曦云®系列通用计算 GPU

运行时 API 编程指南

CSPG-23003-020-F3_V06　　2024-09-06

沐曦专有和三级保密信息　　本文档受 NDA 管控

更新记录

版本	日期	更新说明
V06	2024-09-06	更新以下章节： 4.4 环境变量 新增以下章节： 2.7 MPS 多进程服务 3.9 MPS 多进程服务
V05	2024-06-14	更新以下章节： 4.4 环境变量 新增以下章节： 3.8.4 图编程的注意事项
V04	2024-05-15	更新以下章节： 1 概述 3.2 内存管理 3.7.2.6 私有内存
V03	2024-03-15	新增曦云®系列 GPU 产品信息
V02	2023-12-29	更新以下章节： 4.4 环境变量
V01	2023-10-16	正式版本首次发布

目录

图目录

表目录

1. 概述

本文档详细描述了如何在曦云®系列的 GPU 硬件上使用 MXMACA 软件栈进行编程，旨在帮助开发人员利用曦云系列 GPU 提供的计算资源，快速构建自己的应用。

曦云系列 GPU 的整体系统架构如图 1-1 所示：

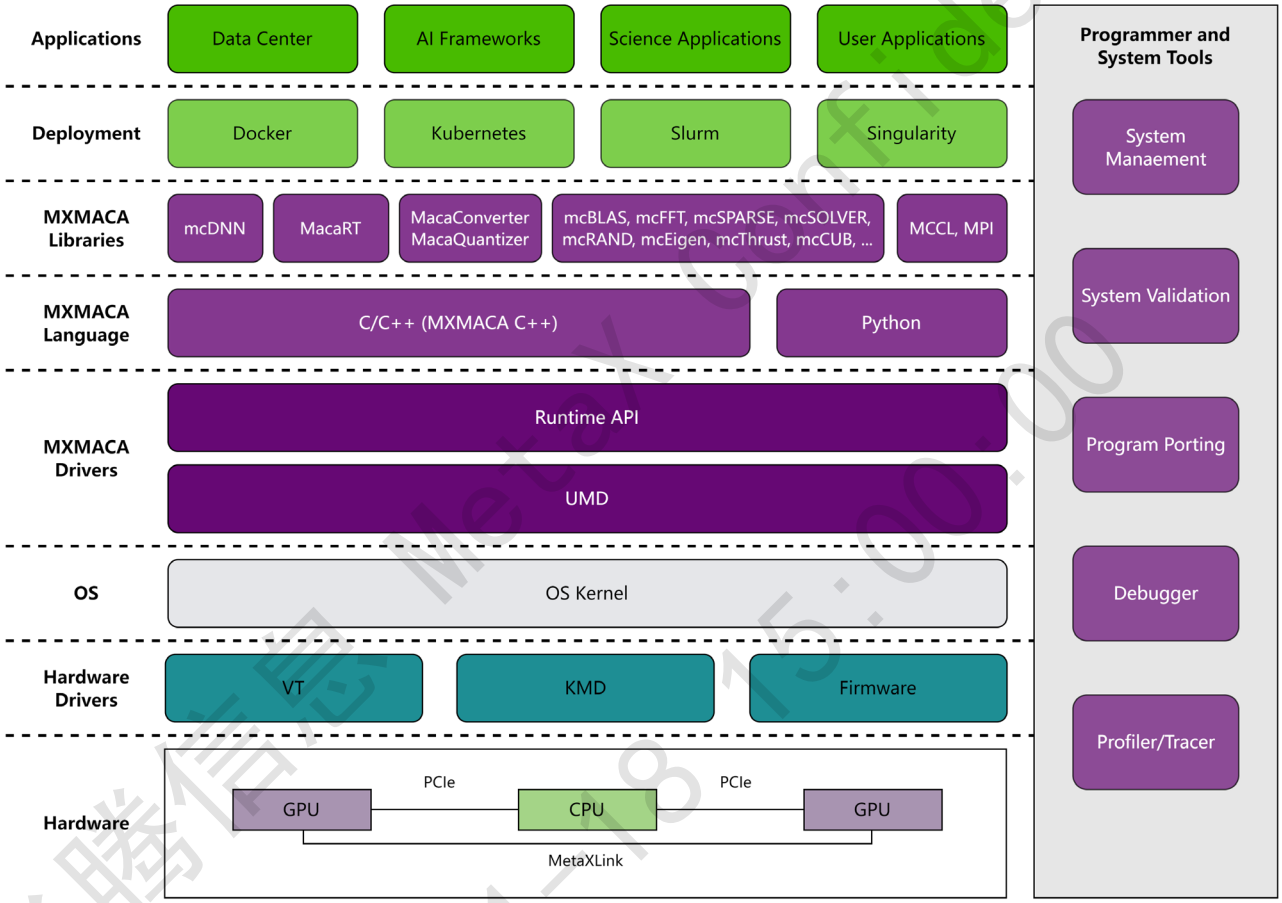

图 1-1 系统架构

MXMACA®（MetaX Advanced Compute Architecture）是采用通用并行计算架构解决复杂计算问题的异构计算平台，包含了指令集架构（ISA）、GPU 并行计算硬件引擎和 GPU 软件开发平台。它提供了规范化的编程接口，包括 MXMACA 驱动层提供的运行时 API、MXMACA 语言层提供的类 C/C++编程语言、MXMACA 库封装层提供的人工智能和计算加速库，方便用户编写 MXMACA 程序，使其在 GPU 处理器上以超高性能运行。

本文档主要介绍 MXMACA 驱动层提供的运行时 API，它提供了分配和释放设备内存、在主机内存和设备内存之间传输数据、调度和启动 GPU 内核任务、管理具有多个设备的系统等功能。

2. 编程模型

2.1 程序结构

图 2-1 展示了 MXMACA 编程模型和 MXMACA 程序结构全景图，MXMACA 程序是一个定义了上下文的宿主机程序。一个 MXMACA 上下文对象内具有两个计算设备：一个 CPU 设备和一个 GPU 设备。每个计算设备具有自己的命令队列，所以有两种命令队列：一种是面向 CPU 的乱序命令队列，另一种是面向 GPU 的有序命令队列。然后 MXMACA 宿主机程序定义一个程序对象，这个程序对象编译后将为两个 MXMACA 设备（CPU 和 GPU）生成内核。接下来 MXMACA 宿主机程序定义程序所需的内存对象，并把它们映射到内核的参数。最后，MXMACA 宿主机程序将命令放入命令队列来执行这些内核。

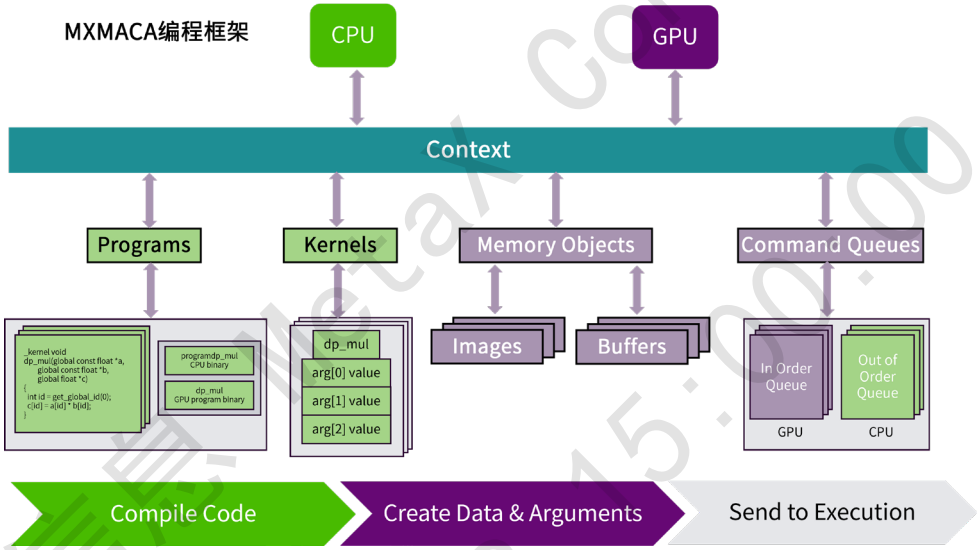

图 2-1 MXMACA 程序结构全景图

MXMACA 编程使用由 C/C++语言扩展生成的注释代码在异构计算系统中执行应用程序。一个异构环境包含多个 CPU 和 GPU，每个 CPU 和 GPU 都由一条 PCIe 总线隔开，因此需要注意区分这两个内容：

- **主机**：CPU 及其内存（主机内存）
- **设备**：GPU 及其内存（设备内存）

核函数（kernel）是 MXMACA 编程模型的重要组成部分，其代码在 GPU 上运行。多数情况下主机可以独立地对设备进行操作，而内核一旦启动，管理权限立刻返回给主机，释放 CPU 执行其他的任务。因此 MXMACA 编程模型主要是异步的。

一个典型的 MXMACA 程序包括并行代码和串行代码，如图 2-2 所示。串行代码在主机 CPU 上执行，而并行代码在 GPU 上执行。主机代码按照标准 C/C++编写，而设备代码使用 MXMACA C/C++编写，mxcc 编译器为主机和设备生成可执行代码。

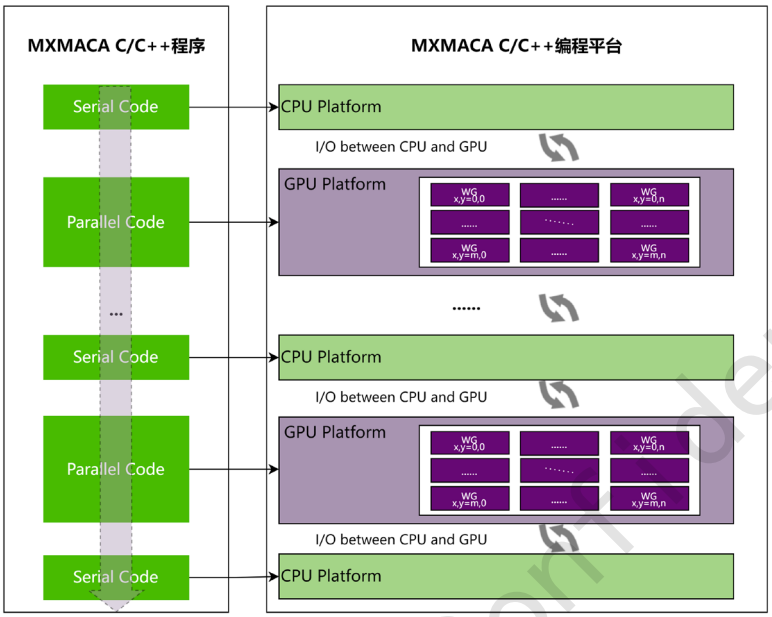

图 2-2 MXMACA C/C++程序常见范式

一个典型的 MXMACA 程序实现流程应遵循下面的模式：

1. 把数据从 CPU 内存拷贝到 GPU 内存
2. 调用核函数对 GPU 内存的数据进行处理
3. 将数据从 GPU 内存传送回 CPU 内存

2.2 执行模型

曦云系列 GPU 编程的执行模型如图 2-3 所示：

- GPU 硬件将执行内核的各个实例定义为一个工作项（work-item），等同于 MXMACA 程序里的一个 GPU 线程（thread），工作项由它在索引空间中的坐标来标识。这些坐标就是工作项的全局 ID。
- GPU 硬件会相应地提交相同内核执行的命令创建一个工作项集合，称之为工作组（work-group），等同于 MXMACA 程序里的一个线程块（thread block）。工作组中的各个工作项使用内核定义的相同指令序列。尽管指令序列是相同的，但是由于代码中的分支语句或者通过全局 ID 选择的数据可能不同，因此各个工作项的行为可能不同。
  - 工作组提供了对索引空间更粗粒度的分解，跨越整个全局索引空间。工作组在相应维度的大小相同，这个大小可以整除各维度中的全局大小。为工作组指定一个唯一的 ID，这个 ID 与工作项使用的索引空间有相同的维度。为工作项指定一个局部 ID，这个局部 ID 在工作组中是唯一的，这样就能由其全局 ID 或者由其局部 ID 和工作组 ID 唯一地标识一个工作项。

- 一个软件 Grid 是一个 N 维的值网格，称为线程网格。这个 N 维索引空间中的 N 可以是 1、2 或 3。一个软件 Grid（OpenCL 的 ND-Range）包括多个工作组（Work-Group），每个工作组负责执行 Grid 指定的一个工作项集合。

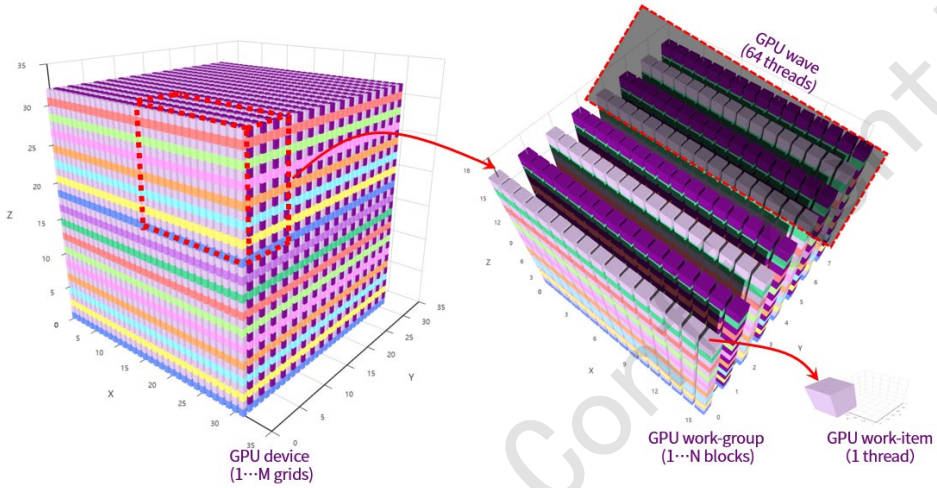

图 2-3 GPU 编程的执行模型

MXMACA 编程采用单指令多线程（SIMT）架构来管理和执行线程。每 64 个线程为一组，被称为线程束（wave）。线程束中所有线程同时执行相同的指令，每个线程都有自己的指令地址计数器和寄存器状态，利用自身数据执行当前指令。曦云系列 GPU 硬件里的加速处理器（Accelerated Processor，AP）都将分配给它的线程划分到线程束中，然后在可用的硬件资源上调度执行。64 个线程组成一个线程束来自于沐曦 GPU 硬件系统设计，和 OpenCL 的线程束大小是一致的（waveSize=64），它是加速处理器上用 SIMD 方式所同时处理的工作粒度。MXMACA 软件的程序设计需要参照 waveSize 来定义线程块大小，优化工作负载以适应线程束的边界，可以更有效地利用 GPU 资源。

SIMT 架构与 SIMD 架构相似，两者都将相同的指令广播给多个执行单元来实现并行。一个关键区别是，SIMD 要求同一向量的所有元素在一个同步组中一起执行，而 SIMT 则允许同一线程束的多个线程独立执行。尽管一个线程束中所有线程在相同的程序地址同时开始执行，但是单独的线程仍有可能有不同的行为。SIMT 确保可以编写独立的线程级并行代码、标量线程、以及用于协调线程的数据并行代码。SIMT 模型包含 3 个 SIMD 所不具备的关键特征：

- 每个线程都有自己的指令地址计数器
- 每个线程都有自己的寄存器状态
- 每个线程都可以有一个独立的执行路径

2.3 内存层次模型

MXMACA 编程的内存层次模型如图 2-4 所示，内存是分层次的，每种不同类型的内存空间都有不同的作用域、生命周期和缓存行为。在一个核函数中，每个线程都有自己的私有内存（Private Memory），每个

线程块有自己的工作组共享内存（Workgroup Shared Memory，WSM）并对块内的所有线程可见，一个线程网格中的所有线程都可以访问全局内存和常量内存，其中常量内存为只读内存空间。理解了基于沐曦 GPU 硬件架构的 MXMACA 内存层次模型，有助于帮助 MXMACA 程序设计如何通过改善访存策略来提升 MXMACA 核函数的性能。

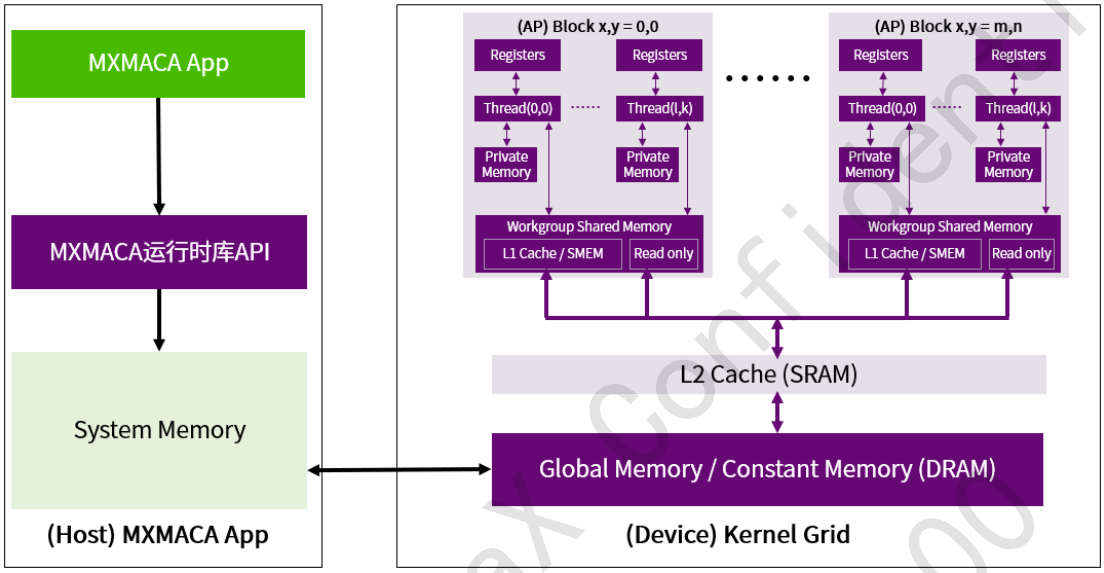

图 2-4 MXMACA 编程的内存层次模型

MXMACA 编程和其他 GPU 编程语言一样，根据存储器是否可以被程序员控制，可将其分为两种类型：

- 可编程存储器：需要显式控制哪些数据放在可编程内存中
- 不可编程存储器：不能决定哪些数据放在这些存储器中，也不能决定数据在存储器中的位置

GPU 的可编程存储单元包括全局存储，常量存储，共享存储，本地存储和寄存器等。而不可编程存储单元则包括一级缓存、二级缓存等。另外，CPU 端（主机端）存储类型，以及 CPU 和 GPU 的通信接口和通信方式也会影响 GPU 程序执行的性能。

2.4 核函数

MXMACA 核函数调用是对 C 语言函数调用语句的延伸，核函数启动代码如下：

```
kernel_name <<<grid, block>>>(argument_list);
```

其中<<<>>>运算符内是核函数的执行配置：

- 第一个值是线程网格维度，即启动线程块的数目
- 第二个值是线程块的维度，即每个线程块的线程数目

通过指定线程网格和线程块的大小，可以配置核函数中线程的使用数目和线程的使用布局。二维线程网格的线程布局如图 2-5 所示：

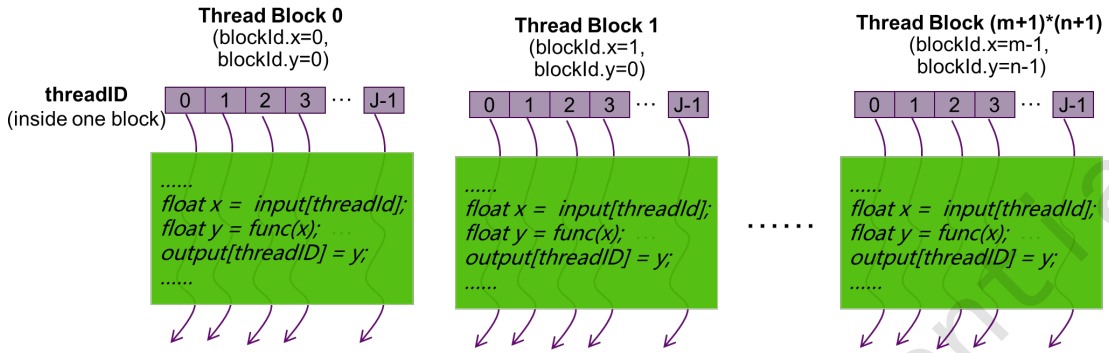

图 2-5 线程网格布局示例

由于数据在全局内存中是线性存储的，二维线程网格可以使用变量(blockIdx.x, blockIdx.y)，二维线程块可以使用(threadIdx.x, threadIdx.y)：

- 在线程网格中标识唯一的线程
- 建立线程块和数据元素之间映射关系

2.5 动态并行

MXMACA 的动态并行是一个扩展功能，能够在 GPU 上直接创建任务和同步操作，减少了主机和设备之间传输控制和数据的需求。此外，依赖于数据的并行工作可以在核函数里面直接进行决策和调整，以前需要修改代码以消除递归、不规则循环结构或者其他不适合扁平、单级并行的算法和编程模式可以更透彻的表达。

MXMACA 执行模型基于线程、线程块和网格的基元，核函数定义了线程块和线程网格中各个线程执行的程序。当调用核函数时，网格的属性由执行配置描述，该配置在 MXMACA 中具有特殊语法。MXMACA 的动态并行扩展了配置、启动和隐式同步新网格到设备上运行的线程能力，如图 2-6 所示。

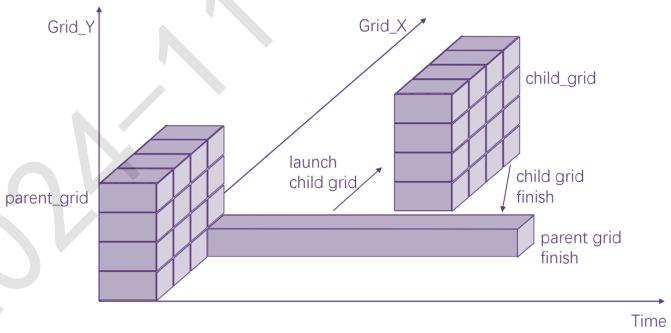

图 2-6 动态并行示意图

配置和启动新网格的设备线程属于父网格，调用创建的网格是子网格。子网格的调用和完成是正确嵌套的，这意味着父网格在其线程创建的所有子网格完成之前不会被认为是完整的，并且设备运行时由驱动软件保证父网格和子网格之间的隐式同步。

2.6 图编程

MXMACA 编程中经典的启动内核就是采用<<<>>>语法的接口，这个三尖号语法在编译时会被替换为 MXMACA 运行时库提供的 `mcLaunchKernel` 函数。

当这些内核很多且持续时间很短时，启动开销有时会成为一个问题。MXMACA 图编程提供了一种减少开销的方法，通过将用户的一系列操作定义为任务图，可以将任意数量的异步 MXMACA API 调用（包括内核启动）组合到一个只需要一次启动的操作中，可以显著减少启动大量用户操作的开销。图 2-7 给出了 MXMACA 图编程与经典 MXMACA 流和并发编程的主要区别。此外，通过提前了解任务图的整个工作流，MXMACA 驱动程序也能够应用各种优化。当然，这种性能提升是以牺牲灵活性为代价的：如果事先不知道整个工作流，则 GPU 执行必须中断，才能返回 CPU 做出决定。

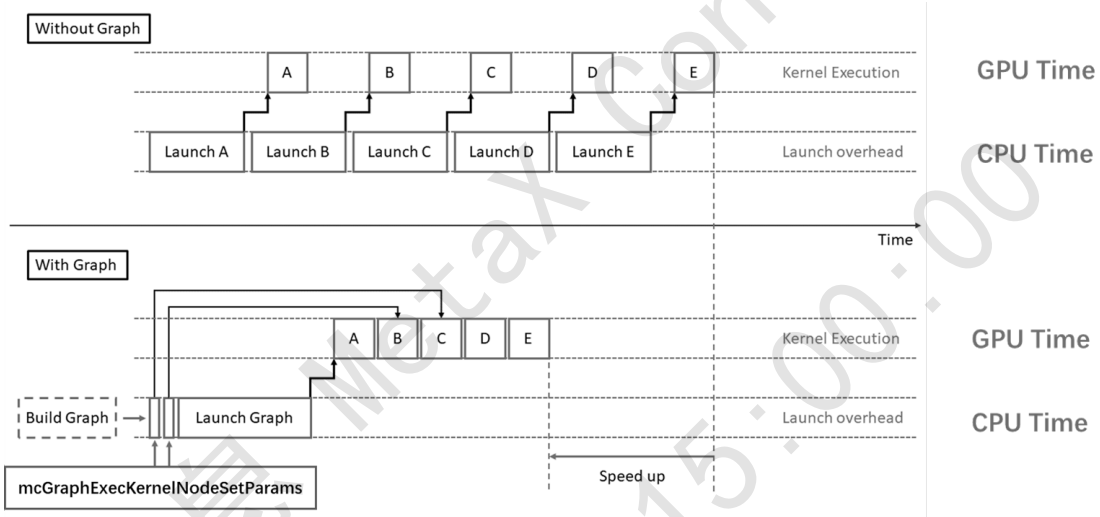

图 2-7 MXMACA 图编程场景示例

2.7 MPS 多进程服务

随着 GPU 性能的显著提升，单一应用程序可能难以完全发挥 GPU 的全部潜力。MXMACA 多进程服务（MPS）是一种先进的机制，它使得多个 CPU 进程能够通过设定的计算任务队列使用策略，在同一 GPU 上实现多个进程任务的并发提交和执行。

3. 编程接口

3.1 设备管理

一个服务器主机系统可以有多个设备，设备之间可以通过 PCIe 或者 MetaXLink 进行通信。曦云系列 GPU 提供了查询和管理设备的方法。通过这些方法，可选择合适的设备进行任务工作。

3.1.1 查询设备信息

主机线程可以通过调用 `mcGetDeviceCount()` 随时查看可以使用的 GPU 数量，并通过调用 `mcGetDeviceProperties()` 查询每个设备的属性。

代码示例

可使用以下代码，查询设备信息：

```
int main( void ) {
    mcDeviceProp prop;

    int count;
    HANDLE_ERROR( mcGetDeviceCount( &count ) );
    for (int i=0; i< count; i++) {
        HANDLE_ERROR( mcGetDeviceProperties( &prop, i ) );
        printf( " --- General Information for device %d ---\n", i );
        printf( "Name: %s\n", prop.name );
        printf( "Compute capability: %d.%d\n", prop.major, prop.minor );
        printf( "Clock rate: %d\n", prop.clockRate );
        printf( "Device copy overlap: " );
        if (prop.deviceOverlap)
            printf( "Enabled\n" );
        else
            printf( "Disabled\n" );
        printf( "Kernel execition timeout : " );
        if (prop.kernelExecTimeoutEnabled)
            printf( "Enabled\n" );
        else
            printf( "Disabled\n" );
        printf( " --- Memory Information for device %d ---\n", i );
        printf( "Total global mem: %ld\n", prop.totalGlobalMem );
        printf( "Total constant Mem: %ld\n", prop.totalConstMem );
        printf( "Max mem pitch: %ld\n", prop.memPitch );
```

```
        printf( "Texture Alignment: %ld\n", prop.textureAlignment );

        printf( " --- Multi-Processor(MP) Info for device %d ---\n", i );
        printf( "Multi-processor count: %d\n",
                prop.multiProcessorCount );
        printf( "Shared mem per MP: %ld\n", prop.sharedMemPerBlock );
        printf( "Registers per MP: %d\n", prop.regsPerBlock );
        printf( "Threads in wave: %d\n", prop.waveSize );
        printf( "Max threads per block: %d\n",
                prop.maxThreadsPerBlock );
        printf( "Max thread dimensions: (%d, %d, %d)\n",
                prop.maxThreadsDim[0], prop.maxThreadsDim[1],
                prop.maxThreadsDim[2] );
        printf( "Max grid dimensions: (%d, %d, %d)\n",
                prop.maxGridSize[0], prop.maxGridSize[1],
                prop.maxGridSize[2] );
        printf( "\n" );
    }
}
```

编译运行上述示例代码，打印结果如下图所示：

```
 --- General Information for device 0 ---
Name: Device 4000
Compute capability: 10.0
Clock rate: 1600000
Device copy overlap: Enabled
Kernel execition timeout : Disabled
 --- Multi-Processor(MP) Information for device 0 ---
Multi-processor count: 110
Shared mem per MP: 65536
Registers per MP: 131072
Threads in wave: 64
Max threads per block: 1024
Max thread dimensions: (1024, 1024, 1024)
Max grid dimensions: (2147483647, 2147483647, 2147483647)
```

图 3-1 设备信息查询的打印示例

3.1.2 选择运行设备

主机线程可以通过调用 `mcSetDevice()` 随时设置要执行操作的设备。在当前设置的设备上进行设备内存分配和内核启动，创建与当前设置的设备相关联的流和事件。如果未调用 `mcSetDevice()`，则当前设备为设备 0。

代码示例

以下代码示例说明了设置当前设备如何影响内存分配和内核执行：

```
size_t size = 1024 * sizeof(float);
mcSetDevice(0);             // Set device 0 as current
float* p0;
mcMalloc(&p0, size);        // Allocate memory on device 0
MyKernel<<<1000, 128>>>(p0); // Launch kernel on device 0
mcSetDevice(1);             // Set device 1 as current
float* p1;
mcMalloc(&p1, size);        // Allocate memory on device 1
MyKernel<<<1000, 128>>>(p1); // Launch kernel on device 1
```

3.1.3 初始化设备

MXMACA 编程不需要显式进行设备初始化，驱动程序会在用户程序第一次调用 MXMACA 的运行时 API 函数时进行隐式初始化。

MXMACA 的驱动程序为系统中的每个设备创建 MXMACA 上下文，该上下文是该设备的主要上下文（Primary Context），也是在第一次调用 MXMACA 的运行时 API 函数时初始化，并且在应用程序的所有主机线程之间共享。作为上下文创建的一部分，设备代码在必要时进行即时编译并加载到设备内存中，详情参见 4.2 运行时编译和动态加载。

当主机线程调用 `mcDeviceReset()` 时，会破坏主机线程当前操作设备（即 3.1.2 选择运行设备中定义的当前设备）的主上下文。将此设备作为当前设备的任何主机线程，进行下一次运行时函数调用将为此设备创建新的主上下文。

3.2 内存管理

曦云系列 GPU 内存可以分为主机系统内存（system memory）和设备内存（device memory）。

3.2.1 内存申请与释放

申请系统内存时，使用 `mcMallocHost()` API。

```
mcMallocHost(void ** size_t sizeBytes, unsignet int flags)
```

当需要分配设备内存时：

1. 通过 `mcSetDevice(deviceId)` API 选择需要分配内存的设备，若不显示指定设备，则默认分配到 `deviceId=0`。
2. 通过 `mcMalloc(void **ptr,size_t sizeBytes)` API 分配设备内存。使用上述接口分配出来的内存均为 SVM 内存，即设备和 CPU 均可以访问。

系统内存和设备内存的释放分别使用 `mcFreeHost(void *ptr)` 和 `mcFree(void *ptr)`。

3.2.2 内存拷贝

曦云系列 GPU 支持系统内存与设备内存之间进行拷贝，或者设备内存之间以及系统内存之间拷贝。内存拷贝分为两种类型：阻塞拷贝与异步拷贝。

- 阻塞拷贝与 glibc 提供的 memcpy 类似，可以使用 `mcMemcpy()` API，直到全部数据拷贝完成，该 API 才会返回。
- 异步拷贝依赖于 3.3.1 流，异步拷贝将拷贝任务放到流队列后，API 直接返回，需要用户通过流机制查询是否执行完成。异步拷贝使用 `mcMemcpyAsync()` APl。

3.3 流和事件

3.3.1 流

在使用曦云系列 GPU 时，通常都是基于一个或多个流进行编程（曦云系列 GPU 秉承了流式编程思想）。流（stream）是 GPU 异构编程模型的核心，它是在 Host 侧发起并在一个 Device 上执行的操作序列，操作的下发和实际执行是异步的，Device 按照 Host 侧发起的顺序执行对应操作，保证在同一个流上先发起的操作会先被执行。如图 3-2 所示，一个流上的任务会调度到一个硬件工作队列，不同流上的任务在硬件资源充足时会调度到不同的硬件工作队列支持 GPU 任务并发执行，硬件资源不足时可能会调度到同一个硬件工作队列串行执行。

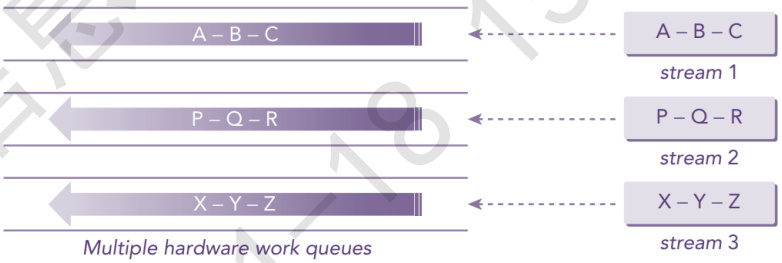

图 3-2 MXMACA 流和并发示意图

按照创建方式分类，流可以分为默认流和用户自定义流。

对于内存拷贝（`mcMemcpy`）、内存赋值（`mcMemset`）、内核启动等操作，如果没有显式指定流，MXMACA 将把这些操作放入默认流中执行。

如果想实现操作的并发，则需要使用用户自定义流。曦云系列 GPU 当前支持将以下操作设置为独立任务，可以彼此并发操作：

- 在主机上进行计算
- 在设备上进行计算
- 从主机到设备的内存传输

- 从设备到主机的内存传输
- 在给定设备的存储器内进行存储器传输
- 设备之间的内存传输

代码示例

使用流进行同步 API 和异步 API 的内存拷贝，代码如下所示：

```
/***** 同步 API *****/
mcMalloc();
mcMemcpy(…, mcMemcpyHostToDevice);
kernel<<<Dg, Db, 0, 0>>>(…);
mcMemcpy(…,mcMemcpyDeviceToHost);

/***** 异步 API *****/
mcStreamCreate(&stream);
mcMallocAsync(&ptr, mem_size, stream);
mcMemcpyAsync(…, mcMemcpyHostToDevice, stream);
kernel<<<Dg, Db, Ns, stream>>>(…);
mcMemcpyAsync(…,mcMemcpyDeviceToHost, stream);
mcFreeAsync(ptr, stream);
/* 等待，直到之前的异步操作全部完成 */
mcStreamSynchronize(stream);
```

流管理函数

曦云系列 GPU 提供的流管理运行时 API 函数见表 3-1，支持创建、销毁流，流上启动内核函数，流的查询与同步，流属性的设置、获取和拷贝，以及流主机函数回调等操作。

表 3-1 曦云系列 GPU 流管理函数

功能	运行时 API 函数
流的创建与销毁	mcStreamCreate
	mcStreamCreateWithFlags
	mcStreamCreateWithPriority
	mcStreamDestroy
流上启动内核函数	mcLaunchKernel
流的查询与同步	mcStreamQuery
	mcStreamSynchronize
	mcStreamWaitEvent
流属性的设置、获取和拷贝	mcStreamGetFlags

功能	运行时 API 函数
	mcStreamGetPriority
	mcStreamSetAttribute
	mcStreamGetAttribute
	mcStreamCopyAttribute
流主机函数回调	mcStreamAddCallback

3.3.2 事件

事件一般用作流中的标记，作为流内特定的同步点。可以使用事件来执行以下两个基本任务：

- 流同步
- 监控流中操作的执行进度

代码示例

使用事件进行核函数执行时间查询，代码如下所示：

```
// create two events
mcEvent_t start, stop;
mcEventCreate(&start);
mcEventCreate(&stop);
// record start event on the default stream
mcEventRecord(start);
// execte kernel
kernel<<<grid, block>>>(arguments);

// record stop event on the default stream
mcEventRecord(stop);
// wait until the stop event completes
mcEventSynchronize(stop);
// calculate the elapsed time between two events
float time;
mcEventElapsedTime(&time, start, stop);
// clean up the two events
mcEventDestroy(start);
mcEventDestroy(stop);
```

事件管理函数

曦云系列 GPU 提供的事件管理运行时 API 函数见表 3-2，支持创建、销毁事件，查询事件状态，记录事件和同步等操作。

表 3-2 曦云系列 GPU 事件管理函数

功能	运行时 API 函数
事件的创建与销毁	mcEventCreate
	mcEventCreateWithFlags
	mcEventDestroy
事件的记录	mcEventRecord
	mcEventRecordWithFlags
事件的查询与同步	mcEventQuery
	mcEventSynchronize
	mcStreamWaitEvent
事件的持续时间	mcEventElapsedTime

3.3.3 操作并发

3.3.3.1 使用异步流进行并发

在使用曦云系列 GPU 时，可以通过将操作发布在不同的异步流上以实现操作的并发。

代码示例

使用异步流进行模型推理的并发操作，代码如下所示：

```
#define STREAM_NUMS 5
mcStream_t stream[STREAN_NUMS];
for (uint32_t i = 0; i < STREAM_NUMS; i++) {
    mcStreamCreate(&stream[i]);
    mcnnExecuteAsync(ctx, network_id, batch_size, inputs, outputs,
stream[i]);
}
for(uint32_t i = 0; i < STREAM_NUMS; i++) {
    mcStreamSynchronized(stream[i]);
    mcStreamDestroy(stream[i]);
}
```

3.3.3.2 设置流之间依赖

在一些复杂的应用场景下，可能需要在一个流中阻塞调用线程以等待另一个流中的操作执行完成，这时需要设置流之间的依赖。

使用如下 API，在一个流中插入对另一个流中同步点的依赖：

```
mcError_t mcStreamWaitEvent(mcStream_t stream, mcEvent_t event, unsigned int flags)
```

代码示例

设置两个流的推理操作与内存拷贝的依赖关系，代码如下所示：

```
mcStream_t stream1, stream2;
mcEvent_t sync_event;
mcStreamCreateWithFlags(&stream1, mcStreamNonBlocking);
mcStreamCreateWithFlags(&stream2, mcStreamNonBlocking);
mcEventCreateWithFlags(&sync_event, mcEventBlockingSync);
mcMemcpyAsync(dst, src, byte_size, mcMemcpyHostToDevice, stream1);
mcEventRecord(sync_event, stream1);
mcStreamWaitEvent(stream2, sync_event, mcEventWaitDefault);
mcnnExecuteAsync(ctx, network_id, batch_ size, inputs, outputs, stream2);
```

其中，stream1 及 stream2 上的操作均不会阻塞调用线程，但是 stream2 上的推理操作需等待 stream1 上的内存拷贝完成后再开始。

3.3.4 流回调

流回调用于设置添加当流上的操作全部完成后执行的动作，属于一种设备与主机之间的同步机制，回调函数由调用者提供。使用如下 API 添加回调：

```
mcError_t mcStreamAddcallback(mcStream_t stream,mcstreamcallback_t callback, void *userData,unsigned int flags)
```

每次使用 `mcStreamAddCallback` 添加一次回调，回调每次只执行一次。在执行回调时，会阻塞添加回调函数后的操作执行。

此外，`userData` 可用于指定调用者传递给回调函数的数据。

说明

- flags 当前未使用，需设置为 0。
- 通常，不应在回调函数中添加设备 API 相关的代码。

代码示例

在推理完成后添加回调，代码如下所示：

```
static my_callback(mcStream_t stream, mcError_t status, void *userData) {
printf("receive call back from stream[%p] with status[%d]", stream, status);
mcStream_t stream;
mcStreamCreate(&stream);
```

```
mcnnExecuteAsync(ctx, network_id, batch_size, inputs, outputs, stream);
mcStreamAddCallback(stream, my_callback, nullptr, 0);
```

3.4 同步

MXMACA 提供了一些同步原语用于完成 GPU 与 CPU 之间以及 GPU 内部的同步，主要包含以下三种同步机制：

- 系统级同步：完成 CPU 和 GPU 之间的同步
- 线程块级/线程束级同步：完成一个线程块/线程束内所有线程之间的同步
- 用户细粒度并行同步：用户以其他粒度来定义和同步线程组

3.4.1 系统级同步

MXMACA 为系统级同步提供了多个 API，可以执行不同粒度的同步操作：

- `mcError_t mcDeviceSynchronize(void)`
- `mcError_t mcStreamSynchronize(mcStream_t stream)`
- `mcError_t mcEventSynchronize(mcEvent_t event)`

3.4.2 线程块级/线程束级同步

MXMACA 为线程块级同步提供了一些内置同步函数，可以完成线程块内或者线程束内的同步，典型的同步函数有：

- `void __sync_threads()`
- `void __sync_wave(unsigned mask=0xffffffff)`

3.4.3 用户细粒度并行同步

但是一些高效的并行算法往往需要线程协作（threads cooperate）以及共享数据（share data）来完成集体计算（collective computations）。要共享数据，线程间必然会涉及同步，而共享的粒度因算法而异，因此线程间的同步应尽量足够灵活，比如开发者可以显式地指定线程间同步，这样就可以确保程序的安全性、可维护性和模块化设计。基于该思想，MXMACA 编程模型支持了协作组（Cooperative Groups）的概念，以允许内核动态组织线程组来满足这些需求，如图 3-3 所示。

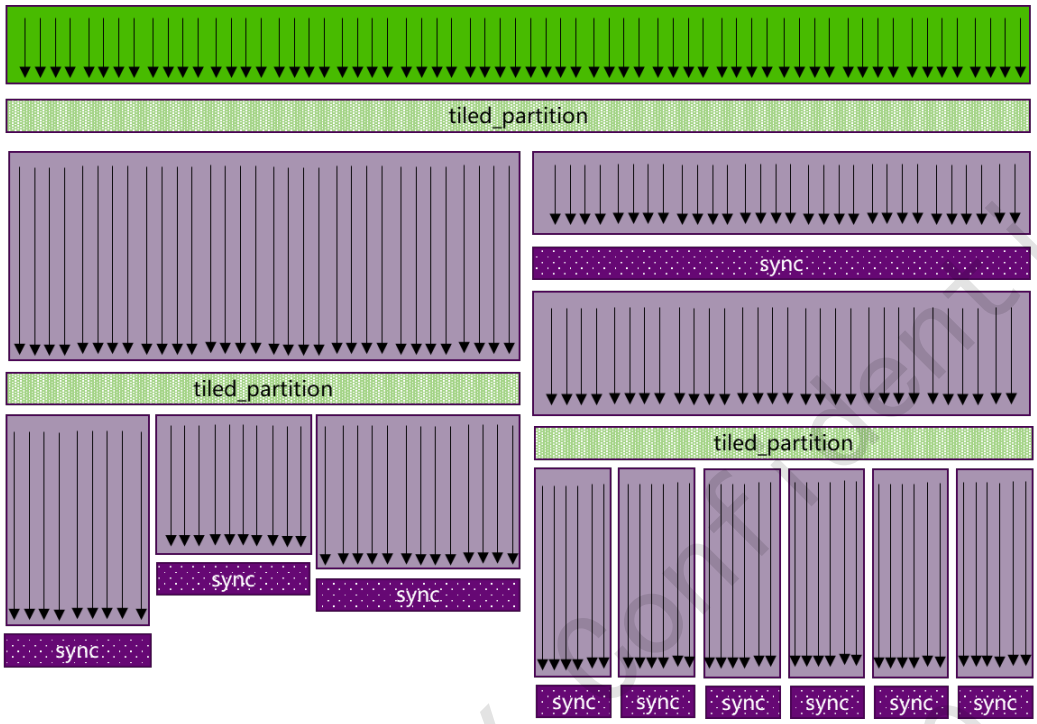

图 3-3 协作组支持灵活线程组的显式同步

协作组编程模型提供了 MXMACA 线程块内和跨线程块的同步模式，并提供了一套定义、划分和同步线程组的 MXMACA 设备代码 API。它还提供了主机端 API 来启动网格，网格的线程都保证同时执行，以实现跨线程块的同步。这些原语在 MXMACA 中启用新的协作并行模式，包括生产者-消费者并行和跨整个线程网格甚至多个 GPU 的全局同步。

将分组表示为一级程序对象可以改进软件的组合：集体函数可以采用显式参数表示参与线程的组。考虑一个库函数，它对调用者施加了要求。显式分组将这些需求显式化，从而减少误用库函数的机会。显式分组和同步有助于使代码不那么脆弱，减少对编译器优化的限制，并提高向前兼容性。

协作组编程模型由以下元素组成：

- 表示协作线程组的数据类型
- 获取由 MXMACA Launch API 定义的隐式线程组的操作
- 用于将现有组划分为新组的集体操作
- 用于数据移动和修改的集体算法（例如 memcpy_async、reduce、scan）
- 同步组内所有线程的操作
- 检查线程组属性的操作
- 公开低级别、特定组且通常由硬件加速的集体操作

协作组 API

协作组 API 用于定义和同步 MXMACA 程序中的线程组，要使用协作组，请包含其头文件。

```
#include <cooperative_groups.h>
```

协作组类型和接口是在 cooperative_groups C++命名空间中定义的，所以你可以用 cooperative_groups:: 作为所有名称和函数的前缀，或者用 using 指令加载命名空间或其类型。

```
using namespace cooperative_groups; // or...
using cooperative_groups::thread_group; // etc.
```

通常也会定义一个别名，本文后面的示例中会用到如下命名空间别名。

```
namespace cg = cooperative_groups;
```

包含任何块内协作组功能的代码可以使用 mxcc 正常方式编译。

协作组中的基本类型是 thread_group，它是一组线程的句柄。该句柄只能由它所代表的组的成员访问。线程组公开一个简单的接口。您可以使用 size() 方法获取线程组的大小（线程总数）：

```
unsigned size();
```

要在组中查找调用线程（介于 0 和 size()-1 之间）的索引，请使用以下 thread_rank() 方法：

```
unsigned thread_rank();
```

最后，您可以使用该方法 is_valid() 检查分组的有效性：

```
bool is_valid();
```

线程组集体操作

线程组提供了在组中所有线程之间执行集体操作（collective operations）的能力。集体操作是需要在一组指定的线程之间进行同步或通信的操作。由于需要同步，每个被标识为参与集体操作的线程都必须对该集体操作进行匹配调用。最简单的集体操作就是一个屏障，不传输任何数据，只是同步组中的线程。

MXMACA 程序员可以通过调用 sync() 方法或调用 cooperative_groups::sync() 函数来同步组，此时组内的所有线程之间将会进行一个同步，如图 3-3 所示。这个就类似 _synthreads() 以一个 block 为单位进行同步；__syncwave() 以一个 wave 为单位进行同步；而 g.sync() 线程组同步以指定的线程组为单位进行同步。

```
g.sync();            // synchronize group g
cg::synchronize(g); // an equivalent way to synchronize g
```

接下来讨论如何在 MXMACA 程序中创建线程组。协作组引入了一种新的数据类型 thread_block，按照如下方法初始化得到的 thread_block 实例是 MXMACA 线程块中线程组的句柄。

```
thread_block block = this_thread_block();
```

与其它 MXMACA 程序一样，执行该行的每个线程都有自己的变量块实例。MXMACA 内置变量 blockIdx 值相同的线程属于同一个线程块组。同步一个 thread_block 组和__syncthreads() 作用一样，下面几行代码都是做同步操作，只是同步的颗粒度不一样。

```
__syncthreads();
block.sync();
cg::synchronize(block);
this_thread_block().sync();
cg::synchronize(this_thread_block());
```

thread_block 数据类型使用以下特定于块的方法扩展了 thread_group 接口，这些分别相当于 MXMACA 常规线程块里的 blockIdx 和 threadIdx。

```
dim3 group_index();  // 3-dimensional block index within the grid
dim3 thread_index(); // 3-dimensional thread index within the block
```

协作组示例

下面将介绍一个简单的内核，使用 reduce_sum()函数来计算输入数组中所有值的总和。

1. 首先调用 thread_sum()并行计算许多部分和，其中每个线程跨过数组 blockDim.x * gridDim.x 大小先计算一部分和（这里使用了向量化加载以获得更高的内存访问效率）。
2. 然后使用 thread_block 线程组执行协作求和，负责每个 block 内的求和。
3. 最后调用 atomicAdd()完成每个 block 的求和。

完整的示例代码如下所示：

```
#include<iostream>
#include<mc_runtime.h>
#include<MXMACA_cooperative_groups.h>

using namespace cooperative_groups;
__device__ int reduce_sum(thread_group g, int *temp, int val)
{
    int lane = g.thread_rank();

    // Each iteration halves the number of active threads
    // Each thread adds its partial sum[i] to sum[lane+i]
    for (int i = g.size() / 2; i > 0; i /= 2)
    {
        temp[lane] = val;
        g.sync(); // wait for all threads to store
        if(lane<i) val += temp[lane + i];
        g.sync(); // wait for all threads to load
    }
    return val; // note: only thread 0 will return full sum
}
```

```
__device__ int thread_sum(int *input, int n)
{
    int sum = 0;

    for(int i = blockIdx.x * blockDim.x + threadIdx.x;
        i < n / 4;
        i += blockDim.x * gridDim.x)
    {
        int4 in = ((int4*)input)[i];
        sum += in.x + in.y + in.z + in.w;
    }
    return sum;
}

__global__ void sum_kernel_block(int *sum, int *input, int n)
{
    int my_sum = thread_sum(input, n);

    extern __shared__ int temp[];
    auto g = this_thread_block();
    int block_sum = reduce_sum(g, temp, my_sum);

    if (g.thread_rank() == 0) atomicAdd(sum, block_sum);
}

int main()
{
  int n = 5 * 1024;
  int blockSize = 256;
  int nBlocks = (n + blockSize - 1) / blockSize;
  int sharedBytes = blockSize * sizeof(int);

  int *sum, *data;
  mcMallocManaged(&sum, sizeof(int));
  mcMallocManaged(&data, n * sizeof(int));
  std::fill_n(data, n, 1); // initialize data
  mcMemset(sum, 0, sizeof(int));

  void *kernelArgs[]={
    (void*)&sum,
    (void*)&data,
    (void*)&n,
```

```
  };

  mcStream_t stream;
  mcStreamCreate(&stream);
  mcLaunchCooperativeKernel((void *)sum_kernel_block, nBlocks,
                            blockSize, kernelArgs, sharedBytes, stream);
  mcStreamSynchronize(stream);
  mcStreamDestroy(stream);
  std::cout<<"sum="<<*sum<<std::endl;
  return 0;
};
```

将代码保存到 gpuVectorAdd.cpp 文件，然后用 mxcc 编译它：

```
$ mxcc -x MXMACA syncWithCooperativeGroups.cpp -o syncWithCG
$ ./syncWithCG
sum =5120
```

3.5 流序内存分配

3.5.1 基本接口

曦云系列 GPU 支持流序内存分配：mcMallocAsync 返回分配，mcFreeAsync 释放分配。两个 API 都接受流参数来定义分配何时可用以及何时停止可用。mcMallocAsync 返回的指针值是同步确定的，可用于构建未来的工作。重要的是要注意 mcMallocAsync 在确定分配的位置时会忽略当前设备/上下文。相反，mcMallocAsync 根据指定的内存池或提供的流来确定常驻设备。

最简单的使用模式是分配、使用和释放内存到同一个流中。

```
void *ptr;
size_t size = 512;
mcMallocAsync(&ptr, size, mcStreamPerThread);
// do work using the allocation
kernel<<<..., mcStreamPerThread>>>(ptr, ...);
// An asynchronous free can be specified without synchronizing the CPU and
GPU
mcFreeAsync(ptr, mcStreamPerThread);
```

3.5.2 内存池和内存池数据结构

内存池封装了虚拟地址和物理内存资源，根据内存池的特性（Attribute）和属性（Property）进行分配和管理。内存池的主要方面是它所管理内存的种类和位置。

所有对 mcMallocAsync 的调用都使用内存池的资源。在没有指定内存池的情况下，mcMallocAsync API 使用所提供流设备的当前内存池。可以分别使用 mcDeviceSetMempool 和 mcDeviceGetMempool 来设置和查询设备的当前内存池。默认情况下（没有 mcDeviceSetMempool 调用），当前内存池是设备的默认内存池。mcMallocFromPoolAsync API 和 C++重载允许用户指定要用于分配的池，而无需将其设置为当前池。mcDeviceGetDefaultMempool 和 mcMemPoolCreate API 为用户提供内存池的句柄。

说明

- 设备的当前内存池缺省是该设备的本地内存池。因此，在不指定内存池的情况下进行分配将始终产生流设备的本地分配。
- mcMemPoolSetAttribute 和 mcMemPoolGetAttribute 可以设置和查询内存池的属性。

3.5.3 默认内存池

可以使用 mcDeviceGetDefaultMempool API 检索设备的默认内存池。来自设备默认内存池的分配是位于该设备上的不可迁移设备分配。这些分配将始终可以从该设备访问。默认内存池的可访问性可以通过 mcMemPoolSetAccess 进行修改，并通过 mcMemPoolGetAccess 进行查询。由于不需要显式创建默认池，因此有时将它们称为隐式池。设备默认内存池不支持 IPC。

3.5.4 显式内存池

mcMemPoolCreate API 创建一个显式内存池。目前显式内存池只能分配设备内存，必须在属性结构中指定显式内存池将要驻留的设备。显式内存池的主要用例是 IPC 功能。

```
// create a pool similar to the implicit pool on device 0
int device = 0;
mcMemPoolProps poolProps = { };
poolProps.allocType = mcMemAllocationTypePinned;
poolProps.location.id = device;
poolProps.location.type = mcMemLocationTypeDevice;

mcMemPoolCreate(&memPool, &poolProps));
```

3.5.5 物理页面缓存行为

应用程序可以为内存池配置释放阈值属性（mcMemPoolAttrReleaseThreshold），让流序内存分配器为该内存池保留指定大小的物理内存占用，当用户程序在该内存池进行频繁分配和释放时，可以有效减少对物理内存的操作系统调用操作，提高应用程序效率。流序内存分配器这个功能可以视为物理页面的缓存行为。默认情况下，流序内存分配器会尝试将池的物理内存最小化，也就是 mcMemPoolAttrReleaseThreshold 默认设置为 0。

释放阈值是内存池在尝试将内存释放回操作系统之前应保留的内存量（以字节为单位）。 当内存池持有超

过释放阈值字节的内存时，分配器将尝试在下一次调用流、事件或设备同步时将内存释放回操作系统。将释放阈值设置为 UINT64_MAX 可防止驱动程序在每次同步后尝试收缩池。

```
mcuint64_t setVal = UINT64_MAX;
mcMemPoolSetAttribute(memPool, mcMemPoolAttrReleaseThreshold, &setVal);
```

将 mcMemPoolAttrReleaseThreshold 设置得足够高以有效禁用内存池收缩的应用程序可能希望显式收缩内存池的内存占用。mcMemPoolTrimTo 允许此类应用程序这样做。在调整内存池的占用空间时，minBytesToKeep 参数允许应用程序保留它预期在后续执行阶段需要的内存量。

```
mcuint64_t setVal = UINT64_MAX;
mcMemPoolSetAttribute(memPool, mcMemPoolAttrReleaseThreshold, &setVal);

// Application phase needing a lot of memory from the stream ordered
allocator
for (i=0; i<10; i++) {
    for (j=0; j<10; j++) {
        mcMallocAsync(&ptrs[j],size[j], stream);
    }
    kernel<<<...,stream>>>(ptrs,...);
    for (j=0; j<10; j++) {
        mcFreeAsync(ptrs[j], stream);
    }
}

// Process does not need as much memory for the next phase.
// Synchronize so that the trim operation will know that the allocations are
no
// longer in use.
mcStreamSynchronize(stream);
mcMemPoolTrimTo(mempool, 0);

// Some other process/allocation mechanism can now use the physical memory
// released by the trimming operation.
```

3.5.6 多 GPU 的设备可访问性支持

内存池分配可访问性不遵循 mcDeviceEnablePeerAccess 或 mcCtxEnablePeerAccess。相反，流序内存分配器提供了 mcMemPoolSetAccess API 用于设置哪些设备可以访问流序内存池中的分配。默认情况下，可以从流序内存池分配所在的设备访问分配，并且无法撤销此访问权限。要启用其他设备的访问，访问设备必须与内存池的设备对等，可以通过检查 mcDeviceCanAccessPeer 来查询。如果未检查对等功能，则设置访问可能会失败并显示 mcErrorInvalidDevice。如果没有从池中进行分配，即使设备不具备对等能力，mcMemPoolSetAccess 调用也可能成功；在这种情况下，内存池中的下一次

分配将失败。

值得注意的是，mcMemPoolSetAccess 会影响内存池中的所有分配，而不仅仅是未来的分配。此外，mcMemPoolGetAccess 报告的可访问性适用于池中的所有分配，而不仅仅是未来的分配。建议不要频繁更改给定 GPU 的内存池的可访问性设置；一旦可以从给定 GPU 访问内存池，则应在内存池的整个生命周期内都可以从该 GPU 进行访问。

```
// snippet showing usage of mcMemPoolSetAccess:
mcError_t setAccessOnDevice(mcMemPool_t memPool, int residentDevice,
                int accessingDevice) {
    mcMemAccessDesc accessDesc = {};
    accessDesc.location.type = mcMemLocationTypeDevice;
    accessDesc.location.id = accessingDevice;
    accessDesc.flags = mcMemAccessFlagsProtReadWrite;

    int canAccess = 0;
    mcError_t error = mcDeviceCanAccessPeer(&canAccess, accessingDevice,
                residentDevice);
    if (error != mcSuccess) {
        return error;
    } else if (canAccess == 0) {
        return mcErrorPeerAccessUnsupported;
    }

    // Make the address accessible
    return mcMemPoolSetAccess(memPool, &accessDesc, 1);
}
```

3.5.7 IPC 内存池

在具有内存池的进程之间共享内存有两个阶段。进程首先需要设置内存池的共享访问权限，然后共享来自该内存池的特定分配。第一阶段用于建立并实施安全性，第二阶段协调每个进程中使用的虚拟地址以及映射在导入进程中何时需要有效。

3.5.7.1 创建和共享 IPC 内存池

共享对 IPC 内存池的访问涉及检索 IPC 内存池的操作系统本机句柄（使用 mcMemPoolExportToShareableHandle() API），使用常规的操作系统本机 IPC 机制将句柄转移到导入进程，并创建导入的 IPC 内存池（使用 mcMemPoolImportFromShareableHandle API）。要使 mcMemPoolExportToShareableHandle 成功，必须使用在内存池属性结构中指定的请求句柄类型创建内存池。在设备上创建一个可导出的内存池，并在不同的进程之间共享该内存池，其详细过程参见以下代码片段。

```
// in exporting process
// create an exportable IPC capable pool on device 0
mcMemPoolProps poolProps = { };
poolProps.allocType = mcMemAllocationTypePinned;
poolProps.location.id = 0;
poolProps.location.type = mcMemLocationTypeDevice;

// Setting handleTypes to a non zero value will make the pool exportable
(IPC capable)
poolProps.handleTypes = mcDevAttrHandleTypePosixFileDescriptor;

mcMemPoolCreate(&memPool, &poolProps));

// FD based handles are integer types
int fdHandle = 0;


// Retrieve an OS native handle to the pool.
// Note that a pointer to the handle memory is passed in here.
mcMemPoolExportToShareableHandle(&fdHandle,
            memPool,
            mcDevAttrHandleTypePosixFileDescriptor,
            0);

// The handle must be sent to the importing process with the appropriate
// OS specific APIs.
// in importing process
 int fdHandle;
// The handle needs to be retrieved from the exporting process with the
// appropriate OS specific APIs.
// Create an imported pool from the shareable handle.
// Note that the handle is passed by value here.
mcMemPoolImportFromShareableHandle(&importedMemPool,
          (void*)fdHandle,
          mcDevAttrHandleTypePosixFileDescriptor,
          0);
```

3.5.7.2 在导入进程中设置访问权限

导入的 IPC 内存池最初只能从其常驻设备访问，因为导入的内存池不继承导出进程设置的任何可访问性。如果导入的 IPC 内存池在导入进程中属于不可见的设备，则用户必须使用 `mcMemPoolSetAccess` API 来启用指定 GPU 对该 IPC 内存池的访问。

3.5.7.3 从导出的 IPC 内存池创建和共享分配

共享 IPC 内存池后，在导出进程中使用 mcMallocAsync()从池中进行的分配可以与已导入池的其他进程共享。由于池的安全策略是在池级别建立和验证的，操作系统不需要额外的记录来为特定的池分配提供安全性；换句话说，可以使用任何机制将导入池分配所需的不透明 mcMemPoolPtrExportData 发送到导入进程。

在没有与分配流同步的情况下，流序内存分配也可能原进程导出或导入另一个进程。所以在访问流序内存分配时，导入进程与导出进程必须遵循相同的规则。即，对分配的访问必须发生在分配流中流序内存分配之后。以下代码片段展示了 mcMemPoolExportPointer()和 mcMemPoolImportPointer()与 IPC 事件共享分配，该 IPC 事件用于保证在准备好分配之前，不会在导入进程中访问分配。

```
// preparing an allocation in the exporting process
mcMemPoolPtrExportData exportData;
mcEvent_t readyIpcEvent;
mcIpcEventHandle_t readyIpcEventHandle;

// IPC event for coordinating between processes
// mcEventInterprocess flag makes the event an IPC event
// mcEventDisableTiming  is set for performance reasons

mcEventCreate(
        &readyIpcEvent, mcEventDisableTiming | mcEventInterprocess)

// allocate from the exporting mem pool
mcMallocAsync(&ptr, size,exportMemPool, stream);

// event for sharing when the allocation is ready.
mcEventRecord(readyIpcEvent, stream);
mcMemPoolExportPointer(&exportData, ptr);
mcIpcGetEventHandle(&readyIpcEventHandle, readyIpcEvent);

// Share IPC event and pointer export data with the importing process using
//  any mechanism. Here we copy the data into shared memory
shmem->ptrData = exportData;
shmem->readyIpcEventHandle = readyIpcEventHandle;
// signal consumers data is ready
// Importing an allocation
mcMemPoolPtrExportData *importData = &shmem->prtData;
mcEvent_t readyIpcEvent;
mcIpcEventHandle_t *readyIpcEventHandle = &shmem->readyIpcEventHandle;
```

```
// Need to retrieve the IPC event handle and the export data from the
// exporting process using any mechanism.  Here we are using shmem and just
// need synchronization to make sure the shared memory is filled in.

mcIpcOpenEventHandle(&readyIpcEvent, readyIpcEventHandle);

// import the allocation. The operation does not block on the allocation
being ready.
mcMemPoolImportPointer(&ptr, importedMemPool, importData);

// Wait for the prior stream operations in the allocating stream to complete
before
// using the allocation in the importing process.
mcStreamWaitEvent(stream, readyIpcEvent);
kernel<<<..., stream>>>(ptr, ...);
```

释放分配时，需要先在导入进程中释放分配，然后在导出进程中释放分配。以下代码片段演示了在导入和导出进程中的 `mcFreeAsync` 操作之间使用 MXMACA IPC 事件提供所需的同步。导入进程中对分配的访问显然受到导入进程侧释放操作的限制。值得注意的是，`mcFree` 可用于释放两个进程中的分配，并且可以使用其他流同步 API 代替 MXMACA IPC 事件。

```
// The free must happen in importing process before the exporting process
kernel<<<..., stream>>>(ptr, ...);

// Last access in importing process
mcFreeAsync(ptr, stream);

// Access not allowed in the importing process after the free
mcIpcEventRecord(finishedIpcEvent, stream);
// Exporting process
// The exporting process needs to coordinate its free with the stream order
// of the importing process’s free.
mcStreamWaitEvent(stream, finishedIpcEvent);
kernel<<<..., stream>>>(ptrInExportingProcess, ...);

// The free in the importing process doesn’t stop the exporting process
// from using the allocation.
mcFreeAsync(ptrInExportingProcess,stream);
```

3.5.7.4 IPC 导出内存池限制

IPC 内存池目前不支持将物理块释放回操作系统。因此，`mcMemPoolTrimTo` API 充当空操作，即该 API 设置的参数 `mcMemPoolAttrReleaseThreshold` 不会生效。此行为由驱动程序控制，而不是运行时控

制，且未来可能会随驱动程序更新而发生变化。

3.5.7.5 IPC 导入内存池限制

不支持从导入池中分配；具体来说，导入池不能设置为当前内存池，也不能在 `mcMallocFromPoolAsync` API 中使用。因此，分配重用策略属性对这些 IPC 导入内存池没有意义。

3.6 多设备系统编程

3.6.1 多设备管理

在使用曦云系列 GPU 时，单个主机线程可以管理多个设备。一般来说，第一步是确定系统内可用的使能 MXMACA 设备的数量，使用如下函数获得：

```
mcError_t mcGetDeviceCount(int* count);
```

下面的代码说明了如何确定使能 MXMACA 的设备的数量，对这些设备进行遍历，并查询性能。

```
int nGpus;
mcGetDeviceCount(&nGpus);
for (int i = 0; i < nGpus; i++) {
      mcDeviceProp devProp;
      mcGetDeviceProperties(&devProp, i);
      printf( “Device %d has compute capability %d.%d.\n” ,
                  i, devProp.major, devProp.minor);
}
```

对于利用多 GPU 一起工作的 MXMACA 应用程序，必须显式指定哪个 GPU 是当前所有 MXMACA 运算的目标。使用以下函数设置当前设备：

```
mcError_t mcSetDeviceCount(int id);
```

该函数将具有标识符 `id` 的设备设置为当前设备。该函数不会与其他设备同步，因此是一个低开销的调用。使用此函数，可以在任何时间从任何主机线程中选择任何设备。有效的设备标识符范围为从 `0` 到 `ngpus-1`。如果在首个 MXMACA API 调用之前，没有显式调用 `mcSetDevice` 函数，那么当前设备会被自动设置为设备 `0`。

一旦选定了当前设备，所有的 MXMACA 运算都将应用到该设备上：

- 任何从主线程中分配来的设备内存将完全地常驻于该设备上
- 任何由 MXMACA 运行时函数分配的主机内存都会有与该设备相关的生存时间
- 任何由主机线程创建的流或事件都会与该设备相关
- 任何由主机线程启动的内核都会在该设备上执行

可以在以下情况中同时使用多 GPU：

- 在一个节点的单 CPU 线程上
- 在一个节点的多 CPU 线程上
- 在一个节点的多 CPU 进程上
- 在多个节点的多 CPU 进程上

以下代码展示了如何执行内核以及在单一的主机线程中进行内存拷贝，使用循环遍历设备：

```
for (int i = 0; i < nGpus; i++) {
     // set the current device
     mcSetDevice(i);

     // execute kernel on current device
     Kernel <<<grid, block>>>(…);

     // asynchronously transfer data between the host and current device
     mcMemcpyAsync(…);
}
```

3.6.2 点对点通信

3.6.2.1 点对点访问

点对点访问允许各 GPU 连接到同一个 PCIe 根节点上，使一个 GPU 设备直接引用存储在其他 GPU 设备内存上的数据。对于透明的内核，引用的数据将通过 PCIe 总线传输到请求的线程上。因为不是所有的 GPU 都支持点对点访问，所以需要使用以下函数显式地检查设备是否支持点对点访问：

```
mcError_t mcDeviceCanAccessPeer(int* canAccessPeer,
                                int device, int peerDevice);
```

如果设备 `device` 能够直接访问对等设备 `peerDevice` 的全局内存，那么函数变量 `canAccessPeer` 返回值为整型 `1`，否则返回 `0`。

在两个设备间，必须使用以下函数显式地启用点对点内存访问：

```
mcError_t mcDeviceEnableAccessPeer(int peerDevice, unsigned int flag);
```

这个函数允许从当前设备到 `peerDevice` 进行点对点访问。保留 `flag` 参数以备将来使用，目前必须将其设置为 `0`。一旦成功，该对等设备的内存将立即由当前设备进行访问。

这个函数授权的访问是单向的，即这个函数允许从当前设备到 `peerDevice` 的访问，但不允许从 `peerDevice` 到当前设备的访问。如果希望对等设备能直接访问当前设备的内存，则需要另一个方向单独的匹配调用。

点对点访问保持启用状态，直到使用以下函数显式地禁用：

```
mcError_t mcDeviceDisableAccessPeer(int peerDevice);
```

3.6.2.2 点对点内存复制

两个设备之间启用点对点访问之后，使用以下函数，可以异步地复制设备上的数据：

```
mcError_t mcMemcpyPeerAsync(void* dst, int dstDev, void* src, int srcDev,
                            size_t nBytes, msStream_t stream);
```

这个函数将数据从设备 srcDev 的设备内存传输到设备 dstDev 的设备内存中。函数 mcMemcpyPeerAsync 对于主机和所有其他设备来说是异步的。如果 srcDev 和 dstDev 共享相同的 PCIe 根节点或者有 MetaXLink 链路，那么数据传输是沿着 PCIe 最短路径执行的，不需要通过主机内存中转。

3.6.3 多 GPU 设备间的同步

在多 GPU 应用程序上，可以使用和单 GPU 应用程序相同的同步函数，但是必须指定适合的当前设备。多 GPU 应用程序中使用流和事件的典型工作流程如下所示：

1. 选择应用程序将使用的 GPU 集。
2. 为每个设备创建流和事件。
3. 为每个设备分配设备资源（如设备内存）。
4. 通过流在每个 GPU 上启动任务（例如，数据传输或内核执行）。
5. 使用流和事件来查询和等待任务完成。
6. 清空所有设备的资源。

只有与该流相关联的设备是当前设备时，在流中才能启动内核，才可以在流中记录事件。

任何时间都可以在任何流中进行内存拷贝，无论该流与什么设备相关联或当前设备是什么。即使流或事件与当前设备不相关，也可以查询或同步它们。

3.7 动态并行

MXMACA 的动态并行接口由以下两部分组成：

- 用 MXMACA C++语言拓展的语法编写的核函数
- 设备端运行时 API

设备端运行时 API 是主机端运行时 API 的功能子集，主要包含核函数启动、设备内存管理等操作。MXMACA 驱动尽可能保证编程接口在主机端和设备端的语法和语义，以便于编写代码在主机端或设端运

行时重用。

3.7.1 核函数

动态并行的核函数启动可以用以下代码来完成：

```
kernel_name<<< Dg, Db, Ns, S >>>([kernel arguments]);
```

- Dg 的类型为 dim3，指定网格的维度和大小
- Db 的类型为 dim3，指定每个线程块的维度和大小
- Ns 的类型为 size_t，指定除了静态分配的内存之外，每个线程块动态分配给此调用的共享内存大小（以字节为单位）。Ns 是可选参数，默认为 0。
- S 的类型为 mcStream_t，指定此次核函数启动相关联的流。S 是可选参数，默认为空流。目前 MXMACA 的设备运行时暂不支持创建流，此参数无效。

3.7.1.1 异步启动

与主机端启动核函数相同，所有设备端内核启动都与启动线程异步。也就是说，<<<>>>命令启动核函数之后将立即返回，启动线程将继续执行，直到到达隐式同步点（核函数结束点）或显式同步点（mcDeviceSynchronize()）。

子网格启动到设备，并将独立于父线程执行。子网格可以在启动后的任何时间开始执行，但不保证在启动线程到达同步点之前开始执行。

3.7.1.2 启动环境配置

全局设备配置的所有设置（例如，堆栈大小等设备限制）将与父网格保持一致，可以通过 mcDeviceGetLimit()接口进行统一配置。

对于主机启动的内核，从主机设置的核函数配置将优先于全局设置。这些配置也将用于从设备启动核函数。无法从设备重新配置核函数的环境。

3.7.1.3 用设备端运行时 API 启动

动态并行核函数也可以通过设备端运行时 API 启动，用户程序也可以直接调用 MXMACA 驱动软件提供的 mcGetParameterBuffer() 和 mcLaunchDevice()，或者 mcGetParameterBufferV2() 和 mcLaunchDeviceV2()直接启动核函数。在这两种情况下，用户程序必须按照规范以正确的格式正确填充所有必要的数据结构。这些数据结构保证了向后兼容性。与主机端启动核函数一样，设备端操作符 <<<>>>映射到底层 API，并且编译器前端可以将<<<>>>转换为 mcGetParameterBufferV2()和 mcLaunchDeviceV2()，并将用户参数等数据正确填入对应位置。

设备端运行时 API 启动核函数与主机端运行时 API 启动核函数不同，其定义如下：

```
extern __device__ void *mcGetParameterBuffer(size_t alignment, size_t size);
extern __device__ void *mcGetParameterBufferV2(void *func, dim3 gridDim,
                                               dim3 blockDim,
                                               unsigned int sharedMemSize = 0);
extern __device__ mcError_t mcLaunchDevice(void *func, void *params,
                                           dim3 gridDim,
                                           dim3 blockDim,
                                           unsigned int sharedMemSize = 0,
                                           mcStream_t stream = 0);
extern __device__ mcError_t mcLaunchDeviceV2(void *params,
                                           mcStream_t stream = 0);
void host_launch(int *data) {
    parent_launch<<< 1, 256 >>>(data);
}
```

3.7.2 内存管理

父网格和子网格共享相同的全局和常量内存存储，但具有不同的私有和共享内存。

3.7.2.1 内存声明和使用

使用设备运行时，在文件作用域中使用__device__或__constant__内存空间说明符所声明的内存，其行为一致。无论核函数最初是由主机还是设备运行时启动的，所有核函数都可以读取或写入设备变量。同样，module 中所有核函数将具有相同的内存视图。

在 MXMACA C++中，可以将共享内存声明为具有静态大小的文件作用域变量或函数作用域变量，或者声明为外部变量，其大小由核函数的调用者在运行时通过启动配置参数确定。这两种类型的声明在设备运行时下都有效。

设备端的符号（即标有__device__的变量）可以简单地通过&操作符从内核中引用，因为所有全局设备变量都在内核的可见地址空间中。这也适用于__constant__符号，尽管在这种情况下指针将引用只读数据。

鉴于设备端符号可以直接引用，那些引用符号的 MXMACA 运行时 API（例如，mcMemcpyToSymbol()或mcGetSymbolAddress()）是冗余的，因此设备运行时不支持。请注意，即使在子内核启动之前，也不能在运行的内核中更改常量数据，因为__constant__空间在设备端是只读的。

3.7.2.2 全局内存

父网格和子网格具有对全局内存的一致访问，子网格和父网格之间的一致性保证较弱。当子网格的内存视图与父线程完全一致时，在子网格的执行过程中只有两个时间点：父线程调用子网格时或者子网格结束时。

如下所示，在父 kernel 结束后 `data[threadIdx.x] = threadIdx.x + 2`

```
__global__ void child_launch(int *data, int tid) {
    data[tid] = data[tid]+1;
}

__global__ void parent_launch(int *data) {
    data[threadIdx.x] = threadIdx.x;
    __syncthreads();
    child_launch<<< 1, 1 >>>(data, threadIdx.x);
    mcDeviceSynchronize();
    data[threadIdx.x] = data[threadIdx.x]+1;
}
```

另外如果是如下代码，则因为不同子网格之间的竞争导致产生无法预测的结果：

```
__global__ void child_launch(int *data) {
    data[threadIdx.x] = data[threadIdx.x]+1;
}

__global__ void parent_launch(int *data) {
    data[threadIdx.x] = threadIdx.x;
    __syncthreads();
    child_launch<<< 1, 1 >>>(data);
    mcDeviceSynchronize();
    data[threadIdx.x] = data[threadIdx.x]+1;
}

void host_launch(int *data) {
    parent_launch<<< 1, 256 >>>(data);
}
```

3.7.2.3 零拷贝内存

零拷贝系统内存与全局内存具有相同的一致性和一致性保证。内核可以不分配或释放零拷贝内存，但可以使用从主机程序传入的指向零拷贝的指针。

3.7.2.4 常量内存

常量是不可变的，即使在父级和子级启动之间，也不能从设备中修改。也就是说，在启动之前，必须从主机设置所有`__constant__`变量的值。常量内存由所有子核函数从其各自的父核函数自动继承。

从核函数线程中获取常量内存对象的地址，与所有 MXMACA 程序具有相同的语义。并且将指针从父级传递到子级或从子级传递到父级是天然受支持的。

3.7.2.5 共享内存和私有内存

共享内存和私有内存分别属于线程块或线程，在父级和子级之间不可见或不一致。当任一位置中的对象在其所属作用域之外被引用时，行为未定义，并且可能导致错误。

如果 mxcc 编译器能够检测到指向私有或共享内存的指针作为内核启动的参数被传递，它将尝试发出警告。在运行时，程序员可以使用`__isGlobal()`内在函数来确定指针是否引用全局内存，从而可以安全地传递给子启动。

请注意，调用 `mcMemcpy*Async()`或 `mcMemset*Async()`可能会在设备上调用新的子内核，以保留流语义。因此，将共享或私有内存指针传递给这些 API 是非法的，并可能返回错误。

3.7.2.6 私有内存

私有内存是执行线程的专用存储，在该线程外部不可见。在启动子内核时，将指向私有内存的指针作为启动参数传递是非法的。从子级取消引用这样的私有内存指针将产生未定义结果。

例如，如果 `child_launch` 访问 `x_array`，则以下行为是非法的，且行为未定义：

```
int x_array[10];         // Creates x_array in parent's private memory
child_launch<<< 1, 1 >>>(x_array);
```

程序员有时很难感知编译器何时将变量放入私有内存。一般来说，传递给子内核的所有存储都应该从全局内存堆中显式分配，可以使用 `mcMalloc()`、`new()`，也可以在全局范围内声明`__device__`存储。例如：

```
// Correct - "val" is global storage
__device__ int val;
__device__ void x() {
    val = 5;
    child<<< 1, 1 >>>(&val);
}
// Invalid - "val" is private storage
__device__ void y() {
    int val = 5;
    child<<< 1, 1 >>>(&val);
}
```

3.7.3 设备管理

MXMACA 设备运行时暂不支持多 GPU；设备运行时仅能够在其当前执行的设备上操作。同时，也暂不支持在设备端查询 MXMACA 设备的属性。

3.7.4 流和事件

MXMACA 设备运行时目前仅提供了未命名（NULL）流，且不支持用户创建并使用命名流，同时也不支持将主机端申请的流传到设备上使用。

对于 MXMACA 设备运行时，一个主机端进程只有一个默认流，同一时间只能使用一个动态并行的核函数，一个程序中在主机端同时异步启动两个动态并行的核函数也只会顺序执行。

MXMACA 设备运行时目前暂不支持事件。

3.7.4.1 排序和并发

在一个网格中，从设备运行时启动的所有核函数默认启动到同一个流（隐式的 NULL 流）中是并发执行的。当同一个线程中启动多个核函数也可能是并发执行，当需要一个核函数在另一个核函数执行完之后执行，则需要在两个核函数之间添加 `mcDeviceSynchronize()` 来保证前一个执行结束。当同一网格中的多个线程启动多个相同或不同核函数时，流中的排序取决于网格中的线程调度，这可以通过 `__syncthreads()` 和 `mcDeviceSynchronize()` 等同步语句来控制。

如下代码所示，核函数 `parentKernel` 每个线程内均保证 `childKernelA` 在 `childKernelB` 之前执行，但不保证两个线程之间的 `childKernelA` 和 `childKernelA` 顺序、两个线程之间的 `childKernelA` 和 `childKernelB` 顺序，如需保证线程间的顺序，则可以使用 `_syncthreads()` 等接口。

```
__global__ void childKernelB()
{
    printf("This is child kernel B\n");
}

__global__ void childKernelA()
{
    printf("This is child kernel A\n");
}

__global__ void parentKernel()
{
    // launch child A
    childKernelA<<<1,1>>>();
    mcDeviceSynchronize()
    // launch child B
    childKernelB<<<1,1>>>();
}

void host_launch(int *data) {
```

```
    parentKernel<<< 1, 256 >>>(data);
}
```

另外，设备运行时在 MXMACA 执行模型中没有引入新的并发保证。不能保证在设备上任意数量的不同线程块之间并发执行。缺乏并发性保证扩展到父网格及其子网格。当父网格启动子网格时，一旦满足流依赖性并且硬件资源可用于启动子网格，子网格就可以开始执行，但在父网格到达隐式同步点或`mcDeviceSynchronize()`之前，不能保证子网格开始执行。

3.7.4.2 同步

由于暂不支持命名流与事件，MXMACA 设备运行时不支持调用线程与从其他线程调用的子网格之间的同步，但可以通过隐式同步或者显式同步达到调用线程与调用的子网格之间的同步，并保证子网格中发生的更改对父网格中的线程可见。

- **隐式同步**：在网格中所有线程的所有启动完成之后，网格的执行才被视为完成。如果网格中的所有线程在所有子启动完成之前退出，则将自动触发隐式同步操作。
- **显式同步**：来自任何线程的 MXMACA 设备运行时操作（包括核函数启动）在网格中的所有线程中都可见。这意味着父网格中的调用线程可以执行同步，以控制在网格任意线程上的网格启动顺序。MXMACA 设备运行时提供 `mcDeviceSynchronize()` API 用于同步，保证当前网格中在`mcDeviceSynchronize()`之前启动的所有子网格全部执行完毕，并保证子网格操作的全局地址数据对父网格可见。

3.7.5 API 附加说明

下表提供了设备端运行时 API 和主机端运行时 API 的对照说明。

表 3-3 MXMACA 事件管理函数

设备端运行时 API	说明
mcDeviceSynchronize	和主机端运行时相应的 API 行为一致
mcMalloc	
mcLaunchDevice	
mcLaunchDeviceV2	
mcMemcpy2DAsync	
mcMemcpy3DAsync	
mcMemsetAsync	
mcMemset2DAsync	
mcMemset3DAsync	
mcGetParameterBuffer	禁止在参数缓冲区中进行参数重排序，并且放置在参数缓冲中的每个单独参数需要对齐，对齐方式参考结构体的对齐。参数缓冲区的最大值为 4KB。
mcGetParameterBufferV2	
mcFree	不能释放主机端使用 `mcMalloc` 接口分配的指针
mcMemcpyAsync	对于所有的 `memcpy/memset` 函数： ● 仅支持异步的 `memcpy/set` 函数 ● 仅支持设备内的内存拷贝 使用私有内存或共享内存结果可能是未知的

3.7.6 代码示例

以下示例显示了一个包含动态并行性的简单程序：

```
#include <stdio.h>

__global__ void childKernel()
{
    printf("This is child kernel\n");
}

__global__ void parentKernel()
{
    // launch child
```

```
    childKernel<<<1,1>>>();
    printf("This is parent kernel!\n");
}

int main(int argc, char *argv[])
{
    // launch parent
    parentKernel<<<1,1>>>();
    if (mcSuccess != mcGetLastError()) {
        return 1;
    }

    // wait for parent to complete
    if (mcSuccess != mcDeviceSynchronize()) {
        return 2;
    }

    return 0;
}
```

将上面示例代码保存到 deviceHelloWorld.MXMACA，可以使用以下命令编译代码：

```
$ mxcc -x MXMACA deviceHelloWorld.MXMACA -o deviceHelloWorld
```

3.7.7 性能分析

当动态并行功能启动时，无论核函数是否调用别的核函数，系统软件和核函数的开销将增加。该开销来自设备运行时所需要准备的资源以及设备端启动核函数，并可能导致性能下降。通常，这种开销是由链接到设备运行时库的应用程序产生的。

3.7.8 限制和注意事项

动态并行保证了本文档中描述的所有语义。但是，某些硬件特性和软件实现会限制使用设备运行时的程序规模、性能和其他属性。

3.7.8.1 内存限制

MXMACA 设备运行时的驱动软件为各种管理目的保留内存，特别是为跟踪和保存待处理的父子网格状态而保留内存。用户可配置此保留项的大小，但可能带来某些启动限制。有关详细信息，请参阅下面的配置选项。

核函数启动池槽位数

当核函数启动时，将跟踪所有相关的配置和参数数据，直到核函数完成。该数据存储在系统管理的启动池中。

可通过从主机调用 `mcDeviceSetLimit()`并指定 `mcLimitDevRuntimePendingLaunchCount` 来配置启动池（固定大小）的大小。

配置选项

设备运行时驱动软件的资源分配通过主机程序的 `mcDeviceSetLimit()` API 进行控制。在启动任何内核之前必须设置限制，并且当 GPU 正在运行程序时不能更改限制。

可以设置表 3-4 中所列的命名限制：

表 3-4 内存限制配置

限制	表现
mcLimitDevRuntimePendingLaunchCount	控制核函数启动相关资源的数量，同时分配对应大小的缓冲区，存放有依赖关系的核函数相关资源。当缓冲区已满时，在设备端内核启动期间分配启动槽将失败并报 trap，提示用户修改此配置。默认大小为 2048。
mcLimitStackSize	控制每个 GPU 线程的栈内存大小（以字节为单位）。MXMACA 驱动程序根据需要自动增加每次内核启动的每线程堆栈大小。每次启动后，此大小不会重置回原始值。
mcLimitMallocHeapSize	控制 GPU 堆大小（以字节为单位）。MXMACA 设备运行时在启动核函数时需要一些额外的内存开销，会隐式使用一部分堆内存，如果堆内存不足，可能导致启动核函数失败。

内存分配和生命周期

`mcMalloc()`和 `mcFree()`在主机和设备环境之间具有不同的语义。当从主机调用时，`mcMalloc()`从未使用的设备内存中分配一个新区域。当从设备运行时调用时，这些函数映射到设备端 `malloc()`和 `free()`。这意味着在设备环境中，可分配的总内存被限制为设备 `malloc()`堆大小，该大小可能小于可用的未使用设备内存。此外，在由设备上 `mcMalloc()`分配的指针上，从主机程序调用 `mcFree()`是错误的，反之亦然。

表 3-5 内存分配功能和限制因素

	mcMalloc() on Host	mcMalloc() on Device
mcFree() on Host	Supported	Not Supported
mcFree() on Device	Not Supported	Supported
Allocation limit	Free device memory	mcLimitMallocHeapSize

3.7.8.2 API 错误和启动失败

与 MXMACA 主机端运行时一样，任何函数都可能返回错误代码，错误代码的类型为 `mcError_t`。MXMACA 设备运行时暂不支持通过 `mcGetLastError()` 调用检索每个线程的错误值。

与主机端启动类似，设备端启动可能因多种原因（无效参数等）而失败。但启动后没有错误并不意味着子内核成功完成。目前有部分情况出现错误会报 trap 进行提示。

3.8 图编程接口

3.8.1 显式图编程接口

曦云系列 GPU 支持图编程接口，我们以 **VectorAdd** 这个计算任务为例，它的任务图如图 3-4 所示。

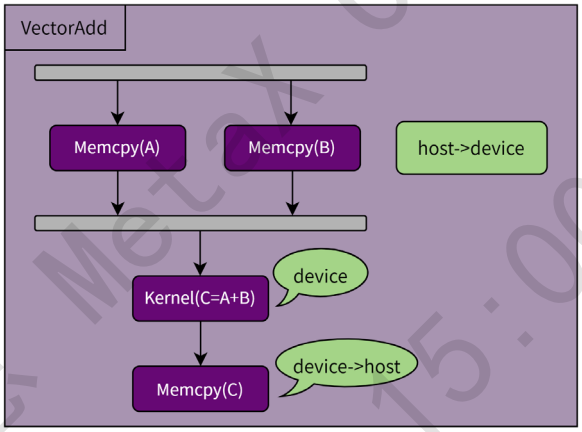

图 3-4 VectorAdd 任务图

如下代码所示，使用显式图编程接口创建 **VectorAdd** 任务图：

```
// Create a graph with graph API - it starts out empty
mcGraph_t graph;
mcGraphCreate(&graph, 0);

// Add two memory copy nodes into the graph
mcGraphAddMemcpyNode1D(&a, graph, NULL, 0, &nodeParams);
mcGraphAddMemcpyNode1D(&b, graph, NULL, 0, &nodeParams);

// Add one kernel node into the graph
mcGraphAddKernelNode(&c, graph, NULL, 0, &nodeParams);

// Add one memory copy nodes into the graph
mcGraphAddMemcpyNode1D(&d, graph, NULL, 0, &nodeParams);
```

```
// Now set up dependencies on each node
mcGraphAddDependencies(graph, &a, &c, 1);      // A->C
mcGraphAddDependencies(graph, &b, &c, 1);      // B->C
mcGraphAddDependencies(graph, &c, &d, 1);      // C->D
```

3.8.2 流捕获图编程接口

流捕获图编程接口提供了一种机制，即从现有基于流的 API 创建任务图。用于将工作启动到流中的代码片段（包括现有代码），可以用 `mcStreamBeginCapture()` 和 `mcStreamEndCapture()` 调用括起来。

如下代码所示，使用流捕获图编程接口创建 VectorAdd 任务图：

```
// Create a graph with Stream Capture API
mcGraph_t graph;
mcStreamBeginCapture(stream, mcStreamCaptureModeGlobal);

mcMemcpyAsync(device_A, host_A, size, stream);
mcMemcpyAsync(device_B, host_B, size, stream);
vectorAdd<<< ..., stream >>>( device_A, device_B, device_C...);
mcMemcpyAsync(host_C, device_C, size, stream);

mcStreamEndCapture(stream, &graph);
```

3.8.3 实例化图更新

曦云系列 GPU 提供了两种更新实例化图参数的机制，即整图更新和单个节点更新。整图更新允许用户提供拓扑相同的 `mcGraph_t` 对象，其节点包含更新的参数。单个节点更新允许用户显式更新单个节点的参数。当大量节点正在更新时，或者当调用方不知道图拓扑（即，库调用的流捕获所产生的图）时，使用更新的 `mcGraph_t`。当更新数量较少且用户拥有需要更新的节点的句柄时，最好使用单个节点更新。

3.8.3.1 整图更新

`mcGraphExecUpdate()` 支持把一个已实例化的原始任务图，通过参数更新生成另外一个拓扑相同的新任务图，不过有一些限制：

- 新任务图的拓扑必须与用于实例化 `mcGraphExec_t` 的原始任务图相同；
- 此外，节点添加到原始任务图或从原始任务图中移除的顺序必须与节点添加到新任务图（或从新任务图中移除）的顺序匹配。因此，
  - 当使用流捕获图编程接口时，必须以相同的顺序捕获节点
  - 当使用显式图编程接口时，必须按相同的顺序添加和/或删除所有节点

如下代码所示，使用流捕获图编程接口更新实例化任务图：

```
mcGraphExec_t graphExec = NULL;

for (int i = 0; i < 10; i++) {
    mcGraph_t graph;
    mcGraphExecUpdateResult updateResult;
    mcGraphNode_t errorNode;

    // use stream capture to create the graph.
    // You can also use the Graph API to produce a graph.
    mcStreamBeginCapture(stream, mcStreamCaptureModeGlobal);

    // Call a user-defined, stream based workload, for example
    do_MACA_work(stream);

    mcStreamEndCapture(stream, &graph);

    // If already instantiated the graph, update it directly
    // and avoid the instantiation overhead
    if (graphExec != NULL) {
        // If fails to update, errorNode will be set to the
        // node causing the failure and updateResult will be
        // set to a reason code.
        mcGraphExecUpdate(graphExec, graph, &errorNode, &updateResult);
    }

    // Instantiate during the first iteration or whenever the update
    // fails for any reason
    if (graphExec == NULL || updateResult != mcGraphExecUpdateSuccess) {

        // If a previous update failed, destroy the mcGraphExec_t
        // before re-instantiating it
        if (graphExec != NULL) {
            mcGraphExecDestroy(graphExec);
        }
        // Instantiate graphExec from graph. The error node and
        // error message parameters are unused here.
        mcGraphInstantiate(&graphExec, graph, NULL, NULL, 0);
    }

    mcGraphDestroy(graph);
    mcGraphLaunch(graphExec, stream);
    mcStreamSynchronize(stream);
```

```
}
```

典型的工作流是使用流捕获图编程接口或显式图编程接口创建初始 mcGraph_t；然后将 mcGraph_t 实例化并正常启动。初始启动后，使用与原始任务图相同的方法创建新的 mcGraph_t，并调用 mcGraphExecUpdate()。如果任务图更新成功（如上面示例中的 updateResult 参数所示），则启动更新的 mcGraphExec_t。如果更新因任何原因失败，将调用 mcGraphExecDestroy() 和 mcGraphInstantiate()，以销毁原始的 mcGraphExec_t 并实例化一个新的。

3.8.3.2 单个节点更新

实例化的图节点参数可以直接更新。这消除了实例化的开销以及创建新 mcGraph_t 的开销。如果需要更新的节点数量相对于图中的节点总数较少，则最好单独更新节点。以下方法可用于更新 mcGraphExec_t 节点：

- mcGraphExecKernelNodeSetParams()
- mcGraphExecMemcpyNodeSetParams()
- mcGraphExecMemsetNodeSetParams()
- mcGraphExecHostNodeSetParams()
- mcGraphExecChildGraphNodeSetParams()
- mcGraphExecEventRecordNodeSetEvent()
- mcGraphExecEventWaitNodeSetEvent()
- mcGraphExecExternalSemaphoresSignalNodeSetParams()
- mcGraphExecExternalSemaphoresWaitNodeSetParams()

3.8.3.3 启用或禁用节点（Individual Node Enable）

实例化图中的 **kernel**、**memset** 和 **memcpy** 节点可以使用 mcGraphNodeSetEnabled() API 启用或禁用。这允许创建一个图，该图包含所需功能的超集，该超集可以针对每次启动进行自定义。可以使用 mcGraphNodeGetEnabled() API 查询节点的启用状态。

在重新启用之前，禁用的节点在功能上等同于空节点。节点参数不受启用/禁用节点的影响。启用状态不受单个节点更新或使用 mcGraphExecUpdate() 进行整图更新的影响。禁用节点时的参数更新将在重新启用节点时生效。

3.8.3.4 实例化图更新的限制

更新实例化图的一些限制因节点类型不同会有所差异：

- **Kernel** 节点：最初未使用 MXMACA 动态并行的函数节点无法更新为使用 MXMACA 动态并行的函数节点
- **mcMemset** 和 **mcMemcpy** 节点：
  - 分配/映射的 MXMACA 设备无法更改。
  - 源内存（或目标内存）必须从与原始源内存（或目标内存）相同的设备中分配。
  - 只支持更改 1D **mcMemset**/**mcMemcpy** 节点。
  - 不支持更改源内存或目标内存的类型（即 `mcPitchedPtr`、`mcArray_t` 等）或传输类型（即 `mcMemcpyKind`）。
- **external semaphore** 相关节点（外部信号灯等待节点和记录节点）：不支持更改信号量的数量。
- **host** 节点和 **event** 相关节点（事件记录节点或事件等待节点）：更新没有限制。

3.8.4 图编程的注意事项

图编程使开发者能够通过流捕获或显式创建的方法，将多个核函数整合成一个任务图。与核函数融合不同，任务图内部保持了多个核函数的形式，但只需一次提交操作。理论上，将尽可能多的核函数整合到图中可以节省大量核函数提交的开销。然而，图编程 API 也有其局限性：

- **固定性**：图编程 API 的设计是将多个任务单元作为一个固定的快照进行组合，这意味着一旦快照创建，其参数和结构的更改需要严格遵循 3.8.3 实例化图更新的相关限制。
- **实例化开销**：虽然实例化图快照的过程可能耗时，但如果该任务图快照被多次执行，那么初始的实例化时间可以被忽略不计。
- **依赖和资源限制**：如果一个任务图中包含许多相互依赖的任务节点，可能会由于硬件资源限制（如有限的硬件队列个数）而导致排队或挂起问题，这需要通过实际测试来确定图编程一次提交操作是否适用。因此，图编程的任务提交默认采用标准核函数提交模式，开发者需要在开发阶段手动启用图编程任务提交模式（通过设置环境变量 `export MACA_GRAPH_LAUNCH_MODE=1`），并进行充分测试和验证。只有在结果符合预期且性能得到提升时，才应在程序部署中启用图编程任务提交模式。

因此，图编程接口提供了一种优化 GPU 任务执行的方法，但需要仔细考虑其适用性和对性能的实际影响。

3.8.5 图编程的调试接口

图编程调试接口提供了一种快速方便的方法，可以调用以下调试接口获取任务图的基本信息，对任务图中的每个图节点或节点间的依赖进行调试：

- mcGraphGetNodes
- mcGraphGetEdges
- mcGraphHostNodeGetParams
- mcGraphKernelNodeGetParams

此外，图编程调试接口也可以通过创建整个图形的全面概述和 DOT 图，把程序任务图进行可视化展开和检查。通过调用 `mcGraphDebugDotPrint` API 可以构建任何未实例化图的详细视图，演示拓扑结构、节点几何结构、属性配置和参数值。给定一个 MXMACA 任务图，它输出一个 DOT 图，其中 DOT 是一种图描述语言。该图的详细视图使您更容易识别明显的配置问题，并能够创建易于理解的错误报告，供其他人用于分析和调试问题。通过将问题隔离到特定节点，将此 API 与调试器相结合增加了实用性。

```
mcGraphDebugDotPrint(mcGraph_t hGraph, const char *path,
                     unsigned int flags);
```

例如通过调用 `mcGraphDebugDotPrint` API 输出调试信息后，**vectorAdd** 任务图可以转换成如图 3-5 所示的 DOT 图。

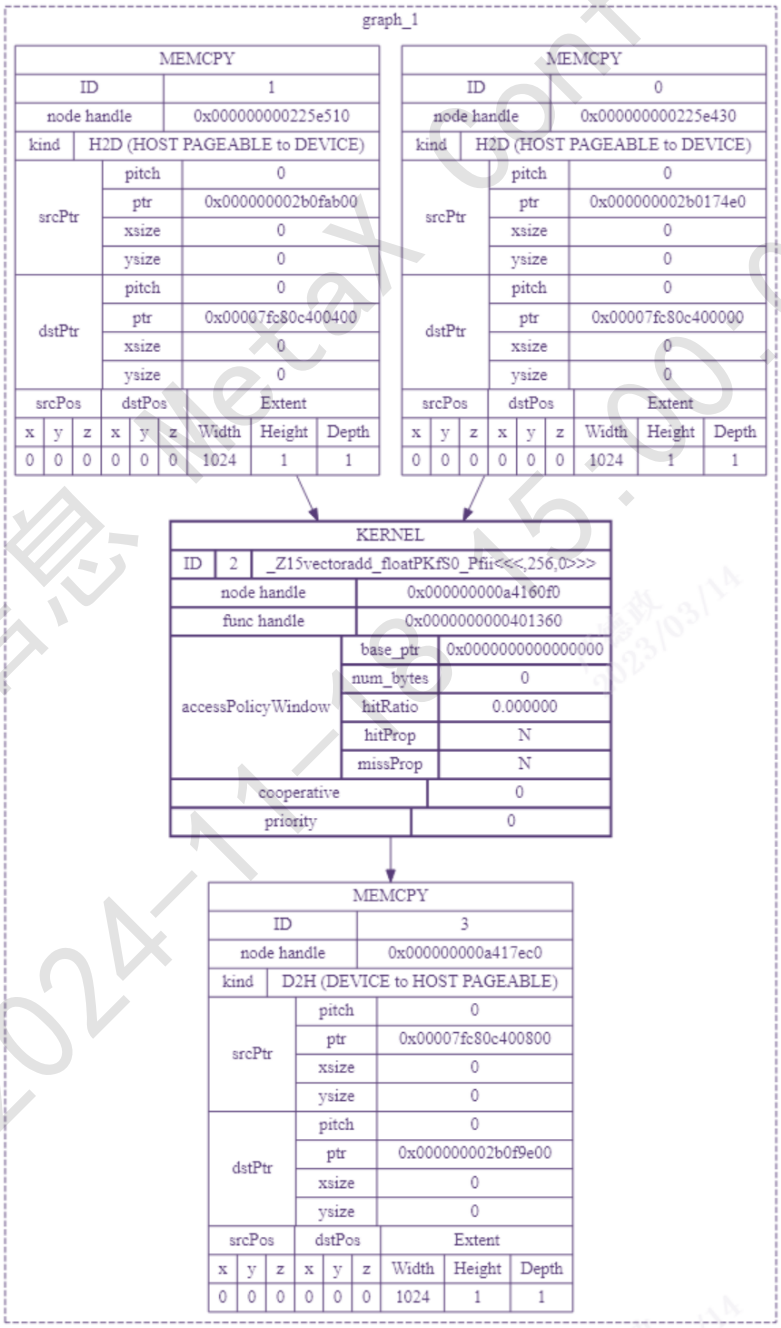

图 3-5 vectorAdd 任务图的 DOT 图

3.9 MPS 多进程服务

3.9.1 MPS 多进程服务控制

使用 MPS 多进程服务时，可以通过 `MACA_MPS_MODE` 环境变量进行控制，设定不同进程使用 GPU 计算任务队列的策略，在同一 GPU 上实现多个进程任务的并发提交和执行：

- `0`（EXCLUSIVE MODE，独占模式）：一个进程对申请到的 GPU 硬件 queue 进行独占使用。其它进程可以通过其它可用的 GPU 硬件 queue 同时向 GPU 提交工作。
- `1`（SHARED MODE，共享模式）：多个进程可以同时使用共享的 GPU 硬件 queue，通过一个或多个共享的 GPU 硬件 queue 向 GPU 提交工作。

3.9.2 MPS 多进程服务行为说明

无论是在独占模式还是共享模式下，MPS 多进程服务都支持显存保护，错误隔离和提前终止，其行为特点如下：

- 显存保护
  - MPS 为每个用户进程分配了完全隔离的 GPU 地址空间。
  - 当其他进程的内核函数尝试通过指针访问显存时，任何越界写入都不会影响当前进程的显存状态，也不会引发错误。
  - 内核函数通过指针进行的超出范围的读取操作，无法访问另一个进程修改的显存，同样不会引发错误或导致未定义行为。
  - 使用 MXMACA API（如 `mcMemcpy`）不会导致显存越界访问，因为 MPS 会限制这类操作访问其他 MPS 用户进程的显存。
- 有限的错误隔离

  如果一个 MPS 用户进程产生了致命的 GPU 异常（例如，触发了陷阱），这可能会影响到其他共享同一 GPU 硬件资源的用户进程。然而，不共享 GPU 硬件资源的进程则不会受到影响。
- 提前终止

  支持通过 Ctrl+C 或信号来终止 MPS 用户进程。这可能会暂时影响到使用共享 GPU 硬件资源的相关进程。

MPS 多进程服务目前仅支持 MXMACA 驱动的默认 QoS 算法。

此外，当一个进程使用多个优先级的流时，不同优先级的流之间的调度策略会有所不同：

- 在独占模式下，进程内不同优先级的流会严格按照优先级进行区分。也就是说，只要高优先级流中还有未完成的任务，低优先级流中的任务就不会被硬件调度执行。
- 在共享模式下，进程内不同优先级的流会通过软件权重来区分优先级。即使高优先级流中有许多未完成的任务，低优先级流中的任务也有一定的机会被 GPU 硬件服务。

3.9.3 MPS 多进程服务使用限制

MPS 多进程服务仅支持 Linux 平台，需要系统支持统一虚拟寻址（UVA）。曦云系列 GPU 均支持 UVA 且默认开启。

- 如果 Linux 内核没有启用 `CONFIG_DRM_SCHED`，则 `MACA_MPS_MODE` 仅支持 `0`（独占模式）。即使用户配置该环境变量为 `1`（共享模式），MXMACA 驱动仍将采用独占模式。
- 如果 Linux 内核支持 `DRM_SCHED`，但是 metax.ko 加载过程出现 `unknown symbol drm_sched_xxx` 的字样，是由于系统缺少一些组件导致，需要安装 **linux-modules-extra**。

  目前不支持 `DRM_SCHED` 的 Linux 内核有：

  - ucloud RHEL7 5.10.0-19-el7.ucloud.x86_64
  - alios7 4.19.91-007.ali4000.alios7.x86_64

4. 编译和调试

4.1 离线编译和静态运行

4.1.1 Makefile 编译和示例

以图 4-1 所示的一个简单 MXMACA 源代码项目文件目录为例：

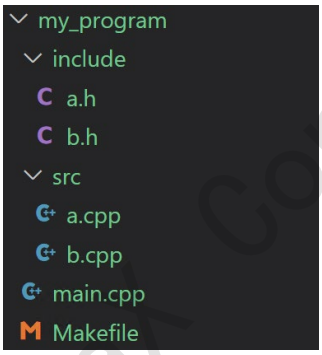

图 4-1 一个简单的 MXMACA 源代码项目文件目录

```
//a.cpp:
#include <mc_runtime.h>
#include <string.h>
extern "C"  __global__  void vector_add(int *A_d, size_t num)
{
      size_t offset = (blockIdx.x * blockDim.x + threadIdx.x);
      size_t stride = blockDim.x * gridDim.x;
      for (size_t i = offset; i < num; i += stride) {
            A_d[i]++;
      }
}
void func_a()
{
      size_t arrSize = 100;
      mcDeviceptr_t a_d;
      int *a_h = (int *)malloc(sizeof(int) * arrSize);
      memset(a_h, 0, sizeof(int) * arrSize);
      mcMalloc(&a_d, sizeof(int) * arrSize);
      mcMemcpyHtoD(a_d, a_h, sizeof(int) * arrSize);
      vector_add<<<1, arrSize>>>(reinterpret_cast<int *>(a_d), arrSize);
      mcMemcpyDtoH(a_h, a_d, sizeof(int) * arrSize);
      bool resCheck = true;
```

```
      for (int i; i < arrSize; i++) {
            if (a_h[i] != 1){
                  resCheck = false;
            }
      }
      printf("vector add result: %s\n", resCheck ? "success": "fail");
      free(a_h);
      mcFree(a_d);
}

//a.h:
extern void func_a();
```

```
//b.cpp:
#include<mc_runtime.h>
__global__ void kernel_b()
{
/* kernel code*/
}
void func_b()
{
      /* launch kernel */
      kernel_b<<<1, 1>>>();
}

//b.h:
extern void func_b();
```

```
//main.cpp:
#include <stdio.h>
#include  “a.h”
#include  “b.h”
int main()
{
      func_a();
      func_b();
      printf( “my program!\n” );
      return 1;
}
```

上述工程中包含 **main.cpp**，**a.cpp**，**b.cpp** 三个源文件，其中 **a.cpp** 中包含 `vector_add` 核函数，**b.cpp** 中包含 `kernel_b` 核函数。若要将该工程编译成可执行文件，可按照如下方法编写 Makefile 文件：

Makefile 文件：

```
# MXMACA Compiler
MXCC = $(MACA_PATH)/mxgpu_llvm/bin/mxcc

# Compiler flags
MXCCFLAGS = -xmaca

# Source files
SRCS= main.cpp src/a.cpp src/b.cpp

# Object files
OBJS = $(SRCS:.cpp=.o)

# Executable
EXEC = my_program

# Default target
all: $(EXEC)

# Link object files to create executable
$(EXEC): $(OBJS)
$(MXCC) $(OBJS) -o $(EXEC)

%.o: %.cpp
$(MXCC) $(MXCCFLAGS) -c $< -o $@ -I include

# clean up object files and executable
clean:
rm -f $(OBJS) $(EXEC)
```

值得注意的是，执行 make 命令之前，需要正确设置环境变量，以缺省安装位置（/opt/maca）为例：

```
$ export MACA_PATH=/opt/maca
$ export LD_LIBRARY_PATH=${MACA_PATH}/lib:${LD_LIBRARY_PATH}
```

然后在 Makefile 同级目录下执行 make 命令，如图 4-2 所示，就能得到可执行程序 my_program。该工程中，源文件 a.cpp 例举了一种利用核函数实现向量加法的典型用法。

```
(base) sw@LG-PC-10-2-120-162:~/my_program$ tree -L 2
.
├── include
│   ├── a.h
│   └── b.h
├── main.cpp
├── Makefile
└── src
    ├── a.cpp
    └── b.cpp

2 directories, 6 files
(base) sw@LG-PC-10-2-120-162:~/my_program$ export MACA_PATH=/opt/maca
(base) sw@LG-PC-10-2-120-162:~/my_program$ export LD_LIBRARY_PATH=${MACA_PATH}/lib:${LD_LIBRARY_PATH}
(base) sw@LG-PC-10-2-120-162:~/my_program$ make
/opt/maca/mxgpu_llvm/bin/mxcc -xmaca -c main.cpp -o main.o -I include
/opt/maca/mxgpu_llvm/bin/mxcc -xmaca -c src/a.cpp -o src/a.o -I include
/opt/maca/mxgpu_llvm/bin/mxcc -xmaca -c src/b.cpp -o src/b.o -I include
/opt/maca/mxgpu_llvm/bin/mxcc main.o src/a.o src/b.o -o my_program
(base) sw@LG-PC-10-2-120-162:~/my_program$ ./my_program
vector add result: success
my program!
(base) sw@LG-PC-10-2-120-162:~/my_program$
```

图 4-2 Makefile 编译和示例

4.1.2 CMake 编译和示例

继续以图 4-1 所示的项目结构为例，如果用 cmake 工具来构建项目，则需在 **main.cpp** 同级目录下创建 **CMakeLists.txt** 文件，可以按照如下方式编写 **CMakeLists.txt** 文件：

```
# Specify the minimum CMake version required
cmake_minimum_required(VERSION 3.0)

# Set the project name
project(my_program)

# Set the path to the compiler
set(MXCC_PATH $ENV{MACA_PATH})
set(CMAKE_CXX_COMPILER ${MXCC_PATH}/mxgpu_llvm/bin/mxcc)

# Set the compiler flags
set(MXCC_COMPILE_FLAGS -x maca)
add_compile_options(${MXCC_COMPILE_FLAGS})

# Add source files
File(GLOB SRCS src/*.cpp main.cpp)
add_executable(my_program ${SRCS})

# Set the include paths
target_include_directories(my_program PRIVATE include)
```

同理，cmake 之前也需要正确设置环境变量，以缺省安装位置（/opt/maca）为例：

```
$ export  MACA_PATH=/opt/maca
$ export  LD_LIBRARY_PATH=${MACA_PATH}/lib:${LD_LIBRARY_PATH}
```

如图 4-3 所示，在 CMakeLists.txt 同级目录下创建 build 文件夹，进入 build 目录执行 cmake 命令，再执行 make 命令，即可得到可执行程序 my_program。

```
(base) sw@LG-PC-10-2-120-162:~/my_program$ cat CMakeLists.txt
# Specify the minimum CMake version required
cmake_minimum_required(VERSION 3.0)

# Set the project name
project(my_program)

# Set the path to the compiler
set(MXCC_PATH $ENV{MACA_PATH})
set(CMAKE_CXX_COMPILER ${MXCC_PATH}/mxgpu_llvm/bin/mxcc)

# Set the compiler flags
set(MXCC_COMPILE_FLAGS -x maca)
add_compile_options(${MXCC_COMPILE_FLAGS})

# Add source files
File(GLOB SRCS src/*.cpp main.cpp)
add_executable(my_program ${SRCS})

# Set the include paths
target_include_directories(my_program PRIVATE include)

(base) sw@LG-PC-10-2-120-162:~/my_program$ export  MACA_PATH=/opt/maca
(base) sw@LG-PC-10-2-120-162:~/my_program$ export  LD_LIBRARY_PATH=${MACA_PATH}/lib:${LD_LIBRARY_PATH}
(base) sw@LG-PC-10-2-120-162:~/my_program$ mkdir build
(base) sw@LG-PC-10-2-120-162:~/my_program$ cd build/
(base) sw@LG-PC-10-2-120-162:~/my_program/build$ cmake ..
-- The C compiler identification is GNU 7.5.0
-- The CXX compiler identification is GNU 7.5.0
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: /usr/bin/cc - skipped
-- Detecting C compile features
-- Detecting C compile features - done
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++ - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: /home/sw/my_program/build
(base) sw@LG-PC-10-2-120-162:~/my_program/build$ make
Scanning dependencies of target my_program
[ 25%] Building CXX object CMakeFiles/my_program.dir/main.cpp.o
[ 50%] Building CXX object CMakeFiles/my_program.dir/src/a.cpp.o
[ 75%] Building CXX object CMakeFiles/my_program.dir/src/b.cpp.o
[100%] Linking CXX executable my_program
[100%] Built target my_program
(base) sw@LG-PC-10-2-120-162:~/my_program/build$ ./my_program
vector add result: success
my program!
(base) sw@LG-PC-10-2-120-162:~/my_program/build$
```

图 4-3 CMake 编译和示例

4.2 运行时编译和动态加载

曦云系列 GPU 支持运行时编译和动态加载功能，整个流程如图 4-4 所示。其中运行时进行实时编译（Just-In-Time, JIT）采用了 LLVM bitcode 格式文件，有关该格式的详细内容，参见官方文档介绍 https://www.llvm.org/docs/BitCodeFormat.html。

用户在使用时通过引入头文件 mcrtc.h，即可使用 MCRTC 提供的所有功能：

- MCRTC 将原始的 C++语法的 MXMACA 代码，通过 mcrtcGetBitCode 接口编译生成 bitcode 格式的二进制代码。
- 将生成的 bitcode 代码，通过 mcModuleLoad 进行加载，曦云系列 GPU 的驱动（运行时库 API）会继续进行后续编译并生成设备侧的可执行代码，用户在调用 mcModuleLaunchKernel API 接口的时候会将这些设备侧可执行代码送入 GPU 执行。

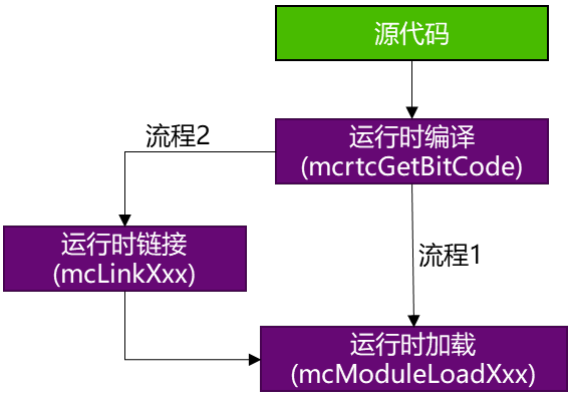

图 4-4 即时编译流程

代码示例

用户可以将 device 代码和 host 代码分别写在不同的文件中，生成可执行程序时，只编译 host 代码，device 代码在程序运行时编译。以下介绍了这种编程范式的简单实现。

1. device 代码写在单独的文件中：

```
//my_kernel.cu:
extern “C” __global__ void test_kernel()
{
  /* kernel code */
  printf( “my kernel\n” );
}
```

2. host 代码写在另外的文件中：

```
host 文件 rtc_test.cpp:
#include <fstream>
#include <vector>
#include<mc_runtime.h>
#include<mcrtc.h>
static inline std::vector<char> load_file_data(const char *filename)
{
  std::ifstream file(filename, std::ios::binary | std::ios::ate);
  std::streamsize fsize = file.tellg();
  file.seekg(0, std::ios::beg);
  std::vector<char> buffer(fsize + 1);
  file.read(buffer.data(), fsize);
  buffer[fsize] = ‘\x0’ ;
  file.close();
  return buffer;
}
```

```
void rtcTest()
{
  /* load kernel file to buffer*/
  std::vector<char> buffer = load_file_data( “my_kernel.cu” );
  /* Create an instance of mcrtcProgram */
  mcrtcProgram prog;
  mcrtcCreateProgram(    &prog,                        // prog
                    (char *)&buffer[0],    // buffer
                      “” ,              //name
                    0,                  // numHeaders
                    NULL,               //headers
                    NULL);                    //includeNames
  const char *opts[] = {"-xmaca"};
  /* Compile the program */
  mcrtcCompileProgram(prog,          // prog
                        1,            // numOptions
                        opts);        // options
  size_t codeSize;
  mcrtcGetCodeSize(prog, &codeSize);
  char *code = new char[codeSize];
  /* get binary file */
  mcrtcGetCode(prog, code);
  mcrtcDestroyProgram(&prog);
  mcModule_t module;
  mcFunction_t kernel_addr;
  /* load binary file to buffer */
  mcModuleLoadData(&module, code);
  /* get kernel function point */
  mcModuleGetFunction(&kernel_addr, module, "test_kernel");
  /* launch kernel function */
  mcModuleLaunchKernel(kernel_addr, 1, 1, 1, 1, 1, 1, 0,
NULL,NULL,NULL);
  mcModuleUnload(module);
  delete[] code;
}
int main()
{
  rtcTest();
  return 1;
}
```

3. 正确设置环境变量（以缺省安装位置/opt/maca 为例）：

```
$ export MACA_PATH=/opt/maca
```

```
$ export LD_LIBRARY_PATH=${MACA_PATH}/lib:${LD_LIBRARY_PATH}
$ export PATH=${MACA_PATH}/mxgpu_llvm/bin:${PATH}
```

4. 在源文件 **rtc_test.cpp** 同级目录下执行以下命令，即可得到可执行文件 **a.out**，如图 4-5 所示。

```
mxcc -xmaca rtc_test.cpp
```

此时 device 代码 **my_kernel.cu** 并没有编译到 **a.out** 中，而是在运行 **a.out** 过程中编译该 device 文件。

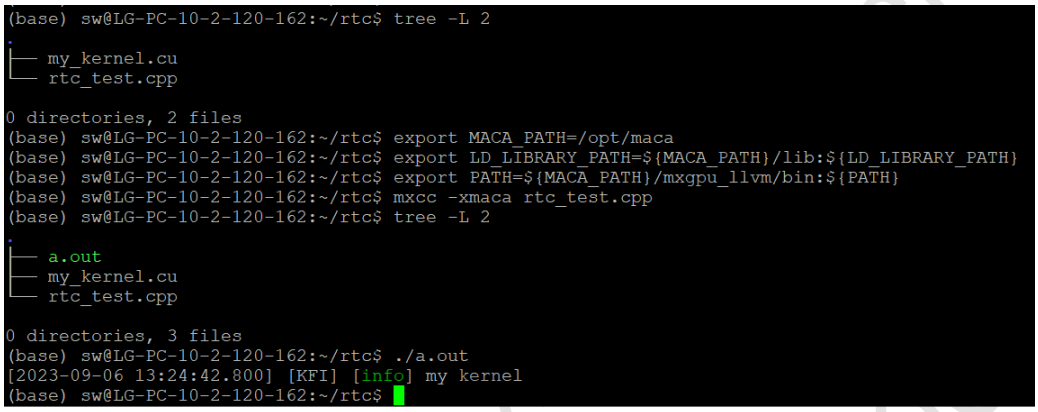

图 4-5 可执行文件 a.out 获取

4.3 代码托管（Binary Cache）

使用默认设置 binary cache（cache 文件保存路径以及所支持文件大小均采用默认值，无需设置环境变量）。

代码示例：

```
#include<mc_runtime.h>

__global__ void vectorADD(const float* A_d, const float* B_d, float* C_d,
size_t NELEM) {
  size_t offset = (blockIdx.x * blockDim.x + threadIdx.x);
  size_t stride = blockDim.x * gridDim.x;

  for (size_t i = offset; i < NELEM; i += stride) {
    C_d[i] = A_d[i] + B_d[i];
  }
}

int main()
{
  int blocks=20;
  int threadsPerBlock=1024;
```

```
int numSize=1024*1024;

  float *A_d=nullptr;
  float *B_d=nullptr;
  float *C_d=nullptr;

  float *A_h=nullptr;
  float *B_h=nullptr;
  float *C_h=nullptr;

  mcMalloc((void**)&A_d,numSize*sizeof(float));
  mcMalloc((void**)&B_d,numSize*sizeof(float));
  mcMalloc((void**)&C_d,numSize*sizeof(float));

  A_h=(float*)malloc(numSize*sizeof(float));
  B_h=(float*)malloc(numSize*sizeof(float));
  C_h=(float*)malloc(numSize*sizeof(float));

  for(int i=0;i<numSize;i++)
  {
    A_h[i]=3;
    B_h[i]=4;
    C_h[i]=0;
  }

  mcMemcpy(A_d,A_h,numSize*sizeof(float),mcMemcpyHostToDevice);
  mcMemcpy(B_d,B_h,numSize*sizeof(float),mcMemcpyHostToDevice);

  vectorADD<<<dim3(blocks),dim3(threadsPerBlock)>>>(A_d,B_d,C_d,numSize);

  mcMemcpy(C_h,C_d,numSize*sizeof(float),mcMemcpyDeviceToHost);

  mcFree(A_d);
  mcFree(B_d);
  mcFree(C_d);

  free(A_h);
  free(B_h);
  free(C_h);

  return 0;
}
```

此示例每个 block 的线程数量大于 512，会触发重编译动作，因此也会触发 binary cache 功能，编译运行。然后执行以下命令：

```
$ cd ~/.metax/shadercache/
$ ls -l
```

结果如下图所示，可以看到编译运行生成的 cache 文件，命名规则和 4.2 运行时编译和动态加载描述一致。

```
(base) sw@LG-PC-10-2-120-162:~/binary_cache$ ls -lt
total 4
-rw-rw-r-- 1 sw sw 1254 Sep  6 13:27 vectorAdd.cpp
(base) sw@LG-PC-10-2-120-162:~/binary_cache$ mxcc -x maca vectorAdd.cpp -o vectorAdd
(base) sw@LG-PC-10-2-120-162:~/binary_cache$ ls -lt
total 52
-rwxrwxr-x 1 sw sw 46392 Sep  6 13:37 vectorAdd
-rw-rw-r-- 1 sw sw  1254 Sep  6 13:27 vectorAdd.cpp
(base) sw@LG-PC-10-2-120-162:~/binary_cache$ cd ~/.metax/shadercache/
(base) sw@LG-PC-10-2-120-162:~/.metax/shadercache$ ls -lt
total 29816
-rw-rw-r-- 1 sw   sw      73576 Sep  5 23:30 c3c8d64032995488473c269e846d638c_9f247c80fb.cache
```

图 4-6 Binary Cache 生成

4.3.1 更改 Binary Cache 文件支持 Size

需要设置环境变量：MACA_CACHE_MAXSIZE

1. 编辑~/.bashrc 文件：

```
$ vim ~/.bashrc
```

2. 在文件末尾加入以下内容：

```
export MACA_CACHE_MAXSIZE = xxx // 所设置的文件大小，单位字节
```

3. 保存后执行以下命令：

```
$ source ~/.bashrc
```

4.3.2 自定义 Cache 文件路径

需要设置环境变量：MACA_CACHE_PATH

1. 编辑~/.bashrc 文件

```
$ vim ~/.bashrc
```

2. 在文件末尾加入以下内容：

```
export MACA_CACHE_PATH=your/specific/path  //用户自定义路径
```

4.3.3 关闭 Binary Cache 功能

1. 编辑~/.bashrc 文件

```
$ vim ~/.bashrc
```

2. 在文件末尾加入以下内容：

```
export MACA_CACHE_DISABLE=1
```

说明

将 MACA_CACHE_DISABLE 的值改为 0 或者不设置即可重新启用 binary cache 功能。

4.4 环境变量

曦云系列 GPU 支持在程序启动前用环境变量管控一些程序运行行为，运行时编译和动态加载功能。支持的环境变量，参见表 4-1。

表 4-1 环境变量

环境变量	可设置值	注释
设备枚举和属性控制		
MACA_VISIBLE_DEVICES	GPU UUID、Device Node ID	可以指定 GPU UUID，例如 export MACA_VISIBLE_DEVICES= GPU-ad2367dd-a40e-6b86-6fc3-c44a2cc92c7e；也可以指定设备节点 ID，例如 export MACA_VISIBLE_DEVICES=0,2。
MACA_DEVICE_ORDER	FASTEST_FIRST、PCI_BUS_ID（默认为 FASTEST_FIRST）	FASTEST_FIRST：根据设备计算能力从快到慢排序。 PCI_BUS_ID：根据 PCI 总线 ID 升序排列设备。
编译控制		
MACA_CACHE_DISABLE	0 或 1（默认为 0）	如果设置为 1，则禁用 binary cache。 设置为 0 或不设置时，启用 binary cache。
MACA_CACHE_PATH	filepath	指定 cache 文件存储位置。 当未设置时，cache 文件存储在默认目录下（<user home>/.metax/shadercache/）。
MACA_CACHE_MAXSIZE	integer（desktop/server 平台默认为 268435456（256 MB）且最大为 4294967296（4 GB））	指定能缓存的单个 cache 文件的最大 size。 当生成的 cache 文件超过这个值时，则不进行缓存。 当不设置时，默认为 256 MB。如果设置超过 4 GB，则只按 4 GB 生效。
执行控制		
MACA_LAUNCH_BLOCKING	0 或 1（默认为 0）	设置为 1 时，Stream 上启动内核表现为同步。 设置为 0 时，Stream 上启动内核表现为异步。
MACA_TRAP_HANDLER	0-2（默认为 1）	设置为 0 时，关闭 trap 上报功能，trap 指令会替换成 snop 指令继续执行至 kernel 完成。

环境变量	可设置值	注释
		设置为 1 时，开启 trap 上报功能，仅对 Fatal 类异常进行上报。 设置为 2 时，开启 trap 上报功能，对 Fatal 类异常和 Numeric 类异常都进行上报。
MACA_MPS_MODE	0 或 1（默认为 0）	设置为 1 时，多个进程可以同时使用共享的 GPU 硬件 queue，通过一个或多个共享的 GPU 硬件 queue 向 GPU 提交工作。 设置为 0 时，一个进程对申请到的 GPU 硬件 queue 进行独占使用。其它进程可以通过其它可用的 GPU 硬件 queue 同时向 GPU 提交工作。
MACA_GRAPH_LAUNCH_MODE	0 或 1（默认为 0）	设置为 1 时，图编程的任务提交采用任务图提交模式。 设置为 0 时，图编程的任务提交采用标准核函数提交模式。
pri_mem_size	0-36（默认为 4），单位为 KB	内核态环境变量，对服务器上的所有用户进程生效。在 `insmod ko` 时，使用： `insmod metax.ko pri_mem_size= XX` （`XX` 为需要设置的 private memory size，单位为 KB）

4.5 主机代码调试信息

曦云系列 GPU 支持通过环境变量 `MXLOG_LEVEL` 设置 MXMACA 驱动软件的日志输出等级，可选等级如下：

- `off`：关闭日志输出
- `error`：仅打印 error 级别日志
- `warning`：输出 warning 及 error 级别日志
- `info`：输出 information、warning 及 error 级别日志
- `debug`：输出全部日志

曦云系列 GPU 上编程，既支持直接使用原生的 **printf** 自行设计和管理，也可以借助 MXMACA 驱动软件的日志管理方案，mcanalyzer 动态库，提供 `mcLog` API，如图 4-7 所示：

- 使用 `mcLog` API 之前，需要 include 它的头文件，**mxlog.h** 位于 MXMACA 软件包成功安装后 **include** 目录的子目录 **mxlog**；
- 根据应用程序对于日志等级的定义，选用相应的 `mcLog` API：`LOGE/LOGW/LOGI/LOGD/LOGV`；

- 使用 mxcc 编译时，需要指定`-lmcanalyzer` 编译选项；
- 使用环境变量`MXLOG_LEVEL`，可以设置日志输出最低等级。如果该环境变量未设置，MXMACA会使用一个缺省的日志输出最低等级，一般是 `error` 或者 `info`。

```
(base) sw@LG-PC-10-2-120-162:~/mclog_demo$ ls
mcLogDemo.cpp
(base) sw@LG-PC-10-2-120-162:~/mclog_demo$ cat mcLogDemo.cpp
#include <mclog/mclog.h>

int main()
{
    LOGI("#include <mclog/mclog.h> before using mcLog APIs\n");
    LOGI("Compiling with -lmcanalyzer option additionally\n");
    LOGI("export MXLOG_LEVEL=debug in case debug logs are expected\n");

    LOGE("Demo:use LOGE for logs with level = error\n");
    LOGW("Demo:use LOGW for logs with level = warning\n");
    LOGI("Demo:use LOGI for logs with level = info\n");
    LOGD("Demo:use LOGD for logs with level = debug\n");
    exit(0);
}
(base) sw@LG-PC-10-2-120-162:~/mclog_demo$ mxcc -x maca mcLogDemo.cpp -o mcLogDemo -lmcanalyzer
(base) sw@LG-PC-10-2-120-162:~/mclog_demo$ export MXLOG_LEVEL=debug
(base) sw@LG-PC-10-2-120-162:~/mclog_demo$ ls
mcLogDemo  mcLogDemo.cpp
(base) sw@LG-PC-10-2-120-162:~/mclog_demo$ ./mcLogDemo
[2023-09-06 14:25:28.241] [APP] [info] #include <mclog/mclog.h> before using mcLog APIs
[2023-09-06 14:25:28.241] [APP] [info] Compiling with -lmcanalyzer option additionally
[2023-09-06 14:25:28.241] [APP] [info] export MXLOG_LEVEL=debug in case debug logs are expected
[2023-09-06 14:25:28.241] [APP] [error] Demo:use LOGE for logs with level = error
[2023-09-06 14:25:28.241] [APP] [warning] Demo:use LOGW for logs with level = warning
[2023-09-06 14:25:28.241] [APP] [info] Demo:use LOGI for logs with level = info
[2023-09-06 14:25:28.241] [APP] [debug] Demo:use LOGD for logs with level = debug
(base) sw@LG-PC-10-2-120-162:~/mclog_demo$
```

图 4-7 mcanalyzer 动态库提供的 mcLog API 示例

4.6 设备代码调试信息

4.6.1 使用 GPU printf

当需要在设备侧代码输出调试信息时，曦云系列 GPU 支持在设备核函数里面使用 `printf` 函数打印相关信息。

代码示例

使用 GPU printf，示例代码如下：

```
__global__ void vectorAdd(const float* a, const float* b, float* c, int
width, int height)
{
    int x = blockDim.x * blockIdx.x + threadIdx.x;
    int y = blockDim.y * blockIdx.y + threadIdx.y;

    int i = x;
    if (i < (width * height))
    {
        c[i] = a[i] + b[i];
```

```
        printf("c[%d]:%f \n", i, c[i]);
}
```

在 kernel 里面添加了 printf 后，执行程序，得到输出结果部分如图 4-8 所示。

```
(base) sw@LG-PC-10-2-120-162:~/dyan/samples/0_Introduction/vectorAdd$ ./vectorAdd
[Vector addition of 500 elements]
Copy input data from the host memory to the MACA device
MACA kernel launch with 2 blocks of 256 threads
Copy output data from the MACA device to the host memory
[2023-09-06 14:33:45.075] [KFI] [info] c[128]:1.059104
[2023-09-06 14:33:45.075] [KFI] [info] c[129]:0.942651
[2023-09-06 14:33:45.075] [KFI] [info] c[130]:1.605671
[2023-09-06 14:33:45.075] [KFI] [info] c[131]:0.910396
[2023-09-06 14:33:45.075] [KFI] [info] c[132]:0.620085
[2023-09-06 14:33:45.075] [KFI] [info] c[133]:1.137881
[2023-09-06 14:33:45.075] [KFI] [info] c[134]:0.779865
[2023-09-06 14:33:45.075] [KFI] [info] c[135]:0.472877
[2023-09-06 14:33:45.075] [KFI] [info] c[136]:1.169902
[2023-09-06 14:33:45.075] [KFI] [info] c[137]:0.955334
[2023-09-06 14:33:45.075] [KFI] [info] c[138]:1.064147
[2023-09-06 14:33:45.075] [KFI] [info] c[139]:0.519393
[2023-09-06 14:33:45.075] [KFI] [info] c[140]:0.987942
[2023-09-06 14:33:45.075] [KFI] [info] c[141]:1.158870
[2023-09-06 14:33:45.075] [KFI] [info] c[142]:0.624698
[2023-09-06 14:33:45.075] [KFI] [info] c[143]:1.877567
[2023-09-06 14:33:45.075] [KFI] [info] c[144]:1.463947
[2023-09-06 14:33:45.075] [KFI] [info] c[145]:0.806929
[2023-09-06 14:33:45.075] [KFI] [info] c[146]:1.158105
[2023-09-06 14:33:45.075] [KFI] [info] c[147]:1.859324
[2023-09-06 14:33:45.075] [KFI] [info] c[148]:1.065465
```

图 4-8 printf 输出结果

从图 4-8 中我们可以看出，当在核函数里面使用 `printf` 输出结果时，一个线程束里面 64 个线程将按照顺序依次打印结果，但是无法保证线程束与线程束之间的打印顺序。因此，我们不能用打印信息的顺序来体现程序的执行顺序。

4.6.2 使用 GPU Trap Handler

曦云架构 GPU 的硬件支持将 Shader 执行期间产生的异常写进 TRAP_STATUS 寄存器中，并且能选择性地在异常发生时插入一条 trap 指令。Trap 指令会使 Shader 以更高的特权级别执行 trap kernel，在 trap kernel 中可以定义和决定如何处理该异常，处理流程如图 4-9 所示。

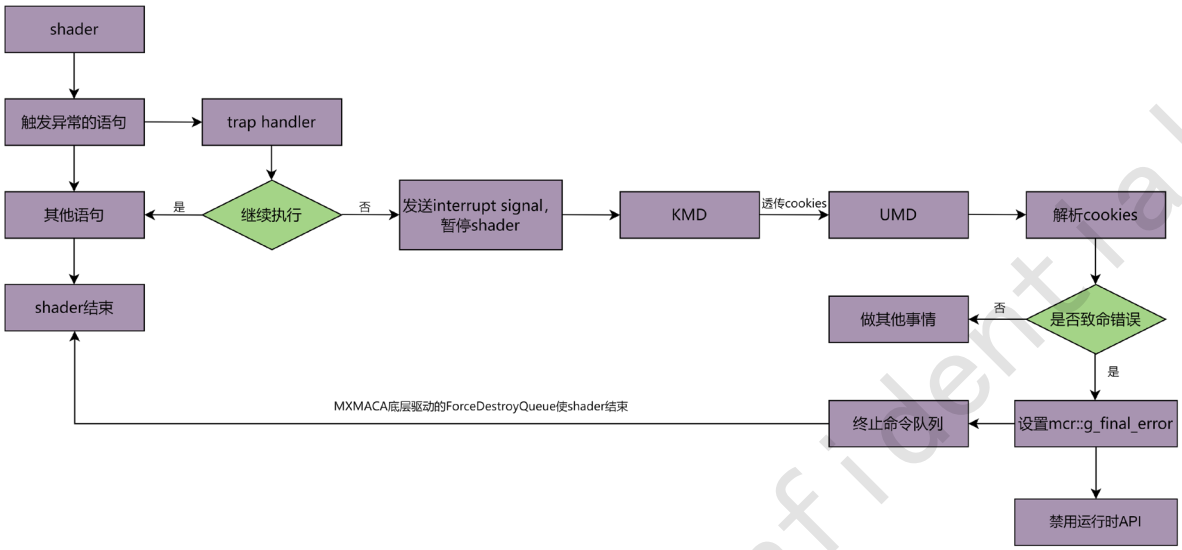

图 4-9 GPU Trap Handler 处理流程

基于 trap kernel 功能，当用户层程序触发相应异常信号时，MXMACA 将进行相应处理并给出相关提示信息以供用户进行程序代码调试。

代码示例

使用 GPU trap handler，示例代码如下所示：

```
#include<mc_runtime.h>

typedef struct
{
  alignas(4)float f;
  double d;
}__attribute__((packed)) test_type_mem_violation;

__global__ void trigger_memory_violation(test_type_mem_violation *dst)
{
  atomicAdd(&dst->f,1.23);
  atomicAdd(&dst->d,20);
  dst->f=9.8765;
}

int main()
{
  test_type_mem_violation hd={0};
  test_type_mem_violation *ddd;
  mcMalloc((void**)&ddd,sizeof(test_type_mem_violation));
  mcMemcpy(ddd,&hd,sizeof(test_type_mem_violation),mcMemcpyHostToDevice);
```

```
  trigger_memory_violation<<<dim3(1),dim3(1)>>>(ddd);
  mcMemcpy(&hd,ddd,sizeof(test_type_mem_violation),mcMemcpyDeviceToHost);
  mcFree(ddd);
  return 0;
}
```

如果运行以上代码将会得到以下错误信息：

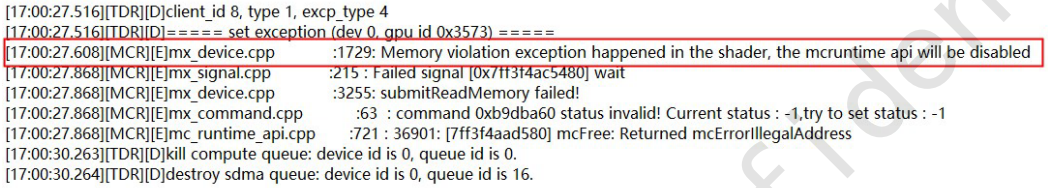

图 4-10 trap 错误信息

重点关注红框内给出信息，MCR 层给出提示信息“**[MCR][E]mx_device.cpp:1729:Memory violation exception happened in the shader, mcruntime api will be disabled**”并设置了 `g_final_error`，对应异常类型来看：内核函数触发了异常类型 **3—Memory violation**，对照触发异常条件可以知道，我们的核函数代码可能存在访问内存的偏移量小于 0，越界或数据未按要求对齐。

检查代码可发现：由于在结构体 `test_type_mem_violation` 中 `alignas(4)`对 float 进行强制对齐以及对结构体进行了 `attribute((packed))`，所以结构体 `test_type_mem_violation` 的 size 为 `double:8` 加上 `float:4` 等于 `12`，由于 64-bit 的 atomic 操作要求数据按照 8 字节对齐，故这里的结构体没有按照要求对齐，所以引发了 trap。

将程序做如下修改即可解决该问题：

```
#include<mc_runtime.h>

typedef struct
{
  float f;
  double d;
}test_type_mem_violation;

__global__ void trigger_memory_violation(test_type_mem_violation *dst)
{
  atomicAdd(&dst->f,1.23);
  atomicAdd(&dst->d,20);
  dst->f=9.8765;
}

int main()
```

```
{
  test_type_mem_violation hd={0};
  test_type_mem_violation *ddd;
  mcMalloc((void**)&ddd,sizeof(test_type_mem_violation));
  mcMemcpy(ddd,&hd,sizeof(test_type_mem_violation),mcMemcpyHostToDevice);
  trigger_memory_violation<<<dim3(1),dim3(1)>>>(ddd);
  mcMemcpy(&hd,ddd,sizeof(test_type_mem_violation),mcMemcpyDeviceToHost);
  mcFree(ddd);
  return 0;
}
```

5. 附录

5.1 术语/缩略语

术语/缩略语	全名	描述
DOT		一种图描述语言，详细信息可参见：https://www.graphviz.org/doc/info/lang.html
Grid		线程网格
IPC	Interprocess Communication	进程间通信
Makefile		描述了整个代码工程所有文件的编译顺序、编译规则。Makefile 有自己的书写格式、关键字、函数，可以使用系统 shell 所提供的任何命令来完成想要的工作
MCRTC	MetaX Runtime Compilation	用于 MXMACA C++的运行时编译库
MetaXLink		沐曦 GPU D2D 接口总线
mxcc		MXMACA 软件栈中，针对 MetaX GPU 的硬件架构和功能特性设计和发布的编译器
MXMACA	MetaX Advanced Compute Architecture	沐曦推出的 GPU 软件栈，包含了沐曦 GPU 的底层驱动、编译器、数学库及整套软件工具套件
PCIe	Peripheral Component Interconnect-Express	一种高速串行计算机扩展总线标准
SIMD	Single Instruction Multiple Data	单指令、多数据
SIMT	Single Instruction Multiple Thread	单指令、多线程
SVM	Shared Virtual Memory	共享虚拟内存

声明