曦云® 系列通用计算 GPU

mcRAND API 参考

CSRD-23016-020-F3_V03　　2024-03-15

沐曦专有和三级保密信息　　本文档受 NDA 管控

声明

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

目录

1 介绍

mcRAND 库提供的功能侧重于简单高效地生成高质量的伪随机数和准随机数。

伪随机数序列满足大部分真随机序列的统计特性，但是它是由确定性算法生成的。

n 维点的准随机数序列是由一种确定性算法生成的，该算法旨在均匀地填充一个 n 维空间。

1.1 安装

安装 MXMACA® 工具包将会把 RAND 头文件 (mcrand.h 和 mcrand_kernel.h) 以及库文件 (libmcrand.so 和 libmcrand_static.a) 复制到您系统的标准 MXMACA 包含目录中。MXMACA 工具包的安装，参见《曦云系列通用计算 GPU 快速上手指南》。安装后，确保设置了环境变量 MACA_PATH。

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

这里的 /opt/maca 路径为 MXMACA 工具包的默认安装路径。如果安装时选择了其它路径，则将 MACA_PATH 设置为该路径。以下是与 RAND API 相关的文件列表。

```
#header
${MACA_PATH}/include/mcrand/mcrand.h
${MACA_PATH}/include/mcrand/mcrand_kernel.h

#libraries
${MACA_PATH}/lib/libmcrand.so
${MACA_PATH}/lib/libmcrand_static.a
```

mcRAND 由两部分组成：主机（CPU）端的库和设备（GPU）端的头文件。

主机端的库类似于其他 CPU 库：用户需要包含头文件 ${MACA_PATH}/include/mcrand/mcrand.h 以获取函数声明，并链接到库文件。随机数可以在设备端或主机 CPU 上生成。

对于 device generation ，库的调用发生在主机上，但随机数生成的实际工作发生在设备上。生成的随机数存储在设备的全局内存中。用户可以调用自己的内核来使用这些随机数，或者将随机数复制回主机进行进一步处理。

对于 host CPU generation ，所有的工作都在主机上完成，随机数存储在主机内存中。

mcRAND 的第二部分是设备端的头文件 ${MACA_PATH}/include/mcrand/mcrand_kernel.h 。

该文件定义了用于设置随机数生成器状态和生成随机数序列的设备函数。用户代码可以包含这个头文件，并且用户编写的内核可以调用头文件中定义的设备函数。这使得随机数可以被生成并立即被用户内核使用，而不需要将随机数写入全局内存然后再读取。

1.2 Hello mcRAND

以下是一个使用 mcRAND API 的简单示例：

```
/*
* 这个程序使用主机 MCRAND API 生成 100 个随机数。
* 伪随机浮点数。
*/
#include <cstdio>
#include <cstdlib>
#include <cstring>
#include <stdexcept>
#include <vector>
#include <mc_runtime.h>
#include <mcrand.h>

int main(int argc, char *argv[]) {

    mcStream_t stream = NULL;
    mcrandGenerator_t gen = NULL;
    mcrandOrdering_t order = MCRAND_ORDERING_PSEUDO_BEST;

    const int n = 10;
    const unsigned long long offset = 0ULL;
    const unsigned long long seed = 1234ULL;
    const float mean = 1.0f;
    const float stddev = 2.0f;

    mcStreamCreateWithFlags(&stream, mcStreamNonBlocking);

    std::vector<float> h_data(n, 0);
    float *d_data = nullptr;
    mcMalloc(reinterpret_cast<void **>(&d_data),
               sizeof(float) * h_data.size());

    mcrandCreateGenerator(&gen, MCRAND_RNG_PSEUDO_XORWOW);
    mcrandSetStream(gen, stream);
    mcrandSetGeneratorOffset(gen, offset);
    mcrandSetGeneratorOrdering(gen, order);
    mcrandSetPseudoRandomGeneratorSeed(gen, seed);

    mcrandGenerateLogNormal(gen, d_data, h_data.size(), mean, stddev);

    mcMemcpyAsync(h_data.data(), d_data,
                    sizeof(float) * h_data.size(),
                    mcMemcpyDeviceToHost, stream);
    mcStreamSynchronize(stream);

    mcFree(d_data);
    mcrandDestroyGenerator(gen);
    mcStreamDestroy(stream);
    mcDeviceReset();

    return EXIT_SUCCESS;
}
```

假设上面的示例包含在名为 myMcrandApp.cu 的文件中，在 Linux 上使用 mxcc 编译器对其进行编译，并链接到动态库，可以使用以下命令：

```
export MACA_PATH=/opt/maca
export MACA_CLANG_PATH=${MACA_PATH}/mxgpu_llvm/bin
```

(下页继续)

(续上页)

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

${MACA_CLANG_PATH}/mxcc myMcrandApp.cu \
   -I${MACA_PATH}/include/mcrand \
   -lmcrand -o myMcrandApp
```

这里的 /opt/maca 路径为 MXMACA 工具包的默认安装路径。如果安装时选择了其它路径，则将 MACA_PATH 设置为该路径。而要编译针对静态 mcRAND 库的应用程序，则需要使用以下命令：

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

${MACA_CLANG_PATH}/mxcc myMcrandApp.cu \
   -I${MACA_PATH}/include/mcrand \
   -lmcrand_static -o myMcrandApp
```

这里的 /opt/maca 路径为 MXMACA 工具包的默认安装路径。如果安装时选择了其它路径，则将 MACA_PATH 设置为该路径。

2 主机 API 概述

要使用主机 API，用户代码应该包含库的头文件 mcrand.h，并且动态链接到 mcRAND 库。该库使用 MXMACA 运行时，因此用户代码也必须使用该运行时。

随机数是由生成器产生的。在 mcRAND 中的生成器封装了生成伪随机数或准随机数序列所需的所有内部状态。通常的操作顺序如下：

1 使用 mcrandCreateGenerator() 创建所需类型的新生成器（参见生成器类型）。

2 设置生成器选项（参见生成器选项）；例如，使用 mcrandSetPseudoRandomGeneratorSeed() 设置种子。

3 使用 mcMalloc() 在设备上分配内存。

4 使用 mcrandGenerate() 或其他生成函数生成随机数。

5 使用生成的结果。

6 如果需要，可以通过多次调用 mcrandGenerate() 生成更多的随机数。

7 使用 mcrandDestroyGenerator() 进行清理。

为了在主机 CPU 上生成随机数，在上述步骤 1 中调用 mcrandCreateGeneratorHost() 函数，并在步骤 3 中分配一个主机内存缓冲区以接收结果。除此之外，无论是在设备上还是在主机 CPU 上生成随机数，所有其他调用的工作原理都是一样的。

同时创建多个生成器是合法的。每个生成器封装了一个独立的状态，并且与所有其他生成器都是独立的。每个生成器产生的数字序列是确定的。在相同的设置参数下，程序的每次运行都会生成相同的序列。在设备上生成的随机数将导致在主机 CPU 上生成相同的序列。

请注意，上述步骤 4 中的 mcrandGenerate() 函数会启动一个内核并异步返回。如果您在不同的流中启动另一个内核，并且该内核需要使用 mcrandGenerate() 的结果，您必须调用 mcThreadSynchronize() 或使用流管理/事件管理例程，以确保随机生成内核在新内核启动前已执行完毕。

请注意，将主机内存指针传递给在设备上运行的生成器是无效的，同样地，将设备内存指针传递给在 CPU 上运行的生成器也是不合法的。在这些情况下的行为是未被定义的。

2.1 生成器类型

随机数生成器是通过将类型传递给 mcrandCreateGenerator() 函数来创建的。

mcRAND 中有九种类型的随机数生成器，总共分为两类。

MCRAND_RNG_PSEUDO_XORWOW, MCRAND_RNG_PSEUDO_MRG32K3A, MCRAND_RNG_PSEUDO_MTGP32, MCRAND_RNG_PSEUDO_PHILOX4_32_10 和 MCRAND_RNG_PSEUDO_MT19937 是伪随机数生成器。

MCRAND_RNG_PSEUDO_XORWOW 是使用 XORWOW 算法实现的，它是 xor-shift 伪随机数生成器系列的成员之一。

MCRAND_RNG_PSEUDO_MRG32K3A 是 Combined Multiple Recursive 伪随机数生成器系列的成员之一。

MCRAND_RNG_PSEUDO_MT19937 和 MCRAND_RNG_PSEUDO_MTGP32 是 Mersenne Twister 伪随机数生成器系列的成员。

MCRAND_RNG_PSEUDO_MTGP32 具有专为在 GPU 上运行而定制的参数。

MCRAND_RNG_PSEUDO_MT19937 具有与 CPU 版本相同的参数，但顺序不同。

MCRAND_RNG_PSEUDO_MT19937 仅支持 HOST API。

MCRAND_RNG_PHILOX4_32_10 是 Philox 系列的成员之一，它是 D E Shaw Research 在 SC11 会议上介绍的三种基于计数器的非密码学随机数生成器之一。

基本的 Sobol 准随机数生成器有 4 个变种。所有的变种都可以生成多达 20,000 维的序列。

MCRAND_RNG_QUASI_SOBOL32, MCRAND_RNG_QUASI_SCRAMBLED_SOBOL32, MCRAND_RNG_QUASI_SOBOL64, 和 MCRAND_RNG_QUASI_SCRAMBLED_SOBOL64 是准随机数生成器类型.

MCRAND_RNG_QUASI_SOBOL32 是一个 32 位序列的 Sobol 生成器。

MCRAND_RNG_QUASI_SCRAMBLED_SOBOL32 是一个 32 位序列的 Scrambled Sobol 生成器。

MCRAND_RNG_QUASI_SOBOL64 是一个 64 位序列的 Sobol 生成器。

MCRAND_RNG_QUASI_SCRAMBLED_SOBOL64 是一个 64 位序列的 Scrambled Sobol 生成器。

2.2 生成器选项

创建后，可使用 seed、offset 和 order 等常规选项定义随机数生成器。

2.2.1 Seed

seed 参数是一个 64 位整数，用于初始化伪随机数生成器的起始状态。相同的 seed 始终会产生相同的结果序列。

2.2.2 Offset

offset 参数用于在序列中跳过一定数量的随机数。如果 offset=100，那么生成的第一个随机数将是序列中的第 100 个。这使得同一个程序的多次运行可以继续从同一个序列中生成结果，而不会重叠。请注意，MCRAND_RNG_PSEUDO_MTGP32 和 MCRAND_RNG_PSEUDO_MT19937 生成器不支持跳过功能。

2.2.3 Order

order 参数用于选择结果在全局内存中的排序方式。它还直接影响 mcRAND 生成函数的性能。

对于伪随机序列，有五种排序选择：MCRAND_ORDERING_PSEUDO_DEFAULT、MCRAND_ORDERING_PSEUDO_LEGACY、MCRAND_ORDERING_PSEUDO_BEST、MCRAND_ORDERING_PSEUDO_SEEDED 和 MCRAND_ORDERING_PSEUDO_DYNAMIC 。

对于准随机数，有一种排序选择：MCRAND_ORDERING_QUASI_DEFAULT 。

伪随机数生成器的默认排序方式是 MCRAND_ORDERING_PSEUDO_DEFAULT ，而准随机数生成器的默认排序方式是 MCRAND_ORDERING_QUASI_DEFAULT 。

两种伪随机排序方式 MCRAND_ORDERING_PSEUDO_DEFAULT 和 MCRAND_ORDERING_PSEUDO_BEST 对于所有伪随机生成器产生相同的输出排序，除了 MT19937，MT19937 中 MCRAND_ORDERING_PSEUDO_DEFAULT 与 MCRAND_ORDERING_PSEUDO_LEGACY 相同。

对于 MT19937，MCRAND_ORDERING_PSEUDO_BEST 在不同型号的 GPU 上可能会生成不同的输出，并且不能与使用 mcrandCreateGeneratorHost() 创建的主机生成器一起使用。

mcRAND 的未来版本可能会改变与 MCRAND_ORDERING_PSEUDO_BEST 相关的排序方式，以提高性能或结果的质量。

使用 MCRAND_ORDERING_PSEUDO_BEST 获得的排序方式始终是确定的，并且每次运行程序时都是相同的。

使用 MCRAND_ORDERING_PSEUDO_LEGACY 获得的排序方式在所有 mcRAND 版本中保证保持不变。

MCRAND_ORDERING_PSEUDO_DYNAMIC 排序方式不能与使用 mcrandCreateGeneratorHost() 创建的主机生成器一起使用，目前仅支持以下伪随机生成器：MCRAND_RNG_PSEUDO_XORWOW、MCRAND_RNG_PSEUDO_PHILOX4_32_10、MCRAND_RNG_PSEUDO_MRG32K3A 和 MCRAND_RNG_PSEUDO_MTGP32。

当选择 MCRAND_ORDERING_PSEUDO_DYNAMIC 排序方式时，mcRAND 会尽力最大化 GPU 利用率，以提供最佳性能。

使用 MCRAND_ORDERING_PSEUDO_DYNAMIC 获得的排序方式可能在不同的 GPU 上有所不同。不能保证在所有 mcRAND 版本中保持相同，并且不能保证在所有分布中保持相同。但是，能保证是确定的。

每种生成器类型的排序参数行为差异概述如下：

- XORWOW 伪随机生成器
  - MCRAND_ORDERING_PSEUDO_DEFAULT

    在当前版本中，MCRAND_ORDERING_PSEUDO_DEFAULT 的输出顺序与 MCRAND_ORDERING_PSEUDO_BEST 相同。
  - MCRAND_ORDERING_PSEUDO_BEST

    在当前版本中，MCRAND_ORDERING_PSEUDO_BEST 的输出顺序与 MCRAND_ORDERING_PSEUDO_LEGACY 相同。
  - MCRAND_ORDERING_PSEUDO_LEGACY

    在全局内存中，偏移量为 n 处的结果来自于原始 XORWOW 序列中的位置 $(n \bmod 4096) \cdot 2^{67} + \lfloor n/4096 \rfloor$。
  - MCRAND_ORDERING_PSEUDO_DYNAMIC

    在不同的 GPU 上，MCRAND_ORDERING_PSEUDO_DYNAMIC 的输出顺序可能不同。
  - MCRAND_ORDERING_PSEUDO_SEEDED

    在全局内存中，偏移为 n 处的结果来自于 XORWOW 序列中的位置 $n/4096\rfloor$，该序列使用用户种子和数字 $n \bmod 4096$ 的组合进行播种。换句话说，4096 个线程中的每一个线程都使用不同的种子。这种种子设置方法减少了状态设置时间，但可能导致某些用户种子值的伪随机输出存在统计上的缺陷。
- MRG32k3a 伪随机生成器
  - MCRAND_ORDERING_PSEUDO_DEFAULT

    在当前版本中，MCRAND_ORDERING_PSEUDO_DEFAULT 的输出顺序与 MCRAND_ORDERING_PSEUDO_BEST 相同。
  - MCRAND_ORDERING_PSEUDO_BEST

    在全局内存中，偏移量为 n 的结果来自原始 MRG32k3a 序列中的位置 $(n \bmod 81920) \cdot 2^{76} + \lfloor n/81920 \rfloor$。（请注意，MRG32k3a 的连续样本之间的步长与 XORWOW 不同）
  - MCRAND_ORDERING_PSEUDO_LEGACY

    在全局内存中，偏移量为 n 的结果来自原始 MRG32k3a 序列中的位置 $(n \bmod 4096) \cdot 2^{76} + \lfloor n/4096 \rfloor$。（请注意，MRG32k3a 的连续样本之间的步长与 XORWOW 不同）
  - MCRAND_ORDERING_PSEUDO_DYNAMIC

    MCRAND_ORDERING_PSEUDO_DYNAMIC 的输出顺序在不同的 GPU 上可能会有所不同。

- MTGP32 伪随机生成器
  - MCRAND_ORDERING_PSEUDO_DEFAULT

    在当前版本中，MCRAND_ORDERING_PSEUDO_DEFAULT 的输出顺序与 MCRAND_ORDERING_PSEUDO_BEST 相同。
  - MCRAND_ORDERING_PSEUDO_BEST

    MTGP32 生成器实际上基于基本算法的不同的参数集生成 192 个不同的序列。设 $S(p)$ 为参数集 p 对应的序列。在全局内存中，偏移量为 n 的结果来自序列 $S(\lfloor n/256 \rfloor \bmod 192)$ 中的位置 $n \bmod 256$。换句话说，首先会生成来自 $S(0)$ 的 256 个样本，然后是来自 $S(1)$ 的 256 个样本，以此类推，直到 $S(191)$。这个模式会重复，所以接下来的 256 个样本又是来自 $S(0)$，然后是来自 $S(1)$，以此类推。
  - MCRAND_ORDERING_PSEUDO_LEGACY

    MTGP32 生成器实际上基于基本算法的不同参数集生成 64 个不同的序列。设 $S(p)$ 为参数集 p 的序列。全局内存中偏移为 n 的结果来自于序列 $S(\lfloor n/256 \rfloor \bmod 64)$ 中的位置 $n \bmod 256$。换句话说，从 $S(0)$ 中取出 256 个样本，然后是从 $S(1)$ 中取出 256 个样本，依此类推，直到 $S(63)$。这个模式会重复，所以接下来的 256 个样本是从 $S(0)$ 中取出的，然后是从 $S(1)$ 中取出的，以此类推。
  - MCRAND_ORDERING_PSEUDO_DYNAMIC

    MCRAND_ORDERING_PSEUDO_DYNAMIC 的输出顺序在不同的 GPU 上可能会有所不同。在这种顺序下，MTGP32 可以使用不同于原始 MTGP32 实现的预计算参数。
- MT19937 伪随机生成器
  - MCRAND_ORDERING_PSEUDO_DEFAULT

    在当前版本中，MCRAND_ORDERING_PSEUDO_DEFAULT 的输出顺序与 MCRAND_ORDERING_PSEUDO_LEGACY 相同。
  - MCRAND_ORDERING_PSEUDO_LEGACY

    排序主要基于标准的 MT19937 CPU 实现。输出是由 8192 个独立的生成器生成的。每个生成器生成原始序列的连续子序列。每个子序列的长度为 2^{1000}。随机数是按照八个一组生成的，因此前 8 个元素来自第一个子序列，接下来的 8 个元素来自第二个子序列，依此类推。为了提高性能，结果的排列方式与原始排列方式不同。顺序与使用的硬件无关。
  - MCRAND_ORDERING_PSEUDO_BEST

    为了获得更好的性能，MCRAND_ORDERING_PSEUDO_BEST 的输出顺序取决于组成您的 GPU 的 SMs 数量。随机数的生成方式与 MCRAND_ORDERING_PSEUDO_LEGACY 相同，但生成器的数量可能不同，以获得更好的性能。使用这种顺序生成种子的速度更快。MCRAND_ORDERING_PSEUDO_BEST 仅支持 GPU mcRAND 随机数生成器，不能与 mcrandCreateGeneratorHost() 创建的主机生成器一起使用。
- Philox_4x32_10 伪随机生成器
  - MCRAND_ORDERING_PSEUDO_DEFAULT

    在当前版本中，MCRAND_ORDERING_PSEUDO_DEFAULT 的输出顺序与 MCRAND_ORDERING_PSEUDO_BEST 相同。
  - MCRAND_ORDERING_PSEUDO_BEST

    在当前版本中，MCRAND_ORDERING_PSEUDO_BEST 的输出顺序与 MCRAND_ORDERING_PSEUDO_LEGACY 相同。
  - MCRAND_ORDERING_PSEUDO_LEGACY

    Philox_4x32_10 生成器中的每个线程根据基本算法的不同参数集生成不同的序列。在主机 API 中，有 65536 个不同的序列。每个序列的四个值后面跟着下一个序列的四个值。
  - MCRAND_ORDERING_PSEUDO_DYNAMIC

    在不同的 GPU 上，MCRAND_ORDERING_PSEUDO_DYNAMIC 的输出顺序可能会有所不同。
- 32 位和 64 位的 SOBOL 和 Scrambled SOBOL 准随机生成器

- MCRAND_ORDERING_QUASI_DEFAULT

  在生成 d 维度中的 n 个结果时，输出将由来自第一维度的 n/d 个结果组成，然后是来自第二维度的 n/d 个结果，依此类推，直到第 d 维度。只能生成维度大小的精确倍数。维度参数 d 可通过 mcrandSetQuasiRandomGeneratorDimensions() 进行设置，默认值为 1。

2.3 返回值

所有 mcRAND 主机库调用都具有 mcrandStatus_t 的返回值。如果调用成功且没有错误，则返回 MCRAND_STATUS_SUCCESS。如果发生错误，则根据错误返回其他值。由于 MXMACA 允许内核与 CPU 代码异步执行，因此在调用库函数时可能会检测到非 mcRAND 内核中的错误。在这种情况下，将返回 MCRAND_STATUS_PREEXISTING_ERROR。

2.4 生成函数

```
mcrandStatus_t
mcrandGenerate(
   mcrandGenerator_t generator,
   unsigned int *outputPtr, size_t num)

mcrandStatus_t
mcrandGenerateLongLong(
   mcrandGenerator_t generator,
   unsigned long long *outputPtr, size_t num)
```

mcrandGenerate() 函数用于为 XORWOW、MRG32k3a、MTGP32、MT19937、Philox_4x32_10 和 SOBOL32 生成器生成伪随机或准随机的输出位。每个输出元素是一个 32 位无符号整数，其中所有位都是随机的。对于 SOBOL64 生成器，每个输出元素是一个 64 位无符号双长（long long）整数，其中所有位都是随机的。mcrandGenerate() 对于 SOBOL64 生成器会返回错误。使用 mcrandGenerateLongLong() 和 SOBOL64 生成器生成 64 位整数。

```
mcrandStatus_t
mcrandGenerateUniform(
    mcrandGenerator_t generator,
    float *outputPtr, size_t num)
```

mcrandGenerateUniform() 函数用于生成在 0.0 到 1.0 之间均匀分布的浮点数值，其中 0.0 被排除在外，而 1.0 被包含在内。

```
mcrandStatus_t
mcrandGenerateNormal(
    mcrandGenerator_t generator,
    float *outputPtr, size_t n,
    float mean, float stddev)
```

mcrandGenerateNormal() 函数用于生成具有给定均值和标准差的正态分布的浮点数值。

```
mcrandStatus_t
mcrandGenerateLogNormal(
    mcrandGenerator_t generator,
    float *outputPtr, size_t n,
    float mean, float stddev)
```

mcrandGenerateLogNormal() 函数用于基于具有给定均值和标准差的正态分布生成对数正态分布的浮点数值。

```
mcrandStatus_t
mcrandGeneratePoisson(
    mcrandGenerator_t generator,
    unsigned int *outputPtr, size_t n,
    double lambda)
```

mcrandGeneratePoisson() 函数用于根据给定的 lambda 值生成符合泊松分布的整数值。

```
mcrandStatus_t
mcrandGenerateUniformDouble(
    mcrandGenerator_t generator,
    double *outputPtr, size_t num)
```

mcrandGenerateUniformDouble() 函数用于生成双精度的均匀分布随机数。

```
mcrandStatus_t
mcrandGenerateNormalDouble(
    mcrandGenerator_t generator,
    double *outputPtr, size_t n,
    double mean, double stddev)
```

mcrandGenerateNormalDouble() 函数用于生成具有给定均值和标准差的双精度正态分布结果。

```
mcrandStatus_t
mcrandGenerateLogNormalDouble(
    mcrandGenerator_t generator,
    double *outputPtr, size_t n,
    double mean, double stddev)
```

mcrandGenerateLogNormalDouble() 函数基于具有给定均值和标准差的正态分布，生成双精度的对数正态分布结果。

对于准随机生成，返回的结果数量必须是生成器维度的倍数。

可以在同一个生成器上多次调用生成函数来生成连续的结果块。对于伪随机生成器，多次调用生成函数的结果与单次调用大尺寸生成器的结果相同。对于准随机生成器，由于内存中维度的排序，许多较短的调用将不会在内存中产生与一个较大的调用相同的结果；然而生成的 n 维向量将是相同的。

2.5 主机 API 示例

```
/*
* 这个程序使用主机 MCRAND API 生成 100 个数值。
* 伪随机浮点数。
*/
#include <stdio.h>
#include <stdlib.h>
#include <mc_runtime.h>
#include <mcrand.h>

#define MC_CALL(x) do { if((x)!=mcSuccess) { \
        printf("Error at %s:%d\n",__FILE__,__LINE__);\
        return EXIT_FAILURE;}} while(0)
#define MCRAND_CALL(x) do { if((x)!=MCRAND_STATUS_SUCCESS) { \
        printf("Error at %s:%d\n",__FILE__,__LINE__);\
        return EXIT_FAILURE;}} while(0)
```

(下页继续)

(续上页)

```
int main(int argc, char *argv[])
{
    size_t n = 100;
    size_t i;
    mcrandGenerator_t gen;
    float *devData, *hostData;

    /* 在主机上分配 n 个浮点数的内存空间。 */
    hostData = (float *)calloc(n, sizeof(float));

    /* 在设备上分配 n 个浮点数的内存空间。 */
    MC_CALL(mcMalloc((void **)&devData, n*sizeof(float)));

    /* 创建伪随机数生成器。 */
    MCRAND_CALL(mcrandCreateGenerator(&gen,
                MCRAND_RNG_PSEUDO_DEFAULT));

    /* 设置随机数种子 */
    MCRAND_CALL(mcrandSetPseudoRandomGeneratorSeed(gen,
                1234ULL));

    /* 在设备上生成 n 个浮点数。*/
    MCRAND_CALL(mcrandGenerateUniform(gen, devData, n));

    /* 将设备内存复制到主机。 */
    MC_CALL(mcMemcpy(hostData, devData, n * sizeof(float),
        mcMemcpyDeviceToHost));

    /* 展示结果 */
    for(i = 0; i < n; i++) {
        printf("%1.4f ", hostData[i]);
    }
    printf("\n");

    /* 清除 */
    MCRAND_CALL(mcrandDestroyGenerator(gen));
    MC_CALL(mcFree(devData));
    free(hostData);
    return EXIT_SUCCESS;
}
```

2.6 静态库支持

在 Linux 上，mcRAND 库也提供了一个名为 libmcrand_static.a 的静态库供使用。

例如，在 linux 上，要使用 mcRAND 针对动态库编译一个小型应用程序，可以使用以下命令:

```
mxcc myMCrandApp.c
   -lmcrand -o myMCrandApp
```

如果要针对静态的 mcRAND 库进行编译，则必须使用以下命令:

```
mxcc myMCrandApp.c
   -lmcrand_static -o myMCrandApp
```

2.7 性能说明

一般来说，通过生成尽可能大的随机数块，可以从 mcRAND 库中获得最佳性能。用更少的调用生成多个随机数，比多次调用只生成几个随机数更有效。默认的伪随机生成器 XORWOW，在第一次调用时需要一些时间进行设置。后续的生成调用不需要进行这个设置。为了省去这个设置时间，请使用 MCRAND_ORDERING_PSEUDO_SEEDED 排序方式。

MTGP32 Mersenne Twister 算法与线程计数和块计数密切相关。MTGP32 的状态结构实际上包含给定序列的 256 个连续样本的状态，由特定的参数集确定。每 64 个块使用不同的参数集，每 256 个线程从状态中生成一个样本，并更新状态。因此，MTGP32 的最有效使用方式是生成 16384 个样本的倍数。

MT19937 算法的性能取决于单个调用过程中产生的样本数量。在生成超过 2GB 的数据时可以达到峰值性能，但仅生成 80MB 的数据就可以达到峰值性能的 80%。

Philox_4x32_10 算法与线程计数和块计数密切相关。每个线程同时计算 4 个随机数，因此 Philox_4x32_10 的最有效使用方式是生成 4 倍线程数的样本数。

为了获得 mcRAND 主机 API 的最佳性能，鼓励用户使用 MCRAND_ORDERING_PSEUDO_BEST 或 MCRAND_ORDERING_PSEUDO_DYNAMIC 排序。

2.8 线程安全性

不同的主机线程使用不同的生成器，生成器不是 MT19937（MCRAND_RNG_PSEUDO_MT19937）且输出不重叠的情况下，mcRAND 主机 API 就是线程安全的。

请注意，当与 MT19937 生成器 (MCRAND_RNG_PSEUDO_MT19937) 一起使用时，mcRAND 主机 API 不是线程安全的。

3 设备 API 概述

要使用设备 API，在定义使用 mcRAND 设备函数的内核文件时，需要包含文件 mcrand_kernel.h。设备 API 包括用于 pseudorandom generation 和 quasirandom generation 函数。

3.1 伪随机序列

伪随机序列的功能支持位（bit）生成和分布式生成。

3.1.1 使用 XORWOW 和 MRG32k3a 的位生成

```
__device__ unsigned int
mcrand (mcrandState_t *state)
```

在调用 mcrand_init() 之后，mcrand() 会返回一个周期大于 2^{190} 的伪随机数序列。如果每次调用 mcrand() 时使用相同的初始状态，并且在 mcrand() 多次调用之间不修改状态，那么生成的序列总是相同的。

```
__device__ void
mcrand_init (
     unsigned long long seed, unsigned long long sequence,
     unsigned long long offset, mcrandState_t *state)
```

mcrand_init() 函数使用给定的种子、序列号和序列内偏移量来设置由调用者分配的初始状态。不同的种子保证产生不同的起始状态和不同的序列。相同的种子总是产生相同的状态和相同的序列。设置的状态将是从种子状态开始经过 2^{67} * sequence + offset 次调用 mcrand() 后的状态。

虽然使用不同种子生成的序列通常不具有统计相关性，但某些特定的种子选择可能会产生例外。使用相同种子和不同序列号生成的序列不会有统计相关性。

为了生成质量最高的并行伪随机数，每个实验都应该分配一个唯一的种子。在实验中，每个计算线程都应该分配一个唯一的序号。如果一次实验跨越了多次内核启动，那么建议内核启动之间的线程使用相同的种子，并以单调递增的方式分配序号。如果启动相同的线程配置，启动之间可以在全局内存中保存随机状态，以省去状态设置时间。

3.1.2 使用 MTGP32 生成器的位生成

MTGP32 生成器改编自广岛大学开发的代码。

在该算法中，样本是为多个序列而生成，每个序列基于一组计算得到的参数。mcRAND 使用了为 32 位生成器预先生成的 200 个参数集，周期为 2^{11214} 。也可以生成其他参数集，并使用它们。每个参数集 (序列) 都有一个状态结构，该算法允许对 200 个序列中的每个序列进行多达 256 个并发线程 (在单个块内) 的线程安全生成和状态更新。

需要注意的是，两个不同的块不能对同一状态进行安全操作。还要注意的是，在一个块内，给定状态最多可以操作 256 个线程。

对于 MTGP32 生成器，提供了两个主机函数，用于帮助设置设备内存中不同序列的参数，以及设置初始状态。

```
__host__ mcrandStatust mcrandMakeMTGP32Constants(mtgp32paramsfastt
↪params[],
                                                  mtgp32kernelparamst *p)
```

该函数将预生成格式 (mtgp32_params_fast_t) 的参数集数据重新组织为内核函数使用的格式 (mtgp32_kernel_params_t)，并将其复制到设备内存中。

```
__host__ mcrandStatus_t
mcrandMakeMTGP32KernelState(mcrandStateMtgp32_t *s,
                            mtgp32_params_fast_t params[],
                            mtgp32_kernel_params_t *k,
                            int n,
                            unsigned long long seed)
```

该函数根据指定的参数集和种子初始化的 n 个状态，并将它们复制到由 s 指示的设备内存中。请注意，如果您使用预生成的状态，n 的最大值为 200。

mcRAND MTGP32 生成器提供了两个内核函数来生成随机位。

```
__device__ unsigned int
mcrand (mcrandStateMtgp32_t *state)
```

该函数计算线程索引，并为该索引生成结果并更新状态。线程索引 t 的计算如下:

t= (blockDim.x * blockDim.y * threadIdx.z) + (blockDim.x * threadIdx.y) + threadIdx.x

该函数可以在单个内核启动中重复调用，但需要满足以下约束条件：

- 它只能在具有 256 个或更少线程的块中安全调用。
- 一个给定的状态不能被多个块使用。
- 一个给定的块可以使用多个状态生成随机数。
- 在代码的某个给定点，块中的所有线程要么全部调用此函数，要么全部不调用。

```
__device__ unsigned int
mcrandmtgp32specific(mcrandStateMtgp32_t *state, unsigned char index,
                     unsigned char n)
```

此函数根据线程特定的索引生成一个结果并更新状态，然后将状态中的偏移量向前推进 n 个位置。mcrand_mtgp32_specific 可以在一个内核启动中多次调用，但需要满足以下约束条件：

- 对于给定的状态，最多 256 个线程可以调用这个函数。
- 在一个块内，对于给定的状态，如果有 n 个线程调用该函数，索引必须从 0 运行到 n-1。索引不必与线程号匹配，并且可以根据调用程序的要求在线程之间分布。在代码的某个特定点，必须使用所有的索引从 0 到 n-1，或者不使用任何索引。
- 一个给定的状态不能被多个块使用。
- 一个给定的块可以使用多个状态生成随机数。

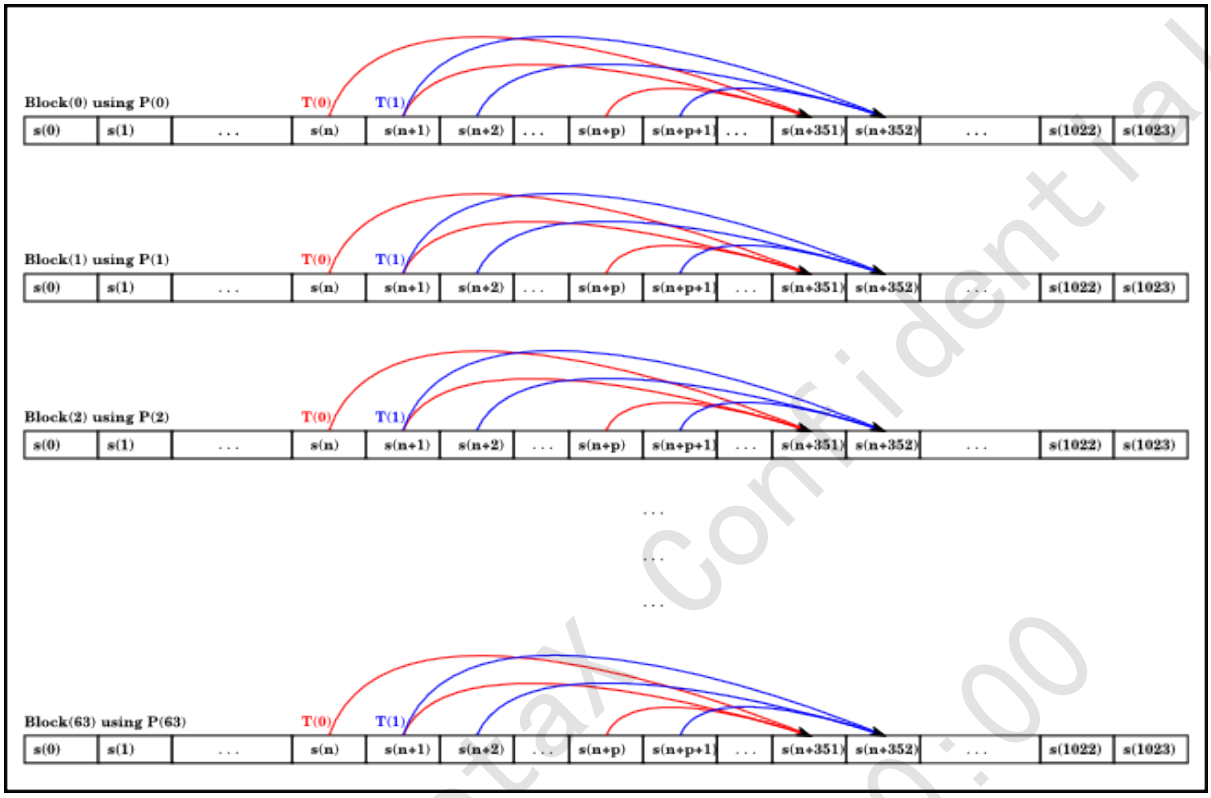

图 1.MTGP32 块和线程操作

图 1 是 MTGP32 中块和线程对生成器状态进行操作的示意图。每一行代表一个由 32 位整数 s(n) 组成的循环状态数组。操作数组的线程被标识为 T(m)。所示的特定情况与主机 API 的内部实现相匹配，它启动了 64 个块，每个块有 256 个线程。每个块根据一组唯一的参数 P(n) 操作不同的序列。一个完整的 MTGP32 序列状态由 351 个 32 位整数定义。每个线程 T(m) 操作其中一个整数 s(n+m)，将其与 s(n+m+1) 和一个取样元素 s(n+m+p)（其中 p <= 95）组合，它将新状态存储在状态数组的位置 s(n+m+351)。线程同步后，基础索引 n 会根据已更新状态的线程数量进行推进。为了避免被覆盖，数组本身的长度必须至少为 256 + 351 个整数。实际上，为了索引的效率，它的大小为 1024 个整数。

对于可以操作给定状态数组的块中的线程数量的限制，是为了确保在需要作为取样状态之前，状态 s(n+351) 已经被更新。如果有一个线程 T(256)，它可以使用 s(n+256+95)，即在零号线程更新 s(n+351) 之前使用 s(n+351)。如果一个应用程序需要在一个块中调用超过 256 个线程的 MTGP32 生成器函数，它必须使用多个 MTGP32 状态，可以通过使用多个参数集，或者使用具有不同种子的多个生成器来实现。还要注意，生成器函数在每次调用结束时同步线程，因此在一个块中调用 256 个线程来使用生成器是最有效的。

3.1.3 使用 Philox_4x32_10 生成器的位生成

```
__device__ unsigned int
mcrand (mcrandState_t *state)
```

在调用 mcrand_init() 之后，mcrand() 返回一个带句点的伪随机数序列 2^{128} 。如果每次以相同的初始状态调用 mcrand()，并且在 mcrand() 多次调用之间没有修改状态，那么生成的序列总是相同的。

```
__device__ void
mcrand_init (
    unsigned long long seed, unsigned long long subsequence,
    unsigned long long offset, mcrandState_t *state)
```

mcrand_init() 函数使用给定的种子、子序列和偏移量设置由调用者分配的初始状态。不同的种子保证产生不同的起始状态和不同的序列。子序列和偏移量共同定义了周期为 2^{128} 的序列中的偏移量。偏移量定

义了长度为 2^{64} 的子序列中的偏移量。当子序列的最后一个元素被生成时，下一个随机数就是连续子序列的第一个元素。相同的种子总是产生相同的状态和序列。

虽然使用不同种子生成的序列通常不具有统计相关性，但某些特定的种子选择可能会产生例外。

为了获得最高质量的并行伪随机数生成，每个实验应分配一个唯一的种子值。在一个实验中，每个计算线程应分配一个唯一的 ID 号码。如果一个实验跨越多个内核启动，建议在内核启动之间的线程给予相同的种子，并以单调递增的方式分配 ID 号码。如果启动相同配置的线程，可以在启动之间在全局内存中保留随机状态，以省去状态设置时间。

3.1.4 分布

```
__device__ float
mcrand_uniform (mcrandState_t *state)
```

这个函数返回一个均匀分布在 0.0 到 1.0 之间的伪随机浮点数序列。它的返回范围是 0.0 到 1.0，包含 1.0，不包含 0.0。分布函数可以使用基本生成器提供的任意数量的无符号整数值。使用的值的数量不保证是固定的。

```
__device__ float
mcrand_normal (mcrandState_t *state)
```

该函数返回一个具有均值为 0.0 和标准差为 1.0 的正态分布浮点数。此结果可以进行缩放和移动，以产生具有任意均值和标准差的正态分布值。

```
__device__ float
mcrand_log_normal (mcrandState_t *state, float mean, float stddev)
```

该函数基于具有给定均值和标准差的正态分布，返回一个单精度对数正态分布浮点数。

```
__device__ unsigned int
mcrand_poisson (mcrandState_t *state, double lambda)
```

该函数根据给定的 lambda，返回一个泊松分布无符号整数。用于从均匀分布的结果中得出泊松结果的算法取决于 lambda 的值和生成器的类型。有些算法在一个输出中会产生多个样本。

```
__device__ double
mcrand_uniform_double (mcrandState_t *state)
```

```
__device__ double
mcrand_normal_double (mcrandState_t *state)
```

```
__device__ double
mcrand_log_normal_double (mcrandState_t *state, double mean, double stddev)
```

上面的三个函数是双精度版本的 mcrand_uniform()、mcrand_normal() 和 mcrand_log_normal()。

对于伪随机生成器，双精度函数使用多次调用 mcrand() 来生成 53 个随机位。

```
__device__ float2
mcrand_normal2 (mcrandState_t *state)
```

```
__device__ float2
mcrand_log_normal2 (mcrandState_t *state)
```

```
__device__ double2
mcrand_normal2_double (mcrandState_t *state)
```

```
__device__ double2
mcrand_log_normal2_double (mcrandState_t *state)
```

上述函数每次调用生成两个正态分布或对数正态分布的伪随机结果。由于底层实现使用了 Box-Muller 变换，这通常比每次调用生成一个结果更高效。

```
__device__ uint4
mcrand4 (mcrandStatePhilox4_32_10_t *state)
```

```
__device__ float4
mcrand_uniform4 (mcrandStatePhilox4_32_10_t *state)
```

```
__device__ float4
mcrand_normal4 (mcrandStatePhilox4_32_10_t *state)
```

```
__device__ float4
mcrand_log_normal4 (mcrandStatePhilox4_32_10_t *state, float mean, float
↪stddev)
```

```
__device__ uint4
mcrand_poisson4 (mcrandStatePhilox4_32_10_t *state, double lambda)
```

```
__device__ uint4
mcrand_discrete4 (mcrandStatePhilox4_32_10_t *state,
↪mcrandDiscreteDistribution_t discrete_distribution)
```

```
__device__ double2
mcrand_uniform2_double (mcrandStatePhilox4_32_10_t *state)
```

```
__device__ double2
mcrand_normal2_double (mcrandStatePhilox4_32_10_t *state)
```

```
__device__ double2
mcrand_log_normal2_double (mcrandStatePhilox4_32_10_t *state, double mean,
↪double stddev)
```

上述函数每次调用生成四个单精度或两个双精度的结果。由于底层实现使用了 Philox 生成器，这通常比每次产生调用只生成一个结果更高效。

3.2 准随机序列

尽管默认的生成器类型是来自 XORWOW 的伪随机数，但可以使用以下函数生成基于 Sobol’ 32 位整数的 Sobol’ 序列：

```
__device__ void
mcrand_init (
    unsigned int *direction_vectors,
    unsigned int offset,
    mcrandStateSobol32_t *state)
```

```
__device__ void
mcrand_init (
    unsigned int *direction_vectors,
```

(下页继续)

(续上页)

```
    unsigned int scramble_c,
    unsigned int offset,
    mcrandStateScrambledSobol32_t *state)

__device__ unsigned int
mcrand (mcrandStateSobol32_t *state)

__device__ float
mcrand_uniform (mcrandStateSobol32_t *state)

__device__ float
mcrand_normal (mcrandStateSobol32_t *state)

__device__ float
mcrand_log_normal (
    mcrandStateSobol32_t *state,
    float mean,
    float stddev)

__device__ unsigned int
mcrand_poisson (mcrandStateSobol32_t *state, double lambda)

__device__ double
mcrand_uniform_double (mcrandStateSobol32_t *state)

__device__ double
mcrand_normal_double (mcrandStateSobol32_t *state)

__device__ double
mcrand_log_normal_double (
    mcrandStateSobol32_t *state,
    double mean,
    double stddev)
```

mcrand_init() 函数用于初始化准随机数生成器的状态。它没有种子参数，只有方向向量和偏移量。对于 Scrambled Sobol 生成器，还有一个额外的参数 scramble_c，它是 Scrambled 序列的初始值。对于 mcrandStateSobol32_t 类型和 mcrandStateScrambledSobol32_t 类型，方向向量是一个包含 32 个无符号整数值的数组。对于 mcrandStateSobol64_t 类型和 mcrandStateScrambledSobol64_t 类型，方向向量是一个包含 64 个无符号双长整数值的数组。对于 32 位 Sobol 生成器，偏移量和 Scrambled 序列的初始常数的类型是无符号整数。对于 64 位 Sobol 生成器，这些参数的类型是无符号双长整数。对于 mcrandStateSobol32_t 类型和 mcrandStateScrambledSobol32_t 类型，序列的长度恰好为 2^{32} 个元素，每个元素为 32 位。对于 mcrandStateSobol64_t 类型和 mcrandStateScrambledSobol64_t 类型，序列的长度恰好为 2^{64} 个元素，每个元素为 64 位。每次调用 mcrand() 都会返回下一个准随机元素。调用 mcrand_uniform() 返回从 0.0 到 1.0 的准随机浮点数或双精度数，其中 1.0 包含在内，0.0 不包含在内。类似地，调用 mcrand_normal() 返回均值为 0.0，标准差为 1.0 的正态分布浮点数或双精度数。调用 mcrand_log_normal() 返回从指定均值和标准差的正态分布派生的对数正态分布浮点数或双精度数。所有生成函数都可以使用任何类型的 Sobol 生成器进行调用。

例如，生成填充单位立方体的准随机坐标需要跟踪三个准随机生成器。这三个生成器都从偏移量为 0 开始，维度分别为 0、1 和 2。对于每个生成器状态，调用 mcrand_uniform() 一次将生成 x, y, 和 z 坐标。方向向量的表可以通过 mcrandGetDirectionVectors32() 和 mcrandGetDirectionVectors64() 函数在主机上访问。在使用之前，需要将所需的方向向量复制到设备内存中。

用于生成准随机序列的正态分布函数使用反累积密度函数来保持准随机序列的维数。因此，不存在像伪随机生成器那样一次生成多个结果的函数。

双精度 Sobol32 函数以双精度返回结果，使用底层生成器的 32 位内部精度。

双精度 Sobol64 函数以双精度返回结果，使用底层生成器的 64 位样本的高阶 53 位作为内部精度。

3.3 跳过（Skip-Ahead）

有几个函数可以从生成器状态跳过。

```
__device__ void
skipahead(unsigned long long n, mcrandState_t *state)
```

```
__device__ void
skipahead(unsigned int n, mcrandStateSobol32_t *state)
```

使用这个函数等价于调用 mcrand() n 次而不使用返回值，但速度更快。

```
__device__ void
skipahead_sequence(unsigned long long n, mcrandState_t *state)
```

这个函数相当于调用 mcrand() $n \cdot 2^{67}$ 次而不使用返回值，速度更快。

3.4 用于离散分布的设备 API

离散分布（如泊松分布）需要额外的 API 在主机端进行预处理，以生成特定分布的直方图。在泊松分布的情况下，不同的 lambda 值会产生不同的直方图。这些分布的最佳性能将在具有至少 48KB L1 缓存的 GPU 上体现。

```
mcrandStatus_t
mcrandCreatePoissonDistribution(
    double lambda,
    mcrandDiscreteDistribution_t *discrete_distribution)
```

mcrandCreatePoissonDistribution() 函数用于创建给定 lambda 表达式下的泊松分布直方图。

```
__device__ unsigned int
mcrand_discrete (
    mcrandState_t *state,
    mcrandDiscreteDistribution_t discrete_distribution)
```

该函数根据给定离散分布直方图的分布返回单个离散无符号整型分布。

```
mcrandStatus_t
mcrandDestroyDistribution(
    mcrandDiscreteDistribution_t discrete_distribution)
```

mcrandDestroyDistribution() 函数用于清理与直方图相关的结构。

3.5 性能说明

调用 mcrand_init() 比调用 mcrand() 或 mcrand_uniform() 慢。较大的偏移量对 mcrand_init() 的调用比较小的偏移量需要更多时间。保存和恢复随机生成器状态比重复计算起始状态要快得多。

如下所示，生成器状态可以在内核启动之间存储在全局内存中，使用本地内存进行快速生成，然后再存回全局内存中。

```
__global__ void example(mcrandState *global_state)
{
    mcrandState local_state;
    local_state = global_state[threadIdx.x];
    for(int i = 0; i < 10000; i++) {
        unsigned int x = mcrand(&local_state);
        ...
    }
    global_state[threadIdx.x] = local_state;
}
```

随机生成器状态的初始化通常需要比随机数生成更多的寄存器和本地内存。将 mcrand_init() 和 mcrand() 的调用分开成独立的内核可以获得最佳性能。

设置状态可能是一个昂贵的操作。加快设置的一种方法是为每个线程使用不同的种子和一个常数序列号为 0。如果需要创建许多生成器，这种方法尤其有帮助。虽然设置速度更快，但该方法对生成序列的数学属性提供的保证较少。如果从种子初始化生成器状态的哈希函数与生成器的周期性之间存在不良交互，那么对于某些种子值，可能会出现高度相关的输出线程。我们还不知道有任何有问题的值；但即使存在，也很可能是罕见的。

3.6 设备 API 示例 (Device API Examples)

本示例使用 mcRAND 设备 API，借助 XORWOW 或 MRG32k3a 生成器生成伪随机数。对于整数，它计算具有低位集的部分。对于均匀分布的实数，它计算出大于 0.5 的部分。对于正态分布的实数，它计算出在平均值的一个标准差范围内的部分

```
/*
    * 这个程序使用设备 MCRAND API 来计算伪随机整数中具有低位设置的的比例。
    * 然后生成统一结果，计算有多少大于 0.5。
    * 然后，它会生成正态结果，计算有多少在平均值的一个标准差内。
    */
#include <stdio.h>
#include <stdlib.h>

#include <mc_runtime.h>
#include <mcrand_kernel.h>

#define MC_CALL(x) do { if((x) != mcSuccess) { \
    printf("Error at %s:%d\n",__FILE__,__LINE__); \
    return EXIT_FAILURE;}} while(0)

__global__ void setup_kernel(mcrandState *state)
{
    int id = threadIdx.x + blockIdx.x * blockDim.x;
    /* 每个线程得到相同的种子，一个不同的序列号，
        没有偏移量 */
    mcrand_init(1234, id, 0, &state[id]);
}

__global__ void setup_kernel(mcrandStatePhilox4_32_10_t *state)
{
    int id = threadIdx.x + blockIdx.x * blockDim.x;
    /* 每个线程得到相同的种子，一个不同的序列号，
        没有偏移量 */
    mcrand_init(1234, id, 0, &state[id]);
```

(下页继续)

(续上页)

```
}

__global__ void setup_kernel(mcrandStateMRG32k3a *state)
{
    int id = threadIdx.x + blockIdx.x * blockDim.x;
    /* 每个线程得到相同的种子，一个不同的序列号，
        没有偏移量 */
    mcrand_init(0, id, 0, &state[id]);
}

__global__ void generate_kernel(mcrandState *state,
                                int n,
                                unsigned int *result)
{
    int id = threadIdx.x + blockIdx.x * blockDim.x;
    int count = 0;
    unsigned int x;
    /* 将状态复制到本地内存中以提高效率 */
    mcrandState localState = state[id];
    /* 生成伪随机的无符号整数 */
    for(int i = 0; i < n; i++) {
        x = mcrand(&localState);
        /* Check if low bit set */
        if(x & 1) {
            count++;
        }
    }
    /* 将状态复制回全局内存 */
    state[id] = localState;
    /* 存储结果 */
    result[id] += count;
}

__global__ void generate_kernel(mcrandStatePhilox4_32_10_t *state,
                                int n,
                                unsigned int *result)
{
    int id = threadIdx.x + blockIdx.x * blockDim.x;
    int count = 0;
    unsigned int x;
    /* 将状态复制到本地内存中以提高效率 */
    mcrandStatePhilox4_32_10_t localState = state[id];
    /* 生成伪随机的无符号整数 */
    for(int i = 0; i < n; i++) {
        x = mcrand(&localState);
        /* Check if low bit set */
        if(x & 1) {
            count++;
        }
    }
    /* 将状态复制回全局内存 */
    state[id] = localState;
    /* 存储结果 */
    result[id] += count;
}
```

(下页继续)

(续上页)

```
__global__ void generate_uniform_kernel(mcrandState *state,
                                int n,
                                unsigned int *result)
{
    int id = threadIdx.x + blockIdx.x * blockDim.x;
    unsigned int count = 0;
    float x;
    /* 将状态复制到本地内存中以提高效率 */
    mcrandState localState = state[id];
    /* 生成伪随机的无符号整数 */
    for(int i = 0; i < n; i++) {
        x = mcrand_uniform(&localState);
        /* Check if > .5 */
        if(x > .5) {
            count++;
        }
    }
    /* 将状态复制回全局内存 */
    state[id] = localState;
    /* 存储结果 */
    result[id] += count;
}

__global__ void generate_uniform_kernel(mcrandStatePhilox4_32_10_t *state,
                                int n,
                                unsigned int *result)
{
    int id = threadIdx.x + blockIdx.x * blockDim.x;
    unsigned int count = 0;
    float x;
    /* 将状态复制到本地内存中以提高效率 */
    mcrandStatePhilox4_32_10_t localState = state[id];
    /* 生成伪随机的无符号整数 */
    for(int i = 0; i < n; i++) {
        x = mcrand_uniform(&localState);
        /* Check if > .5 */
        if(x > .5) {
            count++;
        }
    }
    /* 将状态复制回全局内存 */
    state[id] = localState;
    /* 存储结果 */
    result[id] += count;
}

__global__ void generate_normal_kernel(mcrandState *state,
                                int n,
                                unsigned int *result)
{
    int id = threadIdx.x + blockIdx.x * blockDim.x;
    unsigned int count = 0;
    float2 x;
    /* 将状态复制到本地内存中以提高效率 */
    mcrandState localState = state[id];
    /* 生成伪随机的正态分布 */
```

(下页继续)

(续上页)

```
    for(int i = 0; i < n/2; i++) {
        x = mcrand_normal2(&localState);
        /* Check if within one standard deviaton */
        if((x.x > -1.0) && (x.x < 1.0)) {
            count++;
        }
        if((x.y > -1.0) && (x.y < 1.0)) {
            count++;
        }
    }
    /* 将状态复制回全局内存 */
    state[id] = localState;
    /* 存储结果 */
    result[id] += count;
}

__global__ void generate_normal_kernel(mcrandStatePhilox4_32_10_t *state,
                                       int n,
                                       unsigned int *result)
{
    int id = threadIdx.x + blockIdx.x * blockDim.x;
    unsigned int count = 0;
    float2 x;
    /* 将状态复制到本地内存中以提高效率 */
    mcrandStatePhilox4_32_10_t localState = state[id];
    /* 生成伪随机正态分布 */
    for(int i = 0; i < n/2; i++) {
        x = mcrand_normal2(&localState);
        /* Check if within one standard deviaton */
        if((x.x > -1.0) && (x.x < 1.0)) {
            count++;
        }
        if((x.y > -1.0) && (x.y < 1.0)) {
            count++;
        }
    }
    /* 将状态复制回全局内存 */
    state[id] = localState;
    /* 存储结果 */
    result[id] += count;
}

__global__ void generate_kernel(mcrandStateMRG32k3a *state,
                                int n,
                                unsigned int *result)
{
    int id = threadIdx.x + blockIdx.x * blockDim.x;
    unsigned int count = 0;
    unsigned int x;
    /* 将状态复制到本地内存中以提高效率 */
    mcrandStateMRG32k3a localState = state[id];
    /* 生成伪随机的无符号整数 */
    for(int i = 0; i < n; i++) {
        x = mcrand(&localState);
        /* Check if low bit set */
        if(x & 1) {
```

(下页继续)

(续上页)

```
            count++;
        }
    }
    /* 将状态复制回全局内存 */
    state[id] = localState;
    /* 存储结果 */
    result[id] += count;
}

__global__ void generate_uniform_kernel(mcrandStateMRG32k3a *state,
                                int n,
                                unsigned int *result)
{
    int id = threadIdx.x + blockIdx.x * blockDim.x;
    unsigned int count = 0;
    double x;
    /* 将状态复制到本地内存中以提高效率 */
    mcrandStateMRG32k3a localState = state[id];
    /* 生成伪随机的无符号整数 */
    for(int i = 0; i < n; i++) {
        x = mcrand_uniform_double(&localState);
        /* Check if > .5 */
        if(x > .5) {
            count++;
        }
    }
    /* 将状态复制回全局内存 */
    state[id] = localState;
    /* 存储结果 */
    result[id] += count;
}

__global__ void generate_normal_kernel(mcrandStateMRG32k3a *state,
                                int n,
                                unsigned int *result)
{
    int id = threadIdx.x + blockIdx.x * blockDim.x;
    unsigned int count = 0;
    double2 x;
    /* 将状态复制到本地内存中以提高效率 */
    mcrandStateMRG32k3a localState = state[id];
    /* 生成伪随机的正态分布 */
    for(int i = 0; i < n/2; i++) {
        x = mcrand_normal2_double(&localState);
        /* Check if within one standard deviaton */
        if((x.x > -1.0) && (x.x < 1.0)) {
            count++;
        }
        if((x.y > -1.0) && (x.y < 1.0)) {
            count++;
        }
    }
    /* 将状态复制回全局内存 */
    state[id] = localState;
    /* 存储结果 */
    result[id] += count;
```

(下页继续)

(续上页)

```
}

int main(int argc, char *argv[])
{
    const unsigned int threadsPerBlock = 64;
    const unsigned int blockCount = 64;
    const unsigned int totalThreads = threadsPerBlock * blockCount;

    unsigned int i;
    unsigned int total;
    mcrandState *devStates;
    mcrandStateMRG32k3a *devMRGStates;
    mcrandStatePhilox4_32_10_t *devPHILOXStates;
    unsigned int *devResults, *hostResults;
    bool useMRG = 0;
    bool usePHILOX = 0;
    int sampleCount = 10000;
    bool doubleSupported = 0;
    int device;
    struct mcDeviceProp properties;

    /* 检查是否支持双精度 */
    MC_CALL(mcGetDevice(&device));
    MC_CALL(mcGetDeviceProperties(&properties,device));
    if ( properties.major >= 2 || (properties.major == 1 && properties.
↪minor >= 3) ) {
        doubleSupported = 1;
    }

    /* 检查 MRG32k3a 选项 (默认为 XORWOW) */
    if (argc >= 2)  {
        if (strcmp(argv[1],"-m") == 0) {
            useMRG = 1;
            if (!doubleSupported){
                printf("MRG32k3a requires double precision\n");
                printf("^^^^ test WAIVED due to lack of double precision\n
↪");
                return EXIT_SUCCESS;
            }
        }else if (strcmp(argv[1],"-p") == 0) {
        usePHILOX = 1;
    }
        /* 允许覆盖样本计数 */
        sscanf(argv[argc-1],"%d",&sampleCount);
    }

    /* 为主机上的结果分配空间 */
    hostResults = (unsigned int *)calloc(totalThreads, sizeof(int));

    /* 为设备上的结果分配空间 */
    MC_CALL(mcMalloc((void **)&devResults, totalThreads *
                sizeof(unsigned int)));

    /* 将结果设置为 0 */
    MC_CALL(mcMemset(devResults, 0, totalThreads *
                sizeof(unsigned int)));
```

(下页继续)

(续上页)

```
    /* 为设备上的 prng 状态分配空间 */
    if (useMRG) {
        MC_CALL(mcMalloc((void **)&devMRGStates, totalThreads *
                    sizeof(mcrandStateMRG32k3a)));
    }else if(usePHILOX) {
        MC_CALL(mcMalloc((void **)&devPHILOXStates, totalThreads *
                    sizeof(mcrandStatePhilox4_32_10_t)));
    }else {
        MC_CALL(mcMalloc((void **)&devStates, totalThreads *
                    sizeof(mcrandState)));
    }

    /* 设置 prng 状态 */
    if (useMRG) {
        setup_kernel<<<64, 64>>>(devMRGStates);
    }else if(usePHILOX)
    {
        setup_kernel<<<64, 64>>>(devPHILOXStates);
    }else {
        setup_kernel<<<64, 64>>>(devStates);
    }

    /* 生成并使用伪随机参数  */
    for(i = 0; i < 50; i++) {
        if (useMRG) {
            generate_kernel<<<64, 64>>>(devMRGStates, sampleCount,
↪devResults);
        }else if (usePHILOX){
            generate_kernel<<<64, 64>>>(devPHILOXStates, sampleCount,
↪devResults);
    }else {
            generate_kernel<<<64, 64>>>(devStates, sampleCount,
↪devResults);
        }
    }

    /* 将设备内存复制到主机 */
    MC_CALL(mcMemcpy(hostResults, devResults, totalThreads *
        sizeof(unsigned int), mcMemcpyDeviceToHost));

    /* 显示结果 */
    total = 0;
    for(i = 0; i < totalThreads; i++) {
        total += hostResults[i];
    }
    printf("Fraction with low bit set was %10.13f\n",
        (float)total / (totalThreads * sampleCount * 50.0f));

    /* 将结果设置为 0*/
    MC_CALL(mcMemset(devResults, 0, totalThreads *
                sizeof(unsigned int)));

    /* 生成和使用均匀的伪随机参数  */
    for(i = 0; i < 50; i++) {
        if (useMRG) {
```

(下页继续)

(续上页)

```
            generate_uniform_kernel<<<64, 64>>>(devMRGStates, sampleCount,
↪devResults);
        }else if(usePHILOX) {
            generate_uniform_kernel<<<64, 64>>>(devPHILOXStates,
↪sampleCount, devResults);
    }else {
            generate_uniform_kernel<<<64, 64>>>(devStates, sampleCount,
↪devResults);
        }
    }

    /* 将设备内存复制到主机 */
    MC_CALL(mcMemcpy(hostResults, devResults, totalThreads *
        sizeof(unsigned int), mcMemcpyDeviceToHost));

    /* 显示结果 */
    total = 0;
    for(i = 0; i < totalThreads; i++) {
        total += hostResults[i];
    }
    printf("Fraction of uniforms > 0.5 was %10.13f\n",
        (float)total / (totalThreads * sampleCount * 50.0f));
    /* 将结果设置为 0*/
    MC_CALL(mcMemset(devResults, 0, totalThreads *
                sizeof(unsigned int)));

    /* 生成和使用正态分布的伪随机参数  */
    for(i = 0; i < 50; i++) {
        if (useMRG) {
            generate_normal_kernel<<<64, 64>>>(devMRGStates, sampleCount,
↪devResults);
        }else if(usePHILOX) {
            generate_normal_kernel<<<64, 64>>>(devPHILOXStates,
↪sampleCount, devResults);
    }else {
            generate_normal_kernel<<<64, 64>>>(devStates, sampleCount,
↪devResults);
        }
    }

    /* 将设备内存复制到主机 */
    MC_CALL(mcMemcpy(hostResults, devResults, totalThreads *
        sizeof(unsigned int), mcMemcpyDeviceToHost));

    /* 显示结果 */
    total = 0;
    for(i = 0; i < totalThreads; i++) {
        total += hostResults[i];
    }
    printf("Fraction of normals within 1 standard deviation was %10.13f\n",
        (float)total / (totalThreads * sampleCount * 50.0f));

    /* 清理 */
    if (useMRG) {
        MC_CALL(mcFree(devMRGStates));
    }else if(usePHILOX)
```

(下页继续)

(续上页)

```
    {
        MC_CALL(mcFree(devPHILOXStates));
    }else {
        MC_CALL(mcFree(devStates));
    }
    MC_CALL(mcFree(devResults));
    free(hostResults);
    printf("^^^^ kernel_example PASSED\n");
    return EXIT_SUCCESS;
}
```

下面的示例使用 mcRAND 的主机 MTGP 的设置 API 和 mcRAND 的设备 API，借助 MTGP32 生成器生成整数，并计算具有低位设置的比例。

```
/*
    * 该程序使用设备 MCRAND API 来计算伪随机整数具有低位设置的比例。
    */
#include <stdio.h>
#include <stdlib.h>
#include <mc_runtime.h>
#include <mcrand_kernel.h>
/* 调用 MTGP 主机的 helper 函数 */
#include <mcrand_mtgp32_host.h>
/* 导入 MTGP 预先计算的参数集 */
#include <mcrand_mtgp32dc_p_11213.h>


#define MC_CALL(x) do { if((x) != mcSuccess) { \
    printf("Error at %s:%d\n",__FILE__,__LINE__); \
    return EXIT_FAILURE;}} while(0)

#define MCRAND_CALL(x) do { if((x) != MCRAND_STATUS_SUCCESS) { \
    printf("Error at %s:%d\n",__FILE__,__LINE__); \
    return EXIT_FAILURE;}} while(0)

__global__ void generate_kernel(mcrandStateMtgp32 *state,
                                int n,
                                int *result)
{
    int id = threadIdx.x + blockIdx.x * 256;
    int count = 0;
    unsigned int x;
    /* 生成伪随机的无符号整数 */
    for(int i = 0; i < n; i++) {
        x = mcrand(&state[blockIdx.x]);
        /* Check if low bit set */
        if(x & 1) {
            count++;
        }
    }
    /* 存储结果 */
    result[id] += count;
}

int main(int argc, char *argv[])
{
```

(下页继续)

(续上页)

```
    int i;
    long long total;
    mcrandStateMtgp32 *devMTGPStates;
    mtgp32_kernel_params *devKernelParams;
    int *devResults, *hostResults;
    int sampleCount = 10000;

    /* 允许覆盖样本计数 */
    if (argc == 2) {
        sscanf(argv[1],"%d",&sampleCount);
    }

    /* 为主机上的结果分配空间 */
    hostResults = (int *)calloc(64 * 256, sizeof(int));

    /* 为设备上的结果分配空间 */
    MC_CALL(mcMalloc((void **)&devResults, 64 * 256 *
                sizeof(int)));

    /* 将结果设置为 0*/
    MC_CALL(mcMemset(devResults, 0, 64 * 256 *
                sizeof(int)));

    /* 为设备上的 prng 状态分配空间 */
    MC_CALL(mcMalloc((void **)&devMTGPStates, 64 *
                sizeof(mcrandStateMtgp32)));

    /* 设置 MTGP 的 prng 状态 */

    /* 为 MTGP 内核参数分配空间 */
    MC_CALL(mcMalloc((void**)&devKernelParams, sizeof(mtgp32_kernel_
↪params)));

    /* 将预定义的参数集重新格式化为内核格式, */
    /* 并将内核参数复制到设备内存中          */
    MCRAND_CALL(mcrandMakeMTGP32Constants(mtgp32dc_params_fast_11213,
↪devKernelParams));

    /* 为每个线程块初始化一个状态 */
    MCRAND_CALL(mcrandMakeMTGP32KernelState(devMTGPStates,
                mtgp32dc_params_fast_11213, devKernelParams, 64, 1234));

    /* 状态设置已完成 */

    /* 生成并使用伪随机参数  */
    for(i = 0; i < 10; i++) {
        generate_kernel<<<64, 256>>>(devMTGPStates, sampleCount,
↪devResults);
    }

    /* 将设备内存复制到主机 */
    MC_CALL(mcMemcpy(hostResults, devResults, 64 * 256 *
        sizeof(int), mcMemcpyDeviceToHost));

    /* 显示结果 */
    total = 0;
```

(下页继续)

(续上页)

```
    for(i = 0; i < 64 * 256; i++) {
        total += hostResults[i];
    }


    printf("Fraction with low bit set was %10.13g\n",
        (double)total / (64.0f * 256.0f * sampleCount * 10.0f));

    /* 清理 */
    MC_CALL(mcFree(devKernelParams));
    MC_CALL(mcFree(devMTGPStates));
    MC_CALL(mcFree(devResults));
    free(hostResults);
    printf("^^^^ kernel_mtgp_example PASSED\n");
    return EXIT_SUCCESS;
}
```

下面的示例使用 mcRAND 设备 API，使用 64 位 Scrambled Sobol 生成器生成均匀的双精度数。它利用生成的结果来推导出一个球体体积的近似值。

```
/*
    * 该程序使用设备 mcRAND API 来计算
    * 准随机 3D 点落在球面内的比例
    * 半径为 1 的球内，并且从中推导出球的体积。
    *
    * 本程序特地使用由
    * mcrandGetDirectionVectors64 返回的 64 位
    * Scrambled Sobol 方向向量来生成双精度均匀样本。
    *
    */

#include <stdio.h>
#include <stdlib.h>
#include <mc_runtime.h>
#include <mcrand_kernel.h>
#include <mcrand.h>

#define THREADS_PER_BLOCK 64
#define BLOCK_COUNT 64
#define TOTAL_THREADS (THREADS_PER_BLOCK * BLOCK_COUNT)

/* 每个维度的 64 位向量数量 */
#define VECTOR_SIZE 64


#define MC_CALL(x) do { if((x) != mcSuccess) { \
    printf("Error at %s:%d\n",__FILE__,__LINE__); \
    return EXIT_FAILURE;}} while(0)

#define MCRAND_CALL(x) do { if((x) != MCRAND_STATUS_SUCCESS) { \
    printf("Error at %s:%d\n",__FILE__,__LINE__); \
    return EXIT_FAILURE;}} while(0)

/* 该内核为每个线程的 x、y 和 z 维度初始化状态 */

__global__ void setup_kernel(unsigned long long * sobolDirectionVectors,
```

(下页继续)

(续上页)

```
                                unsigned long long *sobolScrambleConstants,
                                mcrandStateScrambledSobol64 *state)
{
    int id = threadIdx.x + blockIdx.x * THREADS_PER_BLOCK;
    int dim = 3*id;
    /* 每个线程使用 3 个不同维度 */
    mcrand_init(sobolDirectionVectors + VECTOR_SIZE*dim,
                sobolScrambleConstants[dim],
                1234,
                &state[dim]);

    mcrand_init(sobolDirectionVectors + VECTOR_SIZE*(dim + 1),
                sobolScrambleConstants[dim + 1],
                1234,
                &state[dim + 1]);

    mcrand_init(sobolDirectionVectors + VECTOR_SIZE*(dim + 2),
                sobolScrambleConstants[dim + 2],
                1234,
                &state[dim + 2]);
}

/* 这个内核会生成随机的 3D 点，并且如果一个点在单位球内，
    * 它会将计数器 +1
    */
__global__ void generate_kernel(mcrandStateScrambledSobol64 *state,
                                int n,
                                long long int *result)
{
    int id = threadIdx.x + blockIdx.x * THREADS_PER_BLOCK;
    int baseDim = 3 * id;
    long long int count = 0;
    double x, y, z;

    /* 生成准随机的双精度坐标 */
    for(int i = 0; i < n; i++) {
        x = mcrand_uniform_double(&state[baseDim]);
        y = mcrand_uniform_double(&state[baseDim + 1]);
        z = mcrand_uniform_double(&state[baseDim + 2]);

        /* 检查是否在半径为 1 的球内 */
        if( (x*x + y*y + z*z) < 1.0) {
            count++;
        }
    }
    /* 存储结果 */
    result[id] += count;
}

int main(int argc, char *argv[])
{
    int i;
    long long total;
    mcrandStateScrambledSobol64 *devSobol64States;
    mcrandDirectionVectors64_t *hostVectors64;
    unsigned long long int * hostScrambleConstants64;
```

(下页继续)

(续上页)

```
    unsigned long long int * devDirectionVectors64;
    unsigned long long int * devScrambleConstants64;
    long long int *devResults, *hostResults;
    int sampleCount = 10000;
    int iterations = 100;
    double fraction;
    double pi = 3.1415926535897932;

    /* 允许覆盖样本数量 */
    if (argc == 2) {
        sscanf(argv[1],"%d",&sampleCount);
    }

    /* 为主机上的结果分配空间 */
    hostResults = (long long int*)calloc(TOTAL_THREADS,
                                         sizeof(long long int));

    /* 为设备上的结果分配空间 */
    MC_CALL(mcMalloc((void **)&devResults,
                           TOTAL_THREADS * sizeof(long long int)));

    /* 把结果设置为 0 */
    MC_CALL(mcMemset(devResults, 0,
                           TOTAL_THREADS * sizeof(long long int)));

    /* 获取指向 64 位 Scrambled 方向向量和常数的指针 */
    MCRAND_CALL(mcrandGetDirectionVectors64( &hostVectors64,
                                             MCRAND_SCRAMBLED_DIRECTION_
↪VECTORS_64_JOEKUO6));

    MCRAND_CALL(mcrandGetScrambleConstants64( &hostScrambleConstants64));


    /* 为每个线程的 3 个状态 (x,y,z) 分配内存，每个状态用于获取特定的维度 */
    MC_CALL(mcMalloc((void **)&devSobol64States,
                TOTAL_THREADS * 3 * sizeof(mcrandStateScrambledSobol64)));

    /* 分配内存并将每个线程的 3 组向量复制到设备 */

    MC_CALL(mcMalloc((void **)&(devDirectionVectors64),
                           3 * TOTAL_THREADS * VECTOR_SIZE * sizeof(long
↪long int)));

    MC_CALL(mcMemcpy(devDirectionVectors64, hostVectors64,
                           3 * TOTAL_THREADS * VECTOR_SIZE * sizeof(long
↪long int),
                           mcMemcpyHostToDevice));

    /* 分配内存并将每个线程的 3 个 Scramble 常数（每个维度一个常数）
        复制到设备 */

    MC_CALL(mcMalloc((void **)&(devScrambleConstants64),
                           3 * TOTAL_THREADS * sizeof(long long int)));

    MC_CALL(mcMemcpy(devScrambleConstants64, hostScrambleConstants64,
                           3 * TOTAL_THREADS * sizeof(long long int),
```

(下页继续)

(续上页)

```
                            mcMemcpyHostToDevice));

    /* 初始化状态 */

    setup_kernel<<<BLOCK_COUNT, THREADS_PER_BLOCK>>>(devDirectionVectors64,
↪devScrambleConstants64,
                                                     devSobol64States);

    /* 生成和计数准随机点  */

    for(i = 0; i < iterations; i++) {
        generate_kernel<<<BLOCK_COUNT, THREADS_PER_BLOCK>>>
↪(devSobol64States, sampleCount, devResults);
    }

    /* 将设备内存复制到主机 */

    MC_CALL(mcMemcpy(hostResults,
                devResults,
                TOTAL_THREADS * sizeof(long long int),
                mcMemcpyDeviceToHost));

    /* 统计并显示结果 */

    total = 0;
    for(i = 0; i < TOTAL_THREADS; i++) {
        total += hostResults[i];
    }

    fraction = (double)total / ((double)TOTAL_THREADS *
↪(double)sampleCount * (double)iterations);
    printf("Fraction inside sphere was %g\n", fraction);
    printf("(4/3) pi = %g, sampled volume = %g\n",(4.0*pi/3.0),8.0 *
↪fraction);

    /* 清理 */

    MC_CALL(mcFree(devSobol64States));
    MC_CALL(mcFree(devDirectionVectors64));
    MC_CALL(mcFree(devScrambleConstants64));
    MC_CALL(mcFree(devResults));
    free(hostResults);
    printf("^^^^ kernel_sobol_example PASSED\n");


    return EXIT_SUCCESS;
}
```

3.7 Thrust 和 mcRAND 示例

下面的示例演示了如何混合使用 mcRAND 和 Thrust。这是标准 Thrust 示例之一，monte_carlo.cu 的最小修改版本。该示例通过随机选择单位正方形中的点，并计算其到原点的距离，来判断其是否位于四分之一单位圆内，以估算 π。

```
#include <thrust/iterator/counting_iterator.h>
#include <thrust/functional.h>
#include <thrust/transform_reduce.h>
#include <mcrand_kernel.h>

#include <iostream>
#include <iomanip>

// 我们可以调整 M 和 N 的值来找到性能最佳点

struct estimate_pi :
    public thrust::unary_function<unsigned int, float>
{
    __device__
    float operator()(unsigned int thread_id)
    {
    float sum = 0;
    unsigned int N = 10000; // 每个线程的样本数

    unsigned int seed = thread_id;

    mcrandState s;

    // 给一个随机数生成器设定种子
    mcrand_init(seed, 0, 0, &s);

    // 在四分之一圆内取 N 个样本
    for(unsigned int i = 0; i < N; ++i)
    {
        // 从单位正方形中获取样本
        float x = mcrand_uniform(&s);
        float y = mcrand_uniform(&s);

        // 计算到原点的距离
        float dist = sqrtf(x*x + y*y);

        // 若 (u0,u1) 落在四分之一圆内，则将 1.0f 加入 sum
        if(dist <= 1.0f)
        sum += 1.0f;
    }

    // 乘以 4 得到整个圆的面积
    sum *= 4.0f;

    // 除以 N
    return sum / N;
    }
};

int main(void)
{
    // 使用 30,000 个独立种子
    int M = 30000;

    float estimate = thrust::transform_reduce(
        thrust::counting_iterator<int>(0),
        thrust::counting_iterator<int>(M),
```

(下页继续)

(续上页)

```
        estimate_pi(),
        0.0f,
        thrust::plus<float>());
    estimate /= M;

    std::cout << std::setprecision(3);
    std::cout << "pi is approximately ";
    std::cout << estimate << std::endl;
    return 0;
}
```

3.8 泊松分布 API 示例

这个示例展示了泊松分布的三种 API 类型之间的区别。它是一个模拟商店排队的例子。主机 API (host API) 对于生成泊松分布随机数的大向量是最稳健的（也就是它在 lambda 值的整个范围内具有最好的统计性质）。离散设备 API(discrete Device API) 几乎和主机 API 一样健壮，并允许在内核中生成泊松分布的随机数。简单设备 API(simple Device API) 是最不稳健的，但在为许多不同的 lambda 值生成泊松分布的随机数时更高效。

```
/*
    * 这个程序使用 MCRAND 库的泊松分布
    * 来模拟商店的队列，模拟时间为 16 小时。
    * 它展示了使用 3 种不同的 API 的区别：
    * - HOST API -顾客到达的频率由 Poisson(4) 来描述
    * - SIMPLE DEVICE API -顾客到达的频率
    *     由 Poisson(4*(sin(x/100)+1)) 来描述，
    *     其中 x 是从开店时间开始计算的分钟数。
    * - ROBUST DEVICE API -顾客到达的频率如下：
    *     - 前 3 小时内由 Poisson(2) 来描述。
    *     - 接下来的 3 小时内由 Poisson(1) 来描述。
    *     - 6 小时后由 Poisson(3) 来描述。
    */

#include <stdio.h>
#include <stdlib.h>
#include <mc_runtime.h>
#include <mcrand_kernel.h>
#include <mcrand.h>

#define MC_CALL(x) do { if((x) != mcSuccess) { \
    printf("Error at %s:%d\n",__FILE__,__LINE__); \
    return EXIT_FAILURE;}} while(0)
#define MCRAND_CALL(x) do { if((x)!=MCRAND_STATUS_SUCCESS) { \
    printf("Error at %s:%d\n",__FILE__,__LINE__);\
    return EXIT_FAILURE;}} while(0)


#define HOURS 16
#define OPENING_HOUR 7
#define CLOSING_HOUR (OPENING_HOUR + HOURS)

#define access_2D(type, ptr, row, column, pitch)\
    *((type*)((char*)ptr + (row) * pitch) + column)
```

(下页继续)

(续上页)

```
enum API_TYPE {
    HOST_API = 0,
    SIMPLE_DEVICE_API = 1,
    ROBUST_DEVICE_API = 2,
};

/* 全局变量 */
API_TYPE api;
int report_break;
int cashiers_load_h[HOURS];
__constant__ int cashiers_load[HOURS];

__global__ void setup_kernel(mcrandState *state)
{
    int id = threadIdx.x + blockIdx.x * blockDim.x;
    /* 每个线程使用相同的种子，不同的
       序列号，无偏移 */
    mcrand_init(1234, id, 0, &state[id]);
}

__inline__ __device__
void update_queue(int id, int min, unsigned int new_customers,
                  unsigned int &queue_length,
                  unsigned int *queue_lengths, size_t pitch)
{
    int balance;
    balance = new_customers - 2 * cashiers_load[(min-1)/60];
    if (balance + (int)queue_length <= 0){
        queue_length = 0;
    }else{
        queue_length += balance;
    }
    /* 存储结果 */
    access_2D(unsigned int, queue_lengths, min-1, id, pitch)
      = queue_length;
}


__global__ void simple_device_API_kernel(mcrandState *state,
                  unsigned int *queue_lengths, size_t pitch)
{
    int id = threadIdx.x + blockIdx.x * blockDim.x;
    unsigned int new_customers;
    unsigned int queue_length = 0;
    /* 将状态复制到本地内存，以提高效率 */
    mcrandState localState = state[id];
    /* 在时间上模拟排队 */
    for(int min = 1; min <= 60 * HOURS; min++) {
        /* 根据 API 得出新顾客数量 */
        new_customers = mcrand_poisson(&localState,
                                4*(sin((float)min/100.0)+1));
        /* 更新队列 */
        update_queue(id, min, new_customers, queue_length,
                     queue_lengths, pitch);
    }
```

(下页继续)

(续上页)

```
    /* 将状态复制回全局内存 */
    state[id] = localState;
}


__global__ void host_API_kernel(unsigned int *poisson_numbers,
                    unsigned int *queue_lengths, size_t pitch)
{
    int id = threadIdx.x + blockIdx.x * blockDim.x;
    unsigned int new_customers;
    unsigned int queue_length = 0;
    /* 按时间模拟排队 */
    for(int min = 1; min <= 60 * HOURS; min++) {
        /* 从全局内存中获取随机数 */
        new_customers = poisson_numbers
                    [blockDim.x * gridDim.x * (min -1) + id];
        /* 更新队列 */
        update_queue(id, min, new_customers, queue_length,
                        queue_lengths, pitch);
    }
}

__global__ void robust_device_API_kernel(mcrandState *state,
                    mcrandDiscreteDistribution_t poisson_1,
                    mcrandDiscreteDistribution_t poisson_2,
                    mcrandDiscreteDistribution_t poisson_3,
                    unsigned int *queue_lengths, size_t pitch)
{
    int id = threadIdx.x + blockIdx.x * 64;
    unsigned int new_customers;
    unsigned int queue_length = 0;
    /* 将状态从复制到局部内存，以提高效率 */
    mcrandState localState = state[id];
    /* 按时间模拟排队 */
    /* 前 3 小时 */
    for(int min = 1; min <= 60 * 3; min++) {
        /* 根据 API 得出新顾客数量 */
        new_customers =
                    mcrand_discrete(&localState, poisson_2);
        /* 更新队列 */
        update_queue(id, min, new_customers, queue_length,
                                    queue_lengths, pitch);
    }
    /* 接下来的 3 小时 */
    for(int min = 60 * 3 + 1; min <= 60 * 6; min++) {
        /* 根据 API 得出顾客数量 */
        new_customers =
                    mcrand_discrete(&localState, poisson_1);
        /* 更新队列 */
        update_queue(id, min, new_customers, queue_length,
                                    queue_lengths, pitch);
    }
    /* 6 小时后 */
    for(int min = 60 * 6 + 1; min <= 60 * HOURS; min++) {
        /* 根据 API 得出新顾客数量 */
        new_customers =
```

(下页继续)

(续上页)

```
                    mcrand_discrete(&localState, poisson_3);
        /* 更新队列 */
        update_queue(id, min, new_customers, queue_length,
                                    queue_lengths, pitch);
    }
    /* 将状态复制回全局内存 */
    state[id] = localState;
}

/* 设置报告时间间隔 */
void report_settings()
{
    do{
        printf("Set time intervals between queue reports");
        printf("(in minutes > 0)\n");
        if (scanf("%d", &report_break) == 0) continue;
    }while(report_break <= 0);
}


/* 设置每小时的收银员数量 */
void add_cachiers(int *cashiers_load)
{
    int i, min, max, begin, end;
    printf("Cashier serves 2 customers per minute...\n");
    for (i = 0; i < HOURS; i++){
        cashiers_load_h[i] = 0;
    }
    while (true){
        printf("Adding cashier...\n");
        min = OPENING_HOUR;
        max = CLOSING_HOUR-1;
        do{
            printf("Set hour that cahier comes (%d-%d)",
                                                min, max);
            printf(" [type 0 to finish adding cashiers]\n");
            if (scanf("%d", &begin) == 0) continue;
        }while (begin > max || (begin < min && begin != 0));
        if (begin == 0) break;
        min = begin+1;
        max = CLOSING_HOUR;
        do{
            printf("Set hour that cahier leaves (%d-%d)",
                                                min, max);
            printf(" [type 0 to finish adding cashiers]\n");
            if (scanf("%d", &end) == 0) continue;
        }while (end > max || (end < min && end != 0));
        if (end == 0) break;
        for (i = begin - OPENING_HOUR;
                i < end   - OPENING_HOUR; i++){
            cashiers_load_h[i]++;
        }
    }
    for (i = OPENING_HOUR; i < CLOSING_HOUR; i++){
        printf("\n%2d:00 - %2d:00     %d cashier",
                i, i+1, cashiers_load_h[i-OPENING_HOUR]);
```

(下页继续)

(续上页)

```
        if (cashiers_load[i-OPENING_HOUR] != 1) printf("s");
    }
    printf("\n");
}

/* 设置 API 类型 */
API_TYPE set_API_type()
{
    printf("Choose API type:\n");
    int choose;
    do{
        printf("type 1 for HOST API\n");
        printf("type 2 for SIMPLE DEVICE API\n");
        printf("type 3 for ROBUST DEVICE API\n");
        if (scanf("%d", &choose) == 0) continue;
    }while( choose < 1 || choose > 3);
    switch(choose){
        case 1: return HOST_API;
        case 2: return SIMPLE_DEVICE_API;
        case 3: return ROBUST_DEVICE_API;
        default:
            fprintf(stderr, "wrong API\n");
            return HOST_API;
    }
}

void settings()
{
    add_cachiers(cashiers_load);
    mcMemcpyToSymbol("cashiers_load", cashiers_load_h,
          HOURS * sizeof(int), 0, mcMemcpyHostToDevice);
    report_settings();
    api = set_API_type();
}

void print_statistics(unsigned int *hostResults, size_t pitch)
{
    int min, i, hour, minute;
    unsigned int sum;
    for(min = report_break; min <= 60 * HOURS;
                            min += report_break) {
        sum = 0;
        for(i = 0; i < 64 * 64; i++) {
            sum += access_2D(unsigned int, hostResults,
                                      min-1, i, pitch);
        }
        hour = OPENING_HOUR + min/60;
        minute = min%60;
        printf("%2d:%02d   # of waiting customers = %10.4g |",
                    hour, minute, (float)sum/(64.0 * 64.0));
        printf("  # of cashiers = %d  |  ",
                    cashiers_load_h[(min-1)/60]);
        printf("# of new customers/min ~= ");
        switch (api){
            case HOST_API:
                printf("%2.2f\n", 4.0);
```

(下页继续)

(续上页)

```
                break;
            case SIMPLE_DEVICE_API:
                printf("%2.2f\n",
                            4*(sin((float)min/100.0)+1));
                break;
            case ROBUST_DEVICE_API:
                if (min <= 3 * 60){
                    printf("%2.2f\n", 2.0);
                }else{
                    if (min <= 6 * 60){
                        printf("%2.2f\n", 1.0);
                    }else{
                        printf("%2.2f\n", 3.0);
                    }
                }
                break;
            default:
                fprintf(stderr, "Wrong API\n");
        }
    }
}


int main(int argc, char *argv[])
{
    int n;
    size_t pitch;
    mcrandState *devStates;
    unsigned int *devResults, *hostResults;
    unsigned int *poisson_numbers_d;
    mcrandDiscreteDistribution_t poisson_1, poisson_2;
    mcrandDiscreteDistribution_t poisson_3;
    mcrandGenerator_t gen;

    /* 设置收银员、汇报频率 (report) 和 API */
    settings();

    /* 为设备上的结果分配空间 */
    MC_CALL(mcMallocPitch((void **)&devResults, &pitch,
                64 * 64 * sizeof(unsigned int), 60 * HOURS));

    /* 为主机上的结果分配空间 */
    hostResults = (unsigned int *)calloc(pitch * 60 * HOURS,
                sizeof(unsigned int));

    /* 为设备上的 prng 状态分配空间 */
    MC_CALL(mcMalloc((void **)&devStates, 64 * 64 *
                sizeof(mcrandState)));

    /* 设置 prng 状态 */
    if (api != HOST_API){
        setup_kernel<<<64, 64>>>(devStates);
    }
    /* 模拟排队  */
    switch (api){
        case HOST_API:
```

(下页继续)

(续上页)

```
            /* 创建伪随机数生成器 */
            MCRAND_CALL(mcrandCreateGenerator(&gen,
                                MCRAND_RNG_PSEUDO_DEFAULT));
            /* 设置种子 */
            MCRAND_CALL(mcrandSetPseudoRandomGeneratorSeed(
                                            gen, 1234ULL));
            /* 计算 n */
            n = 64 * 64 * HOURS * 60;
            /* 在设备上分配 n 个无符号整数 */
            MC_CALL(mcMalloc((void **)&poisson_numbers_d,
                                n * sizeof(unsigned int)));
            /* 在设备上生成 n 个无符号整数 */
            MCRAND_CALL(mcrandGeneratePoisson(gen,
                                poisson_numbers_d, n, 4.0));
            host_API_kernel<<<64, 64>>>(poisson_numbers_d,
                                        devResults, pitch);
            /* 清理 */
            MCRAND_CALL(mcrandDestroyGenerator(gen));
            break;
        case SIMPLE_DEVICE_API:
            simple_device_API_kernel<<<64, 64>>>(devStates,
                                        devResults, pitch);
            break;
        case ROBUST_DEVICE_API:
            /* 创建 Poisson(1) 的直方图 */
            MCRAND_CALL(mcrandCreatePoissonDistribution(1.0,
                                                &poisson_1));
            /* 创建 Poisson(2) 的直方图 */
            MCRAND_CALL(mcrandCreatePoissonDistribution(2.0,
                                                &poisson_2));
            /* 创建 Poisson(3) 的直方图 */
            MCRAND_CALL(mcrandCreatePoissonDistribution(3.0,
                                                &poisson_3));
            robust_device_API_kernel<<<64, 64>>>(devStates,
                            poisson_1, poisson_2, poisson_3,
                            devResults, pitch);
            /* 清理 */
            MCRAND_CALL(mcrandDestroyDistribution(poisson_1));
            MCRAND_CALL(mcrandDestroyDistribution(poisson_2));
            MCRAND_CALL(mcrandDestroyDistribution(poisson_3));
            break;
        default:
            fprintf(stderr, "Wrong API\n");
    }
    /* 将设备内存复制到主机 */
    MC_CALL(mcMemcpy2D(hostResults, pitch, devResults,
              pitch, 64 * 64 * sizeof(unsigned int),
              60 * HOURS, mcMemcpyDeviceToHost));
    /* 显示结果 */
    print_statistics(hostResults, pitch);
    /* 清理 */
    MC_CALL(mcFree(devStates));
    MC_CALL(mcFree(devResults));
    free(hostResults);
    return EXIT_SUCCESS;
}
```

4 模块

以下是所有模块的列表:

4.1 主机 API

4.1.1 函数列表

```
mcrandStatus_t MCRANDAPI mcrandCreateGenerator ( mcrandGenerator_t*
↪generator, mcrandRngType_t rng_type )
```

创建新的随机数生成器。

```
mcrandStatus_t MCRANDAPI mcrandCreateGeneratorHost ( mcrandGenerator_t*
↪generator, mcrandRngType_t rng_type )
```

创建新的主机 CPU 随机数生成器。

```
mcrandStatus_t MCRANDAPI mcrandCreatePoissonDistribution ( double  lambda,
↪mcrandDiscreteDistribution_t* discrete_distribution )
```

构建泊松分布的直方图数组。

```
mcrandStatus_t MCRANDAPI mcrandDestroyDistribution (
↪mcrandDiscreteDistribution_t discrete_distribution )
```

销毁离散分布（例如泊松分布）的直方图数组。

```
mcrandStatus_t MCRANDAPI mcrandDestroyGenerator ( mcrandGenerator_t
↪generator )
```

销毁一个已有的生成器。

```
mcrandStatus_t MCRANDAPI mcrandGenerate ( mcrandGenerator_t generator,
↪unsigned int* outputPtr, size_t num )
```

生成 32 位的伪随机或准随机数。

```
mcrandStatus_t MCRANDAPI mcrandGenerateLogNormal ( mcrandGenerator_t
↪generator, float* outputPtr, size_t n, float  mean, float  stddev )
```

生成对数正态分布的浮点数。

```
mcrandStatus_t MCRANDAPI mcrandGenerateLogNormalDouble ( mcrandGenerator_t
↪generator, double* outputPtr, size_t n, double  mean, double  stddev )
```

生成对数正态分布的双精度浮点数。

```
mcrandStatus_t MCRANDAPI mcrandGenerateLongLong ( mcrandGenerator_t
↪generator, unsigned long long* outputPtr, size_t num )
```

生成 64 位的准随机数。

```
mcrandStatus_t MCRANDAPI mcrandGenerateNormal ( mcrandGenerator_t
↪generator, float* outputPtr, size_t n, float  mean, float  stddev )
```

生成正态分布的双精度浮点数。

```
mcrandStatus_t MCRANDAPI mcrandGenerateNormalDouble ( mcrandGenerator_t
↪generator, double* outputPtr, size_t n, double  mean, double  stddev )
```

生成正态分布的双精度浮点数。

```
mcrandStatus_t MCRANDAPI mcrandGeneratePoisson ( mcrandGenerator_t
↪generator, unsigned int* outputPtr, size_t n, double  lambda )
```

生成泊松分布的无符号整数。

```
mcrandStatus_t MCRANDAPI mcrandGenerateSeeds ( mcrandGenerator_t generator
↪)
```

设置起始状态。

```
mcrandStatus_t MCRANDAPI mcrandGenerateUniform ( mcrandGenerator_t
↪generator, float* outputPtr, size_t num )
```

生成均匀分布的浮点数。

```
mcrandStatus_t MCRANDAPI mcrandGenerateUniformDouble ( mcrandGenerator_t
↪generator, double* outputPtr, size_t num )
```

生成均匀分布的双精度浮点数。

```
mcrandStatus_t MCRANDAPI mcrandGetDirectionVectors32 (
↪mcrandDirectionVectors32_t* vectors, mcrandDirectionVectorSet_t set )
```

获取 32 位准随机数生成的方向向量。

```
mcrandStatus_t MCRANDAPI mcrandGetDirectionVectors64 (
↪mcrandDirectionVectors64_t* vectors, mcrandDirectionVectorSet_t set )
```

获取 64 位准随机数生成的方向向量。

```
mcrandStatus_t MCRANDAPI mcrandGetProperty ( libraryPropertyType type,
↪int* value )
```

返回 mcrand 属性的值。

```
mcrandStatus_t MCRANDAPI mcrandGetScrambleConstants32 ( unsigned int**
↪constants )
```

获取用于 32 位 Scrambled Sobol 序列的 Scramble 常数。

```
mcrandStatus_t MCRANDAPI mcrandGetScrambleConstants64 ( unsigned long
↪long** constants )
```

获取用于 64 位 Scrambled Sobol 序列的 Scramble 常数。

```
mcrandStatus_t MCRANDAPI mcrandGetVersion ( int* version )
```

返回库的版本号。

```
mcrandStatus_t MCRANDAPI mcrandSetGeneratorOffset ( mcrandGenerator_t
↪generator, unsigned long long offset )
```

设置伪或准随机数生成器的绝对偏移量。

```
mcrandStatus_t MCRANDAPI mcrandSetGeneratorOrdering ( mcrandGenerator_t
↪generator, mcrandOrdering_t order )
```

设置伪或准随机数生成器的结果排序。

```
mcrandStatus_t MCRANDAPI mcrandSetPseudoRandomGeneratorSeed (
↪mcrandGenerator_t generator, unsigned long long seed )
```

设置伪随机数生成器的种子值。

```
mcrandStatus_t MCRANDAPI mcrandSetQuasiRandomGeneratorDimensions (
↪mcrandGenerator_t generator, unsigned int  num_dimensions )
```

设置维度数量。

```
mcrandStatus_t MCRANDAPI mcrandSetStream ( mcrandGenerator_t generator,
↪mcStream_t stream )
```

为 MCRAND 内核启动设置当前流。

4.1.2 枚举

```
enum HOST::mcrandDirectionVectorSet [inherited]
```

MCRAND 方向向量集的选择

值

```
MCRAND_DIRECTION_VECTORS_32_JOEKUO6 = 101
```

根据 S. Joe 和 F. Y. Kuo 推荐的多项式生成的特定 32 位方向向量集，适用于高达 20000 个维度。

```
MCRAND_SCRAMBLED_DIRECTION_VECTORS_32_JOEKUO6 = 102
```

根据 S. Joe 和 F. Y. Kuo 推荐的多项式生成的特定 32 位方向向量集，适用于高达 20000 个维度，并进行了混淆处理。

```
MCRAND_DIRECTION_VECTORS_64_JOEKUO6 = 103
```

根据 S. Joe 和 F. Y. Kuo 推荐的多项式生成的特定 64 位方向向量集，适用于高达 20000 个维度。

```
MCRAND_SCRAMBLED_DIRECTION_VECTORS_64_JOEKUO6 = 104
```

根据 S. Joe 和 F. Y. Kuo 推荐的多项式生成的特定 32 位方向向量集，适用于高达 20000 个维度，并进行了混淆处理。

```
enum HOST::mcrandOrdering [inherited]
```

MCRAND 在内存中的结果排序

值

MCRAND_ORDERING_PSEUDO_BEST = 100

伪随机结果的最佳排序。

MCRAND_ORDERING_PSEUDO_DEFAULT = 101

伪随机结果的特定默认线程顺序，与 MCRAND_ORDERING_PSEUDO_BEST 相同。

MCRAND_ORDERING_PSEUDO_SEEDED = 102

用于获得快速、低质量伪随机结果的特定种子模式。

MCRAND_ORDERING_PSEUDO_LEGACY = 103

伪随机结果的特定遗留序列，保证在所有 mcRAND 发布中保持相同。

MCRAND_ORDERING_PSEUDO_DYNAMIC = 104

根据执行设备进行调整的特定顺序，提供最佳性能。

MCRAND_ORDERING_QUASI_DEFAULT = 201

用于拟随机结果的特定 n 维顺序。

enum HOST::mcrandRngType [inherited]

MCRAND 生成器类型

值

MCRAND_RNG_TEST = 0

MCRAND_RNG_PSEUDO_DEFAULT = 100

默认伪随机生成器。

MCRAND_RNG_PSEUDO_XORWOW = 101

XORWOW 伪随机生成器。

MCRAND_RNG_PSEUDO_MRG32K3A = 121

MRG32k3a 伪随机生成器。

MCRAND_RNG_PSEUDO_MTGP32 = 141

Mersenne Twister MTGP32 伪随机生成器。

MCRAND_RNG_PSEUDO_MT19937 = 142

Mersenne Twister MT19937 伪随机生成器。

MCRAND_RNG_PSEUDO_PHILOX4_32_10 = 161

PHILOX-4x32-10 伪随机生成器。

MCRAND_RNG_QUASI_DEFAULT = 200

默认准随机生成器

```
MCRAND_RNG_QUASI_SOBOL32 = 201
```

Sobol32 准随机生成器

```
MCRAND_RNG_QUASI_SCRAMBLED_SOBOL32 = 202
```

Scrambled Sobol32 准随机生成器

```
MCRAND_RNG_QUASI_SOBOL64 = 203
```

Sobol64 准随机生成器

```
MCRAND_RNG_QUASI_SCRAMBLED_SOBOL64 = 204
```

Scrambled Sobol64 准随机生成器

```
enum HOST::mcrandStatus [inherited]
```

MCRAND 函数调用状态类型

值

```
MCRAND_STATUS_SUCCESS = 0
```

无错误。

```
MCRAND_STATUS_VERSION_MISMATCH = 100
```

头文件和链接库版本不匹配。

```
MCRAND_STATUS_NOT_INITIALIZED = 101
```

生成器未初始化。

```
MCRAND_STATUS_ALLOCATION_FAILED = 102
```

内存分配失败。

```
MCRAND_STATUS_TYPE_ERROR = 103
```

生成器类型错误。

```
MCRAND_STATUS_OUT_OF_RANGE = 104
```

参数超出范围。

```
MCRAND_STATUS_LENGTH_NOT_MULTIPLE = 105
```

请求的长度不是维度的倍数。

```
MCRAND_STATUS_DOUBLE_PRECISION_REQUIRED = 106
```

GPU 没有 MRG32k3a 所需的双精度浮点数。

```
MCRAND_STATUS_LAUNCH_FAILURE = 201
```

内核启动错误。

```
MCRAND_STATUS_PREEXISTING_FAILURE = 202
```

库入口先前存在的故障。

```
MCRAND_STATUS_INITIALIZATION_FAILED = 203
```

MXMACA 初始化失败。

```
MCRAND_STATUS_ARCH_MISMATCH = 204
```

结构不匹配，GPU 不支持请求的功能。

```
MCRAND_STATUS_INTERNAL_ERROR = 999
```

库内部错误。

4.1.3 函数

```
mcrandStatus_t MCRANDAPI mcrandCreateGenerator ( mcrandGenerator_t*
↪generator, mcrandRngType_t rng_type )
```

创建一个新的随机数生成器。

参数

generator

- 生成器的指针

rng_type

- 要创建的生成器类型

返回值

- 如果无法分配内存，则返回 MCRAND_STATUS_ALLOCATION_FAILED
- 如果 GPU 设置出现问题，则返回 MCRAND_STATUS_INITIALIZATION_FAILED
- 如果头文件版本与动态链接库版本不匹配，则返回 MCRAND_STATUS_VERSION_MISMATCH
- 如果 rng_type 的值无效，则返回 MCRAND_STATUS_TYPE_ERROR
- 如果生成器创建成功，则返回 MCRAND_STATUS_SUCCESS

描述

创建了一个 rng_type 类型的新随机数生成器，并将其返回到 *generator 中。

rng_type 的合法值包括：

- MCRAND_RNG_PSEUDO_DEFAULT
- MCRAND_RNG_PSEUDO_XORWOW
- MCRAND_RNG_PSEUDO_MRG32K3A
- MCRAND_RNG_PSEUDO_MTGP32
- MCRAND_RNG_PSEUDO_MT19937
- MCRAND_RNG_PSEUDO_PHILOX4_32_10
- MCRAND_RNG_QUASI_DEFAULT
- MCRAND_RNG_QUASI_SOBOL32
- MCRAND_RNG_QUASI_SCRAMBLED_SOBOL32
- MCRAND_RNG_QUASI_SOBOL64
- MCRAND_RNG_QUASI_SCRAMBLED_SOBOL64

当 rng_type 为 MCRAND_RNG_PSEUDO_DEFAULT 时，选择的类型是 MCRAND_RNG_PSEUDO_XORWOW。当 rng_type 为 MCRAND_RNG_QUASI_DEFAULT 时，选择的类型是 MCRAND_RNG_QUASI_SOBOL32。

rng_type = MCRAND_RNG_PSEUDO_XORWOW 的默认值为：

- seed = 0
- offset = 0
- ordering = MCRAND_ORDERING_PSEUDO_DEFAULT

rng_type = MCRAND_RNG_PSEUDO_MRG32K3A 的默认值为：

- seed = 0
- offset = 0
- ordering = MCRAND_ORDERING_PSEUDO_DEFAULT

rng_type = MCRAND_RNG_PSEUDO_MTGP32 的默认值为：

- seed = 0
- offset = 0
- ordering = MCRAND_ORDERING_PSEUDO_DEFAULT

rng_type = MCRAND_RNG_PSEUDO_MT19937 的默认值为：

- seed = 0
- offset = 0
- ordering = MCRAND_ORDERING_PSEUDO_DEFAULT

rng_type = MCRAND_RNG_PSEUDO_PHILOX4_32_10 的默认值为：

- seed = 0
- offset = 0
- ordering = MCRAND_ORDERING_PSEUDO_DEFAULT

rng_type = MCRAND_RNG_QUASI_SOBOL32 的默认值为：

- dimensions = 1
- offset = 0
- ordering = MCRAND_ORDERING_QUASI_DEFAULT

rng_type = MCRAND_RNG_QUASI_SOBOL64 的默认值为：

- dimensions = 1
- offset = 0
- ordering = MCRAND_ORDERING_QUASI_DEFAULT

rng_type = MCRAND_RNG_QUASI_SCRAMBBLED_SOBOL32 的默认值为：

- dimensions = 1
- offset = 0
- ordering = MCRAND_ORDERING_QUASI_DEFAULT

rng_type = MCRAND_RNG_QUASI_SCRAMBLED_SOBOL64 的默认值为：

- dimensions = 1
- offset = 0
- ordering = MCRAND_ORDERING_QUASI_DEFAULT

```
mcrandStatus_t MCRANDAPI mcrandCreateGeneratorHost ( mcrandGenerator_t*
↪generator, mcrandRngType_t rng_type )
```

创建新的主机 CPU 随机数生成器。

参数

generator

- 生成器的指针

rng_type

- 要创建的生成器类型

返回值

- 如果无法分配内存，则返回 MCRAND_STATUS_ALLOCATION_FAILED
- 如果 GPU 设置出现问题，则返回 MCRAND_STATUS_INITIALIZATION_FAILED
- 如果头文件版本与动态链接库版本不匹配，则返回 MCRAND_STATUS_VERSION_MISMATCH
- 如果 rng_type 的值无效，则返回 MCRAND_STATUS_TYPE_ERROR
- 如果生成器创建成功，则返回 MCRAND_STATUS_SUCCESS

描述

创建一个新的类型为 rng_type 的主机 CPU 随机数生成器，并将其返回到 *generator 中。

rng_type 的合法值为：

- MCRAND_RNG_PSEUDO_DEFAULT
- MCRAND_RNG_PSEUDO_XORWOW
- MCRAND_RNG_PSEUDO_MRG32K3A
- MCRAND_RNG_PSEUDO_MTGP32
- MCRAND_RNG_PSEUDO_MT19937
- MCRAND_RNG_PSEUDO_PHILOX4_32_10
- MCRAND_RNG_QUASI_DEFAULT
- MCRAND_RNG_QUASI_SOBOL32

当 rng_type 为 MCRAND_RNG_PSEUDO_DEFAULT 时，选择的类型为 MCRAND_RNG_PSEUDO_XORWOW。当 rng_type 为 MCRAND_RNG_QUASI_DEFAULT 时，选择的类型为 MCRAND_RNG_QUASI_SOBOL32。

rng_type = MCRAND_RNG_PSEUDO_XORWOW 的默认值为：

- seed = 0
- offset = 0
- ordering = MCRAND_ORDERING_PSEUDO_DEFAULT

rng_type = MCRAND_RNG_PSEUDO_MRG32K3A 的默认值为：

- seed = 0
- offset = 0
- ordering = MCRAND_ORDERING_PSEUDO_DEFAULT

rng_type = MCRAND_RNG_PSEUDO_MTGP32 的默认值为：

- seed = 0
- offset = 0

- ordering = MCRAND_ORDERING_PSEUDO_DEFAULT

rng_type = MCRAND_RNG_PSEUDO_MT19937 的默认值为：

- seed = 0
- offset = 0
- ordering = MCRAND_ORDERING_PSEUDO_DEFAULT

rng_type = MCRAND_RNG_PSEUDO_PHILOX4_32_10 的默认值为：

- seed = 0
- offset = 0
- ordering = MCRAND_ORDERING_PSEUDO_DEFAULT

rng_type = MCRAND_RNG_QUASI_SOBOL32 的默认值为：

- dimensions = 1
- offset = 0
- ordering = MCRAND_ORDERING_QUASI_DEFAULT

rng_type = MCRAND_RNG_QUASI_SOBOL64 的默认值为：

- dimensions = 1
- offset = 0
- ordering = MCRAND_ORDERING_QUASI_DEFAULT

rng_type = MCRAND_RNG_QUASI_SCRAMBLED_SOBOL32 的默认值为：

- dimensions = 1
- offset = 0
- ordering = MCRAND_ORDERING_QUASI_DEFAULT

rng_type = MCRAND_RNG_QUASI_SCRAMBLED_SOBOL64 的默认值为：

- dimensions = 1
- offset = 0
- ordering = MCRAND_ORDERING_QUASI_DEFAULT

```
mcrandStatus_t MCRANDAPI mcrandCreatePoissonDistribution ( double  lambda,
↪mcrandDiscreteDistribution_t* discrete_distribution )
```

构建泊松分布的直方图数组。

参数

lambda

- 泊松分布的 lambda

discrete_distribution

- 指向设备内存中直方图的指针

返回值

- 如果无法分配内存，则返回 MCRAND_STATUS_ALLOCATION_FAILED
- 如果 GPU 不支持双精度浮点数，则返回 MCRAND_STATUS_DOUBLE_PRECISION_REQUIRED
- 如果 GPU 设置有问题，则返回 MCRAND_STATUS_INITIALIZATION_FAILED
- 如果分布指针为 null，则返回 MCRAND_STATUS_NOT_INITIALIZED
- 如果先前的内核启动存在错误，则返回 MCRAND_STATUS_PREEXISTING_FAILURE

- 如果 lambda 非正或大于 400,000，则返回 MCRAND_STATUS_OUT_OF_RANGE
- 如果直方图生成成功，则返回 MCRAND_STATUS_SUCCESS

描述

构造泊松分布的直方图数组，lambda 的值为 lambda。如果 lambda 大于 2000，则使用近似的正态分布。

```
mcrandStatus_t MCRANDAPI mcrandDestroyDistribution (
↪mcrandDiscreteDistribution_t discrete_distribution )
```

销毁离散分布（例如泊松分布）的直方图数组。

参数

discrete_distribution

- 指向存储直方图的设备内存的指针

返回值

- 如果生成器未创建，则返回 MCRAND_STATUS_NOT_INITIALIZED
- 如果直方图销毁成功，则返回 MCRAND_STATUS_SUCCESS

描述

销毁由 mcrandCreatePoissonDistribution 创建的离散分布的直方图数组。

```
mcrandStatus_t MCRANDAPI mcrandDestroyGenerator ( mcrandGenerator_t
↪generator )
```

销毁一个现有的生成器。

参数

generator

- 待销毁的生成器

返回值

- 如果生成器未创建，则返回 MCRAND_STATUS_NOT_INITIALIZED
- 如果直方图销毁成功，则返回 MCRAND_STATUS_SUCCESS

描述

销毁现有生成器并释放与其状态相关的所有内存。

```
mcrandStatus_t MCRANDAPI mcrandGenerate ( mcrandGenerator_t generator,
↪unsigned int* outputPtr, size_t num )
```

生成 32 位伪随机数或准随机数。

参数

generator

- 可用的生成器

outputPtr

- 指向设备内存的指针，用于存储 MXMACA 生成的结果，或者指向主机内存的指针，用于存储 CPU 生成的结果

num

- 生成的 32 位随机值的个数

返回值

- 如果无法分配内存，则返回 MCRAND_STATUS_ALLOCATION_FAILED
- 如果生成器未创建，则返回 MCRAND_STATUS_NOT_INITIALIZED
- 如果先前的内核启动存在错误，则返回 MCRAND_STATUS_PREEXISTING_FAILURE
- 如果输出样本的数量不是准随机维数的倍数，则返回 MCRAND_STATUS_LENGTH_NOT_MULTIPLE
- 如果内核因某种原因启动失败，则返回 MCRAND_STATUS_LAUNCH_FAILURE
- 如果生成器是 64 位的准随机生成器（使用 64 位准随机生成器的 mcrandGenerateLongLong()），则返回 MCRAND_STATUS_TYPE_ERROR
- 如果成功生成，则返回 MCRAND_STATUS_SUCCESS

描述

使用 generator 在 outputPtr 的设备内存中生成 num 个 32 位随机数。设备内存必须是预先分配的，并且足够大以容纳所有结果。启动是使用 mcrandSetStream() 设置的流完成的，如果没有设置流，则为空流。

结果是 32 位值，每一位都是随机的。

```
mcrandStatus_t MCRANDAPI mcrandGenerateLogNormal ( mcrandGenerator_t
↪generator, float* outputPtr, size_t n, float  mean, float  stddev )
```

生成对数正态分布的浮点数。

参数

generator

- 可用的生成器

outputPtr

- 指向设备内存的指针，用于存储 MXMACA 生成的结果，或者指向主机内存的指针，用于存储 CPU 生成的结果

n

- 要生成的浮点数个数

mean

- 正态分布的均值

stddev

- 正态分布的标准差

返回值

- 如果无法分配内存，则返回 MCRAND_STATUS_ALLOCATION_FAILED
- 如果生成器未创建，则返回 MCRAND_STATUS_NOT_INITIALIZED
- 如果先前的内核启动存在错误，则返回 MCRAND_STATUS_PREEXISTING_FAILURE
- 如果内核因某种原因启动失败，则返回 MCRAND_STATUS_LAUNCH_FAILURE
- 如果输出样本的数量不是准随机维数的倍数或不是伪随机生成器的 2 的倍数，则返回 MCRAND_STATUS_LENGTH_NOT_MULTIPLE
- 如果成功生成，则返回 MCRAND_STATUS_SUCCESS

描述

使用 generator 将 n 个浮点数结果生成到设备内存的 outputPtr 位置。设备内存必须是预先分配的，并且足够大以容纳所有结果。启动是使用 mcrandSetStream() 设置的流完成的，如果没有设置流，则为空流。

结果是 32 位浮点值，其平均值为 mean，标准偏差为 stddev。

通常，使用伪随机数生成器和 Box-Muller 变换来生成正态分布的结果，因此要求 n 为偶数。准随机生成器使用逆累积分布函数来保持维数。生成的正态分布结果会转换为对数正态分布。

使用 mcrandCreateGenerator() 创建的生成器在 GPU 上生成的结果与使用 mcrandCreateGeneratorHost() 创建的生成器在 CPU 上计算的结果之间可能会有轻微的数值差异。这些差异是由于超越函数计算结果的差异造成的。此外，未来版本的 MCRAND 可能会使用更新版本的 MXMACA math 库，因此不同版本的 MCRAND 可能会给出略有不同的数值。

```
mcrandStatus_t MCRANDAPI mcrandGenerateLogNormalDouble ( mcrandGenerator_t
↪generator, double* outputPtr, size_t n, double  mean, double  stddev )
```

生成对数正态分布的双精度浮点数。

参数

generator

- 可用的生成器

outputPtr

- 指向设备内存的指针，用于存储 MXMACA 生成的随机数，或者指向主机内存的指针，用于存储 CPU 生成的随机数

n

- 要生成的双精度浮点数个数

mean

- 正态分布的平均值

stddev

- 正态分布的标准差

返回值

- 如果无法分配内存，则返回 MCRAND_STATUS_ALLOCATION_FAILED
- 如果生成器未创建，则返回 MCRAND_STATUS_NOT_INITIALIZED
- 如果先前的内核启动存在错误，则返回 MCRAND_STATUS_PREEXISTING_FAILURE
- 如果内核因某种原因启动失败，则返回 MCRAND_STATUS_LAUNCH_FAILURE
- 如果输出样本的数量不是准随机维数的倍数或不是伪随机生成器的 2 的倍数，则返回 MCRAND_STATUS_LENGTH_NOT_MULTIPLE
- 如果 GPU 不支持双精度浮点数，则返回 MCRAND_STATUS_DOUBLE_PRECISION_REQUIRED
- 如果成功生成，则返回 MCRAND_STATUS_SUCCESS

描述

使用 generator 将 n 个浮点数结果生成到设备内存的 outputPtr 位置。设备内存必须是预先分配的，并且足够大以容纳所有结果。启动是使用 mcrandSetStream() 设置的流完成的，如果没有设置流，则为空流。

结果是 64 位浮点值，其平均值为 mean，标准偏差为 stddev。

通常，使用伪随机数生成器和 Box-Muller 变换来生成正态分布的结果，因此要求 n 为偶数。准随机生成器使用逆累积分布函数来保持维数。生成的正态分布结果会转换为对数正态分布。

使用 mcrandCreateGenerator() 创建的生成器在 GPU 上生成的结果与使用 mcrandCreateGeneratorHost() 创建的生成器在 CPU 上计算的结果之间可能会有轻微的数值差异。这些差异是由于超越函数计算结果的差异造成的。此外，未来版本的 MCRAND 可能会使用更新版本的 MXMACA math 库，因此不同版本的 MCRAND 可能会给出略有不同的数值。

```
mcrandStatus_t MCRANDAPI mcrandGenerateLongLong ( mcrandGenerator_t
↪generator, unsigned long long* outputPtr, size_t num )
```

生成 64 位的准随机数。

参数

generator

- 可用的生成器

outputPtr

- 指向设备内存的指针，用于存储 MXMACA 生成的结果，或者指向主机内存的指针，用于存储 CPU 生成的结果

num

- 要生成的随机 64 位值的个数

返回值

- 如果生成器未创建，则返回 MCRAND_STATUS_NOT_INITIALIZED
- 如果先前的内核启动存在错误，则返回 MCRAND_STATUS_PREEXISTING_FAILURE
- 如果输出样本的数量不是准随机维数的倍数，则返回 MCRAND_STATUS_LENGTH_NOT_MULTIPLE
- 如果内核因某种原因启动失败，则返回 MCRAND_STATUS_LAUNCH_FAILURE
- 如果生成器不是 64 位的准随机生成器，则返回 MCRAND_STATUS_TYPE_ERROR
- 如果成功生成，则返回 MCRAND_STATUS_SUCCESS

描述

使用 generator 在 outputPtr 的设备内存中生成 num 个 64 位的结果。设备内存必须是预先分配的，并且足够大以容纳所有结果。启动是使用 mcrandSetStream() 设置的流完成的，如果没有设置流，则为空流。

结果是 64 位值，每个位都是随机的。

```
mcrandStatus_t MCRANDAPI mcrandGenerateNormal ( mcrandGenerator_t
↪generator, float* outputPtr, size_t n, float  mean, float  stddev )
```

生成正态分布的双精度浮点数。

参数

generator

- 可用的生成器

outputPtr

- 指向设备内存的指针，用于存储 MXMACA 生成的结果，或者指向主机内存的指针，用于存储 CPU 生成的结果

n

- 要生成的浮点数个数

mean

- 正态分布的平均值

stddev

- 正态分布的标准差

返回值

- 如果无法分配内存，则返回 MCRAND_STATUS_ALLOCATION_FAILED
- 如果生成器未创建，则返回 MCRAND_STATUS_NOT_INITIALIZED
- 如果先前的内核启动存在错误，则返回 MCRAND_STATUS_PREEXISTING_FAILURE

- 如果内核因某种原因启动失败，则返回 MCRAND_STATUS_LAUNCH_FAILURE
- 如果输出样本的数量不是准随机维数的倍数或不是伪随机生成器的 2 的倍数，则返回 MCRAND_STATUS_LENGTH_NOT_MULTIPLE
- 如果成功生成，则返回 MCRAND_STATUS_SUCCESS

描述

使用 generator 将 n 个浮点数结果生成到设备内存的 outputPtr 位置。设备内存必须是预先分配的，并且足够大以容纳所有结果。启动是使用 mcrandSetStream() 设置的流完成的，如果没有设置流，则为空流。

随机数是 32 位浮点值，其平均值为 mean，标准偏差为 stddev。

通常，使用伪随机数生成器和 Box-Muller 变换来生成正态分布的结果，因此要求 n 为偶数。准随机生成器使用逆累积分布函数来保持维数。

使用 mcrandCreateGenerator() 创建的生成器在 GPU 上生成的结果与使用 mcrandCreateGeneratorHost() 创建的生成器在 CPU 上计算的结果之间可能会有轻微的数值差异。这些差异是由于超越函数计算结果的差异造成的。此外，未来版本的 MCRAND 可能会使用更新版本的 MXMACA math 库，因此不同版本的 MCRAND 可能会给出略有不同的数值。

```
mcrandStatus_t MCRANDAPI mcrandGenerateNormalDouble ( mcrandGenerator_t
↪generator, double* outputPtr, size_t n, double  mean, double  stddev )
```

生成正态分布的双精度浮点数。

参数

generator

- 可用的生成器

outputPtr

- 指向设备内存的指针，用于存储 MXMACA 生成的结果，或者指向主机内存的指针，用于存储 CPU 生成的结果

n

- 要生成的双精度浮点数个数

mean

- 正态分布的平均值

stddev

- 正态分布的标准差

返回值

- 如果无法分配内存，则返回 MCRAND_STATUS_ALLOCATION_FAILED
- 如果生成器未创建，则返回 MCRAND_STATUS_NOT_INITIALIZED
- 如果先前的内核启动存在错误，则返回 MCRAND_STATUS_PREEXISTING_FAILURE
- 如果内核因某种原因启动失败，则返回 MCRAND_STATUS_LAUNCH_FAILURE
- 如果输出样本的数量不是准随机维数的倍数或不是伪随机生成器的 2 的倍数，则返回 MCRAND_STATUS_LENGTH_NOT_MULTIPLE
- 如果 GPU 不支持双精度浮点数，则返回 MCRAND_STATUS_DOUBLE_PRECISION_REQUIRED
- 如果成功生成，则返回 MCRAND_STATUS_SUCCESS

描述

使用 generator 将 n 个浮点数结果生成到设备内存的 outputPtr 位置。设备内存必须是预先分配的，并且足够大以容纳所有结果。启动是使用 mcrandSetStream() 设置的流完成的，如果没有设置流，则为空流。

随机数是 64 位浮点值，其平均值为 mean，标准偏差为 stddev。

通常，使用伪随机数生成器和 Box-Muller 变换来生成正态分布的结果，因此要求 n 为偶数。准随机生成器使用逆累积分布函数来保持维数。

使用 mcrandCreateGenerator() 创建的生成器在 GPU 上生成的结果与使用 mcrandCreateGeneratorHost() 创建的生成器在 CPU 上计算的结果之间可能会有轻微的数值差异。这些差异是由于超越函数计算结果的差异造成的。此外，未来版本的 MCRAND 可能会使用更新版本的 MXMACA math 库，因此不同版本的 MCRAND 可能会给出略有不同的数值。

```
mcrandStatus_t MCRANDAPI mcrandGeneratePoisson ( mcrandGenerator_t
↪generator, unsigned int* outputPtr, size_t n, double  lambda )
```

生成泊松分布的无符号整型数。

参数

generator

- 可用的生成器

outputPtr

- 指向设备内存的指针，用于存储 MXMACA 生成的结果，或者指向主机内存的指针，用于存储 CPU 生成的结果

n

- 要生成的无符号整型数个数

lambda

- 泊松分布的 lambda

返回值

- 如果无法分配内存，则返回 MCRAND_STATUS_ALLOCATION_FAILED
- 如果生成器未创建，则返回 MCRAND_STATUS_NOT_INITIALIZED
- 如果先前的内核启动存在错误，则返回 MCRAND_STATUS_PREEXISTING_FAILURE
- 如果内核因某种原因启动失败，则返回 MCRAND_STATUS_LAUNCH_FAILURE
- 如果输出样本的数量不是准随机维数的倍数，则返回 MCRAND_STATUS_LENGTH_NOT_MULTIPLE
- 如果 GPU 或 sm 不支持双精度浮点数，则返回 MCRAND_STATUS_DOUBLE_PRECISION_REQUIRED
- 如果 lambda 非正或大于 400,000，则返回 MCRAND_STATUS_OUT_OF_RANGE
- 如果成功生成，则返回 MCRAND_STATUS_SUCCESS

描述

使用 generator 在 outputPtr 的设备内存中生成 n 个无符号整型结果。设备内存必须是预先分配的，并且必须足够大以容纳所有结果。启动是使用 mcrandSetStream() 设置的流完成的，如果没有设置流，则为空流。

结果是泊松分布的 32 位无符号整型点值，使用 lambda 表达式。

```
mcrandStatus_t MCRANDAPI mcrandGenerateSeeds ( mcrandGenerator_t generator
↪)
```

设置启动状态。

参数

generator

- 要更新的生成器

返回值

- 如果无法分配内存，则返回 MCRAND_STATUS_ALLOCATION_FAILED
- 如果生成器未创建，则返回 MCRAND_STATUS_NOT_INITIALIZED
- 如果先前的内核启动存在错误，则返回 MCRAND_STATUS_PREEXISTING_FAILURE
- 如果内核因某种原因启动失败，则返回 MCRAND_STATUS_LAUNCH_FAILURE
- 如果种子成功生成，则返回 MCRAND_STATUS_SUCCESS

描述

生成生成器的启动状态。该函数由生成函数自动调用，如 mcrandGenerate() 和 mcrandGenerateUniform()。出于性能测试的目的，可以手动调用它来分离起始状态生成的时间和随机数生成的时间。

```
mcrandStatus_t MCRANDAPI mcrandGenerateUniform ( mcrandGenerator_t
↪generator, float* outputPtr, size_t num )
```

生成均匀分布的浮点数。

参数

generator

- 可用的生成器

outputPtr

- 指向设备内存的指针，用于存储 MXMACA 生成的结果，或者指向主机内存的指针，用于存储 CPU 生成的结果

num

- 要生成的浮点数个数

返回值

- 如果无法分配内存，则返回 MCRAND_STATUS_ALLOCATION_FAILED
- 如果生成器未创建，则返回 MCRAND_STATUS_NOT_INITIALIZED
- 如果先前的内核启动存在错误，则返回 MCRAND_STATUS_PREEXISTING_FAILURE
- 如果内核因某种原因启动失败，则返回 MCRAND_STATUS_LAUNCH_FAILURE
- 如果输出样本的数量不是准随机维数的倍数，则返回 MCRAND_STATUS_LENGTH_NOT_MULTIPLE
- 如果成功生成，则返回 MCRAND_STATUS_SUCCESS

描述

使用 generator 在 outputPtr 的设备内存中生成 num 个浮点数结果。设备内存必须是预先分配的，并且足够大以容纳所有结果。启动是使用 mcrandSetStream() 设置的流完成的，如果没有设置流，则为空流。

结果是介于 0.0f 和 1.0f 之间的 32 位浮点值，包括 1.0f 但不包括 0.0f。

```
mcrandStatus_t MCRANDAPI mcrandGenerateUniformDouble ( mcrandGenerator_t
↪generator, double* outputPtr, size_t num )
```

生成均匀分布的双精度浮点数。

参数

generator

- 可用的生成器

outputPtr

- 指向设备内存的指针，用于存储 MXMACA 生成的结果，或者指向主机内存的指针，用于存储 CPU 生成的结果

num

- 要生成的双精度浮点数个数

返回值

- 如果无法分配内存，则返回 MCRAND_STATUS_ALLOCATION_FAILED
- 如果生成器未创建，则返回 MCRAND_STATUS_NOT_INITIALIZED
- 如果先前的内核启动存在错误，则返回 MCRAND_STATUS_PREEXISTING_FAILURE
- 如果内核因某种原因启动失败，则返回 MCRAND_STATUS_LAUNCH_FAILURE
- 如果输出样本的数量不是准随机维数的倍数，则返回 MCRAND_STATUS_LENGTH_NOT_MULTIPLE
- 如果 GPU 不支持双精度浮点数，则返回 MCRAND_STATUS_DOUBLE_PRECISION_REQUIRED
- 如果成功生成，则返回 MCRAND_STATUS_SUCCESS

描述

使用 generator 在 outputPtr 的设备内存中生成 num 个双精度浮点数结果。设备内存必须是预先分配的，并且足够大以容纳所有结果。启动是使用 mcrandSetStream() 设置的流完成的，如果没有设置流，则为空流。

结果是介于 0.0 和 1.0 之间的 64 位双精度浮点值，包括 1.0 但不包括 0.0。

```
mcrandStatus_t MCRANDAPI mcrandGetDirectionVectors32 (
↪mcrandDirectionVectors32_t* vectors, mcrandDirectionVectorSet_t set )
```

获得用于生成 32 位准随机数的方向向量。

参数

vectors

- 返回方向向量的指针地址

set

- 一组选用的方向向量

返回值

- 如果选择的集合无效, 则返回 MCRAND_STATUS_OUT_OF_RANGE
- 如果指针设置成功，则返回 MCRAND_STATUS_SUCCESS

描述

获取指向可用于准随机数生成的方向向量数组的指针。随机数指针将引用主机内存中的方向向量数组。

该数组包含多个维度的向量。每个维度有 32 个向量。每个单独的向量都是一个无符号整型。

set 的合法值为:

- MCRAND_DIRECTION_VECTORS_32_JOEKUO6 （20000 维度）
- MCRAND_SCRAMBLED_DIRECTION_VECTORS_32_JOEKUO6 （20000 维度）

```
mcrandStatus_t MCRANDAPI mcrandGetDirectionVectors64 (
↪mcrandDirectionVectors64_t* vectors, mcrandDirectionVectorSet_t set )
```

获得用于生成 64 位准随机数的方向向量。

参数

vectors

- 返回方向向量的指针地址

set

- 一组选用的方向向量

返回值

- 如果选择的集合无效, 则返回 MCRAND_STATUS_OUT_OF_RANGE
- 如果指针设置成功，则返回 MCRAND_STATUS_SUCCESS

描述

获取指向可用于准随机数生成的方向向量数组的指针。随机数指针将引用主机内存中的方向向量数组。

该数组包含多个维度的向量。每个维度有 64 个向量。每个单独的向量都是一个无符号双长整型。

set 的合法值为:

- MCRAND_DIRECTION_VECTORS_64_JOEKUO6 （20000 维度）
- MCRAND_SCRAMBLED_DIRECTION_VECTORS_64_JOEKUO6 （20000 维度）

```
mcrandStatus_t MCRANDAPI mcrandGetProperty ( libraryPropertyType type,
↪int* value )
```

返回 mcrand 属性的值。

参数

type

- MXMACA 库属性

value

- 请求的属性的整数值

返回值

- 如果属性值已成功返回，则返回 MCRAND_STATUS_SUCCESS
- 如果无法识别属性类型，则返回 MCRAND_STATUS_OUT_OF_RANGE

描述

在 value 中返回由动态链接的 MCRAND 库的类型描述的属性数字。

```
mcrandStatus_t MCRANDAPI mcrandGetScrambleConstants32 ( unsigned int**
↪constants )
```

获取 32 位 Scrambled Sobol 序列的 Scramble 常数。

参数

constants

- 返回 Scramble 常数的指针地址

返回值

如果指针设置成功，则返回 MCRAND_STATUS_SUCCESS

描述

获取指向可用于生成准随机数的 Scramble 常数数组的指针。所得到的指针将引用位于主机内存中的无符号整型数组。

该数组包含多个维度的常量。每个维度都有一个无符号整型常量。

```
mcrandStatus_t MCRANDAPI mcrandGetScrambleConstants64 ( unsigned long
↪long** constants )
```

获得 64 位 Scrambled Sobol 序列的 Scramble 常数。

参数

constants

- 返回 Scramble 常数的指针地址

返回值

如果指针设置成功，则返回 MCRAND_STATUS_SUCCESS

描述

获取指向可用于生成准随机数的 Scramble 常数数组的指针。所得到的指针将引用位于主机内存中的无符号双长整型数组。

该数组包含多个维度的常量。每个维度都有一个无符号双长整型常量。

```
mcrandStatus_t MCRANDAPI mcrandGetVersion ( int* version )
```

返回库的版本号。

参数

version

- MCRAND 库的版本号

返回值

如果版本号成功返回，则返回 MCRAND_STATUS_SUCCESS

描述

在 *version 中返回动态链接的 mcRAND 库的版本号。格式与 MXMACA Runtime 中的 MACART_VERSION 相同。唯一支持的配置是与 MXMACA Runtime 版本相同的 MCRAND 版本。

```
mcrandStatus_t MCRANDAPI mcrandSetGeneratorOffset ( mcrandGenerator_t
↪generator, unsigned long long offset )
```

设置伪随机数或准随机数生成器的绝对偏移量。

参数

generator

- 待修改的生成器

offset

- 绝对偏移位置

返回值

- 如果生成器未创建，则返回 MCRAND_STATUS_NOT_INITIALIZED
- 如果生成器偏移量设置成功，则返回 MCRAND_STATUS_SUCCESS

描述

设置伪随机数或准随机数生成器的绝对偏移量。

所有的 offset 值都是有效的。偏移位置是绝对的，而不是相对于序列中的当前位置。

```
mcrandStatus_t MCRANDAPI mcrandSetGeneratorOrdering ( mcrandGenerator_t
↪generator, mcrandOrdering_t order )
```

设置伪随机数或准随机数生成器的随机数排序。

参数

generator

- 待修改的生成器

order

- 结果的排序

返回值

- 如果生成器未创建，则返回 MCRAND_STATUS_NOT_INITIALIZED
- 如果排序无效，则返回 MCRAND_STATUS_OUT_OF_RANGE
- 如果生成器顺序设置成功，则返回 MCRAND_STATUS_SUCCESS

描述

设置伪随机或准随机数生成器的结果排序方式。

伪随机数生成器的合法排序值为:

- MCRAND_ORDERING_PSEUDO_DEFAULT
- MCRAND_ORDERING_PSEUDO_BEST
- MCRAND_ORDERING_PSEUDO_SEEDED
- MCRAND_ORDERING_PSEUDO_LEGACY

准随机数生成器的合法排序值为:

- MCRAND_ORDERING_QUASI_DEFAULT

```
mcrandStatus_t MCRANDAPI mcrandSetPseudoRandomGeneratorSeed (
↪mcrandGenerator_t generator, unsigned long long seed )
```

设置伪随机数生成器的种子值。

参数

generator

- 待修改的生成器

seed

- 种子值

返回值

- 如果生成器未创建，则返回 MCRAND_STATUS_NOT_INITIALIZED
- 如果排序无效，则返回 MCRAND_STATUS_OUT_OF_RANGE
- 如果生成器顺序设置成功，则返回 MCRAND_STATUS_SUCCESS

描述

设置伪随机数生成器的种子值。所有的种子值都是有效的。不同的种子将产生不同的序列。不同的种子之间往往不具有统计相关性，但某些种子对的值可能生成具有统计相关性的序列。

```
mcrandStatus_t MCRANDAPI mcrandSetQuasiRandomGeneratorDimensions (
↪mcrandGenerator_t generator, unsigned int  num_dimensions )
```

设置维度数量。

参数

generator

- 待修改的生成器

num_dimensions

- 维度数量

返回值

- 如果生成器未创建，则返回 MCRAND_STATUS_NOT_INITIALIZED
- 如果 num_dimensions 无效，则返回 MCRAND_STATUS_OUT_OF_RANGE
- 如果生成器不是准随机数生成器，则返回 MCRAND_STATUS_TYPE_ERROR
- 如果生成器顺序设置成功，MCRAND_STATUS_SUCCESS

描述

设置准随机数生成器生成的维度数量。

合法的 num_dimensions 值为 1 ~ 20000。

```
mcrandStatus_t MCRANDAPI mcrandSetStream ( mcrandGenerator_t generator,
↪mcStream_t stream )
```

为 MCRAND 内核启动设置当前流。

参数

generator

- 待修改的生成器

stream

- 要使用的数据流，空数据流则为 NULL

返回值

- 如果生成器未创建，则返回 MCRAND_STATUS_NOT_INITIALIZED
- 如果流设置成功，则返回 MCRAND_STATUS_SUCCESS

描述

为 MCRAND 内核启动设置当前数据流。在重新设置之前，所有库函数都将使用该数据流。

4.2 设备 API

4.2.1 命名空间

命名空间 mcrand_detail

4.2.2 函数列表

```
__device__ unsigned int mcrand ( mcrandStateMtgp32_t* state )
```

从 mtgp32 生成器中返回 32 位的伪随机数。

```
__device__ unsigned long long mcrand ( mcrandStateScrambledSobol64_t*
↪state )
```

从 Scrambled Sobol64 生成器中返回 64 位的准随机数。

```
__device__ unsigned long long mcrand ( mcrandStateSobol64_t* state )
```

从 Sobol64 生成器中返回 64 位准随机数。

```
__device__ unsigned int mcrand ( mcrandStateScrambledSobol32_t* state )
```

从 Scrambled Sobol32 生成器中返回 32 位的准随机数。

```
__device__ unsigned int mcrand ( mcrandStateSobol32_t* state )
```

从 Sobol32 生成器中返回 32 位的准随机数。

```
__device__ unsigned int mcrand ( mcrandStateMRG32k3a_t* state )
```

从 MRG32k3a 生成器返回 32 位的伪随机数。

```
__device__ unsigned int mcrand ( mcrandStatePhilox4_32_10_t* state )
```

从 Philox4_32_10 生成器返回 32 位的伪随机数。

```
__device__ unsigned int mcrand ( mcrandStateXORWOW_t* state )
```

从 XORWOW 生成器返回 32 位的伪随机数。

```
__device__ uint4 mcrand4 ( mcrandStatePhilox4_32_10_t* state )
```

从 Philox4_32_10 生成器返回一个包含四个 32 位伪随机数的元组。

```
__host__ __forceinline__ mcrandStatus_t mcrandMakeMTGP32Constants ( const
↪mtgp32_params_fast_t params[], mtgp32_kernel_params_t* p )
```

设置 mtgp32 生成器的常数参数

```
__host__ __forceinline__ mcrandStatus_t MCRANDAPI
↪mcrandMakeMTGP32KernelState ( mcrandStateMtgp32_t* s, mtgp32_params_fast_
↪t params[], mtgp32_kernel_params_t* k, int  n, unsigned long long seed )
```

设置 mtgp32 生成器的初始状态

```
__device__ void mcrand_init ( mcrandDirectionVectors64_t direction_vectors,
↪ unsigned long long scramble_c, unsigned long long offset,
↪mcrandStateScrambledSobol64_t* state )
```

初始化 Scrambled Sobol64 生成器的状态

```
__device__ void mcrand_init ( mcrandDirectionVectors64_t direction_vectors,
↪ unsigned long long offset, mcrandStateSobol64_t* state )
```

初始化 Sobol64 生成器的状态

```
__device__ void mcrand_init ( mcrandDirectionVectors32_t direction_vectors,
↪ unsigned int  scramble_c, unsigned int  offset,
↪mcrandStateScrambledSobol32_t* state )
```

初始化 Scrambled Sobol32 生成器的状态

```
__device__ void mcrand_init ( mcrandDirectionVectors32_t direction_vectors,
↪ unsigned int  offset, mcrandStateSobol32_t* state )
```

初始化 Sobol32 生成器的状态

```
__device__ void mcrand_init ( unsigned long long seed, unsigned long long
↪subsequence, unsigned long long offset, mcrandStateMRG32k3a_t* state )
```

初始化 MRG32k3a 生成器的状态

```
__device__ void mcrand_init ( unsigned long long seed, unsigned long long
↪subsequence, unsigned long long offset, mcrandStatePhilox4_32_10_t*
↪state )
```

初始化 Philox4_32_10 生成器的状态

```
__device__ void mcrand_init ( unsigned long long seed, unsigned long long
↪subsequence, unsigned long long offset, mcrandStateXORWOW_t* state )
```

初始化 XORWOW 生成器的状态

```
__device__ float mcrand_log_normal ( mcrandStateScrambledSobol64_t* state,
↪float  mean, float  stddev )
```

从 Scrambled Sobol64 生成器中返回一个对数正态分布的浮点数

```
__device__ float mcrand_log_normal ( mcrandStateSobol64_t* state, float
↪mean, float  stddev )
```

从 Sobol64 生成器返回一个对数正态分布的浮点数。

```
__device__ float mcrand_log_normal ( mcrandStateScrambledSobol32_t* state,
↪float  mean, float  stddev )
```

从 Scrambled Sobol32 生成器返回一个对数正态分布的浮点数。

```
__device__ float mcrand_log_normal ( mcrandStateSobol32_t* state, float
↪mean, float  stddev )
```

从 Sobol32 生成器返回一个对数正态分布的浮点数。

```
__device__ float mcrand_log_normal ( mcrandStateMtgp32_t* state, float
↪mean, float  stddev )
```

从 MTGP32 生成器中返回一个对数正态分布的浮点数

```
__device__ float mcrand_log_normal ( mcrandStateMRG32k3a_t* state, float
↪mean, float  stddev )
```

从 MRG32k3a 生成器中返回一个对数正态分布的浮点数

```
__device__ float mcrand_log_normal ( mcrandStatePhilox4_32_10_t* state,
↪float  mean, float  stddev )
```

从 Philox4_32_10 生成器中返回一个对数正态分布的浮点数

```
__device__ float mcrand_log_normal ( mcrandStateXORWOW_t* state, float
↪mean, float  stddev )
```

从 XORWOW 生成器返回一个对数正态分布的浮点数。

```
__device__ float2 mcrand_log_normal2 ( mcrandStateMRG32k3a_t* state, float
↪ mean, float  stddev )
```

从 MRG32k3a 生成器返回两个正态分布的浮点数。

```
__device__ float2 mcrand_log_normal2 ( mcrandStatePhilox4_32_10_t* state,
↪float  mean, float  stddev )
```

从 Philox4_32_10 生成器中返回两个正态分布的浮点数

```
__device__ float2 mcrand_log_normal2 ( mcrandStateXORWOW_t* state, float
↪mean, float  stddev )
```

从 XORWOW 生成器中返回两个正态分布的浮点数

```
__device__ double2 mcrand_log_normal2_double ( mcrandStateMRG32k3a_t*
↪state, double  mean, double  stddev )
```

从 MRG32k3a 生成器返回两个对数正态分布的双精度浮点数。

```
__device__ double2 mcrand_log_normal2_double ( mcrandStatePhilox4_32_10_t*
↪state, double  mean, double  stddev )
```

从 Philox4_32_10 生成器返回两个对数正态分布的双精度浮点数。

```
__device__ double2 mcrand_log_normal2_double ( mcrandStateXORWOW_t* state,
↪double  mean, double  stddev )
```

从 XORWOW 生成器返回两个对数正态分布的双精度浮点数。

```
__device__ float4 mcrand_log_normal4 ( mcrandStatePhilox4_32_10_t* state,
↪float  mean, float  stddev )
```

从 Philox4_32_10 生成器返回四个正态分布的浮点数

```
__device__ double mcrand_log_normal_double ( mcrandStateScrambledSobol64_
↪t* state, double  mean, double  stddev )
```

从 Scrambled Sobol64 生成器返回一个对数正态分布的双精度浮点数

```
__device__ double mcrand_log_normal_double ( mcrandStateSobol64_t* state,
↪double  mean, double  stddev )
```

从 Sobol64 生成器返回一个对数正态分布的双精度浮点数

```
__device__ double mcrand_log_normal_double ( mcrandStateScrambledSobol32_
↪t* state, double  mean, double  stddev )
```

从 Scrambled Sobol32 生成器返回一个对数正态分布的双精度浮点数。

```
__device__ double mcrand_log_normal_double ( mcrandStateSobol32_t* state,
↪double  mean, double  stddev )
```

从 Sobol32 生成器返回一个对数正态分布的双精度浮点数。

```
__device__ double mcrand_log_normal_double ( mcrandStateMtgp32_t* state,
↪double  mean, double  stddev )
```

从 MTGP32 生成器返回一个对数正态分布的双精度浮点数。

```
__device__ double mcrand_log_normal_double ( mcrandStateMRG32k3a_t* state,
↪double  mean, double  stddev )
```

从 MRG32k3a 生成器返回一个对数正态分布的双精度浮点数。

```
__device__ double mcrand_log_normal_double ( mcrandStatePhilox4_32_10_t*
↪state, double  mean, double  stddev )
```

从 Philox4_32_10 生成器返回一个对数正态分布的双精度浮点数。

```
__device__ double mcrand_log_normal_double ( mcrandStateXORWOW_t* state,
↪double  mean, double  stddev )
```

从 XORWOW 生成器返回一个对数正态分布的双精度浮点数。

```
__device__ float mcrand_mtgp32_single ( mcrandStateMtgp32_t* state )
```

从 mtgp32 生成器返回一个均匀分布的浮点数。

```
__device__ float mcrand_mtgp32_single_specific ( mcrandStateMtgp32_t*
↪state, unsigned char  index, unsigned char  n )
```

从 mtgp32 生成器的特定位置返回一个均匀分布的浮点数。

```
__device__ unsigned int mcrand_mtgp32_specific ( mcrandStateMtgp32_t*
↪state, unsigned char  index, unsigned char  n )
```

从 mtgp32 生成器的特定位置返回一个 32 位伪随机数。

```
__device__ float mcrand_normal ( mcrandStateScrambledSobol64_t* state )
```

从 Scrambled Sobol64 生成器返回一个正态分布的浮点数。

```
__device__ float mcrand_normal ( mcrandStateSobol64_t* state )
```

从 Sobol64 生成器返回一个正态分布的浮点数。

```
__device__ float mcrand_normal ( mcrandStateScrambledSobol32_t* state )
```

从 Scrambled Sobol32 生成器返回一个正态分布的浮点数。

```
__device__ float mcrand_normal ( mcrandStateSobol32_t* state )
```

从 Sobol32 生成器返回一个正态分布的浮点数。

```
__device__ float mcrand_normal ( mcrandStateMtgp32_t* state )
```

从 MTGP32 生成器返回一个正态分布的浮点数。

```
__device__ float mcrand_normal ( mcrandStateMRG32k3a_t* state )
```

从 MRG32k3a 生成器返回一个正态分布的浮点数。

```
__device__ float mcrand_normal ( mcrandStatePhilox4_32_10_t* state )
```

从 Philox4_32_10 生成器返回一个正态分布浮点数。

```
__device__ float mcrand_normal ( mcrandStateXORWOW_t* state )
```

从 XORWOW 生成器返回一个正态分布的浮点数。

```
__device__ float2 mcrand_normal2 ( mcrandStateMRG32k3a_t* state )
```

从 MRG32k3a 生成器返回两个正态分布的浮点数。

```
__device__ float2 mcrand_normal2 ( mcrandStatePhilox4_32_10_t* state )
```

从 Philox4_32_10 生成器返回两个正态分布的浮点数。

```
__device__ float2 mcrand_normal2 ( mcrandStateXORWOW_t* state )
```

从 XORWOW 生成器返回两个正态分布的浮点数。

```
__device__ double2 mcrand_normal2_double ( mcrandStateMRG32k3a_t* state )
```

从 MRG32k3a 生成器返回两个正态分布的双精度浮点数。

```
__device__ double2 mcrand_normal2_double ( mcrandStatePhilox4_32_10_t*
↪state )
```

从 Philox4_32_10 生成器返回两个正态分布的双精度浮点数。

```
__device__ double2 mcrand_normal2_double ( mcrandStateXORWOW_t* state )
```

从 XORWOW 生成器中返回两个正态分布的双精度浮点数。

```
__device__ float4 mcrand_normal4 ( mcrandStatePhilox4_32_10_t* state )
```

从 Philox4_32_10 生成器返回 4 个正态分布的浮点数。

```
__device__ double mcrand_normal_double ( mcrandStateScrambledSobol64_t*
↪state )
```

从 Scrambled Sobol64 生成器返回一个正态分布的双精度浮点数。

```
__device__ double mcrand_normal_double ( mcrandStateSobol64_t* state )
```

从 Sobol64 生成器返回一个正态分布的双精度浮点数。

```
__device__ double mcrand_normal_double ( mcrandStateScrambledSobol32_t*
↪state )
```

从 Scrambled Sobol32 生成器返回一个正态分布的双精度浮点数。

```
__device__ double mcrand_normal_double ( mcrandStateSobol32_t* state )
```

从 Sobol32 生成器返回一个正态分布的双精度浮点数。

```
__device__ double mcrand_normal_double ( mcrandStateMtgp32_t* state )
```

从 MTGP32 生成器中返回一个正态分布的双精度浮点数。

```
__device__ double mcrand_normal_double ( mcrandStateMRG32k3a_t* state )
```

从 MRG32k3a 生成器中返回一个正态分布的双精度浮点数。

```
__device__ double mcrand_normal_double ( mcrandStatePhilox4_32_10_t* state
↪)
```

从 Philox4_32_10 生成器中返回一个正态分布的双精度浮点数。

```
__device__ double mcrand_normal_double ( mcrandStateXORWOW_t* state )
```

从 XORWOW 生成器中返回一个正态分布的双精度浮点数。

```
__device__ unsigned int mcrand_poisson ( mcrandStateScrambledSobol64_t*
↪state, double  lambda )
```

从 Scrambled Sobol64 生成器中返回一个泊松分布的无符号整数。

```
__device__ unsigned int mcrand_poisson ( mcrandStateSobol64_t* state,
↪double  lambda )
```

从 Sobol64 生成器中返回一个泊松分布的无符号整数。

```
__device__ unsigned int mcrand_poisson ( mcrandStateScrambledSobol32_t*
↪state, double  lambda )
```

从 Scrambled Sobol32 生成器中返回一个泊松分布的无符号整数。

```
__device__ unsigned int mcrand_poisson ( mcrandStateSobol32_t* state,
↪double  lambda )
```

从 Sobol32 生成器中返回一个泊松分布的无符号整数。

```
__device__ unsigned int mcrand_poisson ( mcrandStateMtgp32_t* state,
↪double  lambda )
```

从 MTGP32 生成器中返回一个泊松分布的无符号整数。

```
__device__ unsigned int mcrand_poisson ( mcrandStateMRG32k3a_t* state,
↪double  lambda )
```

从 MRG32k3A 生成器中返回一个泊松分布的无符号整数。

```
__device__ unsigned int mcrand_poisson ( mcrandStatePhilox4_32_10_t* state,
↪ double  lambda )
```

从 Philox4_32_10 生成器中返回一个泊松分布的无符号整数。

```
__device__ unsigned int mcrand_poisson ( mcrandStateXORWOW_t* state,
↪double  lambda )
```

从 XORWOW 生成器中返回一个泊松分布的无符号整数。

```
__device__ uint4 mcrand_poisson4 ( mcrandStatePhilox4_32_10_t* state,
↪double  lambda )
```

从 Philox4_32_10 生成器中返回四个泊松分布的无符号整数。

```
__device__ float mcrand_uniform ( mcrandStateScrambledSobol64_t* state )
```

从 Scrambled Sobol64 生成器中返回一个均匀分布的浮点数。

```
__device__ float mcrand_uniform ( mcrandStateSobol64_t* state )
```

从 Sobol64 生成器中返回一个均匀分布的浮点数。

```
__device__ float mcrand_uniform ( mcrandStateScrambledSobol32_t* state )
```

从 Scrambled Sobol32 生成器中返回一个均匀分布的浮点数。

```
__device__ float mcrand_uniform ( mcrandStateSobol32_t* state )
```

从 Sobol32 生成器中返回一个均匀分布的浮点数。

```
__device__ float mcrand_uniform ( mcrandStateMtgp32_t* state )
```

从 MTGP32 生成器中返回一个均匀分布的浮点数。

```
__device__ float mcrand_uniform ( mcrandStatePhilox4_32_10_t* state )
```

从 Philox4_32_10 生成器中返回一个均匀分布的浮点数。

```
__device__ float mcrand_uniform ( mcrandStateMRG32k3a_t* state )
```

从 MRG32k3a 生成器中返回一个均匀分布的浮点数。

```
__device__ float mcrand_uniform ( mcrandStateXORWOW_t* state )
```

从 XORWOW 生成器中返回一个均匀分布的浮点数。

```
__device__ double2 mcrand_uniform2_double ( mcrandStatePhilox4_32_10_t*
↪state )
```

从 Philox4_32_10 生成器中返回一个均匀分布的元组，其中包含两个双精度浮点数。

```
__device__ float4 mcrand_uniform4 ( mcrandStatePhilox4_32_10_t* state )
```

从 Philox4_32_10 生成器中返回一个均匀分布的元组，包含四个浮点数。

```
__device__ double mcrand_uniform_double ( mcrandStateScrambledSobol64_t*
↪state )
```

从 Scrambled Sobol64 生成器中返回一个均匀分布的双精度浮点数。

```
__device__ double mcrand_uniform_double ( mcrandStateSobol64_t* state )
```

从 Sobol64 生成器中返回一个均匀分布的双精度浮点数。

```
__device__ double mcrand_uniform_double ( mcrandStateScrambledSobol32_t*
↪state )
```

从 Scrambled Sobol32 生成器中返回一个均匀分布的双精度浮点数。

```
__device__ double mcrand_uniform_double ( mcrandStateSobol32_t* state )
```

从 Sobol32 生成器中返回一个均匀分布的双精度浮点数。

```
__device__ double mcrand_uniform_double ( mcrandStatePhilox4_32_10_t*
↪state )
```

从 Philox4_32_10 生成器中返回一个均匀分布的双精度浮点数。

```
__device__ double mcrand_uniform_double ( mcrandStateMtgp32_t* state )
```

从 MTGP32 生成器中返回一个均匀分布的双精度浮点数。

```
__device__ double mcrand_uniform_double ( mcrandStateMRG32k3a_t* state )
```

从 MRG32k3a 生成器中返回一个均匀分布的双精度浮点数。

```
__device__ double mcrand_uniform_double ( mcrandStateXORWOW_t* state )
```

从 XORWOW 生成器中返回一个均匀分布的双精度浮点数。

```
template < typename T >
__device__ MCRAND_STD::enable_if < MCRAND_STD::is_same &lt;
↪mcrandStateSobol64_t* > ::type skipahead ( unsigned long long n, T state
↪)
```

更新 Sobol64 生成器的状态以跳过 n 个元素。

```
template < typename T >
__device__ MCRAND_STD::enable_if < MCRAND_STD::is_same &lt;
↪mcrandStateSobol32_t* > ::type skipahead ( unsigned int  n, T state )
```

更新 Sobol32 生成器的状态以跳过 n 个元素。

```
__device__ void skipahead ( unsigned long long n, mcrandStateMRG32k3a_t*
↪state )
```

更新 MRG32k3a 生成器的状态以跳过 n 个元素。

```
__device__ void skipahead ( unsigned long long n, mcrandStatePhilox4_32_10_
↪t* state )
```

更新 Philox4_32_10 生成器的状态以跳过 n 个元素。

```
__device__ void skipahead ( unsigned long long n, mcrandStateXORWOW_t*
↪state )
```

更新 XORWOW 生成器的状态以跳过 n 个元素。

```
__device__ void skipahead_sequence ( unsigned long long n,
↪mcrandStateMRG32k3a_t* state )
```

更新 MRG32k3a 生成器的状态以跳过 n 个序列。

```
__device__ void skipahead_sequence ( unsigned long long n,
↪mcrandStatePhilox4_32_10_t* state )
```

更新 Philox4_32_10 生成器的状态以跳过 n 个序列。

```
__device__ void skipahead_sequence ( unsigned long long n,
↪mcrandStateXORWOW_t* state )
```

更新 XORWOW 生成器的状态以跳过 n 个序列。

```
__device__ void skipahead_subsequence ( unsigned long long n,
↪mcrandStateMRG32k3a_t* state )
```

更新 MRG32k3a 生成器的状态以跳过 n 个序列。

4.2.3 函数

```
__device__ unsigned int mcrand ( mcrandStateMtgp32_t* state )
```

从 mtgp32 生成器返回 32 位伪随机数。

参数

state

- 指向要更新状态的指针。

返回值

32 位伪随机数作为无符号整数返回，所有位均有效。

描述

从 state 状态的 MTGP32 生成器中返回 32 位伪随机数，生成器的位置按块中的线程数递增。注意，块中的线程数不能超过 256。

```
__device__ unsigned long long mcrand ( mcrandStateScrambledSobol64_t*
↪state )
```

从 Sobol64 生成器返回 64 位伪随机数。

参数

state

- 指向要更新状态的指针

返回值

64 位准随机数作为无符号双长整型返回，所有位均有效。

描述

从 state 状态的 Scrambled Sobol32 生成器中返回 64 位准随机数，生成器的位置递增一位。

```
__device__ unsigned long long mcrand ( mcrandStateSobol64_t* state )
```

从 Sobol64 生成器返回 64 位伪随机数。

参数

state

- 指向要更新状态的指针

返回值

64 位准随机数作为无符号长整型返回，所有位均有效。

描述

从 state 状态的 Sobol64 生成器中返回 64 位准随机数，生成器的位置递增一位。

```
__device__ unsigned int mcrand ( mcrandStateScrambledSobol32_t* state )
```

从 Scrambled Sobol32 生成器返回 32 位伪随机数。

参数

state

- 指向要更新状态的指针

返回值

32 位准随机数作为无符号整数返回，所有位均有效。

描述

从 state 状态的 Scrambled Sobol32 生成器中返回 32 位准随机数，生成器的位置递增一位。

```
__device__ unsigned int mcrand ( mcrandStateSobol32_t* state )
```

从 Sobol32 生成器返回 32 位伪随机数。

参数

state

- 指向要更新状态的指针

返回值

32 位准随机数作为无符号整数返回，所有位均有效。

描述

从 state 状态的 Sobol32 生成器中返回 32 位准随机数，生成器的位置递增一位。

```
__device__ unsigned int mcrand ( mcrandStateMRG32k3a_t* state )
```

从 MRG32k3a 生成器返回 32 位伪随机数。

参数

state

- 指向要更新状态的指针

返回值

32 位伪随机数作为无符号整数返回，所有位均有效。

描述

从 state 状态的 MRG32k3a 生成器中返回 32 位伪随机数，生成器的位置递增一位。

```
__device__ unsigned int mcrand ( mcrandStatePhilox4_32_10_t* state )
```

从 Philox4_32_10 生成器返回 32 位伪随机数。

参数

state

- 指向要更新状态的指针

返回值

32 位伪随机数作为无符号整数返回，所有位均有效。

描述

从 state 状态的 Philox4_32_10 生成器中返回 32 位伪随机数，生成器的位置递增一位。

```
__device__ unsigned int mcrand ( mcrandStateXORWOW_t* state )
```

从 XORWOW 生成器返回 32 位伪随机数。

参数

state

- 指向要更新状态的指针

返回值

32 位伪随机数作为无符号整数返回，所有位均有效。

描述

从 state 状态的 XORWOW 生成器返回 32 位伪随机数，生成器的位置递增一位。

```
__device__ uint4 mcrand4 ( mcrandStatePhilox4_32_10_t* state )
```

从 Philox4_32_10 生成器返回 4 个 32 位伪随机数的元组。

参数

state

- 指向要更新状态的指针

返回值

128 位伪随机数作为 uint4 返回，所有位均有效。

描述

从 state 状态的 Philox4_32_10 生成器中返回 128 位伪随机数，生成器的位置递增四位。

```
__host__ __forceinline__ mcrandStatus_t mcrandMakeMTGP32Constants ( const
↪mtgp32_params_fast_t params[], mtgp32_kernel_params_t* p )
```

为 MTGP32 生成器设置常量参数。

参数

params

- 指向主机内存中 mtgp32_params_fast_t 类型的数组的指针。

p

- 指向设备内存中为 mtgp32_kernel_params_t 的结构体的指针。

返回值

- 如果主机内存无法分配，则返回 MCRAND_STATUS_ALLOCATION_FAILED;
- 如果复制到设备内存时失败，则返回 MCRAND_STATUS_INITIALIZATION_FAILED;
- 否则返回 MCRAND_STATUS_SUCCESS。

描述

这个主机端辅助函数重新组织了 MCRAND_NUM_MTGP32_PARAMS 组生成器参数，供内核函数使用，并将结果复制到设备内存中的指定位置。

```
__host__ __forceinline__ mcrandStatus_t MCRANDAPI
↪mcrandMakeMTGP32KernelState ( mcrandStateMtgp32_t* s, mtgp32_params_fast_
↪t params[], mtgp32_kernel_params_t* k, int  n, unsigned long long seed )
```

为 mtgp32 生成器设置初始状态。

参数

s

- 指向设备内存中状态数组的指针

params

- 指向主机内存中 mtgp32_params_fast_t 类型的数组的指针

k

- 指向设备内存中 mtgp32_kernel_params_t 类型的结构体的指针

n

- 要初始化的参数集/状态的数量。

seed

- 种子值

返回值

- 如果主机内存状态无法分配，则返回 MCRAND_STATUS_ALLOCATION_FAILED;
- 如果复制到设备内存时失败，则返回 MCRAND_STATUS_INITIALIZATION_FAILED;
- 否则返回 MCRAND_STATUS_SUCCESS。

描述

这个主机端的辅助函数用于初始化 MTGP32 生成器的多个状态 (每个状态对应一个参数集)。为了实现这一点，它在主机内存中分配了一个状态数组，然后初始化该数组，并将结果复制到设备内存。

```
__device__ void mcrand_init ( mcrandDirectionVectors64_t direction_vectors,
↪ unsigned long long scramble_c, unsigned long long offset,
↪mcrandStateScrambledSobol64_t* state )
```

初始化 Scrambled Sobol64 生成器的状态。

参数

direction_vectors

- 指向 64 个无符号双长整型数组的指针，这些长度表示所需维度的方向向量

scramble_c

- 用于 Scrambled Sobol64 序列的常量

offset

- 按顺序绝对偏移量

state

- 要初始化的状态指针

描述

使用给定的方向向量和偏移量初始化处于 state 中的 Sobol64 状态。

方向向量是一个指向 64 个无符号双长整型数组的设备指针。任意输入的偏移量值都是合法的。

```
__device__ void mcrand_init ( mcrandDirectionVectors64_t direction_vectors,
↪ unsigned long long offset, mcrandStateSobol64_t* state )
```

初始化 Sobol64 状态。

参数

direction_vectors

- 指向 64 个无符号双长整型的数组的指针。这些长度表示所需维度的方向向量。

offset

- 按顺序绝对偏移量

state

- 要初始化的状态指针

描述

使用给定的方向向量和偏移量初始化处于 state 中的 Sobol64 状态。

方向向量是一个指向 64 个无符号双长整型数组的设备指针。任意输入的偏移量值都是合法的。

```
__device__ void mcrand_init ( mcrandDirectionVectors32_t direction_vectors,
↪ unsigned int  scramble_c, unsigned int  offset,
↪mcrandStateScrambledSobol32_t* state )
```

初始化 Scrambled Sobol32 生成器的状态

参数

direction_vectors

- 指向表示所需维度的方向向量的 32 个无符号整型数组的指针

scramble_c

- 用于 Scrambled Sobol32 序列的常量

offset

- 按顺序绝对偏移量

state

- 要初始化的状态指针

描述

使用给定的方向向量和偏移量初始化处于 state 中的 Sobol32 状态。

方向向量是一个指向 64 个无符号双长整型数组的设备指针。任意输入的偏移量值都是合法的。

```
__device__ void mcrand_init ( mcrandDirectionVectors32_t direction_vectors,
↪ unsigned int  offset, mcrandStateSobol32_t* state )
```

初始化 Sobol32 状态。

参数

direction_vectors

- 指向表示所需维度的方向向量的 32 个无符号整数的数组的指针

offset

- 按顺序绝对偏移量

state

- 要初始化的状态指针

描述

使用给定的方向向量和偏移量初始化处于 state 中的 Sobol32 状态。

方向向量是一个指向 64 个无符号双长整型数组的设备指针。任意输入的偏移量值都是合法的。

```
__device__ void mcrand_init ( unsigned long long seed, unsigned long long
↪subsequence, unsigned long long offset, mcrandStateMRG32k3a_t* state )
```

初始化 MRG32k3a 状态。

参数

seed

- 用作种子的任意位

subsequence

- 开始的子序列

offset

- 按顺序绝对偏移量

state

- 要初始化的状态指针

描述

使用给定的种子、子序列和偏移量初始化处于 state 中的 MRG32k3a 的状态。

种子、子序列和偏移量的所有输入值都是合法的。将子序列截断为 51 位以避免进入下一个序列。

种子值为 0 时，则将状态设置为 MRG32k3a 算法原始发布版本的值。

```
__device__ void mcrand_init ( unsigned long long seed, unsigned long long
↪subsequence, unsigned long long offset, mcrandStatePhilox4_32_10_t*
↪state )
```

初始化 Philox4_32_10 状态。

参数

seed

- 用作种子的任意位

subsequence

- 开始的子序列

offset

- 进入子序列的绝对偏移量

state

- 要初始化的状态指针

描述

使用给定的种子、p 子序列和偏移量初始化处于 state 中的 Philox4_32_10 状态。

种子、子序列和偏移量的所有输入值都是合法的。种子的 264 个可能值中的每一个都选择一个长度为 2130 的独立序列。序列的前 266 * subsequence + offset. 个值被跳过。也就是说，子序列的长度为 266。

```
__device__ void mcrand_init ( unsigned long long seed, unsigned long long
↪subsequence, unsigned long long offset, mcrandStateXORWOW_t* state )
```

初始化 XORWOW 状态。

参数

seed

- 用作种子的任意位

subsequence

- 开始的子序列

offset

- 按顺序绝对偏移量

state

- 要初始化的状态指针

描述

使用给定的种子、子序列和偏移量初始化处于 state 中的 XORWOW 状态。

输入任意种子、子序列以及偏移量的值都是合法的。子序列和偏移值越大，需要的计算量就越大，因此完成时间也就越长。

如果种子值为 0，则将状态设置为原始发布的 xorwow 算法的值。

```
__device__ float mcrand_log_normal ( mcrandStateScrambledSobol64_t* state,
↪float  mean, float  stddev )
```

从 Scrambled Sobol64 生成器返回一个对数正态分布的浮点数。

参数

state

- 要更新的状态指针

mean

- 相关正态分布的均值

stddev

- 相关正态分布的标准差

返回值

均值为 mean，标准差为 stddev 的对数正态分布浮点数

描述

从 state 状态的 Scrambled Sobol64 生成器中返回一个从正态分布派生，均值为 mean，标准差为 stddev 的对数正态分布浮点数，生成器的位置递增一位。

实现使用分位函数生成正态分布的结果，然后将结果转换为对数正态分布。

```
__device__ float mcrand_log_normal ( mcrandStateSobol64_t* state, float
↪mean, float  stddev )
```

从 Sobol64 生成器返回一个对数正态分布的浮点数。

参数

state

- 要更新的状态指针

mean

- 相关正态分布的均值

stddev

- 相关正态分布的标准差

返回值

均值为 mean，标准差为 stddev 的对数正态分布浮点数

描述

从 state 状态的 Sobol64 生成器中返回一个从正态分布派生，均值为 mean，标准差为 stddev 的对数正态分布浮点数，生成器的位置递增一位。

实现使用分位函数生成正态分布的结果，然后将结果转换为对数正态分布。

```
__device__ float mcrand_log_normal ( mcrandStateScrambledSobol32_t* state,
↪float  mean, float  stddev )
```

从 Scrambled Sobol32 生成器返回一个对数正态分布的浮点数。

参数

state

- 要更新的状态指针

mean

- 相关正态分布的均值

stddev

- 相关正态分布的标准差

返回值

均值为 mean，标准差为 stddev 的对数正态分布浮点数

描述

从 state 状态的 Scrambled Sobol32 生成器返回一个从正态分布派生，均值为 mean，标准差为 stddev 的对数正态分布浮点数，生成器的位置递增一位。

该实现使用分位函数生成一个正态分布的结果，然后将结果转换为对数正态分布。

```
__device__ float mcrand_log_normal ( mcrandStateSobol32_t* state, float
↪mean, float  stddev )
```

(下页继续)

(续上页)

从 Sobol32 生成器返回一个对数正态分布的浮点数。

参数

state

- 要更新的状态指针

mean

- 相关正态分布的均值

stddev

- 相关正态分布的标准差

返回值

均值为 mean，标准差为 stddev 的对数正态分布浮点数

描述

从 state 状态的 Sobol32 生成器返回一个从正态分布派生，均值为 mean，标准差为 stddev 的对数正态分布浮点数，生成器的位置递增一位。

该实现使用分位函数生成一个正态分布的结果，然后将结果转换为对数正态分布。

```
__device__ float mcrand_log_normal ( mcrandStateMtgp32_t* state, float
↪mean, float  stddev )
```

从 MTGP32 生成器返回一个对数正态分布的浮点数。

参数

state

- 要更新的状态指针

mean

- 相关正态分布的均值

stddev

- 相关正态分布的标准差

返回值

均值为 mean，标准差为 stddev 的对数正态分布浮点数

描述

从 state 状态的 MTGP32 生成器返回一个从正态分布派生，均值为 mean，标准差为 stddev 的对数正态分布浮点数，生成器的位置递增一位。

该实现使用分位函数生成一个正态分布的结果，然后将结果转换为对数正态分布。

```
__device__ float mcrand_log_normal ( mcrandStateMRG32k3a_t* state, float
↪mean, float  stddev )
```

从 MRG32k3a 生成器返回一个对数正态分布的浮点数。

参数

state

- 要更新的状态指针

mean

- 相关正态分布的均值

stddev

- 相关正态分布的标准差

返回值

均值为 mean，标准差为 stddev 的对数正态分布浮点数

描述

从 state 状态的 MRG32k3a 生成器返回一个从正态分布派生，均值为 mean，标准差为 stddev 的对数正态分布浮点数，生成器的位置递增一位。

实现过程中使用了 Box-Muller 变换生成的两个正态分布结果，将它们转换为对数正态分布，然后一次返回一个。有关一次返回两个结果的更高效的版本，请参见 mcrand_log_normal2()。

```
__device__ float mcrand_log_normal ( mcrandStatePhilox4_32_10_t* state,
↪float  mean, float  stddev )
```

从 Philox4_32_1 生成器返回一个对数正态分布的浮点数。

参数

state

- 要更新的状态指针

mean

- 相关正态分布的均值

stddev

- 相关正态分布的标准差

返回值

均值为 mean，标准差为 stddev 的对数正态分布浮点数

描述

从 state 状态的 Philox4_32_1 生成器中返回一个从正态分布派生，均值为 mean，标准差为 stddev 的对数正态分布浮点数，生成器的位置递增一位。

实现过程使用了 Box-Muller 变换生成的两个正态分布结果，将它们转换为对数正态分布，然后一次返回一个。有关一次返回两个结果的更高效的版本，请参见 mcrand_log_normal2()。

```
__device__ float mcrand_log_normal ( mcrandStateXORWOW_t* state, float
↪mean, float  stddev )
```

从 XORWOW 生成器返回一个对数正态分布的浮点数。

参数

state

- 要更新的状态指针

mean

- 相关正态分布的均值

stddev

- 相关正态分布的标准差

返回值

均值为 mean，标准差为 stddev 的对数正态分布浮点数

描述

从 state 状态的 XORWOW 生成器中返回一个从正态分布派生，均值为 mean，标准差为 stddev 的对数正态分布浮点数，生成器的位置递增一位。

该实现使用 Box-Muller 变换生成两个符合正态分布的结果, 将它们转换为对数正态分布, 并逐个返回。如果希望以更高效的方式同时返回两个结果，请参见 mcrand_log_normal2() 函数。

```
__device__ float2 mcrand_log_normal2 ( mcrandStateMRG32k3a_t* state, float
↪ mean, float  stddev )
```

从 MRG32k3a 生成器返回两个正态分布的浮点数。

参数

state

- 指向要更新状态的指针

mean

- 相关正态分布的均值

stddev

- 相关正态分布的标准差

返回值

由均值为 mean，标准差为 stddev 的正态分布派生的对数正态分布 float2

描述

从 state 状态的 MRG32k3a 生成器中返回两个由均值为 mean，标准差为 stddev 的正态分布派生的对数正态分布浮点数，生成器的位置递增两位。

该实现使用 Box-Muller 变换生成两个符合正态分布的结果, 并将其转换为对数正态分布。

```
__device__ float2 mcrand_log_normal2 ( mcrandStatePhilox4_32_10_t* state,
↪float  mean, float  stddev )
```

从 Philox4_32_10 生成器返回两个正态分布的浮点数

参数

state

- 指向要更新状态的指针

mean

- 相关正态分布的均值

stddev

- 相关正态分布的标准差

返回值

由均值为 mean，标准差为 stddev 的正态分布派生的对数正态分布 float2

描述

从 state 状态的 Philox4_32_10 生成器中返回两个由均值为 mean，标准差为 stddev 的正态分布派生的对数正态分布浮点数，生成器的位置递增两位。

该实现使用 Box-Muller 变换生成两个符合正态分布的结果, 并将其转换为对数正态分布。

```
__device__ float2 mcrand_log_normal2 ( mcrandStateXORWOW_t* state, float
↪mean, float  stddev )
```

从 XORWOW 生成器返回两个正态分布的浮点数

参数

state

- 指向要更新状态的指针

mean

- 相关正态分布的均值

stddev

- 相关正态分布的标准差

返回值

由均值为 mean，标准差为 stddev 的正态分布派生的对数正态分布

描述

从 state 状态的 XORWOW 生成器中返回两个由均值为 mean，标准差为 stddev 的正态分布派生的对数正态分布浮点数，生成器的位置递增两位。

该实现使用 Box-Muller 变换生成两个符合正态分布的结果, 并将其转换为对数正态分布。

```
__device__ double2 mcrand_log_normal2_double ( mcrandStateMRG32k3a_t*
↪state, double  mean, double  stddev )
```

从 MRG32k3a 生成器返回两个对数正态分布的双精度浮点数

参数

state

- 指向要更新状态的指针

mean

- 相关正态分布的均值

stddev

- 相关正态分布的标准差

返回值

由均值为 mean，标准差为 stddev 的正态分布派生的对数正态分布 double2

描述

从 state 状态的 MRG32k3a 生成器中返回两个由均值为 mean，标准差为 stddev 的正态分布派生的对数正态分布双精度数，生成器的位置递增两位。

该实现使用 Box-Muller 变换生成两个符合正态分布的结果, 并将其转换为对数正态分布。

```
__device__ double2 mcrand_log_normal2_double ( mcrandStatePhilox4_32_10_t*
↪state, double  mean, double  stddev )
```

从 Philox4_32_10 生成器返回两个对数正态分布的双精度浮点数

参数

state

- 指向要更新状态的指针

mean

- 相关正态分布的均值

stddev

- 相关正态分布的标准差

返回值

由均值为 mean，标准差为 stddev 的正态分布派生的对数正态分布 double2

描述

从 state 状态的 Philox4_32_10 生成器中返回两个由均值为 mean，标准差为 stddev 的正态分布派生的对数正态分布双精度数，生成器的位置递增两位。

该实现使用 Box-Muller 变换生成两个符合正态分布的结果, 并将其转换为对数正态分布。

```
__device__ double2 mcrand_log_normal2_double ( mcrandStateXORWOW_t* state,
↪double  mean, double  stddev )
```

从 XORWOW 生成器返回两个对数正态分布的双精度浮点数

参数

state

- 指向要更新状态的指针

mean

- 相关正态分布的均值

stddev

- 相关正态分布的标准差

返回值

由均值为 mean，标准差为 stddev 的正态分布派生的对数正态分布 double2

描述

从 state 状态的 XORWOW 生成器中返回两个由均值为 mean，标准差为 stddev 的正态分布派生的对数正态分布双精度数，生成器的位置递增两位。

该实现使用 Box-Muller 变换生成两个符合正态分布的结果, 并将其转换为对数正态分布。

```
__device__ float4 mcrand_log_normal4 ( mcrandStatePhilox4_32_10_t* state,
↪float  mean, float  stddev )
```

从 Philox4_32_10 生成器返回 4 个正态分布的浮点数。

参数

state

- 指向要更新状态的指针

mean

- 相关正态分布的均值

stddev

- 相关正态分布的标准差

返回值

由均值为 mean，标准差为 stddev 的正态分布派生的对数正态分布 float4

描述

从 state 状态的 MRG32k3a 生成器中四个由均值为 mean，标准差为 stddev 的正态分布派生的对数正态分布浮点数，生成器的位置递增四位。

该实现使用 Box-Muller 变换生成两个符合正态分布的结果, 并将其转换为对数正态分布。

```
__device__ double mcrand_log_normal_double ( mcrandStateScrambledSobol64_
↪t* state, double  mean, double  stddev )
```

从 Scrambled Sobol64 生成器返回一个对数正态分布的双精度浮点数

参数

state

- 指向要更新状态的指针

mean

- 相关正态分布的均值

stddev

- 相关正态分布的标准差

返回值

均值为 mean，标准差为 stddev 的对数正态分布双精度浮点数

描述

从 state 状态的 Scrambled Sobol64 生成器中返回一个由均值为 mean，标准差为 stddev 正态分布派生的对数正态分布双精度浮点数，生成器的位置递增一位。

该实现使用逆累积分布函数来生成正态分布的结果。

```
__device__ double mcrand_log_normal_double ( mcrandStateSobol64_t* state,
↪double  mean, double  stddev )
```

从 Sobol64 生成器返回一个对数正态分布的双精度浮点数

参数

state

- 指向要更新状态的指针

mean

- 相关正态分布的均值

stddev

- 相关正态分布的标准差

返回值

均值为 mean，标准差为 stddev 的对数正态分布双精度浮点数

描述

从 state 状态的 Sobol64 生成器中返回一个由均值为 mean，标准差为 stddev 正态分布派生的对数正态分布双精度浮点数，生成器的位置递增一位。

该实现使用逆累积分布函数来生成正态分布的结果。

```
__device__ double mcrand_log_normal_double ( mcrandStateScrambledSobol32_
↪t* state, double  mean, double  stddev )
```

从 Scrambled Sobol32 生成器返回一个对数正态分布的双精度浮点数

参数

state

- 指向要更新状态的指针

mean

- 相关正态分布的均值

stddev

- 相关正态分布的标准差

返回值

均值为 mean，标准差为 stddev 的对数正态分布双精度浮点数

描述

从 state 状态的 Scrambled Sobol32 生成器中返回一个由均值为 mean，标准差为 stddev 正态分布派生的对数正态分布双精度浮点数，生成器的位置递增一位。

该实现使用逆累积分布函数来生成正态分布的结果，并将其转换为对数正态分布。

```
__device__ double mcrand_log_normal_double ( mcrandStateSobol32_t* state,
↪double  mean, double  stddev )
```

从 Sobol32 生成器返回一个对数正态分布的双精度浮点数

参数

state

- 指向要更新状态的指针

mean

- 相关正态分布的均值

stddev

- 相关正态分布的标准差

返回值

均值为 mean，标准差为 stddev 的对数正态分布双精度浮点数

描述

从 state 状态的 Sobol32 生成器中返回一个由均值为 mean，标准差为 stddev 正态分布派生的对数正态分布双精度浮点数，生成器的位置递增一位。

该实现使用逆累积分布函数来生成正态分布的结果，并将其转换为对数正态分布。

```
__device__ double mcrand_log_normal_double ( mcrandStateMtgp32_t* state,
↪double  mean, double  stddev )
```

从 MTGP32 生成器返回一个对数正态分布的双精度浮点数

参数

state

- 指向要更新状态的指针

mean

- 相关正态分布的均值

stddev

- 相关正态分布的标准差

返回值

均值为 mean，标准差为 stddev 的对数正态分布双精度浮点数

描述

从 state 状态的 MTGP32 生成器中返回一个由均值为 mean，标准差为 stddev 正态分布派生的对数正态分布双精度浮点数，生成器的位置递增一位。

该实现使用逆累积分布函数来生成正态分布的结果，并将其转换为对数正态分布。

```
__device__ double mcrand_log_normal_double ( mcrandStateMRG32k3a_t* state,
↪double  mean, double  stddev )
```

从 MRG32k3a 生成器返回一个对数正态分布的双精度浮点数

参数

state

- 指向要更新状态的指针

mean

- 相关正态分布的均值

stddev

- 相关正态分布的标准差

返回值

均值为 mean，标准差为 stddev 的对数正态分布双精度浮点数

描述

从 state 状态的 MRG32k3a 生成器中返回一个由均值为 mean，标准差为 stddev 正态分布派生的对数正态分布双精度浮点数，生成器的位置递增一位。

该实现使用 Box-Muller 变换生成两个符合正态分布的结果, 将其转换为对数正态分布, 并逐个返回。如果希望以更高效的方式同时返回两个结果，请参见 mcrand_log_normal2_double() 函数。

```
__device__ double mcrand_log_normal_double ( mcrandStatePhilox4_32_10_t*
↪state, double  mean, double  stddev )
```

从 Philox4_32_10 生成器返回一个对数正态分布的双精度浮点数

参数

state

- 指向要更新状态的指针

mean

- 相关正态分布的均值

stddev

- 相关正态分布的标准差

返回值

均值为 mean，标准差为 stddev 的对数正态分布双精度浮点数

描述

从 state 状态的 Philox4_32_10 生成器中返回一个由均值为 mean，标准差为 stddev 正态分布派生的对数正态分布双精度浮点数，生成器的位置递增一位。

该实现使用 Box-Muller 变换生成两个符合正态分布的结果, 将它们转换为对数正态分布, 并逐个返回。如果希望以更高效的方式同时返回两个结果，请参见 mcrand_log_normal2_double() 函数。

```
__device__ double mcrand_log_normal_double ( mcrandStateXORWOW_t* state,
↪double  mean, double  stddev )
```

从 XORWOW 生成器返回一个对数正态分布的双精度浮点数

参数

state

- 指向要更新状态的指针

mean

- 相关正态分布的均值

stddev

- 相关正态分布的标准差

返回值

均值为 mean，标准差为 stddev 的对数正态分布双精度浮点数

描述

从 state 状态的 XORWOW 生成器中返回一个由均值为 mean，标准差为 stddev 正态分布派生的对数正态分布双精度浮点数，生成器的位置递增一位。

该实现使用 Box-Muller 变换生成两个符合正态分布的结果, 将它们转换为对数正态分布, 并逐个返回。如果希望以更高效的方式同时返回两个结果，请参见 mcrand_log_normal2_double() 函数。

```
__device__ float mcrand_mtgp32_single ( mcrandStateMtgp32_t* state )
```

从 mtgp32 生成器返回一个均匀分布的浮点数。

参数

state

- 指向要更新状态的指针

返回值

均匀分布在 0.0f 到 1.0f 之间的浮点数

描述

从 state 状态的 mtgp32 生成器中返回一个在 0.0f 和 1.0f 之间均匀分布的浮点数，生成器的位置递增一位。输出范围不包括 0.0f 但包括 1.0f。不会返回非规格化的浮点数。

注意：提供这种备选的均匀浮点数派生方式是为了与源代码保持完整性。

```
__device__ float mcrand_mtgp32_single_specific ( mcrandStateMtgp32_t*
↪state, unsigned char  index, unsigned char  n )
```

从 mtgp32 生成器的特定位置返回一个均匀分布的浮点数。

参数

state

- 指向要更新状态的指针

index

- 此状态下要提取和更新的位置索引（0..255）

n

- 此状态下要通过此次调用更新的位置总数

返回值

均匀分布在 0.0f 到 1.0f 之间的浮点数

描述

从 state 状态的 mtgp32 生成器的位置索引处返回一个在 0.0f 和 1.0f 之间均匀分布的浮点数，并将生成器增加 n 个位置，其中 n 必须为线程块在此次调用中的状态下更新的位置总数。输出范围不包括 0.0f 但包括 1.0f。不会返回非规格化的浮点数。

注解:

- 线程索引必须在 0 到 n - 1 范围内。更新的位置数不能超过 256。一个线程块可以更新多个状态，但一个给定状态不能由多个线程块更新。
- 提供这种备选的均匀浮点数派生方式是为了与源代码保持完整性。

```
__device__ unsigned int mcrand_mtgp32_specific ( mcrandStateMtgp32_t*
↪state, unsigned char  index, unsigned char  n )
```

从 mtgp32 生成器的特定位置返回 32 位的伪随机数。

参数

state

- 指向要更新状态的指针

index

- 此状态下要提取和更新的位置索引（0..255）

n

- 此状态下要通过此次调用更新的位置总数

返回值

作为无符号整数的 32 位伪随机数，所有位都可用。

描述

从 state 状态的 mtgp32 生成器的位置索引处返回 32 位的伪随机数，并将生成器递增 n 个位置，其中 n 必须为线程块在此调用中的状态下更新的位置总数。

注解: 线程索引必须在 0 到 n - 1 范围内。更新的位置数不能超过 256。一个线程块可以更新多个状态，但是一个给定的状态不能同时被多个线程块更新。

```
__device__ float mcrand_normal ( mcrandStateScrambledSobol64_t* state )
```

从一个 Scrambled Sobol64 生成器返回一个正态分布的浮点数

参数

state

- 指向要更新状态的指针

返回值

均值为 0.0f，标准差为 1.0f 的正态分布浮点数

描述

从 state 状态的 Scrambled Sobol64 生成器中返回一个均值为 0.0f，标准差为 1.0f 的正态分布浮点数，生成器的位置递增一位。

该实现使用逆累积分布函数来生成正态分布的结果。

```
__device__ float mcrand_normal ( mcrandStateSobol64_t* state )
```

从一个 Sobol64 生成器返回一个正态分布的浮点数

参数

state

- 指向要更新状态的指针

返回值

均值为 0.0f，标准差为 1.0f 的正态分布浮点数

描述

从 state 状态的 Sobol64 生成器中返回一个均值为 0.0f，标准差为 1.0f 的正态分布浮点数，生成器的位置递增一位。

该实现使用逆累积分布函数来生成正态分布的结果。

```
__device__ float mcrand_normal ( mcrandStateScrambledSobol32_t* state )
```

从一个 Scrambled Sobol32 生成器返回一个正态分布的浮点数

参数

state

- 指向要更新状态的指针

返回值

均值为 0.0f，标准差为 1.0f 的正态分布浮点数

描述

从 state 状态的 Scrambled Sobol32 生成器中返回一个均值为 0.0f，标准差为 1.0f 的正态分布浮点数，生成器的位置递增一位。

该实现使用逆累积分布函数来生成正态分布的结果。

```
__device__ float mcrand_normal ( mcrandStateSobol32_t* state )
```

从 Sobol32 生成器返回一个正态分布的浮点数。

参数

state

- 指向要更新的状态的指针

返回值

平均值为 0.0f，标准差为 1.0f 的正态分布浮点数

描述

从 state 状态的 Sobol32 生成器中返回一个均值为 0.0f，标准差为 1.0f 的正态分布浮点数，生成器的位置递增一位。

该实现使用逆累积分布函数来生成正态分布的结果。

```
__device__ float mcrand_normal ( mcrandStateMtgp32_t* state )
```

从 MTGP32 生成器返回一个正态分布的浮点数。

参数

state

- 指向要更新的状态的指针

返回值

平均值为 0.0f，标准差为 1.0f 的正态分布浮点数

描述

从 state 状态的 MTGP32 生成器返回一个均值为 0.0f，标准差为 1.0f 的正态分布浮点数，生成器的位置递增一位。

该实现使用逆累积分布函数来生成正态分布的结果。

```
__device__ float mcrand_normal ( mcrandStateMRG32k3a_t* state )
```

从 MRG32k3a 生成器返回一个正态分布的浮点数。

参数

state

- 指向要更新的状态的指针

返回值

平均值为 0.0f，标准差为 1.0f 的正态分布浮点数

描述

从 state 状态的 MRG32k3a 生成器返回一个均值为 0.0f，标准差为 1.0f 的正态分布浮点数，生成器的位置递增一位。

该实现使用 Box-Muller 变换生成两个符合正态分布的结果，并逐个返回。如果希望以更高效的方式同时返回两个结果，请参见 mcrand_normal2() 函数。

```
__device__ float mcrand_normal ( mcrandStatePhilox4_32_10_t* state )
```

从 Philox4_32_10 生成器返回一个正态分布的浮点数。

参数

state

- 指向要更新的状态的指针

返回值

平均值为 0.0f，标准差为 1.0f 的正态分布浮点数

描述

从 state 状态的 Philox4_32_10 生成器返回一个均值为 0.0f，标准差为 1.0f 的正态分布浮点数，生成器的位置递增一位。

该实现使用 Box-Muller 变换生成两个符合正态分布的结果，并逐个返回。如果希望以更高效的方式同时返回两个结果，请参见 mcrand_normal2() 函数。

```
__device__ float mcrand_normal ( mcrandStateXORWOW_t* state )
```

从 XORWOW 生成器返回一个正态分布的浮点数。

参数

state

- 指向要更新的状态的指针

返回值

平均值为 0.0f，标准差为 1.0f 的正态分布浮点数

描述

从 state 状态的 XORWOW 生成器返回一个均值为 0.0f，标准差为 1.0f 的正态分布浮点数，生成器的位置递增一位。

该实现使用 Box-Muller 变换生成两个符合正态分布的结果，并逐个返回。如果希望以更高效的方式同时返回两个结果，请参见 mcrand_normal2() 函数。

```
__device__ float2 mcrand_normal2 ( mcrandStateMRG32k3a_t* state )
```

从 MRG32k3a 生成器返回两个正态分布的浮点数。

参数

state

- 指向要更新的状态的指针

返回值

正态分布的 float2 类型浮点数，其中每个元素都来自均值为 0.0f，标准差为 1.0f 的分布

描述

从 state 状态的 MRG32k3a 生成器返回两个均值为 0.0f，标准差为 1.0f 的正态分布浮点数，生成器的位置递增两位。

该实现使用 Box-Muller 变换生成两个符合正态分布的结果。

```
__device__ float2 mcrand_normal2 ( mcrandStatePhilox4_32_10_t* state )
```

从 Philox4_32_10 生成器返回两个正态分布的浮点数。

参数

state

- 指向要更新的状态的指针

返回值

正态分布 float2 类型浮点数，其中每个元素都来自均值为 0.0f，标准差为 1.0f 的分布

描述

从 state 状态的 Philox4_32_10 生成器返回两个均值为 0.0f，标准差为 1.0f 的正态分布浮点数，生成器的位置递增两位。

该实现使用 Box-Muller 变换生成两个符合正态分布的结果。

```
__device__ float2 mcrand_normal2 ( mcrandStateXORWOW_t* state )
```

从 XORWOW 生成器返回两个正态分布的浮点数。

参数

state

- 指向要更新的状态的指针

返回值

正态分布 float2 类型浮点数，其中每个元素都来自均值为 0.0f，标准差为 1.0f 的分布

描述

从 state 状态的 XORWOW 生成器返回两个均值为 0.0f，标准差为 1.0f 的正态分布浮点数，生成器的位置递增两位。

该实现使用 Box-Muller 变换生成两个符合正态分布的结果。

```
__device__ double2 mcrand_normal2_double ( mcrandStateMRG32k3a_t* state )
```

从 MRG32k3a 生成器返回两个正态分布的双精度浮点数。

参数

state

- 指向要更新的状态的指针

返回值

正态分布 double2 类型双精度浮点数，其中每个元素都来自均值为 0.0，标准差为 1.0 的分布

描述

从 state 状态的 MRG32k3a 生成器返回两个均值为 0.0，标准差为 1.0 的正态分布双精度浮点数，生成器的位置递增两位。

该实现使用 Box-Muller 变换生成两个符合正态分布的结果。

```
__device__ double2 mcrand_normal2_double ( mcrandStatePhilox4_32_10_t*
↪state )
```

从 Philox4_32_10 生成器返回两个正态分布的双精度浮点数。

参数

state

- 指向要更新的状态的指针

返回值

正态分布 double2 类型双精度浮点数，其中每个元素都来自均值为 0.0，标准差为 1.0 的分布

描述

从 state 状态的 Philox4_32_10 生成器返回两个均值为 0.0，标准差为 1.0 的正态分布双精度浮点数，生成器的位置递增两位。

该实现使用 Box-Muller 变换生成两个符合正态分布的结果。

```
__device__ double2 mcrand_normal2_double ( mcrandStateXORWOW_t* state )
```

从 XORWOW 生成器返回两个正态分布的双精度浮点数。

参数

state

- 指向要更新的状态的指针

返回值

正态分布 double2 类型双精度浮点数，其中每个元素都来自均值为 0.0，标准差为 1.0 的分布

描述

从 state 状态的 XORWOW 生成器返回两个均值为 0.0，标准差为 1.0 的正态分布双精度浮点数，生成器的位置递增两位。

该实现使用 Box-Muller 变换生成两个符合正态分布的结果。

```
__device__ float4 mcrand_normal4 ( mcrandStatePhilox4_32_10_t* state )
```

从 Philox4_32_10 生成器返回四个正态分布的浮点数。

参数

state

- 指向要更新的状态的指针

返回值

正态分布 float2 类型浮点数，其中每个元素都来自均值为 0.0f，标准差为 1.0f 的分布

描述

从 state 状态的 Philox4_32_10 生成器返回四个均值为 0.0f，标准差为 1.0f 的正态分布浮点数，生成器的位置递增四位。

该实现使用 Box-Muller 变换生成两个符合正态分布的结果。

```
__device__ double mcrand_normal_double ( mcrandStateScrambledSobol64_t*
↪state )
```

从 Scrambled Sobol64 生成器返回一个正态分布的双精度浮点数。

参数

state

- 指向要更新的状态的指针

返回值

均值为 0.0，标准差为 1.0 的正态分布双精度浮点数

描述

从 state 状态的 Scrambled Sobol64 生成器返回一个均值为 0.0，标准差为 1.0 的正态分布双精度浮点数，生成器的位置递增一位。

该实现使用逆累积分布函数来生成正态分布的结果。

```
__device__ double mcrand_normal_double ( mcrandStateSobol64_t* state )
```

从 Sobol64 生成器返回一个正态分布的双精度浮点数。

参数

state

- 指向要更新的状态的指针

返回值

均值为 0.0，标准差为 1.0 的正态分布双精度浮点数

描述

从 state 状态的 Scrambled Sobol64 生成器返回一个均值为 0.0，标准差为 1.0 的正态分布双精度浮点数，生成器的位置递增一位。

该实现使用逆累积分布函数来生成正态分布的结果。

```
__device__ double mcrand_normal_double ( mcrandStateScrambledSobol32_t*
↪state )
```

从 Scrambled Sobol32 生成器返回一个正态分布的双精度浮点数。

参数

state

- 指向要更新的状态的指针

返回值

均值为 0.0，标准差为 1.0 的正态分布双精度浮点数

描述

从 state 状态的 Scrambled Sobol32 生成器返回一个均值为 0.0，标准差为 1.0 的正态分布双精度浮点数，生成器的位置递增一位。

该实现使用逆累积分布函数来生成正态分布的结果。

```
__device__ double mcrand_normal_double ( mcrandStateSobol32_t* state )
```

从 Sobol32 生成器返回一个正态分布的双精度浮点数。

参数

state

- 指向要更新的状态的指针

返回值

均值为 0.0，标准差为 1.0 的正态分布双精度浮点数

描述

从 state 状态的 Sobol32 生成器返回一个均值为 0.0，标准差为 1.0 的正态分布双精度浮点数，生成器的位置递增一位。

该实现使用逆累积分布函数来生成正态分布的结果。

```
__device__ double mcrand_normal_double ( mcrandStateMtgp32_t* state )
```

从 MTGP32 生成器返回一个正态分布的双精度浮点数。

参数

state

- 指向要更新的状态的指针

返回值

均值为 0.0，标准差为 1.0 的正态分布双精度浮点数

描述

从 state 状态的 MTGP32 生成器返回一个均值为 0.0，标准差为 1.0 的正态分布双精度浮点数，生成器的位置递增一位。

该实现使用逆累积分布函数来生成正态分布的结果。

```
__device__ double mcrand_normal_double ( mcrandStateMRG32k3a_t* state )
```

从 MRG32k3a 生成器返回一个正态分布的双精度浮点数。

参数

state

- 指向要更新的状态的指针

返回值

均值为 0.0，标准差为 1.0 的正态分布双精度浮点数

描述

从 state 状态的 XORWOW 生成器返回一个均值为 0.0，标准差为 1.0 的正态分布双精度浮点数，生成器的位置递增一位。

该实现使用 Box-Muller 变换生成两个符合正态分布的结果，并逐个返回。如果希望以更高效的方式同时返回两个结果，请参见 mcrand_normal2() 函数。

```
__device__ double mcrand_normal_double ( mcrandStatePhilox4_32_10_t* state
↪)
```

从 Philox4_32_10 生成器返回一个正态分布的双精度浮点数。

参数

state

- 指向要更新的状态的指针

返回值

均值为 0.0，标准差为 1.0 的正态分布双精度浮点数

描述

从 state 状态的 Philox4_32_10 生成器返回一个均值为 0.0，标准差为 1.0 的正态分布双精度浮点数，生成器的位置递增一位。

该实现使用 Box-Muller 变换生成两个符合正态分布的结果，并逐个返回。如果希望以更高效的方式同时返回两个结果，请参见 mcrand_normal2() 函数。

```
__device__ double mcrand_normal_double ( mcrandStateXORWOW_t* state )
```

从 XORWOW 生成器返回一个正态分布的双精度浮点数。

参数

state

- 指向要更新的状态的指针

返回值

均值为 0.0，标准差为 1.0 的正态分布双精度浮点数

描述

从 state 状态的 XORWOW 生成器返回一个均值为 0.0，标准差为 1.0 的正态分布双精度浮点数，生成器的位置递增一位。

该实现使用 Box-Muller 变换生成两个符合正态分布的结果，并逐个返回。如果希望以更高效的方式同时返回两个结果，请参见 mcrand_normal2() 函数。

```
__device__ unsigned int mcrand_poisson ( mcrandStateScrambledSobol64_t*
↪state, double  lambda )
```

从 Scrambled Sobol64 生成器返回一个泊松分布的无符号整型数。

参数

state

- 指向要更新的状态的指针

lambda

- 泊松分布的 Lambda

返回值

lambda 的值为 lambda 的泊松分布无符号整数

描述

从 state 状态的 Scrambled Sobol64 生成器中使用 lambda 的值为 lambda 的泊松分布返回单个无符号整型，生成器的位置递增一位。

```
__device__ unsigned int mcrand_poisson ( mcrandStateSobol64_t* state,
↪double  lambda )
```

从 Sobol64 生成器返回一个泊松分布的无符号整型数。

参数

state

- 指向要更新的状态的指针

lambda

- 泊松分布的 Lambda

返回值

lambda 的值为 lambda 的泊松分布无符号整数

描述

从 state 状态的 Sobol64 生成器中使用 lambda 的值为 lambda 的泊松分布返回单个无符号整型，生成器的位置递增一位。

```
__device__ unsigned int mcrand_poisson ( mcrandStateScrambledSobol32_t*
↪state, double  lambda )
```

从 Scrambled Sobol32 生成器返回一个泊松分布的无符号整型数。

参数

state

- 指向要更新的状态的指针

lambda

- 泊松分布的 Lambda

返回值

lambda 的值为 lambda 的泊松分布无符号整数

描述

从 state 状态的 Scrambled Sobol32 生成器中使用 lambda 的值为 lambda 的泊松分布返回单个无符号整型，生成器的位置递增一位。

```
__device__ unsigned int mcrand_poisson ( mcrandStateSobol32_t* state,
↪double  lambda )
```

从 Sobol32 生成器返回泊松分布的无符号整数。

参数

state

- 指向要更新的状态的指针

lambda

- 泊松分布的 Lambda

返回值

lambda 的值为 lambda 的泊松分布无符号整数

描述

从 state 状态的 Sobol32 生成器中使用 lambda 的值为 lambda 的泊松分布返回单个无符号整型，生成器的位置递增一位。

```
__device__ unsigned int mcrand_poisson ( mcrandStateMtgp32_t* state,
↪double  lambda )
```

从 MTGP32 生成器返回泊松分布的无符号整数。

参数

state

- 指向要更新的状态的指针

lambda

- 泊松分布的 Lambda

返回值

lambda 的值为 lambda 的泊松分布无符号整数

描述

从 state 状态的 MTGP32 生成器中返回 lambda 的值为 lambda 的泊松分布的单个整型，生成器的位置递增一位。

```
__device__ unsigned int mcrand_poisson ( mcrandStateMRG32k3a_t* state,
↪double  lambda )
```

从 MRG32k3A 生成器返回泊松分布的无符号整数。

参数

state

- 指向要更新的状态的指针

lambda

- 泊松分布的 Lambda

返回值

lambda 的值为 lambda 的泊松分布无符号整数

描述

从 state 状态的 MRG32k3a 生成器返回一个 lambda 的值为 lambda 的泊松分布无符号整数，并根据所使用的算法，以可变的数量递增生成器的位置。

```
__device__ unsigned int mcrand_poisson ( mcrandStatePhilox4_32_10_t* state,
↪ double  lambda )
```

从 Philox4_32_10 生成器返回泊松分布的无符号整数。

参数

state

- 指向要更新的状态的指针

lambda

- 泊松分布的 Lambda

返回值

lambda 的值为 lambda 的泊松分布无符号整数

描述

从 state 状态的 Philox4_32_10 生成器中返回一个 lambda 的值为 lambda 的泊松分布无符号整数，并根据所使用的算法，以可变的数量递增生成器的位置。

```
__device__ unsigned int mcrand_poisson ( mcrandStateXORWOW_t* state,
↪double  lambda )
```

从 XORWOW 生成器返回泊松分布的无符号整数。

参数

state

- 指向要更新的状态的指针

lambda

- 泊松分布的 Lambda

返回值

lambda 的值为 lambda 的泊松分布无符号整数

描述

从 state 状态的 XORWOW 生成器中返回一个 lambda 的值为 lambda 的泊松分布无符号整数，并根据所使用的算法，以可变的数量递增生成器的位置。

```
__device__ uint4 mcrand_poisson4 ( mcrandStatePhilox4_32_10_t* state,
↪double  lambda )
```

从 Philox4_32_10 生成器返回四个泊松分布的无符号整数。

参数

state

- 指向要更新的状态的指针

lambda

- 泊松分布的 Lambda

返回值

lambda 的值为 lambda 的泊松分布无符号整数

描述

从 state 状态的 Philox4_32_10 生成器返回四个 lambda 的值为 lambda 的泊松分布无符号整数，并根据所使用的算法，以可变的数量递增生成器的位置。

```
__device__ float mcrand_uniform ( mcrandStateScrambledSobol64_t* state )
```

从 Scrambled Sobol64 生成器返回一个均匀分布的浮点数。

参数

state

- 指向要更新的状态的指针

返回值

介于 0.0f 和 1.0f 之间的均匀分布的浮点数

描述

从 state 状态的 Scrambled Sobol64 生成器返回一个分布在 0.0f 到 1.0f 之间的均匀浮点数，生成器的位置递增一位。输出范围不包括 0.0f，但包括 1.0f。不返回非标准化的浮点数输出。

该实现保证使用了对 mcrand() 的单次调用。

```
__device__ float mcrand_uniform ( mcrandStateSobol64_t* state )
```

从 Sobol64 生成器返回均匀分布的浮点数。

参数

state

- 指向要更新的状态的指针

返回值

介于 0.0f 和 1.0f 之间的均匀分布的浮点数

描述

从 state 状态的 Sobol64 生成器返回一个分布在 0.0f 到 1.0f 之间的均匀浮点数，生成器的位置递增一位。输出范围不包括 0.0f，但包括 1.0f。不返回非标准化的浮点数输出。

该实现保证使用了对 mcrand() 的单次调用。

```
__device__ float mcrand_uniform ( mcrandStateScrambledSobol32_t* state )
```

从 Scrambled Sobol32 生成器返回均匀分布的浮点数。

参数

state

- 指向要更新的状态的指针

返回值

介于 0.0f 和 1.0f 之间的均匀分布的浮点数

描述

从 state 状态的 Scrambled Sobol32 生成器返回一个分布在 0.0f 到 1.0f 之间的均匀浮点数，生成器的位置递增一位。输出范围不包括 0.0f，但包括 1.0f。不返回非标准化的浮点数输出。

该实现保证使用了对 mcrand() 的单次调用。

```
__device__ float mcrand_uniform ( mcrandStateSobol32_t* state )
```

从 Sobol32 生成器返回均匀分布的浮点数。

参数

state

- 指向要更新的状态的指针

返回值

介于 0.0f 和 1.0f 之间的均匀分布的浮点数

描述

从 state 状态的 Sobol32 生成器返回一个分布在 0.0f 到 1.0f 之间的均匀浮点数，生成器的位置递增一位。输出范围不包括 0.0f，但包括 1.0f。不返回非标准化的浮点数输出。

该实现保证使用了对 mcrand() 的单次调用。

```
__device__ float mcrand_uniform ( mcrandStateMtgp32_t* state )
```

从 MTGP32 生成器返回均匀分布的浮点数。

参数

state

- 指向要更新的状态的指针

返回值

介于 0.0f 和 1.0f 之间的均匀分布的浮点数

描述

从 state 状态的 MTGP32 生成器返回一个分布在 0.0f 到 1.0f 之间的均匀浮点数，生成器的位置递增一位。输出范围不包括 0.0f，但包括 1.0f。不返回非标准化的浮点数输出。

```
__device__ float mcrand_uniform ( mcrandStatePhilox4_32_10_t* state )
```

从 Philox4_32_10 生成器返回均匀分布的浮点数。

参数

state

- 指向要更新的状态的指针

返回值

0.0f 到 1.0f 之间均匀分布的浮点数

描述

从 state 状态的 Philox4_32_10 生成器返回一个分布在 0.0f 到 1.0f 之间的均匀浮点数，生成器的位置递增一位。输出范围不包括 0.0f，但包括 1.0f。不返回非标准化的浮点数输出。

```
__device__ float mcrand_uniform ( mcrandStateMRG32k3a_t* state )
```

从 MRG32k3a 生成器返回均匀分布的浮点数。

参数

state

- 指向要更新的状态的指针

返回值

介于 0.0f 和 1.0f 之间的均匀分布的浮点数

描述

从 state 状态的 MRG32k3a 生成器返回一个分布在 0.0f 到 1.0f 之间的均匀浮点数，生成器的位置递增一位。输出范围不包括 0.0f，但包括 1.0f。不返回非标准化的浮点数输出。

该实现返回最多 23 位尾数，并带有最小返回值

```
__device__ float mcrand_uniform ( mcrandStateXORWOW_t* state )
```

从 XORWOW 生成器返回均匀分布的浮点数。

参数

state

- 指向要更新的状态的指针

返回值

介于 0.0f 和 1.0f 之间的均匀分布的浮点数

描述

从 state 状态的 XORWOW 生成器返回一个分布在 0.0f 到 1.0f 之间的均匀浮点数，生成器的位置递增一位。输出范围不包括 0.0f，但包括 1.0f。不返回非标准化的浮点数输出。

该实现可以使用任意数量的 mcrand() 调用来获取足够的随机位来创建返回值。当前的实现使用一次调用。

```
__device__ double2 mcrand_uniform2_double ( mcrandStatePhilox4_32_10_t*
↪state )
```

从 Philox4_32_10 生成器返回包含 2 个双精度浮点数的均匀分布元组。

参数

state

- 指向要更新的状态的指针

返回值

2 个均匀分布在 0.0f 和 1.0f 之间的双精度浮点数

描述

从 state 状态的 Philox4_32_10 生成器返回 0.0f 和 1.0f 之间均匀分布的 2 个双精度浮点数 (double4)，将生成器的位置递增四位。输出范围不包括 0.0f，但包括 1.0f。不返回非标准化的浮点数输出。

```
__device__ float4 mcrand_uniform4 ( mcrandStatePhilox4_32_10_t* state )
```

从 Philox4_32_10 生成器返回包含 4 个浮点数的均匀分布元组。

参数

state

- 指向要更新的状态的指针

返回值

0.0f 到 1.0f 之间均匀分布的浮点数

描述

从 state 状态的 Philox4_32_10 生成器返回 0.0f 和 1.0f 之间均匀分布的 4 个浮点数，将生成器的位置递增四位。输出范围不包括 0.0f，但包括 1.0f。不返回非标准化的浮点数输出。

```
__device__ double mcrand_uniform_double ( mcrandStateScrambledSobol64_t*
↪state )
```

从 Scrambled Sobol64 生成器返回均匀分布的双精度浮点数。

参数

state

- 指向要更新的状态的指针

返回值

0.0f 和 1.0f 之间均匀分布的双精度浮点数

描述

从 state 状态的打乱后的 Sobol64 生成器返回一个在 0.0f 到 1.0f 之间的均匀分布的双精度浮点数，生成器的位置递增一位。输出范围不包括 0.0f，但包括 1.0f。不返回非标准化的浮点数输出。

该实现保证使用对 mcrand() 的单次调用以保持序列的准随机属性。

```
__device__ double mcrand_uniform_double ( mcrandStateSobol64_t* state )
```

从 Sobol64 生成器返回均匀分布的双精度浮点数。

参数

state

- 指向要更新的状态的指针

返回值

0.0f 和 1.0f 之间均匀分布的双精度浮点数

描述

从 state 状态的 Sobol64 生成器返回一个在 0.0f 到 1.0f 之间的均匀分布的双精度浮点数，生成器的位置递增一位。输出范围不包括 0.0f，但包括 1.0f。不返回非标准化的浮点数输出。

该实现保证使用对 mcrand() 的单次调用以保持序列的准随机属性。

```
__device__ double mcrand_uniform_double ( mcrandStateScrambledSobol32_t*
↪state )
```

从 Scrambled Sobol32 生成器返回均匀分布的双精度浮点数。

参数

state

- 指向要更新的状态的指针

返回值

0.0f 和 1.0f 之间均匀分布的双精度浮点数

描述

从 state 状态的打乱后的 Sobol32 生成器返回一个在 0.0f 到 1.0f 之间的均匀分布的双精度浮点数，生成器的位置递增一位。输出范围不包括 0.0f，但包括 1.0f。不返回非标准化的浮点数输出。

该实现保证使用对 mcrand() 的单次调用以保持序列的准随机属性。

请注意，该实现仅使用 32 个随机位来生成单个双精度值。

```
__device__ double mcrand_uniform_double ( mcrandStateSobol32_t* state )
```

从 Sobol32 生成器返回均匀分布的双精度浮点数。

参数

state

- 指向要更新的状态的指针

返回值

0.0f 和 1.0f 之间均匀分布的双精度浮点数

描述

从 state 状态的 Sobol32 生成器返回一个 0.0f 到 1.0f 之间的均匀分布的双精度浮点数，生成器的位置递增一位。输出范围不包括 0.0f，但包括 1.0f。不返回非标准化的浮点数输出。

该实现保证使用对 mcrand() 的单次调用以保持序列的准随机属性。

请注意，该实现仅使用 32 个随机位来生成单个双精度值。

```
__device__ double mcrand_uniform_double ( mcrandStatePhilox4_32_10_t*
↪state )
```

从 Philox4_32_10 生成器返回均匀分布的双精度浮点数。

参数

state

- 指向要更新的状态的指针

返回值

0.0f 和 1.0f 之间均匀分布的双精度浮点数

描述

从 state 状态的 Philox4_32_10 生成器返回一个在 0.0f 到 1.0f 之间的均匀分布的双精度浮点数，生成器的位置递增一位。输出范围不包括 0.0f，但包括 1.0f。不返回非标准化的浮点数输出。

请注意，该实现仅使用 32 个随机位来生成单个双精度值。

建议使用 mcrand_uniform2_double() 来生成更高质量的均匀分布双精度值。

```
__device__ double mcrand_uniform_double ( mcrandStateMtgp32_t* state )
```

从 MTGP32 生成器返回均匀分布的双精度浮点数。

参数

state

- 指向要更新的状态的指针

返回值

0.0f 和 1.0f 之间均匀分布的双精度浮点数

描述

从 state 状态的 MTGP32 生成器返回一个介于 0.0f 和 1.0f 之间的均匀分布双精度浮点数，生成器的位置递增一位。输出范围不包括 0.0f，但包括 1.0f。不返回非标准化的浮点数输出。

请注意，该实现仅使用 32 个随机位来生成单个双精度值。

```
__device__ double mcrand_uniform_double ( mcrandStateMRG32k3a_t* state )
```

从 MRG32k3a 生成器返回均匀分布的双精度浮点数。

参数

state

- 指向要更新的状态的指针

返回值

0.0f 和 1.0f 之间均匀分布的双精度浮点数

描述

从 state 状态的 MRG32k3a 生成器返回一个介于 0.0f 和 1.0f 之间的均匀分布双精度浮点数，生成器的位置递增一位。输出范围不包括 0.0f，但包括 1.0f。不返回非标准化的浮点数输出。

请注意，如 L’ Ecuyer 的开创性论文中所述，该实现最多返回 32 个随机尾数位。

```
__device__ double mcrand_uniform_double ( mcrandStateXORWOW_t* state )
```

从 XORWOW 生成器返回均匀分布的双精度浮点数。

参数

state

- 指向要更新的状态的指针

返回值

0.0f 和 1.0f 之间均匀分布的双精度浮点数

描述

从 state 状态的 XORWOW 生成器返回一个在 0.0f 到 1.0f 之间的均匀分布的双精度浮点数，生成器的位置递增一位。输出范围不包括 0.0f，但包括 1.0f。不返回非标准化的浮点数输出。

该实现可以使用任意数量的 mcrand() 调用来获取足够的随机位来创建返回值。当前的实现仅使用两次调用。

```
template < typename T >
__device__ MCRAND_STD::enable_if < MCRAND_STD::is_same &lt;
↪mcrandStateSobol64_t* > ::type skipahead ( unsigned long long n, T state
↪) [inline]
```

更新 Sobol64 状态，以跳过 n 个元素。

参数

n

- 要跳过的元素数

state

- 指向要更新的状态的指针

描述

更新 state 中的 Sobol64 状态，以跳过 n 个元素。

所有 n 值均有效。

```
template < typename T >
__device__ MCRAND_STD::enable_if < MCRAND_STD::is_same &lt;
↪mcrandStateSobol32_t* > ::type skipahead ( unsigned int  n, T state )
↪[inline]
```

更新 Sobol32 状态，以跳过 n 个元素。

参数

n

- 要跳过的元素数

state

- 指向要更新的状态的指针

描述

更新 state 中的 Sobol32 状态，以跳过 n 个元素。

所有 n 值均有效。

```
__device__ void skipahead ( unsigned long long n, mcrandStateMRG32k3a_t*
↪state )
```

更新 MRG32k3a 状态，以跳过 n 个元素。

参数

n

- 要跳过的元素数

state

- 指向要更新的状态的指针

描述

更新 state 中的 MRG32k3a 状态以跳过 n 个元素。

所有 n 值均有效。较大的数值需要更多计算，因此需要更多时间来完成。

```
__device__ void skipahead ( unsigned long long n, mcrandStatePhilox4_32_10_
↪t* state )
```

更新 Philox4_32_10 状态，以跳过 n 个元素。

参数

n

- 要跳过的元素数

state

- 指向要更新的状态的指针

描述

更新 state 中的 Philox4_32_10 状态以跳过 n 个元素。

所有 n 值均有效。

```
__device__ void skipahead ( unsigned long long n, mcrandStateXORWOW_t*
↪state )
```

更新 XORWOW 状态，跳过 n 个元素。

参数

n

- 要跳过的元素数

state

- 指向要更新的状态的指针

描述

更新 state 中的 XORWOW 状态，以跳过 n 个元素。

所有 n 值均有效。较大的数值需要更多计算，因此需要更多时间来完成。

```
__device__ void skipahead_sequence ( unsigned long long n,
↪mcrandStateMRG32k3a_t* state )
```

更新 MRG32k3a 状态，跳过 n 个序列。

参数

n

- 要跳过的序列数

state

- 指向要更新的状态的指针

描述

更新 state 中的 MRG32k3a 状态，跳过 n 个序列。每个序列长 2127 个元素，因此该函数将跳过 2127 * n 个元素。

所有 n 值均有效。较大的数值需要更多计算，因此需要更多时间来完成。

```
__device__ void skipahead_sequence ( unsigned long long n,
↪mcrandStatePhilox4_32_10_t* state )
```

更新 Philox4_32_10 状态，以跳过 n 个子序列。

参数

n

- 要跳过的子序列数

state

- 指向要更新的状态的指针

描述

更新 state 中的 Philox4_32_10 状态，以跳过 n 个子序列。每个子序列有 266 个元素，因此该函数将跳过 266 * n 个元素。

所有 n 值均有效。

```
__device__ void skipahead_sequence ( unsigned long long n,
↪mcrandStateXORWOW_t* state )
```

更新 XORWOW 状态，跳过 n 个子序列。

参数

n

- 要跳过的子序列数

state

- 指向要更新的状态的指针

描述

更新 state 中的 XORWOW 状态，以跳过 n 个子序列。每个子序列有 267 个元素长，这意味着函数将跳过 267 * n 个元素。

所有 n 值均有效。较大的数值需要更多计算，因此需要更多时间来完成。

```
__device__ void skipahead_subsequence ( unsigned long long n,
↪mcrandStateMRG32k3a_t* state )
```

更新 MRG32k3a 状态，以跳过 n 个子序列。

参数

n

- 要跳过的子序列数

state

- 指向要更新的状态的指针

描述

更新 state 中的 MRG32k3a 状态，以跳过 n 个子序列。每个子序列为 2127276 个元素长，因此这意味着该函数将跳过 267 * n 个元素。

n 的有效值为 0 到 251。注意 n 将被屏蔽为 51 位。