的 set_union 使用函数对象 comp 比较元素。

以下代码片段演示了如何使用 set_symmetric_difference 计算两组降序整数集合的对称差集：

```
#include <thrust/set_operations.h>
...
int A1[7] = {7, 6, 4, 2, 2, 1, 0};
int A2[5] = {8, 5, 2, 1, 1};

int result[6];

int *result_end = thrust::set_symmetric_difference(A1, A1 + 7, A2, A2 + 5,
↪result);
// result = {8, 7, 6, 5, 4, 0}
```

模板参数

- **InputIterator1**：是 Input Iterator 的模型。InputIterator1 和 InputIterator2 具有相同的 value_type ，InputIterator1 的 value_type 是 LessThan Comparable 的模型，如 LessThan Comparable 要求中定义的，InputIterator1 的 value_type 上的排序是严格弱序，并且可转换为 OutputIterator 的 value_types 集合中的类型
- **InputIterator2**：是 Input Iterator 的模型。InputIterator2 和 InputIterator1 具有相同的 value_type ，InputIterator2 的 value_type 是 LessThan Comparable 的模型，如 LessThan Comparable 要求中定义的，InputIterator2 的 value_type 上的排序是严格弱序，并且可转换为 OutputIterator 的 value_types 集合中的类型

飞腾信息 MetaX Confidential 2024-11-18 00:00

- **OutputIterator**：是 Output Iterator 的模型

函数参数

- **first1**：第一个输入范围的起始
- **last1**-：第一个输入范围的末尾
- **first2**：第二个输入范围的起始
- **last2**：第二个输入范围的末尾
- **result**：输出范围的起始
- **comp**：比较运算符

前提条件

- 范围 [first1, last1) 和 [first2, last2) 应按照 comp 进行排序
- 结果范围不应与任何一个输入范围重叠

返回值

输出范围的末尾

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/set_symmetric_difference
- merge
- includes
- set_difference
- set_union
- set_intersection
- sort
- is_sorted

2.1.7.13 thrust::set_union

```
template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator>
__host__ __device__ OutputIterator
set_union(const thrust::detail::execution_policy_base< DerivedPolicy > &
 ↪exec,
  InputIterator1 first1,
  InputIterator1 last1,
  InputIterator2 first2,
  InputIterator2 last2,
  OutputIterator result);
```

set_union 构造一个有序范围，是有序范围 [first1, last1) 和 [first2, last2) 的并集。返回值是输出范围的末尾。

在最简单的情况下，set_union 执行集合的并运算：输出范围由属于 [first2, last1) 或 [first1, last1)，或者属于两者的元素副本组成。一般情况更为复杂，因为输入范围内可能包含重复元素。总而言之，如果 [first1, last1) 包含 m 个彼此等价的元素，[first2, last2) 包含 n 个彼此等价的元素，则第一个范围内的所有 m 个元素都应按顺序复制到输出范围，第二个范围内的 max(n - m, 0) 个元素应按顺序复制到输出范围。

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

此版本的 set_union 使用 operator< 比较元素。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::host 并行化执行策略，使用 set_union 计算两组升序整数集合的并集：

```
#include <thrust/set_operations.h>
#include <thrust/execution_policy.h>
...
int A1[7] = {0, 2, 4, 6, 8, 10, 12};
int A2[5] = {1, 3, 5, 7, 9};

int result[11];

int *result_end = thrust::set_union(thrust::host, A1, A1 + 7, A2, A2 + 5,
↪result);
// result = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12}
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **InputIterator1**：是 Input Iterator 的模型。InputIterator1 和 InputIterator2 具有相同的 value_type，InputIterator1 的 value_type 是 LessThan Comparable 的模型，如 LessThan Comparable 要求中定义的，InputIterator1 的 value_type 上的排序是严格弱序，并且可转换为 OutputIterator 的 value_types 集合中的类型
- **InputIterator2**：是 Input Iterator 的模型。InputIterator2 和 InputIterator1 具有相同的 value_type，InputIterator2 的 value_type 是 LessThan Comparable 的模型，如 LessThan Comparable 要求中定义的，InputIterator2 的 value_type 上的排序是严格弱序，并且可转换为 OutputIterator 的 value_types 集合中的类型
- **OutputIterator**：是 Output Iterator 的模型

函数参数

- **exec**：用于并行化的执行策略
- **first1**：第一个输入范围的起始
- **last1**-：第一个输入范围的末尾
- **first2**：第二个输入范围的起始
- **last2**：第二个输入范围的末尾
- **result**：输出范围的起始

前提条件

- 范围 [first1, last1) 和 [first2, last2) 应按照 operator< 进行排序
- 结果范围不应与任何一个输入范围重叠

返回值

输出范围的末尾

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/set_union
- merge
- includes
- set_union
- set_intersection

飞猪信息 MetaX Confidential 2024-11-18 15:00:00

- set_symmetric_difference
- sort
- is_sorted

2.1.7.14 thrust::set_union

```
template <typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator>
OutputIterator
set_union(InputIterator1 first1,
  InputIterator1 last1,
  InputIterator2 first2,
  InputIterator2 last2,
  OutputIterator result);
```

set_union 构造一个有序范围，是有序范围 [first1, last1) 和 [first2, last2) 的并集。返回值是输出范围的末尾。

在最简单的情况下，set_union 执行集合的并运算：输出范围由属于 [first2, last1) 或 [first1, last1)，或者属于两者的元素副本组成。一般情况更为复杂，因为输入范围内可能包含重复元素。总而言之，如果 [first1, last1) 包含 m 个彼此等价的元素，[first2, last2) 包含 n 个彼此等价的元素，则第一个范围内的所有 m 个元素都应按顺序复制到输出范围，第二个范围内的 max(n - m, 0) 个元素应按顺序复制到输出范围。

此版本的 set_union 使用 operator< 比较元素。

以下代码片段演示了如何使用 set_union 计算两组升序整数集合的并集：

```
#include <thrust/set_operations.h>
...
int A1[7] = {0, 2, 4, 6, 8, 10, 12};
int A2[5] = {1, 3, 5, 7, 9};

int result[11];

int *result_end = thrust::set_union(A1, A1 + 7, A2, A2 + 5, result);
// result = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12}
```

模板参数

- **InputIterator1**：是 Input Iterator 的模型。InputIterator1 和 InputIterator2 具有相同的 value_type，InputIterator1 的 value_type 是 LessThan Comparable 的模型，如 LessThan Comparable 要求中定义的，InputIterator1 的 value_type 上的排序是严格弱序，并且可转换为 OutputIterator 的 value_types 集合中的类型
- **InputIterator2**：是 Input Iterator 的模型。InputIterator2 和 InputIterator1 具有相同的 value_type，InputIterator2 的 value_type 是 LessThan Comparable 的模型，如 LessThan Comparable 要求中定义的，InputIterator2 的 value_type 上的排序是严格弱序，并且可转换为 OutputIterator 的 value_types 集合中的类型
- **OutputIterator**：是 Output Iterator 的模型

函数参数

- **first1**：第一个输入范围的起始
- **last1**-：第一个输入范围的末尾
- **first2**：第二个输入范围的起始
- **last2**：第二个输入范围的末尾

飞腾信息 MetaX Confidential 2025-11-18 15:00:00

- **result**：输出范围的起始

前提条件

- 范围 [first1, last1) 和 [first2, last2) 应按照 operator< 进行排序
- 结果范围不应与任何一个输入范围重叠

返回值

输出范围的末尾

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/set_union
- merge
- includes
- set_union
- set_intersection
- set_symmetric_difference
- sort
- is_sorted

2.1.7.15 thrust::set_union

```
template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator,
  typename StrictWeakCompare>
__host__ __device__ OutputIterator
set_union(const thrust::detail::execution_policy_base< DerivedPolicy > &
↪exec,
  InputIterator1 first1,
  InputIterator1 last1,
  InputIterator2 first2,
  InputIterator2 last2,
  OutputIterator result,
  StrictWeakCompare comp);
```

set_union 构造一个有序范围，是有序范围 [first1, last1) 和 [first2, last2) 的并集。返回值是输出范围的末尾。

在最简单的情况下，set_union 执行集合的并运算：输出范围由属于 [first2, last1) 或 [first1, last1)，或者属于两者的元素副本组成。一般情况更为复杂，因为输入范围内可能包含重复元素。总而言之，如果 [first1, last1) 包含 m 个彼此等价的元素，[first2, last2) 包含 n 个彼此等价的元素，则第一个范围内的所有 m 个元素都应按顺序复制到输出范围，第二个范围内的 max(n - m, 0) 个元素应按顺序复制到输出范围。

此版本的 set_union 使用函数对象 comp 比较元素。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::host 并行化执行策略，使用 set_union 计算两组降序整数集合的并集：

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

```
#include <thrust/set_operations.h>
#include <thrust/functional.h>
#include <thrust/execution_policy.h>
...
int A1[7] = {12, 10, 8, 6, 4, 2, 0};
int A2[5] = {9, 7, 5, 3, 1};

int result[11];

int *result_end = thrust::set_union(thrust::host, A1, A1 + 7, A2, A2 + 5,
↪result, thrust::greater<int>());
// result = {12, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0}
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **InputIterator1**：是 Input Iterator 的模型，InputIterator1 的 value_type 可转换为 StrictWeakCompare 的 first_argument_type 以及 OutputIterator 的 value_types 集合中的类型
- **InputIterator2**：是 Input Iterator 的模型，InputIterator2 的 value_type 可转换为 StrictWeakCompare 的 second_argument_type 以及 OutputIterator 的 value_types 集合中的类型
- **OutputIterator**：是 Output Iterator 的模型
- **StrictWeakCompare**：是 Strict Weak Ordering 的模型

函数参数

- **exec**：用于并行化的执行策略
- **first1**：第一个输入范围的起始
- **last1**-：第一个输入范围的末尾
- **first2**：第二个输入范围的起始
- **last2**：第二个输入范围的末尾
- **result**：输出范围的起始
- **comp**：比较运算符

前提条件

- 范围 [first1, last1) 和 [first2, last2) 应按照 comp 进行排序
- 结果范围不应与任何一个输入范围重叠

返回值

输出范围的末尾

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/set_union
- merge
- includes
- set_union
- set_intersection
- set_symmetric_difference
- sort
- is_sorted

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

2.1.7.16 thrust::set_union

```
template <typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator,
  typename StrictWeakCompare>
OutputIterator
set_union(InputIterator1 first1,
  InputIterator1 last1,
  InputIterator2 first2,
  InputIterator2 last2,
  OutputIterator result,
  StrictWeakCompare comp);
```

set_union 构造一个有序范围，是有序范围 [first1, last1) 和 [first2, last2) 的并集。返回值是输出范围的末尾。

在最简单的情况下，set_union 执行集合的并运算：输出范围由属于 [first2, last1) 或 [first1, last1)，或者属于两者的元素副本组成。一般情况更为复杂，因为输入范围内可能包含重复元素。总而言之，如果 [first1, last1) 包含 m 个彼此等价的元素，[first2, last2) 包含 n 个彼此等价的元素，则第一个范围内的所有 m 个元素都应按顺序复制到输出范围，第二个范围内的 max(n - m, 0) 个元素应按顺序复制到输出范围。

此版本的 set_union 使用函数对象 comp 比较元素。

以下代码片段演示了如何使用 set_union 计算两组降序整数集合的并集：

```
#include <thrust/set_operations.h>
#include <thrust/functional.h>
...
int A1[7] = {12, 10, 8, 6, 4, 2, 0};
int A2[5] = {9, 7, 5, 3, 1};

int result[11];

int *result_end = thrust::set_union(A1, A1 + 7, A2, A2 + 5, result,
↪thrust::greater<int>());
// result = {12, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0}
```

模板参数

- **InputIterator1**：是 Input Iterator 的模型，InputIterator1 的 value_type 可转换为 StrictWeakCompare 的 first_argument_type 以及 OutputIterator 的 value_types 集合中的类型
- **InputIterator2**：是 Input Iterator 的模型，InputIterator2 的 value_type 可转换为 StrictWeakCompare 的 second_argument_type 以及 OutputIterator 的 value_types 集合中的类型
- **OutputIterator**：是 Output Iterator 的模型
- **StrictWeakCompare**：是 Strict Weak Ordering 的模型

函数参数

- **first1**：第一个输入范围的起始
- **last1**-：第一个输入范围的末尾
- **first2**：第二个输入范围的起始
- **last2**：第二个输入范围的末尾
- **result**：输出范围的起始
- **comp**：比较运算符

飞腾信息 MetaX confidential 2024-11-18 15:00:00

前提条件

- 范围 [first1, last1) 和 [first2, last2) 应按照 comp 进行排序
- 结果范围不应与任何一个输入范围重叠

返回值

输出范围的末尾

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/set_union
- merge
- includes
- set_union
- set_intersection
- set_symmetric_difference
- sort
- is_sorted

2.1.7.17 thrust::set_difference_by_key

```
template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2,
  typename InputIterator3,
  typename InputIterator4,
  typename OutputIterator1,
  typename OutputIterator2>
__host__ __device__ thrust::pair< OutputIterator1, OutputIterator2 >
set_difference_by_key(const thrust::detail::execution_policy_base<
 ↪DerivedPolicy > & exec,
  InputIterator1 keys_first1,
  InputIterator1 keys_last1,
  InputIterator2 keys_first2,
  InputIterator2 keys_last2,
  InputIterator3 values_first1,
  InputIterator4 values_first2,
  OutputIterator1 keys_result,
  OutputIterator2 values_result);
```

set_difference_by_key 执行集合的键值差运算。set_difference_by_key 构造一个有序范围，是有序范围 [keys_first1, keys_last1) 和 [keys_first2, keys_last2) 的差集。与输入和输出键范围内的每个元素相关联的是一个值元素。关联的输入值范围不需要排序。

在最简单的情况下，set_difference_by_key 执行集合的键值差运算：键输出范围由 属于 [keys_first1, keys_last1) 而不属于 [keys_first2, keys_last2) 的元素副本组成。一般情况更为复杂，因为输入范围内可能包含重复元素。总而言之，如果 [keys_first1, keys_last1) 包含 m 个彼此等价的元素，且 [keys_first2, keys_last2) 包含 n 个与它们等价的元素，则 [keys_first1, keys_last1) 内最后 max(m-n,0) 个元素会被复制到输出范围。

每次将键元素从 [keys_first1, keys_last1) 或 [keys_first2, keys_last2) 复制到键输出范围时，将相应的值元素从相应的值输入范围（从 values_first1 或 values_first2 开始）复制到值输出范围。

此版本的 set_difference_by_key 使用 operator< 比较键元素。

算法的执行由 exec 确定并行化。

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

以下代码片段演示了如何通过 thrust::host 并行化执行策略，使用 set_difference_by_key 计算两组键为升序整数的键-值对集合的差集：

```
#include <thrust/set_operations.h>
#include <thrust/execution_policy.h>
...
int A_keys[6] = {0, 1, 3, 4, 5, 6, 9};
int A_vals[6] = {0, 0, 0, 0, 0, 0, 0};

int B_keys[5] = {1, 3, 5, 7, 9};
int B_vals[5] = {1, 1, 1, 1, 1};

int keys_result[3];
int vals_result[3];

thrust::pair<int*,int*> end =
  thrust::set_difference_by_key(thrust::host,
            A_keys,
            A_keys + 6,
            B_keys,
            B_keys + 5,
            A_vals,
            B_vals,
            keys_result,
            vals_result);

// keys_result is now {0, 4, 6}
// vals_result is now {0, 0, 0}
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **InputIterator1**：是 Input Iterator 的模型。InputIterator1 和 InputIterator2 具有相同的 value_type，InputIterator1 的 value_type 是 LessThan Comparable 的模型，如 LessThan Comparable 要求中定义的，InputIterator1 的 value_type 上的排序是严格弱序，并且可转换为 OutputIterator 的 value_types 集合中的类型
- **InputIterator2**：是 Input Iterator 的模型。InputIterator2 和 InputIterator1 具有相同的 value_type，InputIterator2 的 value_type 是 LessThan Comparable 的模型，如 LessThan Comparable 要求中定义的，InputIterator2 的 value_type 上的排序是严格弱序，并且可转换为 OutputIterator 的 value_types 集合中的类型
- **InputIterator3**：是 Input Iterator 的模型，InputIterator3 的 value_type 可转换为 OutputIterator2 的 value_types 集合中的类型
- **InputIterator4**：是 Input Iterator 的模型，InputIterator4 的 value_type 可转换为 OutputIterator2 的 value_types 集合中的类型
- **OutputIterator1**：是 Output Iterator 的模型
- **OutputIterator2**：是 Output Iterator 的模型

函数参数

- **exec**：用于并行化的执行策略
- **keys_first1**：第一个键输入范围的起始
- **keys_last1**：第一个键输入范围的末尾
- **keys_first2**：第二个键输入范围的起始
- **keys_last2**：第二个键输入范围的末尾

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

- **values_first1**：第一个值输入范围的起始
- **values_first2**：第二个值输入范围的起始
- **keys_result**：键输出范围的起始
- **values_result**：值输出范围的起始

前提条件

- 范围 [keys_first1, keys_last1) 和 [keys_first2, keys_last2) 应按照 operator< 进行排序
- 结果范围不应与任何一个输入范围重叠

返回值

p pair ，其中 p.first 是键输出范围的末尾，p.second 是值输出范围的末尾

另请参阅

- set_union_by_key
- set_intersection_by_key
- set_symmetric_difference_by_key
- sort_by_key
- is_sorted

2.1.7.18 thrust::set_difference_by_key

```
template <typename InputIterator1,
  typename InputIterator2,
  typename InputIterator3,
  typename InputIterator4,
  typename OutputIterator1,
  typename OutputIterator2>
thrust::pair< OutputIterator1, OutputIterator2 >
set_difference_by_key(InputIterator1 keys_first1,
  InputIterator1 keys_last1,
  InputIterator2 keys_first2,
  InputIterator2 keys_last2,
  InputIterator3 values_first1,
  InputIterator4 values_first2,
  OutputIterator1 keys_result,
  OutputIterator2 values_result);
```

set_difference_by_key 执行集合的键值差运算。set_difference_by_key 构造一个有序范围，是有序范围 [keys_first1, keys_last1) 和 [keys_first2, keys_last2) 的差集。与输入和输出键范围内的每个元素相关联的是一个值元素。关联的输入值范围不需要排序。

在最简单的情况下，set_difference_by_key 执行集合的键值差运算：键输出范围由 属于 [keys_first1, keys_last1) 而不属于 [keys_first2, keys_last2) 的元素副本组成。一般情况更为复杂，因为输入范围内可能包含重复元素。总而言之，如果 [keys_first1, keys_last1) 包含 m 个彼此等价的元素，且 [keys_first2, keys_last2) 包含 n 个与它们等价的元素，则 [keys_first1, keys_last1) 内最后 max(m-n,0) 个元素会被复制到输出范围。

每次将键元素从 [keys_first1, keys_last1) 或 [keys_first2, keys_last2) 复制到键输出范围时，将相应的值元素从相应的值输入范围（从 values_first1 或 values_first2 开始）复制到值输出范围。

此版本的 set_difference_by_key 使用 operator< 比较键元素。

以下代码片段演示了如何使用 set_difference_by_key 计算两组升序整数集合与其值的差集：

MetaX Confidential 2023-11-16 15:00:00

```
#include <thrust/set_operations.h>
...
int A_keys[6] = {0, 1, 3, 4, 5, 6, 9};
int A_vals[6] = {0, 0, 0, 0, 0, 0, 0};

int B_keys[5] = {1, 3, 5, 7, 9};
int B_vals[5] = {1, 1, 1, 1, 1};

int keys_result[3];
int vals_result[3];

thrust::pair<int*,int*> end = thrust::set_difference_by_key(A_keys, A_keys
↪+ 6, B_keys, B_keys + 5, A_vals, B_vals, keys_result, vals_result);
// keys_result is now {0, 4, 6}
// vals_result is now {0, 0, 0}
```

模板参数

- **InputIterator1**：是 Input Iterator 的模型。InputIterator1 和 InputIterator2 具有相同的 value_type，InputIterator1 的 value_type 是 LessThan Comparable 的模型，如 LessThan Comparable 要求中定义的，InputIterator1 的 value_type 上的排序是严格弱序，并且可转换为 OutputIterator 的 value_types 集合中的类型
- **InputIterator2**：是 Input Iterator 的模型。InputIterator2 和 InputIterator1 具有相同的 value_type，InputIterator2 的 value_type 是 LessThan Comparable 的模型，如 LessThan Comparable 要求中定义的，InputIterator2 的 value_type 上的排序是严格弱序，并且可转换为 OutputIterator 的 value_types 集合中的类型
- **InputIterator3**：是 Input Iterator 的模型，InputIterator3 的 value_type 可转换为 OutputIterator2 的 value_types 集合中的类型
- **InputIterator4**：是 Input Iterator 的模型，InputIterator4 的 value_type 可转换为 OutputIterator2 的 value_types 集合中的类型
- **OutputIterator1**：是 Output Iterator 的模型
- **OutputIterator2**：是 Output Iterator 的模型

函数参数

- **keys_first1**：第一个键输入范围的起始
- **keys_last1**：第一个键输入范围的末尾
- **keys_first2**：第二个键输入范围的起始
- **keys_last2**：第二个键输入范围的末尾
- **values_first1**：第一个值输入范围的起始
- **values_first2**：第二个值输入范围的起始
- **keys_result**：键输出范围的起始
- **values_result**：值输出范围的起始

前提条件

- 范围 [keys_first1, keys_last1) 和 [keys_first2, keys_last2) 应按照 operator< 进行排序
- 结果范围不应与任何一个输入范围重叠

返回值

p pair，其中 p.first 是键输出范围的末尾，p.second 是值输出范围的末尾

另请参阅

- set_union_by_key

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

- set_intersection_by_key
- set_symmetric_difference_by_key
- sort_by_key
- is_sorted

2.1.7.19 thrust::set_difference_by_key

```
template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2,
  typename InputIterator3,
  typename InputIterator4,
  typename OutputIterator1,
  typename OutputIterator2,
  typename StrictWeakCompare>
__host__ __device__ thrust::pair< OutputIterator1, OutputIterator2 >
set_difference_by_key(const thrust::detail::execution_policy_base<
↪DerivedPolicy > & exec,
  InputIterator1 keys_first1,
  InputIterator1 keys_last1,
  InputIterator2 keys_first2,
  InputIterator2 keys_last2,
  InputIterator3 values_first1,
  InputIterator4 values_first2,
  OutputIterator1 keys_result,
  OutputIterator2 values_result,
  StrictWeakCompare comp);
```

set_difference_by_key 执行集合的键值差运算。set_difference_by_key 构造一个有序范围，是有序范围 [keys_first1, keys_last1) 和 [keys_first2, keys_last2) 的差集。与输入和输出键范围内的每个元素相关联的是一个值元素。关联的输入值范围不需要排序。

在最简单的情况下，set_difference_by_key 执行集合的键值差运算：键输出范围由 属于 [keys_first1, keys_last1) 而不属于 [keys_first2, keys_last2) 的元素副本组成。一般情况更为复杂，因为输入范围内可能包含重复元素。总而言之，如果 [keys_first1, keys_last1) 包含 m 个彼此等价的元素，且 [keys_first2, keys_last2) 包含 n 个与它们等价的元素，则 [keys_first1, keys_last1) 内最后 max(m-n,0) 个元素会被复制到输出范围。

每次将键元素从 [keys_first1, keys_last1) 或 [keys_first2, keys_last2) 复制到键输出范围时，将相应的值元素从相应的值输入范围（从 values_first1 或 values_first2 开始）复制到值输出范围。

此版本的 set_difference_by_key 使用函数对象 comp 比较键元素。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::host 并行化执行策略，使用 set_difference_by_key 计算两组降序整数集合与其值的差集：

```
#include <thrust/set_operations.h>
#include <thrust/functional.h>
#include <thrust/execution_policy.h>
...
int A_keys[6] = {9, 6, 5, 4, 3, 1, 0};
int A_vals[6] = {0, 0, 0, 0, 0, 0, 0};

int B_keys[5] = {9, 7, 5, 3, 1};
int B_vals[5] = {1, 1, 1, 1, 1};
```

(下页继续)

沐曦 MetaX Confidential 2024-11-15 15:00:00

(续上页)

```
int keys_result[3];
int vals_result[3];

thrust::pair<int*,int*> end = thrust::set_difference_by_key(thrust::host,
↪A_keys, A_keys + 6, B_keys, B_keys + 5, A_vals, B_vals, keys_result,
↪vals_result, thrust::greater<int>());
// keys_result is now {0, 4, 6}
// vals_result is now {0, 0, 0}
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **InputIterator1**：是 Input Iterator 的模型。InputIterator1 和 InputIterator2 具有相同的 value_type，InputIterator1 的 value_type 是 LessThan Comparable 的模型，如 LessThan Comparable 要求中定义的，InputIterator1 的 value_type 上的排序是严格弱序，并且可转换为 OutputIterator 的 value_types 集合中的类型
- **InputIterator2**：是 Input Iterator 的模型。InputIterator2 和 InputIterator1 具有相同的 value_type，InputIterator2 的 value_type 是 LessThan Comparable 的模型，如 LessThan Comparable 要求中定义的，InputIterator2 的 value_type 上的排序是严格弱序，并且可转换为 OutputIterator 的 value_types 集合中的类型
- **InputIterator3**：是 Input Iterator 的模型，InputIterator3 的 value_type 可转换为 OutputIterator2 的 value_types 集合中的类型
- **InputIterator4**：是 Input Iterator 的模型，InputIterator4 的 value_type 可转换为 OutputIterator2 的 value_types 集合中的类型
- **OutputIterator1**：是 Output Iterator 的模型
- **OutputIterator2**：是 Output Iterator 的模型
- **StrictWeakCompare**：是 Strict Weak Ordering 的模型

函数参数

- **exec**：用于并行化的执行策略
- **keys_first1**：第一个键输入范围的起始
- **keys_last1**：第一个键输入范围的末尾
- **keys_first2**：第二个键输入范围的起始
- **keys_last2**：第二个键输入范围的末尾
- **values_first1**：第一个值输入范围的起始
- **values_first2**：第二个值输入范围的起始
- **keys_result**：键输出范围的起始
- **values_result**：值输出范围的起始
- **comp**：比较运算符

前提条件

- 范围 [keys_first1, keys_last1) 和 [keys_first2, keys_last2) 应按照 comp 进行排序
- 结果范围不应与任何一个输入范围重叠

返回值

p pair，其中 p.first 是键输出范围的末尾，p.second 是值输出范围的末尾

另请参阅

飞猫信息 MetaX Confidential 2024-11-18 15:00:01

- set_union_by_key
- set_intersection_by_key
- set_symmetric_difference_by_key
- sort_by_key
- is_sorted

2.1.7.20 thrust::set_difference_by_key

```
template <typename InputIterator1,
  typename InputIterator2,
  typename InputIterator3,
  typename InputIterator4,
  typename OutputIterator1,
  typename OutputIterator2,
  typename StrictWeakCompare>
thrust::pair< OutputIterator1, OutputIterator2 >
set_difference_by_key(InputIterator1 keys_first1,
  InputIterator1 keys_last1,
  InputIterator2 keys_first2,
  InputIterator2 keys_last2,
  InputIterator3 values_first1,
  InputIterator4 values_first2,
  OutputIterator1 keys_result,
  OutputIterator2 values_result,
  StrictWeakCompare comp);
```

set_difference_by_key 执行集合的键值差运算。set_difference_by_key 构造一个有序范围，是有序范围 [keys_first1, keys_last1) 和 [keys_first2, keys_last2) 的差集。与输入和输出键范围内的每个元素相关联的是一个值元素。关联的输入值范围不需要排序。

在最简单的情况下，set_difference_by_key 执行集合的键值差运算：键输出范围由 属于 [keys_first1, keys_last1) 而不属于 [keys_first2, keys_last2) 的元素副本组成。一般情况更为复杂，因为输入范围内可能包含重复元素。总而言之，如果 [keys_first1, keys_last1) 包含 m 个彼此等价的元素，且 [keys_first2, keys_last2) 包含 n 个与它们等价的元素，则 [keys_first1, keys_last1) 内最后 max(m-n,0) 个元素会被复制到输出范围。

每次将键元素从 [keys_first1, keys_last1) 或 [keys_first2, keys_last2) 复制到键输出范围时，将相应的值元素从相应的值输入范围（从 values_first1 或 values_first2 开始）复制到值输出范围。

此版本的 set_difference_by_key 使用函数对象 comp 比较键元素。

以下代码片段演示了如何使用 set_difference_by_key 计算两组降序整数集合与其值的差集：

```
#include <thrust/set_operations.h>
#include <thrust/functional.h>
...
int A_keys[6] = {9, 6, 5, 4, 3, 1, 0};
int A_vals[6] = {0, 0, 0, 0, 0, 0, 0};

int B_keys[5] = {9, 7, 5, 3, 1};
int B_vals[5] = {1, 1, 1, 1, 1};

int keys_result[3];
int vals_result[3];

thrust::pair<int*,int*> end = thrust::set_difference_by_key(A_keys, A_keys
↪+ 6, B_keys, B_keys + 5, A_vals, B_vals, keys_result, vals_result,
↪thrust::greater<int>());
```

(下页继续)

沐曦 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
// keys_result is now {0, 4, 6}
// vals_result is now {0, 0, 0}
```

模板参数

- **InputIterator1**：是 Input Iterator 的模型。InputIterator1 和 InputIterator2 具有相同的 value_type，InputIterator1 的 value_type 是 LessThan Comparable 的模型，如 LessThan Comparable 要求中定义的，InputIterator1 的 value_type 上的排序是严格弱序，并且可转换为 OutputIterator 的 value_types 集合中的类型
- **InputIterator2**：是 Input Iterator 的模型。InputIterator2 和 InputIterator1 具有相同的 value_type，InputIterator2 的 value_type 是 LessThan Comparable 的模型，如 LessThan Comparable 要求中定义的，InputIterator2 的 value_type 上的排序是严格弱序，并且可转换为 OutputIterator 的 value_types 集合中的类型
- **InputIterator3**：是 Input Iterator 的模型，InputIterator3 的 value_type 可转换为 OutputIterator2 的 value_types 集合中的类型
- **InputIterator4**：是 Input Iterator 的模型，InputIterator4 的 value_type 可转换为 OutputIterator2 的 value_types 集合中的类型
- **OutputIterator1**：是 Output Iterator 的模型
- **OutputIterator2**：是 Output Iterator 的模型
- **StrictWeakCompare**：是 Strict Weak Ordering 的模型

函数参数

- **keys_first1**：第一个键输入范围的起始
- **keys_last1**：第一个键输入范围的末尾
- **keys_first2**：第二个键输入范围的起始
- **keys_last2**：第二个键输入范围的末尾
- **values_first1**：第一个值输入范围的起始
- **values_first2**：第二个值输入范围的起始
- **keys_result**：键输出范围的起始
- **values_result**：值输出范围的起始
- **comp**：比较运算符

前提条件

- 范围 [keys_first1, keys_last1) 和 [keys_first2, keys_last2) 应按照 comp 进行排序
- 结果范围不应与任何一个输入范围重叠

返回值

p pair，其中 p.first 是键输出范围的末尾，p.second 是值输出范围的末尾

另请参阅

- set_union_by_key
- set_intersection_by_key
- set_symmetric_difference_by_key
- sort_by_key
- is_sorted

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

2.1.7.21 thrust::set_intersection_by_key

```
template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2,
  typename InputIterator3,
  typename OutputIterator1,
  typename OutputIterator2>
__host__ __device__ thrust::pair< OutputIterator1, OutputIterator2 >
set_intersection_by_key(const thrust::detail::execution_policy_base<
 ↪DerivedPolicy > & exec,
  InputIterator1 keys_first1,
  InputIterator1 keys_last1,
  InputIterator2 keys_first2,
  InputIterator2 keys_last2,
  InputIterator3 values_first1,
  OutputIterator1 keys_result,
  OutputIterator2 values_result);
```

set_intersection_by_key 执行集合的键值交运算。set_intersection_by_key 构造一个有序范围，是有序范围 [keys_first1, keys_last1) 和 [keys_first2, keys_last2) 的交集。与输入和输出键范围内的每个元素相关联的是一个值元素。关联的输入值范围不需要排序。

在最简单的情况下，set_intersection_by_key 执行集合的键值交运算：键输出范围由属于 [keys_first1, keys_last1) 和 [keys_first2, keys_last2) 的元素副本组成。一般情况更为复杂，因为输入范围内可能包含重复元素。总而言之，如果一个元素在 [keys_first1, keys_last1) 内出现 m 次，在 [keys_first2, keys_last2) 内出现 n 次（其中 m 可能为零），则它在键输出范围中会出现 min(m,n) 次。set_intersection_by_key 是稳定的，指的是元素都是从第一个输入范围而不是第二个范围复制的，并且输出范围中元素的相对顺序与第一个输入范围中的相同。

每次将键元素从 [keys_first1, keys_last1) 复制到键输出范围时，将相应的值元素从 [values_first1, values_last1) 复制到值输出范围。

此版本的 set_intersection_by_key 使用 operator< 比较对象。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::host 并行化执行策略，使用 set_intersection_by_key 计算两组升序整数集合与其值的交集：

```
#include <thrust/set_operations.h>
#include <thrust/execution_policy.h>
...
int A_keys[6] = {1, 3, 5, 7, 9, 11};
int A_vals[6] = {0, 0, 0, 0, 0,  0};

int B_keys[7] = {1, 1, 2, 3, 5,  8, 13};

int keys_result[7];
int vals_result[7];

thrust::pair<int*,int*> end =
  thrust::set_intersection_by_key(thrust::host,
            A_keys,
            A_keys + 6,
            B_keys,
            B_keys + 7,
            A_vals,
            keys_result,
            vals_result);
```

(下页继续)

飞腾信息 沐曦 Confidential 2024-11-18 15:16:08

(续上页)

```
// keys_result is now {1, 3, 5}
// vals_result is now {0, 0, 0}
```

备注

与其他键值集合运算不同，set_intersection_by_key 的唯一性在于它没有 values_first2 参数，因为第二个输入范围中的元素从未复制到输出范围。

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **InputIterator1**：是 Input Iterator 的模型。InputIterator1 和 InputIterator2 具有相同的 value_type，InputIterator1 的 value_type 是 LessThan Comparable 的模型，如 LessThan Comparable 要求中定义的，InputIterator1 的 value_type 上的排序是严格弱序，并且可转换为 OutputIterator 的 value_types 集合中的类型
- **InputIterator2**：是 Input Iterator 的模型。InputIterator2 和 InputIterator1 具有相同的 value_type，InputIterator2 的 value_type 是 LessThan Comparable 的模型，如 LessThan Comparable 要求中定义的，InputIterator2 的 value_type 上的排序是严格弱序，并且可转换为 OutputIterator 的 value_types 集合中的类型
- **InputIterator3**：是 Input Iterator 的模型，InputIterator3 的 value_type 可转换为 OutputIterator2 的 value_types 集合中的类型
- **OutputIterator1**：是 Output Iterator 的模型
- **OutputIterator2**：是 Output Iterator 的模型

函数参数

- **exec**：用于并行化的执行策略
- **keys_first1**：第一个键输入范围的起始
- **keys_last1**：第一个键输入范围的末尾
- **keys_first2**：第二个键输入范围的起始
- **keys_last2**：第二个键输入范围的末尾
- **values_first1**：第一个值输入范围的起始
- **keys_result**：键输出范围的起始
- **values_result**：值输出范围的起始

前提条件

- 范围 [keys_first1, keys_last1) 和 [keys_first2, keys_last2) 应按照 operator< 进行排序
- 结果范围不应与任何一个输入范围重叠

返回值

p pair，其中 p.first 是键输出范围的末尾，p.second 是值输出范围的末尾

另请参阅

- set_union_by_key
- set_difference_by_key
- set_symmetric_difference_by_key
- sort_by_key
- is_sorted

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

2.1.7.22 thrust::set_intersection_by_key

```
template <typename InputIterator1,
  typename InputIterator2,
  typename InputIterator3,
  typename OutputIterator1,
  typename OutputIterator2>
thrust::pair< OutputIterator1, OutputIterator2 >
set_intersection_by_key(InputIterator1 keys_first1,
  InputIterator1 keys_last1,
  InputIterator2 keys_first2,
  InputIterator2 keys_last2,
  InputIterator3 values_first1,
  OutputIterator1 keys_result,
  OutputIterator2 values_result);
```

set_intersection_by_key 执行集合的键值交运算。set_intersection_by_key 构造一个有序范围，是有序范围 [keys_first1, keys_last1) 和 [keys_first2, keys_last2) 的交集。与输入和输出键范围内的每个元素相关联的是一个值元素。关联的输入值范围不需要排序。

在最简单的情况下，set_intersection_by_key 执行集合的交运算：键输出范围由属于 [keys_first1, keys_last1) 和 [keys_first2, keys_last2) 的元素副本组成。一般情况更为复杂，因为输入范围内可能包含重复元素。总而言之，如果一个元素在 [keys_first1, keys_last1) 内出现 m 次，在 [keys_first2, keys_last2) 内出现 n 次（其中 m 可能为零），则它在键输出范围中会出现 min(m,n) 次。set_intersection_by_key 是稳定的，指的是元素都是从第一个输入范围而不是第二个范围复制的，并且输出范围中元素的相对顺序与第一个输入范围中的相同。

每次将键元素从 [keys_first1, keys_last1) 复制到键输出范围时，将相应的值元素从 [values_first1, values_last1) 复制到值输出范围。

此版本的 set_intersection_by_key 使用 operator< 比较对象。

以下代码片段演示了如何使用 set_intersection_by_key 计算两组升序整数集合与其值的交集：

```
#include <thrust/set_operations.h>
...
int A_keys[6] = {1, 3, 5, 7, 9, 11};
int A_vals[6] = {0, 0, 0, 0, 0,  0};

int B_keys[7] = {1, 1, 2, 3, 5,  8, 13};

int keys_result[7];
int vals_result[7];

thrust::pair<int*,int*> end =
  thrust::set_intersection_by_key(A_keys,
            A_keys + 6,
            B_keys,
            B_keys + 7,
            A_vals,
            keys_result,
            vals_result);

// keys_result is now {1, 3, 5}
// vals_result is now {0, 0, 0}
```

备注

与其他键值集合运算不同，set_intersection_by_key 的唯一性在于它没有 values_first2 参数，因为第二个输入范围中的元素从未复制到输出范围。

飞腾信息 Mettax Confidential 2024-11-18 15:00:00

模板参数

- **InputIterator1**：是 Input Iterator 的模型。InputIterator1 和 InputIterator2 具有相同的 value_type，InputIterator1 的 value_type 是 LessThan Comparable 的模型，如 LessThan Comparable 要求中定义的，InputIterator1 的 value_type 上的排序是严格弱序，并且可转换为 OutputIterator 的 value_types 集合中的类型
- **InputIterator2**：是 Input Iterator 的模型。InputIterator2 和 InputIterator1 具有相同的 value_type，InputIterator2 的 value_type 是 LessThan Comparable 的模型，如 LessThan Comparable 要求中定义的，InputIterator2 的 value_type 上的排序是严格弱序，并且可转换为 OutputIterator 的 value_types 集合中的类型
- **InputIterator3**：是 Input Iterator 的模型，InputIterator3 的 value_type 可转换为 OutputIterator2 的 value_types 集合中的类型
- **OutputIterator1**：是 Output Iterator 的模型
- **OutputIterator2**：是 Output Iterator 的模型

函数参数

- **keys_first1**：第一个键输入范围的起始
- **keys_last1**：第一个键输入范围的末尾
- **keys_first2**：第二个键输入范围的起始
- **keys_last2**：第二个键输入范围的末尾
- **values_first1**：第一个值输入范围的起始
- **keys_result**：键输出范围的起始
- **values_result**：值输出范围的起始

前提条件

- 范围 [keys_first1, keys_last1) 和 [keys_first2, keys_last2) 应按照 operator< 进行排序
- 结果范围不应与任何一个输入范围重叠

返回值

p pair，其中 p.first 是键输出范围的末尾，p.second 是值输出范围的末尾

另请参阅

- set_union_by_key
- set_difference_by_key
- set_symmetric_difference_by_key
- sort_by_key
- is_sorted

2.1.7.23 thrust::set_intersection_by_key

```
template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2,
  typename InputIterator3,
  typename OutputIterator1,
  typename OutputIterator2,
  typename StrictWeakCompare>
__host__ __device__ thrust::pair< OutputIterator1, OutputIterator2 >
set_intersection_by_key(const thrust::detail::execution_policy_base<
↪DerivedPolicy > & exec,
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
  InputIterator1 keys_first1,
  InputIterator1 keys_last1,
  InputIterator2 keys_first2,
  InputIterator2 keys_last2,
  InputIterator3 values_first1,
  OutputIterator1 keys_result,
  OutputIterator2 values_result,
  StrictWeakCompare comp);
```

set_intersection_by_key 执行集合的键值交运算。set_intersection_by_key 构造一个有序范围，是有序范围 [keys_first1, keys_last1) 和 [keys_first2, keys_last2) 的交集。与输入和输出键范围内的每个元素相关联的是一个值元素。关联的输入值范围不需要排序。

在最简单的情况下，set_intersection_by_key 执行集合的键值交运算：键输出范围由属于 [keys_first1, keys_last1) 和 [keys_first2, keys_last2) 的元素副本组成。一般情况更为复杂，因为输入范围内可能包含重复元素。总而言之，如果一个元素在 [keys_first1, keys_last1) 内出现 m 次，在 [keys_first2, keys_last2) 内出现 n 次（其中 m 可能为零），则它在键输出范围中会出现 min(m,n) 次。set_intersection_by_key 是稳定的，指的是元素都是从第一个输入范围而不是第二个范围复制的，并且输出范围中元素的相对顺序与第一个输入范围中的相同。

每次将键元素从 [keys_first1, keys_last1) 复制到键输出范围时，将相应的值元素从 [values_first1, values_last1) 复制到值输出范围。

此版本的 set_intersection_by_key 使用函数对象 comp 比较对象。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::host 并行化执行策略，使用 set_intersection_by_key 计算两组降序整数集合与其值的交集：

```
#include <thrust/set_operations.h>
#include <thrust/functional.h>
#include <thrust/execution_policy.h>
...
int A_keys[6] = {11, 9, 7, 5, 3, 1};
int A_vals[6] = { 0, 0, 0, 0, 0, 0};

int B_keys[7] = {13, 8, 5, 3, 2, 1, 1};

int keys_result[7];
int vals_result[7];

thrust::pair<int*,int*> end = thrust::set_intersection_by_key(thrust::host,
↪ A_keys, A_keys + 6, B_keys, B_keys + 7, A_vals, keys_result, vals_
↪result, thrust::greater<int>());

// keys_result is now {5, 3, 1}
// vals_result is now {0, 0, 0}
```

备注

与其他键值集合运算不同，set_intersection_by_key 的唯一性在于它没有 values_first2 参数，因为第二个输入范围中的元素从未复制到输出范围。

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **InputIterator1**：是 Input Iterator 的模型。InputIterator1 和 InputIterator2 具有相同的 value_type，InputIterator1 的 value_type 是 LessThan Comparable 的模型，如 LessThan Comparable 要求中定义的，InputIterator1 的 value_type 上的排序是严格弱序，并且可转换为 OutputIterator 的 value_types 集合中的类型

飞腾信息 MetaX Confidential 2025-11-18 15:00:00

- **InputIterator2**：是 Input Iterator 的模型。InputIterator2 和 InputIterator1 具有相同的 value_type，InputIterator2 的 value_type 是 LessThan Comparable 的模型，如 LessThan Comparable 要求中定义的，InputIterator2 的 value_type 上的排序是严格弱序，并且可转换为 OutputIterator 的 value_types 集合中的类型
- **InputIterator3**：是 Input Iterator 的模型，InputIterator3 的 value_type 可转换为 OutputIterator2 的 value_types 集合中的类型
- **OutputIterator1**：是 Output Iterator 的模型
- **OutputIterator2**：是 Output Iterator 的模型
- **StrictWeakCompare**：是 Strict Weak Ordering 的模型

函数参数

- **exec**：用于并行化的执行策略
- **keys_first1**：第一个键输入范围的起始
- **keys_last1**：第一个键输入范围的末尾
- **keys_first2**：第二个键输入范围的起始
- **keys_last2**：第二个键输入范围的末尾
- **values_first1**：第一个值输入范围的起始
- **keys_result**：键输出范围的起始
- **values_result**：值输出范围的起始
- **comp**：比较运算符

前提条件

- 范围 [keys_first1, keys_last1) 和 [keys_first2, keys_last2) 应按照 comp 进行排序
- 结果范围不应与任何一个输入范围重叠

返回值

p pair，其中 p.first 是键输出范围的末尾，p.second 是值输出范围的末尾

另请参阅

- set_union_by_key
- set_difference_by_key
- set_symmetric_difference_by_key
- sort_by_key
- is_sorted

2.1.7.24 thrust::set_intersection_by_key

```
template <typename InputIterator1,
  typename InputIterator2,
  typename InputIterator3,
  typename OutputIterator1,
  typename OutputIterator2,
  typename StrictWeakCompare>
thrust::pair< OutputIterator1, OutputIterator2 >
set_intersection_by_key(InputIterator1 keys_first1,
  InputIterator1 keys_last1,
  InputIterator2 keys_first2,
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
  InputIterator2 keys_last2,
  InputIterator3 values_first1,
  OutputIterator1 keys_result,
  OutputIterator2 values_result,
  StrictWeakCompare comp);
```

set_intersection_by_key 执行集合的键值交运算。set_intersection_by_key 构造一个有序范围，是有序范围 [keys_first1, keys_last1) 和 [keys_first2, keys_last2) 的交集。与输入和输出键范围内的每个元素相关联的是一个值元素。关联的输入值范围不需要排序。

在最简单的情况下，set_intersection_by_key 执行集合的键值交运算：键输出范围由属于 [keys_first1, keys_last1) 和 [keys_first2, keys_last2) 的元素副本组成。一般情况更为复杂，因为输入范围内可能包含重复元素。总而言之，如果一个元素在 [keys_first1, keys_last1) 内出现 m 次，在 [keys_first2, keys_last2) 内出现 n 次（其中 m 可能为零），则它在键输出范围中会出现 min(m,n) 次。set_intersection_by_key 是稳定的，指的是元素都是从第一个输入范围而不是第二个范围复制的，并且输出范围中元素的相对顺序与第一个输入范围中的相同。

每次将键元素从 [keys_first1, keys_last1) 复制到键输出范围时，将相应的值元素从 [values_first1, values_last1) 复制到值输出范围。

此版本的 set_intersection_by_key 使用函数对象 comp 比较对象。

以下代码片段演示了如何使用 set_intersection_by_key 两组降序整数集合与其值的交集：

```
#include <thrust/set_operations.h>
#include <thrust/functional.h>
...
int A_keys[6] = {11, 9, 7, 5, 3, 1};
int A_vals[6] = { 0, 0, 0, 0, 0, 0};

int B_keys[7] = {13, 8, 5, 3, 2, 1, 1};

int keys_result[7];
int vals_result[7];

thrust::pair<int*,int*> end = thrust::set_intersection_by_key(A_keys, A_
↪keys + 6, B_keys, B_keys + 7, A_vals, keys_result, vals_result,
↪thrust::greater<int>());

// keys_result is now {5, 3, 1}
// vals_result is now {0, 0, 0}
```

备注

与其他键值集合运算不同，set_intersection_by_key 的唯一性在于它没有 values_first2 参数，因为第二个输入范围中的元素从未复制到输出范围。

模板参数

- **InputIterator1**：是 Input Iterator 的模型。InputIterator1 和 InputIterator2 具有相同的 value_type，InputIterator1 的 value_type 是 LessThan Comparable 的模型，如 LessThan Comparable 要求中定义的，InputIterator1 的 value_type 上的排序是严格弱序，并且可转换为 OutputIterator 的 value_types 集合中的类型
- **InputIterator2**：是 Input Iterator 的模型。InputIterator2 和 InputIterator1 具有相同的 value_type，InputIterator2 的 value_type 是 LessThan Comparable 的模型，如 LessThan Comparable 要求中定义的，InputIterator2 的 value_type 上的排序是严格弱序，并且可转换为 OutputIterator 的 value_types 集合中的类型
- **InputIterator3**：是 Input Iterator 的模型，InputIterator3 的 value_type 可转换为 OutputIterator2 的 value_types 集合中的类型

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

- **OutputIterator1**：是 Output Iterator 的模型
- **OutputIterator2**：是 Output Iterator 的模型
- **StrictWeakCompare**：是 Strict Weak Ordering 的模型

函数参数

- **keys_first1**：第一个键输入范围的起始
- **keys_last1**：第一个键输入范围的末尾
- **keys_first2**：第二个键输入范围的起始
- **keys_last2**：第二个键输入范围的末尾
- **values_first1**：第一个值输入范围的起始
- **keys_result**：键输出范围的起始
- **values_result**：值输出范围的起始
- **comp**：比较运算符

前提条件

- 范围 [keys_first1, keys_last1) 和 [keys_first2, keys_last2) 应按照 comp 进行排序
- 结果范围不应与任何一个输入范围重叠

返回值

p pair，其中 p.first 是键输出范围的末尾，p.second 是值输出范围的末尾

另请参阅

- set_union_by_key
- set_difference_by_key
- set_symmetric_difference_by_key
- sort_by_key
- is_sorted

2.1.7.25 thrust::set_symmetric_difference_by_key

```
template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2,
  typename InputIterator3,
  typename InputIterator4,
  typename OutputIterator1,
  typename OutputIterator2>
__host__ __device__ thrust::pair< OutputIterator1, OutputIterator2 >
set_symmetric_difference_by_key(const thrust::detail::execution_policy_base
↪< DerivedPolicy > & exec,
  InputIterator1 keys_first1,
  InputIterator1 keys_last1,
  InputIterator2 keys_first2,
  InputIterator2 keys_last2,
  InputIterator3 values_first1,
  InputIterator4 values_first2,
  OutputIterator1 keys_result,
  OutputIterator2 values_result);
```

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

set_symmetric_difference_by_key 执行集合的键值对称差运算。set_difference_by_key 构造一个有序范围，是有序范围 [keys_first1, keys_last1) 和 [keys_first2, keys_last2) 的对称差集。与输入和输出键范围内的每个元素相关联的是一个值元素。关联的输入值范围不需要排序。

在最简单的情况下，set_symmetric_difference_by_key 执行以下集合运算：构造两个集合 A - B 和 B - A 的并集，其中 A 和 B 是两个输入范围。即，输出范围由属于 [keys_first1, keys_last1) 但不属于 [keys_first2, keys_last1) 的元素副本，和属于 [keys_first2, keys_last2) 但不属于 [keys_first1, keys_last1) 的元素副本组成。一般情况更为复杂，因为输入范围内可能包含重复元素。总而言之，如果 [keys_first1, keys_last1) 包含 m 个彼此等价的元素，[keys_first2, keys_last1) 包含 n 个彼此等价的元素，则 |m - n| 个这样的元素应被复制到输出范围：如果 m > n ，则复制 [keys_first1, keys_last1) 内最后 m - n 个元素；如果 m < n ，则复制 [keys_first2, keys_last2) 中最后 n - m 个元素。

每次将键元素从 [keys_first1, keys_last1) 或 [keys_first2, keys_last2) 复制到键输出范围时，将相应的值元素从相应的值输入范围（从 values_first1 或 values_first2 开始）复制到值输出范围。此版本的 set_symmetric_difference_by_key 使用 operator< 比较键元素。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::host 并行化执行策略，使用 set_symmetric_difference_by_key 计算两组升序整数集合与其值的对称差集：

```
#include <thrust/set_operations.h>
#include <thrust/execution_policy.h>
...
int A_keys[6] = {0, 1, 2, 2, 4, 6, 7};
int A_vals[6] = {0, 0, 0, 0, 0, 0, 0};

int B_keys[5] = {1, 1, 2, 5, 8};
int B_vals[5] = {1, 1, 1, 1, 1};

int keys_result[6];
int vals_result[6];

thrust::pair<int*,int*> end = thrust::set_symmetric_difference_by_
↪key(thrust::host, A_keys, A_keys + 6, B_keys, B_keys + 5, A_vals, B_vals,
↪ keys_result, vals_result);
// keys_result is now {0, 4, 5, 6, 7, 8}
// vals_result is now {0, 0, 1, 0, 0, 1}
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **InputIterator1**：是 Input Iterator 的模型。InputIterator1 和 InputIterator2 具有相同的 value_type ，InputIterator1 的 value_type 是 LessThan Comparable 的模型，如 LessThan Comparable 要求中定义的，InputIterator1 的 value_type 上的排序是严格弱序，并且可转换为 OutputIterator 的 value_types 集合中的类型
- **InputIterator2**：是 Input Iterator 的模型。InputIterator2 和 InputIterator1 具有相同的 value_type ，InputIterator2 的 value_type 是 LessThan Comparable 的模型，如 LessThan Comparable 要求中定义的，InputIterator2 的 value_type 上的排序是严格弱序，并且可转换为 OutputIterator 的 value_types 集合中的类型
- **InputIterator3**：是 Input Iterator 的模型，InputIterator3 的 value_type 可转换为 OutputIterator2 的 value_types 集合中的类型
- **InputIterator4**：是 Input Iterator 的模型，InputIterator4 的 value_type 可转换为 OutputIterator2 的 value_types 集合中的类型
- **OutputIterator1**：是 Output Iterator 的模型
- **OutputIterator2**：是 Output Iterator 的模型

函数参数

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

- **exec**：用于并行化的执行策略
- **keys_first1**：第一个键输入范围的起始
- **keys_last1**：第一个键输入范围的末尾
- **keys_first2**：第二个键输入范围的起始
- **keys_last2**：第二个键输入范围的末尾
- **values_first1**：第一个值输入范围的起始
- **values_first2**：第二个值输入范围的起始
- **keys_result**：键输出范围的起始
- **values_result**：值输出范围的起始

前提条件

- 范围 [keys_first1, keys_last1) 和 [keys_first2, keys_last2) 应按照 operator< 进行排序
- 结果范围不应与任何一个输入范围重叠

返回值

p pair ，其中 p.first 是键输出范围的末尾，p.second 是值输出范围的末尾

另请参阅

- set_union_by_key
- set_intersection_by_key
- set_difference_by_key
- sort_by_key
- is_sorted

2.1.7.26 thrust::set_symmetric_difference_by_key

```
template <typename InputIterator1,
  typename InputIterator2,
  typename InputIterator3,
  typename InputIterator4,
  typename OutputIterator1,
  typename OutputIterator2>
thrust::pair< OutputIterator1, OutputIterator2 >
set_symmetric_difference_by_key(InputIterator1 keys_first1,
  InputIterator1 keys_last1,
  InputIterator2 keys_first2,
  InputIterator2 keys_last2,
  InputIterator3 values_first1,
  InputIterator4 values_first2,
  OutputIterator1 keys_result,
  OutputIterator2 values_result);
```

set_symmetric_difference_by_key 执行集合的键值对称差运算。set_difference_by_key 构造一个有序范围，是有序范围 [keys_first1, keys_last1) 和 [keys_first2, keys_last2) 的对称差集。与输入和输出键范围内的每个元素相关联的是一个值元素。关联的输入值范围不需要排序。

在最简单的情况下，set_symmetric_difference_by_key 执行以下集合运算：构造两个集合 A - B 和 B - A 的并集，其中 A 和 B 是两个输入范围。即，输出范围由属于 [keys_first1, keys_last1) 但不属于 [keys_first2, keys_last1) 的元素副本，和属于 [keys_first2, keys_last2) 但不属于 [keys_first1, keys_last1) 的元素副本组成。一般情况更为复杂，因为输入范围内可能包含重复元素。总而言之，如果 [keys_first1, keys_last1) 包含 m 个彼此等价的元素，[keys_first2, keys_last1) 包含 n 个彼此等价的元素，则 |m - n|

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

个这样的元素应被复制到输出范围：如果 m > n ，则复制 [keys_first1, keys_last1) 内最后 m - n 个元素；如果 m < n ，则复制 [keys_first2, keys_last2) 中最后 n - m 个元素。

每次将键元素从 [keys_first1, keys_last1) 或 [keys_first2, keys_last2) 复制到键输出范围时，将相应的值元素从相应的值输入范围（从 values_first1 或 values_first2 开始）复制到值输出范围。

此版本的 set_symmetric_difference_by_key 使用 operator< 比较键元素。

以下代码片段演示了如何使用 set_symmetric_difference_by_key 计算两组升序整数集合与其值的对称差集：

```
#include <thrust/set_operations.h>
...
int A_keys[6] = {0, 1, 2, 2, 4, 6, 7};
int A_vals[6] = {0, 0, 0, 0, 0, 0, 0};

int B_keys[5] = {1, 1, 2, 5, 8};
int B_vals[5] = {1, 1, 1, 1, 1};

int keys_result[6];
int vals_result[6];

thrust::pair<int*,int*> end = thrust::set_symmetric_difference_by_key(A_
↪keys, A_keys + 6, B_keys, B_keys + 5, A_vals, B_vals, keys_result, vals_
↪result);
// keys_result is now {0, 4, 5, 6, 7, 8}
// vals_result is now {0, 0, 1, 0, 0, 1}
```

模板参数

- **InputIterator1**：是 Input Iterator 的模型。InputIterator1 和 InputIterator2 具有相同的 value_type ，InputIterator1 的 value_type 是 LessThan Comparable 的模型，如 LessThan Comparable 要求中定义的，InputIterator1 的 value_type 上的排序是严格弱序，并且可转换为 OutputIterator 的 value_types 集合中的类型
- **InputIterator2**：是 Input Iterator 的模型。InputIterator2 和 InputIterator1 具有相同的 value_type ，InputIterator2 的 value_type 是 LessThan Comparable 的模型，如 LessThan Comparable 要求中定义的，InputIterator2 的 value_type 上的排序是严格弱序，并且可转换为 OutputIterator 的 value_types 集合中的类型
- **InputIterator3**：是 Input Iterator 的模型，InputIterator3 的 value_type 可转换为 OutputIterator2 的 value_types 集合中的类型
- **InputIterator4**：是 Input Iterator 的模型，InputIterator4 的 value_type 可转换为 OutputIterator2 的 value_types 集合中的类型
- **OutputIterator1**：是 Output Iterator 的模型
- **OutputIterator2**：是 Output Iterator 的模型

函数参数

- **keys_first1**：第一个键输入范围的起始
- **keys_last1**：第一个键输入范围的末尾
- **keys_first2**：第二个键输入范围的起始
- **keys_last2**：第二个键输入范围的末尾
- **values_first1**：第一个值输入范围的起始
- **values_first2**：第二个值输入范围的起始
- **keys_result**：键输出范围的起始
- **values_result**：值输出范围的起始

飞腾信息 MetaX Confidential 2024-11-11 15:00:00

前提条件

- 范围 [keys_first1, keys_last1) 和 [keys_first2, keys_last2) 应按照 operator< 进行排序
- 结果范围不应与任何一个输入范围重叠

返回值

p pair ，其中 p.first 是键输出范围的末尾，p.second 是值输出范围的末尾

另请参阅

- set_union_by_key
- set_intersection_by_key
- set_difference_by_key
- sort_by_key
- is_sorted

2.1.7.27 thrust::set_symmetric_difference_by_key

```
template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2,
  typename InputIterator3,
  typename InputIterator4,
  typename OutputIterator1,
  typename OutputIterator2,
  typename StrictWeakCompare>
__host__ __device__ thrust::pair< OutputIterator1, OutputIterator2 >
set_symmetric_difference_by_key(const thrust::detail::execution_policy_base
↪< DerivedPolicy > & exec,
  InputIterator1 keys_first1,
  InputIterator1 keys_last1,
  InputIterator2 keys_first2,
  InputIterator2 keys_last2,
  InputIterator3 values_first1,
  InputIterator4 values_first2,
  OutputIterator1 keys_result,
  OutputIterator2 values_result,
  StrictWeakCompare comp);
```

set_symmetric_difference_by_key 执行集合的键值对称差运算。set_difference_by_key 构造一个有序范围，是有序范围 [keys_first1, keys_last1) 和 [keys_first2, keys_last2) 的对称差集。与输入和输出键范围内的每个元素相关联的是一个值元素。关联的输入值范围不需要排序。

在最简单的情况下，set_symmetric_difference_by_key 执行以下集合运算：构造两个集合 A - B 和 B - A 的并集，其中 A 和 B 是两个输入范围。即，输出范围由属于 [keys_first1, keys_last1) 但不属于 [keys_first2, keys_last1) 的元素副本，和属于 [keys_first2, keys_last2) 但不属于 [keys_first1, keys_last1) 的元素副本组成。一般情况更为复杂，因为输入范围内可能包含重复元素。总而言之，如果 [keys_first1, keys_last1) 包含 m 个彼此等价的元素，[keys_first2, keys_last1) 包含 n 个彼此等价的元素，则 |m - n| 个这样的元素应被复制到输出范围：如果 m > n ，则复制 [keys_first1, keys_last1) 内最后 m - n 个元素；如果 m < n ，则复制 [keys_first2, keys_last2) 中最后 n - m 个元素。

每次将键元素从 [keys_first1, keys_last1) 或 [keys_first2, keys_last2) 复制到键输出范围时，将相应的值元素从相应的值输入范围（从 values_first1 或 values_first2 开始）复制到值输出范围。

此版本的 set_symmetric_difference_by_key 使用函数对象 comp 比较键元素。

算法的执行由 exec 确定并行化。

沐曦 MetaX Confidential 2024-11-18 15:00:00

以下代码片段演示了如何通过 thrust::host 并行化执行策略，使用 set_symmetric_difference_by_key 计算两组降序整数集合与其值的对称差集：

```
#include <thrust/set_operations.h>
#include <thrust/functional.h>
#include <thrust/execution_policy.h>
...
int A_keys[6] = {7, 6, 4, 2, 2, 1, 0};
int A_vals[6] = {0, 0, 0, 0, 0, 0, 0};

int B_keys[5] = {8, 5, 2, 1, 1};
int B_vals[5] = {1, 1, 1, 1, 1};

int keys_result[6];
int vals_result[6];

thrust::pair<int*,int*> end = thrust::set_symmetric_difference_by_
↪key(thrust::host, A_keys, A_keys + 6, B_keys, B_keys + 5, A_vals, B_vals,
↪ keys_result, vals_result);
// keys_result is now {8, 7, 6, 5, 4, 0}
// vals_result is now {1, 0, 0, 1, 0, 0}
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **InputIterator1**：是 Input Iterator 的模型。InputIterator1 和 InputIterator2 具有相同的 value_type，InputIterator1 的 value_type 是 LessThan Comparable 的模型，如 LessThan Comparable 要求中定义的，InputIterator1 的 value_type 上的排序是严格弱序，并且可转换为 OutputIterator 的 value_types 集合中的类型
- **InputIterator2**：是 Input Iterator 的模型。InputIterator2 和 InputIterator1 具有相同的 value_type，InputIterator2 的 value_type 是 LessThan Comparable 的模型，如 LessThan Comparable 要求中定义的，InputIterator2 的 value_type 上的排序是严格弱序，并且可转换为 OutputIterator 的 value_types 集合中的类型
- **InputIterator3**：是 Input Iterator 的模型，InputIterator3 的 value_type 可转换为 OutputIterator2 的 value_types 集合中的类型
- **InputIterator4**：是 Input Iterator 的模型，InputIterator4 的 value_type 可转换为 OutputIterator2 的 value_types 集合中的类型
- **OutputIterator1**：是 Output Iterator 的模型
- **OutputIterator2**：是 Output Iterator 的模型
- **StrictWeakCompare**：是 Strict Weak Ordering 的模型

函数参数

- **exec**：用于并行化的执行策略
- **keys_first1**：第一个键输入范围的起始
- **keys_last1**：第一个键输入范围的末尾
- **keys_first2**：第二个键输入范围的起始
- **keys_last2**：第二个键输入范围的末尾
- **values_first1**：第一个值输入范围的起始
- **values_first2**：第二个值输入范围的起始
- **keys_result**：键输出范围的起始
- **values_result**：值输出范围的起始

飞腾信息 MetaX Confidential 2024-11-18 15:0:00

- **comp**：比较运算符

前提条件

- 范围 [keys_first1, keys_last1) 和 [keys_first2, keys_last2) 应按照 comp 进行排序
- 结果范围不应与任何一个输入范围重叠

返回值

p pair ，其中 p.first 是键输出范围的末尾，p.second 是值输出范围的末尾

另请参阅

- set_union_by_key
- set_intersection_by_key
- set_difference_by_key
- sort_by_key
- is_sorted

2.1.7.28 thrust::set_symmetric_difference_by_key

```
template <typename InputIterator1,
  typename InputIterator2,
  typename InputIterator3,
  typename InputIterator4,
  typename OutputIterator1,
  typename OutputIterator2,
  typename StrictWeakCompare>
thrust::pair< OutputIterator1, OutputIterator2 >
set_symmetric_difference_by_key(InputIterator1 keys_first1,
  InputIterator1 keys_last1,
  InputIterator2 keys_first2,
  InputIterator2 keys_last2,
  InputIterator3 values_first1,
  InputIterator4 values_first2,
  OutputIterator1 keys_result,
  OutputIterator2 values_result,
  StrictWeakCompare comp);
```

set_symmetric_difference_by_key 执行集合的键值对称差运算。set_difference_by_key 构造一个有序范围，是有序范围 [keys_first1, keys_last1) 和 [keys_first2, keys_last2) 的对称差集。与输入和输出键范围内的每个元素相关联的是一个值元素。关联的输入值范围不需要排序。

在最简单的情况下，set_symmetric_difference_by_key 执行以下集合运算：构造两个集合 A - B 和 B - A 的并集，其中 A 和 B 是两个输入范围。即，输出范围由属于 [keys_first1, keys_last1) 但不属于 [keys_first2, keys_last1) 的元素副本，和属于 [keys_first2, keys_last2) 但不属于 [keys_first1, keys_last1) 的元素副本组成。一般情况更为复杂，因为输入范围内可能包含重复元素。总而言之，如果 [keys_first1, keys_last1) 包含 m 个彼此等价的元素，[keys_first2, keys_last1) 包含 n 个彼此等价的元素，则 |m - n| 个这样的元素应被复制到输出范围：如果 m > n ，则复制 [keys_first1, keys_last1) 内最后 m - n 个元素；如果 m < n ，则复制 [keys_first2, keys_last2) 中最后 n - m 个元素。

每次将键元素从 [keys_first1, keys_last1) 或 [keys_first2, keys_last2) 复制到键输出范围时，将相应的值元素从相应的值输入范围（从 values_first1 或 values_first2 开始）复制到值输出范围。

此版本的 set_symmetric_difference_by_key 使用函数对象 comp 比较键元素。

以下代码片段演示了如何使用 set_symmetric_difference_by_key 计算两组降序整数集合与其值的对称差集：

沐曦 MetaX Confidential 2024-11-18 15:00:00

```
#include <thrust/set_operations.h>
#include <thrust/functional.h>
...
int A_keys[6] = {7, 6, 4, 2, 2, 1, 0};
int A_vals[6] = {0, 0, 0, 0, 0, 0, 0};

int B_keys[5] = {8, 5, 2, 1, 1};
int B_vals[5] = {1, 1, 1, 1, 1};

int keys_result[6];
int vals_result[6];

thrust::pair<int*,int*> end = thrust::set_symmetric_difference_by_key(A_
↪keys, A_keys + 6, B_keys, B_keys + 5, A_vals, B_vals, keys_result, vals_
↪result);
// keys_result is now {8, 7, 6, 5, 4, 0}
// vals_result is now {1, 0, 0, 1, 0, 0}
```

模板参数

- **InputIterator1**：是 Input Iterator 的模型。InputIterator1 和 InputIterator2 具有相同的 value_type，InputIterator1 的 value_type 是 LessThan Comparable 的模型，如 LessThan Comparable 要求中定义的，InputIterator1 的 value_type 上的排序是严格弱序，并且可转换为 OutputIterator 的 value_types 集合中的类型
- **InputIterator2**：是 Input Iterator 的模型。InputIterator2 和 InputIterator1 具有相同的 value_type，InputIterator2 的 value_type 是 LessThan Comparable 的模型，如 LessThan Comparable 要求中定义的，InputIterator2 的 value_type 上的排序是严格弱序，并且可转换为 OutputIterator 的 value_types 集合中的类型
- **InputIterator3**：是 Input Iterator 的模型，InputIterator3 的 value_type 可转换为 OutputIterator2 的 value_types 集合中的类型
- **InputIterator4**：是 Input Iterator 的模型，InputIterator4 的 value_type 可转换为 OutputIterator2 的 value_types 集合中的类型
- **OutputIterator1**：是 Output Iterator 的模型
- **OutputIterator2**：是 Output Iterator 的模型
- **StrictWeakCompare**：是 Strict Weak Ordering 的模型

函数参数

- **keys_first1**：第一个键输入范围的起始
- **keys_last1**：第一个键输入范围的末尾
- **keys_first2**：第二个键输入范围的起始
- **keys_last2**：第二个键输入范围的末尾
- **values_first1**：第一个值输入范围的起始
- **values_first2**：第二个值输入范围的起始
- **keys_result**：键输出范围的起始
- **values_result**：值输出范围的起始
- **comp**：比较运算符

前提条件

- 范围 [keys_first1, keys_last1) 和 [keys_first2, keys_last2) 应按照 comp 进行排序
- 结果范围不应与任何一个输入范围重叠

飞腾信息 Metax Confidential 2024-11-18 15:00:00

返回值

p pair，其中 p.first 是键输出范围的末尾，p.second 是值输出范围的末尾

另请参阅

- set_union_by_key
- set_intersection_by_key
- set_difference_by_key
- sort_by_key
- is_sorted

2.1.7.29 thrust::set_union_by_key

```
template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2,
  typename InputIterator3,
  typename InputIterator4,
  typename OutputIterator1,
  typename OutputIterator2>
__host__ __device__ thrust::pair< OutputIterator1, OutputIterator2 >
set_union_by_key(const thrust::detail::execution_policy_base<
 ↪DerivedPolicy > & exec,
  InputIterator1 keys_first1,
  InputIterator1 keys_last1,
  InputIterator2 keys_first2,
  InputIterator2 keys_last2,
  InputIterator3 values_first1,
  InputIterator4 values_first2,
  OutputIterator1 keys_result,
  OutputIterator2 values_result);
```

set_union_by_key 执行集合的键值并运算。set_union_by_key 构造一个有序范围，是有序范围 [keys_first1, keys_last1) 和 [keys_first2, keys_last2) 的并集。与输入和输出键范围内的每个元素相关联的是一个值元素。关联的输入值范围不需要排序。

在最简单的情况下，set_union_by_key 执行集合的并运算：输出范围由属于 [keys_first2, keys_last1) 或 [keys_first1, keys_last1)，或者属于两者的元素副本组成。一般情况更为复杂，因为输入范围内可能包含重复元素。总而言之，如果 [keys_first1, keys_last1) 包含 m 个彼此等价的元素，[keys_first2, keys_last2) 包含 n 个彼此等价的元素，则第一个范围内的所有 m 个元素都应按顺序复制到输出范围，第二个范围内的 max(n - m, 0) 个元素应按顺序复制到输出范围。

每次将键元素从 [keys_first1, keys_last1) 或 [keys_first2, keys_last2) 复制到键输出范围时，将相应的值元素从相应的值输入范围（从 values_first1 或 values_first2 开始）复制到值输出范围。

此版本的 set_union_by_key 使用 operator< 比较键元素。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::host 并行化执行策略，使用 set_symmetric_difference_by_key 计算两组升序整数集合与其值的对称差集：

```
#include <thrust/set_operations.h>
#include <thrust/execution_policy.h>
...
int A_keys[6] = {0, 2, 4, 6, 8, 10, 12};
int A_vals[6] = {0, 0, 0, 0, 0,  0,  0};
```

(下页继续)

沐曦信息 MetaX Confidential 2023-11-18 15:00:00

(续上页)

```
int B_keys[5] = {1, 3, 5, 7, 9};
int B_vals[5] = {1, 1, 1, 1, 1};

int keys_result[11];
int vals_result[11];

thrust::pair<int*,int*> end = thrust::set_symmetric_difference_by_
↪key(thrust::host, A_keys, A_keys + 6, B_keys, B_keys + 5, A_vals, B_vals,
↪ keys_result, vals_result);
// keys_result is now {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12}
// vals_result is now {0, 1, 0, 1, 0, 1, 0, 1, 0, 1,  0,  0}
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **InputIterator1**：是 Input Iterator 的模型。InputIterator1 和 InputIterator2 具有相同的 value_type，InputIterator1 的 value_type 是 LessThan Comparable 的模型，如 LessThan Comparable 要求中定义的，InputIterator1 的 value_type 上的排序是严格弱序，并且可转换为 OutputIterator 的 value_types 集合中的类型
- **InputIterator2**：是 Input Iterator 的模型。InputIterator2 和 InputIterator1 具有相同的 value_type，InputIterator2 的 value_type 是 LessThan Comparable 的模型，如 LessThan Comparable 要求中定义的，InputIterator2 的 value_type 上的排序是严格弱序，并且可转换为 OutputIterator 的 value_types 集合中的类型
- **InputIterator3**：是 Input Iterator 的模型，InputIterator3 的 value_type 可转换为 OutputIterator2 的 value_types 集合中的类型
- **InputIterator4**：是 Input Iterator 的模型，InputIterator4 的 value_type 可转换为 OutputIterator2 的 value_types 集合中的类型
- **OutputIterator1**：是 Output Iterator 的模型
- **OutputIterator2**：是 Output Iterator 的模型

函数参数

- **exec**：用于并行化的执行策略
- **keys_first1**：第一个键输入范围的起始
- **keys_last1**：第一个键输入范围的末尾
- **keys_first2**：第二个键输入范围的起始
- **keys_last2**：第二个键输入范围的末尾
- **values_first1**：第一个值输入范围的起始
- **values_first2**：第二个值输入范围的起始
- **keys_result**：键输出范围的起始
- **values_result**：值输出范围的起始

前提条件

- 范围 [keys_first1, keys_last1) 和 [keys_first2, keys_last2) 应按照 operator< 进行排序
- 结果范围不应与任何一个输入范围重叠

返回值

p pair，其中 p.first 是键输出范围的末尾，p.second 是值输出范围的末尾

另请参阅

沐曦信息 MetaX Confidential 2024-11-18 15:00:00

- set_symmetric_difference_by_key
- set_intersection_by_key
- set_difference_by_key
- sort_by_key
- is_sorted

2.1.7.30 thrust::set_union_by_key

```
template <typename InputIterator1,
  typename InputIterator2,
  typename InputIterator3,
  typename InputIterator4,
  typename OutputIterator1,
  typename OutputIterator2>
thrust::pair< OutputIterator1, OutputIterator2 >
set_union_by_key(InputIterator1 keys_first1,
  InputIterator1 keys_last1,
  InputIterator2 keys_first2,
  InputIterator2 keys_last2,
  InputIterator3 values_first1,
  InputIterator4 values_first2,
  OutputIterator1 keys_result,
  OutputIterator2 values_result);
```

set_union_by_key 执行集合的键值并运算。set_union_by_key 构造一个有序范围，是有序范围 [keys_first1, keys_last1) 和 [keys_first2, keys_last2) 的并集。与输入和输出键范围内的每个元素相关联的是一个值元素。关联的输入值范围不需要排序。

在最简单的情况下，set_union_by_key 执行集合的并运算：输出范围由属于 [keys_first2, keys_last1) 或 [keys_first1, keys_last1)，或者属于两者的元素副本组成。一般情况更为复杂，因为输入范围内可能包含重复元素。总而言之，如果 [keys_first1, keys_last1) 包含 m 个彼此等价的元素，[keys_first2, keys_last2) 包含 n 个彼此等价的元素，则第一个范围内的所有 m 个元素都应按顺序复制到输出范围，第二个范围内的 max(n - m, 0) 个元素应按顺序复制到输出范围。

每次将键元素从 [keys_first1, keys_last1) 或 [keys_first2, keys_last2) 复制到键输出范围时，将相应的值元素从相应的值输入范围（从 values_first1 或 values_first2 开始）复制到值输出范围。

此版本的 set_union_by_key 使用 operator< 比较键元素。

以下代码片段演示了如何使用 set_symmetric_difference_by_key 计算两组升序整数集合与其值的对称差集：

```
#include <thrust/set_operations.h>
...
int A_keys[6] = {0, 2, 4, 6, 8, 10, 12};
int A_vals[6] = {0, 0, 0, 0, 0,  0,  0};

int B_keys[5] = {1, 3, 5, 7, 9};
int B_vals[5] = {1, 1, 1, 1, 1};

int keys_result[11];
int vals_result[11];

thrust::pair<int*,int*> end = thrust::set_symmetric_difference_by_key(A_
↪keys, A_keys + 6, B_keys, B_keys + 5, A_vals, B_vals, keys_result, vals_
↪result);
```

(下页继续)

沐曦集成电路（南京）有限公司 MetaX Confidential 2024-11-15 15:00:00

(续上页)

```
// keys_result is now {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12}
// vals_result is now {0, 1, 0, 1, 0, 1, 0, 1, 0, 1,  0,  0}
```

模板参数

- **InputIterator1**：是 Input Iterator 的模型。InputIterator1 和 InputIterator2 具有相同的 value_type，InputIterator1 的 value_type 是 LessThan Comparable 的模型，如 LessThan Comparable 要求中定义的，InputIterator1 的 value_type 上的排序是严格弱序，并且可转换为 OutputIterator 的 value_types 集合中的类型
- **InputIterator2**：是 Input Iterator 的模型。InputIterator2 和 InputIterator1 具有相同的 value_type，InputIterator2 的 value_type 是 LessThan Comparable 的模型，如 LessThan Comparable 要求中定义的，InputIterator2 的 value_type 上的排序是严格弱序，并且可转换为 OutputIterator 的 value_types 集合中的类型
- **InputIterator3**：是 Input Iterator 的模型，InputIterator3 的 value_type 可转换为 OutputIterator2 的 value_types 集合中的类型
- **InputIterator4**：是 Input Iterator 的模型，InputIterator4 的 value_type 可转换为 OutputIterator2 的 value_types 集合中的类型
- **OutputIterator1**：是 Output Iterator 的模型
- **OutputIterator2**：是 Output Iterator 的模型

函数参数

- **keys_first1**：第一个键输入范围的起始
- **keys_last1**：第一个键输入范围的末尾
- **keys_first2**：第二个键输入范围的起始
- **keys_last2**：第二个键输入范围的末尾
- **values_first1**：第一个值输入范围的起始
- **values_first2**：第二个值输入范围的起始
- **keys_result**：键输出范围的起始
- **values_result**：值输出范围的起始

前提条件

- 范围 [keys_first1, keys_last1) 和 [keys_first2, keys_last2) 应按照 operator< 进行排序
- 结果范围不应与任何一个输入范围重叠

返回值

p pair ，其中 p.first 是键输出范围的末尾，p.second 是值输出范围的末尾

另请参阅

- set_symmetric_difference_by_key
- set_intersection_by_key
- set_difference_by_key
- sort_by_key
- is_sorted

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

2.1.7.31 thrust::set_union_by_key

```
template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2,
  typename InputIterator3,
  typename InputIterator4,
  typename OutputIterator1,
  typename OutputIterator2,
  typename StrictWeakCompare>
__host__ __device__ thrust::pair< OutputIterator1, OutputIterator2 >
set_union_by_key(const thrust::detail::execution_policy_base<
 ↪DerivedPolicy > & exec,
  InputIterator1 keys_first1,
  InputIterator1 keys_last1,
  InputIterator2 keys_first2,
  InputIterator2 keys_last2,
  InputIterator3 values_first1,
  InputIterator4 values_first2,
  OutputIterator1 keys_result,
  OutputIterator2 values_result,
  StrictWeakCompare comp);
```

set_union_by_key 执行集合的键值并运算。set_union_by_key 构造一个有序范围，是有序范围 [keys_first1, keys_last1) 和 [keys_first2, keys_last2) 的并集。与输入和输出键范围内的每个元素相关联的是一个值元素。关联的输入值范围不需要排序。

在最简单的情况下，set_union_by_key 执行集合的并运算：输出范围由属于 [keys_first2, keys_last1) 或 [keys_first1, keys_last1)，或者属于两者的元素副本组成。一般情况更为复杂，因为输入范围内可能包含重复元素。总而言之，如果 [keys_first1, keys_last1) 包含 m 个彼此等价的元素，[keys_first2, keys_last2) 包含 n 个彼此等价的元素，则第一个范围内的所有 m 个元素都应按顺序复制到输出范围，第二个范围内的 max(n - m, 0) 个元素应按顺序复制到输出范围。

每次将键元素从 [keys_first1, keys_last1) 或 [keys_first2, keys_last2) 复制到键输出范围时，将相应的值元素从相应的值输入范围（从 values_first1 或 values_first2 开始）复制到值输出范围。

此版本的 set_union_by_key 使用函数对象 comp 比较键元素。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::host 并行化执行策略，使用 set_symmetric_difference_by_key 计算两组降序整数集合与其值的对称差集：

```
#include <thrust/set_operations.h>
#include <thrust/functional.h>
#include <thrust/execution_policy.h>
...
int A_keys[6] = {12, 10, 8, 6, 4, 2, 0};
int A_vals[6] = { 0,  0, 0, 0, 0, 0, 0};

int B_keys[5] = {9, 7, 5, 3, 1};
int B_vals[5] = {1, 1, 1, 1, 1};

int keys_result[11];
int vals_result[11];

thrust::pair<int*,int*> end = thrust::set_symmetric_difference_by_
 ↪key(thrust::host, A_keys, A_keys + 6, B_keys, B_keys + 5, A_vals, B_vals,
 ↪ keys_result, vals_result, thrust::greater<int>());
// keys_result is now {12, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0}
```

(下页继续)

(续上页)

```
// vals_result is now { 0,  1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0}
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **InputIterator1**：是 Input Iterator 的模型。InputIterator1 和 InputIterator2 具有相同的 value_type，InputIterator1 的 value_type 是 LessThan Comparable 的模型，如 LessThan Comparable 要求中定义的，InputIterator1 的 value_type 上的排序是严格弱序，并且可转换为 OutputIterator 的 value_types 集合中的类型
- **InputIterator2**：是 Input Iterator 的模型。InputIterator2 和 InputIterator1 具有相同的 value_type，InputIterator2 的 value_type 是 LessThan Comparable 的模型，如 LessThan Comparable 要求中定义的，InputIterator2 的 value_type 上的排序是严格弱序，并且可转换为 OutputIterator 的 value_types 集合中的类型
- **InputIterator3**：是 Input Iterator 的模型，InputIterator3 的 value_type 可转换为 OutputIterator2 的 value_types 集合中的类型
- **InputIterator4**：是 Input Iterator 的模型，InputIterator4 的 value_type 可转换为 OutputIterator2 的 value_types 集合中的类型
- **OutputIterator1**：是 Output Iterator 的模型
- **OutputIterator2**：是 Output Iterator 的模型
- **StrictWeakCompare**：是 Strict Weak Ordering 的模型

函数参数

- **exec**：用于并行化的执行策略
- **keys_first1**：第一个键输入范围的起始
- **keys_last1**：第一个键输入范围的末尾
- **keys_first2**：第二个键输入范围的起始
- **keys_last2**：第二个键输入范围的末尾
- **values_first1**：第一个值输入范围的起始
- **values_first2**：第二个值输入范围的起始
- **keys_result**：键输出范围的起始
- **values_result**：值输出范围的起始
- **comp**：比较运算符

前提条件

- 范围 [keys_first1, keys_last1) 和 [keys_first2, keys_last2) 应按照 comp 进行排序
- 结果范围不应与任何一个输入范围重叠

返回值

p pair，其中 p.first 是键输出范围的末尾，p.second 是值输出范围的末尾

另请参阅

- set_symmetric_difference_by_key
- set_intersection_by_key
- set_difference_by_key
- sort_by_key
- is_sorted

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

2.1.7.32 thrust::set_union_by_key

```
template <typename InputIterator1,
  typename InputIterator2,
  typename InputIterator3,
  typename InputIterator4,
  typename OutputIterator1,
  typename OutputIterator2,
  typename StrictWeakCompare>
thrust::pair< OutputIterator1, OutputIterator2 >
set_union_by_key(InputIterator1 keys_first1,
  InputIterator1 keys_last1,
  InputIterator2 keys_first2,
  InputIterator2 keys_last2,
  InputIterator3 values_first1,
  InputIterator4 values_first2,
  OutputIterator1 keys_result,
  OutputIterator2 values_result,
  StrictWeakCompare comp);
```

set_union_by_key 执行集合的键值并运算。set_union_by_key 构造一个有序范围，是有序范围 [keys_first1, keys_last1) 和 [keys_first2, keys_last2) 的并集。与输入和输出键范围内的每个元素相关联的是一个值元素。关联的输入值范围不需要排序。

在最简单的情况下，set_union_by_key 执行集合的并运算：输出范围由属于 [keys_first2, keys_last1) 或 [keys_first1, keys_last1)，或者属于两者的元素副本组成。一般情况更为复杂，因为输入范围内可能包含重复元素。总而言之，如果 [keys_first1, keys_last1) 包含 m 个彼此等价的元素，[keys_first2, keys_last2) 包含 n 个彼此等价的元素，则第一个范围内的所有 m 个元素都应按顺序复制到输出范围，第二个范围内的 max(n - m, 0) 个元素应按顺序复制到输出范围。

每次将键元素从 [keys_first1, keys_last1) 或 [keys_first2, keys_last2) 复制到键输出范围时，将相应的值元素从相应的值输入范围（从 values_first1 或 values_first2 开始）复制到值输出范围。

此版本的 set_union_by_key 使用函数对象 comp 比较键元素。

以下代码片段演示了如何使用 set_symmetric_difference_by_key 计算两组降序整数集合与其值的对称差集：

```
#include <thrust/set_operations.h>
#include <thrust/functional.h>
...
int A_keys[6] = {12, 10, 8, 6, 4, 2, 0};
int A_vals[6] = { 0,  0, 0, 0, 0, 0, 0};

int B_keys[5] = {9, 7, 5, 3, 1};
int B_vals[5] = {1, 1, 1, 1, 1};

int keys_result[11];
int vals_result[11];

thrust::pair<int*,int*> end = thrust::set_symmetric_difference_by_key(A_
↪keys, A_keys + 6, B_keys, B_keys + 5, A_vals, B_vals, keys_result, vals_
↪result, thrust::greater<int>());
// keys_result is now {12, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0}
// vals_result is now { 0,  1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0}
```

模板参数

- **InputIterator1**：是 Input Iterator 的模型。InputIterator1 和 InputIterator2 具有相同的 value_type，InputIterator1 的 value_type 是 LessThan Comparable 的模型，如 LessThan Comparable 要求

中定义的，InputIterator1 的 value_type 上的排序是严格弱序，并且可转换为 OutputIterator 的 value_types 集合中的类型

- **InputIterator2**：是 Input Iterator 的模型。InputIterator2 和 InputIterator1 具有相同的 value_type，InputIterator2 的 value_type 是 LessThan Comparable 的模型，如 LessThan Comparable 要求中定义的，InputIterator2 的 value_type 上的排序是严格弱序，并且可转换为 OutputIterator 的 value_types 集合中的类型
- **InputIterator3**：是 Input Iterator 的模型，InputIterator3 的 value_type 可转换为 OutputIterator2 的 value_types 集合中的类型
- **InputIterator4**：是 Input Iterator 的模型，InputIterator4 的 value_type 可转换为 OutputIterator2 的 value_types 集合中的类型
- **OutputIterator1**：是 Output Iterator 的模型
- **OutputIterator2**：是 Output Iterator 的模型
- **StrictWeakCompare**：是 Strict Weak Ordering 的模型

函数参数

- **keys_first1**：第一个键输入范围的起始
- **keys_last1**：第一个键输入范围的末尾
- **keys_first2**：第二个键输入范围的起始
- **keys_last2**：第二个键输入范围的末尾
- **values_first1**：第一个值输入范围的起始
- **values_first2**：第二个值输入范围的起始
- **keys_result**：键输出范围的起始
- **values_result**：值输出范围的起始
- **comp**：比较运算符

前提条件

- 范围 [keys_first1, keys_last1) 和 [keys_first2, keys_last2) 应按照 comp 进行排序
- 结果范围不应与任何一个输入范围重叠

返回值

p pair，其中 p.first 是键输出范围的末尾，p.second 是值输出范围的末尾

另请参阅

- set_symmetric_difference_by_key
- set_intersection_by_key
- set_difference_by_key
- sort_by_key
- is_sorted

2.1.8 排序（Sorting）

函数

飞腾信息 MetaX Confident 1 | 2024-11-18 15:00:00

```
template <typename DerivedPolicy,
  typename RandomAccessIterator>
__host__ __device__ void
thrust::sort(const thrust::detail::execution_policy_base< DerivedPolicy > &
↪ exec,
  RandomAccessIterator first,
  RandomAccessIterator last);

template <typename RandomAccessIterator>
void
thrust::sort(RandomAccessIterator first,
  RandomAccessIterator last);

template <typename DerivedPolicy,
  typename RandomAccessIterator,
  typename StrictWeakOrdering>
__host__ __device__ void
thrust::sort(const thrust::detail::execution_policy_base< DerivedPolicy > &
↪ exec,
  RandomAccessIterator first,
  RandomAccessIterator last,
  StrictWeakOrdering comp);

template <typename RandomAccessIterator,
  typename StrictWeakOrdering>
__host__ __device__ void
thrust::sort(RandomAccessIterator first,
  RandomAccessIterator last,
  StrictWeakOrdering comp);

template <typename DerivedPolicy,
  typename RandomAccessIterator>
__host__ __device__ void
thrust::stable_sort(const thrust::detail::execution_policy_base<
↪DerivedPolicy > & exec,
  RandomAccessIterator first,
  RandomAccessIterator last);

template <typename RandomAccessIterator>
void
thrust::stable_sort(RandomAccessIterator first,
  RandomAccessIterator last);

template <typename DerivedPolicy,
  typename RandomAccessIterator,
  typename StrictWeakOrdering>
__host__ __device__ void
thrust::stable_sort(const thrust::detail::execution_policy_base<
↪DerivedPolicy > & exec,
  RandomAccessIterator first,
  RandomAccessIterator last,
  StrictWeakOrdering comp);

template <typename RandomAccessIterator,
  typename StrictWeakOrdering>
void
thrust::stable_sort(RandomAccessIterator first,
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
  RandomAccessIterator last,
  StrictWeakOrdering comp);

template <typename DerivedPolicy,
  typename RandomAccessIterator1,
  typename RandomAccessIterator2>
__host__ __device__ void
thrust::sort_by_key(const thrust::detail::execution_policy_base<
 ↪DerivedPolicy > & exec,
  RandomAccessIterator1 keys_first,
  RandomAccessIterator1 keys_last,
  RandomAccessIterator2 values_first);

template <typename RandomAccessIterator1,
  typename RandomAccessIterator2>
void
thrust::sort_by_key(RandomAccessIterator1 keys_first,
  RandomAccessIterator1 keys_last,
  RandomAccessIterator2 values_first);

template <typename DerivedPolicy,
  typename RandomAccessIterator1,
  typename RandomAccessIterator2,
  typename StrictWeakOrdering>
__host__ __device__ void
thrust::sort_by_key(const thrust::detail::execution_policy_base<
 ↪DerivedPolicy > & exec,
  RandomAccessIterator1 keys_first,
  RandomAccessIterator1 keys_last,
  RandomAccessIterator2 values_first,
  StrictWeakOrdering comp);

template <typename RandomAccessIterator1,
  typename RandomAccessIterator2,
  typename StrictWeakOrdering>
void
thrust::sort_by_key(RandomAccessIterator1 keys_first,
  RandomAccessIterator1 keys_last,
  RandomAccessIterator2 values_first,
  StrictWeakOrdering comp);

template <typename DerivedPolicy,
  typename RandomAccessIterator1,
  typename RandomAccessIterator2>
__host__ __device__ void
thrust::stable_sort_by_key(const thrust::detail::execution_policy_base<
 ↪DerivedPolicy > & exec,
  RandomAccessIterator1 keys_first,
  RandomAccessIterator1 keys_last,
  RandomAccessIterator2 values_first);

template <typename RandomAccessIterator1,
  typename RandomAccessIterator2>
void
thrust::stable_sort_by_key(RandomAccessIterator1 keys_first,
 RandomAccessIterator1 keys_last,
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
  RandomAccessIterator2 values_first);

template <typename DerivedPolicy,
  typename RandomAccessIterator1,
  typename RandomAccessIterator2,
  typename StrictWeakOrdering>
__host__ __device__ void
thrust::stable_sort_by_key(const thrust::detail::execution_policy_base<
↪DerivedPolicy > & exec,
  RandomAccessIterator1 keys_first,
  RandomAccessIterator1 keys_last,
  RandomAccessIterator2 values_first,
  StrictWeakOrdering comp);

template <typename RandomAccessIterator1,
  typename RandomAccessIterator2,
  typename StrictWeakOrdering>
void
thrust::stable_sort_by_key(RandomAccessIterator1 keys_first,
  RandomAccessIterator1 keys_last,
  RandomAccessIterator2 values_first,
  StrictWeakOrdering comp);
```

2.1.8.1 thrust::sort

```
template <typename DerivedPolicy,
  typename RandomAccessIterator>
__host__ __device__ void
sort(const thrust::detail::execution_policy_base< DerivedPolicy > & exec,
  RandomAccessIterator first,
  RandomAccessIterator last);
```

sort 按升序对 [first, last) 内的元素进行排序，指的是如果 i 和 j 是 [first, last) 内任意满足 i 先于 j 的两个有效迭代器，那么 *j 不小于 *i。

不能保证 sort 是稳定的。即，假设 *i 和 *j 是等价的（两者都不小于另一个），不能保证 sort 会保留这两个元素的相对顺序。

此版本的 sort 使用 operator< 比较对象。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::host 并行化执行策略，使用 sort 对整数序列进行排序：

```
#include <thrust/sort.h>
#include <thrust/execution_policy.h>
...
const int N = 6;
int A[N] = {1, 4, 2, 8, 5, 7};
thrust::sort(thrust::host, A, A + N);
// A is now {1, 2, 4, 5, 7, 8}
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **RandomAccessIterator**：是 Random Access Iterator 的模型，RandomAccessIterator 是可变的。

飞谱信息 MetaX Confidential 2024-11-18 15:00:00

RandomAccessIterator 的 value_type 是 LessThan Comparable 的模型，如在 LessThan Comparable 要求中定义的，RandomAccessIterator 的 value_type 上的排序是 严格弱序

函数参数

- **exec**：用于并行化的执行策略
- **first**：序列起始
- **last**：序列末尾

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/sort
- stable_sort
- sort_by_key

2.1.8.2 thrust::sort

```
template <typename RandomAccessIterator>
void
sort(RandomAccessIterator first,
  RandomAccessIterator last);
```

sort 按升序对 [first, last) 内的元素进行排序，指的是如果 i 和 j 是 [first, last) 内任意满足 i 先于 j 的两个有效迭代器，那么 *j 不小于 *i。

不能保证 sort 是稳定的。即，假设 *i 和 *j 是等价的（两者都不小于另一个），不能保证 sort 会保留这两个元素的相对顺序。

此版本的 sort 使用 operator< 比较对象。

以下代码片段演示了如何使用 sort 对整数序列进行排序：

```
#include <thrust/sort.h>
...
const int N = 6;
int A[N] = {1, 4, 2, 8, 5, 7};
thrust::sort(A, A + N);
// A is now {1, 2, 4, 5, 7, 8}
```

模板参数

RandomAccessIterator：是 Random Access Iterator 的模型，RandomAccessIterator 是可变的。RandomAccessIterator 的 value_type 是 LessThan Comparable 的模型，如在 LessThan Comparable 要求中定义的，RandomAccessIterator 的 value_type 上的排序是 严格弱序

函数参数

- **first**：序列起始
- **last**：序列末尾

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/sort
- stable_sort
- sort_by_key

沐曦信息 MetaX Confidential 2024-11-18 15:00:00

2.1.8.3 thrust::sort

```
template <typename DerivedPolicy,
  typename RandomAccessIterator,
  typename StrictWeakOrdering>
__host__ __device__ void
sort(const thrust::detail::execution_policy_base< DerivedPolicy > & exec,
  RandomAccessIterator first,
  RandomAccessIterator last,
  StrictWeakOrdering comp);
```

sort 按升序对 [first, last) 内的元素进行排序，指的是如果 i 和 j 是 [first, last) 内任意满足 i 先于 j 的两个有效迭代器，那么 *j 不小于 *i。

不能保证 sort 是稳定的。即，假设 *i 和 *j 是等价的（两者都不小于另一个），不能保证 sort 会保留这两个元素的相对顺序。

此版本的 sort 使用函数对象 comp 比较对象。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::host 并行化执行策略，使用大于比较运算符按降序对整数进行排序：

```
#include <thrust/sort.h>
#include <thrust/functional.h>
#include <thrust/execution_policy.h>
...
const int N = 6;
int A[N] = {1, 4, 2, 8, 5, 7};
thrust::sort(thrust::host, A, A + N, thrust::greater<int>());
// A is now {8, 7, 5, 4, 2, 1};
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **RandomAccessIterator**：是 Random Access Iterator 的模型，RandomAccessIterator ‘是可变的，并且 ‘RandomAccessIterator 的 value_type 可转换为 StrictWeakOrdering 的 first_argument_type 和 second_argument_type
- **StrictWeakOrdering**：是 Strict Weak Ordering 的模型

函数参数

- **exec**：用于并行化的执行策略
- **first**：序列起始
- **last**：序列末尾
- **comp**：比较运算符

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/sort
- stable_sort
- sort_by_key

2.1.8.4 thrust::sort

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

```
template <typename RandomAccessIterator,
  typename StrictWeakOrdering>
__host__ __device__ void
sort(RandomAccessIterator first,
  RandomAccessIterator last,
  StrictWeakOrdering comp);
```

sort 按升序对 [first, last) 内的元素进行排序，这意味着如果 i 和 j 是 [first, last) 内任意两个有效迭代器，其中 i 在 j 之前，那么 *j 不小于 *i。

不能保证 sort 是稳定的。即，假设 *i 和 *j 是等价的（两者都不小于另一个），不能保证 sort 会保留这两个元素的相对顺序。

此版本的 sort 使用函数对象 comp 比较对象。

以下代码片段演示了如何使用大于比较运算符按降序对整数进行排序：

```
#include <thrust/sort.h>
#include <thrust/functional.h>
...
const int N = 6;
int A[N] = {1, 4, 2, 8, 5, 7};
thrust::sort(A, A + N, thrust::greater<int>());
// A is now {8, 7, 5, 4, 2, 1};
```

模板参数

- **RandomAccessIterator**：是 Random Access Iterator 的模型，RandomAccessIterator ‘是可变的，并且 ‘RandomAccessIterator 的 value_type 可转换为 StrictWeakOrdering 的 first_argument_type 和 second_argument_type
- **StrictWeakOrdering**：是 Strict Weak Ordering 的模型

函数参数

- **first**：序列起始
- **last**：序列末尾
- **comp**：比较运算符

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/sort
- stable_sort
- sort_by_key

2.1.8.5 thrust::stable_sort

```
template <typename DerivedPolicy,
  typename RandomAccessIterator>
__host__ __device__ void
stable_sort(const thrust::detail::execution_policy_base< DerivedPolicy > &
↪exec,
  RandomAccessIterator first,
  RandomAccessIterator last);
```

stable_sort 很像 sort，按升序对 [first, last) 内的元素进行排序，指的是如果 i 和 j 是 [first, last) 内任意满足 i 先于 j 的两个有效迭代器，那么 *j 不小于 *i。

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

顾名思义，stable_sort 是稳定的：它保留了等价元素的相对顺序。即，如果 x 和 y 是 [first, last) 中的元素，满足 x 先于 y，并且如果这两个元素是等价的（既不是 x < y 也不是 y < x），那么 stable_sort 的后置条件是 x 仍然先于 y。

此版本的 stable_sort 使用 operator< 比较对象。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::host 并行化执行策略，使用 sort 对整数序列进行排序：

```
#include <thrust/sort.h>
#include <thrust/execution_policy.h>
...
const int N = 6;
int A[N] = {1, 4, 2, 8, 5, 7};
thrust::stable_sort(thrust::host, A, A + N);
// A is now {1, 2, 4, 5, 7, 8}
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **RandomAccessIterator**：是 Random Access Iterator 的模型，RandomAccessIterator 是可变的。RandomAccessIterator 的 value_type 是 LessThan Comparable 的模型，如在 LessThan Comparable 要求中定义的，RandomAccessIterator 的 value_type 上的排序是 严格弱序

函数参数

- **exec**：用于并行化的执行策略
- **first**：序列起始
- **last**：序列末尾

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/stable_sort
- sort
- stable_sort_by_key

2.1.8.6 thrust::stable_sort

```
template <typename RandomAccessIterator>
void
stable_sort(RandomAccessIterator first,
  RandomAccessIterator last);
```

stable_sort 很像 sort，按升序对 [first, last) 内的元素进行排序，指的是如果 i 和 j 是 [first, last) 内任意满足 i 先于 j 的两个有效迭代器，那么 *j 不小于 *i。

顾名思义，stable_sort 是稳定的：它保留了等价元素的相对顺序。即，如果 x 和 y 是 [first, last) 中的元素，满足 x 先于 y，并且如果这两个元素是等价的（既不是 x < y 也不是 y < x），那么 stable_sort 的后置条件是 x 仍然先于 y。

此版本的 stable_sort 使用 operator< 比较对象。

以下代码片段演示了如何使用 sort 对整数序列进行排序：

```
#include <thrust/sort.h>
...
const int N = 6;
int A[N] = {1, 4, 2, 8, 5, 7};
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
thrust::stable_sort(A, A + N);
// A is now {1, 2, 4, 5, 7, 8}
```

模板参数

RandomAccessIterator：是 Random Access Iterator 的模型，RandomAccessIterator 是可变的。RandomAccessIterator 的 value_type 是 LessThan Comparable 的模型，如在 LessThan Comparable 要求中定义的，RandomAccessIterator 的 value_type 上的排序是 严格弱序

函数参数

- **first**：序列起始
- **last**：序列末尾

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/stable_sort
- sort
- stable_sort_by_key

2.1.8.7 thrust::stable_sort

```
template <typename DerivedPolicy,
  typename RandomAccessIterator,
  typename StrictWeakOrdering>
__host__ __device__ void
stable_sort(const thrust::detail::execution_policy_base< DerivedPolicy > &
↪exec,
  RandomAccessIterator first,
  RandomAccessIterator last,
  StrictWeakOrdering comp);
```

stable_sort 很像 sort，按升序对 [first, last) 内的元素进行排序，指的是如果 i 和 j 是 [first, last) 内任意满足 i 先于 j 的两个有效迭代器，那么 *j 不小于 *i。

顾名思义，stable_sort 是稳定的：它保留了等价元素的相对顺序。即，如果 x 和 y 是 [first, last) 中的元素，满足 x 先于 y，并且如果这两个元素是等价的（既不是 x < y 也不是 y < x），那么 stable_sort 的后置条件是 x 仍然先于 y。

此版本的 stable_sort 使用函数对象 comp 比较对象。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::host 并行化执行策略，使用大于比较运算符按降序对整数进行排序：

```
#include <thrust/sort.h>
#include <thrust/functional.h>
#include <thrust/execution_policy.h>
...
const int N = 6;
int A[N] = {1, 4, 2, 8, 5, 7};
thrust::sort(A, A + N, thrust::greater<int>());
// A is now {8, 7, 5, 4, 2, 1};
```

模板参数

- **DerivedPolicy**：派生执行策略的名称

沐曦信息 MetaX Confidential 2024-11-18 15:00:00

- **RandomAccessIterator**：是 Random Access Iterator 的模型，RandomAccessIterator ‘是可变的，并且 ‘RandomAccessIterator 的 value_type 可转换为 StrictWeakOrdering 的 first_argument_type 和 second_argument_type
- **StrictWeakOrdering**：是 Strict Weak Ordering 的模型

函数参数

- **exec**：用于并行化的执行策略
- **first**：序列起始
- **last**：序列末尾
- **comp**：比较运算符

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/stable_sort
- sort
- stable_sort_by_key

2.1.8.8 thrust::stable_sort

```
template <typename RandomAccessIterator,
  typename StrictWeakOrdering>
void
stable_sort(RandomAccessIterator first,
  RandomAccessIterator last,
  StrictWeakOrdering comp);
```

stable_sort 很像 sort，按升序对 [first, last) 内的元素进行排序，指的是如果 i 和 j 是 [first, last) 内任意满足 i 先于 j 的两个有效迭代器，那么 *j 不小于 *i。

顾名思义，stable_sort 是稳定的：它保留了等价元素的相对顺序。即，如果 x 和 y 是 [first, last) 中的元素，满足 x 先于 y，并且如果这两个元素是等价的（既不是 x < y 也不是 y < x），那么 stable_sort 的后置条件是 x 仍然先于 y。

此版本的 stable_sort 使用函数对象 comp 比较对象。

以下代码片段演示了如何使用大于比较运算符按降序对整数进行排序：

```
#include <thrust/sort.h>
#include <thrust/functional.h>
...
const int N = 6;
int A[N] = {1, 4, 2, 8, 5, 7};
thrust::sort(A, A + N, thrust::greater<int>());
// A is now {8, 7, 5, 4, 2, 1};
```

模板参数

- **RandomAccessIterator**：是 Random Access Iterator 的模型，RandomAccessIterator ‘是可变的，并且 ‘RandomAccessIterator 的 value_type 可转换为 StrictWeakOrdering 的 first_argument_type 和 second_argument_type
- **StrictWeakOrdering**：是 Strict Weak Ordering 的模型

函数参数

- **first**：序列起始
- **last**：序列末尾

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

- **comp**：比较运算符

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/stable_sort
- sort
- stable_sort_by_key

2.1.8.9 thrust::sort_by_key

```
template <typename DerivedPolicy,
  typename RandomAccessIterator1,
  typename RandomAccessIterator2>
__host__ __device__ void
sort_by_key(const thrust::detail::execution_policy_base< DerivedPolicy > &
↪exec,
  RandomAccessIterator1 keys_first,
  RandomAccessIterator1 keys_last,
  RandomAccessIterator2 values_first);
```

sort_by_key 执行键值排序。即，sort_by_key 将 [keys_first, keys_last) 和 [values_first, values_first + (keys_last - keys_first)) 内的元素按升序排序，指的是如果 i 和 j 是 [keys_first, keys_last) 内任意两个有效迭代器，其中 i 在 j 之前，且 p 和 q 是 [values_first, values_first + (keys_last - keys_first)) 内 i 和 j 分别对应的迭代器，则 *j 不小于 *i。

不能保证 sort_by_key 是稳定的。即，假设 *i 和 *j 是等价的（两者都不小于另一个），不能保证 sort_by_key 会保留这两个键的相对顺序或它们对应值的相对顺序。

此版本的 sort_by_key 使用 operator< 比较键对象。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::host 并行化执行策略，使用 sort_by_key 以整数作为排序键对字符值数组进行排序：

```
#include <thrust/sort.h>
#include <thrust/execution_policy.h>
...
const int N = 6;
int    keys[N] = {  1,   4,   2,   8,   5,   7};
char values[N] = {'a', 'b', 'c', 'd', 'e', 'f'};
thrust::sort_by_key(thrust::host, keys, keys + N, values);
// keys is now   {  1,   2,   4,   5,   7,   8}
// values is now {'a', 'c', 'b', 'e', 'f', 'd'}
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **RandomAccessIterator1**：是 Random Access Iterator 的模型，RandomAccessIterator1 是可变的。RandomAccessIterator1 的 value_type 是 LessThan Comparable 的模型，如在 LessThan Comparable 要求中定义的，RandomAccessIterator1 的 value_type 上的排序是 严格弱序
- **RandomAccessIterator2**：是 Random Access Iterator 的模型，且 RandomAccessIterator2 是可变的

函数参数

- **exec**：用于并行化的执行策略
- **keys_first**：键序列的起始

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

- **keys_last**：键序列的末尾
- **values_first**：值序列的起始

前提条件

范围 [keys_first, keys_last)) 不应与范围 [values_first, values_first + (keys_last - keys_first)) 重叠

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/sort
- stable_sort_by_key
- sort

2.1.8.10 thrust::sort_by_key

```
template <typename RandomAccessIterator1,
  typename RandomAccessIterator2>
void
sort_by_key(RandomAccessIterator1 keys_first,
  RandomAccessIterator1 keys_last,
  RandomAccessIterator2 values_first);
```

sort_by_key 执行键值排序。即，sort_by_key 将 [keys_first, keys_last) 和 [values_first, values_first + (keys_last - keys_first)) 内的元素按升序排序，指的是如果 i 和 j 是 [keys_first, keys_last) 内任意两个有效迭代器，其中 i 在 j 之前，且 p 和 q 是 [values_first, values_first + (keys_last - keys_first)) 内 i 和 j 分别对应的迭代器，则 *j 不小于 *i。

不能保证 sort_by_key 是稳定的。即，假设 *i 和 *j 是等价的（两者都不小于另一个），不能保证 sort_by_key 会保留这两个键的相对顺序或它们对应值的相对顺序。

此版本的 sort_by_key 使用 operator< 比较键对象。

以下代码片段演示了如何使用 sort_by_key 以整数作为排序键对字符值数组进行排序：

```
#include <thrust/sort.h>
...
const int N = 6;
int    keys[N] = {  1,   4,   2,   8,   5,   7};
char values[N] = {'a', 'b', 'c', 'd', 'e', 'f'};
thrust::sort_by_key(keys, keys + N, values);
// keys is now   {  1,   2,   4,   5,   7,   8}
// values is now {'a', 'c', 'b', 'e', 'f', 'd'}
```

模板参数

- **RandomAccessIterator1**：是 Random Access Iterator 的模型，RandomAccessIterator1 是可变的。RandomAccessIterator1 的 value_type 是 LessThan Comparable 的模型，如在 LessThan Comparable 要求中定义的，RandomAccessIterator1 的 value_type 上的排序是 严格弱序
- **RandomAccessIterator2**：是 Random Access Iterator 的模型，且 RandomAccessIterator2 是可变的

函数参数

- **keys_first**：键序列的起始
- **keys_last**：键序列的末尾
- **values_first**：值序列的起始

前提条件

范围 [keys_first, keys_last)) 不应与范围 [values_first, values_first + (keys_last - keys_first)) 重叠

飞腾信息 MetaX Confidential 2024-11-18 15:05:00

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/sort
- stable_sort_by_key
- sort

2.1.8.11 thrust::sort_by_key

```
template <typename DerivedPolicy,
  typename RandomAccessIterator1,
  typename RandomAccessIterator2,
  typename StrictWeakOrdering>
__host__ __device__ void
sort_by_key(const thrust::detail::execution_policy_base< DerivedPolicy > &
↪exec,
  RandomAccessIterator1 keys_first,
  RandomAccessIterator1 keys_last,
  RandomAccessIterator2 values_first,
  StrictWeakOrdering comp);
```

sort_by_key 执行键值排序。即，sort_by_key 将 [keys_first, keys_last) 和 [values_first, values_first + (keys_last - keys_first)) 内的元素按升序排序，指的是如果 i 和 j 是 [keys_first, keys_last) 内任意两个有效迭代器，其中 i 在 j 之前，且 p 和 q 是 [values_first, values_first + (keys_last - keys_first)) 内 i 和 j 分别对应的迭代器，则 *j 不小于 *i。

不能保证 sort_by_key 是稳定的。即，假设 *i 和 *j 是等价的(两者都不小于另一个)，不能保证 sort_by_key 会保留这两个键的相对顺序或它们对应值的相对顺序。

此版本的 sort_by_key 使用函数对象 comp 比较键对象。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::host 并行化执行策略，使用 sort_by_key 以整数作为排序键对字符值数组进行排序，并使用 greater<int> 比较运算符按降序对键进行排序：

```
#include <thrust/sort.h>
#include <thrust/execution_policy.h>
...
const int N = 6;
int    keys[N] = {  1,   4,   2,   8,   5,   7};
char values[N] = {'a', 'b', 'c', 'd', 'e', 'f'};
thrust::sort_by_key(thrust::host, keys, keys + N, values, thrust::greater
↪<int>());
// keys is now   {  8,   7,   5,   4,   2,   1}
// values is now {'d', 'f', 'e', 'b', 'c', 'a'}
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **RandomAccessIterator1**：是 Random Access Iterator 的模型，RandomAccessIterator1 是可变的，并且 RandomAccessIterator1 的 value_type 可转换为 StrictWeakOrdering 的 first_argument_type 和 second_argument_type
- **RandomAccessIterator2**：是 Random Access Iterator 的模型，且 RandomAccessIterator2 是可变的
- **StrictWeakOrdering**：是 Strict Weak Ordering 的模型

函数参数

- **exec**：用于并行化的执行策略

飞腾信息技术有限公司 MetaX Confidential 2024-11-18 15:00:00

- **keys_first**：键序列的起始
- **keys_last**：键序列的末尾
- **values_first**：值序列的起始
- **comp**：比较运算符

前提条件

范围 [keys_first, keys_last)) 不应与范围 [values_first, values_first + (keys_last - keys_first)) 重叠

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/sort
- stable_sort_by_key
- sort

2.1.8.12 thrust::sort_by_key

```
template <typename RandomAccessIterator1,
  typename RandomAccessIterator2,
  typename StrictWeakOrdering>
void
sort_by_key(RandomAccessIterator1 keys_first,
  RandomAccessIterator1 keys_last,
  RandomAccessIterator2 values_first,
  StrictWeakOrdering comp);
```

sort_by_key 执行键值排序。即，sort_by_key 将 [keys_first, keys_last) 和 [values_first, values_first + (keys_last - keys_first)) 内的元素按升序排序，指的是如果 i 和 j 是 [keys_first, keys_last) 内任意两个有效迭代器，其中 i 在 j 之前，且 p 和 q 是 [values_first, values_first + (keys_last - keys_first)) 内 i 和 j 分别对应的迭代器，则 *j 不小于 *i。

不能保证 sort_by_key 是稳定的。即，假设 *i 和 *j 是等价的（两者都不小于另一个），不能保证 sort_by_key 会保留这两个键的相对顺序或它们对应值的相对顺序。

此版本的 sort_by_key 使用函数对象 comp 比较键对象。

以下代码片段演示了如何使用 sort_by_key 以整数作为排序键对字符值数组进行排序，使用大于比较运算符按降序对键进行排序：

```
#include <thrust/sort.h>
...
const int N = 6;
int    keys[N] = {  1,   4,   2,   8,   5,   7};
char values[N] = {'a', 'b', 'c', 'd', 'e', 'f'};
thrust::sort_by_key(keys, keys + N, values, thrust::greater<int>());
// keys is now   {  8,   7,   5,   4,   2,   1}
// values is now {'d', 'f', 'e', 'b', 'c', 'a'}
```

模板参数

- **RandomAccessIterator1**：是 Random Access Iterator 的模型，RandomAccessIterator1 是可变的，并且 RandomAccessIterator1 的 value_type 可转换为 StrictWeakOrdering 的 first_argument_type 和 second_argument_type
- **RandomAccessIterator2**：是 Random Access Iterator 的模型，且 RandomAccessIterator2 是可变的
- **StrictWeakOrdering**：是 Strict Weak Ordering 的模型

函数参数

MetaX Confidential 2024-11-15 11:00:00

- **keys_first**：键序列的起始
- **keys_last**：键序列的末尾
- **values_first**：值序列的起始
- **comp**：比较运算符

前提条件

范围 [keys_first, keys_last)) 不应与范围 [values_first, values_first + (keys_last - keys_first)) 重叠

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/sort
- stable_sort_by_key
- sort

2.1.8.13 thrust::stable_sort_by_key

```
template <typename DerivedPolicy,
  typename RandomAccessIterator1,
  typename RandomAccessIterator2>
__host__ __device__ void
stable_sort_by_key(const thrust::detail::execution_policy_base<
↪DerivedPolicy > & exec,
  RandomAccessIterator1 keys_first,
  RandomAccessIterator1 keys_last,
  RandomAccessIterator2 values_first);
```

stable_sort_by_key 执行键值排序。即，stable_sort_by_key 将 [keys_first, keys_last) 和 [values_first, values_first + (keys_last - keys_first)) 内的元素按升序排序，指的是如果 i 和 j 是 [keys_first, keys_last) 内任意两个有效迭代器，其中 i 在 j 之前，且 p 和 q 是 [values_first, values_first + (keys_last - keys_first)) 内 i 和 j 分别对应的迭代器，则 *j 不小于 *i。

顾名思义，stable_sort_by_key 是稳定的：它保留了等价元素的相对顺序。即，如果 x 和 y 是 [keys_first, keys_last) 中的元素，满足 x 先于 y ，并且如果这两个元素是等价的（既不是 x < y 也不是 y < x），那么 stable_sort_by_key 的后置条件是 x 仍然先于 y。

此版本的 stable_sort_by_key 使用 operator< 比较键对象。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::host 并行化执行策略，使用 stable_sort_by_key 以整数作为排序键对字符数组进行排序：

```
#include <thrust/sort.h>
#include <thrust/execution_policy.h>
...
const int N = 6;
int    keys[N] = {  1,   4,   2,   8,   5,   7};
char values[N] = {'a', 'b', 'c', 'd', 'e', 'f'};
thrust::stable_sort_by_key(thrust::host, keys, keys + N, values);
// keys is now   {  1,   2,   4,   5,   7,   8}
// values is now {'a', 'c', 'b', 'e', 'f', 'd'}
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **RandomAccessIterator1**：是 Random Access Iterator 的模型，RandomAccessIterator1 是可变的。RandomAccessIterator1 的 value_type 是 LessThan Comparable 的模型，如在 LessThan Comparable 要求中定义的，RandomAccessIterator1 的 value_type 上的排序是 严格弱序

沐曦 MetaX Confidential 2024-11-18 15:00:00

- **RandomAccessIterator2**：是 Random Access Iterator 的模型，且 RandomAccessIterator2 是可变的

函数参数

- **exec**：用于并行化的执行策略
- **keys_first**：键序列的起始
- **keys_last**：键序列的末尾
- **values_first**：值序列的起始

前提条件

范围 [keys_first, keys_last)) 不应与范围 [values_first, values_first + (keys_last - keys_first)) 重叠

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/sort
- sort_by_key
- stable_sort

2.1.8.14 thrust::stable_sort_by_key

```
template <typename RandomAccessIterator1,
  typename RandomAccessIterator2>
void
stable_sort_by_key(RandomAccessIterator1 keys_first,
  RandomAccessIterator1 keys_last,
  RandomAccessIterator2 values_first);
```

stable_sort_by_key 执行键值排序。即，stable_sort_by_key 将 [keys_first, keys_last) 和 [values_first, values_first + (keys_last - keys_first)) 内的元素按升序排序，指的是如果 i 和 j 是 [keys_first, keys_last) 内任意两个有效迭代器，其中 i 在 j 之前，且 p 和 q 是 [values_first, values_first + (keys_last - keys_first)) 内 i 和 j 分别对应的迭代器，则 *j 不小于 *i。

顾名思义，stable_sort_by_key 是稳定的：它保留了等价元素的相对顺序。即，如果 x 和 y 是 [keys_first, keys_last) 中的元素，满足 x 先于 y，并且如果这两个元素是等价的（既不是 x < y 也不是 y < x），那么 stable_sort_by_key 的后置条件是 x 仍然先于 y。

此版本的 stable_sort_by_key 使用 operator< 比较键对象。以下代码片段演示了如何使用 stable_sort_by_key 以整数作为排序键对字符数组进行排序：

```
#include <thrust/sort.h>
...
const int N = 6;
int    keys[N] = {  1,   4,   2,   8,   5,   7};
char values[N] = {'a', 'b', 'c', 'd', 'e', 'f'};
thrust::stable_sort_by_key(keys, keys + N, values);
// keys is now   {  1,   2,   4,   5,   7,   8}
// values is now {'a', 'c', 'b', 'e', 'f', 'd'}
```

模板参数

- **RandomAccessIterator1**：是 Random Access Iterator 的模型，RandomAccessIterator1 是可变的。RandomAccessIterator1 的 value_type 是 LessThan Comparable 的模型，如在 LessThan Comparable 要求中定义的，RandomAccessIterator1 的 value_type 上的排序是 严格弱序
- **RandomAccessIterator2**：是 Random Access Iterator 的模型，且 RandomAccessIterator2 是可变的

函数参数

沐曦 MetaX Confidential 2024-11-18 15:00:00

- **keys_first**：键序列的起始
- **keys_last**：键序列的末尾
- **values_first**：值序列的起始

前提条件

范围 [keys_first, keys_last)) 不应与范围 [values_first, values_first + (keys_last - keys_first)) 重叠

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/sort
- sort_by_key
- stable_sort

2.1.8.15 thrust::stable_sort_by_key

```
template <typename DerivedPolicy,
  typename RandomAccessIterator1,
  typename RandomAccessIterator2,
  typename StrictWeakOrdering>
__host__ __device__ void
stable_sort_by_key(const thrust::detail::execution_policy_base<
↪DerivedPolicy > & exec,
  RandomAccessIterator1 keys_first,
  RandomAccessIterator1 keys_last,
  RandomAccessIterator2 values_first,
  StrictWeakOrdering comp);
```

stable_sort_by_key 执行键值排序。即，stable_sort_by_key 将 [keys_first, keys_last) 和 [values_first, values_first + (keys_last - keys_first)) 内的元素按升序排序，指的是如果 i 和 j 是 [keys_first, keys_last) 内任意两个有效迭代器，其中 i 在 j 之前，且 p 和 q 是 [values_first, values_first + (keys_last - keys_first)) 内 i 和 j 分别对应的迭代器，则 *j 不小于 *i。

顾名思义，stable_sort_by_key 是稳定的：它保留了等价元素的相对顺序。即，如果 x 和 y 是 [keys_first, keys_last) 中的元素，满足 x 先于 y ，并且如果这两个元素是等价的（既不是 x < y 也不是 y < x），那么 stable_sort_by_key 的后置条件是 x 仍然先于 y。

此版本的 stable_sort_by_key 使用函数对象 comp 比较键对象。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::host 并行化执行策略，使用 sort_by_key 以整数作为排序键对字符值数组进行排序：使用 greater<int> 比较运算符按降序对键进行排序。

```
#include <thrust/sort.h>
#include <thrust/execution_policy.h>
...
const int N = 6;
int    keys[N] = {  1,   4,   2,   8,   5,   7};
char values[N] = {'a', 'b', 'c', 'd', 'e', 'f'};
thrust::stable_sort_by_key(thrust::host, keys, keys + N, values,
↪thrust::greater<int>());
// keys is now   {  8,   7,   5,   4,   2,   1}
// values is now {'d', 'f', 'e', 'b', 'c', 'a'}
```

模板参数

- **DerivedPolicy**：派生执行策略的名称

沐曦 MetaX Confidential 2024-11-18 15:00:00

- **RandomAccessIterator1**：是 Random Access Iterator 的模型，RandomAccessIterator1 是可变的，并且 RandomAccessIterator1 的 value_type 可转换为 StrictWeakOrdering 的 first_argument_type 和 second_argument_type
- **RandomAccessIterator2**：是 Random Access Iterator 的模型，且 RandomAccessIterator2 是可变的
- **StrictWeakOrdering**：是 Strict Weak Ordering 的模型

函数参数

- **exec**：用于并行化的执行策略
- **keys_first**：键序列的起始
- **keys_last**：键序列的末尾
- **values_first**：值序列的起始
- **comp**：比较运算符

前提条件

范围 [keys_first, keys_last)) 不应与范围 [values_first, values_first + (keys_last - keys_first)) 重叠

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/sort
- sort_by_key
- stable_sort

2.1.8.16 thrust::stable_sort_by_key

```
template <typename RandomAccessIterator1,
  typename RandomAccessIterator2,
  typename StrictWeakOrdering>
void
stable_sort_by_key(RandomAccessIterator1 keys_first,
  RandomAccessIterator1 keys_last,
  RandomAccessIterator2 values_first,
  StrictWeakOrdering comp);
```

stable_sort_by_key 执行键值排序。即，stable_sort_by_key 将 [keys_first, keys_last) 和 [values_first, values_first + (keys_last - keys_first)) 内的元素按升序排序，指的是如果 i 和 j 是 [keys_first, keys_last) 内任意两个有效迭代器，其中 i 在 j 之前，且 p 和 q 是 [values_first, values_first + (keys_last - keys_first)) 内 i 和 j 分别对应的迭代器，则 *j 不小于 *i。

顾名思义，stable_sort_by_key 是稳定的：它保留了等价元素的相对顺序。即，如果 x 和 y 是 [keys_first, keys_last) 中的元素，满足 x 先于 y ，并且如果这两个元素是等价的（既不是 x < y 也不是 y < x），那么 stable_sort_by_key 的后置条件是 x 仍然先于 y。

此版本的 stable_sort_by_key 使用函数对象 comp 比较键对象。

以下代码片段演示了如何使用 sort_by_key 以整数作为排序键对字符值数组进行排序，使用大于比较运算符按降序对键进行排序：

```
#include <thrust/sort.h>
...
const int N = 6;
int    keys[N] = {  1,   4,   2,   8,   5,   7};
char values[N] = {'a', 'b', 'c', 'd', 'e', 'f'};
thrust::stable_sort_by_key(keys, keys + N, values, thrust::greater<int>());
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
// keys is now   {  8,   7,   5,   4,   2,   1}
// values is now {'d', 'f', 'e', 'b', 'c', 'a'}
```

模板参数

- **RandomAccessIterator1**：是 Random Access Iterator 的模型，RandomAccessIterator1 是可变的，并且 RandomAccessIterator1 的 value_type 可转换为 StrictWeakOrdering 的 first_argument_type 和 second_argument_type
- **RandomAccessIterator2**：是 Random Access Iterator 的模型，且 RandomAccessIterator2 是可变的
- **StrictWeakOrdering**：是 Strict Weak Ordering 的模型

函数参数

- **keys_first**：键序列的起始
- **keys_last**：键序列的末尾
- **values_first**：值序列的起始
- **comp**：比较运算符

前提条件

范围 [keys_first, keys_last)) 不应与范围 [values_first, values_first + (keys_last - keys_first)) 重叠

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/sort
- sort_by_key
- stable_sort

2.1.9 变换（Transformation）

2.1.9.1 变换

函数

```
template <typename DerivedPolicy,
  typename InputIterator,
  typename OutputIterator>
__host__ __device__ OutputIterator
thrust::adjacent_difference(const thrust::detail::execution_policy_base<
↪DerivedPolicy > & exec,
  InputIterator first,
  InputIterator last,
  OutputIterator result);

template <typename DerivedPolicy,
  typename InputIterator,
  typename OutputIterator,
  typename BinaryFunction>
__host__ __device__ OutputIterator
thrust::adjacent_difference(const thrust::detail::execution_policy_base<
↪DerivedPolicy > & exec,
  InputIterator first,
  InputIterator last,
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
  OutputIterator result,
  BinaryFunction binary_op);

template <typename InputIterator,
  typename OutputIterator>
OutputIterator
thrust::adjacent_difference(InputIterator first,
  InputIterator last,
  OutputIterator result);

template <typename InputIterator,
  typename OutputIterator,
  typename BinaryFunction>
OutputIterator
thrust::adjacent_difference(InputIterator first,
  InputIterator last,
  OutputIterator result,
  BinaryFunction binary_op);

template <typename DerivedPolicy,
  typename ForwardIterator,
  typename Generator>
__host__ __device__ void
thrust::generate(const thrust::detail::execution_policy_base<
 ↪DerivedPolicy > & exec,
  ForwardIterator first,
  ForwardIterator last,
  Generator gen);

template <typename ForwardIterator,
  typename Generator>
void
thrust::generate(ForwardIterator first,
  ForwardIterator last,
  Generator gen);

template <typename DerivedPolicy,
  typename OutputIterator,
  typename Size,
  typename Generator>
__host__ __device__ OutputIterator
thrust::generate_n(const thrust::detail::execution_policy_base<
 ↪DerivedPolicy > & exec,
  OutputIterator first,
  Size n,
  Generator gen);

template <typename OutputIterator,
  typename Size,
  typename Generator>
OutputIterator
thrust::generate_n(OutputIterator first,
  Size n,
  Generator gen);

template <typename DerivedPolicy,
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
  typename ForwardIterator>
__host__ __device__ void
thrust::sequence(const thrust::detail::execution_policy_base<
↪DerivedPolicy > & exec,
  ForwardIterator first,
  ForwardIterator last);

template <typename ForwardIterator>
void
thrust::sequence(ForwardIterator first,
  ForwardIterator last);

template <typename DerivedPolicy,
  typename ForwardIterator,
  typename T>
__host__ __device__ void
thrust::sequence(const thrust::detail::execution_policy_base<
↪DerivedPolicy > & exec,
  ForwardIterator first,
  ForwardIterator last,
  T init);

template <typename ForwardIterator,
  typename T>
void
thrust::sequence(ForwardIterator first,
  ForwardIterator last,
  T init);

template <typename DerivedPolicy,
  typename ForwardIterator,
  typename T>
__host__ __device__ void
thrust::sequence(const thrust::detail::execution_policy_base<
↪DerivedPolicy > & exec,
  ForwardIterator first,
  ForwardIterator last,
  T init,
  T step);

template <typename ForwardIterator,
  typename T>
void
thrust::sequence(ForwardIterator first,
  ForwardIterator last,
  T init,
  T step);

template <typename DerivedPolicy,
  typename ForwardIterator,
  typename UnaryOperation>
__host__ __device__ void
thrust::tabulate(const thrust::detail::execution_policy_base<
↪DerivedPolicy > & exec,
  ForwardIterator first,
  ForwardIterator last,
```

(下页继续)

飞谱信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
  UnaryOperation unary_op);

template <typename ForwardIterator,
  typename UnaryOperation>
void
thrust::tabulate(ForwardIterator first,
  ForwardIterator last,
  UnaryOperation unary_op);

template <typename DerivedPolicy,
  typename InputIterator,
 typename OutputIterator,
  typename UnaryFunction>
__host__ __device__ OutputIterator
thrust::transform(const thrust::detail::execution_policy_base<
 ↪DerivedPolicy > & exec,
  InputIterator first,
  InputIterator last,
  OutputIterator result,
  UnaryFunction op);

template <typename InputIterator,
  typename OutputIterator,
  typename UnaryFunction>
OutputIterator
thrust::transform(InputIterator first,
  InputIterator last,
  OutputIterator result,
  UnaryFunction op);

template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator,
  typename BinaryFunction>
__host__ __device__ OutputIterator
thrust::transform(const thrust::detail::execution_policy_base<
 ↪DerivedPolicy > & exec,
  InputIterator1 first1,
  InputIterator1 last1,
  InputIterator2 first2,
  OutputIterator result,
  BinaryFunction op);

template <typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator,
  typename BinaryFunction>
OutputIterator
thrust::transform(InputIterator1 first1,
  InputIterator1 last1,
  InputIterator2 first2,
  OutputIterator result,
  BinaryFunction op);

template <typename DerivedPolicy,
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
  typename InputIterator,
  typename ForwardIterator,
  typename UnaryFunction,
  typename Predicate>
__host__ __device__ ForwardIterator
thrust::transform_if(const thrust::detail::execution_policy_base<
 ↪DerivedPolicy > & exec,
  InputIterator first,
  InputIterator last,
  ForwardIterator result,
  UnaryFunction op,
  Predicate pred);

template <typename InputIterator,
  typename ForwardIterator,
  typename UnaryFunction,
  typename Predicate>
ForwardIterator
thrust::transform_if(InputIterator first,
  InputIterator last,
  ForwardIterator result,
  UnaryFunction op,
  Predicate pred);

template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2,
  typename ForwardIterator,
  typename UnaryFunction,
  typename Predicate>
__host__ __device__ ForwardIterator
thrust::transform_if(const thrust::detail::execution_policy_base<
 ↪DerivedPolicy > & exec,
  InputIterator1 first,
  InputIterator1 last,
  InputIterator2 stencil,
  ForwardIterator result,
  UnaryFunction op,
  Predicate pred);

template <typename InputIterator1,
  typename InputIterator2,
  typename ForwardIterator,
  typename UnaryFunction,
  typename Predicate>
ForwardIterator
thrust::transform_if(InputIterator1 first,
  InputIterator1 last,
  InputIterator2 stencil,
  ForwardIterator result,
  UnaryFunction op,
 Predicate pred);

template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2,
```

(下页继续)

飞曦软件 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
  typename InputIterator3,
  typename ForwardIterator,
  typename BinaryFunction,
  typename Predicate>
__host__ __device__ ForwardIterator
thrust::transform_if(const thrust::detail::execution_policy_base<
 ↪DerivedPolicy > & exec,
  InputIterator1 first1,
  InputIterator1 last1,
  InputIterator2 first2,
  InputIterator3 stencil,
  ForwardIterator result,
  BinaryFunction binary_op,
  Predicate pred);

template <typename InputIterator1,
  typename InputIterator2,
  typename InputIterator3,
  typename ForwardIterator,
  typename BinaryFunction,
  typename Predicate>
orwardIterator
thrust::transform_if(InputIterator1 first1,
  InputIterator1 last1,
  InputIterator2 first2,
  InputIterator3 stencil,
  ForwardIterator result,
  BinaryFunction binary_op,
  Predicate pred);
```

thrust::adjacent_difference

```
template <typename DerivedPolicy,
  typename InputIterator,
  typename OutputIterator>
__host__ __device__ OutputIterator
adjacent_difference(const thrust::detail::execution_policy_base<
 ↪DerivedPolicy > & exec,
  InputIterator first,
  InputIterator last,
  OutputIterator result);
```

adjacent_difference 计算 [first, last) 内相邻元素的差。即，将 *first 赋给 *result ，并且对于范围 [first + 1, last) 内的每个迭代器 i ，将 *i 和 *(i - 1) 的差赋给 *(result + (i - first))。

此版本的 adjacent_difference 使用 operator- 计算差。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::device 执行策略，使用 adjacent_difference 计算范围内相邻元素的差：

```
#include <thrust/adjacent_difference.h>
#include <thrust/device_vector.h>
#include <thrust/execution_policy.h>
...
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
int h_data[8] = {1, 2, 1, 2, 1, 2, 1, 2};
thrust::device_vector<int> d_data(h_data, h_data + 8);
thrust::device_vector<int> d_result(8);

thrust::adjacent_difference(thrust::device, d_data.begin(), d_data.end(),
↪d_result.begin());

// d_result is now [1, 1, -1, 1, -1, 1, -1, 1]
```

备注

允许 result 与 first 是同一个迭代器。这对于就地计算差非常有用。

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **InputIterator**：是 Input Iterator 的模型，x 和 y 是 InputIterator 的 value_type 的对象，定义 x - is。InputIterator 的 value_type 可转换为 OutputIterator 的 value_types 集合中的类型，x - y 的返回类型可转换为 OutputIterator 的 value_types 集合中的类型
- **OutputIterator**：是 Output Iterator 的模型

函数参数

- **exec**：用于并行化的执行策略
- **first**：输入范围的起始
- **last**：输入范围的末尾
- **result**：输出范围的起始

返回值

迭代器 result + (last - first)

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/adjacent_difference
- inclusive_scan

thrust::adjacent_difference

```
template <typename DerivedPolicy,
  typename InputIterator,
  typename OutputIterator,
  typename BinaryFunction>
__host__ __device__ OutputIterator
adjacent_difference(const thrust::detail::execution_policy_base<
↪DerivedPolicy > & exec,
  InputIterator first,
  InputIterator last,
  OutputIterator result,
  BinaryFunction binary_op);
```

adjacent_difference 计算 [first, last) 内相邻元素的差。即，将 *first 赋给 *result ，并且对于范围 [first + 1, last) 内的每个迭代器 i ，将 binary_op(*i, *(i - 1)) 赋给 *(result + (i - first))。

此版本的 adjacent_difference 使用二元函数 binary_op 计算差。

算法的执行由 exec 确定并行化。

飞腾信息 MetaX Confidential 2024-11-12 15:00:00

以下代码片段演示了如何通过 thrust::device 执行策略，使用 adjacent_difference 计算范围内相邻元素的和：

```
#include <thrust/adjacent_difference.h>
#include <thrust/functional.h>
#include <thrust/device_vector.h>
#include <thrust/execution_policy.h>
...
int h_data[8] = {1, 2, 1, 2, 1, 2, 1, 2};
thrust::device_vector<int> d_data(h_data, h_data + 8);
thrust::device_vector<int> d_result(8);

thrust::adjacent_difference(thrust::device, d_data.begin(), d_data.end(),
 ↪d_result.begin(), thrust::plus<int>());

// d_result is now [1, 3, 3, 3, 3, 3, 3, 3]
```

备注

允许 result 与 first 是同一个迭代器。这对于就地计算差非常有用。

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **InputIterator**：是 Input Iterator 的模型，InputIterator 的 value_type 可转换为 BinaryFunction 的 first_argument_type 和 second_argument_type ，以及 OutputIterator 的 value_types 集合中的类型
- **OutputIterator**：是 Output Iterator 的模型
- **BinaryFunction 的** result_type：可转换为 OutputIterator 的 value_types 集合中的类型

函数参数

- **exec**：用于并行化的执行策略
- **first**：输入范围的起始
- **last**：输入范围的末尾
- **result**：输出范围的起始
- **binary_op**：计算差的二元函数

返回值

迭代器 result + (last - first)

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/adjacent_difference
- inclusive_scan

thrust::adjacent_difference

```
template <typename InputIterator,
  typename OutputIterator>
OutputIterator
adjacent_difference(InputIterator first,
  InputIterator last,
  OutputIterator result);
```

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

adjacent_difference 计算 [first, last) 内相邻元素的差。即，将 *first 赋给 *result ，并且对于范围 [first + 1, last) 内的每个迭代器 i ，将 *i 和 *(i - 1) 的差赋给 *(result + (i - first))。

此版本的 adjacent_difference 使用 operator- 计算差。

以下代码片段演示了如何使用 adjacent_difference 计算范围内相邻元素的差：

```
#include <thrust/adjacent_difference.h>
#include <thrust/device_vector.h>
...
int h_data[8] = {1, 2, 1, 2, 1, 2, 1, 2};
thrust::device_vector<int> d_data(h_data, h_data + 8);
thrust::device_vector<int> d_result(8);

thrust::adjacent_difference(d_data.begin(), d_data.end(), d_result.
↪begin());

// d_result is now [1, 1, -1, 1, -1, 1, -1, 1]
```

备注

允许 result 与 first 是同一个迭代器。这对于就地计算差非常有用。

模板参数

- **InputIterator**：是 Input Iterator 的模型，x 和 y 是 InputIterator 的 value_type 的对象，定义 x - is。InputIterator 的 value_type 可转换为 OutputIterator 的 value_types 集合中的类型，x - y 的返回类型可转换为 OutputIterator 的 value_types 集合中的类型
- **OutputIterator**：是 Output Iterator 的模型

函数参数

- **first**：输入范围的起始
- **last**：输入范围的末尾
- **result**：输出范围的起始

返回值

迭代器 result + (last - first)

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/adjacent_difference
- inclusive_scan

thrust::adjacent_difference

```
template <typename InputIterator,
  typename OutputIterator,
  typename BinaryFunction>
OutputIterator
adjacent_difference(InputIterator first,
  InputIterator last,
  OutputIterator result,
  BinaryFunction binary_op);
```

adjacent_difference 计算 [first, last) 内相邻元素的差。即，将 *first 赋给 *result ，并且对于范围 [first + 1, last) 内的每个迭代器 i ，将 binary_op(*i, *(i - 1)) 赋给 *(result + (i - first))。

此版本的 adjacent_difference 使用二元函数 binary_op 计算差。

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

以下代码片段演示了如何使用 adjacent_difference 计算范围内相邻元素的和：

```
#include <thrust/adjacent_difference.h>
#include <thrust/functional.h>
#include <thrust/device_vector.h>
...
int h_data[8] = {1, 2, 1, 2, 1, 2, 1, 2};
thrust::device_vector<int> d_data(h_data, h_data + 8);
thrust::device_vector<int> d_result(8);

thrust::adjacent_difference(d_data.begin(), d_data.end(), d_result.begin(),
↪ thrust::plus<int>());

// d_result is now [1, 3, 3, 3, 3, 3, 3, 3]
```

备注

允许 result 与 first 是同一个迭代器。这对于就地计算差非常有用。

模板参数

- **InputIterator1**：是 Input Iterator 的模型，InputIterator 的 value_type 可转换为 BinaryFunction 的 first_argument_type 和 second_argument_type ，以及 OutputIterator 的 value_types 集合中的类型
- **OutputIterator**：是 Output Iterator 的模型
- **BinaryFunction 的** result_type：可转换为 OutputIterator 的 value_types 集合中的类型

函数参数

- **first**：输入范围的起始
- **last**：输入范围的末尾
- **result**：输出范围的起始
- **binary_op**：计算差的二元函数

返回值

迭代器 result + (last - first)

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/adjacent_difference
- inclusive_scan

thrust::generate

```
template <typename DerivedPolicy,
  typename ForwardIterator,
  typename Generator>
__host__ __device__ void
generate(const thrust::detail::execution_policy_base< DerivedPolicy > &
↪exec,
  ForwardIterator first,
  ForwardIterator last,
  Generator gen);
```

generate 将调用 gen （无参函数对象）的结果赋给范围 [first,last) 内每个元素。

算法的执行由 exec 确定并行化。

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

以下代码片段演示了如何通过 thrust::host 并行化执行策略，使用标准的 C 库函数 rand 用随机数填充 host_vector：

```
#include <thrust/generate.h>
#include <thrust/host_vector.h>
#include <thrust/execution_policy.h>
#include <cstdlib>
...
thrust::host_vector<int> v(10);
srand(13);
thrust::generate(thrust::host, v.begin(), v.end(), rand);

// the elements of v are now pseudo-random numbers
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **ForwardIterator**：是 Forward Iterator 的模型，且 ForwardIterator 是可变的
- **Generator**：是 Generator 的模型，Generator 的 result_type 可转换为 ForwardIterator 的 value_type

函数参数

- **exec**：用于并行化的执行策略
- **first**：感兴趣范围中的第一个元素
- **last**：感兴趣范围中的最后一个元素
- **gen**：无参函数对象，用于生成要赋给范围 [first,last) 内元素的值

另请参阅

- generate_n
- https://en.cppreference.com/w/cpp/algorithm/generate

thrust::generate

```
template <typename ForwardIterator,
  typename Generator>
void
generate(ForwardIterator first,
  ForwardIterator last,
   Generator gen);
```

generate 将调用 gen（无参函数对象）的结果赋给范围 [first,last) 内每个元素。

以下代码片段演示了如何使用标准的 C 库函数 rand 用随机数填充 host_vector：

```
#include <thrust/generate.h>
#include <thrust/host_vector.h>
#include <thrust/execution_policy.h>
#include <cstdlib>
...
thrust::host_vector<int> v(10);
srand(13);
thrust::generate(v.begin(), v.end(), rand);

// the elements of v are now pseudo-random numbers
```

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

模板参数

- **ForwardIterator**：是 Forward Iterator 的模型，且 ForwardIterator 是可变的
- **Generator**：是 Generator 的模型，Generator 的 result_type 可转换为 ForwardIterator 的 value_type

函数参数

- **first**：感兴趣范围中的第一个元素
- **last**：感兴趣范围中的最后一个元素
- **gen**：无参函数对象，用于生成要赋给范围 [first,last) 内元素的值

另请参阅

- generate_n
- https://en.cppreference.com/w/cpp/algorithm/generate

thrust::generate_n

```
template <typename DerivedPolicy,
  typename OutputIterator,
  typename Size,
  typename Generator>
__host__ __device__ OutputIterator
generate_n(const thrust::detail::execution_policy_base< DerivedPolicy > &
↪exec,
  OutputIterator first,
  Size n,
  Generator gen);
```

generate_n 将调用 gen（无参函数对象）的结果赋给范围 [first,first + n) 内每个元素。

返回值是 first + n。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::host 并行化执行策略，使用标准的 C 库函数 rand 用随机数填充 host_vector：

```
#include <thrust/generate.h>
#include <thrust/host_vector.h>
#include <thrust/execution_policy.h>
#include <cstdlib>
...
thrust::host_vector<int> v(10);
srand(13);
thrust::generate_n(thrust::host, v.begin(), 10, rand);

// the elements of v are now pseudo-random numbers
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **OutputIterator**：是 Output Iterator 的模型
- **Size**：整型类型（有符号或无符号）
- **Generator**：是 Generator 的模型，Generator 的 result_type 可转换为 OutputIterator 的 value_types 集合中的类型

飞谱信息 MetaX Confidential 2024-11-18 15:00:00

函数参数

- **exec**：用于并行化的执行策略
- **first**：感兴趣范围中的第一个元素
- **n**：感兴趣范围的大小
- **gen**：无参函数对象，用于生成要赋给范围 [first,first + n) 内元素的值

另请参阅

- generate
- https://en.cppreference.com/w/cpp/algorithm/generate

thrust::generate_n

```
template <typename OutputIterator,
  typename Size,
  typename Generator>
OutputIterator
generate_n(OutputIterator first,
  Size n,
  Generator gen);
```

generate_n 将调用 gen （无参函数对象）的结果赋给范围 [first,first + n) 内每个元素。返回值是 first + n。

以下代码片段演示了如何使用标准的 C 库函数 rand 用随机数填充 host_vector：

```
#include <thrust/generate.h>
#include <thrust/host_vector.h>
#include <stdlib.h>
...
thrust::host_vector<int> v(10);
srand(13);
thrust::generate_n(v.begin(), 10, rand);

// the elements of v are now pseudo-random numbers
```

模板参数

- **OutputIterator**：是 Output Iterator 的模型
- **Size**：整型类型（有符号或无符号）
- **Generator**：是 Generator 的模型，Generator 的 result_type 可转换为 OutputIterator 的 value_types 集合中的类型

函数参数

- **first**：感兴趣范围中的第一个元素
- **n**：感兴趣范围的大小
- **gen**：无参函数对象，用于生成要赋给范围 [first,first + n) 内元素的值

另请参阅

- generate
- https://en.cppreference.com/w/cpp/algorithm/generate

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

thrust::sequence

```
template <typename DerivedPolicy,
  typename ForwardIterator>
__host__ __device__ void
sequence(const thrust::detail::execution_policy_base< DerivedPolicy > &
 ↪exec,
  ForwardIterator first,
  ForwardIterator last);
```

sequence 用数字序列填充范围 [first, last)。

对于范围 [first, last) 内的每个迭代器 i ，此版本的 sequence 执行赋值 *i = (i - first)。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::host 并行化执行策略，使用 sequence 用数字序列填充范围：

```
#include <thrust/sequence.h>
#include <thrust/execution_policy.h>
...
const int N = 10;
int A[N];
thrust::sequence(thrust::host, A, A + 10);
// A is now {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

备注

与类似的 C++STL 函数 std::iota 不同，sequence 不能保证执行顺序。

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **ForwardIterator**：是 Forward Iterator 的模型，ForwardIterator 是可变的。如果 x 和 y 是 ForwardIterator 的 value_type 的对象，则定义 x + y；如果 T 是 ForwardIterator 的 value_type ，则定义 T(0)

函数参数

- **exec**：用于并行化的执行策略
- **first**：序列起始
- **last**：序列末尾

另请参阅

https://en.cppreference.com/w/cpp/algorithm/iota

thrust::sequence

```
template <typename ForwardIterator>
void
sequence(ForwardIterator first,
  ForwardIterator last);
```

sequence 用数字序列填充范围 [first, last)。

对于范围 [first, last) 内的每个迭代器 i ，此版本的 sequence 执行赋值 *i = (i - first)。

以下代码片段演示了如何使用 sequence 用数字序列填充范围：

飞曦信息 MetaX Confidential 2024-11-18 15:00:00

```
#include <thrust/sequence.h>
...
const int N = 10;
int A[N];
thrust::sequence(A, A + 10);
// A is now {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

备注

与类似的 C++STL 函数 std::iota 不同，sequence 不能保证执行顺序。

模板参数

ForwardIterator：是 Forward Iterator 的模型，ForwardIterator 是可变的。如果 x 和 y 是 ForwardIterator 的 value_type 的对象，则定义 x + y；如果 T 是 ForwardIterator 的 value_type，则定义 T(0)

函数参数

- **first**：序列起始
- **last**：序列末尾

另请参阅

https://en.cppreference.com/w/cpp/algorithm/iota

thrust::sequence

```
template <typename DerivedPolicy,
  typename ForwardIterator,
  typename T>
__host__ __device__ void
sequence(const thrust::detail::execution_policy_base< DerivedPolicy > &
↪exec,
  ForwardIterator first,
  ForwardIterator last,
  T init);
```

sequence 用数字序列填充范围 [first, last)。

对于范围 [first, last) 内的每个迭代器 i，此版本的 sequence 执行赋值 *i = init + (i - first)。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::host 并行化执行策略，使用 sequence 用从 1 开始的数字序列填充范围：

```
#include <thrust/sequence.h>
#include <thrust/execution_policy.h>
...
const int N = 10;
int A[N];
thrust::sequence(thrust::host, A, A + 10, 1);
// A is now {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
```

备注

与类似的 C++STL 函数 std::iota 不同，sequence 不能保证执行顺序。

模板参数

- **DerivedPolicy**：派生执行策略的名称

飞曦信息 MetaX Confidential 2024-11-18 15:00:00

- **ForwardIterator**：是 Forward Iterator 的模型，ForwardIterator 是可变的。如果 x 和 y 是 ForwardIterator 的 value_type 的对象，则定义 x + y；如果 T 是 ForwardIterator 的 value_type，则定义 T(0)
- **T**：是 Assignable 的模型，且 T 可转换为 ForwardIterator 的 value_type

函数参数

- **exec**：用于并行化的执行策略
- **first**：序列起始
- **last**：序列末尾
- **init**：数字序列的第一个值

另请参阅

https://en.cppreference.com/w/cpp/algorithm/iota

thrust::sequence

```
template <typename ForwardIterator,
  typename T>
void
sequence(ForwardIterator first,
  ForwardIterator last,
  T init);
```

sequence 用数字序列填充范围 [first, last)。

对于范围 [first, last) 内的每个迭代器 i，此版本的 sequence 执行赋值 *i = init + (i - first)。

以下代码片段演示了如何使用 sequence 用从 1 开始的数字序列填充范围：

```
#include <thrust/sequence.h>
...
const int N = 10;
int A[N];
thrust::sequence(A, A + 10, 1);
// A is now {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
```

备注

与类似的 C++STL 函数 std::iota 不同，sequence 不能保证执行顺序。

模板参数

- **ForwardIterator**：是 Forward Iterator 的模型，ForwardIterator 是可变的。如果 x 和 y 是 ForwardIterator 的 value_type 的对象，则定义 x + y；如果 T 是 ForwardIterator 的 value_type，则定义 T(0)
- **T**：是 Assignable 的模型，且 T 可转换为 ForwardIterator 的 value_type

函数参数

- **first**：序列起始
- **last**：序列末尾
- **init**：数字序列的第一个值

另请参阅

https://en.cppreference.com/w/cpp/algorithm/iota

飞书信息 MetaX Confidential 2024-11-18 15:00:00

thrust::sequence

```
template <typename DerivedPolicy,
  typename ForwardIterator,
  typename T>
__host__ __device__ void
sequence(const thrust::detail::execution_policy_base< DerivedPolicy > &
↪exec,
  ForwardIterator first,
  ForwardIterator last,
  T init,
  T step);
```

sequence 用数字序列填充范围 [first, last)。

对于范围 [first, last) 内的每个迭代器 i ，此版本的 sequence 执行赋值 *i = init + step * (i - first)。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::host 并行化执行策略，使用 sequence 用从 1 开始，步长为 3 的数字序列填充范围：

```
#include <thrust/sequence.h>
#include <thrust/execution_policy.h>
...
const int N = 10;
int A[N];
thrust::sequence(thrust::host, A, A + 10, 1, 3);
// A is now {1, 4, 7, 10, 13, 16, 19, 22, 25, 28}
```

备注

与类似的 C++STL 函数 std::iota 不同，sequence 不能保证执行顺序。

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **ForwardIterator**：是 Forward Iterator 的模型，ForwardIterator 是可变的。如果 x 和 y 是 ForwardIterator 的 value_type 的对象，则定义 x + y；如果 T 是 ForwardIterator 的 value_type ，则定义 T(0)
- **T**：是 Assignable 的模型，且 T 可转换为 ForwardIterator 的 value_type

函数参数

- **exec**：用于并行化的执行策略
- **first**：序列起始
- **last**：序列末尾
- **init**：数字序列的第一个值
- **step**：连续元素之间的差

另请参阅

https://en.cppreference.com/w/cpp/algorithm/iota

thrust::sequence

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

```
template <typename ForwardIterator,
  typename T>
void
sequence(ForwardIterator first,
  ForwardIterator last,
  T init,
  T step);
```

sequence 用数字序列填充范围 [first, last)。

对于范围 [first, last) 内的每个迭代器 i ，此版本的 sequence 执行赋值 *i = init + step * (i - first)。

以下代码片段演示了如何使用 sequence 用从 1 开始，步长为 3 的数字序列填充范围：

```
#include <thrust/sequence.h>
...
const int N = 10;
int A[N];
thrust::sequence(A, A + 10, 1, 3);
// A is now {1, 4, 7, 10, 13, 16, 19, 22, 25, 28}
```

备注

与类似的 C++STL 函数 std::iota 不同，sequence 不能保证执行顺序。

模板参数

- **ForwardIterator**：是 Forward Iterator 的模型，ForwardIterator 是可变的。如果 x 和 y 是 ForwardIterator 的 value_type 的对象，则定义 x + y；如果 T 是 ForwardIterator 的 value_type ，则定义 T(0)
- **T**：是 Assignable 的模型，且 T 可转换为 ForwardIterator 的 value_type

函数参数

- **first**：序列起始
- **last**：序列末尾
- **init**：数字序列的第一个值
- **step**：连续元素之间的差

另请参阅

https://en.cppreference.com/w/cpp/algorithm/iota

thrust::tabulate

```
template <typename DerivedPolicy,
  typename ForwardIterator,
  typename UnaryOperation>
__host__ __device__ void
tabulate(const thrust::detail::execution_policy_base< DerivedPolicy > &
↪exec,
  ForwardIterator first,
  ForwardIterator last,
  UnaryOperation unary_op);
```

tabulate 使用应用于每个元素索引的函数值填充范围 [first, last)。

对于范围 [first, last) 中的每个迭代器 i ，tabulate 执行赋值 *i = unary_op(i - first)。

算法的执行由 exec 确定并行化。

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

以下代码片段演示了如何通过 thrust::host 并行化执行策略，使用 tabulate 生成前 n 个非正整数：

```
#include <thrust/tabulate.h>
#include <thrust/functional.h>
#include <thrust/execution_policy.h>
...
const int N = 10;
int A[N];
thrust::tabulate(thrust::host, A, A + 10, thrust::negate<int>());
// A is now {0, -1, -2, -3, -4, -5, -6, -7, -8, -9}
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **ForwardIterator**：是 Forward Iterator 的模型，ForwardIterator 是可变的。如果 x 和 y 是 ForwardIterator 的 value_type 的对象，则定义 x + y；如果 T 是 ForwardIterator 的 value_type，则定义 T(0)
- **UnaryOperation**：是 Unary Function 的模型，UnaryFunction 的 result_type 可转换为 OutputIterator 的 value_type

函数参数

- **exec**：用于并行化的执行策略
- **first**：范围起始
- **last**：范围末尾
- **unary_op**：待应用的一元运算

另请参阅

- thrust::fill
- thrust::generate
- thrust::sequence

thrust::tabulate

```
template <typename ForwardIterator,
  typename UnaryOperation>
void
tabulate(ForwardIterator first,
  ForwardIterator last,
  UnaryOperation unary_op);
```

tabulate 使用应用于每个元素索引的函数值填充范围 [first, last)。

对于范围 [first, last) 中的每个迭代器 i，tabulate 执行赋值 *i = unary_op(i - first)。

以下代码片段演示了如何使用 tabulate 生成前 n 个非正整数：

```
#include <thrust/tabulate.h>
#include <thrust/functional.h>
...
const int N = 10;
int A[N];
thrust::tabulate(A, A + 10, thrust::negate<int>());
// A is now {0, -1, -2, -3, -4, -5, -6, -7, -8, -9}
```

模板参数

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

- **ForwardIterator**：是 Forward Iterator 的模型，ForwardIterator 是可变的。如果 x 和 y 是 ForwardIterator 的 value_type 的对象，则定义 x + y；如果 T 是 ForwardIterator 的 value_type，则定义 T(0)
- **UnaryOperation**：是 Unary Function 的模型，UnaryFunction 的 result_type 可转换为 OutputIterator 的 value_type

函数参数

- **first**：范围起始
- **last**：范围末尾
- **unary_op**：待应用的一元运算

另请参阅

- thrust::fill
- thrust::generate
- thrust::sequence

thrust::transform

```
template <typename DerivedPolicy,
  typename InputIterator,
  typename OutputIterator,
  typename UnaryFunction>
__host__ __device__ OutputIterator
transform(const thrust::detail::execution_policy_base< DerivedPolicy > &
↪exec,
  InputIterator first,
  InputIterator last,
  OutputIterator result,
  UnaryFunction op);
```

此版本的 transform 将一元函数应用于输入序列的每个元素，并将结果存储在输出序列的相应位置。具体而言，对于范围 [first, last) 内的每个迭代器 i，执行运算 op(*i)，并将结果赋给 *o，其中 o 是范围 [result, result + (last - first)) 内的相应输出迭代器。输入和输出序列可以重合，从而产生就地变换。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::host 并行化执行策略，使用 transform 就地取一个范围的相反数：

```
#include <thrust/transform.h>
#include <thrust/functional.h>
#include <thrust/execution_policy.h>
...

int data[10] = {-5, 0, 2, -3, 2, 4, 0, -1, 2, 8};

thrust::negate<int> op;

thrust::transform(thrust::host, data, data + 10, data, op); // in-place
↪transformation

// data is now {5, 0, -2, 3, -2, -4, 0, 1, -2, -8};
```

模板参数

- **DerivedPolicy**：派生执行策略的名称

沐曦信息 MetaX Confidential 2024-11-18 15:00:00

- **InputIterator**：是 Input Iterator 的模型，且 InputIterator 的 value_type 可转换为 UnaryFunction 的 value_type
- **OutputIterator**：是 Output Iterator 的模型
- **UnaryFunction**：是 Unary Function 的模型，UnaryFunction 的 result_type 可转换为 OutputIterator 的 value_type

函数参数

- **exec**：用于并行化的执行策略
- **first**：输入序列的起始
- **last**：输入序列的末尾
- **result**：输出序列的起始
- **op**：变换运算

前提条件

first 可以等于 result ，但除此外，范围 [first, last) 不应与范围 [result, result + (last - first)) 重叠

返回值

输出序列的末尾

另请参阅

https://en.cppreference.com/w/cpp/algorithm/transform

thrust::transform

```
template <typename InputIterator,
  typename OutputIterator,
  typename UnaryFunction>
OutputIterator
transform(InputIterator first,
  InputIterator last,
  OutputIterator result,
  UnaryFunction op);
```

此版本的 transform 将一元函数应用于输入序列的每个元素，并将结果存储在输出序列的相应位置。具体而言，对于范围 [first, last) 内的每个迭代器 i，执行运算 op(*i)，并将结果赋给 *o，其中 o 是范围 [result, result + (last - first)) 内的相应输出迭代器。输入和输出序列可以重合，从而产生就地变换。

以下代码片段演示了如何使用 transform。

```
#include <thrust/transform.h>
#include <thrust/functional.h>

int data[10] = {-5, 0, 2, -3, 2, 4, 0, -1, 2, 8};

thrust::negate<int> op;

thrust::transform(data, data + 10, data, op); // in-place transformation

// data is now {5, 0, -2, 3, -2, -4, 0, 1, -2, -8};
```

模板参数

- **InputIterator**：是 Input Iterator 的模型，InputIterator 的 value_type 可转换为 UnaryFunction 的 argument_type

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

- **OutputIterator**：是 Output Iterator 的模型
- **UnaryFunction**：是 Unary Function 的模型，UnaryFunction 的 result_type 可转换为 OutputIterator 的 value_type

函数参数

- **first**：输入序列的起始
- **last**：输入序列的末尾
- **result**：输出序列的起始
- **op**：变换运算

前提条件

first 可以等于 result ，但除此外，范围 [first, last) 不应与范围 [result, result + (last - first)) 重叠

返回值

输出序列的末尾

另请参阅

https://en.cppreference.com/w/cpp/algorithm/transform

thrust::transform

```
template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator,
  typename BinaryFunction>
__host__ __device__ OutputIterator
transform(const thrust::detail::execution_policy_base< DerivedPolicy > &
↪exec,
  InputIterator1 first1,
  InputIterator1 last1,
  InputIterator2 first2,
  OutputIterator result,
  BinaryFunction op);
```

此版本的 transform 将二元函数应用于两个输入序列的每对元素，并将结果存储在输出序列中的相应位置。具体而言，对于范围 [first1, last1) 内的每个迭代器 i 和 [first2, last2) 内的 j = first + (i - first1)，执行运算 op(*i,*j)，并将结果赋给 *o，其中 o 是范围 [result, result + (last - first)) 内的相应输出迭代器。输入和输出序列可以重合，从而产生就地变换。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::host 并行化执行策略，使用 transform 计算两个范围的和：

```
#include <thrust/transform.h>
#include <thrust/functional.h>
#include <thrust/execution_policy.h>
...

int input1[6] = {-5,  0,  2,  3,  2,  4};
int input2[6] = { 3,  6, -2,  1,  2,  3};
int output[6];

thrust::plus<int> op;
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
thrust::transform(thrust::host, input1, input1 + 6, input2, output, op);

// output is now {-2,  6,  0,  4,  4,  7};
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **InputIterator1**：是 Input Iterator 的模型，InputIterator1 的 value_type 可转换为 BinaryFunction 的 first_argument_type
- **InputIterator2**：是 Input Iterator 的模型，InputIterator2 的 value_type 可转换为 BinaryFunction 的 second_argument_type
- **OutputIterator**：是 Output Iterator 的模型
- **BinaryFunction**：是 Binary Function 的模型，BinaryFunction 的 result_type 可转换为 OutputIterator 的 value_type

函数参数

- **exec**：用于并行化的执行策略
- **first1**：第一个输入序列的起始
- **last1**：第一个输入序列的末尾
- **first2**：第二个输入序列的起始
- **result**：输出序列的起始
- **op**：变换运算

前提条件

- first1 可以等于 result，但除此外，范围 [first1, last1) 不应与范围 [result, result + (last1 - first1)) 重叠
- first2 可以等于 result，但除此外，范围 [first2, first2 + (last1 - first1)) 不应与范围 [result, result + (last1 - first1)) 重叠

返回值

输出序列的末尾

另请参阅

https://en.cppreference.com/w/cpp/algorithm/transform

thrust::transform

```
template <typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator,
  typename BinaryFunction>
OutputIterator
transform(InputIterator1 first1,
  InputIterator1 last1,
  InputIterator2 first2,
  OutputIterator result,
  BinaryFunction op);
```

此版本的 transform 将二元函数应用于两个输入序列的每对元素，并将结果存储在输出序列中的相应位置。具体而言，对于范围 [first1, last1) 内的每个迭代器 i 和 [first2, last2) 内的 j = first + (i - first1)，执行运算 op(*i,*j)，并将结果赋给 *o，其中 o 是范围 [result, result + (last - first)) 内的相应输出迭代器。输入和输出序列可以重合，从而产生就地变换。

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

以下代码片段演示了如何使用 transform。

```
#include <thrust/transform.h>
#include <thrust/functional.h>

int input1[6] = {-5,  0,  2,  3,  2,  4};
int input2[6] = { 3,  6, -2,  1,  2,  3};
int output[6];

thrust::plus<int> op;

thrust::transform(input1, input1 + 6, input2, output, op);

// output is now {-2,  6,  0,  4,  4,  7};
```

模板参数

- **InputIterator1**：是 Input Iterator 的模型，InputIterator1 的 value_type 可转换为 BinaryFunction 的 first_argument_type
- **InputIterator2**：是 Input Iterator 的模型，InputIterator2 的 value_type 可转换为 BinaryFunction 的 second_argument_type
- **OutputIterator**：是 Output Iterator 的模型
- **BinaryFunction**：是 Binary Function 的模型，BinaryFunction 的 result_type 可转换为 OutputIterator 的 value_type

函数参数

- **first1**：第一个输入序列的起始
- **last1**：第一个输入序列的末尾
- **first2**：第二个输入序列的起始
- **result**：输出序列的起始
- **op**：变换运算

前提条件

- first1 可以等于 result ，但除此外，范围 [first1, last1) 不应与范围 [result, result + (last1 - first1)) 重叠
- first2 可以等于 result ，但除此外，范围 [first2, first2 + (last1 - first1)) 不应与范围 [result, result + (last1 - first1)) 重叠

返回值

输出序列的末尾

另请参阅

https://en.cppreference.com/w/cpp/algorithm/transform

thrust::transform_if

```
template <typename DerivedPolicy,
  typename InputIterator,
  typename ForwardIterator,
  typename UnaryFunction,
  typename Predicate>
__host__ __device__ ForwardIterator
transform_if(const thrust::detail::execution_policy_base< DerivedPolicy > &
↪ exec,
```

(下页继续)

飞曦信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
  InputIterator first,
  InputIterator last,
  ForwardIterator result,
  UnaryFunction op,
  Predicate pred);
```

此版本的 transform_if 有条件地将一元函数应用于输入序列的每个元素，并且如果输入序列中的对应位置满足谓词，则将结果存储在输出序列中的相应位置。否则，输出序列中的相应位置不会被修改。

具体而言，对于范围 [first, last) 内的每个迭代器 i ，对谓词 pred(*i) 进行求值如果此谓词的计算结果为 true ，则将 op(*i) 的结果赋给 *o ，其中 o 是范围 [result, result + (last - first)) 内的相应输出迭代器。否则，不对 op(*i) 进行求值，也不进行赋值。输入和输出序列可以重合，从而产生就地变换。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::host 并行化执行策略，使用 transform_if 取范围内奇值元素的相反数：

```
#include <thrust/transform.h>
#include <thrust/functional.h>
#include <thrust/execution_policy.h>
...

int data[10]    = {-5, 0, 2, -3, 2, 4, 0, -1, 2, 8};

struct is_odd
{
__host__ __device__
bool operator()(int x)
{
return x % 2;
}
};

thrust::negate<int> op;
thrust::identity<int> identity;

// negate odd elements
thrust::transform_if(thrust::host, data, data + 10, data, op, is_odd()); //
↪ in-place transformation

// data is now {5, 0, 2, 3, 2, 4, 0, 1, 2, 8};
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **InputIterator**：是 Input Iterator 的模型，并且 InputIterator 的 value_type 可转换为 Predicate 的 argument_type 和 UnaryFunction 的 argument_type
- **ForwardIterator**：是 Forward Iterator 的模型
- **UnaryFunction**：是 Unary Function 的模型，UnaryFunction 的 result_type 可转换为 OutputIterator 的 value_type
- **Predicate**：是 Predicate 的模型

函数参数

- **exec**：用于并行化的执行策略
- **first**：输入序列的起始

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

- **last**：输入序列的末尾
- **result**：输出序列的起始
- **op**：变换运算
- **pred**：谓词运算

前提条件

first 可以等于 result ，但除此外，范围 [first, last) 不应与范围 [result, result + (last - first)) 重叠

返回值

输出序列的末尾

另请参阅

thrust::transform

thrust::transform_if

```
template <typename InputIterator,
  typename ForwardIterator,
  typename UnaryFunction,
  typename Predicate>
ForwardIterator
transform_if(InputIterator first,
  InputIterator last,
  ForwardIterator result,
  UnaryFunction op,
  Predicate pred);
```

此版本的 transform_if 有条件地将一元函数应用于输入序列的每个元素，并且如果输入序列中的对应位置满足谓词，则将结果存储在输出序列中的相应位置。否则，输出序列中的相应位置不会被修改。

具体而言，对于范围 [first, last) 内的每个迭代器 i ，对谓词 pred(*i) 进行求值如果此谓词的计算结果为 true ，则将 op(*i) 的结果赋给 *o ，其中 o 是范围 [result, result + (last - first)) 内的相应输出迭代器。否则，不对 op(*i) 进行求值，也不进行赋值。输入和输出序列可以重合，从而产生就地变换。

以下代码片段演示了如何使用 transform_if。

```
#include <thrust/transform.h>
#include <thrust/functional.h>

int data[10]    = {-5, 0, 2, -3, 2, 4, 0, -1, 2, 8};

struct is_odd
{
__host__ __device__
bool operator()(int x)
{
return x % 2;
}
};

thrust::negate<int> op;
thrust::identity<int> identity;

// negate odd elements
thrust::transform_if(data, data + 10, data, op, is_odd()); // in-place
↪transformation
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-13 15:00:00

(续上页)

```
// data is now {5, 0, 2, 3, 2, 4, 0, 1, 2, 8};
```

模板参数

- **InputIterator**：是 Input Iterator 的模型，并且 InputIterator 的 value_type 可转换为 Predicate 的 argument_type 和 UnaryFunction 的 argument_type
- **ForwardIterator**：是 Forward Iterator 的模型
- **UnaryFunction**：是 Unary Function 的模型，UnaryFunction 的 result_type 可转换为 OutputIterator 的 value_type
- **Predicate**：是 Predicate 的模型

函数参数

- **first**：输入序列的起始
- **last**：输入序列的末尾
- **result**：输出序列的起始
- **op**：变换运算
- **pred**：谓词运算

前提条件

first 可以等于 result ，但除此外，范围 [first, last) 不应与范围 [result, result + (last - first)) 重叠

返回值

输出序列的末尾

另请参阅

thrust::transform

thrust::transform_if

```
template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2,
  typename ForwardIterator,
  typename UnaryFunction,
  typename Predicate>
__host__ __device__ ForwardIterator
transform_if(const thrust::detail::execution_policy_base< DerivedPolicy > &
↪ exec,
  InputIterator1 first,
  InputIterator1 last,
  InputIterator2 stencil,
  ForwardIterator result,
  UnaryFunction op,
  Predicate pred);
```

此版本的 transform_if 有条件地将一元函数应用于输入序列的每个元素，并且如果输入序列中的对应位置满足谓词，则将结果存储在输出序列中的相应位置。否则，输出序列中的相应位置不会被修改。

具体而言，对于范围 [first, last) 内的每个迭代器 i ，对 pred(*s) 进行求值，其中 s 是范围 [stencil, stencil + (last - first)) 内相应的输入迭代器。如果此谓词的计算结果为 true ，则将 op(*i) 的结果赋给 *o ，其中 o 是范围 [result, result + (last - first)) 内的相应输出迭代器。否则，不对 op(*i) 进行求值，也不进行赋值。输入和输出序列可以重合，从而产生就地变换。

飞曦信息 MetaX Confidential 2021-11-18 15:00:00

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::host 并行化执行策略，使用 transform_if：

```
#include <thrust/transform.h>
#include <thrust/functional.h>
#include <thrust/execution_policy.h>
...

int data[10]    = {-5, 0, 2, -3, 2, 4, 0, -1, 2, 8};
int stencil[10] = { 1, 0, 1,  0, 1, 0, 1,  0, 1, 0};

thrust::negate<int> op;
thrust::identity<int> identity;

thrust::transform_if(thrust::host, data, data + 10, stencil, data, op,
↪identity); // in-place transformation

// data is now {5, 0, -2, -3, -2,  4, 0, -1, -2,  8};
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **InputIterator1**：是 Input Iterator 的模型，InputIterator1 的 value_type 可转换为 UnaryFunction 的 argument_type
- **InputIterator2**：是 Input Iterator 的模型，InputIterator2 的 value_type 可转换为 Predicate 的 argument_type
- **ForwardIterator**：是 Forward Iterator 的模型
- **UnaryFunction**：是 Unary Function 的模型，UnaryFunction 的 result_type 可转换为 OutputIterator 的 value_type
- **Predicate**：是 Predicate 的模型

函数参数

- **exec**：用于并行化的执行策略
- **first**：输入序列的起始
- **last**：输入序列的末尾
- **stencil**：模版序列的起始
- **result**：输出序列的起始
- **op**：变换运算
- **pred**：谓词运算

前提条件

- first 可以等于 result，但除此外，范围 [first, last) 不应与范围 [result, result + (last - first)) 重叠
- stencil 可以等于 result，但除此外，范围 [stencil, stencil + (last - first)) 不应与范围 [result, result + (last - first)) 重叠

返回值

输出序列的末尾

另请参阅

thrust::transform

沐曦信息 MetaX Confidential 2024-11-18 15:00:00

thrust::transform_if

```
template <typename InputIterator1,
  typename InputIterator2,
  typename ForwardIterator,
  typename UnaryFunction,
  typename Predicate>
ForwardIterator
transform_if(InputIterator1 first,
  InputIterator1 last,
  InputIterator2 stencil,
  ForwardIterator result,
  UnaryFunction op,
  Predicate pred);
```

此版本的 transform_if 有条件地将一元函数应用于输入序列的每个元素，并且如果输入序列中的对应位置满足谓词，则将结果存储在输出序列中的相应位置。否则，输出序列中的相应位置不会被修改。

具体而言，对于范围 [first, last) 内的每个迭代器 i，对 pred(*s) 进行求值，其中 s 是范围 [stencil, stencil + (last - first)) 内相应的输入迭代器。如果此谓词的计算结果为 true，则将 op(*i) 的结果赋给 *o，其中 o 是范围 [result, result + (last - first)) 内的相应输出迭代器。否则，不计算 op(*i)，也不进行赋值。输入和输出序列可以重合，从而产生就地变换。

以下代码片段演示了如何使用 transform_if。

```
#include <thrust/transform.h>
#include <thrust/functional.h>

int data[10]    = {-5, 0, 2, -3, 2, 4, 0, -1, 2, 8};
int stencil[10] = { 1, 0, 1,  0, 1, 0, 1,  0, 1, 0};

thrust::negate<int> op;
thrust::identity<int> identity;

thrust::transform_if(data, data + 10, stencil, data, op, identity); // in-
↪place transformation

// data is now {5, 0, -2, -3, -2,  4, 0, -1, -2,  8};
```

模板参数

- **InputIterator1**：是 Input Iterator 的模型，InputIterator1 的 value_type 可转换为 UnaryFunction 的 argument_type
- **InputIterator2**：是 Input Iterator 的模型，InputIterator2 的 value_type 可转换为 Predicate 的 argument_type
- **ForwardIterator**：是 Forward Iterator 的模型
- **UnaryFunction**：是 Unary Function 的模型，UnaryFunction 的 result_type 可转换为 OutputIterator 的 value_type
- **Predicate**：是 Predicate 的模型

函数参数

- **first**：输入序列的起始
- **last**：输入序列的末尾
- **stencil**：模版序列的起始
- **result**：输出序列的起始

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

- **op**：变换运算
- **pred**：谓词运算

前提条件

- first 可以等于 result ，但除此外，范围 [first, last) 不应与范围 [result, result + (last - first)) 重叠
- stencil 可以等于 result ，但除此外，范围 [stencil, stencil + (last - first)) 不应与范围 [result, result + (last - first)) 重叠

返回值

输出序列的末尾

另请参阅

thrust::transform

thrust::transform_if

```
template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2,
  typename InputIterator3,
  typename ForwardIterator,
  typename BinaryFunction,
  typename Predicate>
__host__ __device__ ForwardIterator
transform_if(const thrust::detail::execution_policy_base< DerivedPolicy > &
↪ exec,
  InputIterator1 first1,
  InputIterator1 last1,
  InputIterator2 first2,
  InputIterator3 stencil,
  ForwardIterator result,
  BinaryFunction binary_op,
  Predicate pred);
```

此版本的 transform_if 有条件地将二元函数应用于两个输入序列的每对元素，并且如果模板序列中的对应位置满足谓词，则将结果存储在输出序列中的相应位置。否则，输出序列中的相应位置不会被修改。

具体而言，对于范围 [first1, last1) 内的每个迭代器 i 和 [first2, first2 + (last1 - first1)) 内的 j = first2 + (i - first1) ，对 pred(*s) 进行求值，其中 s 是范围 [stencil, stencil + (last1 - first1)) 内相应的输入迭代器。如果此谓词的计算结果为 true ，则将 binary_op(*i,*j) 的结果赋给 *o ，其中 o 是范围 [result, result + (last1 - first1)) 内的相应输出迭代器。否则，不对 binary_op(*i,*j) 进行求值，也不进行赋值。输入和输出序列可以重合，从而产生就地变换。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::host 并行化执行策略，使用 transform_if：

```
#include <thrust/transform.h>
#include <thrust/functional.h>
#include <thrust/execution_policy.h>
...

int input1[6]   = {-5,  0,  2,  3,  2,  4};
int input2[6]   = { 3,  6, -2,  1,  2,  3};
int stencil[8] = { 1,  0,  1,  0,  1,  0};
int output[6];
```

(下页继续)

(续上页)

```
thrust::plus<int> op;
thrust::identity<int> identity;

thrust::transform_if(thrust::host, input1, input1 + 6, input2, stencil,
↪output, op, identity);

// output is now {-2,  0,  0,  3,  4,  4};
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **InputIterator1**：是 Input Iterator 的模型，InputIterator1 的 value_type 可转换为 BinaryFunction 的 first_argument_type
- **InputIterator2**：是 Input Iterator 的模型，InputIterator2 的 value_type 可转换为 BinaryFunction 的 second_argument_type
- **ForwardIterator**：是 Forward Iterator 的模型
- **BinaryFunction**：是 Binary Function 的模型，BinaryFunction 的 result_type 可转换为 OutputIterator 的 value_type
- **Predicate**：是 Predicate 的模型

函数参数

- **exec**：用于并行化的执行策略
- **first1**：第一个输入序列的起始
- **last1**：第一个输入序列的末尾
- **first2**：第二个输入序列的起始
- **stencil**：模版序列的起始
- **result**：输出序列的起始
- **binary_op**：变换运算
- **pred**：谓词运算

前提条件

- first1 可以等于 result ，但除此外，范围 [first1, last1) 不应与范围 [result, result + (last1 - first1)) 重叠
- first2 可以等于 result ，但除此外，范围 [first2, first2 + (last1 - first1)) 不应与范围 [result, result + (last1 - first1)) 重叠
- stencil 可以等于 result ，但除此外，范围 [stencil, stencil + (last1 - first1)) 不应与范围 [result, result + (last1 - first1)) 重叠

返回值

输出序列的末尾

另请参阅

thrust::transform

thrust::transform_if

沐曦信息 MetaX Confidential 2024-11-18 15:00:00

```
template <typename InputIterator1,
  typename InputIterator2,
  typename InputIterator3,
  typename ForwardIterator,
  typename BinaryFunction,
  typename Predicate>
ForwardIterator
transform_if(InputIterator1 first1,
  InputIterator1 last1,
  InputIterator2 first2,
  InputIterator3 stencil,
  ForwardIterator result,
  BinaryFunction binary_op,
  Predicate pred);
```

此版本的 transform_if 有条件地将二元函数应用于两个输入序列的每对元素，并且如果模板序列中的对应位置满足谓词，则将结果存储在输出序列中的相应位置。否则，输出序列中的相应位置不会被修改。

具体而言，对于范围 [first1, last1) 内的每个迭代器 i 和 [first2, first2 + (last1 - first1)) 内的 j = first2 + (i - first1) ，对 pred(*s) 进行求值，其中 s 是范围 [stencil, stencil + (last1 - first1)) 内相应的输入迭代器。如果此谓词的计算结果为 true ，则将 binary_op(*i,*j) 的结果赋给 *o ，其中 o 是范围 [result, result + (last1 - first1)) 内的相应输出迭代器。否则，不对 binary_op(*i,*j) 进行求值，也不进行赋值。输入和输出序列可以重合，从而产生就地变换。

以下代码片段演示了如何使用 transform_if。

```
#include <thrust/transform.h>
#include <thrust/functional.h>

int input1[6]   = {-5,  0,  2,  3,  2,  4};
int input2[6]   = { 3,  6, -2,  1,  2,  3};
int stencil[8] = { 1,  0,  1,  0,  1,  0};
int output[6];

thrust::plus<int> op;
thrust::identity<int> identity;

thrust::transform_if(input1, input1 + 6, input2, stencil, output, op,
↪identity);

// output is now {-2,  0,  0,  3,  4,  4};
```

模板参数

- **InputIterator1**：是 Input Iterator 的模型，InputIterator1 的 value_type 可转换为 BinaryFunction 的 first_argument_type
- **InputIterator2**：是 Input Iterator 的模型，InputIterator2 的 value_type 可转换为 BinaryFunction 的 second_argument_type
- **ForwardIterator**：是 Forward Iterator 的模型
- **BinaryFunction**：是 Binary Function 的模型，BinaryFunction 的 result_type 可转换为 OutputIterator 的 value_type
- **Predicate**：是 Predicate 的模型

函数参数

- **first1**：第一个输入序列的起始
- **last1**：第一个输入序列的末尾

飞曦信息 MetaX Confidential 2024-11-18 15:00:07

- **first2**：第二个输入序列的起始
- **stencil**：模版序列的起始
- **result**：输出序列的起始
- **binary_op**：变换运算
- **pred**：谓词运算

前提条件

- first1 可以等于 result，但除此外，范围 [first1, last1) 不应与范围 [result, result + (last1 - first1)) 重叠
- first2 可以等于 result，但除此外，范围 [first2, first2 + (last1 - first1)) 不应与范围 [result, result + (last1 - first1)) 重叠
- stencil 可以等于 result，但除此外，范围 [stencil, stencil + (last1 - first1)) 不应与范围 [result, result + (last1 - first1)) 重叠

返回值

输出序列的末尾

另请参阅

thrust::transform

2.1.9.2 填充（Filling）

函数

```
template <typename DerivedPolicy,
  typename ForwardIterator,
  typename T>
__host__ __device__ void
thrust::fill(const thrust::detail::execution_policy_base< DerivedPolicy > &
↪ exec,
  ForwardIterator first,
  ForwardIterator last,
  const T & value);

template <typename ForwardIterator,
  typename T>
__host__ __device__ void
thrust::fill(ForwardIterator first,
  ForwardIterator last,
  const T & value);

template <typename DerivedPolicy,
  typename OutputIterator,
  typename Size,   typename T>
__host__ __device__ OutputIterator
thrust::fill_n(const thrust::detail::execution_policy_base< DerivedPolicy >
↪ & exec,
  OutputIterator first,
  Size n,
  const T & value);

template <typename OutputIterator,
  typename Size,
  typename T>
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
__host__ __device__ OutputIterator
thrust::fill_n(OutputIterator first,
  Size n,
  const T & value);

template <typename DerivedPolicy,
  typename ForwardIterator,
  typename T>
__host__ __device__ void
thrust::uninitialized_fill(const thrust::detail::execution_policy_base<
↪DerivedPolicy > & exec,
  ForwardIterator first,
  ForwardIterator last,
  const T & x);

template <typename ForwardIterator,
  typename T>
void
thrust::uninitialized_fill(ForwardIterator first,
  ForwardIterator last,
  const T & x);

template <typename DerivedPolicy,
  typename ForwardIterator,
  typename Size,
  typename T>
__host__ __device__ ForwardIterator
thrust::uninitialized_fill_n(const thrust::detail::execution_policy_base<
↪DerivedPolicy > & exec,
  ForwardIterator first,
  Size n,
  const T & x);

template <typename ForwardIterator,
  typename Size,
  typename T>
ForwardIterator
thrust::uninitialized_fill_n(ForwardIterator first,
  Size n,
  const T & x);
```

thrust::fill

```
template <typename DerivedPolicy,
  typename ForwardIterator,
  typename T>
__host__ __device__ void
fill(const thrust::detail::execution_policy_base< DerivedPolicy > & exec,
  ForwardIterator first,
  ForwardIterator last,
  const T & value);
```

fill 将值 value 赋给范围 [first, last) 内的每个元素。即，对于 [first, last) 内的每个迭代器 i，它执行赋值 *i = value。

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::device 并行化执行策略，使用 fill 将 thrust::device_vector 的元素设置为给定值：

```
#include <thrust/fill.h>
#include <thrust/device_vector.h>
#include <thrust/execution_policy.h>
...
thrust::device_vector<int> v(4);
thrust::fill(thrust::device, v.begin(), v.end(), 137);

// v[0] == 137, v[1] == 137, v[2] == 137, v[3] == 137
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **ForwardIterator**：是 Forward Iterator 的模型，且 ForwardIterator 是可变的
- **T**：是 Assignable 的模型，T 的 value_type 可转换为 ForwardIterator 的 value_type

函数参数

- **exec**：用于并行化的执行策略
- **first**：序列起始
- **last**：序列末尾
- **value**：要复制的值

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/fill
- fill_n
- uninitialized_fill

thrust::fill

```
template <typename ForwardIterator,
  typename T>
__host__ __device__ void
fill(ForwardIterator first,
  ForwardIterator last,
  const T & value);
```

fill 将值 value 赋给范围 [first, last) 内的每个元素。即，对于 [first, last) 内的每个迭代器 i，它执行赋值 *i = value。

以下代码片段演示了如何使用 fill 将 thrust::device_vector 的元素设置为给定值：

```
#include <thrust/fill.h>
#include <thrust/device_vector.h>
...
thrust::device_vector<int> v(4);
thrust::fill(v.begin(), v.end(), 137);

// v[0] == 137, v[1] == 137, v[2] == 137, v[3] == 137
```

模板参数

- **ForwardIterator**：是 Forward Iterator 的模型，且 ForwardIterator 是可变的

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

- **T**：是 Assignable 的模型，T 的 value_type 可转换为 ForwardIterator 的 value_type

函数参数

- **first**：序列起始
- **last**：序列末尾
- **value**：要复制的值

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/fill
- fill_n
- uninitialized_fill

thrust::fill_n

```
template <typename DerivedPolicy,
  typename OutputIterator,
  typename Size,
  typename T>
__host__ __device__ OutputIterator
fill_n(const thrust::detail::execution_policy_base< DerivedPolicy > & exec,
  OutputIterator first,
  Size n,
  const T & value);
```

fill_n 将值 value 赋给范围 [first, first+n) 内的每个元素。即，对于 [first, first+n) 内的每个迭代器 i ，它执行赋值 *i = value。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::device 并行化执行策略，使用 fill 将 thrust::device_vector 的元素设置为给定值：

```
#include <thrust/fill.h>
#include <thrust/device_vector.h>
#include <thrust/execution_policy.h>
...
thrust::device_vector<int> v(4);
thrust::fill_n(thrust::device, v.begin(), v.size(), 137);

// v[0] == 137, v[1] == 137, v[2] == 137, v[3] == 137
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **OutputIterator**：是 Output Iterator 的模型
- **Size**：是一个整数类型
- **T**：是 Assignable 的模型，T 的 value_type 可转换为 OutputIterator 的 value_type 集合中的类型

函数参数

- **exec**：用于并行化的执行策略
- **first**：序列起始
- **n**：序列的大小
- **value**：要复制的值

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

返回值

first + n

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/fill_n
- fill
- uninitialized_fill_n

thrust::fill_n

```
template <typename OutputIterator,
  typename Size,
  typename T>
__host__ __device__ OutputIterator
fill_n(OutputIterator first,
  Size n,
  const T & value);
```

fill_n 将值 value 赋给范围 [first, first+n) 内的每个元素。即，对于 [first, first+n) 内的每个迭代器 i ，它执行赋值 *i = value。

以下代码片段演示了如何使用 fill 将 thrust::device_vector 的元素设置为给定值：

```
#include <thrust/fill.h>
#include <thrust/device_vector.h>
...
thrust::device_vector<int> v(4);
thrust::fill_n(v.begin(), v.size(), 137);

// v[0] == 137, v[1] == 137, v[2] == 137, v[3] == 137
```

模板参数

- **OutputIterator**：是 Output Iterator 的模型
- **Size**：是一个整数类型
- **T**：是 Assignable 的模型，T 的 value_type 可转换为 OutputIterator 的 value_type 集合中的类型

函数参数

- **first**：序列起始
- **n**：序列的大小
- **value**：要复制的值

返回值

first + n

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/fill_n
- fill
- uninitialized_fill_n

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

thrust::uninitialized_fill

```
template <typename DerivedPolicy,
  typename ForwardIterator,
  typename T>
__host__ __device__ void
uninitialized_fill(const thrust::detail::execution_policy_base<
↪DerivedPolicy > & exec,
  ForwardIterator first,
  ForwardIterator last,
  const T & x);
```

在 thrust 中，函数 thrust::device_new 为对象分配内存，然后通过调用构造函数在该位置创建对象。然而，偶尔将这两个操作分开非常有用。如果范围 [first, last) 内的每个迭代器都指向未初始化的内存，则 uninitialized_fill 在该范围内创建 x 的副本。即，对于 [first, last) 范围内的每个迭代器 i，uninitialized_fill 通过调用 ForwardIterator 的 value_type 的拷贝构造函数在指向 i 的位置创建 x 的副本。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::device 并行化执行策略，使用 uninitialized_fill 初始化未初始化的内存范围：

```
#include <thrust/uninitialized_fill.h>
#include <thrust/device_malloc.h>
#include <thrust/execution_policy.h>

struct Int
{
__host__ __device__
Int(int x) : val(x) {}
int val;
};
...
const int N = 137;

Int val(46);
thrust::device_ptr<Int> array = thrust::device_malloc<Int>(N);
thrust::uninitialized_fill(thrust::device, array, array + N, val);

// Int x = array[i];
// x.val == 46 for all 0 <= i < N
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **ForwardIterator**：是 Forward Iterator 的模型，ForwardIterator 是可变的，并且 ForwardIterator 的 value_type 有一个采用类型 T 单参数的构造函数
- **T**：是 Assignable 的模型，T 的 value_type 可转换为 OutputIterator 的 value_type 集合中的类型

函数参数

- **exec**：用于并行化的执行策略
- **first**：感兴趣范围内的第一个元素
- **last**：感兴趣范围内的最后一个元素
- **x**：用于拷贝构造函数示例的值

另请参阅

- https://en.cppreference.com/w/cpp/memory/uninitialized_fill

飞燃信息 MetaX Confidential 2024-11-18 15:00:00

- uninitialized_fill_n
- fill
- uninitialized_copy
- device_new
- device_malloc

thrust::uninitialized_fill

```
template <typename ForwardIterator,
  typename T>
void
uninitialized_fill(ForwardIterator first,
  ForwardIterator last,
  const T & x);
```

在 thrust 中，函数 thrust::device_new 为对象分配内存，然后通过调用构造函数在该位置创建对象。然而，偶尔将这两个操作分开非常有用。如果范围 [first, last) 内的每个迭代器都指向未初始化的内存，则 uninitialized_fill 在该范围内创建 x 的副本。即，对于 [first, last) 范围内的每个迭代器 i，uninitialized_fill 通过调用 ForwardIterator 的 value_type 的拷贝构造函数在指向 i 的位置创建 x 的副本。

以下代码片段演示了如何使用 uninitialized_fill 初始化未初始化的内存范围：

```
#include <thrust/uninitialized_fill.h>
#include <thrust/device_malloc.h>

struct Int
{
__host__ __device__
Int(int x) : val(x) {}
int val;
};
...
const int N = 137;

Int val(46);
thrust::device_ptr<Int> array = thrust::device_malloc<Int>(N);
thrust::uninitialized_fill(array, array + N, val);

// Int x = array[i];
// x.val == 46 for all 0 <= i < N
```

模板参数

- **ForwardIterator**：是 Forward Iterator 的模型，ForwardIterator 是可变的，并且 ForwardIterator 的 value_type 有一个采用类型 T 单参数的构造函数
- **T**：是 Assignable 的模型，T 的 value_type 可转换为 OutputIterator 的 value_type 集合中的类型

函数参数

- **first**：感兴趣范围内的第一个元素
- **last**：感兴趣范围内的最后一个元素
- **x**：用于拷贝构造函数示例的值

另请参阅

- https://en.cppreference.com/w/cpp/memory/uninitialized_fill

飞腾信息 Metax Confidential 2024-11-18 15:00:00

- uninitialized_fill_n
- fill
- uninitialized_copy
- device_new
- device_malloc

thrust::uninitialized_fill_n

```
template <typename DerivedPolicy,
  typename ForwardIterator,
  typename Size,
  typename T>
__host__ __device__ ForwardIterator
uninitialized_fill_n(const thrust::detail::execution_policy_base<
↪DerivedPolicy > & exec,
  ForwardIterator first,
  Size n,
  const T & x);
```

在 thrust 中，函数 thrust::device_new 为对象分配内存，然后通过调用构造函数在该位置创建对象。然而，偶尔将这两个操作分开非常有用。如果范围 [first, first+n) 内的每个迭代器都指向未初始化的内存，则 uninitialized_fill 在该范围内创建 x 的副本。即，对于 [first, first+n) 范围内的每个迭代器 i，uninitialized_fill 通过调用 ForwardIterator 的 value_type 的拷贝构造函数在指向 i 的位置创建 x 的副本。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::device 并行化执行策略，使用 uninitialized_fill 初始化未初始化的内存范围：

```
#include <thrust/uninitialized_fill.h>
#include <thrust/device_malloc.h>
#include <thrust/execution_policy.h>

struct Int
{
__host__ __device__
Int(int x) : val(x) {}
int val;
};
...
const int N = 137;

Int val(46);
thrust::device_ptr<Int> array = thrust::device_malloc<Int>(N);
thrust::uninitialized_fill_n(thrust::device, array, N, val);

// Int x = array[i];
// x.val == 46 for all 0 <= i < N
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **ForwardIterator**：是 Forward Iterator 的模型，ForwardIterator 是可变的，并且 ForwardIterator 的 value_type 有一个采用类型 T 单参数的构造函数

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

- **Size**：是一个整数类型
- **T**：是 Assignable 的模型，T 的 value_type 可转换为 OutputIterator 的 value_type 集合中的类型

函数参数

- **exec**：用于并行化的执行策略
- **first**：感兴趣范围内的第一个元素
- **n**：感兴趣范围的大小
- **x**：用于拷贝构造函数示例的值

返回值

first+n

另请参阅

- https://en.cppreference.com/w/cpp/memory/uninitialized_fill
- uninitialized_fill
- fill
- uninitialized_copy_n
- device_new
- device_malloc

thrust::uninitialized_fill_n

```
template <typename ForwardIterator,
  typename Size,
  typename T>
ForwardIterator
uninitialized_fill_n(ForwardIterator first,
  Size n,
  const T & x);
```

在 thrust 中，函数 thrust::device_new 为对象分配内存，然后通过调用构造函数在该位置创建对象。然而，偶尔将这两个操作分开非常有用。如果范围 [first, first+n) 内的每个迭代器都指向未初始化的内存，则 uninitialized_fill 在该范围内创建 x 的副本。即，对于 [first, first+n) 范围内的每个迭代器 i，uninitialized_fill 通过调用 ForwardIterator 的 value_type 的拷贝构造函数在指向 i 的位置创建 x 的副本。

以下代码片段演示了如何使用 uninitialized_fill 初始化未初始化的内存范围：

```
#include <thrust/uninitialized_fill.h>
#include <thrust/device_malloc.h>

struct Int
{
__host__ __device__
Int(int x) : val(x) {}
int val;
};
...
const int N = 137;

Int val(46);
thrust::device_ptr<Int> array = thrust::device_malloc<Int>(N);
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
thrust::uninitialized_fill_n(array, N, val);

// Int x = array[i];
// x.val == 46 for all 0 <= i < N
```

模板参数

- **ForwardIterator**：是 Forward Iterator 的模型，ForwardIterator 是可变的，并且 ForwardIterator 的 value_type 有一个采用类型 T 单参数的构造函数
- **Size**：是一个整数类型
- **T**：是 Assignable 的模型，T 的 value_type 可转换为 OutputIterator 的 value_type 集合中的类型

函数参数

- **first**：感兴趣范围内的第一个元素
- **n**：感兴趣范围的大小
- **x**：用于拷贝构造函数示例的值

返回值

first+n

另请参阅

- https://en.cppreference.com/w/cpp/memory/uninitialized_fill
- uninitialized_fill
- fill
- uninitialized_copy_n
- device_new
- device_malloc

2.1.9.3 修改（Modifying）

函数

```
template <typename DerivedPolicy,
  typename InputIterator,
  typename UnaryFunction>
__host__ __device__ InputIterator
thrust::for_each(const thrust::detail::execution_policy_base<
 ↪DerivedPolicy > & exec,
  InputIterator first,
  InputIterator last,
  UnaryFunction f);

template <typename DerivedPolicy,
  typename InputIterator,
  typename Size,
  typename UnaryFunction>
__host__ __device__ InputIterator
thrust::for_each_n(const thrust::detail::execution_policy_base<
 ↪DerivedPolicy > & exec,
  InputIterator first,
  Size n,
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
  UnaryFunction f);

template <typename InputIterator,
  typename UnaryFunction>
InputIterator
thrust::for_each(InputIterator first,
  InputIterator last,
  UnaryFunction f);

template <typename InputIterator,
  typename Size,
  typename UnaryFunction>
InputIterator
thrust::for_each_n(InputIterator first,
  Size n,
  UnaryFunction f);
```

thrust::for_each

```
template <typename DerivedPolicy,
  typename InputIterator,
  typename UnaryFunction>
__host__ __device__ InputIterator
for_each(const thrust::detail::execution_policy_base< DerivedPolicy > &
↪exec,
  InputIterator first,
  InputIterator last,
  UnaryFunction f);
```

for_each 将函数对象 f 应用于范围 [first, last) 内的每个元素，忽略 f 的返回值（如果有）。与 C++ 标准模板库函数 std::for_each 不同，此函数不保证执行顺序。因此，此版本的 for_each 不会返回函数对象的副本。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::device 执行策略，使用 for_each 打印 thrust::device_vector 的元素：

```
#include <thrust/for_each.h>
#include <thrust/device_vector.h>
#include <thrust/execution_policy.h>
#include <cstdio>
...

struct printf_functor
{
__host__ __device__
void operator()(int x)
{
   printf("%d\n", x);
}
};
...
thrust::device_vector<int> d_vec(3);
d_vec[0] = 0; d_vec[1] = 1; d_vec[2] = 2;
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-15 15:00:00

(续上页)

```
thrust::for_each(thrust::device, d_vec.begin(), d_vec.end(), printf_
↪functor());

// 0 1 2 is printed to standard output in some unspecified order
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **InputIterator**：是 Input Iterator 的模型，InputIterator 的 value_type 可转换为 UnaryFunction 的 argument_type
- **UnaryFunction**：是 Unary Function 的模型，且 UnaryFunction 不通过其参数应用任何非常量运算

函数参数

- **exec**：用于并行化的执行策略
- **first**：序列起始
- **last**：序列末尾
- **f**：应用于范围 [first, last) 的函数对象

返回值

last

另请参阅

- for_each_n
- https://en.cppreference.com/w/cpp/algorithm/for_each

thrust::for_each_n

```
template <typename DerivedPolicy,
  typename InputIterator,
  typename Size,
  typename UnaryFunction>
__host__ __device__ InputIterator
for_each_n(const thrust::detail::execution_policy_base< DerivedPolicy > &
↪exec,
  InputIterator first,
  Size n,
  UnaryFunction f);
```

for_each_n 将函数对象 f 应用于范围 [first, first + n) 内的每个元素，忽略 f 的返回值（如果有）。与 C++ 标准模板库函数 std::for_each 不同，此函数不保证执行顺序。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::device 执行策略，使用 for_each_n 打印 device_vector 的元素：

```
#include <thrust/for_each.h>
#include <thrust/device_vector.h>
#include <thrust/execution_policy.h>
#include <cstdio>

struct printf_functor
{
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
__host__ __device__
void operator()(int x)
{
   printf("%d\n", x);
}
};
...
thrust::device_vector<int> d_vec(3);
d_vec[0] = 0; d_vec[1] = 1; d_vec[2] = 2;

thrust::for_each_n(thrust::device, d_vec.begin(), d_vec.size(), printf_
↪functor());

// 0 1 2 is printed to standard output in some unspecified order
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **InputIterator**：是 Input Iterator 的模型，InputIterator 的 value_type 可转换为 UnaryFunction 的 argument_type
- **Size**：是一个整数类型
- **UnaryFunction**：是 Unary Function 的模型，且 UnaryFunction 不通过其参数应用任何非常量运算

函数参数

- **exec**：用于并行化的执行策略
- **first**：序列起始
- **n**：输入序列的大小
- **f**：应用于范围 [first, first + n) 的函数对象

返回值

first + n

另请参阅

- for_each
- https://en.cppreference.com/w/cpp/algorithm/for_each

thrust::for_each

```
template <typename InputIterator,
  typename UnaryFunction>
InputIterator
for_each(InputIterator first,
  InputIterator last,
  UnaryFunction f);
```

for_each 将函数对象 f 应用于范围 [first, last) 内的每个元素，忽略 f 的返回值（如果有）。与 C++ 标准模板库函数 std::for_each 不同，此函数不保证执行顺序。因此，此版本的 for_each 不会返回函数对象的副本。

以下代码片段演示了如何使用 for_each 打印 device_vector 的元素：

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

```
#include <thrust/for_each.h>
#include <thrust/device_vector.h>
#include <stdio.h>

struct printf_functor
{
__host__ __device__
void operator()(int x)
{
   printf("%d\n", x);
}
};
...
thrust::device_vector<int> d_vec(3);
d_vec[0] = 0; d_vec[1] = 1; d_vec[2] = 2;

thrust::for_each(d_vec.begin(), d_vec.end(), printf_functor());

// 0 1 2 is printed to standard output in some unspecified order
```

模板参数

- **InputIterator**：是 Input Iterator 的模型，InputIterator 的 value_type 可转换为 UnaryFunction 的 argument_type
- **UnaryFunction**：是 Unary Function 的模型，且 UnaryFunction 不通过其参数应用任何非常量运算

函数参数

- **first**：序列起始
- **last**：序列末尾
- **f**：应用于范围 [first, last) 的函数对象

返回值

last

另请参阅

- for_each_n
- https://en.cppreference.com/w/cpp/algorithm/for_each

thrust::for_each_n

```
template <typename InputIterator,
  typename Size,
  typename UnaryFunction>
InputIterator
for_each_n(InputIterator first,
  Size n,
  UnaryFunction f);
```

for_each_n 将函数对象 f 应用于范围 [first, first + n) 内的每个元素，忽略 f 的返回值（如果有）。与 C++ 标准模板库函数 std::for_each 不同，此函数不保证执行顺序。

以下代码片段演示了如何使用 for_each_n 打印 device_vector 的元素：

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

```
#include <thrust/for_each.h>
#include <thrust/device_vector.h>
#include <stdio.h>

struct printf_functor
{
__host__ __device__
void operator()(int x)
{
   printf("%d\n", x);
}
};
...
thrust::device_vector<int> d_vec(3);
d_vec[0] = 0; d_vec[1] = 1; d_vec[2] = 2;

thrust::for_each_n(d_vec.begin(), d_vec.size(), printf_functor());

// 0 1 2 is printed to standard output in some unspecified order
```

模板参数

- **InputIterator**：是 Input Iterator 的模型，InputIterator 的 value_type 可转换为 UnaryFunction 的 argument_type
- **Size**：是一个整数类型
- **UnaryFunction**：是 Unary Function 的模型，且 UnaryFunction 不通过其参数应用任何非常量运算

函数参数

- **first**：序列起始
- **n**：输入序列的大小
- **f**：应用于范围 [first, first + n) 的函数对象

返回值

first + n

另请参阅

- for_each
- https://en.cppreference.com/w/cpp/algorithm/for_each

2.1.9.4 替换（Replacing）

函数

```
template <typename DerivedPolicy,
  typename ForwardIterator,
  typename T>
__host__ __device__ void
thrust::replace(const thrust::detail::execution_policy_base< DerivedPolicy
↪> & exec,
  ForwardIterator first,
  ForwardIterator last,
  const T & old_value,
  const T & new_value);
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
template <typename ForwardIterator,
  typename T>
void
thrust::replace(ForwardIterator first,
  ForwardIterator last,
  const T & old_value,
  const T & new_value);

template <typename DerivedPolicy,
  typename ForwardIterator,
  typename Predicate,
  typename T>
__host__ __device__ void
thrust::replace_if(const thrust::detail::execution_policy_base<
 ↪DerivedPolicy > & exec,
  ForwardIterator first,
  ForwardIterator last,
  Predicate pred,
  const T & new_value);

template <typename ForwardIterator,
  typename Predicate,
  typename T>
void
thrust::replace_if(ForwardIterator first,
  ForwardIterator last,
  Predicate pred,
  const T & new_value);

template <typename DerivedPolicy,
  typename ForwardIterator,
  typename InputIterator,
  typename Predicate,
  typename T>
__host__ __device__ void
thrust::replace_if(const thrust::detail::execution_policy_base<
 ↪DerivedPolicy > & exec,
  ForwardIterator first,
  ForwardIterator last,
  InputIterator stencil,
  Predicate pred,
  const T & new_value);

template <typename ForwardIterator,
  typename InputIterator,
  typename Predicate,
  typename T>
void
thrust::replace_if(ForwardIterator first,
  ForwardIterator last,
  InputIterator stencil,
  Predicate pred,
  const T & new_value);

template <typename DerivedPolicy,
  typename InputIterator,
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
  typename OutputIterator,
  typename T>
__host__ __device__ OutputIterator
thrust::replace_copy(const thrust::detail::execution_policy_base<
↪DerivedPolicy > & exec,
  InputIterator first,
  InputIterator last,
  OutputIterator result,
  const T & old_value,
  const T & new_value);

template <typename InputIterator,
  typename OutputIterator,
  typename T>
OutputIterator
thrust::replace_copy(InputIterator first,
  InputIterator last,
  OutputIterator result,
  const T & old_value,
  const T & new_value);

template <typename DerivedPolicy,
  typename InputIterator,
  typename OutputIterator,
  typename Predicate,
  typename T>
__host__ __device__ OutputIterator
thrust::replace_copy_if(const thrust::detail::execution_policy_base<
↪DerivedPolicy > & exec,
  InputIterator first,
  InputIterator last,
  OutputIterator result,
  Predicate pred,
  const T & new_value);

template <typename InputIterator,
  typename OutputIterator,
  typename Predicate,
  typename T>
OutputIterator
thrust::replace_copy_if(InputIterator first,
  InputIterator last,
  OutputIterator result,
  Predicate pred,
  const T & new_value);

template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator,
  typename Predicate,
  typename T>
__host__ __device__ OutputIterator
thrust::replace_copy_if(const thrust::detail::execution_policy_base<
↪DerivedPolicy > & exec,
  InputIterator1 first,
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
  InputIterator1 last,
  InputIterator2 stencil,
  OutputIterator result,
  lPredicate pred,
  const T & new_value);

template <typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator,
  typename Predicate,
  typename T>
OutputIterator
thrust::replace_copy_if(InputIterator1 first,
  InputIterator1 last,
  InputIterator2 stencil,
  OutputIterator result,
  Predicate pred,
  const T & new_value);
```

thrust::replace

```
template <typename DerivedPolicy,
  typename ForwardIterator,
  typename T>
__host__ __device__ void
replace(const thrust::detail::execution_policy_base< DerivedPolicy > &
↪exec,
  ForwardIterator first,
  ForwardIterator last,
  const T & old_value,
  const T & new_value);
```

replace 将范围 [first, last) 内等于 old_value 的每个元素替换为 new_value。即：对于每个迭代器 i，如果 *i == old_value，那么它执行赋值 *i = new_value。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::device 并行化执行策略，使用 replace 将 device_vector 内感兴趣的值替换为另一个值：

```
#include <thrust/replace.h>
#include <thrust/device_vector.h>
#include <thrust/execution_policy.h>

...

thrust::device_vector<int> A(4);
A[0] = 1;
A[1] = 2;
A[2] = 3;
A[3] = 1;

thrust::replace(thrust::device, A.begin(), A.end(), 1, 99);

// A contains [99, 2, 3, 99]
```

飞谱信息 MetaX Confidential 2024-11-18 15:00:00

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **ForwardIterator**：是 Forward Iterator 的模型，且 ForwardIterator 是可变的
- **T**：是 Assignable 的模型，T 是 EqualityComparable 的模型，可以将 T 的对象与 ForwardIterator 的 value_type 的对象进行相等性比较，且 T 可转换为 ForwardIterator 的 value_type

函数参数

- **exec**：用于并行化的执行策略
- **first**：感兴趣序列的起始
- **last**：感兴趣序列的末尾
- **old_value**：待替换的值
- **new_value**：待替换 old_value 的新值

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/replace
- replace_if
- replace_copy
- replace_copy_if

thrust::replace

```
template <typename ForwardIterator,
  typename T>
void
replace(ForwardIterator first,
  ForwardIterator last,
  const T & old_value,
  const T & new_value);
```

replace 将范围 [first, last) 内等于 old_value 的每个元素替换为 new_value。即：对于每个迭代器 i，如果 *i == old_value，那么它执行赋值 *i = new_value。

以下代码片段演示了如何使用 replace 将 device_vector 内感兴趣的值替换为另一个值：

```
#include <thrust/replace.h>
#include <thrust/device_vector.h>

...

thrust::device_vector<int> A(4);
A[0] = 1;
A[1] = 2;
A[2] = 3;
A[3] = 1;

thrust::replace(A.begin(), A.end(), 1, 99);

// A contains [99, 2, 3, 99]
```

模板参数

- **ForwardIterator**：是 Forward Iterator 的模型，且 ForwardIterator 是可变的

飞谱信息 MetaX Confidential 2024-11-18 15:00:00

- **T**：是 Assignable 的模型，T 是 EqualityComparable 的模型，可以将 T 的对象与 ForwardIterator 的 value_type 的对象进行相等性比较，且 T 可转换为 ForwardIterator 的 value_type

函数参数

- **first**：感兴趣序列的起始
- **last**：感兴趣序列的末尾
- **old_value**：待替换的值
- **new_value**：待替换 old_value 的新值

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/replace
- replace_if
- replace_copy
- replace_copy_if

thrust::replace_if

```
template <typename DerivedPolicy,
  typename ForwardIterator,
  typename Predicate,
  typename T>
__host__ __device__ void
replace_if(const thrust::detail::execution_policy_base< DerivedPolicy > &
↪exec,
  ForwardIterator first,
  ForwardIterator last,
  Predicate pred,
  const T & new_value);
```

replace_if 将范围 [first, last) 内使得 pred 返回 true 的每个元素替换为 new_value。即：对于每个迭代器 i，如果 pred(*i) 为 true，那么它执行赋值 *i = new_value。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::device 并行化执行策略，使用 replace_if 将 device_vector 的负元素替换为 0：

```
#include <thrust/replace.h>
#include <thrust/device_vector.h>
#include <thrust/execution_policy.h>
...
struct is_less_than_zero
{
__host__ __device__
bool operator()(int x)
{
   return x < 0;
}
};

...

thrust::device_vector<int> A(4);
A[0] =  1;
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
A[1] = -3;
A[2] =  2;
A[3] = -1;

is_less_than_zero pred;

thrust::replace_if(thrust::device, A.begin(), A.end(), pred, 0);

// A contains [1, 0, 2, 0]
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **ForwardIterator**：是 Forward Iterator 的模型，ForwardIterator 是可变的，并且 ForwardIterator 的 value_type 可转换为 Predicate 的 argument_type
- **Predicate**：是 Predicate 的模型
- **T**：是 Assignable 的模型，且 T 可转换为 ForwardIterator 的 value_type

函数参数

- **exec**：用于并行化的执行策略
- **first**：感兴趣序列的起始
- **last**：感兴趣序列的末尾
- **pred**：对范围 [first,last) 内每个值进行测试的谓词
- **new_value**：用于替换使得 pred(*i) 为 true 的元素的新值

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/replace
- replace
- replace_copy
- replace_copy_if

thrust::replace_if

```
template <typename ForwardIterator,
  typename Predicate,
  typename T>
void
replace_if(ForwardIterator first,
  ForwardIterator last,
  Predicate pred,
  const T & new_value);
```

replace_if 将范围 [first, last) 内使得 pred 返回 true 的每个元素替换为 new_value。即：对于每个迭代器 i ，如果 pred(*i) 为 true ，那么它执行赋值 *i = new_value。

以下代码片段演示了如何使用 replace_if 将 device_vector 的负元素替换为 0：

```
#include <thrust/replace.h>
#include <thrust/device_vector.h>
...
struct is_less_than_zero
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
{
__host__ __device__
bool operator()(int x)
{
    return x < 0;
}
};

...

thrust::device_vector<int> A(4);
A[0] =  1;
A[1] = -3;
A[2] =  2;
A[3] = -1;

is_less_than_zero pred;

thrust::replace_if(A.begin(), A.end(), pred, 0);

// A contains [1, 0, 2, 0]
```

模板参数

- **ForwardIterator**：是 Forward Iterator 的模型，ForwardIterator 是可变的，并且 ForwardIterator 的 value_type 可转换为 Predicate 的 argument_type
- **Predicate**：是 Predicate 的模型
- **T**：是 Assignable 的模型，且 T 可转换为 ForwardIterator 的 value_type

函数参数

- **first**：感兴趣序列的起始
- **last**：感兴趣序列的末尾
- **pred**：对范围 [first,last) 内每个值进行测试的谓词
- **new_value**：用于替换使得 pred(*i) 为 true 的元素的新值

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/replace
- replace
- replace_copy
- replace_copy_if

thrust::replace_if

```
template <typename DerivedPolicy,
  typename ForwardIterator,
  typename InputIterator,
  typename Predicate,
  typename T>
__host__ __device__ void
replace_if(const thrust::detail::execution_policy_base< DerivedPolicy > &
↪exec,
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
  ForwardIterator first,
  ForwardIterator last,
  InputIterator stencil,
  Predicate pred,
  const T & new_value);
```

replace_if 将范围 [first, last) 内使得 pred(*s) 返回 true 的每个元素替换为 new_value。即：对于范围 [first, last) 内的每个迭代器 i 和范围 [stencil, stencil + (last - first)) 内的 s ，如果 pred(*s) 为 true ，则它执行赋值 *i = new_value。

算法的执行由 exec 确定并行化。

以下代码片段演示了如何通过 thrust::device 并行化执行策略，当对应的模板元素小于零时，使用 replace_if 将 device_vector 的元素替换为 0：

```
#include <thrust/replace.h>
#include <thrust/device_vector.h>
#include <thrust/execution_policy.h>

struct is_less_than_zero
{
__host__ __device__
bool operator()(int x)
{
   return x < 0;
}
};

...

thrust::device_vector<int> A(4);
A[0] =  10;
A[1] =  20;
A[2] =  30;
A[3] =  40;

thrust::device_vector<int> S(4);
S[0] = -1;
S[1] =  0;
S[2] = -1;
S[3] =  0;

is_less_than_zero pred;
thrust::replace_if(thrust::device, A.begin(), A.end(), S.begin(), pred, 0);

// A contains [0, 20, 0, 40]
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **ForwardIterator**：是 Forward Iterator 的模型，且 ForwardIterator 是可变的
- **InputIterator**：是 Input Iterator 的模型，InputIterator 的 value_type 可转换为 Predicate 的 argument_type
- **Predicate**：是 Predicate 的模型
- **T**：是 Assignable 的模型，且 T 可转换为 ForwardIterator 的 value_type

函数参数

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

- **exec**：用于并行化的执行策略
- **first**：感兴趣序列的起始
- **last**：感兴趣序列的末尾
- **stencil**：模版序列的起始
- **pred**：对范围 [first,last) 内每个值进行测试的谓词
- **new_value**：用于替换使得 pred(*i) 为 true 的元素的新值

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/replace
- replace
- replace_copy
- replace_copy_if

thrust::replace_if

```
template <typename ForwardIterator,
  typename InputIterator,
  typename Predicate,
  typename T>
void
replace_if(ForwardIterator first,
  ForwardIterator last,
  InputIterator stencil,
  Predicate pred,
  const T & new_value);
```

replace_if 将范围 [first, last) 内使得 pred(*s) 返回 true 的每个元素替换为 new_value。即：对于范围 [first, last) 内的每个迭代器 i 和范围 [stencil, stencil + (last - first)) 内的 s ，如果 pred(*s) 为 true ，则它执行赋值 *i = new_value。

以下代码片段演示了当对应的模板元素小于零时，如何使用 replace_if 将 device_vector 的元素替换为 0：

```
#include <thrust/replace.h>
#include <thrust/device_vector.h>

struct is_less_than_zero
{
__host__ __device__
bool operator()(int x)
{
   return x < 0;
}
};

...

thrust::device_vector<int> A(4);
A[0] =  10;
A[1] =  20;
A[2] =  30;
A[3] =  40;
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
thrust::device_vector<int> S(4);
S[0] = -1;
S[1] =  0;
S[2] = -1;
S[3] =  0;

is_less_than_zero pred;
thrust::replace_if(A.begin(), A.end(), S.begin(), pred, 0);

// A contains [0, 20, 0, 40]
```

模板参数

- **ForwardIterator**：是 Forward Iterator 的模型，且 ForwardIterator 是可变的
- **InputIterator**：是 Input Iterator 的模型，InputIterator 的 value_type 可转换为 Predicate 的 argument_type
- **Predicate**：是 Predicate 的模型
- **T**：是 Assignable 的模型，且 T 可转换为 ForwardIterator 的 value_type

函数参数

- **first**：感兴趣序列的起始
- **last**：感兴趣序列的末尾
- **stencil**：模版序列的起始
- **pred**：对范围 [first,last) 内每个值进行测试的谓词
- **new_value**：用于替换使得 pred(*i) 为 true 的元素的新值

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/replace
- replace
- replace_copy
- replace_copy_if

thrust::replace_copy

```
template <typename DerivedPolicy,
  typename InputIterator,
  typename OutputIterator,
  typename T>
__host__ __device__ OutputIterator
replace_copy(const thrust::detail::execution_policy_base< DerivedPolicy > &
↪ exec,
  InputIterator first,
  InputIterator last,
  OutputIterator result,
  const T & old_value,
  const T & new_value);
```

replace_copy 将 new_value 而不是等于 old_value 的元素从范围 [first, last) 复制到范围 [result, result + (last-first))。

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

具体而言，对于每个满足 0 <= n < last-first 的整数 n ，如果 *(first+n) == old_value ，replace_copy 执行赋值 *(result+n) = new_value ；否则，执行赋值 *(result+n) = *(first+n)。

算法的执行由 exec 确定并行化。

```
#include <thrust/replace.h>
#include <thrust/device_vector.h>
#include <thrust/execution_policy.h>
...
thrust::device_vector<int> A(4);
A[0] = 1;
A[1] = 2;
A[2] = 3;
A[3] = 1;

thrust::device_vector<int> B(4);

thrust::replace_copy(thrust::device, A.begin(), A.end(), B.begin(), 1, 99);

// B contains [99, 2, 3, 99]
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **InputIterator**：是 Input Iterator 的模型
- **OutputIterator**：是 Output Iterator 的模型
- **T**：是 Assignable 的模型，T 是 Equality Comparable 的模型，可以将 T 与 InputIterator 的 value_type 进行相等性比较，且 T 可转换为 OutputIterator 的 value_type

函数参数

- **exec**：用于并行化的执行策略
- **first**：待从中复制的序列的起始
- **last**：待从中复制的序列的末尾
- **result**：待复制到的序列的起始
- **old_value**：待替换的值
- **new_value**：使得 *i == old_value 为 true 的替换值

前提条件

first 可以等于 result ，但除此外，范围 [first, last) 不应与范围 [result, result + (last - first)) 重叠

返回值

result + (last-first)

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/replace_copy
- copy
- replace
- replace_if
- replace_copy_if

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

thrust::replace_copy

```
template <typename InputIterator,
  typename OutputIterator,
  typename T>
OutputIterator
replace_copy(InputIterator first,
  InputIterator last,
  OutputIterator result,
  const T & old_value,
  const T & new_value);
```

replace_copy 将 new_value 而不是等于 old_value 的元素从范围 [first, last) 复制到范围 [result, result + (last-first))。

具体而言，对于每个满足 0 <= n < last-first 的整数 n ，如果 *(first+n) == old_value ，replace_copy 执行赋值 *(result+n) = new_value ；否则，执行赋值 *(result+n) = *(first+n)。

```
#include <thrust/replace.h>
#include <thrust/device_vector.h>
...
thrust::device_vector<int> A(4);
A[0] = 1;
A[1] = 2;
A[2] = 3;
A[3] = 1;

thrust::device_vector<int> B(4);

thrust::replace_copy(A.begin(), A.end(), B.begin(), 1, 99);

// B contains [99, 2, 3, 99]
```

模板参数

- **InputIterator**：是 Input Iterator 的模型
- **OutputIterator**：是 Output Iterator 的模型
- **T**：是 Assignable 的模型，T 是 Equality Comparable 的模型，可以将 T 与 InputIterator 的 value_type 进行相等性比较，且 T 可转换为 OutputIterator 的 value_type

函数参数

- **first**：待从中复制的序列的起始
- **last**：待从中复制的序列的末尾
- **result**：待复制到的序列的起始
- **old_value**：待替换的值
- **new_value**：使得 *i == old_value 为 true 的替换值

前提条件

first 可以等于 result ，但除此外，范围 [first, last) 不应与范围 [result, result + (last - first)) 重叠

返回值

result + (last-first)

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/replace_copy

飞曦信息 MetaX Confidential 2024-11-18 15:00:00

- copy
- replace
- replace_if
- replace_copy_if

thrust::replace_copy_if

```
template <typename DerivedPolicy,
  typename InputIterator,
  typename OutputIterator,
  typename Predicate,
  typename T>
__host__ __device__ OutputIterator
replace_copy_if(const thrust::detail::execution_policy_base< DerivedPolicy
↪> & exec,
  InputIterator first,
  InputIterator last,
  OutputIterator result,
  Predicate pred,
  const T & new_value);
```

replace_copy_if 将 new_value 而不是使得 pred 为 true 的元素从范围 [first, last) 复制到范围 [result, result + (last-first))。

具体而言，对于每个满足 0 <= n < last-first 的整数 n ，如果 pred(*(first+n)) ，replace_copy_if 执行赋值 *(result+n) = new_value ； 否则，执行赋值 *(result+n) = *(first+n)。

算法的执行由 exec 确定并行化。

```
#include <thrust/replace.h>
#include <thrust/device_vector.h>
#include <thrust/execution_policy.h>

struct is_less_than_zero
{
__host__ __device__
bool operator()(int x)
{
   return x < 0;
}
};

...

thrust::device_vector<int> A(4);
A[0] =  1;
A[1] = -3;
A[2] =  2;
A[3] = -1;

thrust::device_vector<int> B(4);
is_less_than_zero pred;

thrust::replace_copy_if(thrust::device, A.begin(), A.end(), B.begin(),
↪pred, 0);
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
// B contains [1, 0, 2, 0]
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **InputIterator**：是 Input Iterator 的模型，InputIterator 的 value_type 可转换为 Predicate 的 argument_type
- **OutputIterator**：是 Output Iterator 的模型
- **Predicate**：是 Predicate 的模型
- **T**：是 Assignable 的模型，且 T 可转换为 OutputIterator 的 value_type

函数参数

- **exec**：用于并行化的执行策略
- **first**：待从中复制的序列的起始
- **last**：待从中复制的序列的末尾
- **result**：待复制到的序列的起始
- **pred**：对范围 [first,last) 内每个值进行测试的谓词
- **new_value**：使得 pred(*i) 为 true 的替换值

前提条件

first 可以等于 result ，但除此外，范围 [first, last) 不应与范围 [result, result + (last - first)) 重叠

返回值

result + (last-first)

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/replace_copy
- replace
- replace_if
- replace_copy

thrust::replace_copy_if

```
template <typename InputIterator,
  typename OutputIterator,
  typename Predicate,
  typename T>
OutputIterator
replace_copy_if(InputIterator first,
  InputIterator last,
  OutputIterator result,
  Predicate pred,
  const T & new_value);
```

replace_copy_if 将 new_value 而不是使得 pred 为 true 的元素从范围 [first, last) 复制到范围 [result, result + (last-first))。

具体而言，对于每个满足 0 <= n < last-first 的整数 n ，如果 pred(*(first+n)) ，replace_copy_if 执行赋值 *(result+n) = new_value ；否则，执行赋值 *(result+n) = *(first+n)。

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

```
#include <thrust/replace.h>
#include <thrust/device_vector.h>

struct is_less_than_zero
{
__host__ __device__
bool operator()(int x)
{
   return x < 0;
}
};

...

thrust::device_vector<int> A(4);
A[0] =  1;
A[1] = -3;
A[2] =  2;
A[3] = -1;

thrust::device_vector<int> B(4);
is_less_than_zero pred;

thrust::replace_copy_if(A.begin(), A.end(), B.begin(), pred, 0);

// B contains [1, 0, 2, 0]
```

模板参数

- **InputIterator**：是 Input Iterator 的模型，InputIterator 的 value_type 可转换为 Predicate 的 argument_type
- **OutputIterator**：是 Output Iterator 的模型
- **Predicate**：是 Predicate 的模型
- **T**：是 Assignable 的模型，且 T 可转换为 OutputIterator 的 value_type

函数参数

- **first**：待从中复制的序列的起始
- **last**：待从中复制的序列的末尾
- **result**：待复制到的序列的起始
- **pred**：对范围 [first,last) 内每个值进行测试的谓词
- **new_value**：使得 pred(*i) 为 true 的替换值

前提条件

first 可以等于 result，但除此外，范围 [first, last) 不应与范围 [result, result + (last - first)) 重叠

返回值

result + (last-first)

另请参阅

- https://en.cppreference.com/w/cpp/algorithm/replace_copy
- replace
- replace_if
- replace_copy

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

thrust::replace_copy_if

```
template <typename DerivedPolicy,
  typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator,
  typename Predicate,
  typename T>
__host__ __device__ OutputIterator
replace_copy_if(const thrust::detail::execution_policy_base< DerivedPolicy
↪> & exec,
  InputIterator1 first,
  InputIterator1 last,
  InputIterator2 stencil,
  OutputIterator result,
  Predicate pred,
  const T & new_value);
```

此版本的 replace_copy_if 将范围 [first, last) 内除了其相应模版元素使得 pred 为 true 之外的元素 new_value 复制到范围 [result, result + (last-first))。

具体而言，对于每个满足 0 <= n < last-first 的整数 n ，如果 pred(*(stencil+n)) ，replace_copy_if 执行赋值 *(result+n) = new_value ；否则，执行赋值 *(result+n) = *(first+n)。

算法的执行由 exec 确定并行化。

```
#include <thrust/replace.h>
#include <thrust/device_vector.h>
#include <thrust/execution_policy.h>

struct is_less_than_zero
{
__host__ __device__
bool operator()(int x)
{
   return x < 0;
}
};

...

thrust::device_vector<int> A(4);
A[0] =  10;
A[1] =  20;
A[2] =  30;
A[3] =  40;

thrust::device_vector<int> S(4);
S[0] = -1;
S[1] =  0;
S[2] = -1;
S[3] =  0;

thrust::device_vector<int> B(4);
is_less_than_zero pred;

thrust::replace_if(thrust::device, A.begin(), A.end(), S.begin(), B.
↪begin(), pred, 0);
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
// B contains [0, 20, 0, 40]
```

模板参数

- **DerivedPolicy**：派生执行策略的名称
- **InputIterator1**：是 Input Iterator 的模型
- **InputIterator2**：是 Input Iterator 的模型，InputIterator2 的 value_type 可转换为 Predicate 的 argument_type
- **OutputIterator**：是 Output Iterator 的模型
- **Predicate**：是 Predicate 的模型
- **T**：是 Assignable 的模型，且 T 可转换为 OutputIterator 的 value_type

函数参数

- **exec**：用于并行化的执行策略
- **first**：待从中复制的序列的起始
- **last**：待从中复制的序列的末尾
- **stencil**：模版序列的起始
- **result**：待复制到的序列的起始
- **pred**：对范围 [stencil, stencil + (last - first)) 内每个值进行测试的谓词
- **new_value**：使得 pred(*s) 为 true 的替换值

前提条件

- first 可以等于 result，但除此外，范围 [first, last) 不应与范围 [result, result + (last - first)) 重叠
- stencil 可以等于 result，但除此外，范围 [stencil, stencil + (last - first)) 不应与范围 [result, result + (last - first)) 重叠

返回值

result + (last-first)

另请参阅

- replace_copy
- replace_if

thrust::replace_copy_if

```
template <typename InputIterator1,
  typename InputIterator2,
  typename OutputIterator,
  typename Predicate,
  typename T>
OutputIterator
replace_copy_if(InputIterator1 first,
  InputIterator1 last,
  InputIterator2 stencil,
  OutputIterator result,
  Predicate pred,
  const T & new_value);
```

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

此版本的 replace_copy_if 将范围 [first, last) 内除了其相应模版元素使得 pred 为 true 之外的元素 new_value 复制到范围 [result, result + (last-first))。

具体而言，对于每个满足 0 <= n < last-first 的整数 n ，如果 pred(*(stencil+n)) ，replace_copy_if 执行赋值 *(result+n) = new_value ；否则，执行赋值 *(result+n) = *(first+n)。

```
#include <thrust/replace.h>
#include <thrust/device_vector.h>

struct is_less_than_zero
{
__host__ __device__
bool operator()(int x)
{
   return x < 0;
}
};

...

thrust::device_vector<int> A(4);
A[0] =  10;
A[1] =  20;
A[2] =  30;
A[3] =  40;

thrust::device_vector<int> S(4);
S[0] = -1;
S[1] =  0;
S[2] = -1;
S[3] =  0;

thrust::device_vector<int> B(4);
is_less_than_zero pred;

thrust::replace_if(A.begin(), A.end(), S.begin(), B.begin(), pred, 0);

// B contains [0, 20, 0, 40]
```

模板参数

- **InputIterator1**：是 Input Iterator 的模型
- **InputIterator2**：是 Input Iterator 的模型，InputIterator2 的 value_type 可转换为 Predicate 的 argument_type
- **OutputIterator**：是 Output Iterator 的模型
- **Predicate**：是 Predicate 的模型
- **T**：是 Assignable 的模型，且 T 可转换为 OutputIterator 的 value_type

函数参数

- **first**：待从中复制的序列的起始
- **last**：待从中复制的序列的末尾
- **stencil**：模版序列的起始
- **result**：待复制到的序列的起始
- **pred**：对范围 [stencil, stencil + (last - first)) 内每个值进行测试的谓词
- **new_value**：使得 pred(*s) 为 true 的替换值

飞曦信息 MetaX Confidential 2024-11-18 15:00:00

前提条件

- first 可以等于 result ，但除此外，范围 [first, last) 不应与范围 [result, result + (last - first)) 重叠
- stencil 可以等于 result ，但除此外，范围 [stencil, stencil + (last - first)) 不应与范围 [result, result + (last - first)) 重叠

返回值

result + (last-first)

另请参阅

- replace_copy
- replace_if

2.2 容器类（Container Class）

2.2.1 主机容器（Host Container）

主机容器

```
template <typename T,
  typename Alloc = std::allocator<T>>
class
thrust::host_vector;

template <typename T,
  typename Alloc>
void
thrust::swap(host_vector< T, Alloc > & a,
  host_vector< T, Alloc > & b);
```

成员类

2.2.1.1 thrust::host_vector

继承自:

```
detail::vector_base< T, std::allocator< T > >
```

函数

2.2.1.2 thrust::swap

```
template <typename T,
  typename Alloc>
void
swap(host_vector< T, Alloc > & a,
  host_vector< T, Alloc > & b);
```

交换两个向量的值。

- x：感兴趣的第一个 host_vector
- y：感兴趣的第二个 host_vector

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

2.2.2 容器（Container）

容器

```
template <typename T,
  typename Alloc = thrust::device_allocator<T>>
class
thrust::device_vector;

template <typename T,
  typename Alloc>
void
thrust::swap(device_vector< T, Alloc > & a,
  device_vector< T, Alloc > & b);
```

成员类

2.2.2.1 thrust::device_vector

继承自:

```
detail::vector_base< T, thrust::device_allocator< T > >
```

函数

2.2.2.2 thrust::swap

```
template <typename T,
  typename Alloc>
void
swap(device_vector< T, Alloc > & a,
  device_vector< T, Alloc > & b);
```

交换两个向量的值。

- x：感兴趣的第一个 device_vector
- y：感兴趣的第二个 device_vector

2.3 函数对象类（Function Objects Class）

2.3.1 函数对象适配器（Function Object Adaptor）

函数对象适配器

```
template <typename Argument,
  typename Result>
struct
thrust::unary_function;

template <typename Argument1,
  typename Argument2,
  typename Result>
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
struct
hrust::binary_function;

template <typename Predicate>
struct
thrust::unary_negate;

template <typename Predicate>
struct
thrust::binary_negate;

template <typename Function>
class
thrust::zip_function;

template <typename Predicate>
__host__ __device__ unary_negate< Predicate >
thrust::not1(const Predicate & pred);

template <typename BinaryPredicate>
__host__ __device__ binary_negate< BinaryPredicate >
thrust::not2(const BinaryPredicate & pred);

template <typename Function>
__host__ __device__ zip_function< typename std::decay< Function >::type >
thrust::make_zip_function(Function && fun);
```

成员类

thrust::unary_function

thrust::binary_function

thrust::unary_negate

继承自: thrust::unary_function< Predicate::argument_type, bool >

thrust::binary_negate

继承自: thrust::binary_function< Predicate::first_argument_type, Predicate::second_argument_type, bool >

thrust::zip_function

函数

2.3.1.1 thrust::not1

```
template <typename Predicate> __host__ __device__ unary_negate< Predicate >
↪ not1(const Predicate & pred);
```

not1 是一个辅助函数，用于简化 Adaptable Predicate 的创建：它将 Adaptable Predicate pred 作为参数，并返回一个新的 Adaptable Predicate 来表示 pred 的相反数。即：如果 pred 是对 Adaptable Predicate 建模类型的对象，则 not1(pred) 的结果 npred 的类型也是 Adaptable Predicate 的模型，并且 npred(x) 始终返回与 !pred(x) 相同的值。

模板参数

Predicate：是 Adaptable Predicate 的模型

函数参数

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

pred： 用于取相反数的 Adaptable Predicate

返回值

一个使得 npred(x) 始终返回与 !pred(x) 相同值的新对象，npred

另请参阅

- unary_negate
- not2

2.3.1.2 thrust::not2

```
template <typename BinaryPredicate> __host__ __device__ binary_negate<
↪BinaryPredicate > not2(const BinaryPredicate & pred);
```

not2 是一个辅助函数，用于简化 Adaptable Binary Predicate 的创建：它将 Adaptable Binary Predicate pred 作为参数，并返回一个新的 Adaptable Binary Predicate 来表示 pred 的相反数。即：如果 pred 是对 Adaptable Binary Predicate 建模类型的对象，则 not2(pred) 的结果 npred 的类型也是 Adaptable Binary Predicate 的模型，并且 npred(x,y) 始终返回与 !pred(x,y) 相同的值。

模板参数

Binary：Predicate 是 Adaptable Binary Predicate 的模型

函数参数

pred： 用于取相反数的 Adaptable Binary Predicate

返回值

一个使得 npred(x,y) 始终返回与 !pred(x,y) 相同值的新对象，npred

另请参阅

- binary_negate
- not1

2.3.1.3 thrust::make_zip_function

```
template <typename Function> __host__ __device__ zip_function< typename
↪std::decay< Function >::type > make_zip_function(Function && fun);
```

make_zip_function 从函数对象创建 zip_function。

函数参数

fun：N 元函数对象

返回值

采用 N 元组的 zip_function

另请参阅

zip_function

2.3.2 占位符对象（Placeholder Object）

占位符对象

/* Facilities for constructing simple functions inline. */namespace thrust::placeholders { …}

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

2.3.3 预定义函数对象（Predefined Function Object）

2.3.3.1 算术运算（Arithmetic Operation）

算术运算

```
template <typename T = void>
struct
thrust::plus;

struct
thrust::plus< void >;

template <typename T = void>
struct
thrust::minus;

struct
thrust::minus< void >;

template <typename T = void>
struct
thrust::multiplies;

struct
thrust::multiplies< void >;

template <typename T = void>
struct
thrust::divides;

struct
thrust::divides< void >;

template <typename T = void>
struct
thrust::modulus;

struct
thrust::modulus< void >;

template <typename T = void>
struct
thrust::negate;

struct
thrust::negate< void >;

template <typename T = void>
struct
thrust::square;

struct
thrust::square< void >;

#define THRUST_UNARY_FUNCTOR_VOID_SPECIALIZATION = see below;
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
#define THRUST_BINARY_FUNCTOR_VOID_SPECIALIZATION = see below;

#define THRUST_BINARY_FUNCTOR_VOID_SPECIALIZATION_OP = see below;
```

成员类

thrust::plus

thrust::plus< void >

thrust::minus

thrust::minus< void >

thrust::multiplies

thrust::multiplies< void >

thrust::divides

thrust::divides< void >

thrust::modulus

thrust::modulus< void >

thrust::negate

thrust::negate< void >

thrust::square

thrust::square< void >

宏（Macros）

THRUST_UNARY_FUNCTOR_VOID_SPECIALIZATION

THRUST_BINARY_FUNCTOR_VOID_SPECIALIZATION

THRUST_BINARY_FUNCTOR_VOID_SPECIALIZATION_OP

2.3.3.2 位运算（Bitwise Operation）

位运算

```
template <typename T = void>
struct
thrust::bit_and;

struct
thrust::bit_and< void >;

template <typename T = void>
struct
thrust::bit_or;

struct
thrust::bit_or< void >;

template <typename T = void>
struct
thrust::bit_xor;
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
struct
thrust::bit_xor< void >;
```

成员类

thrust::bit_and

thrust::bit_and< void >

thrust::bit_or

thrust::bit_or< void >

thrust::bit_xor

thrust::bit_xor< void >

2.3.3.3 比较运算（Comparison Operation）

比较运算

```
template <typename T = void>
struct
thrust::equal_to;

struct
thrust::equal_to< void >;

template <typename T = void>
struct
thrust::not_equal_to;

struct
thrust::not_equal_to< void >;

template <typename T = void>
struct
thrust::greater;

struct
thrust::greater< void >;

template <typename T = void>
struct
thrust::less;

struct
thrust::less< void >;

template <typename T = void>
struct
thrust::greater_equal;
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
struct
thrust::greater_equal< void >;

template <typename T = void>
struct
thrust::less_equal;

struct
thrust::less_equal< void >;
```

成员类

thrust::equal_to

thrust::equal_to< void >

thrust::not_equal_to

thrust::not_equal_to< void >

thrust::greater

thrust::greater< void >

thrust::less

thrust::less< void >

thrust::greater_equal

thrust::greater_equal< void >

thrust::less_equal

thrust::less_equal< void >

2.3.3.4 通用标识运算（Generalized Identity Operation）

通用标识运算

```
template <typename T = void>
struct
thrust::identity;

struct
thrust::identity< void >;

template <typename T = void>
struct
thrust::maximum;
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
struct
thrust::maximum< void >;

template <typename T = void>
struct
thrust::minimum;

struct
thrust::minimum< void >;

template <typename T1 = void,
  typename T2 = void>
struct
thrust::project1st;

struct
thrust::project1st< void, void >;

template <typename T1 = void,
  typename T2 = void>
struct
thrust::project2nd;

struct
thrust::project2nd< void, void >;
```

成员类

thrust::identity

thrust::identity< void >

thrust::maximum

thrust::maximum< void >

thrust::minimum

thrust::minimum< void >

thrust::project1st

thrust::project1st< void, void >

thrust::project2nd

thrust::project2nd< void, void >

2.3.3.5 逻辑运算（Logical Operation）

逻辑运算

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

```
template <typename T = void>
struct
thrust::logical_and;

struct
thrust::logical_and< void >;

template <typename T = void>
struct
thrust::logical_or;

struct
thrust::logical_or< void >;

template <typename T = void>
struct
thrust::logical_not;

struct
thrust::logical_not< void >;
```

成员类

thrust::logical_and

thrust::logical_and< void >

thrust::logical_or

thrust::logical_or< void >

thrust::logical_not

thrust::logical_not< void >

2.4 迭代器（Iterator）

函数

```
template <typename InputIterator,
  typename Distance>
__host__ __device__ void
thrust::advance(InputIterator & i,
  Distance n);

template <typename InputIterator>
__host__ __device__ thrust::iterator_traits< InputIterator >::difference_
↪type
thrust::distance(InputIterator first,
  InputIterator last);
```

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

2.4.1 thrust::advance

```
template <typename InputIterator,
  typename Distance>
__host__ __device__ void
advance(InputIterator & i,
  Distance n);
```

advance(i, n) 使迭代器 i 递增 n 个位置。如果 n > 0 ，则相当于执行 n 次 ++i；如果 n < 0 ，则相当于执行 n 次 --i。如果 n == 0 ，则调用无效。

以下码片段演示了如何使用 advance 使迭代器递增给定次数：

```
#include <thrust/advance.h>
#include <thrust/device_vector.h>
...
thrust::device_vector<int> vec(13);
thrust::device_vector<int>::iterator iter = vec.begin();

thrust::advance(iter, 7);

// iter - vec.begin() == 7
```

模板参数

- **InputIterator**：是 Input Iterator 的模型
- **Distance**：是可转换为 InputIterator 的 distance type 的整型

函数参数

- **i**：待前进的迭代器
- **n**：迭代器前进的距离

前提条件

仅对于双向和随机访问迭代器，n 才为负数。

另请参阅

https://en.cppreference.com/w/cpp/iterator/advance

2.4.2 thrust::distance

```
template <typename InputIterator>
__host__ __device__ thrust::iterator_traits< InputIterator >::difference_
↪type
distance(InputIterator first,
  InputIterator last);
```

distance 计算 first 和 last 之间的距离，即 first 到等于 last 要递增的次数。

以下代码片段演示了如何使用 distance 计算两个迭代器之间的距离：

```
#include <thrust/distance.h>
#include <thrust/device_vector.h>
...
thrust::device_vector<int> vec(13);
thrust::device_vector<int>::iterator iter1 = vec.begin();
thrust::device_vector<int>::iterator iter2 = iter1 + 7;
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
int d = thrust::distance(iter1, iter2);

// d is 7
```

模板参数

InputIterator：是 Input Iterator 的模型

函数参数

- **first**：感兴趣输入范围的起始
- **last**：感兴趣输入范围的末尾

前提条件

如果 InputIterator 满足随机访问迭代器的要求，则可从 first 访问 last 或从 last 访问 first；否则，仅可从 first 访问 last。

返回值

输入范围的起始和末尾之间的距离。

另请参阅

https://en.cppreference.com/w/cpp/iterator/distance

2.4.3 高级迭代器（Fancy Iterator）

高级迭代器

```
template <typename Value,
  typename Incrementable = use_default,
  typename System = use_default>
class
thrust::constant_iterator;

template <typename Incrementable,
  typename System = use_default,
  typename Traversal = use_default,
  typename Difference = use_default>
class
thrust::counting_iterator;

template <typename System = use_default>
class
thrust::discard_iterator;

template <typename Derived,
  typename Base,
  typename Value = use_default,
  typename System = use_default,
  typename Traversal = use_default,
  typename Reference = use_default,
  typename Difference = use_default>
class
thrust::iterator_adaptor;

template <typename Derived,
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
  typename Value,
  typename System,
  typename Traversal,
  typename Reference,
  typename Difference = std::ptrdiff_t>
class
thrust::iterator_facade;

class
thrust::iterator_core_access;

template <typename ElementIterator,
  typename IndexIterator>
class
thrust::permutation_iterator;

template <typename BidirectionalIterator>
class
thrust::reverse_iterator;

template <typename InputFunction,
  typename OutputFunction,
  typename Iterator>
class
thrust::transform_input_output_iterator;

template <class AdaptableUnaryFunction,
  class Iterator,
  class Reference = use_default,
  class Value = use_default>
class
thrust::transform_iterator;

template <typename UnaryFunction,
  typename OutputIterator>
class
thrust::transform_output_iterator;

template <typename IteratorTuple>
class
thrust::zip_iterator;

template <typename ValueT,
  typename IndexT>
__host__ __device__ constant_iterator< ValueT, IndexT >
thrust::make_constant_iterator(ValueT x,
  IndexT i = int());

template <typename V>
__host__ __device__ constant_iterator< V >
thrust::make_constant_iterator(V x);

template <typename Incrementable>
__host__ __device__ counting_iterator< Incrementable >
thrust::make_counting_iterator(Incrementable x);
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
__host__ __device__ discard_iterator
thrust::make_discard_iterator(discard_iterator<>::difference_type i =
↪discard_iterator<>::difference_type(0));

template <typename ElementIterator,
  typename IndexIterator>
__host__ __device__ permutation_iterator< ElementIterator, IndexIterator >
thrust::make_permutation_iterator(ElementIterator e,
  IndexIterator i);

template <typename BidirectionalIterator>
__host__ __device__ reverse_iterator< BidirectionalIterator >
thrust::make_reverse_iterator(BidirectionalIterator x);

template <typename InputFunction,
  typename OutputFunction,
  typename Iterator>
transform_input_output_iterator< InputFunction, OutputFunction, Iterator >
__host__ __device__ thrust::make_transform_input_output_iterator(Iterator
↪io,
  InputFunction input_function,
  OutputFunction output_function);

template <class AdaptableUnaryFunction,
  class Iterator>
__host__ __device__ transform_iterator< AdaptableUnaryFunction, Iterator >
thrust::make_transform_iterator(Iterator it,
  AdaptableUnaryFunction fun);

template <typename UnaryFunction,
  typename OutputIterator>
transform_output_iterator< UnaryFunction, OutputIterator >
__host__ __device__ thrust::make_transform_output_iterator(OutputIterator
↪out,
  UnaryFunction fun);

template <typename... Iterators>
__host__ __device__ zip_iterator< thrust::tuple< Iterators... > >
thrust::make_zip_iterator(thrust::tuple< Iterators... > t);

template <typename... Iterators>
__host__ __device__ zip_iterator< thrust::tuple< Iterators... > >
thrust::make_zip_iterator(Iterators... its);
```

成员类

thrust::constant_iterator

继承自：detail::constant_iterator_base::type

thrust::counting_iterator

继承自：detail::counting_iterator_base::type

thrust::discard_iterator

继承自：detail::discard_iterator_base::type

thrust::iterator_adaptor

继承自：detail::iterator_adaptor_base::type

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

thrust::iterator_facade

thrust::iterator_core_access

thrust::permutation_iterator

继承自: thrust::detail::permutation_iterator_base::type

thrust::reverse_iterator

继承自：detail::reverse_iterator_base::type

thrust::transform_input_output_iterator

继承自: detail::transform_input_output_iterator_base::type

thrust::transform_iterator

继承自：detail::transform_iterator_base::type

thrust::transform_output_iterator

继承自: detail::transform_output_iterator_base::type

thrust::zip_iterator

继承自：detail::zip_iterator_base::type

函数

2.4.3.1 thrust::make_constant_iterator

```
template <typename ValueT,
  typename IndexT>
__host__ __device__ constant_iterator< ValueT, IndexT >
make_constant_iterator(ValueT x,
  IndexT i = int());
```

此版本的 make_constant_iterator 从为值和索引给定的值创建 constant_iterator。constant_iterator 的类型可以由编译器从其参数的类型中推断出来。

函数参数

- **x**：返回的 constant_iterator 的常数值的值
- **i**：序列中返回的 constant_iterator 的索引，此参数的类型默认为 int，参数值默认为 0

返回值

一个由 x & i 给定常数值和索引的新 constant_iterator

另请参阅

constant_iterator

2.4.3.2 thrust::make_constant_iterator

```
template <typename V> __host__ __device__ constant_iterator< V > make_
↪constant_iterator(V x);
```

此版本的 make_constant_iterator 仅使用所需常数值的参数创建 constant_iterator。 将返回 constant_iterator 的索引值设置为 0。

函数参数

x：返回的 constant_iterator 的 的常数值的值

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

返回值

一个常数值等于 x，索引等于 0 的新 constant_iterator

另请参阅

constant_iterator

2.4.3.3 thrust::make_counting_iterator

```
template <typename Incrementable>
__host__ __device__ counting_iterator< Incrementable >
make_counting_iterator(Incrementable x);
```

make_counting_iterator 使用其 Incrementable 计数器的初始值创建 counting_iterator。

函数参数

x：新 counting_iterator 的计数器的初始值

返回值

计数器已初始化为 x 的新 counting_iterator

2.4.3.4 thrust::make_discard_iterator

```
__host__ __device__ discard_iterator
make_discard_iterator(discard_iterator<>::difference_type i = discard_
↪iterator<>::difference_type(0));
```

make_discard_iterator 从可选索引参数创建 discard_iterator。

函数参数

i：范围内返回的 discard_iterator 的索引，参数值默认为 0

返回值

由 i 给定索引的新 discard_iterator

另请参阅

constant_iterator

2.4.3.5 thrust::make_permutation_iterator

```
template <typename ElementIterator,
  typename IndexIterator>
__host__ __device__ permutation_iterator< ElementIterator, IndexIterator >
make_permutation_iterator(ElementIterator e,
  IndexIterator i);
```

make_permutation_iterator 从 ElementIterator（指向待排列的元素范围）和 IndexIterator（指向对值定义索引方案的索引范围）创建 permutation_iterator。

函数参数

- **e**：ElementIterator，指向值范围
- **i**：IndexIterator，指向 e 上使用的索引方案

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

返回值

用 i 排列范围 e 的新 permutation_iterator

另请参阅

permutation_iterator

2.4.3.6 thrust::make_reverse_iterator

```
template <typename BidirectionalIterator>
__host__ __device__ reverse_iterator< BidirectionalIterator >
make_reverse_iterator(BidirectionalIterator x);
```

make_reverse_iterator 从 BidirectionalIterator（指向待反向遍历的元素范围）创建 reverse_iterator。

函数参数

x：BidirectionalIterator，指向待反向遍历的范围

返回值

反向遍历范围 x 的新 reverse_iterator

2.4.3.7 thrust::make_transform_input_output_iterator

```
template <typename InputFunction,
  typename OutputFunction,
  typename Iterator>
transform_input_output_iterator< InputFunction, OutputFunction, Iterator >
__host__ __device__ make_transform_input_output_iterator(Iterator io,
  InputFunction input_function,
  OutputFunction output_function);
```

make_transform_input_output_iterator 从 Iterator、InputFunction 和 OutputFunction 创建 transform_input_output_iterator。

函数参数

- **io**：Iterator，指向 InputFunction 输入读取和 OutputFunction 结果写入的位置
- **input_function**：待对从迭代器读取的值执行的 InputFunction
- **output_function**：待对写入迭代器的值执行的 OutputFunction

另请参阅

transform_input_output_iterator

2.4.3.8 thrust::make_transform_iterator

```
template <class AdaptableUnaryFunction,
  class Iterator>
__host__ __device__ transform_iterator< AdaptableUnaryFunction, Iterator >
make_transform_iterator(Iterator it,
  AdaptableUnaryFunction fun);
```

make_transform_iterator 从 Iterator 和 AdaptableUnaryFunction 创建 transform_iterator。

函数参数

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

- **it**：Iterator，指向新创建的 transform_iterator 的输入范围.
- **fun**：AdaptableUnaryFunction，用于变换新创建的 transform_iterator 中 it 指向的范围.

返回值

通过 fun 变换 it 处范围的新 transform_iterator

另请参阅

transform_iterator

2.4.3.9 thrust::make_transform_output_iterator

```
template <typename UnaryFunction,
  typename OutputIterator>
transform_output_iterator< UnaryFunction, OutputIterator >
__host__ __device__ make_transform_output_iterator(OutputIterator out,
  UnaryFunction fun);
```

make_transform_output_iterator 从 OutputIterator 和 UnaryFunction 创建 transform_output_iterator。

函数参数

- **out**：OutputIterator，指向新创建的 transform_output_iterator 的输出范围
- **fun**：在用新创建的 transform_output_iterator 将对象赋给 out 之前，UnaryFunction 变换对象

另请参阅

transform_output_iterator

2.4.3.10 thrust::make_zip_iterator

```
template <typename... Iterators>
__host__ __device__ zip_iterator< thrust::tuple< Iterators... > >
make_zip_iterator(thrust::tuple< Iterators... > t);
```

make_zip_iterator 从迭代器元组创建 zip_iterator。

函数参数

T：待复制的迭代器元组

返回值

压缩封装在 T 中的迭代器的新 zip_iterator

另请参阅

zip_iterator

2.4.3.11 thrust::make_zip_iterator

```
template <typename... Iterators>
__host__ __device__ zip_iterator< thrust::tuple< Iterators... > >
make_zip_iterator(Iterators... its);
```

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

make_zip_iterator 从迭代器创建 zip_iterator。

函数参数

its： 待复制的迭代器

返回值

压缩迭代器的新 zip_iterator

另请参阅

zip_iterator

2.4.4 迭代器标签（Iterator Tag）

2.4.4.1 迭代器标签类（Iterator Tag Class）

迭代器标签类

```
struct
thrust::input_device_iterator_tag;

struct
thrust::output_device_iterator_tag;

struct
thrust::forward_device_iterator_tag;

struct
thrust::bidirectional_device_iterator_tag;

struct
thrust::random_access_device_iterator_tag;
```

typedef 参见如下 thrust::input_host_iterator_tag;

typedef 参见如下 thrust::output_host_iterator_tag;

typedef 参见如下 thrust::forward_host_iterator_tag;

typedef 参见如下 thrust::bidirectional_host_iterator_tag;

typedef 参见如下 thrust::random_access_host_iterator_tag;

成员类

thrust::input_device_iterator_tag

继 承 自: thrust::detail::iterator_category_with_system_and_traversal< std::input_iterator_tag, thrust::device_system_tag, thrust::single_pass_traversal_tag >

thrust::output_device_iterator_tag

继 承 自: thrust::detail::iterator_category_with_system_and_traversal< std::output_iterator_tag, thrust::device_system_tag, thrust::single_pass_traversal_tag >

thrust::forward_device_iterator_tag

继 承 自: thrust::detail::iterator_category_with_system_and_traversal< std::forward_iterator_tag, thrust::device_system_tag, thrust::forward_traversal_tag >

thrust::bidirectional_device_iterator_tag

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

继 承 自: thrust::detail::iterator_category_with_system_and_traversal< std::bidirectional_iterator_tag, thrust::device_system_tag, thrust::bidirectional_traversal_tag >

thrust::random_access_device_iterator_tag

继 承 自: thrust::detail::iterator_category_with_system_and_traversal< std::random_access_iterator_tag, thrust::device_system_tag, thrust::random_access_traversal_tag >

类型

2.4.4.2 thrust::input_host_iterator_tag

```
typedef std::input_iterator_taginput_host_iterator_tag;
```

input_host_iterator_tag 是一个空类：没有成员函数、成员变量或嵌套类型。仅用于表示 C++ 类型系统中输入主机迭代器概念的 tag。

另请参阅

- https://en.cppreference.com/w/cpp/iterator/iterator_tags
- iterator_traits,
- input_device_iterator_tag,
- output_device_iterator_tag,
- forward_device_iterator_tag,
- bidirectional_device_iterator_tag,
- random_access_device_iterator_tag,
- output_host_iterator_tag, forward_host_iterator_tag,
- bidirectional_host_iterator_tag,
- random_access_host_iterator_tag

2.4.4.3 thrust::output_host_iterator_tag

```
typedef std::output_iterator_tagoutput_host_iterator_tag;
```

output_host_iterator_tag 是一个空类：没有成员函数、成员变量或嵌套类型。仅用于表示 C++ 类型系统中输出主机迭代器概念的 tag。

另请参阅

- https://en.cppreference.com/w/cpp/iterator/iterator_tags
- iterator_traits,
- input_device_iterator_tag,
- output_device_iterator_tag,
- forward_device_iterator_tag,
- bidirectional_device_iterator_tag,
- random_access_device_iterator_tag,
- input_host_iterator_tag, forward_host_iterator_tag,
- bidirectional_host_iterator_tag,

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

- random_access_host_iterator_tag

2.4.4.4 thrust::forward_host_iterator_tag

```
typedef std::forward_iterator_tagforward_host_iterator_tag;
```

forward_host_iterator_tag 是一个空类：没有成员函数、成员变量或嵌套类型。仅用于表示 C++ 类型系统中正向主机迭代器概念的 tag。

另请参阅

- iterator_traits
- input_device_iterator_tag
- output_device_iterator_tag
- forward_device_iterator_tag
- bidirectional_device_iterator_tag
- random_access_device_iterator_tag
- input_host_iterator_tag
- output_host_iterator_tag
- bidirectional_host_iterator_tag
- random_access_host_iterator_tag

2.4.4.5 thrust::bidirectional_host_iterator_tag

```
typedef std::bidirectional_iterator_tagbidirectional_host_iterator_tag;
```

bidirectional_host_iterator_tag 是一个空类：没有成员函数、成员变量或嵌套类型。仅用于表示 C++ 类型系统中正向主机迭代器概念的 tag。

另请参阅

- iterator_traits
- input_device_iterator_tag
- output_device_iterator_tag
- forward_device_iterator_tag
- bidirectional_device_iterator_tag
- random_access_device_iterator_tag
- input_host_iterator_tag, output_host_iterator_tag
- forward_host_iterator_tag
- random_access_host_iterator_tag

2.4.4.6 thrust::random_access_host_iterator_tag

```
typedef std::random_access_iterator_tagrandom_access_host_iterator_tag;
```

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

random_access_host_iterator_tag 是一个空类：没有成员函数、成员变量或嵌套类型。仅用于表示 C++ 类型系统中正向主机迭代器概念的 tag。

另请参阅

- iterator_traits
- input_device_iterator_tag
- output_device_iterator_tag
- forward_device_iterator_tag
- bidirectional_device_iterator_tag
- random_access_device_iterator_tag
- input_host_iterator_tag, output_host_iterator_tag
- forward_host_iterator_tag
- bidirectional_host_iterator_tag

2.5 内存管理（Memory Management）

2.5.1 内存管理

所有与内存分配和释放相关的 Thrust 功能。

组

```
template <typename T>
class thrust::device_reference;

/* device_ptr is a pointer-like object which points to an object that
↪resides in memory associated with the device system. */
template <typename T>
class thrust::device_ptr;

template <typename T>
void
thrust::device_delete(thrust::device_ptr< T > ptr,
const size_t n = 1);

void
thrust::device_free(thrust::device_ptr< void > ptr);

thrust::device_ptr< void >
thrust::device_malloc(const std::size_t n);

template <typename T>
thrust::device_ptr< T >
thrust::device_malloc(const std::size_t n);

template <typename T>
device_ptr< T >
thrust::device_new(device_ptr< void > p,
const size_t n = 1);

template <typename T>
```

(下页继续)

飞曦信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
device_ptr< T >
thrust::device_new(device_ptr< void > p,
const T & exemplar,
const size_t n = 1);

template <typename T>
device_ptr< T >
thrust::device_new(const size_t n = 1);

template <typename T,
typename CharT,
typename Traits>
__host__ std::basic_ostream< CharT, Traits > &
thrust::operator<<(std::basic_ostream< CharT, Traits > & os,
device_ptr< T > const & dp);

/* Create a device_ptr from a raw pointer. */
template <typename T>
__host__ __device__ device_ptr< T >
thrust::device_pointer_cast(T * ptr);

/* Create a device_ptr from another device_ptr. */
template <typename T>
__host__ __device__ device_ptr< T >
thrust::device_pointer_cast(device_ptr< T > const & dptr);

template <typename T>
__host__ __device__ void
thrust::swap(device_reference< T > & x,
device_reference< T > & y);

template <typename T,
typename charT,
typename traits>
std::basic_ostream< charT, traits > &
thrust::operator<<(std::basic_ostream< charT, traits > & os,
const device_reference< T > & y);

template <typename DerivedPolicy>
__host__ __device__ pointer< void, DerivedPolicy >
thrust::malloc(const thrust::detail::execution_policy_base< DerivedPolicy >
 ↪ & system,
std::size_t n);

template <typename T,
typename DerivedPolicy>
__host__ __device__ pointer< T, DerivedPolicy >
thrust::malloc(const thrust::detail::execution_policy_base< DerivedPolicy >
 ↪ & system,
std::size_t n);

template <typename T,
typename DerivedPolicy>
__host__ __device__ thrust::pair< thrust::pointer< T, DerivedPolicy >,
 ↪typename thrust::pointer< T, DerivedPolicy >::difference_type >
thrust::get_temporary_buffer(const thrust::detail::execution_policy_base<
 ↪DerivedPolicy > & system,
```

(下页继续)

飞谱信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
typename thrust::pointer< T, DerivedPolicy >::difference_type n);

template <typename DerivedPolicy,
typename Pointer>
__host__ __device__ void
thrust::free(const thrust::detail::execution_policy_base< DerivedPolicy > &
↪ system,
Pointer ptr);

template <typename DerivedPolicy,
typename Pointer>
__host__ __device__ void
thrust::return_temporary_buffer(const thrust::detail::execution_policy_base
↪< DerivedPolicy > & system,
Pointer p,
std::ptrdiff_t n);

template <typename Pointer>
__host__ __device__ thrust::detail::pointer_traits< Pointer >::raw_pointer
thrust::raw_pointer_cast(Pointer ptr);

template <typename T>
__host__ __device__ detail::raw_reference< T >::type
thrust::raw_reference_cast(T & ref);

template <typename T>
__host__ __device__ detail::raw_reference< constT >::type
thrust::raw_reference_cast(const T & ref);
```

成员类

2.5.1.1 thrust::device_reference

继承自: thrust::reference< T, thrust::device_ptr< T >, thrust::device_reference< T > >

2.5.1.2 thrust::device_ptr

device_ptr 是一个像指针的对象，指向与设备系统关联的内存中的对象。

继承自: thrust::pointer< T, thrust::device_system_tag, thrust::device_reference< T >, thrust::device_ptr< T > >

函数

2.5.1.3 thrust::device_delete

```
template <typename T>
void
device_delete(thrust::device_ptr< T > ptr,
  const size_t n = 1);
```

device_delete 删除用 device_new 分配的 device_ptr。

函数参数

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

- **ptr**：要删除的 device_ptr ，假设已使用 device_new 进行分配
- **n**：ptr 处要销毁的对象数量。默认为 1 ，类似于 device_new

另请参阅

- device_ptr
- device_new

2.5.1.4 thrust::device_free

```
void device_free(thrust::device_ptr< void > ptr);
```

device_free 释放由函数 device_malloc 分配的内存。

以下代码片段演示了如何使用 device_free 释放由 device_malloc 分配的内存：

```
#include <thrust/device_malloc.h>
#include <thrust/device_free.h>
...
// allocate some integers with device_malloc
const int N = 100;
thrust::device_ptr<int> int_array = thrust::device_malloc<int>(N);

// manipulate integers
...

// deallocate with device_free
thrust::device_free(int_array);
```

函数参数

ptr：指向要释放的内存的 device_ptr

另请参阅

- device_ptr
- device_malloc

2.5.1.5 thrust::device_malloc

```
thrust::device_ptr< void > device_malloc(const std::size_t n);
```

此版本的 device_malloc 为字节分配顺序设备存储。

以下代码片段演示了如何使用 device_malloc 分配一定范围的设备内存：

```
#include <thrust/device_malloc.h>
#include <thrust/device_free.h>
...
// allocate some memory with device_malloc
const int N = 100;
thrust::device_ptr<void> void_ptr = thrust::device_malloc(N);

// manipulate memory
...

// deallocate with device_free
thrust::device_free(void_ptr);
```

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

函数参数

n：在设备内存中顺序分配的字节数

返回值

新分配的内存的 device_ptr

另请参阅

- device_ptr
- device_free

2.5.1.6 thrust::device_malloc

```
template <typename T>
thrust::device_ptr< T >
device_malloc(const std::size_t n);
```

此版本的 device_malloc 为给定类型的新对象分配顺序设备存储。

以下代码片段演示了如何使用 device_malloc 分配一定范围的设备内存：

```
#include <thrust/device_malloc.h>
#include <thrust/device_free.h>
...
// allocate some integers with device_malloc
const int N = 100;
thrust::device_ptr<int> int_array = thrust::device_malloc<int>(N);

// manipulate integers
...

// deallocate with device_free
thrust::device_free(int_array);
```

函数参数

n：在设备内存中顺序分配的类型 T 的对象的数量

返回值

新分配的内存的 device_ptr

另请参阅

- device_ptr
- device_free

2.5.1.7 thrust::device_new

```
template <typename T>
device_ptr< T >
device_new(device_ptr< void > p,
  const size_t n = 1);
```

device_new 为设备内存中的类型实现 placement new 运算符。device_new 在设备内存中的对象数组上调用 T 的 null 构造函数。该函数不分配内存。

函数参数

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

- **p**：指向设备内存区域的 device_ptr ，在其中构造一个或多个 T
- **n**：p 处要构造的对象数量

返回值

转换为 T 类型的 p

另请参阅

device_ptr

2.5.1.8 thrust::device_new

```
template <typename T>
device_ptr< T >
device_new(device_ptr< void > p,
  const T & exemplar,
  const size_t n = 1);
```

device_new 为存在于设备内存中的类型实现 placement new 运算符。device_new 在设备内存中的对象数组上调用 T 的拷贝构造函数。该函数不分配内存。

函数参数

- **p**：指向设备内存区域的 device_ptr，在其中构造一个或多个 T 的函数
- **exemplar**：要从中复制的值
- **n**：p 处要构造的对象数量

返回值

p ，转换为 T 的类型

另请参阅

- device_ptr
- fill

2.5.1.9 thrust::device_new

```
template <typename T>
device_ptr< T >
device_new(const size_t n = 1);
```

device_new 为存在于设备内存中的类型实现 new 运算符。它分配的设备内存足够大，可以容纳 n 个类型 T 的新对象。

函数参数

n：要分配的对象数量默认为 1

返回值

device_ptr，指向设备内存中的新分配区域

2.5.1.10 thrust::operator<<

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

```
template <typename T,
  typename CharT,
  typename Traits>
__host__ std::basic_ostream< CharT, Traits > &
operator<<(std::basic_ostream< CharT, Traits > & os,
  device_ptr< T > const & dp);
```

将 device_ptr 指向的地址写入输出流。

函数参数

- **os**：输出流
- **dp**：指向输出的 device_ptr

返回值

os

2.5.1.11 thrust::device_pointer_cast

```
template <typename T>
__host__ __device__ device_ptr< T >
device_pointer_cast(T * ptr);
```

从原始指针创建 device_ptr。

模板参数

T： 任何类型

函数参数

ptr： 指向设备内存中 T 的原始指针

前提条件

ptr 指向设备内存中的位置

返回值

指向 ptr 的 device_ptr<T>

2.5.1.12 thrust::device_pointer_cast

```
template <typename T>
__host__ __device__ device_ptr< T >
device_pointer_cast(device_ptr< T > const & dptr);
```

从另一个 device_ptr 创建 device_ptr。

模板参数

T： 任何类型

函数参数

dptr： 指向 T 的 device_ptr

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

2.5.1.13 thrust::swap

```
template <typename T>
__host__ __device__ void
swap(device_reference< T > & x,
  device_reference< T > & y);
```

将一个 device_reference 的值与另一个进行交换。

x：感兴趣的第一个 device_reference

y：感兴趣的第二个 device_reference

2.5.1.14 thrust::operator<<

```
template <typename T,
  typename charT,
  typename traits>
std::basic_ostream< charT, traits > &
operator<<(std::basic_ostream< charT, traits > & os,
  const device_reference< T > & y);
```

将 device_reference 的值写入输出流。

函数参数

- **os**：输出流
- **y**：指向输出的 device_reference

返回值

os

2.5.1.15 thrust::malloc

```
template <typename DerivedPolicy>
__host__ __device__ pointer< void, DerivedPolicy >
malloc(const thrust::detail::execution_policy_base< DerivedPolicy > &
↪system,
  std::size_t n);
```

此版本的 malloc 分配与给定系统关联的、未类型化的未初始化存储。

以下代码片段演示了如何使用 malloc 分配与 Thrust 设备系统关联的内存范围：

```
#include <thrust/memory.h>
...
// allocate some memory with thrust::malloc
const int N = 100;
thrust::device_system_tag device_sys;
thrust::pointer<void,thrust::device_space_tag> void_ptr =
↪thrust::malloc(device_sys, N);

// manipulate memory
...

// deallocate void_ptr with thrust::free
thrust::free(device_sys, void_ptr);
```

飞曦信息 MetaX Confidential 2024-11-18 15:00:00

模板参数

DerivedPolicy：派生执行策略的名称

函数参数

- **system**：关联存储的 Thrust 系统
- **n**：要分配的存储字节数

前提条件

DerivedPolicy 必须由 thrust::execution_policy<DerivedPolicy> 公开派生

返回值

如果分配成功，则返回一个指针，指向已分配的存储；否则返回空指针，必须通过 thrust::free 释放指针

另请参阅

- free
- device_malloc

2.5.1.16 thrust::malloc

```
template <typename T,
  typename DerivedPolicy>
__host__ __device__ pointer< T, DerivedPolicy >
malloc(const thrust::detail::execution_policy_base< DerivedPolicy > &
 ↪system,
  std::size_t n);
```

此版本的 malloc 分配与给定系统关联的、类型化的未初始化存储。

以下代码片段演示了如何使用 malloc 分配一个内存范围来容纳与 Thrust 设备系统关联的整数：

```
#include <thrust/memory.h>
...
// allocate storage for 100 ints with thrust::malloc
const int N = 100;
thrust::device_system_tag device_sys;
thrust::pointer<int,thrust::device_system_tag> ptr = thrust::malloc<int>
 ↪(device_sys, N);

// manipulate memory
...

// deallocate ptr with thrust::free
thrust::free(device_sys, ptr);
```

模板参数

DerivedPolicy：派生执行策略的名称

函数参数

- **system**：关联存储的 Thrust 系统
- **n**：存储应容纳的类型 T 的元素的数量

前提条件

DerivedPolicy 必须由 thrust::execution_policy<DerivedPolicy> 公开派生.

返回值

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

如果分配成功，则返回一个指针，指向足够大以容纳 n 个类型 T 的元素的内存分配；否则返回空指针，必须通过 thrust::free 释放指针

另请参阅

- free
- device_malloc

2.5.1.17 thrust::get_temporary_buffer

```
template <typename T,
  typename DerivedPolicy>
__host__ __device__ thrust::pair< thrust::pointer< T, DerivedPolicy >,
 ↪typename thrust::pointer< T, DerivedPolicy >::difference_type >
get_temporary_buffer(const thrust::detail::execution_policy_base<
 ↪DerivedPolicy > & system,
  typename thrust::pointer< T, DerivedPolicy >::difference_type n);
```

get_temporary_buffer 返回一个指针，指向与给定 Thrust 系统关联的、足以存储多达 n 个类型 T 的对象的存储。如果没有足够的存储空间来容纳 n 个对象，则可能会返回较小的缓冲区。返回的缓冲区可以容纳的对象数量也会返回。

Thrust 在分配算法实现所需的临时存储时在内部使用 get_temporary_buffer。

使用 get_temporary_buffer 分配的存储必须使用 return_temporary_buffer 返回到系统。

以下代码片段演示了如何使用 get_temporary_buffer 分配一个内存范围来容纳与 Thrust 设备系统关联的整数：

```
#include <thrust/memory.h>
...
// allocate storage for 100 ints with thrust::get_temporary_buffer
const int N = 100;

typedef thrust::pair<
  thrust::pointer<int,thrust::device_system_tag>,
  std::ptrdiff_t
> ptr_and_size_t;

thrust::device_system_tag device_sys;
ptr_and_size_t ptr_and_size = thrust::get_temporary_buffer<int>(device_sys,
 ↪ N);

// manipulate up to 100 ints
for(int i = 0; i < ptr_and_size.second; ++i)
{
  *ptr_and_size.first = i;
}

// deallocate storage with thrust::return_temporary_buffer
thrust::return_temporary_buffer(device_sys, ptr_and_size.first);
```

模板参数

DerivedPolicy：派生执行策略的名称

函数参数

- **system**：关联存储的 Thrust 系统
- **n**：存储容纳类型 T 的对象所需的数量

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

前提条件

DerivedPolicy 必须由 thrust::execution_policy<DerivedPolicy> 公开派生.

返回值

p pair ，其中 p.first 是指向分配的存储的指针，p.second 是存储可以容纳的类型 T 的连续对象的数量。如果无法分配存储，返回 p.first；如果无法获得存储，必须使用 return_temporary_buffer 将存储返回到系统。

另请参阅

- malloc
- return_temporary_buffer

2.5.1.18 thrust::free

```
template <typename DerivedPolicy,
  typename Pointer>
__host__ __device__ void
free(const thrust::detail::execution_policy_base< DerivedPolicy > & system,
  Pointer ptr);
```

free 释放之前由 thrust::malloc 分配的存储。

以下代码片段演示了如何使用 free 释放之前由 thrust::malloc 分配的存储：

```
#include <thrust/memory.h>
...
// allocate storage for 100 ints with thrust::malloc
const int N = 100;
thrust::device_system_tag device_sys;
thrust::pointer<int,thrust::device_system_tag> ptr = thrust::malloc<int>
↪(device_sys, N);

// mainpulate memory
...

// deallocate ptr with thrust::free
thrust::free(device_sys, ptr);
```

模板参数

DerivedPolicy：派生执行策略的名称

函数参数

- **system**：与存储关联的 Thrust 系统
- **ptr**：之前由 thrust::malloc 返回的指针， 如果 ptr 为空，则 free 不会执行任何操作

前提条件

应已通过之前对一些类型 T 调用 thrust::malloc(system, n) 或 thrust::malloc<T>(system, n) 而返回 ptr

2.5.1.19 thrust::return_temporary_buffer

```
template <typename DerivedPolicy,
  typename Pointer>
__host__ __device__ void
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
return_temporary_buffer(const thrust::detail::execution_policy_base<
↪DerivedPolicy > & system,
  Pointer p,
  std::ptrdiff_t n);
```

return_temporary_buffer 释放之前由 get_temporary_buffer 分配、与给定 Thrust 系统关联的存储。

Thrust 在释放算法实现所需的临时存储时在内部使用 return_temporary_buffer。

以下代码片段演示了如何使用 return_temporary_buffer 释放之前由 get_temporary_buffer 分配的存储范围：

```
#include <thrust/memory.h>
...
// allocate storage for 100 ints with thrust::get_temporary_buffer
const int N = 100;

typedef thrust::pair<
  thrust::pointer<int,thrust::device_system_tag>,
  std::ptrdiff_t
> ptr_and_size_t;

thrust::device_system_tag device_sys;
ptr_and_size_t ptr_and_size = thrust::get_temporary_buffer<int>(device_sys,
↪ N);

// manipulate up to 100 ints
for(int i = 0; i < ptr_and_size.second; ++i)
{
  *ptr_and_size.first = i;
}

// deallocate storage with thrust::return_temporary_buffer
thrust::return_temporary_buffer(device_sys, ptr_and_size.first);
```

模板参数

DerivedPolicy：派生执行策略的名称

函数参数

- **system**：与存储关联的 Thrust 系统
- **p**：之前由 thrust::get_temporary_buffer 返回的指针，如果 ptr 为空，则 return_temporary_buffer 不会执行任何操作

前提条件

p 之前应该已由 thrust::get_temporary_buffer 分配

另请参阅

- free
- get_temporary_buffer

2.5.1.20 thrust::raw_pointer_cast

```
template <typename Pointer> __host__ __device__ thrust::detail::pointer_
↪traits< Pointer >::raw_pointer raw_pointer_cast(Pointer ptr);
```

沐曦信息 MetaX Confidential 2024-11-18 15:00:00

raw_pointer_cast 从像指针的类型创建一个原始（raw）指针，如果存在，只需返回包装指针（wrapped pointer）。

函数参数

ptr：感兴趣的指针

返回值

如果表达式格式正确，返回 ptr.get()；否则，返回 ptr

另请参阅

raw_reference_cast

2.5.1.21 thrust::raw_reference_cast

```
template <typename T> __host__ __device__ detail::raw_reference< T >::type
↪raw_reference_cast(T & ref);
```

raw_reference_cast 从包装的引用类型创建一个原始引用，如果存在，只需返回底层引用。

如果参数不是引用包装器，则结果是对该参数的引用。

备注

raw_reference_cast 有两个版本： 一个用于 const 引用；另一个用于 non-const 引用

函数参数

ref：感兴趣的引用

返回值

*thrust::raw_pointer_cast(&ref)

另请参阅

raw_pointer_cast

2.5.1.22 thrust::raw_reference_cast

```
template <typename T> __host__ __device__ detail::raw_reference< constT >
↪::type raw_reference_cast(const T & ref);
```

raw_reference_cast 从包装的引用类型创建一个原始引用，如果存在，只需返回底层引用。

如果参数不是引用包装器，则结果是对该参数的引用。

备注

raw_reference_cast 有两个版本： 一个用于 const 引用；另一个用于 non-const 引用

函数参数

ref：感兴趣的引用

返回值

*thrust::raw_pointer_cast(&ref)

另请参阅

raw_pointer_cast

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

2.5.2 分配器（Allocator）

分配器

```
template <typename Upstream>
class
thrust::device_ptr_memory_resource;

template <typename T>
class
thrust::device_allocator;

template <typename T>
class
thrust::device_malloc_allocator;

template <typename T>
class
thrust::device_new_allocator;

template <typename T,
  class MR>
class
thrust::mr::allocator;

template <typename T,
  typename Upstream>
class
thrust::mr::stateless_resource_allocator;

template <typename T,
  typename Pointer>
using
thrust::mr::polymorphic_allocator = see below;

template <typename T,
  typename MR>
__host__ __device__ bool
thrust::mr::operator==(const allocator< T, MR > & lhs,
  const allocator< T, MR > & rhs);

template <typename T,
  typename MR>
__host__ __device__ bool
thrust::mr::operator!=(const allocator< T, MR > & lhs,
  const allocator< T, MR > & rhs);
```

成员类

thrust::device_ptr_memory_resource

继承自: thrust::mr::memory_resource< device_ptr< void > >

thrust::device_allocator

在设备可访问的内存中创建新元素的分配器。

继承自:

- thrust::mr::stateless_resource_allocator< T, device_ptr_memory_resource< device_memory_resource > >

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

- thrust::mr::allocator< T, Upstream >
- thrust::mr::validator< MR >

thrust::device_malloc_allocator

thrust::device_new_allocator

thrust::mr::allocator

继承自：thrust::mr::validator< MR >

thrust::mr::stateless_resource_allocator

继承自:

- thrust::mr::allocator< T, Upstream >
- thrust::mr::validator< MR >

类型

2.5.2.1 thrust::mr::polymorphic_allocator

```
template <typename T,
  typename Pointer>
using
polymorphic_allocator = allocator< T, polymorphic_adaptor_resource<
↪Pointer > >;
```

函数

2.5.2.2 thrust::mr::operator==

```
template <typename T,
  typename MR>
__host__ __device__ bool
operator==(const allocator< T, MR > & lhs,
  const allocator< T, MR > & rhs);
```

通过比较底层内存资源来比较分配器的相等性。

2.5.2.3 thrust::mr::operator!=

```
template <typename T,
  typename MR>
__host__ __device__ bool
operator!=(const allocator< T, MR > & lhs,
  const allocator< T, MR > & rhs);
```

通过比较底层内存资源来比较分配器的不相等性。

2.5.3 内存资源（Memory Resources）

内存资源

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

```
template <typename Upstream,
  typename Bookkeeper>
class
thrust::mr::disjoint_unsynchronized_pool_resource;

template <typename Upstream,
  typename Bookkeeper>
struct
thrust::mr::disjoint_synchronized_pool_resource;

template <typename Pointer = void *>
class
thrust::mr::memory_resource;

class
thrust::mr::memory_resource< void * >;

class
thrust::mr::new_delete_resource;

template <typename Upstream>
class
thrust::mr::unsynchronized_pool_resource;

struct
thrust::mr::pool_options;

template <typename Upstream>
struct
thrust::mr::synchronized_pool_resource;
```

typedef 参见如下 thrust::system::cpp::memory_resource;

typedef 参见如下 thrust::system::cpp::universal_memory_resource;

typedef 参见如下 thrust::system::cpp::universal_host_pinned_memory_resource;

typedef 参见如下 thrust::system::omp::memory_resource;

typedef 参见如下 thrust::system::omp::universal_memory_resource;

typedef 参见如下 thrust::system::omp::universal_host_pinned_memory_resource;

typedef 参见如下 thrust::system::tbb::memory_resource;

typedef 参见如下 thrust::system::tbb::universal_memory_resource;

typedef 参见如下 thrust::system::tbb::universal_host_pinned_memory_resource;

```
template <typename T>
using thrust::universal_ptr = see below;

template <typename Upstream,
  typename Bookkeeper>
__host__ thrust::mr::disjoint_unsynchronized_pool_resource< Upstream,
↪Bookkeeper > &
thrust::mr::tls_disjoint_pool(Upstream * upstream = NULL,
  Bookkeeper * bookkeeper = NULL);

template <typename Pointer>
__host__ __device__ bool
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-19 15:00:00

(续上页)

```
thrust::mr::operator==(const memory_resource< Pointer > & lhs,
  const memory_resource< Pointer > & rhs);

template <typename Pointer>
__host__ __device__ bool
thrust::mr::operator!=(const memory_resource< Pointer > & lhs,
  const memory_resource< Pointer > & rhs);

template <typename MR>
__host__ MR * thrust::mr::get_global_resource();

template <typename Upstream,
  typename Bookkeeper>
__host__ thrust::mr::unsynchronized_pool_resource< Upstream > &
thrust::mr::tls_pool(Upstream * upstream = NULL);
```

成员类

thrust::mr::disjoint_unsynchronized_pool_resource

继承自:

- thrust::mr::memory_resource< Upstream::pointer >
- thrust::mr::validator2< Upstream, Bookkeeper >

thrust::mr::disjoint_synchronized_pool_resource

继承自: thrust::mr::memory_resource< Upstream::pointer >

thrust::mr::memory_resource

继承:

- thrust::mr::new_delete_resource
- thrust::mr::polymorphic_adaptor_resource< Pointer >
- thrust::mr::memory_resource< void * >

thrust::mr::new_delete_resource

继承自: thrust::mr::memory_resource<>

thrust::mr::unsynchronized_pool_resource

继承自:

- thrust::mr::memory_resource< Upstream::pointer >
- thrust::mr::validator< Upstream >

thrust::mr::pool_options

thrust::mr::synchronized_pool_resource

继承自: thrust::mr::memory_resource< Upstream::pointer >

类型

2.5.3.1 thrust::system::cpp::memory_resource

typedef detail::native_resourcememory_resource;

标准 C++ 系统的内存资源。使用 mr::new_delete_resource 并用 cpp::pointer 标记它。

飞猪信息 MetaX Confidential 2024-11-18 15:00:00

2.5.3.2 thrust::system::cpp::universal_memory_resource

typedef detail::universal_native_resourceuniversal_memory_resource;
标准 C++ 系统的统一内存资源。使用 mr::new_delete_resource 并用 cpp::universal_pointer 标记它。

2.5.3.3 thrust::system::cpp::universal_host_pinned_memory_resource

typedef detail::native_resourceuniversal_host_pinned_memory_resource;
cpp::universal_memory_resource 的别名。

2.5.3.4 thrust::system::omp::memory_resource

typedef detail::native_resourcememory_resource;
OpenMP 系统的内存资源。使用 mr::new_delete_resource 并用 omp::pointer 标记它。

2.5.3.5 thrust::system::omp::universal_memory_resource

typedef detail::universal_native_resourceuniversal_memory_resource;
OpenMP 系统的统一内存资源。使用 mr::new_delete_resource 并用 omp::universal_pointer 标记它。

2.5.3.6 thrust::system::omp::universal_host_pinned_memory_resource

typedef detail::native_resourceuniversal_host_pinned_memory_resource;
omp::universal_memory_resource 的别名。

2.5.3.7 thrust::system::tbb::memory_resource

typedef detail::native_resourcememory_resource;
TBB 系统的内存资源。使用 mr::new_delete_resource 并用 tbb::pointer 标记它。

2.5.3.8 thrust::system::tbb::universal_memory_resource

typedef detail::universal_native_resourceuniversal_memory_resource;
TBB 系统的统一内存资源。使用 mr::new_delete_resource 并用 tbb::universal_pointer 标记它。

2.5.3.9 thrust::system::tbb::universal_host_pinned_memory_resource

typedef detail::native_resourceuniversal_host_pinned_memory_resource;
tbb::universal_memory_resource 的别名。

飞猪信息 MetaX Confidential 2024-11-18 15:00:00

2.5.3.10 thrust::universal_ptr

```
template <typename T>
using universal_ptr = thrust::system::__THRUST_DEVICE_SYSTEM_
↪NAMESPACE::universal_pointer< T >;
```

universal_ptr 存储一个指针，指向主机和设备都可以访问的内存中分配的对象。

使用此类型指针调度的算法将被调度到主机或设备，具体取决于使用的后端。显式策略（thrust::device，等）可以用于指定算法应该在哪里运行。

universal_ptr 具有指针语义：它可以从主机和设备安全地解引用，并且可以使用指针算法进行控制。

可以通过 universal_allocator ，或使用原始指针显式调用其构造函数来创建 universal_ptr。

universal_ptr 封装的原始指针可以通过其 get 方法或 raw_pointer_cast 自由函数获得。

备注

universal_ptr 不是一个智能指针，释放 universal_ptr 指向的内存需要程序员的操作

另请参阅

- host_ptr，由 universal_ptr 共享的完整接口的文档
- raw_pointer_cast

函数

2.5.3.11 thrust::mr::tls_disjoint_pool

```
template <typename Upstream,
  typename Bookkeeper>
__host__ thrust::mr::disjoint_unsynchronized_pool_resource< Upstream,
↪Bookkeeper > &
tls_disjoint_pool(Upstream * upstream = NULL,
  Bookkeeper * bookkeeper = NULL);
```

如果尚未创建，则可能构造并返回线程本地 disjoint_unsynchronized_pool_resource 的地址。

模板参数

- **Upstream**：池模板的第一个模板参数
- **Bookkeeper**：池模板的第二个模板参数

函数参数

- **upstream**：构造函数的第一个参数（如果调用）
- **bookkeeper**：构造函数的第二个参数（如果调用）

2.5.3.12 thrust::mr::operator==

```
template <typename Pointer>
__host__ __device__ bool
operator==(const memory_resource< Pointer > & lhs,
  const memory_resource< Pointer > & rhs);
```

先根据标识比较内存资源的相等性，再根据 is_equal。

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

2.5.3.13 thrust::mr::operator!=

```
template <typename Pointer>
__host__ __device__ bool
operator!=(const memory_resource< Pointer > & lhs,
  const memory_resource< Pointer > & rhs);
```

先根据标识比较内存资源的不相等性，再根据 is_equal。

2.5.3.14 thrust::mr::get_global_resource

```
template <typename MR>
__host__ MR *
get_global_resource();
```

返回 MR 的全局实例，作为函数局部静态变量而创建。

模板参数

MR：要从中获取实例的内存资源类型，必须为 DefaultConstructible

返回值

指向 MR 全局实例的指针

2.5.3.15 thrust::mr::tls_pool

```
template <typename Upstream,
  typename Bookkeeper>
__host__ thrust::mr::unsynchronized_pool_resource< Upstream > &
tls_pool(Upstream * upstream = NULL);
```

如果尚未创建，则可能构造并返回线程本地 unsynchronized_pool_resource 的地址。

模板参数

Upstream：池模板的模板参数

函数参数：

upstream：构造函数的参数（如果调用）

2.6 数值（Numeric）

2.6.1 复数（Cmplex Number）

```
template <typename T>
struct thrust::complex;

typedef see below thrust::complex::value_type;

__host__ __device__
thrust::complex::complex(const T & re);

__host__ __device__
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
thrust::complex::complex(const T & re,
const T & im);

template <typename U>
__host__ __device__
thrust::complex::complex(const complex< U > & z);

__host__ __device__
thrust::complex::complex(const std::complex< T > & z);

template <typename U>
__host__ __device__
thrust::complex::complex(const std::complex< U > & z);

__host__ __device__ complex &
thrust::complex::operator=(const T & re);

complex &
thrust::complex::operator=(const complex< T > & z) = default;

template <typename U>
__host__ __device__ complex &
thrust::complex::operator=(const complex< U > & z);

__host__ __device__ complex &
thrust::complex::operator=(const std::complex< T > & z);

template <typename U>
__host__ __device__ complex &
thrust::complex::operator=(const std::complex< U > & z);

template <typename U>
__host__ __device__ complex< T > &
thrust::complex::operator+=(const complex< U > & z);

template <typename U>
__host__ __device__ complex< T > &
thrust::complex::operator-=(const complex< U > & z);

template <typename U>
__host__ __device__ complex< T > &
thrust::complex::operator*=(const complex< U > & z);

template <typename U>
__host__ __device__ complex< T > &
thrust::complex::operator/=(const complex< U > & z);

template <typename U>
__host__ __device__ complex< T > &
thrust::complex::operator+=(const U & z);

template <typename U>
__host__ __device__ complex< T > &
thrust::complex::operator-=(const U & z);

template <typename U>
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
__host__ __device__ complex< T > &
thrust::complex::operator*=(const U & z);

template <typename U>
__host__ __device__ complex< T > &
thrust::complex::operator/=(const U & z);

__host__ __device__ T
thrust::complex::real() const;

__host__ __device__ T
thrust::complex::imag() const;

__host__ __device__ T
thrust::complex::real() const;

__host__ __device__ T
thrust::complex::imag() const;

__host__ __device__ void
thrust::complex::real(T re);

__host__ __device__ void
thrust::complex::imag(T im);

__host__ __device__ void
thrust::complex::real(T re);

__host__ __device__ void
thrust::complex::imag(T im);

__host__
thrust::complex::complex< T >() const;

template <typename T>
__host__ __device__ T
thrust::abs(const complex< T > & z);

template <typename T>
__host__ __device__ T
thrust::arg(const complex< T > & z);

template <typename T>
__host__ __device__ T
thrust::norm(const complex< T > & z);

template <typename T>
__host__ __device__ complex< T >
thrust::conj(const complex< T > & z);

template <typename T0,
typename T1>
__host__ __device__ complex< typename detail::promoted_numerical_type< T0,
↪T1 >::type >
thrust::polar(const T0 & m,
const T1 & theta = T1());
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
template <typename T>
__host__ __device__ complex< T >
thrust::proj(const T & z);

template <typename T0,
typename T1>
__host__ __device__ complex< typename detail::promoted_numerical_type< T0,
↪T1 >::type >
thrust::operator+(const complex< T0 > & x,
const complex< T1 > & y);

template <typename T0,
typename T1>
__host__ __device__ complex< typename detail::promoted_numerical_type< T0,
↪T1 >::type >
thrust::operator+(const complex< T0 > & x,
const T1 & y);

template <typename T0,
typename T1>
__host__ __device__ complex< typename detail::promoted_numerical_type< T0,
↪T1 >::type >
thrust::operator+(const T0 & x,
const complex< T1 > & y);

template <typename T0,
typename T1>
__host__ __device__ complex< typename detail::promoted_numerical_type< T0,
↪T1 >::type >
thrust::operator-(const complex< T0 > & x,
const complex< T1 > & y);

template <typename T0,
typename T1>
__host__ __device__ complex< typename detail::promoted_numerical_type< T0,
↪T1 >::type >
thrust::operator-(const complex< T0 > & x,
const T1 & y);

template <typename T0,
typename T1>
__host__ __device__ complex< typename detail::promoted_numerical_type< T0,
↪T1 >::type >
thrust::operator-(const T0 & x,
const complex< T1 > & y);

template <typename T0,
typename T1>
__host__ __device__ complex< typename detail::promoted_numerical_type< T0,
↪T1 >::type >
thrust::operator*(const complex< T0 > & x,
const complex< T1 > & y);

template <typename T0,
typename T1>
```

(下页继续)

飞沐信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
__host__ __device__ complex< typename detail::promoted_numerical_type< T0,
 ↪T1 >::type >
thrust::operator*(const complex< T0 > & x,
const T1 & y);

template <typename T0,
typename T1>
__host__ __device__ complex< typename detail::promoted_numerical_type< T0,
 ↪T1 >::type >
thrust::operator*(const T0 & x,
const complex< T1 > & y);

template <typename T0,
typename T1>
__host__ __device__ complex< typename detail::promoted_numerical_type< T0,
 ↪T1 >::type >
thrust::operator/(const complex< T0 > & x,
const complex< T1 > & y);

template <typename T0,
typename T1>
__host__ __device__ complex< typename detail::promoted_numerical_type< T0,
 ↪T1 >::type >
thrust::operator/(const complex< T0 > & x,
const T1 & y);

template <typename T0,
typename T1>
__host__ __device__ complex< typename detail::promoted_numerical_type< T0,
 ↪T1 >::type >
thrust::operator/(const T0 & x,
const complex< T1 > & y);

template <typename T>
__host__ __device__ complex< T >
thrust::operator+(const complex< T > & y);

template <typename T>
__host__ __device__ complex< T >
thrust::operator-(const complex< T > & y);

template <typename T>
__host__ __device__ complex< T >
thrust::exp(const complex< T > & z);

template <typename T>
__host__ __device__ complex< T >
thrust::log(const complex< T > & z);

template <typename T>
__host__ __device__ complex< T >
thrust::log10(const complex< T > & z);

template <typename T0,
typename T1>
__host__ __device__ complex< typename detail::promoted_numerical_type< T0,
 ↪T1 >::type >
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
thrust::pow(const complex< T0 > & x,
const complex< T1 > & y);

template <typename T0,
typename T1>
__host__ __device__ complex< typename detail::promoted_numerical_type< T0,
↪T1 >::type >
thrust::pow(const complex< T0 > & x,
const T1 & y);

template <typename T0,
typename T1>
__host__ __device__ complex< typename detail::promoted_numerical_type< T0,
↪T1 >::type >
thrust::pow(const T0 & x,
const complex< T1 > & y);

template <typename T>
__host__ __device__ complex< T >
thrust::sqrt(const complex< T > & z);

template <typename T>
__host__ __device__ complex< T >
thrust::cos(const complex< T > & z);

template <typename T>
__host__ __device__ complex< T >
thrust::sin(const complex< T > & z);

template <typename T>
__host__ __device__ complex< T >
thrust::tan(const complex< T > & z);

template <typename T>
__host__ __device__ complex< T >
thrust::cosh(const complex< T > & z);

template <typename T>
__host__ __device__ complex< T >
thrust::sinh(const complex< T > & z);

template <typename T>
__host__ __device__ complex< T >
thrust::tanh(const complex< T > & z);

template <typename T>
__host__ __device__ complex< T >
thrust::acos(const complex< T > & z);

template <typename T>
__host__ __device__ complex< T >
thrust::asin(const complex< T > & z);

template <typename T>
__host__ __device__ complex< T >
thrust::atan(const complex< T > & z);
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
template <typename T>
__host__ __device__ complex< T >
thrust::acosh(const complex< T > & z);

template <typename T>
__host__ __device__ complex< T >
thrust::asinh(const complex< T > & z);

template <typename T>
__host__ __device__ complex< T >
thrust::atanh(const complex< T > & z);

template <typename T,
typename CharT,
typename Traits>
std::basic_ostream< CharT, Traits > &
thrust::operator<<(std::basic_ostream< CharT, Traits > & os,
const complex< T > & z);

template <typename T,
typename CharT,
typename Traits>
__host__ std::basic_istream< CharT, Traits > &
thrust::operator>>(std::basic_istream< CharT, Traits > & is,
complex< T > & z);

template <typename T0,
typename T1>
__host__ __device__ bool
thrust::operator==(const complex< T0 > & x,
const complex< T1 > & y);

template <typename T0,
typename T1>
__host__ __device__ bool
thrust::operator==(const complex< T0 > & x,
const std::complex< T1 > & y);

template <typename T0,
typename T1>
__host__ __device__ bool
thrust::operator==(const std::complex< T0 > & x,
const complex< T1 > & y);

template <typename T0,
typename T1>
__host__ __device__ bool
thrust::operator==(const T0 & x,
const complex< T1 > & y);

template <typename T0,
typename T1>
__host__ __device__ bool
thrust::operator==(const complex< T0 > & x,
const T1 & y);
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
template <typename T0,
typename T1>
__host__ __device__ bool
thrust::operator!=(const complex< T0 > & x,
const complex< T1 > & y);

template <typename T0,
typename T1>
__host__ __device__ bool
thrust::operator!=(const complex< T0 > & x,
const std::complex< T1 > & y);

template <typename T0,
typename T1>
__host__ __device__ bool
thrust::operator!=(const std::complex< T0 > & x,
const complex< T1 > & y);

template <typename T0,
typename T1>
__host__ __device__ bool
thrust::operator!=(const T0 & x,
const complex< T1 > & y);

template <typename T0,
typename T1>
__host__ __device__ bool
thrust::operator!=(const complex< T0 > & x,
const T1 & y);
```

成员类

- Struct thrust::complex

类型

```
typedef Tvalue_type;
```

value_type 是复数实数部分和虚数部分的类型。

函数

2.6.1.1 thrust::complex::complex

```
__host__ __device__
complex(const T & re);
```

构造一个虚数部分为 0 的复数。

函数参数

re： 数字的实数部分

函数

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

2.6.1.2 thrust::complex::complex

```
__host__ __device__
complex(const T & re,
  const T & im);
```

从实数部分和虚数部分构造复数。

函数参数

- re：数字的实数部分
- im：数字的虚数部分

2.6.1.3 thrust::complex::complex

```
complex() = default;
```

默认构造复数。

2.6.1.4 thrust::complex::complex

```
complex(const complex< T > & z) = default;
```

此拷贝构造函数从一个类型可转换为此复数 value_type 的复数进行复制。

函数参数

z：要从中复制的复数

2.6.1.5 thrust::complex::complex

```
template <typename U>
__host__ __device__
complex(const complex< U > & z);
```

此转换拷贝构造函数从一个类型可转换为此复数 value_type 的复数进行复制。

模板参数

U：可转换为 value_type

函数参数

z：要从中复制的复数

2.6.1.6 thrust::complex::complex

```
__host__ __device__
complex(const std::complex< T > & z);
```

此转换拷贝构造函数从一个类型可转换为此复数 value_type 的 std::complex 进行复制。

函数参数

z：要从中复制的复数

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

2.6.1.7 thrust::complex::complex

```
template <typename U>
__host__ __device__
complex(const std::complex< U > & z);
```

此转换拷贝构造函数从一个类型可转换为此复数 value_type 的 std::complex 进行复制。

模板参数

U：可转换为 value_type

函数参数

z：要从中复制的复数

2.6.1.8 thrust::complex::operator=

```
__host__ __device__ complex &
operator=(const T & re);
```

将 re 赋给该复数的实数部分，并将虚数部分设置为 0。

函数参数

re： 数字的实数部分

2.6.1.9 thrust::complex::operator=

```
complex &
operator=(const complex< T > & z) = default;
```

分别将 z.real() 和 z.real() 赋给该复数的实数部分和虚数部分。

函数参数

z：要从中复制的复数

2.6.1.10 thrust::complex::operator=

```
template <typename U>
__host__ __device__ complex &
operator=(const complex< U > & z);
```

分别将 z.real() 和 z.real() 赋给该复数的实数部分和虚数部分。

模板参数

U：可转换为 value_type

函数参数

z：要从中复制的复数

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

2.6.1.11 thrust::complex::operator=

```
__host__ __device__ complex &
operator=(const std::complex< T > & z);
```

分别将 z.real() 和 z.real() 赋给该复数的实数部分和虚数部分。

函数参数

z：要从中复制的复数

2.6.1.12 thrust::complex::operator=

```
template <typename U>
__host__ __device__ complex &
operator=(const std::complex< U > & z);
```

分别将 z.real() 和 z.real() 赋给该复数的实数部分和虚数部分。

模板参数

U：可转换为 value_type

函数参数

z：要从中复制的复数

2.6.1.13 thrust::complex::operator+=

```
template <typename U>
__host__ __device__ complex< T > &
operator+=(const complex< U > & z);
```

将此复数和一个复数相加，并将结果赋给此复数。

模板参数

U：可转换为 value_type

函数参数

z：待相加的复数

2.6.1.14 thrust::complex::operator-=

```
template <typename U>
__host__ __device__ complex< T > &
operator-=(const complex< U > & z);
```

从该复数中减去一个复数，并将结果赋给此复数。

模板参数

U：可转换为 value_type

函数参数

z：要减去的复数

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

2.6.1.15 thrust::complex::operator*=

```
template <typename U>
__host__ __device__ complex< T > &
operator*=(const complex< U > & z);
```

将此复数乘以另一个复数，并将结果赋给此复数。

模板参数

U：可转换为 value_type

函数参数

z： 待相乘的复数

2.6.1.16 thrust::complex::operator/=

```
template <typename U>
__host__ __device__ complex< T > &
operator/=(const complex< U > & z);
```

将此复数除以另一个复数，并将结果赋给此复数。

模板参数

U：可转换为 value_type

函数参数

z： 待被除的复数

2.6.1.17 thrust::complex::operator+=

```
template <typename U>
__host__ __device__ complex< T > &
operator+=(const U & z);
```

将此复数和一个标量相加，并将结果赋给此复数。

模板参数

U：可转换为 value_type

函数参数

z： 待相加的复数

2.6.1.18 thrust::complex::operator-=

```
template <typename U>
__host__ __device__ complex< T > &
operator-=(const U & z);
```

从该复数中减去一个标量，并将结果赋给此复数。

模板参数

U：可转换为 value_type

函数参数

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

z：要减去的标量

2.6.1.19 thrust::complex::operator*=

```
template <typename U>
__host__ __device__ complex< T > &
operator*=(const U & z);
```

将此复数乘以一个标量，并将结果赋给此复数。

模板参数

U：可转换为 value_type

函数参数

z： 待相乘的标量

2.6.1.20 thrust::complex::operator/=

```
template <typename U>
__host__ __device__ complex< T > &
operator/=(const U & z);
```

将此复数除以一个标量，并将结果赋给此复数。

模板参数

U：可转换为 value_type

函数参数

z： 待被除的标量

2.6.1.21 thrust::complex::real

```
__host__ __device__ T
real() const;
```

返回此复数的实数部分。

2.6.1.22 thrust::complex::imag

```
__host__ __device__ T
imag() const;
```

返回此复数的虚数部分。

2.6.1.23 thrust::complex::real

```
__host__ __device__ T
real() const;
```

返回此复数的实数部分。

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

2.6.1.24 thrust::complex::imag

```
__host__ __device__ T
imag() const;
```

返回此复数的虚数部分。

2.6.1.25 thrust::complex::real

```
__host__ __device__ void
real(T re);
```

设置此复数的实数部分。

函数参数

re：此复数新的实数部分。

2.6.1.26 thrust::complex::imag

```
__host__ __device__ void
imag(T im);
```

设置此复数的虚数部分。

函数参数

im：此复数新的虚数部分。

2.6.1.27 thrust::complex::real

```
__host__ __device__ void
real(T re);
```

设置此复数的实数部分。

函数参数

re：此复数新的实数部分。

2.6.1.28 thrust::complex::imag

```
__host__ __device__ void
imag(T im);
```

设置此复数的虚数部分。

函数参数

im：此复数新的虚数部分

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

2.6.1.29 thrust::complex::complex< T >

```
__host__
operator std::complex< T >() const;
```

将此复数强制转换为相同类型的 std::complex。

2.6.1.30 thrust::abs

```
template <typename T>
__host__ __device__ T
abs(const complex< T > & z);
```

返回复数的模（也称为绝对值）。

函数参数

z：从中计算绝对值的复数

2.6.1.31 thrust::arg

```
template <typename T>
__host__ __device__ T
arg(const complex< T > & z);
```

返回复数的辐角（也称为参数），以弧度为单位。

函数参数

z：从中计算辐角的复数

2.6.1.32 thrust::norm

```
template <typename T>
__host__ __device__ T
norm(const complex< T > & z);
```

返回复数的模的平方。

函数参数

z：从中计算模的平方的复数

2.6.1.33 thrust::conj

```
template <typename T>
__host__ __device__ complex< T >
conj(const complex< T > & z);
```

返回复数的复共扼。

函数参数

z：从中计算复共扼的复数

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

2.6.1.34 thrust::polar

```
template <typename T0,
  typename T1>
__host__ __device__ complex< typename detail::promoted_numerical_type< T0,
 ↪T1 >::type >
polar(const T0 & m,
  const T1 & theta = T1());
```

返回具有指定模和辐角的复数。

函数参数

- m：返回复数的模
- theta：返回复数的辐角，以弧度为单位

2.6.1.35 thrust::proj

```
template <typename T>
__host__ __device__ complex< T >
proj(const T & z);
```

返回复数在黎曼球面上的投影。对于所有有限复数，它都返回参数。对于具有非有限部分的复数，返回(INFINITY,+/-0)，其中零的符号与参数虚数部分的符号匹配。

函数参数

z：复数参数

2.6.1.36 thrust::operator+

```
template <typename T0,
  typename T1>
__host__ __device__ complex< typename detail::promoted_numerical_type< T0,
 ↪T1 >::type >
operator+(const complex< T0 > & x,
  const complex< T1 > & y);
```

将两个复数相加。

两个复数的值类型应该是一致的，并且返回复数的类型是两个参数的提升类型。

函数参数

- x：第一个复数
- y：第二个复数

2.6.1.37 thrust::operator+

```
template <typename T0,
  typename T1>
__host__ __device__ complex< typename detail::promoted_numerical_type< T0,
 ↪T1 >::type >
operator+(const complex< T0 > & x,
  const T1 & y);
```

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

将一个标量和一个复数相加。

复数的值类型和标量的应该是一致的，并且返回复数的类型是两个参数的提升类型。

函数参数

- x：复数
- y：标量

2.6.1.38 thrust::operator+

```
template <typename T0,
  typename T1>
__host__ __device__ complex< typename detail::promoted_numerical_type< T0,
↪T1 >::type >
operator+(const T0 & x,
  const complex< T1 > & y);
```

将一个复数和一个标量相加。

复数的值类型和标量的应该是一致的，并且返回复数的类型是两个参数的提升类型。

函数参数

- x：标量
- y：复数

2.6.1.39 thrust::operator-

```
template <typename T0,
  typename T1>
__host__ __device__ complex< typename detail::promoted_numerical_type< T0,
↪T1 >::type >
operator-(const complex< T0 > & x,
  const complex< T1 > & y);
```

将两个复数相减。

两个复数类型的值类型应该是一致的，并且返回复数的类型是两个参数的提升类型。

函数参数

- x：第一个复数（被减数）
- y：第二个复数（减数）

2.6.1.40 thrust::operator-

```
template <typename T0,
  typename T1>
__host__ __device__ complex< typename detail::promoted_numerical_type< T0,
↪T1 >::type >
operator-(const complex< T0 > & x,
  const T1 & y);
```

从复数中减去标量。

复数的值类型和标量的应该是一致的，并且返回复数的类型是两个参数的提升类型。

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

函数参数

- x：复数（被减数）
- y：标量（减数）

2.6.1.41 thrust::operator-

```
template <typename T0,
  typename T1>
__host__ __device__ complex< typename detail::promoted_numerical_type< T0,
↪T1 >::type >
operator-(const T0 & x,
  const complex< T1 > & y);
```

从标量中减去复数。

复数的值类型和标量的应该是一致的，并且返回复数的类型是两个参数的提升类型。

函数参数

- x：标量（被减数）
- y：复数（减数）

2.6.1.42 thrust::operator*

```
template <typename T0,
  typename T1>
__host__ __device__ complex< typename detail::promoted_numerical_type< T0,
↪T1 >::type >
operator*(const complex< T0 > & x,
  const complex< T1 > & y);
```

将两个复数相乘。

两个复数类型的值类型应该是一致的，并且返回复数的类型是两个参数的提升类型。

函数参数

- x：第一个复数
- y：第二个复数

2.6.1.43 thrust::operator*

```
template <typename T0,
  typename T1>
__host__ __device__ complex< typename detail::promoted_numerical_type< T0,
↪T1 >::type >
operator*(const complex< T0 > & x,
  const T1 & y);
```

将复数与标量相乘。

函数参数

- x：复数
- y：标量

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

2.6.1.44 thrust::operator*

```
template <typename T0,
  typename T1>
__host__ __device__ complex< typename detail::promoted_numerical_type< T0,
↪T1 >::type >
operator*(const T0 & x,
  const complex< T1 > & y);
```

将标量与复数相乘。

复数的值类型和标量的应该是一致的，并且返回复数的类型是两个参数的提升类型。

函数参数

- x：标量
- y：复数

2.6.1.45 thrust::operator/

```
template <typename T0,
  typename T1>
__host__ __device__ complex< typename detail::promoted_numerical_type< T0,
↪T1 >::type >
operator/(const complex< T0 > & x,
  const complex< T1 > & y);
```

将两个复数相除。

两个复数类型的值类型应该是一致的，并且返回复数的类型是两个参数的提升类型。

函数参数

- x：分子（被除数）
- y：分母（除数）

2.6.1.46 thrust::operator/

```
template <typename T0,
  typename T1>
__host__ __device__ complex< typename detail::promoted_numerical_type< T0,
↪T1 >::type >
operator/(const complex< T0 > & x,
  const T1 & y);
```

用复数除以标量。

复数的值类型和标量的应该是一致的，并且返回复数的类型是两个参数的提升类型。

函数参数

- x：复数分子（被除数）
- y：标量分母（除数）

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

2.6.1.47 thrust::operator/

```
template <typename T0,
  typename T1>
__host__ __device__ complex< typename detail::promoted_numerical_type< T0,
↪T1 >::type >
operator/(const T0 & x,
  const complex< T1 > & y);
```

用标量除以复数。

复数的值类型和标量的应该是一致的，并且返回复数的类型是两个参数的提升类型。

函数参数

- x：标量分子（被除数）
- y：复数分母（除数）

2.6.1.48 thrust::operator+

```
template <typename T>
__host__ __device__ complex< T >
operator+(const complex< T > & y);
```

一元加法，返回其复数参数。

函数参数

y：复数参数

2.6.1.49 thrust::operator-

```
template <typename T>
__host__ __device__ complex< T >
operator-(const complex< T > & y);
```

一元减法，返回其复数参数的加法逆元（或称相反数）。

函数参数

y：复数参数

2.6.1.50 thrust::exp

```
template <typename T>
__host__ __device__ complex< T >
exp(const complex< T > & z);
```

返回复数的复指数。

函数参数

z：复数参数

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

2.6.1.51 thrust::log

```
template <typename T>
__host__ __device__ complex< T >
log(const complex< T > & z);
```

返回复数的复自然对数。

函数参数

z：复数参数

2.6.1.52 thrust::log10

```
template <typename T>
__host__ __device__ complex< T >
log10(const complex< T > & z);
```

返回复数的复底 10 对数。

函数参数

z：复数参数

2.6.1.53 thrust::pow

```
template <typename T0,
  typename T1>
__host__ __device__ complex< typename detail::promoted_numerical_type< T0,
↪T1 >::type >
pow(const complex< T0 > & x,
  const complex< T1 > & y);
```

返回一个复数的另一个复数次幂。

两个复数类型的值类型应该是一致的，并且返回复数的类型是两个参数的提升类型。

函数参数

- x：底数
- y：指数

2.6.1.54 thrust::pow

```
template <typename T0,
  typename T1>
__host__ __device__ complex< typename detail::promoted_numerical_type< T0,
↪T1 >::type >
pow(const complex< T0 > & x,
  const T1 & y);
```

返回一个复数的标量次幂。

复数的值类型和标量的应该是一致的，并且返回复数的类型是两个参数的提升类型。

函数参数

- x：底数

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

- y：指数

2.6.1.55 thrust::pow

```
template <typename T0,
  typename T1>
__host__ __device__ complex< typename detail::promoted_numerical_type< T0,
↪T1 >::type >
pow(const T0 & x,
  const complex< T1 > & y);
```

返回一个标量的复数次幂。

复数的值类型和标量的应该是一致的，并且返回复数的类型是两个参数的提升类型。

函数参数

- x：底数
- y：指数

2.6.1.56 thrust::sqrt

```
template <typename T>
__host__ __device__ complex< T >
sqrt(const complex< T > & z);
```

返回复数的复平方根。

函数参数

z：复数参数

2.6.1.57 thrust::cos

```
template <typename T>
__host__ __device__ complex< T >
cos(const complex< T > & z);
```

返回复数的复余弦。

函数参数

z：复数参数

2.6.1.58 thrust::sin

```
template <typename T>
__host__ __device__ complex< T >
sin(const complex< T > & z);
```

返回复数的复正弦。

函数参数

z：复数参数

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

2.6.1.59 thrust::tan

```
template <typename T>
__host__ __device__ complex< T >
tan(const complex< T > & z);
```

返回复数的复正切。

函数参数

z：复数参数

2.6.1.60 thrust::cosh

```
template <typename T>
__host__ __device__ complex< T >
cosh(const complex< T > & z);
```

返回复数的复双曲余弦。

函数参数

z：复数参数

2.6.1.61 thrust::sinh

```
template <typename T>
__host__ __device__ complex< T >
sinh(const complex< T > & z);
```

返回复数的复双曲正弦。

函数参数

z：复数参数

2.6.1.62 thrust::tanh

```
template <typename T>
__host__ __device__ complex< T >
tanh(const complex< T > & z);
```

返回复数的复双曲正切。

函数参数

z：复数参数

2.6.1.63 thrust::acos

```
template <typename T>
__host__ __device__ complex< T >
acos(const complex< T > & z);
```

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

返回复数的复反余弦。

结果的实数部分范围是 [0, Pi] ，虚数部分范围是 [-inf, +inf]。

函数参数

z：复数参数

2.6.1.64 thrust::asin

```
template <typename T>
__host__ __device__ complex< T >
asin(const complex< T > & z);
```

返回复数的复反正弦。

结果的实数部分范围是 [-Pi/2, Pi/2] ，虚数部分范围是 [-inf, +inf]。

函数参数

z：复数参数

2.6.1.65 thrust::atan

```
template <typename T>
__host__ __device__ complex< T >
atan(const complex< T > & z);
```

返回复数的复反正切。

结果的实数部分范围是 [-Pi/2, Pi/2] ，虚数部分范围是 [-inf, +inf]。

函数参数

z：复数参数

2.6.1.66 thrust::acosh

```
template <typename T>
__host__ __device__ complex< T >
acosh(const complex< T > & z);
```

返回复数的复反双曲余弦。

结果的实数部分范围是 [0, +inf] ，虚数部分范围是 [-Pi, Pi]。

函数参数

z：复数参数

2.6.1.67 thrust::asinh

```
template <typename T>
__host__ __device__ complex< T >
asinh(const complex< T > & z);
```

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

返回复数的复反双曲正弦。

结果的实数部分范围是 [-inf, +inf] ， 虚数部分范围是 [-Pi/2, Pi/2]。

函数参数

z：复数参数

2.6.1.68 thrust::atanh

```
template <typename T>
__host__ __device__ complex< T >
atanh(const complex< T > & z);
```

返回复数的复反双曲正切。

结果的实数部分范围是 [-inf, +inf] ， 虚数部分范围是 [-Pi/2, Pi/2]。

函数参数

z：复数参数

2.6.1.69 thrust::operator<<

```
template <typename T,
  typename CharT,
  typename Traits>
std::basic_ostream< CharT, Traits > &
operator<<(std::basic_ostream< CharT, Traits > & os,
  const complex< T > & z);
```

将格式为 (real, imaginary) 的复数写入输出流。

函数参数

- os：输出流
- z：待输出的复数

2.6.1.70 thrust::operator>>

```
template <typename T,
  typename CharT,
  typename Traits>
__host__ std::basic_istream< CharT, Traits > &
operator>>(std::basic_istream< CharT, Traits > & is,
  complex< T > & z);
```

从输出流读取一个复数。

公认格式为：

- real
- (real)
- (real, imaginary)

读取的值必须可转换为复数的 value_type。

函数参数

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

- is：输入流
- z：待设置的复数

2.6.1.71 thrust::operator==

```
template <typename T0,
  typename T1>
__host__ __device__ bool
operator==(const complex< T0 > & x,
  const complex< T1 > & y);
```

如果两个复数相等，返回 true；否则，返回 false。

函数参数

- x：第一个复数
- y：第二个复数

2.6.1.72 thrust::operator==

```
template <typename T0,
  typename T1>
__host__ __device__ bool
operator==(const complex< T0 > & x,
  const std::complex< T1 > & y);
```

如果两个复数相等，返回 true；否则，返回 false。

函数参数

- x：第一个复数
- y：第二个复数

2.6.1.73 thrust::operator==

```
template <typename T0,
  typename T1>
__host__ __device__ bool
operator==(const std::complex< T0 > & x,
  const complex< T1 > & y);
```

如果两个复数相等，返回 true；否则，返回 false。

函数参数

- x：第一个复数
- y：第二个复数

2.6.1.74 thrust::operator==

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

```
template <typename T0,
  typename T1>
__host__ __device__ bool
operator==(const T0 & x,
  const complex< T1 > & y);
```

如果复数的虚数部分为零，实数部分和标量相等，返回 true； 否则，返回 false。

函数参数

- x：标量
- y：复数

2.6.1.75 thrust::operator==

```
template <typename T0,
  typename T1>
__host__ __device__ bool
operator==(const complex< T0 > & x,
  const T1 & y);
```

如果复数的虚数部分为零，实数部分和标量相等，返回 true； 否则，返回 false。

函数参数

- x：复数
- y：标量

2.6.1.76 thrust::operator!=

```
template <typename T0,
  typename T1>
__host__ __device__ bool
operator!=(const complex< T0 > & x,
  const complex< T1 > & y);
```

如果两个复数不同，返回 true；否则，返回 false。

函数参数

- x：第一个复数
- y：第二个复数

2.6.1.77 thrust::operator!=

```
template <typename T0,
  typename T1>
__host__ __device__ bool
operator!=(const complex< T0 > & x,
  const std::complex< T1 > & y);
```

如果两个复数不同，返回 true；否则，返回 false。

函数参数

- x：第一个复数

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

- y：第二个复数

2.6.1.78 thrust::operator!=

```
template <typename T0,
  typename T1>
__host__ __device__ bool
operator!=(const std::complex< T0 > & x,
  const complex< T1 > & y);
```

如果两个复数不同，返回 true；否则，返回 false。

函数参数

- x：第一个复数
- y：第二个复数

2.6.1.79 thrust::operator!=

```
template <typename T0,
  typename T1>
__host__ __device__ bool
operator!=(const T0 & x,
  const complex< T1 > & y);
```

如果复数的虚数部分不为零，或者实数部分和标量不同，返回 true；否则，返回 false。

函数参数

- x：标量
- y：复数

2.6.1.80 thrust::operator!=

```
template <typename T0,
  typename T1>
__host__ __device__ bool
operator!=(const complex< T0 > & x,
  const T1 & y);
```

如果复数的虚数部分不为零，或者实数部分和标量不同，返回 true；否则，返回 false。

函数参数

- x：复数
- y：标量

2.7 并行执行策略（Parallel Execution Policy）

并行执行策略

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

```
template <typename DerivedPolicy>
struct
thrust::host_execution_policy;

template <typename DerivedPolicy>
struct
thrust::device_execution_policy;

static const detail::host_t thrust::host;

THRUST_INLINE_CONSTANT detail::device_t thrust::device;
```

成员类

thrust::host_execution_policy

继承自: thrust::system::__THRUST_HOST_SYSTEM_NAMESPACE::execution_policy< DerivedPolicy >

thrust::device_execution_policy

继承自: thrust::system::__THRUST_DEVICE_SYSTEM_NAMESPACE::execution_policy< DerivedPolicy >

变量

2.7.1 thrust::host

static const detail::host_t host;

thrust::host 是由 THRUST_HOST_SYSTEM 宏配置的与 Thrust 的主机后端系统相关联的默认并行执行策略。

用户可以通过提供 thrust::host 作为算法参数，直接将算法调度目标对准 Thrust 的主机系统，而不是通过迭代器系统标签来依赖隐式算法调度。

显式调度有助于避免将数据副本引入容器中，例如 thrust::host_vector。

请注意，即使 thrust::host 以主机 CPU 为目标，它也是一个并行执行策略。即，算法调用仿函数或解引用迭代器的顺序没有定义。

thrust::host 的类型是实现定义的（implementation-defined）。

以下代码片段演示了如何使用 thrust::host 向主机后端系统显式调度对 thrust::for_each 的调用：

```
#include <thrust/for_each.h>
#include <thrust/execution_policy.h>
#include <cstdio>

struct printf_functor
{
  __host__ __device__
  void operator()(int x)
  {
 printf("%d\n", x);
  }
};
...
int vec(3);
vec[0] = 0; vec[1] = 1; vec[2] = 2;
```

(下页继续)

飞书信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
thrust::for_each(thrust::host, vec.begin(), vec.end(), printf_functor());

// 0 1 2 is printed to standard output in some unspecified order
```

另请参阅

- host_execution_policy
- thrust::device

2.7.2 thrust::device

THRUST_INLINE_CONSTANT detail::device_t device;

thrust::device 是由 THRUST_DEVICE_SYSTEM 宏配置的与 Thrust 的设备后端系统相关联的默认并行执行策略。

用户可以通过提供 thrust::device 作为算法参数，直接将算法调度目标对准 Thrust 的设备系统，而不是通过迭代器系统标签来依赖隐式算法调度。

显式调度有助于避免将数据副本引入到诸如 thrust::device_vector 之类的容器中，或避免将 MXMACA API 分配的原始指针包装为诸如 thrust::device_ptr 之类的类型。

用户必须注意保证提供给算法的迭代器与设备后端系统兼容。例如，由 std::malloc 分配的原始指针通常不能被 GPU 解引用。因此，当设备后端为 MXMACA 时，主机 API 分配的原始指针不应与 thrust::device 算法调用混合。

thrust::device 的类型是实现定义的。

以下代码片段演示了如何使用 thrust::device 向设备后端系统显式调度对 thrust::for_each 的调用：

```
#include <thrust/for_each.h>
#include <thrust/device_vector.h>
#include <thrust/execution_policy.h>
#include <cstdio>

struct printf_functor
{
  __host__ __device__
  void operator()(int x)
  {
 printf("%d\n", x);
  }
};
...
thrust::device_vector<int> vec(3);
vec[0] = 0; vec[1] = 1; vec[2] = 2;

thrust::for_each(thrust::device, vec.begin(), vec.end(), printf_functor());

// 0 1 2 is printed to standard output in some unspecified order
```

另请参阅

- host_execution_policy
- thrust::device

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

2.8 随机数生成（Random Number Generation）

- Random Number Engines with Predefined Parameters
- Random Number Distributions Class Templates
- Random Number Engine Adaptor Class Templates
- Random Number Engine Class Templates

```
/* thrust::random 是包含随机数引擎类模板、随机数引擎适配器类模板、带预定义参数的引擎和随机数分布类模板的命名空间。
          它们在一个单独的命名空间中提供，以便于导入，也在顶级 thrust 命名空间中有别名，以便于访问 */
namespace thrust::random {… }
```

2.8.1 带预定义参数的随机数引擎（Random Number Engines with Predefined Parameter）

带预定义参数的随机数引擎

```
/* 一个带预定义参数的随机数引擎，用于实现 RANLUX 3 级随机数生成算法。 */
typedef see below thrust::random::ranlux24;

/* 一个带预定义参数的随机数引擎，用于实现 RANLUX 4 级随机数生成算法。 */
typedef see below thrust::random::ranlux48;

/* 一个带预定义参数的随机数引擎，用于实现 L’Ecuyer 1996 three-component
↪Tausworthe 随机数生成器。 */
typedef see below thrust::random::taus88;

/* 一个实现定义的 `default`随机数引擎 */
typedef see below thrust::random::default_random_engine;

/* 一个带预定义参数的随机数引擎，用于实现最小标准随机数生成算法的一个版本。 */
typedef see below thrust::random::minstd_rand0;

/* 一个带预定义参数的随机数引擎，用于实现最小标准随机数生成算法的一个版本。 */
typedef see below thrust::random::minstd_rand;

/* 一个带预定义参数的随机数引擎，用于实现 ranlux24 随机数引擎的基本引擎。 */
typedef see below thrust::random::ranlux24_base;

/* 一个带预定义参数的随机数引擎，用于实现 ranlux48 随机数引擎的基本引擎。 */
typedef see below thrust::random::ranlux48_base;
```

类型

2.8.1.1 thrust::random::ranlux24

typedef discard_block_engine< ranlux24_base, 223, 23 >ranlux24;

一个带预定义参数的随机数引擎，用于实现 RANLUX 3 级随机数生成算法。

备注

对类型 ranlux24 的默认构造对象的第 10000 次连续调用应生成值 9901578。

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

2.8.1.2 thrust::random::ranlux48

typedef discard_block_engine< ranlux48_base, 389, 11 >ranlux48;

一个带预定义参数的随机数引擎，用于实现 RANLUX 4 级随机数生成算法。

备注

对类型 ranlux48 的默认构造对象的第 10000 次连续调用应生成值 88229545517833。

2.8.1.3 thrust::random::taus88

```
`typedef`
        `xor_combine_engine< linear_feedback_shift_engine<
↪thrust::detail::uint32_t, 32u, 31u, 13u, 12u >,`
        `0, xor_combine_engine< linear_feedback_shift_engine<
↪thrust::detail::uint32_t, 32u, 29u, 2u, 4u >,`
        `0, linear_feedback_shift_engine< thrust::detail::uint32_t, 32u,
↪28u, 3u, 17u >, 0 >, 0 >taus88;`
```

一个带预定义参数的随机数引擎，用于实现 L’Ecuyer 1996 three-component Tausworthe 随机数生成器。

备注

对类型 taus88 的默认构造对象的第 10000 次连续调用应生成值 3535848941。

2.8.1.4 thrust::random::default_random_engine

typedef minstd_randdefault_random_engine;

一个实现定义的 default 随机数引擎。

备注

default_random_engine 是 minstd_rand 的别名，可能在后续版本中有变更。

2.8.1.5 thrust::random::minstd_rand0

typedef linear_congruential_engine< thrust::detail::uint32_t, 16807, 0, 2147483647 >minstd_rand0;

一个带预定义参数的随机数引擎，用于实现最小标准随机数生成算法的一个版本。

备注

对类型 minstd_rand0 的默认构造对象的第 10000 次连续调用应生成值 1043618065。

2.8.1.6 thrust::random::minstd_rand

typedef linear_congruential_engine< thrust::detail::uint32_t, 48271, 0, 2147483647 >minstd_rand;

一个带预定义参数的随机数引擎，用于实现最小标准随机数生成算法的一个版本。

备注

对类型 minstd_rand 的默认构造对象的第 10000 次连续调用应生成值 399268537。

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

2.8.1.7 thrust::random::ranlux24_base

typedef subtract_with_carry_engine< thrust::detail::uint32_t, 24, 10, 24 >ranlux24_base;

一个带预定义参数的随机数引擎，用于实现 ranlux24 随机数引擎的基本引擎。

备注

对类型 ranlux24_base 的默认构造对象的第 10000 次连续调用应生成值 7937952。

2.8.1.8 thrust::random::ranlux48_base

typedef subtract_with_carry_engine< thrust::detail::uint64_t, 48, 5, 12 >ranlux48_base;

一个带预定义参数的随机数引擎，用于实现 ranlux48 随机数引擎的基本引擎。

备注

对类型 ranlux48_base 的默认构造对象的第 10000 次连续调用应生成值 192113843633948。

2.8.2 随机数分布类模板（Random Number Distributions Class Template）

随机数分布类模板

```
/* normal_distribution 随机数分布产生浮点正态分布的随机数。 */
template <typename RealType = double>
class thrust::random::normal_distribution;

/* uniform_int_distribution 随机数分布从给定范围产生有符号或无符号的整数均匀随机数。
↪*/
template <typename IntType = int>
class thrust::random::uniform_int_distribution;

/* uniform_real_distribution 随机数分布从指定范围内产生浮点均匀随机数 */
template <typename RealType = double>
class thrust::random::uniform_real_distribution;

template <typename RealType>
__host__ __device__ bool
thrust::random::operator==(const normal_distribution< RealType > & lhs,
  const normal_distribution< RealType > & rhs);

template <typename RealType>
__host__ __device__ bool
thrust::random::operator!=(const normal_distribution< RealType > & lhs,
  const normal_distribution< RealType > & rhs);

template <typename RealType,
  typename CharT,
  typename Traits>
std::basic_ostream< CharT, Traits > &
thrust::random::operator<<(std::basic_ostream< CharT, Traits > & os,
  const normal_distribution< RealType > & d);

template <typename RealType,
  typename CharT,
  typename Traits>
std::basic_istream< CharT, Traits > &
```

(下页继续)

飞腾信息 MetaX Confidential 2021-11-18 15:00:00

(续上页)

```
thrust::random::operator>>(std::basic_istream< CharT, Traits > & is,
  normal_distribution< RealType > & d);

template <typename IntType>
__host__ __device__ bool
thrust::random::operator==(const uniform_int_distribution< IntType > & lhs,
  const uniform_int_distribution< IntType > & rhs);

template <typename IntType>
__host__ __device__ bool
thrust::random::operator!=(const uniform_int_distribution< IntType > & lhs,
  const uniform_int_distribution< IntType > & rhs);

template <typename IntType,
  typename CharT,
  typename Traits>
std::basic_ostream< CharT, Traits > &
thrust::random::operator<<(std::basic_ostream< CharT, Traits > & os,
  const uniform_int_distribution< IntType > & d);

template <typename IntType,
  typename CharT,
  typename Traits>
std::basic_istream< CharT, Traits > &
thrust::random::operator>>(std::basic_istream< CharT, Traits > & is,
  uniform_int_distribution< IntType > & d);

template <typename RealType>
__host__ __device__ bool
thrust::random::operator==(const uniform_real_distribution< RealType > &
↪lhs,
  const uniform_real_distribution< RealType > & rhs);

template <typename RealType>
__host__ __device__ bool
thrust::random::operator!=(const uniform_real_distribution< RealType > &
↪lhs,
  const uniform_real_distribution< RealType > & rhs);

template <typename RealType,
  typename CharT,
  typename Traits>
std::basic_ostream< CharT, Traits > &
thrust::random::operator<<(std::basic_ostream< CharT, Traits > & os,
  const uniform_real_distribution< RealType > & d);

template <typename RealType,
  typename CharT,
  typename Traits>
std::basic_istream< CharT, Traits > &
thrust::random::operator>>(std::basic_istream< CharT, Traits > & is,
  uniform_real_distribution< RealType > & d);
```

成员类

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

2.8.2.1 thrust::random::normal_distribution

normal_distribution

随机数分布产生浮点正态分布随机数。

2.8.2.2 thrust::random::uniform_int_distribution

uniform_int_distribution

随机数分布从给定范围产生有符号或无符号的整数均匀随机数。

2.8.2.3 thrust::random::uniform_real_distribution

uniform_real_distribution

随机数分布从指定范围内产生浮点均匀随机数。

函数

2.8.2.4 thrust::random::operator==

```
template <typename RealType>
__host__ __device__ bool
operator==(const normal_distribution< RealType > & lhs,
  const normal_distribution< RealType > & rhs);
```

此函数确认两个 normal_distributions 的相等性。

函数参数

- **lhs**：待测试的第一个 normal_distribution
- **rhs**：待测试的第二个 normal_distribution

返回值

如果 lhs 等于 rhs，返回 true；否则，返回 false

2.8.2.5 thrust::random::operator!=

```
template <typename RealType>
__host__ __device__ bool
operator!=(const normal_distribution< RealType > & lhs,
  const normal_distribution< RealType > & rhs);
```

此函数确认两个 normal_distributions 的不相等性。

函数参数

- **lhs**：待测试的第一个 normal_distribution
- **rhs**：待测试的第二个 normal_distribution

返回值

如果 lhs 不等于 rhs，返回 true；否则，返回 false

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

2.8.2.6 thrust::random::operator<<

```
template <typename RealType,
  typename CharT,
  typename Traits>
std::basic_ostream< CharT, Traits > &
operator<<(std::basic_ostream< CharT, Traits > & os,
  const normal_distribution< RealType > & d);
```

此函数将 normal_distribution 流式传输到 std::basic_ostream。

函数参数

- **os**：要流输出到的 basic_ostream
- **d**：要流输出的 normal_distribution

返回值

os

2.8.2.7 thrust::random::operator>>

```
template <typename RealType,
  typename CharT,
  typename Traits>
std::basic_istream< CharT, Traits > &
operator>>(std::basic_istream< CharT, Traits > & is,
  normal_distribution< RealType > & d);
```

此函数从 std::basic_istream 流式传输 normal_distribution。

函数参数

- **is**：要从中进行流式传输的 basic_istream
- **d**：要流输入的 normal_distribution

返回值

is

2.8.2.8 thrust::random::operator==

```
template <typename IntType>
__host__ __device__ bool
operator==(const uniform_int_distribution< IntType > & lhs,
  const uniform_int_distribution< IntType > & rhs);
```

此函数确认两个 uniform_int_distributions 的相等性。

函数参数

- **lhs**：待测试的第一个 uniform_int_distribution
- **rhs**：待测试的第二个 uniform_int_distribution

返回值

如果 lhs 等于 rhs，返回 true；否则，返回 false

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

2.8.2.9 thrust::random::operator!=

```
template <typename IntType>
__host__ __device__ bool
operator!=(const uniform_int_distribution< IntType > & lhs,
  const uniform_int_distribution< IntType > & rhs);
```

此函数确认两个 uniform_int_distributions 的不相等性。

函数参数

- **lhs**：待测试的第一个 uniform_int_distribution
- **rhs**：待测试的第二个 uniform_int_distribution

返回值

如果 lhs 不等于 rhs，返回 true；否则，返回 false

2.8.2.10 thrust::random::operator<<

```
template <typename IntType,
  typename CharT,
  typename Traits>
std::basic_ostream< CharT, Traits > &
operator<<(std::basic_ostream< CharT, Traits > & os,
  const uniform_int_distribution< IntType > & d);
```

此函数将 uniform_int_distribution 流式传输到 std::basic_ostream。

函数参数

- **os**：要流输出到的 basic_ostream
- **d**：要流输出的 uniform_int_distribution

返回值

os

2.8.2.11 thrust::random::operator>>

```
template <typename IntType,
  typename CharT,
  typename Traits>
std::basic_istream< CharT, Traits > &
operator>>(std::basic_istream< CharT, Traits > & is,
  uniform_int_distribution< IntType > & d);
```

此函数从 std::basic_istream 流式传输 uniform_int_distribution。

函数参数

- **is**：要从中进行流式传输的 basic_istream
- **d**：要流输入的 uniform_int_distribution

返回值

is

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

2.8.2.12 thrust::random::operator==

```
template <typename RealType>
__host__ __device__ bool
operator==(const uniform_real_distribution< RealType > & lhs,
  const uniform_real_distribution< RealType > & rhs);
```

此函数确认两个 uniform_real_distributions 是否相等。

函数参数

- **lhs**：待测试的第一个 uniform_real_distribution
- **rhs**：待测试的第二个 uniform_real_distribution

返回值

如果 lhs 等于 rhs，返回 true；否则，返回 false

2.8.2.13 thrust::random::operator!=

```
template <typename RealType>
__host__ __device__ bool
operator!=(const uniform_real_distribution< RealType > & lhs,
  const uniform_real_distribution< RealType > & rhs);
```

此函数确认两个 uniform_real_distributions 的不相等性。

函数参数

- **lhs**：待测试的第一个 uniform_real_distribution
- **rhs**：待测试的第二个 uniform_real_distribution

返回值

如果 lhs 不等于 rhs，返回 true；否则，返回 false

2.8.2.14 thrust::random::operator<<

```
template <typename RealType,
  typename CharT,
  typename Traits>
std::basic_ostream< CharT, Traits > &
operator<<(std::basic_ostream< CharT, Traits > & os,
  const uniform_real_distribution< RealType > & d);
```

此函数将 uniform_real_distribution 流式传输到 std::basic_ostream。

函数参数

- **os**：要流输出到的 basic_ostream
- **d**：要流输出的 uniform_real_distribution

返回值

os

飞曦信息 MetaX Confidential 2024-11-18 15:00:00

2.8.2.15 thrust::random::operator>>

```
template <typename RealType,
  typename CharT,
  typename Traits>
std::basic_istream< CharT, Traits > &
operator>>(std::basic_istream< CharT, Traits > & is,
  uniform_real_distribution< RealType > & d);
```

此函数从 std::basic_istream 流式传输 uniform_real_distribution。

函数参数

- **is**：要从中进行流式传输的 basic_istream
- **d**：要流输入的 uniform_real_distribution

返回值

is

2.8.3 随机数引擎适配器类模板（Random Number Engine Adaptor Class Template）

随机数引擎适配器类模板

```
/* discard_block_engine 适配现有的底层随机数引擎，并通过舍弃其底层引擎返回的一些值来产生随机值。
复合引擎的每个循环以返回底层引擎连续产生的 r 值开始，并以舍弃 p-r 这样的值结束。
引擎的状态是其底层引擎的状态，跟随着自当前周期开始以来对 operator() 的调用次数。 */
template <typename Engine,
  size_t p,
  size_t r>
class thrust::random::discard_block_engine;

/* xor_combine_engine 适配两个现有的底层随机数引擎，并通过合并每个引擎产生的值来产生随机值。 */
template <typename Engine1,
  size_t s1,
  typename Engine2,
  size_t s2 = 0u>
class thrust::random::xor_combine_engine;

template <typename Engine,
  size_t p,
  size_t r>
__host__ __device__ bool
thrust::random::operator==(const discard_block_engine< Engine, p, r > &
↪lhs,
  const discard_block_engine< Engine, p, r > & rhs);

template <typename Engine,
  size_t p,
  size_t r>
__host__ __device__ bool
thrust::random::operator!=(const discard_block_engine< Engine, p, r > &
↪lhs,
  const discard_block_engine< Engine, p, r > & rhs);
```

(下页继续)

飞猪信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
template <typename Engine,
  size_t p,
  size_t r,
  typename CharT,
  typename Traits>
std::basic_ostream< CharT, Traits > &
thrust::random::operator<<(std::basic_ostream< CharT, Traits > & os,
  const discard_block_engine< Engine, p, r > & e);

template <typename Engine,
  size_t p,
  size_t r,
  typename CharT,
  typename Traits>
std::basic_istream< CharT, Traits > &
thrust::random::operator>>(std::basic_istream< CharT, Traits > & is,
  discard_block_engine< Engine, p, r > & e);

template <typename Engine1_,
  size_t s1_,
  typename Engine2_,
  size_t s2_>
__host__ __device__ bool
thrust::random::operator==(const xor_combine_engine< Engine1_, s1_,
 ↪Engine2_, s2_ > & lhs,
  const xor_combine_engine< Engine1_, s1_, Engine2_, s2_ > & rhs);

template <typename Engine1_,
  size_t s1_,
  typename Engine2_,
  size_t s2_>
__host__ __device__ bool
thrust::random::operator!=(const xor_combine_engine< Engine1_, s1_,
 ↪Engine2_, s2_ > & lhs,
  const xor_combine_engine< Engine1_, s1_, Engine2_, s2_ > & rhs);

template <typename Engine1_,
  size_t s1_,
  typename Engine2_,
  size_t s2_,
  typename CharT,
  typename Traits>
std::basic_ostream< CharT, Traits > &
thrust::random::operator<<(std::basic_ostream< CharT, Traits > & os,
  const xor_combine_engine< Engine1_, s1_, Engine2_, s2_ > & e);

template <typename Engine1_,
  size_t s1_,
  typename Engine2_,
  size_t s2_,
  typename CharT,
  typename Traits>
std::basic_istream< CharT, Traits > &
thrust::random::operator>>(std::basic_istream< CharT, Traits > & is,
  xor_combine_engine< Engine1_, s1_, Engine2_, s2_ > & e);
```

飞沐信息 MetaX Confidential 2024-11-18 15:00:00

成员类

2.8.3.1 thrust::random::discard_block_engine

discard_block_engine 适配现有的底层随机数引擎，并通过舍弃其底层引擎返回的一些值来产生随机值。复合引擎的每个循环以返回底层引擎连续产生的 r 值开始，并以舍弃 p-r 这样的值结束。引擎的状态是其底层引擎的状态，跟随着自当前周期开始以来对 operator() 的调用次数。

2.8.3.2 thrust::random::xor_combine_engine

xor_combine_engine 适配两个现有的底层随机数引擎，并通过合并每个引擎产生的值来产生随机值。

函数

2.8.3.3 thrust::random::operator==

```
template <typename Engine,
  size_t p,
  size_t r>
__host__ __device__ bool
operator==(const discard_block_engine< Engine, p, r > & lhs,
  const discard_block_engine< Engine, p, r > & rhs);
```

此函数确认两个 discard_block_engines 是否相等。

函数参数

- **lhs**：待测试的第一个 discard_block_engine
- **rhs**：待测试的第二个 discard_block_engine

返回值

如果 lhs 等于 rhs，返回 true；否则，返回 false

2.8.3.4 thrust::random::operator!=

```
template <typename Engine,
  size_t p,
  size_t r>
__host__ __device__ bool
operator!=(const discard_block_engine< Engine, p, r > & lhs,
  const discard_block_engine< Engine, p, r > & rhs);
```

此函数确认两个 discard_block_engines 的不相等性。

函数参数

- **lhs**：待测试的第一个 discard_block_engine
- **rhs**：待测试的第二个 discard_block_engine

返回值

如果 lhs 不等于 rhs，返回 true；否则，返回 false

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

2.8.3.5 thrust::random::operator<<

```
template <typename Engine,
  size_t p,
  size_t r,
  typename CharT,
  typename Traits>
std::basic_ostream< CharT, Traits > &
operator<<(std::basic_ostream< CharT, Traits > & os,
  const discard_block_engine< Engine, p, r > & e);
```

此函数将 discard_block_engine 流式传输到 std::basic_ostream。

函数参数

- **os**：要流输出到的 basic_ostream
- **e**：要流输出的 discard_block_engine

返回值

os

2.8.3.6 thrust::random::operator>>

```
template <typename Engine,
  size_t p,
  size_t r,
  typename CharT,
  typename Traits>
std::basic_istream< CharT, Traits > &
operator>>(std::basic_istream< CharT, Traits > & is,
  discard_block_engine< Engine, p, r > & e);
```

此函数从 std::basic_istream 流式传输 discard_block_engine。

函数参数

- **is**：要从中进行流式传输的 basic_istream
- **e**：要流输入的 discard_block_engine

返回值

is

2.8.3.7 thrust::random::operator==

```
template <typename Engine1_,
  size_t s1_,
  typename Engine2_,
  size_t s2_>
__host__ __device__ bool
operator==(const xor_combine_engine< Engine1_, s1_, Engine2_, s2_ > & lhs,
  const xor_combine_engine< Engine1_, s1_, Engine2_, s2_ > & rhs);
```

此函数确认两个 xor_combine_engines 是否相等。

函数参数

- **lhs**：待测试的第一个 xor_combine_engine

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

- **rhs**：待测试的第二个 xor_combine_engine

返回值

如果 lhs 等于 rhs，返回 true；否则，返回 false

2.8.3.8 thrust::random::operator!=

```
template <typename Engine1_,
  size_t s1_,
  typename Engine2_,
  size_t s2_>
__host__ __device__ bool
operator!=(const xor_combine_engine< Engine1_, s1_, Engine2_, s2_ > & lhs,
  const xor_combine_engine< Engine1_, s1_, Engine2_, s2_ > & rhs);
```

此函数确认两个 xor_combine_engines 的不相等性。

函数参数

- **lhs**：待测试的第一个 xor_combine_engine
- **rhs**：待测试的第二个 xor_combine_engine

返回值

如果 lhs 不等于 rhs，返回 true；否则，返回 false

2.8.3.9 thrust::random::operator<<

```
template <typename Engine1_,
  size_t s1_,
  typename Engine2_,
  size_t s2_,
  typename CharT,
  typename Traits>
std::basic_ostream< CharT, Traits > &
operator<<(std::basic_ostream< CharT, Traits > & os,
  const xor_combine_engine< Engine1_, s1_, Engine2_, s2_ > & e);
```

此函数将 xor_combine_engine 流式传输到 std::basic_ostream。

函数参数

- **os**：要流输出到的 basic_ostream
- **e**：要流输出的 xor_combine_engine

返回值

os

2.8.3.10 thrust::random::operator>>

```
template <typename Engine1_,
  size_t s1_,
  typename Engine2_,
  size_t s2_,
  typename CharT,
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
  typename Traits>
std::basic_istream< CharT, Traits > &
operator>>(std::basic_istream< CharT, Traits > & is,
  xor_combine_engine< Engine1_, s1_, Engine2_, s2_ > & e);
```

此函数从 std::basic_istream 流式传输 xor_combine_engine。

函数参数

- **is**：要从中进行流式传输的 basic_istream
- **e**：要流输入的 xor_combine_engine

返回值

is

2.8.4 随机数引擎类模板（Random Number Engine Adaptor Class Template）

随机数引擎类模板

```
/* linear_congruential_engine 随机数引擎使用线性同余随机数生成算法生成无符号整数随机
数。 */
template <typename UIntType,
UIntType a,
UIntType c,
UIntType m>
class thrust::random::linear_congruential_engine;

/* linear_feedback_shift_engine 随机数引擎使用线性反馈移位随机数生成算法生成无符号整
数随机数 */
template <typename UIntType,
size_t w,
size_t k,
size_t q,
size_t s>
class thrust::random::linear_feedback_shift_engine;

/* subtract_with_carry_engine 随机数引擎使用 Marsaglia & Zaman 的带进位减法算法
生成无符号整数随机数。 */
template <typename UIntType,
size_t w,
size_t s,
size_t r>
class thrust::random::subtract_with_carry_engine;

template <typename UIntType_,
UIntType_ a_,
UIntType_ c_,
UIntType_ m_>
__host__ __device__ bool
thrust::random::operator==(const linear_congruential_engine< UIntType_, a_,
↪ c_, m_ > & lhs,
const linear_congruential_engine< UIntType_, a_, c_, m_ > & rhs);

template <typename UIntType_,
UIntType_ a_,
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
UIntType_ c_,
UIntType_ m_>
__host__ __device__ bool
thrust::random::operator!=(const linear_congruential_engine< UIntType_, a_,
 ↪ c_, m_ > & lhs,
const linear_congruential_engine< UIntType_, a_, c_, m_ > & rhs);

template <typename UIntType_,
UIntType_ a_,
UIntType_ c_,
UIntType_ m_,
typename CharT,
typename Traits>
std::basic_ostream< CharT, Traits > &
thrust::random::operator<<(std::basic_ostream< CharT, Traits > & os,
const linear_congruential_engine< UIntType_, a_, c_, m_ > & e);

template <typename UIntType_,
UIntType_ a_,
UIntType_ c_,
UIntType_ m_,
typename CharT,
typename Traits>
std::basic_istream< CharT, Traits > &
thrust::random::operator>>(std::basic_istream< CharT, Traits > & is,
linear_congruential_engine< UIntType_, a_, c_, m_ > & e);

template <typename UIntType_,
size_t w_,
size_t k_,
size_t q_,
size_t s_>
__host__ __device__ bool
thrust::random::operator==(const linear_feedback_shift_engine< UIntType_,
 ↪w_, k_, q_, s_ > & lhs,
const linear_feedback_shift_engine< UIntType_, w_, k_, q_, s_ > & rhs);

template <typename UIntType_,
size_t w_,
size_t k_,
size_t q_,
size_t s_>
__host__ __device__ bool
thrust::random::operator!=(const linear_feedback_shift_engine< UIntType_,
 ↪w_, k_, q_, s_ > & lhs,
const linear_feedback_shift_engine< UIntType_, w_, k_, q_, s_ > & rhs);

template <typename UIntType_,
size_t w_,
size_t k_,
size_t q_,
size_t s_,
typename CharT,
typename Traits>
std::basic_ostream< CharT, Traits > &
thrust::random::operator<<(std::basic_ostream< CharT, Traits > & os,
```

(下页继续)

飞沐信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
const linear_feedback_shift_engine< UIntType_, w_, k_, q_, s_ > & e);

template <typename UIntType_,
size_t w_,
size_t k_,
size_t q_,
size_t s_,
typename CharT,
typename Traits>
std::basic_istream< CharT, Traits > &
thrust::random::operator>>(std::basic_istream< CharT, Traits > & is,
linear_feedback_shift_engine< UIntType_, w_, k_, q_, s_ > & e);

template <typename UIntType_,
size_t w_,
size_t s_,
size_t r_>
__host__ __device__ bool
thrust::random::operator==(const subtract_with_carry_engine< UIntType_, w_,
 ↪ s_, r_ > & lhs,
const subtract_with_carry_engine< UIntType_, w_, s_, r_ > & rhs);

template <typename UIntType_,
size_t w_,
size_t s_,
size_t r_>
__host__ __device__ bool
thrust::random::operator!=(const subtract_with_carry_engine< UIntType_, w_,
 ↪ s_, r_ > & lhs,
const subtract_with_carry_engine< UIntType_, w_, s_, r_ > & rhs);

template <typename UIntType_,
size_t w_,
size_t s_,
size_t r_,
typename CharT,
typename Traits>
std::basic_ostream< CharT, Traits > &
thrust::random::operator<<(std::basic_ostream< CharT, Traits > & os,
const subtract_with_carry_engine< UIntType_, w_, s_, r_ > & e);

template <typename UIntType_,
size_t w_,
size_t s_,
size_t r_,
typename CharT,
typename Traits>
std::basic_istream< CharT, Traits > &
thrust::random::operator>>(std::basic_istream< CharT, Traits > & is,
subtract_with_carry_engine< UIntType_, w_, s_, r_ > & e);
```

成员类

2.8.4.1 thrust::random::linear_congruential_engine

linear_congruential_engine

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

随机数引擎使用线性同余随机数生成算法生成无符号整数随机数。

2.8.4.2 thrust::random::linear_feedback_shift_engine

linear_feedback_shift_engine

随机数引擎使用线性反馈移位随机数生成算法生成无符号整数随机数。

2.8.4.3 thrust::random::subtract_with_carry_engine

subtract_with_carry_engine

随机数引擎使用 Marsaglia & Zaman 的带进位减法算法生成无符号整数随机数。

函数

2.8.4.4 thrust::random::operator==

```
template <typename UIntType_,
  UIntType_ a_,
  UIntType_ c_,
  UIntType_ m_>
__host__ __device__ bool
operator==(const linear_congruential_engine< UIntType_, a_, c_, m_ > & lhs,
  const linear_congruential_engine< UIntType_, a_, c_, m_ > & rhs);
```

此函数确认两个 linear_congruential_engines 是否相等。

函数参数

- **lhs**：待测试的第一个 linear_congruential_engine
- **rhs**：待测试的第二个 linear_congruential_engine

返回值

如果 lhs 等于 rhs，返回 true；否则，返回 false

2.8.4.5 thrust::random::operator!=

```
template <typename UIntType_,
  UIntType_ a_,
  UIntType_ c_,
  UIntType_ m_>
__host__ __device__ bool
operator!=(const linear_congruential_engine< UIntType_, a_, c_, m_ > & lhs,
  const linear_congruential_engine< UIntType_, a_, c_, m_ > & rhs);
```

此函数确认两个 linear_congruential_engines 的不相等性。

函数参数

- **lhs**：待测试的第一个 linear_congruential_engine
- **rhs**：待测试的第二个 linear_congruential_engine

返回值

如果 lhs 不等于 rhs，返回 true；否则，返回 false

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

2.8.4.6 thrust::random::operator<<

```
template <typename UIntType_,
  UIntType_ a_,
  UIntType_ c_,
  UIntType_ m_,
  typename CharT,
  typename Traits>
std::basic_ostream< CharT, Traits > &
operator<<(std::basic_ostream< CharT, Traits > & os,
  const linear_congruential_engine< UIntType_, a_, c_, m_ > & e);
```

此函数将 linear_congruential_engine 流式传输到 std::basic_ostream。

函数参数

- **os**：要流输出到的 basic_ostream
- **e**：要流输出的 linear_congruential_engine

返回值

os

2.8.4.7 thrust::random::operator>>

```
template <typename UIntType_,
  UIntType_ a_,
  UIntType_ c_,
  UIntType_ m_,
  typename CharT,
  typename Traits>
std::basic_istream< CharT, Traits > &
operator>>(std::basic_istream< CharT, Traits > & is,
  linear_congruential_engine< UIntType_, a_, c_, m_ > & e);
```

此函数从 std::basic_istream 流式传输 linear_congruential_engine。

函数参数

- **is**：要从中进行流式传输的 basic_istream
- **e**：要流输入的 linear_congruential_engine

返回值

is

2.8.4.8 thrust::random::operator==

```
template <typename UIntType_,
  size_t w_,
  size_t k_,
  size_t q_,
  size_t s_>
__host__ __device__ bool
operator==(const linear_feedback_shift_engine< UIntType_, w_, k_, q_, s_ >
↪& lhs,
  const linear_feedback_shift_engine< UIntType_, w_, k_, q_, s_ > & rhs);
```

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

此函数确认两个 linear_feedback_shift_engines 的相等性。

函数参数

- **lhs**：待测试的第一个 linear_feedback_shift_engine
- **rhs**：待测试的第二个 linear_feedback_shift_engine

返回值

如果 lhs 等于 rhs，返回 true；否则，返回 false

2.8.4.9 thrust::random::operator!=

```
template <typename UIntType_,
  size_t w_,
  size_t k_,
  size_t q_,
  size_t s_>
__host__ __device__ bool
operator!=(const linear_feedback_shift_engine< UIntType_, w_, k_, q_, s_ >
↪& lhs,
  const linear_feedback_shift_engine< UIntType_, w_, k_, q_, s_ > & rhs);
```

此函数确认两个 linear_feedback_shift_engines 的不相等性。

函数参数

- **lhs**：待测试的第一个 linear_feedback_shift_engine
- **rhs**：待测试的第二个 linear_feedback_shift_engine

返回值

如果 lhs 不等于 rhs，返回 true；否则，返回 false

2.8.4.10 thrust::random::operator<<

```
template <typename UIntType_,
  size_t w_,
  size_t k_,
  size_t q_,
  size_t s_,
  typename CharT,
  typename Traits>
std::basic_ostream< CharT, Traits > &
operator<<(std::basic_ostream< CharT, Traits > & os,
  const linear_feedback_shift_engine< UIntType_, w_, k_, q_, s_ > & e);
```

此函数将 linear_feedback_shift_engine 流式传输到 std::basic_ostream。

函数参数

- **os**：要流输出到的 basic_ostream
- **e**：要流输出的 linear_feedback_shift_engine

返回值

os

飞谱信息 MetaX Confidential 2024-11-18 15:00:00

2.8.4.11 thrust::random::operator>>

```
template <typename UIntType_,
  size_t w_,
  size_t k_,
  size_t q_,
  size_t s_,
  typename CharT,
  typename Traits>
std::basic_istream< CharT, Traits > &
operator>>(std::basic_istream< CharT, Traits > & is,
  linear_feedback_shift_engine< UIntType_, w_, k_, q_, s_ > & e);
```

此函数从 std::basic_istream 流式传输 linear_feedback_shift_engine。

函数参数

- **is**：要从中进行流式传输的 basic_istream
- **e**：要流输入的 linear_feedback_shift_engine

返回值

is

2.8.4.12 thrust::random::operator==

```
template <typename UIntType_,
  size_t w_,
  size_t s_,
  size_t r_>
__host__ __device__ bool
operator==(const subtract_with_carry_engine< UIntType_, w_, s_, r_ > & lhs,
  const subtract_with_carry_engine< UIntType_, w_, s_, r_ > & rhs);
```

此函数确认两个 subtract_with_carry_engines 的相等性。

函数参数

- **lhs**：待测试的第一个 subtract_with_carry_engine
- **rhs**：待测试的第二个 subtract_with_carry_engine

返回值

如果 lhs 等于 rhs，返回 true；否则，返回 false

2.8.4.13 thrust::random::operator!=

```
template <typename UIntType_,
  size_t w_,
  size_t s_,
  size_t r_>
__host__ __device__ bool
operator!=(const subtract_with_carry_engine< UIntType_, w_, s_, r_ > & lhs,
  const subtract_with_carry_engine< UIntType_, w_, s_, r_ > & rhs);
```

此函数确认两个 subtract_with_carry_engines 的不相等性。

函数参数

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

- **lhs**：待测试的第一个 subtract_with_carry_engine
- **rhs**：待测试的第二个 subtract_with_carry_engine

返回值

如果 lhs 不等于 rhs，返回 true；否则，返回 false

2.8.4.14 thrust::random::operator<<

```
template <typename UIntType_,
  size_t w_,
  size_t s_,
  size_t r_,
  typename CharT,
  typename Traits>
std::basic_ostream< CharT, Traits > &
operator<<(std::basic_ostream< CharT, Traits > & os,
  const subtract_with_carry_engine< UIntType_, w_, s_, r_ > & e);
```

此函数将 subtract_with_carry_engine 流式传输到 std::basic_ostream。

函数参数

- **os**：要流输出到的 basic_ostream
- **e**：要流输出的 subtract_with_carry_engine

返回值

os

2.8.4.15 thrust::random::operator>>

```
template <typename UIntType_,
  size_t w_,
  size_t s_,
  size_t r_,
  typename CharT,
  typename Traits>
std::basic_istream< CharT, Traits > &
operator>>(std::basic_istream< CharT, Traits > & is,
  subtract_with_carry_engine< UIntType_, w_, s_, r_ > & e);
```

此函数从 std::basic_istream 流式传输 subtract_with_carry_engine。

函数参数

- **is**：要从中进行流式传输的 basic_istream
- **e**：要流输入的 subtract_with_carry_engine

返回值

is

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

2.9 系统（System）

```
/* thrust::system 是包含特定 Thrust 后端系统的命名空间。
它还包含报告错误条件的功能，错误条件源于操作系统或某些低层接口（如 MXMACA 运行时）。
它们在一个单独的命名空间中提供，以便于导入，也在顶级 thrust 命名空间中有别名，以便于访问。
 */
namespace thrust::system { ··· }
```

2.9.1 系统

系统

thrust::cpp 是 thrust::system::cpp 的顶级别名。

thrust::system 是包含特定 Thrust 后端系统的命名空间。它还包含报告错误条件的功能，错误条件源于操作系统或某些低层接口（如 MXMACA 运行时）。它们在一个单独的命名空间中提供，以便于导入，也在顶级 thrust 命名空间中有别名，以便于访问。

thrust::system::mc，此命名空间包含分配、控制和释放内存（Thrust 的 MXMACA 后端系统可用）的功能。标识符在 thrust::system 下面一个单独的命名空间中提供，以便于导入，也在顶级 thrust::mc 命名空间中有别名，以便于访问。

thrust::omp 是 thrust::system::omp 的顶级别名。

thrust::tbb 是 thrust::system::tbb 的顶级别名。

类型

2.9.1.1 thrust::system::cpp::pointer

```
template <typename T>
using pointer = thrust::pointer< T, thrust::system::cpp::tag,
↪thrust::tagged_reference< T, thrust::system::cpp::tag > >;
```

cpp::pointer 存储指向 cpp 系统可访问内存中分配的对象的指针。当在 cpp 内存中的范围上调度算法时，此类型提供类型安全。

cpp::pointer 具有指针语义：可以使用指针算术对其进行解引用和控制。

可以通过 cpp::malloc 函数，或使用原始指针显式调用其构造函数来创建 cpp::pointer。

cpp::pointer 封装的原始指针可以通过其 get 成员函数或 raw_pointer_cast 函数获得。

备注

cpp::pointer 不是一个智能指针，释放 cpp::pointer 指向的内存需要程序员的操作

模板参数

T：指定指针数据的类型

另请参阅

- cpp::malloc
- cpp::free
- raw_pointer_cast

飞书信息 MetaX Confidential 2024-11-18 15:00:00

2.9.1.2 thrust::system::cpp::universal_pointer

```
template <typename T>
using universal_pointer = thrust::pointer< T, thrust::system::cpp::tag,
↪typename std::add_lvalue_reference< T >::type >;
```

cpp::universal_pointer 存储指向 cpp 系统和主机系统可访问内存中分配的对象的指针。

cpp::universal_pointer 具有指针语义：可以使用指针算术对其进行解引用和控制。

可以通过 cpp::universal_allocator ，或使用原始指针显式调用其构造函数来创建 cpp::universal_pointer。

cpp::universal_pointer 封装的原始指针可以通过其 get 成员函数或 raw_pointer_cast 函数获得。

备注

cpp::universal_pointer 不是一个智能指针，释放 cpp::universal_pointer 指向的内存需要程序员的操作

模板参数

T：指定指针数据的类型

另请参阅

- cpp::universal_allocator
- raw_pointer_cast

2.9.1.3 thrust::system::cpp::reference

```
template <typename T>
using reference = thrust::reference< T, thrust::system::cpp::tag >;
```

reference 是对存储在 cpp 系统可用内存中对象的包装引用。reference 是解引用 cpp::pointer 的结果类型。

模板参数

T：指定被引用对象的类型

2.9.1.4 thrust::system::omp::pointer

```
template <typename T>
using pointer = thrust::pointer< T, thrust::system::omp::tag,
↪thrust::tagged_reference< T, thrust::system::omp::tag > >;
```

omp::pointer 存储指向 omp 系统可访问内存中分配的对象的指针。当在 omp 内存中的范围上调度算法时，此类型提供类型安全。

omp::pointer 具有指针语义：可以使用指针算术对其进行解引用和控制。

可以通过 omp::malloc 函数，或使用原始指针显式调用其构造函数来创建 omp::pointer。

omp::pointer 封装的原始指针可以通过其 get 成员函数或 raw_pointer_cast 函数获得。

备注

omp::pointer 不是一个智能指针，释放 omp::pointer 指向的内存需要程序员的操作

模板参数

T：指定指针数据的类型

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

另请参阅

- omp::malloc
- omp::free
- raw_pointer_cast

2.9.1.5 thrust::system::omp::universal_pointer

```
template <typename T>
using universal_pointer = thrust::pointer< T, thrust::system::omp::tag,
↪typename std::add_lvalue_reference< T >::type >;
```

omp::universal_pointer 存储指向 omp 系统和主机系统可访问内存中分配的对象的指针。

omp::universal_pointer 具有指针语义：可以使用指针算术对其进行解引用和控制。

可以通过 omp::universal_allocator ，或使用原始指针显式调用其构造函数来创建 omp::universal_pointer。

omp::universal_pointer 封装的原始指针可以通过其 get 成员函数或 raw_pointer_cast 函数获得。

备注

omp::universal_pointer 不是一个智能指针，释放 omp::universal_pointer 指向的内存需要程序员的操作

模板参数

T：指定指针数据的类型

另请参阅

- omp::universal_allocator
- raw_pointer_cast

2.9.1.6 thrust::system::omp::reference

```
template <typename T>
using reference = thrust::tagged_reference< T, thrust::system::omp::tag >;
```

reference 是对存储在 omp 系统可用内存中对象的包装引用。reference 是解引用 omp::pointer 的结果类型。

模板参数

T：指定被引用对象的类型

2.9.1.7 thrust::system::tbb::pointer

```
template <typename T>
using pointer = thrust::pointer< T, thrust::system::tbb::tag,
↪thrust::tagged_reference< T, thrust::system::tbb::tag > >;
```

tbb::pointer 存储指向 tbb 系统可访问内存中分配的对象的指针。当在 tbb 内存中的范围上调度算法时，此类型提供类型安全。

tbb::pointer 具有指针语义：可以使用指针算术对其进行解引用和控制。

可以通过 tbb::malloc 函数，或使用原始指针显式调用其构造函数来创建 tbb::pointer。

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

tbb::pointer 封装的原始指针可以通过其 get 成员函数或 raw_pointer_cast 函数获得。

备注

tbb::pointer 不是一个智能指针，释放 tbb::pointer 指向的内存需要程序员的操作

模板参数

T：指定指针数据的类型

另请参阅

- tbb::malloc
- tbb::free
- raw_pointer_cast

2.9.1.8 thrust::system::tbb::universal_pointer

```
template <typename T>
using universal_pointer = thrust::pointer< T, thrust::system::tbb::tag,
↪typename std::add_lvalue_reference< T >::type >;
```

tbb::universal_pointer 存储指向 tbb 系统和主机系统可访问内存中分配的对象的指针。

tbb::universal_pointer 具有指针语义：可以使用指针算术对其进行解引用和控制。

可以通过 tbb::universal_allocator，或使用原始指针显式调用其构造函数来创建 tbb::universal_pointer。

tbb::universal_pointer 封装的原始指针可以通过其 get 成员函数或 raw_pointer_cast 函数获得。

备注

tbb::universal_pointer 不是一个智能指针，释放 tbb::universal_pointer 指向的内存需要程序员的操作

模板参数

T：指定指针数据的类型

另请参阅

- tbb::universal_allocator
- raw_pointer_cast

2.9.1.9 thrust::system::tbb::reference

```
template <typename T>
using reference = thrust::tagged_reference< T, thrust::system::tbb::tag >;
```

reference 是对存储在 tbb 系统可用内存中对象的包装引用。reference 是解引用 tbb::pointer 的结果类型。

模板参数

T：指定被引用对象的类型

2.9.2 系统诊断（System Diagnostics）

系统诊断

飞曦信息 MetaX Confidential 2024-11-18 15:00:00

```
namespace thrust::system::errc { … }

template <typename T>
struct
thrust::system::is_error_code_enum;

template <typename T>
struct
thrust::system::is_error_condition_enum;

struct
thrust::system::is_error_condition_enum< errc::errc_t >;
```

类 error_category 作为类型基类，用于标识特定类别错误码的源和编码。除了 C++ 国际标准中定义的错误类别之外，还可以从 error_category 派生类来支持错误类别。

类 error_code 描述了一个对象，用于保存错误码的值，例如源于操作系统或某些低层接口的错误码。

类 error_condition 描述了一个对象，用于保存标识错误条件的值。

类 system_error 描述了一个异常对象，用于报告具有关联 error_code 的错误条件。这种错误条件通常源于操作系统或某些低层接口。

成员类

2.9.2.1 thrust::system::is_error_code_enum

继承自：thrust::detail::false_type

2.9.2.2 thrust::system::is_error_condition_enum

继承自：thrust::detail::false_type

2.9.2.3 thrust::system::is_error_condition_enum< errc::errc_t >

继承自：thrust::detail::true_type

2.9.2.4 thrust::system::error_category

类 error_category 作为类型基类，用于标识特定类别错误码的源和编码。除了 C++ 国际标准中定义的错误类别之外，还可以从 error_category 派生类来支持错误类别。

2.9.2.5 thrust::system::error_code

类 error_code 描述了一个对象，用于保存错误码的值，例如源于操作系统或某些低层接口的错误码。

2.9.2.6 thrust::system::error_condition

类 error_condition 描述了一个对象，用于保存标识错误条件的值。

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

2.9.2.7 thrust::system::system_error

类 system_error 描述了一个异常对象，用于报告具有关联 error_code 的错误条件。这种错误条件通常源于操作系统或某些低层接口。

继承自：std::runtime_error

函数

2.9.2.8 thrust::system::generic_category

const error_category & generic_category(void);

备注

- 对象的 default_error_condition 和 equivalent 虚拟函数的行为应与类 error_category 的规定一致。
- 对象的 name 虚拟函数应返回一个指向字符串 “generic” 的指针。

返回值

对从类 error_category 派生的类型对象的引用。

2.9.2.9 thrust::system::system_category

const error_category & system_category(void);

如果参数 ev 对应 POSIX errno 值 posv，则函数应返回 error_condition(ev,generic_category())。否则，函数将返回 error_condition(ev,system_category())。对于任何给定的操作系统，由什么构成对应关系没有明确说明。

备注

- 对象的 equivalent 虚拟函数的行为应与类 error_category 的规定一致。
- 对象的 name 虚拟函数应返回一个指向字符串 “system” 的指针。
- 对象的 default_error_condition 虚拟函数的行为应如上所示。

返回值

对从类 error_category 派生的类型对象的引用。

2.9.2.10 thrust::system::make_error_code

error_code make_error_code(mc::errc::errc_t e);

返回值

- error_code(static_cast<int>(e), mc::error_category())
- error_code(static_cast<int>(e), generic_category())

2.9.2.11 thrust::system::operator<

```
bool
operator<(const error_code & lhs,
  const error_code & rhs);
```

返回值

lhs.category() < rhs.category() || lhs.category() == rhs.category() && lhs.value() < rhs.value()

飞腾信息 MetaX Confidential 2024-11-16 15:00:00

2.9.2.12 thrust::system::operator<<

```
template <typename charT,
  typename traits>
std::basic_ostream< charT, traits > &
operator<<(std::basic_ostream< charT, traits > & os,
  const error_code & ec);
```

Effects: os << ec.category().name() << ':' << ec.value()

2.9.2.13 thrust::system::make_error_condition

error_condition make_error_condition(mc::errc::errc_t e);

返回值

- error_condition(static_cast<int>(e), mc::error_category())
- error_condition(static_cast<int>(e), generic_category())

2.9.2.14 thrust::system::operator<

```
bool
operator<(const error_condition & lhs,
  const error_condition & rhs);
```

返回值

lhs.category() < rhs.category() || lhs.category() == rhs.category() && lhs.value() < rhs.value()

2.9.2.15 thrust::system::operator==

```
bool
operator==(const error_code & lhs,
  const error_code & rhs);
```

返回值

lhs.category() == rhs.category() && lhs.value() == rhs.value()

2.9.2.16 thrust::system::operator==

```
bool
operator==(const error_code & lhs,
  const error_condition & rhs);
```

返回值

lhs.category().equivalent(lhs.value(), rhs) || rhs.category().equivalent(lhs,rhs.value())

2.9.2.17 thrust::system::operator==

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

```
bool
operator==(const error_condition & lhs,
  const error_code & rhs);
```

返回值

rhs.category().equivalent(lhs.value(), lhs) || lhs.category().equivalent(rhs, lhs.value())

2.9.2.18 thrust::system::operator==

```
bool
operator==(const error_condition & lhs,
  const error_condition & rhs);
```

返回值

lhs.category() == rhs.category() && lhs.value() == rhs.value()

2.9.2.19 thrust::system::operator!=

```
bool
operator!=(const error_code & lhs,
  const error_code & rhs);
```

返回值

!(lhs == rhs)

2.9.2.20 thrust::system::operator!=

```
bool
operator!=(const error_code & lhs,
  const error_condition & rhs);
```

返回值

!(lhs == rhs)

2.9.2.21 thrust::system::operator!=

```
bool
operator!=(const error_condition & lhs,
  const error_code & rhs);
```

返回值

!(lhs == rhs)

2.9.2.22 thrust::system::operator!=

```
bool
operator!=(const error_condition & lhs,
  const error_condition & rhs);
```

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

返回值

!(lhs == rhs)

2.9.2.23 thrust::system::system_error::system_error

```
system_error(error_code ec,   const std::string & what_arg);
```

构建类 system_error 的一个对象。

函数参数

- **ec**：code() 返回的值
- **what_arg**：what() 返回结果中包含的字符串

后置条件

- code() == ec
- std::string(what()).find(what_arg) != string::npos

2.9.2.24 thrust::system::system_error::system_error

```
system_error(error_code ec,   const char * what_arg);
```

构建类 system_error 的一个对象。

函数参数

- **ec**：code() 返回的值
- **what_arg**：what() 返回结果中包含的字符串

后置条件

- code() == ec
- std::string(what()).find(what_arg) != string::npos

2.9.2.25 thrust::system::system_error::system_error

```
system_error(error_code ec);
```

构建类 system_error 的一个对象。

函数参数

ec：code() 返回的值

后置条件

code() == ec

2.9.2.26 thrust::system::system_error::system_error

```
system_error(int ev,   const error_category & ecat,   const std::string &
↪what_arg);
```

构建类 system_error 的一个对象。

函数参数

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

- **ev**：用于创建 error_code 的错误值
- **ecat**：用于创建 error_code 的 error_category
- **what_arg**：what() 返回结果中包含的字符串

后置条件

- code() == error_code(ev, ecat)
- std::string(what()).find(what_arg) != string::npos

2.9.2.27 thrust::system::system_error::system_error

```
system_error(int ev,   const error_category & ecat,   const char * what_
↪arg);
```

构建类 system_error 的一个对象。

函数参数

- **ev**：用于创建 error_code 的错误值
- **ecat**：用于创建 error_code 的 error_category
- **what_arg**：what() 返回结果中包含的字符串

后置条件

- code() == error_code(ev, ecat)
- std::string(what()).find(what_arg) != string::npos

2.9.2.28 thrust::system::system_error::system_error

```
system_error(int ev,   const error_category & ecat);
```

构建类 system_error 的一个对象。

函数参数

- **ev**：用于创建 error_code 的错误值
- **ecat**：用于创建 error_code 的 error_category

后置条件

code() == error_code(ev, ecat)

2.9.2.29 thrust::system::system_error::~system_error

```
virtual ~system_error(void);
```

析构函数不会抛出异常（throw）。

2.9.2.30 thrust::system::system_error::code

```
const error_code & code(void) const;
```

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

返回对错误进行编码的对象。

返回值

从构造函数返回 ec 或 error_code(ev, ecat)，视情况而定

2.9.2.31 thrust::system::system_error::what

```
const char * what(void) const;
```

返回一个指示错误性质的可读字符串。

返回值

包含 code().message() 和构造函数中提供的参数的字符串

2.10 工具库（Utility）

2.10.1 Pair

Pair

```
template <typename T1,
typename T2>
struct
thrust::pair;

template <size_t N,
  class T>
struct
thrust::tuple_element;

template <typename Pair>
struct
thrust::tuple_size;

template <typename T1,
  typename T2>
__host__ __device__ bool
thrust::operator==(const pair< T1, T2 > & x,
  const pair< T1, T2 > & y);

template <typename T1,
  typename T2>
__host__ __device__ bool
thrust::operator<(const pair< T1, T2 > & x,
  const pair< T1, T2 > & y);

template <typename T1,
  typename T2>
__host__ __device__ bool
thrust::operator!=(const pair< T1, T2 > & x,
  const pair< T1, T2 > & y);

template <typename T1,
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
  typename T2>
__host__ __device__ bool
thrust::operator>(const pair< T1, T2 > & x,
  const pair< T1, T2 > & y);

template <typename T1,
  typename T2>
__host__ __device__ bool
thrust::operator<=(const pair< T1, T2 > & x,
  const pair< T1, T2 > & y);

template <typename T1,
  typename T2>
__host__ __device__ bool
thrust::operator>=(const pair< T1, T2 > & x,
  const pair< T1, T2 > & y);

template <typename T1,
  typename T2>
__host__ __device__ void
thrust::swap(pair< T1, T2 > & x,
  pair< T1, T2 > & y);

template <typename T1,
  typename T2>
__host__ __device__ pair< T1, T2 >
thrust::make_pair(T1 x,
  T2 y);
```

成员类

2.10.1.1 thrust::pair

2.10.1.2 thrust::tuple_element

2.10.1.3 thrust::tuple_size

函数

2.10.1.4 thrust::operator==

```
template <typename T1,
  typename T2>
__host__ __device__ bool
operator==(const pair< T1, T2 > & x,
  const pair< T1, T2 > & y);
```

此运算符测试两个 pairs 是否相等。

模板参数

- **T1**：是 Equality Comparable 的模型
- **T2**：是 Equality Comparable 的模型

函数参数

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

- **x**：要比较的第一个 pair
- **y**：要比较的第二个 pair

返回值

当且仅当 x.first == y.first && x.second == y.second，返回 true

2.10.1.5 thrust::operator<

```
template <typename T1,
  typename T2>
__host__ __device__ bool
operator<(const pair< T1, T2 > & x,
  const pair< T1, T2 > & y);
```

此运算符测试两个 pairs 是否为升序。

模板参数

- **T1**：是 LessThan Comparable 的模型
- **T2**：是 LessThan Comparable 的模型

函数参数

- **x**：要比较的第一个 pair
- **y**：要比较的第二个 pair

返回值

当且仅当 x.first < y.first || (!(y.first < x.first) && x.second < y.second)，返回 true

2.10.1.6 thrust::operator!=

```
template <typename T1,
  typename T2>
__host__ __device__ bool
operator!=(const pair< T1, T2 > & x,
  const pair< T1, T2 > & y);
```

此运算符测试两个 pairs 是否不等。

模板参数

- **T1**：是 Equality Comparable 的模型
- **T2**：是 Equality Comparable 的模型

函数参数

- **x**：待比较的第一个 pair
- **y**：待比较的第二个 pair

返回值

当且仅当 !(x == y)，返回 true

2.10.1.7 thrust::operator>

飞络信息 MetaX Confidential 2024-11-18 15:00:00

```
template <typename T1,
  typename T2>
__host__ __device__ bool
operator>(const pair< T1, T2 > & x,
  const pair< T1, T2 > & y);
```

此运算符测试两个 pairs 是否为降序。

模板参数

- **T1**：是 LessThan Comparable 的模型
- **T2**：是 LessThan Comparable 的模型

函数参数

- **x**：待比较的第一个 pair
- **y**：待比较的第二个 pair

返回值

当且仅当 y < x，返回 true

2.10.1.8 thrust::operator<=

```
template <typename T1,
  typename T2>
__host__ __device__ bool
operator<=(const pair< T1, T2 > & x,
  const pair< T1, T2 > & y);
```

此运算符测试两个 pairs 是否为升序或相等。

模板参数

- **T1**：是 LessThan Comparable 的模型
- **T2**：是 LessThan Comparable 的模型

函数参数

- **x**：待比较的第一个 pair
- **y**：待比较的第二个 pair

返回值

当且仅当 !(y < x)，返回 true

2.10.1.9 thrust::operator>=

```
template <typename T1,
  typename T2>
__host__ __device__ bool
operator>=(const pair< T1, T2 > & x,
  const pair< T1, T2 > & y);
```

此运算符测试两个 pairs 是否为降序或相等。

模板参数

- **T1**：是 LessThan Comparable 的模型

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

- **T2**：是 LessThan Comparable 的模型

函数参数

- **x**：待比较的第一个 pair
- **y**：待比较的第二个 pair

返回值

当且仅当 !(x < y)，返回 true

2.10.1.10 thrust::swap

```
template <typename T1,
  typename T2>
__host__ __device__ void
swap(pair< T1, T2 > & x,
  pair< T1, T2 > & y);
```

swap 交换两个 pair 的内容。

函数参数

- **x**：要交换的第一个 pair
- **y**：要交换的第二个 pair

2.10.1.11 thrust::make_pair

```
template <typename T1,
  typename T2>
__host__ __device__ pair< T1, T2 >
make_pair(T1 x,
  T2 y);
```

此便捷函数从两个对象创建一个 pair。

模板参数

- **T1**：T1 的类型没有要求
- **T2**：T2 的类型没有要求

函数参数

- **x**：要复制的第一个对象
- **y**：要复制的第二个对象

返回值

一个从 a 和 b 复制而新创建的 pair

2.10.2 交换（Swap）

交换

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

```
template <typename Assignable1,
  typename Assignable2>
__host__ __device__ void
thrust::swap(Assignable1 & a,
  Assignable2 & b);
```

函数

2.10.2.1 thrust::swap

```
template <typename Assignable1,
  typename Assignable2>
__host__ __device__ void
swap(Assignable1 & a,
  Assignable2 & b);
```

swap 将 a 的内容赋给 b ，并将 b 的内容赋给 a。被其他众多算法用作原语操作。

以下代码片段演示了如何使用 swap 交换两个变量的内容：

```
#include <thrust/swap.h>
...
int x = 1;
int y = 2;
thrust::swap(x,h);

// x == 2, y == 1
```

模板参数

Assignable：是 Assignable 的模型

函数参数

- **a**：第一个感兴趣的值。完成后，b 的值将在此处返回
- **b**：第二个感兴趣的值。完成后，a 的值将在此处返回

2.10.3 元组（Tuple）

元组

```
template <size_t N,
  class T> struct thrust::tuple_element;

template <typename Pair>
struct
thrust::tuple_size;
```

是一个类模板，最多可以使用 10 个参数进行实例化。每个模板参数都指定了 tuple 中的元素类型。因此，元组是异构的，是大小固定的值集合。 带两个参数的 tuple 实例类似于带相同两个参数的 pair 实例。可以使用 get 函数访问 tuple 的各个元素。

成员类

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

2.10.3.1 thrust::tuple_element

2.10.3.2 thrust::tuple_size

2.10.3.3 thrust::tuple

tuple 是一个类模板，最多可以使用 10 个参数进行实例化。每个模板参数都指定了 tuple 中的元素类型。因此，元组是异构的、固定大小的值集合。带两个参数的 tuple 实例类似于带相同两个参数的 pair 实例。可以使用 get 函数访问 tuple 的各个元素。

函数

2.10.3.4 thrust::get

```
template <int N,
  class HT,
  class TT>
__host__ __device__ access_traits< typenametuple_element< N, detail::cons<
↪HT, TT > >::type >::non_const_type
get(detail::cons< HT, TT > & t);
```

get 函数返回对感兴趣的 tuple 元素的引用。以下代码片段演示了如何使用 get 打印 tuple 元素的值：

```
#include <thrust/tuple.h>
#include <iostream>
...
thrust::tuple<int, const char *> t(13, "thrust");

std::cout << "The 1st value of t is " << thrust::get<0>(t) << std::endl;
```

模板参数

n：感兴趣元素的索引

函数参数

T：对感兴趣的 tuple 的引用

返回值

对 T 的第 N 个元素的引用

另请参阅

- pair
- tuple

2.10.3.5 thrust::get

```
template <int N,
  class HT,
  class TT>
__host__ __device__ access_traits< typenametuple_element< N, detail::cons<
↪HT, TT > >::type >::const_type
get(const detail::cons< HT, TT > & t);
```

get 函数返回对感兴趣的 tuple 元素的 const 引用。

以下代码片段演示了如何使用 get 打印 tuple 元素的值：

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

```
#include <thrust/tuple.h>
#include <iostream>
...
thrust::tuple<int, const char *> t(13, "thrust");

std::cout << "The 1st value of t is " << thrust::get<0>(t) << std::endl;
```

模板参数

n：感兴趣元素的索引

函数参数

T: 对感兴趣的 tuple 的引用

返回值

对 T 的第 N 个元素的 const 引用

另请参阅

- pair
- tuple

2.10.3.6 thrust::make_tuple

```
template <class T0> __host__ __device__ detail::make_tuple_mapper< T0 >
↪::type make_tuple(const T0 & t0);
```

此版本的 make_tuple 从单个对象创建一个新的 tuple。

函数参数

t0：要从中复制的对象

返回值

具有单个成员的 tuple 对象，该成员是 t0 的副本

2.10.3.7 thrust::make_tuple

```
template <class T0,
  class T1>
__host__ __device__ detail::make_tuple_mapper< T0, T1 >::type
make_tuple(const T0 & t0,   const T1 & t1);
```

此版本的 make_tuple 从两个对象创建一个新的 tuple。

备注

make_tuple 有 10 种变体，为简洁起见，此处省略了其余变体

函数参数

- **t0**：要从中复制的第一个对象
- **t1**：要从中复制的第二个对象

返回值

具有两个成员的 tuple 对象，这两个成员是 t0 和 t1 的副本

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

2.10.3.8 thrust::tie

```
template <typename T0> __host__ __device__ tuple< T0 & > tie(T0 & t0);
```

这个版本的 tie 创建了一个新的 tuple ，其单个元素引用了该函数的参数。

函数参数

t0： 要引用的对象

返回值

具有一个成员的 tuple 对象，该成员引用了 t0

2.10.3.9 thrust::tie

```
template <typename T0,
  typename T1>
__host__ __device__ tuple< T0 &, T1 & >
tie(T0 & t0,
  T1 & t1);
```

这个版本的 tie 创建了一个新的 tuple ，引用了该函数的参数。

备注

tie 有 10 种变体，为简洁起见，此处省略了其余变体

函数参数

- **t0**：要引用的第一个对象
- **t1**：要引用的第二个对象

返回值

具有两个成员的 tuple 对象，引用了 t0 和 t1

2.10.3.10 thrust::swap

```
template <typename T0,
  typename T1,
  typename T2,
  typename T3,
  typename T4,
  typename T5,
  typename T6,
  typename T7,
  typename T8,
  typename T9,
  typename U0,
  typename U1,
  typename U2,
  typename U3,
  typename U4,
  typename U5,
  typename U6,
  typename U7,
  typename U8,
  typename U9>
```

(下页继续)

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

(续上页)

```
__host__ __device__ void
swap(tuple< T0, T1, T2, T3, T4, T5, T6, T7, T8, T9 > & x,
  tuple< U0, U1, U2, U3, U4, U5, U6, U7, U8, U9 > & y);
```

swap 交换两个 tuple 的内容。

函数参数

- **x**：要交换的第一个 tuple
- **y**：要交换的第二个 tuple

2.10.4 Type Traits

Type Traits

Customization point，可以自定义它来表示迭代器类型 Iterator 满足 ContiguousIterator，即它指向内存中连续的元素。

成员类

2.10.4.1 thrust::proclaim_contiguous_iterator

Customization point，可以自定义它来表示迭代器类型 Iterator 满足 ContiguousIterator，即它指向内存中连续的元素。

继承自：false_type

2.10.4.2 thrust::conjunction_value

std::integral_constant 值为 (…&& Bs)

2.10.4.3 thrust::disjunction_value

std::integral_constant 值为 (…|| Bs)

2.10.4.4 thrust::negation_value

std::integral_constant 值为 !Bs

2.10.4.5 thrust::remove_cvref

UnaryTypeTrait，从 T 中删除 const-volatile qualifiers 和 references。等价于 remove_cv_t<remove_reference_t<T>>。

2.10.4.6 thrust::voider

类型

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

2.10.4.7 thrust::integer_sequence

```
template <typename T,
  T... Is>
using
integer_sequence = std::integer_sequence< T, Is... >;
```

integral constants 的编译时序列，其类型 T 的值为 Is…。

另请参阅

- integral constants
- index_sequence
- make_integer_sequence
- make_reversed_integer_sequence
- make_index_sequence
- make_reversed_index_sequence
- integer_sequence_push_front
- integer_sequence_push_back
- std::integer_sequence

2.10.4.8 thrust::index_sequence

```
template <std::size_t... Is> using index_sequence = std::index_sequence<
↪Is... >;
```

值为 Is…的类型 std::size_t 的编译时序列。

另请参阅

- integer_sequence
- make_integer_sequence
- make_reversed_integer_sequence
- make_index_sequence
- make_reversed_index_sequence
- integer_sequence_push_front
- integer_sequence_push_back
- std::index_sequence

2.10.4.9 thrust::make_integer_sequence

```
template <typename T,
  std::size_t N>
using
make_integer_sequence = std::make_integer_sequence< T, N >;
```

通过类型 T 的元素 0, 1, 2, …, N - 1 创建新的 integer_sequence。

另请参阅

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

- integer_sequence
- index_sequence
- make_reversed_integer_sequence
- make_index_sequence
- make_reversed_index_sequence
- std::make_integer_sequence

2.10.4.10 thrust::make_index_sequence

```
template <std::size_t N> using make_index_sequence = std::make_index_
↪sequence< N >;
```

通过类型 std::size_t 的元素 0, 1, 2, ⋯, N - 1 创建新的 integer_sequence。

另请参阅

- integer_sequence
- index_sequence
- make_integer_sequence
- make_reversed_integer_sequence
- make_reversed_index_sequence
- std::make_index_sequence

2.10.4.11 thrust::make_reversed_integer_sequence

```
template <typename T,
  std::size_t N>
using
make_reversed_integer_sequence = typename detail::make_reversed_integer_
↪sequence_impl< T, N >::type;
```

通过元素 N - 1, N - 2, N - 3, ⋯, 0 创建新的 integer_sequence。

另请参阅

- integer_sequence
- index_sequence
- make_integer_sequence
- make_index_sequence
- make_reversed_index_sequence

2.10.4.12 thrust::make_reversed_index_sequence

```
template <std::size_t N> using make_reversed_index_sequence = make_
↪reversed_integer_sequence< std::size_t, N >;
```

通过元素 N - 1, N - 2, N - 3, ⋯, 0 创建新的 index_sequence。

另请参阅

飞曦信息 MetaX Confidential 2024-11-18 15:00:00

- integer_sequence
- index_sequence
- make_integer_sequence
- make_reversed_integer_sequence
- make_reversed_index_sequence

2.10.4.13 thrust::integer_sequence_push_front

```
template <typename T,
  T Value,
  typename Sequence>
using
integer_sequence_push_front = typename detail::integer_sequence_push_front_
↪impl< T, Value, Sequence >::type;
```

在 integer_sequence 的前面添加一个新元素。

另请参阅

- integer_sequence
- index_sequence
- make_integer_sequence
- make_index_sequence

2.10.4.14 thrust::integer_sequence_push_back

```
template <typename T,
  T Value,
  typename Sequence>
using
integer_sequence_push_back = typename detail::integer_sequence_push_back_
↪impl< T, Value, Sequence >::type;
```

在 integer_sequence 的后面添加一个新元素。

另请参阅

- integer_sequence
- index_sequence
- make_integer_sequence
- make_index_sequence

2.10.4.15 thrust::is_contiguous_iterator

```
template <typename Iterator> using is_contiguous_iterator = detail::is_
↪contiguous_iterator_impl< Iterator >;
```

如果 Iterator 满足 ContiguousIterator，即它指向内存中的连续元素，UnaryTypeTrait 返回 true_type；否则返回 false_type。

另请参阅

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

- is_contiguous_iterator_v
- proclaim_contiguous_iterator
- THRUST_PROCLAIM_CONTIGUOUS_ITERATOR

2.10.4.16 thrust::is_execution_policy

```
template <typename T> using is_execution_policy = detail::is_base_of<
↪detail::execution_policy_marker, T >;
```

如果 T 是 ExecutionPolicy，UnaryTypeTrait 返回 true_type；否则返回 false_type。

2.10.4.17 thrust::is_operator_less_function_object

```
template <typename T> using is_operator_less_function_object = detail::is_
↪operator_less_function_object_impl< T >;
```

如果 T 是一个 BinaryPredicate ，等价于 operator<，UnaryTypeTrait 返回 true_type；否则返回 false_type。

另请参阅

- is_operator_less_function_object_v
- is_operator_greater_function_object
- is_operator_less_or_greater_function_object
- is_operator_plus_function_object

2.10.4.18 thrust::is_operator_greater_function_object

```
template <typename T> using is_operator_greater_function_object =
↪detail::is_operator_greater_function_object_impl< T >;
```

如果 T 是一个 BinaryPredicate ，等价于 operator>，UnaryTypeTrait 返回 true_type；否则返回 false_type。

另请参阅

- is_operator_greater_function_object_v
- is_operator_less_function_object
- is_operator_less_or_greater_function_object
- is_operator_plus_function_object

2.10.4.19 thrust::is_operator_less_or_greater_function_object

```
template <typename T> using is_operator_less_or_greater_function_object =
↪integral_constant< bool, detail::is_operator_less_function_object_impl<
↪T >::value||detail::is_operator_greater_function_object_impl< T >::value
↪>;
```

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

如果 T 是一个 BinaryPredicate，等价于 operator< 或 operator>，UnaryTypeTrait 返回 true_type；否则返回 false_type。

另请参阅

- is_operator_less_or_greater_function_object_v
- is_operator_less_function_object
- is_operator_greater_function_object
- is_operator_plus_function_object

2.10.4.20 thrust::is_operator_plus_function_object

```
template <typename T> using is_operator_plus_function_object = detail::is_
↪operator_plus_function_object_impl< T >;
```

如果 T 是一个 FunctionObject，等价于 operator+，UnaryTypeTrait 返回 true_type；否则返回 false_type。

另请参阅

- is_operator_plus_function_object_v
- is_operator_less_function_object
- is_operator_greater_function_object
- is_operator_less_or_greater_function_object

2.10.4.21 thrust::conjunction

```
template <typename... Ts> using conjunction = std::conjunction< Ts... >;
```

std::integral_constant 值为 (…&& Ts::value)

另请参阅

- conjunction_v
- conjunction_value
- std::conjunction

2.10.4.22 thrust::disjunction

```
template <typename... Ts> using disjunction = std::disjunction< Ts... >;
```

std::integral_constant 值为 (…|| Ts::value)

另请参阅

- disjunction_v
- disjunction_value
- std::disjunction

飞猪信息 MetaX Confidential 2024-11-18 15:00:00

2.10.4.23 thrust::negation

```
template <typename T> using negation = std::negation< T >;
```

std::integral_constant 值为 !Ts::value

另请参阅

- negation_v
- negation_value
- std::negation

2.10.4.24 thrust::remove_cvref_t

```
template <typename T> using remove_cvref_t = typename remove_cvref< T >
↪::type;
```

类型别名，从 T 中删除 const-volatile qualifiers 和 references。等价于 remove_cv_t<remove_reference_t<T>>。

另请参阅

- std::remove_cvref
- std::remove_cv
- std::remove_const
- std::remove_volatile
- std::remove_reference

变量

2.10.4.25 thrust::is_contiguous_iterator_v

```
template <typename Iterator> constexpr bool is_contiguous_iterator_v = is_
↪contiguous_iterator<Iterator>::value;
```

如果 Iterator 满足 ContiguousIterator ，即它指向内存中的连续元素，constexpr bool 为 true；否则为 false。

另请参阅

- is_contiguous_iterator
- proclaim_contiguous_iterator
- THRUST_PROCLAIM_CONTIGUOUS_ITERATOR

2.10.4.26 thrust::is_execution_policy_v

```
template <typename T> constexpr bool is_execution_policy_v = is_execution_
↪policy<T>::value;
```

如果 T 是 ExecutionPolicy，constexpr bool 为 true；否则为 false。

飞端信息 MetaX Confidential 2024-11-18 15:00:00

2.10.4.27 thrust::is_operator_less_function_object_v

```
template <typename T> constexpr bool is_operator_less_function_object_v =
↪is_operator_less_function_object<T>::value;
```

如果 T 是一个 BinaryPredicate ，等价于 operator<，constexpr bool 为 true；否则为 false。

另请参阅

- is_operator_less_function_object
- is_operator_greater_function_object
- is_operator_less_or_greater_function_object
- is_operator_plus_function_object

2.10.4.28 thrust::is_operator_greater_function_object_v

```
template <typename T> constexpr bool is_operator_greater_function_object_v
↪= is_operator_greater_function_object<T>::value;
```

如果 T 是一个 BinaryPredicate ，等价于 operator>，constexpr bool 为 true；否则为 false。

另请参阅

- is_operator_greater_function_object
- is_operator_less_function_object
- is_operator_less_or_greater_function_object
- is_operator_plus_function_object

2.10.4.29 thrust::is_operator_less_or_greater_function_object_v

```
template <typename T> constexpr bool is_operator_less_or_greater_function_
↪object_v = is_operator_less_or_greater_function_object<T>::value;
```

如果 T 是一个 BinaryPredicate ，等价于 operator> 或 operator< ，constexpr bool 为 true；否则为 false。

另请参阅

- is_operator_less_or_greater_function_object
- is_operator_less_function_object
- is_operator_greater_function_object
- is_operator_plus_function_object

2.10.4.30 thrust::is_operator_plus_function_object_v

```
template <typename T> constexpr bool is_operator_plus_function_object_v =
↪is_operator_plus_function_object<T>::value;
```

如果 T 是一个 FunctionObject ，等价于 operator<，constexpr bool 为 true；否则为 false。

另请参阅

- is_operator_plus_function_object

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

- is_operator_less_function_object
- is_operator_greater_function_object
- is_operator_less_or_greater_function_object

2.10.4.31 thrust::conjunction_v

```
template <typename... Ts> constexpr bool conjunction_v = conjunction<Ts...>
↪::value;
```

constexpr bool 值为 (…&& Ts::value)

另请参阅

- conjunction
- conjunction_value
- std::conjunction

2.10.4.32 thrust::disjunction_v

```
template <typename... Ts> constexpr bool disjunction_v = disjunction<Ts...>
↪::value;
```

constexpr bool 值为 (…|| Ts::value)

另请参阅

- disjunction
- disjunction_value
- std::disjunction

2.10.4.33 thrust::negation_v

```
template <typename T> constexpr bool negation_v = negation<T>::value;
```

constexpr bool 值为 !Ts::value

另请参阅

- negation
- negation_value
- std::negation

2.10.4.34 thrust::conjunction_value_v

```
template <bool... Bs> constexpr bool conjunction_value_v = conjunction_
↪value<Bs...>::value;
```

constexpr bool 值为 (…&& Bs)

另请参阅

- conjunction_value

飞腾信息 MetaX Confidential 2024-11-18 15:00:00

- conjunction
- std::conjunction

2.10.4.35 thrust::disjunction_value_v

```
template <bool... Bs> constexpr bool disjunction_value_v = disjunction_
↪value<Bs...>::value;
```

constexpr bool 值为 (…|| Bs)

另请参阅

- disjunction_value
- disjunction
- std::disjunction

2.10.4.36 thrust::negation_value_v

```
template <bool B> constexpr bool negation_value_v = negation_value<B>
↪::value;
```

constexpr bool 值为 !Ts::value

另请参阅

- negation_value
- negation
- std::negation

宏（Macros）

THRUST_PROCLAIM_CONTIGUOUS_ITERATOR

通过特化 proclaim_contiguous_iterator 声明迭代器 Iterator 是 ContiguousIterator。

另请参阅

- is_contiguous_iterator
- proclaim_contiguous_iterator

飞腾信息 MetaX Confidential 2024-11-13 15:00:00