2.编程模型

本章通过概述 CUDA 编程模型是如何在 c++中公开的，来介绍 CUDA 的主要概念。

编程接口中给出了对 CUDA C++ 的广泛描述。

本章和下一章中使用的向量加法示例的完整代码可以在 vectorAdd CUDA 示例中找到。

2.1 内核

CUDA C++ 通过允许程序员定义称为 kernel 的 C++ 函数来扩展 C++，当调用内核时，由 N 个不同的 CUDA 线程并行执行 N 次，而不是像常规 C++ 函数那样只执行一次。

使用 `__global__` 声明说明符定义内核，并使用新的 `<<<...>>>` 执行配置语法指定内核调用的 CUDA 线程数（请参阅 C++ 语言扩展）。 每个执行内核的线程都有一个唯一的线程 ID，可以通过内置变量在内核中访问。

作为说明，以下示例代码使用内置变量 `threadIdx` 将两个大小为 N 的向量 A 和 B 相加，并将结果存储到向量 C 中：

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
```

```
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
    ...
}
```

这里，执行 VecAdd() 的 N 个线程中的每一个线程都会执行一个加法。

2.2 线程层次

为方便起见，threadIdx 是一个 3 分量向量，因此可以使用一维、二维或三维的线程索引来识别线程，形成一个一维、二维或三维的线程块，称为 block。这提供了一种跨域的元素（例如向量、矩阵或体积）调用计算的方法。

线程的索引和它的线程 ID 以一种直接的方式相互关联：对于一维块，它们是相同的； 对于大小为(Dx, Dy)的二维块，索引为(x, y)的线程的线程 ID 为(x + y*Dx)； 对于大小为 (Dx, Dy, Dz) 的三维块，索引为 (x, y, z) 的线程的线程 ID 为 (x + y*Dx + z*Dx*Dy)。

例如，下面的代码将两个大小为 NxN 的矩阵 A 和 B 相加，并将结果存储到矩阵 C 中:

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
                       float C[N][N])
{
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    // Kernel invocation with one block of N * N * 1 threads
    int numBlocks = 1;
```

```
    dim3 threadsPerBlock(N, N);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

每个块的线程数量是有限制的，因为一个块的所有线程都应该驻留在同一个处理器核心上，并且必须共享该核心有限的内存资源。在当前的 gpu 上，一个线程块可能包含多达 1024 个线程。

但是，一个内核可以由多个形状相同的线程块执行，因此线程总数等于每个块的线程数乘以块数。

块被组织成一维、二维或三维的线程块网格(grid)，如下图所示。网格中的线程块数量通常由正在处理的数据的大小决定，通常超过系统中的处理器数量。

<<<...>>> 语法中指定的每个块的线程数和每个网格的块数可以是 int 或 dim3 类型。如上例所示，可以指定二维块或网格。

网格中的每个块都可以由一个一维、二维或三维的惟一索引标识，该索引可以通过内置的 blockIdx 变量在内核中访问。线程块的维度可以通过内置的 blockDim 变量在内核中访问。

扩展前面的 MatAdd()示例来处理多个块，代码如下所示。

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
float C[N][N])
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j < N)
        C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    // Kernel invocation
    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

线程块大小为 16x16(256 个线程)，尽管在本例中是任意更改的，但这是一种常见的选择。网格是用足够的块创建的，这样每个矩阵元素就有一个线程来处理。为简单起见，本例假设每个维度中每个网格的线程数可以被该维度中每个块的线程数整除，尽管事实并非如此。

程块需要独立执行：必须可以以任何顺序执行它们，并行或串行。 这种独立性要求允许跨任意数量的内核以任意顺序调度线程块，如下图所示，使程序员能够编写随内核数量扩展的代码。

块内的线程可以通过一些共享内存共享数据并通过同步它们的执行来协调内存访问来进行协作。 更准确地说，可以通过调用 `__syncthreads()` 内部函数来指定内核中的同步点； `__syncthreads()` 充当屏障，块中的所有线程必须等待，然后才能继续。 Shared Memory 给出了一个使用共享内存的例子。 除了 `__syncthreads()` 之外，Cooperative Groups API 还提供了一组丰富的线程同步示例。

为了高效协作，共享内存是每个处理器内核附近的低延迟内存（很像 L1 缓存），并且 `__syncthreads()` 是轻量级的。

2.3 存储单元层次

CUDA 线程可以在执行期间从多个内存空间访问数据，如下图所示。每个线程都有私有的本地内存。 每个线程块都具有对该块的所有线程可见的共享内存，并且具有与该块相同的生命周期。 所有线程都可以访问相同的全局内存。

还有两个额外的只读内存空间可供所有线程访问：常量和纹理内存空间。 全局、常量和纹理内存空间针对不同的内存使用进行了优化（请参阅设备内存访问）。 纹理内存还为某些特定数据格式提供不同的寻址模式以及数据过滤（请参阅纹理和表面内存）。

全局、常量和纹理内存空间在同一应用程序的内核启动中是持久的。

2.4 异构编程

如下图所示，CUDA 编程模型假定 CUDA 线程在物理独立的设备上执行，该设备作为运行 C++ 程序的主机的协处理器运行。例如，当内核在 GPU 上执行而 C++ 程序的其余部分在 CPU 上执行时，就是这种情况。

CUDA 编程模型还假设主机(host)和设备(device)都在 DRAM 中维护自己独立的内存空间，分别称为主机内存和设备内存。因此，程序通过调用 CUDA 运行时（在编程接口中描述）来管理内核可见的全局、常量和纹理内存空间。这包括设备内存分配和释放以及主机和设备内存之间的数据传输。
统一内存提供托管内存来桥接主机和设备内存空间。托管内存可从系统中的所有 CPU 和 GPU 访问，作为具有公共地址空间的单个连贯内存映像。此功能可实现设备内存的超额订阅，并且无需在主机和设备上显式镜像数据，从而大大简化了移植应用程序的任务。有关统一内存的介绍，请参阅统一内存编程。

注:串行代码在主机(host)上执行，并行代码在设备(device)上执行。

2.5 异步 SIMT 编程模型

在 CUDA 编程模型中，线程是进行计算或内存操作的最低抽象级别。 从基于 NVIDIA Ampere GPU 架构的设备开始，CUDA 编程模型通过异步编程模型为内存操作提供加速。 异步编程模型定义了与 CUDA 线程相关的异步操作的行为。

异步编程模型为 CUDA 线程之间的同步定义了异步屏障的行为。 该模型还解释并定义了如何使用 cuda::memcpy_async 在 GPU 计算时从全局内存中异步移动数据。

2.5.1 异步操作

异步操作定义为由 CUDA 线程发起的操作，并且与其他线程一样异步执行。在结构良好的程序中，一个或多个 CUDA 线程与异步操作同步。发起异步操作的 CUDA 线程不需要在同步线程中.

这样的异步线程（as-if 线程）总是与发起异步操作的 CUDA 线程相关联。异步操作使用同步对象来同步操作的完成。这样的同步对象可以由用户显式管理（例如，`cuda::memcpy_async`）或在库中隐式管理（例如，`cooperative_groups::memcpy_async`）。

同步对象可以是 `cuda::barrier` 或 `cuda::pipeline`。这些对象在 Asynchronous Barrier 和 Asynchronous Data Copies using cuda::pipeline.中进行了详细说明。这些同步对象可以在不同的线程范围内使用。作用域定义了一组线程，这些线程可以使用同步对象与异步操作进行同步。下表定义了 CUDA c++中可用的线程作用域，以及可以与每个线程同步的线程。

Thread Scope	Description
cuda::thread_scope::thread_scope_thread	Only the CUDA thread which initiated asynchronous operations synchronizes.
cuda::thread_scope::thread_scope_block	All or any CUDA threads within the same thread block as the initiating thread synchronizes.

Thread Scope	Description
cuda::thread_scope::thread_scope_device	All or any CUDA threads in the same GPU device as the initiating thread synchronizes.
cuda::thread_scope::thread_scope_system	All or any CUDA or CPU threads in the same system as the initiating thread synchronizes.

这些线程作用域是在 CUDA 标准 c++库中作为标准 c++的扩展实现的。

2.6 Compute Capability

设备的 Compute Capability 由版本号表示，有时也称其“SM 版本”。该版本号标识 GPU 硬件支持的特性，并由应用程序在运行时使用，以确定当前 GPU 上可用的硬件特性和指令。

Compute Capability 包括一个主要版本号 X 和一个次要版本号 Y，用 X.Y 表示

主版本号相同的设备具有相同的核心架构。设备的主要修订号是 8，为 NVIDIA Ampere GPU 的体系结构的基础上,7 基于 Volta 设备架构,6 设备基于 Pascal 架构,5 设备基于 Maxwell 架构,3 基于 Kepler 架构的设备,2 设备基于 Fermi 架构,1 是基于 Tesla 架构的设备。

次要修订号对应于对核心架构的增量改进，可能包括新特性。

Turing 是计算能力 7.5 的设备架构，是基于 Volta 架构的增量更新。

CUDA-Enabled GPUs 列出了所有支持 CUDA 的设备及其计算能力。Compute Capabilities 给出了每个计算能力的技术规格。

注意:特定 GPU 的计算能力版本不应与 CUDA 版本(如 CUDA 7.5、CUDA 8、CUDA 9)混淆，CUDA 版本指的是 CUDA 软件平台的版本。CUDA 平台被应用开发人员用来创建运行在许多代 GPU 架构上的应用程序，包括未来尚未发明的 GPU 架构。尽管 CUDA 平台的新版本通常会通过支持新的 GPU 架构的计算能力版本来增加对该架构的本地支持，但 CUDA 平台的新版本通常也会包含软件功能。

从 CUDA 7.0 和 CUDA 9.0 开始，不再支持 Tesla 和 Fermi 架构。