NVIDIA CUDA
计算统一设备架构

编程指南

版本 2.0

6 / 7 / 2008

目　　录

图表目录

第 1 章 简介

1.1 CUDA：可伸缩并行编程模型

多核 CPU 和众核（manycore） GPU 的出现意味着主流处理器芯片已进入并行时代。此外，根据摩尔定律，其并行性还会不断扩展。这给我们带来了严峻的挑战——我们需要开发出可透明地扩展并行性的应用软件，以利用日益增加的处理器内核数量，这种情况正如 3D 图形应用程序透明地扩展其并行性以支持配备各种数量的内核的众核 GPU。

CUDA 是一种并行编程模型和软件环境，用于应对这种挑战。而对于熟悉 C 语言等标准编程语言的程序员来说，迅速掌握 CUDA 绝非难事。

CUDA 的核心有三个重要抽象概念：线程组层次结构、共享存储器、栅障同步（barrier synchronization），对于程序员来说，它们只是 C 语言的一个极小扩展。

这些抽象提供了细粒度的数据并行和线程并行，嵌套于粗粒度的数据并行和任务并行之中。它们将指导程序员将问题分解为可独立处理的粗粒度子问题，再细分成细粒度的片段，以便通过协作的方法并行解决。这样的分解以允许线程在解决子问题时协作为目的设计了编程语言的表达方式（language expressivity），同时通过在任何可用处理器内核上处理各子问题来支持透明的可伸缩性：因而，编译后的 CUDA 程序可以在任何数量的处理器内核上执行，只有运行时系统（runtime system）需要了解物理处理器数量。

1.2 GPU：高度并行化的多线程、众核处理器

为满足消费者对实时、高清晰度的 3D 图形永无尽头的需求，可编程 GPU 已发展成为一种高度并行化的多线程、众核处理器，具有杰出的计算能力和极高的存储器带宽，如图 1-1 所示。

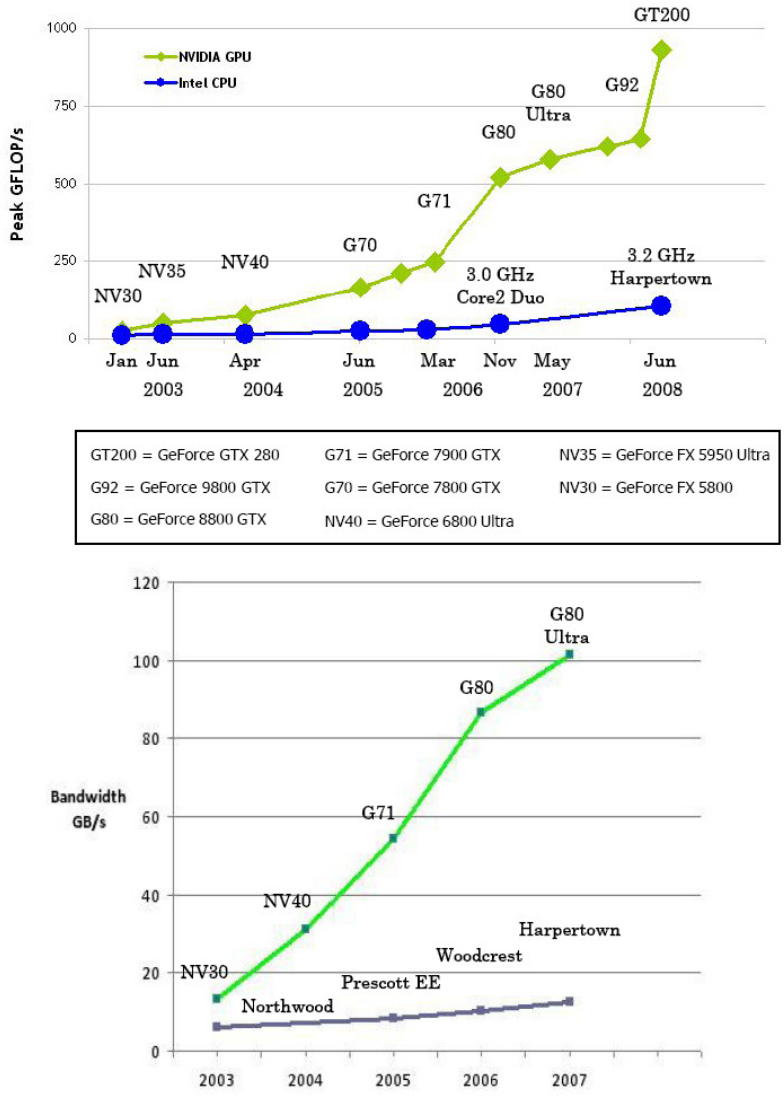


图 1-1. CPU 和 GPU 的每秒浮点运算次数和存储器带宽

CPU 和 GPU 之间浮点能力之所以存在这样的差异，原因就在于 GPU 专为计算密集型、高度并行化的计算而设计，上图显示的正是这种情况，因而，GPU 的设计将更多晶体管用于数据处理，而非数据缓存（caching）和流控制（flow control），如图 1-2 所示。

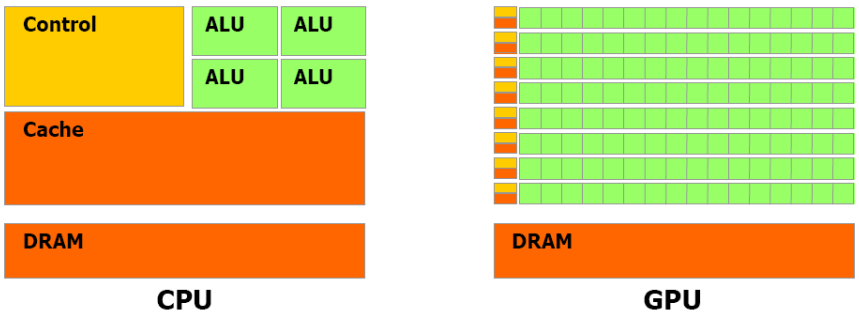


图 1-2. GPU 中的更多晶体管用于数据处理

更具体地说，GPU 专用于解决可表示为数据并行计算的问题——在许多数据元素上并行执行的程序，具有极高的计算密度（数学运算与存储器操作的比率）。由于所有数据元素都执行相同的程序，因此对精密流控制的要求不高；由于在许多数据元素上运行，且具有较高的计算密度，因而可通过计算来隐藏存储器访问延迟，而不必使用较大的数据缓存。

数据并行处理会将数据元素映射到并行处理线程。许多处理大型数据集的应用程序都可使用数据并行编程模型来加速计算。在 3D 渲染中，大量的像素和顶点集将映射到并行线程。类似地，图像和媒体处理应用程序（如渲染图像的后期处理、视频编码和解码、图像缩放、立体视觉和模式识别等）可将图像块和像素映射到并行处理线程。实际上，在图像渲染和处理领域之外的许多算法也都是通过数据并行处理加速的——从普通信号处理或物理仿真一直到计算金融或计算生物学。

CUDA 编程模型非常适合利用 GPU 的并行能力。最新一代的 NVIDIA GPU 基于 Tesla 架构（在附录 A 中可以查看所有支持 CUDA 的 GPU 列表），支持 CUDA 编程模型，可显著加速 CUDA 应用程序。

1.3 文档结构

本文档分为以下几个章节：

- 第 1 章是 CUDA 和 GPU 的简介。
- 第 2 章概述 CUDA 编程模型。
- 第 3 章介绍 GPU 实现。
- 第 4 章介绍 CUDA API 和运行时。
- 第 5 章提供如何实现最高性能的一些指南。
- 第 6 章通过一些简单的示例代码概况之前各章的内容。
- 附录 A 提供各种设备的技术规范。
- 附录 B 列举 CUDA 中支持的数学函数。
- 附录 C 列举 CUDA 中支持的原子函数。
- 附录 D 给出更多纹理拾取（texture fetching）的细节。

第 2 章　编程模型

CUDA 允许程序员定义被称为内核（kernel）的 C 语言函数，这是对 C 语言的一种扩展，在调用此类函数时，它将由 N 个不同的 CUDA 线程并行执行 N 次，这与普通的 C 语言函数只执行一次的方式不同。

在定义内核时，需要使用 __global__ 声明限定符（declaration specifier），使用一种全新的 <<<…>>> 语法指定每次调用的 CUDA 线程数：

```
// Kernel definition
__global__ void vecAdd(float* A, float* B, float* C)
{
}

int main()
{
    // Kernel invocation
    vecAdd<<<1, N>>>(A, B, C);
}
```

执行内核的每个线程都会被分配一个独特的线程 ID，可通过内置的 threadIdx 变量在内核中访问此 ID。以下示例代码将大小为 N 的向量 A 和向量 B 相加，并将结果存储在向量 C 中：

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

int main()
{
    // Kernel invocation
    vecAdd<<<1, N>>>(A, B, C);
}
```

执行 vecAdd( ) 的每个线程都会执行一次两两相加运算。

2.1　线程层次结构

为方便起见，我们将 threadIdx 设置为一个包含 3 个分量的向量，因而可使用一维、二维或三维索引标识线程，构成一维、二维或三维*线程块*（thread block）。这提供了一种自然的方法，可为一个域中的各元素调用计算，如向量、矩阵或字段。下面的示例代码将大小为 NxN 的矩阵 A 和矩阵 B 相加，并将结果存储在矩阵 C 中：

```
__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
    dim3 dimBlock(N, N);
    matAdd<<<1, dimBlock>>>(A, B, C);
}
```

线程的索引与其线程 ID 有着直接的对应关系：对于一维块来说，两者是相同的；对于大小为 (D_x, D_y) 的二维块来说，索引为 (x,y) 的线程的 ID 是 $(x + yD_x)$；对于大小为 (D_x, D_y, D_z) 的三维块来说，索引为 (x, y, z) 的线程的 ID 是 $(x + yD_x + zD_xD_y)$。

一个块内的线程可彼此协作，通过一些*共享存储器*（shared memory）来共享数据，并同步其执行来协调存储器访问。更具体地说，可以通过调用 __syncthreads() 内建函数（intrinsic function）在内核中指定同步点；__syncthreads() 起到屏障的作用，块中的所有线程都必须在这里等待处理。

为实现有效的协作，共享存储器被设计为靠近各处理器内核的低延迟存储器，这很像 L1 缓存，__syncthreads() 被设计为轻量级的，一个块中的所有线程都必须位于同一个处理器内核中。因而，一个处理器内核的有限存储器资源限制了每个块的线程数量。在 NVIDIA Tesla 架构中，一个线程块最多可以包含 512 个线程。

但一个内核函数可能由多个大小相同的线程块执行，因而线程总数应等于每个块的线程数乘以块的数量。这些块被组织为一个一维或二维线程块*网格*（grid），如图 2-1 所示。该网格的维度由 <<<…>>> 语法的第一个参数指定。网格内的每个块多可由一个一维或二维索引标识，可通过内置的 blockIdx 变量在内核中访问此索引。可以通过内置的 blockDim 变量在内核中访问线程块的维度。此时，之前的示例代码可修改为：

```
__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 dimBlock(16, 16);
    dim3 dimGrid((N + dimBlock.x - 1) / dimBlock.x,
                 (N + dimBlock.y - 1) / dimBlock.y);
    matAdd<<<dimGrid, dimBlock>>>(A, B, C);
}
```

我们随机选择了大小为 16x16 的线程块（即包含 256 个线程），此外创建了一个网格，它具有足够的块，可将每个线程处理一个矩阵元素，这与之前完全相同。

线程块需要独立执行：必须能够以任意顺序执行——能够并行或串行执行。这种独立性需求允许为任意数量的处理器内核安排线程块，从而使程序员能够编写出可伸缩的代码。

一个网格内的线程块数量通常是由所处理的数据大小限定的，而不是由系统中的处理器数量决定的，前者可以远远超过后者的数量。

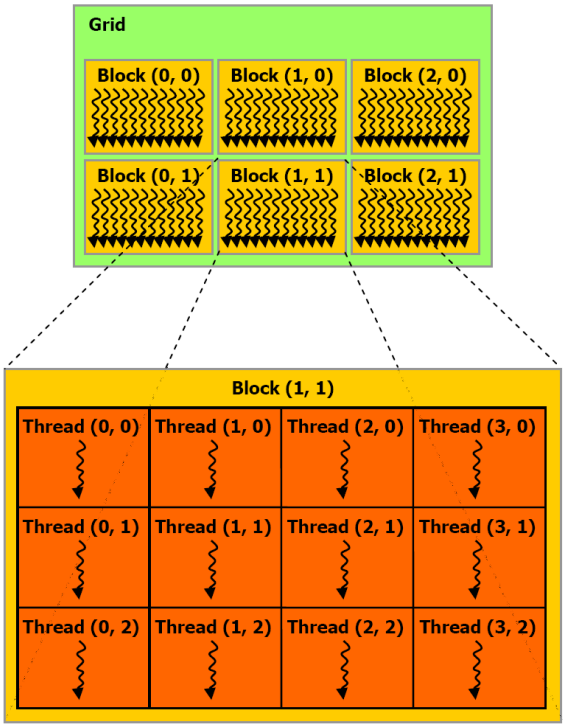


图 2-1. 线程块网格

2.2 存储器层次结构

CUDA 线程可在执行过程中访问多个存储器空间的数据，如图 2-2 所示。每个线程都有一个私有的本地存储器（local memory）。每个线程块都有一个共享存储器（shared memory），该存储器对于块内的所有线程都是可见的，并且与块具有相同的生命周期。最终，所有线程都可访问同一个全局存储器（global memory）。

此外还有两个只读的存储器空间，可由所有线程访问，这两个空间是常量存储器（constant memory）空间和纹理存储器（texture memory）空间。全局、常量和纹理存储器空间经过优化，适于不同的存储器用途（参见第 5.1.2.1、5.1.2.3 和 5.1.2.4）。纹理存储器也为某些特殊的数据格式提供了不同的寻址模式以及数据过滤（参见第 4.3.4）。

对于同一个应用程序启动的内核而言，全局、常量和纹理存储器空间都是持久的。

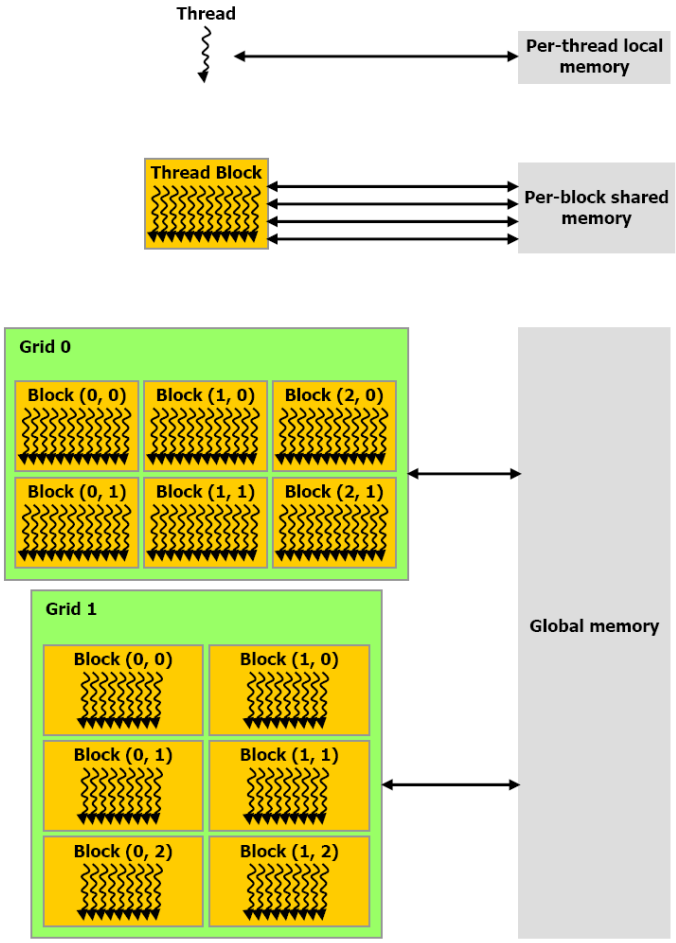
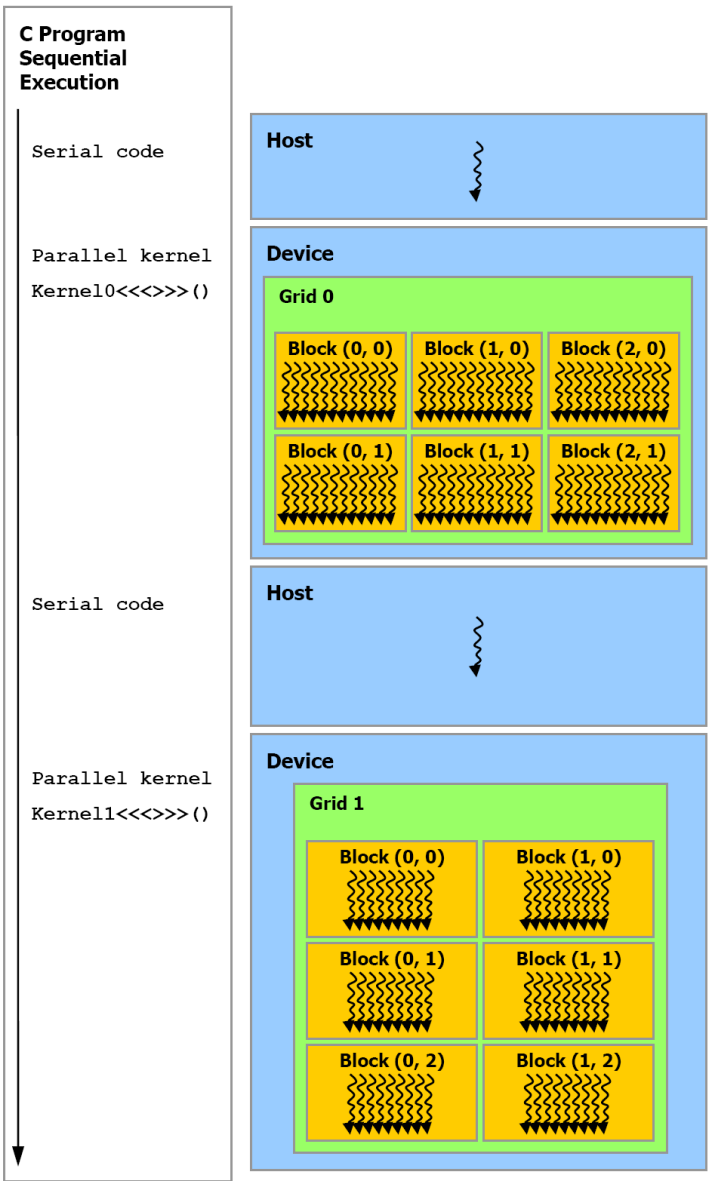


图 2-2. 存储器层次结构

2.3 宿主和设备

如图 2-3 所示，CUDA 假设 CUDA 线程可在物理上独立的*设备*（device）上执行，此类设备作为运行 C 语言程序的*宿主*（host）的协处理器存在。比如，内核在 GPU 上执行，而 C 语言程序的其他部分在 CPU 上执行，就是一例。

此外，CUDA 还假设宿主和设备均维护自己的 DRAM，分别称为宿主存储器（host memory）和设备存储器（device memory）。因而，一个程序通过调用 CUDA 运行时（CUDA runtime）来管理对内核可见的全局、常量和纹理存储器空间（详见第 4 章）。这包括设备存储器分配和取消分配，还包括宿主和设备存储器之间的数据传输。



串行代码在宿主上执行，而并行代码在设备上执行。

图 2-3. 异构编程

2.4 软件栈

CUDA *软件栈*（software stack）包含多个层，如图 2-4 所示：设备驱动程序（device driver）、应用程序编程接口（API）及其运行时、两个较高级别的通用数学库，即 CUFFT 和 CUBLAS，这两个数学库会在其他文档中介绍。

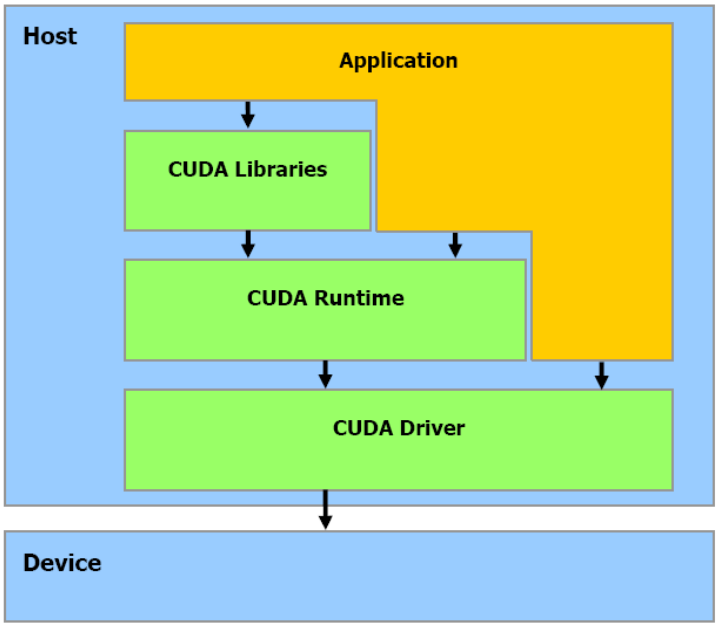


图 2-4. CUDA 软件栈

2.5 计算能力

一个设备的*计算能力*（compute capability）由主要修订号和次要修订号定义。

具有相同主要修订号的设备属于相同的核心架构。附录 A 中列举的设备均为计算能力是 1.x 的设备（其主要修订号为 1）。

次要修订号对应于核心架构的增量式改进，可能包含新特性。

附录 A 提供了各种计算能力的技术规范。

第 3 章 GPU 实现

NVIDIA 于 2006 年 11 月引入的 Tesla 统一图形和计算架构扩展了 GPU，使 GPU 不再局限于图形领域，其强大的多线程处理器阵列已经成为高效的统一平台——同时适用于图形和通用并行计算应用程序。通过扩展处理器和存储器的数量，Tesla 架构覆盖了市场上全线产品，从高性能发烧级 GeForce GTX 280 GPU 和专业 Quadr 与 Tesla 计算产品，一直到多种主流经济型 GeForce GPU（在附录 A 中可查看所有支持 CUDA 的 GPU 的列表）。其计算特性支持利用 CUDA 在 C 语言中直观地编写 GPU 核心程序。Tesla 架构具有在笔记本电脑、台式机、工作站和服务器上的广泛可用性，配以 C 语言编程能力和 CUDA 软件，使这种架构成为最优秀的超级计算平台。

这一章介绍 CUDA 编程模型与 Tesla 架构的对应关系。

3.1 具有片上共享存储器的一组 SIMT 多处理器

Tesla 架构的构建以一个可伸缩的多线程*流式多处理器*（Streaming Multiprocessors, SM）阵列为中心。当宿主 CPU 上的 CUDA 程序调用内核网格时，网格的块将被枚举并分发到具有可用执行能力的多处理器上。一个线程块的线程在一个多处理器上并发执行。在线程块终止时，将在空闲多处理器上启动新块。

多处理器包含 8 个*标量处理器*（Scalar Processor, SP）核心、两个用于超越函数（transcendental）的特殊功能单元（special function unit）、一个多线程指令单元以及片上（on-chip）共享存储器。多处理器会在硬件中创建、管理和执行并发线程，而调度开销保持为 0。它可通过一条内部指令实现 __syncthreads() 栅障同步。快速的栅障同步与轻量级线程创建和零开销的线程调度相结合，有效地为细粒度并行化提供了支持，举例来说，您可以为各数据元素（如图像中的一个像素、语音中的一个语音元素、基于网格的计算中的一个单元）分配一个线程，从而对问题进行细粒度分解。

为了管理运行各种不同程序的数百个线程，多处理器利用了一种称为 SIMT（single-instruction, multiple-thread, 单指令、多线程）的新架构。多处理器会将各线程映射到一个标量处理器核心，各标量线程使用自己的指令地址和寄存器状态独立执行。多处理器 SIMT 单元以 32 个并行线程为一组来创建、管理、调度和执行线程，这样的线程组称为 warp 块。（此术语源于第一种并行线程技术 weaving。半 warp 块可以是一个 warp 块的第一半或第二半。）构成 SIMT warp 块的各个线程在同一个程序地址一起启动，但也可随意分支、独立执行。

为一个多处理器指定了一个或多个要执行的线程块时，它会将其分成 warp 块，并由 SIMT 单元进行调度。将块分割为 warp 块的方法总是相同的，每个 warp 块都包含连续的线程，递增线程 ID，第一个 warp 块中包含线程 0。第 2.1 节介绍了线程 ID 与块中的线程索引之间的关系。

每次准备发出一条指令时，SIMT 单元都会选择一个可待使用的 warp 块，并将待发出的指令发送到该 warp 块的活动线程。Warp 块每次执行一条通用指令，因此在 warp 块的全部 32 个线程均有明确的执行路径时，可达到最高效率。如果一个 warp 块的线程通过独立于数据的条件分支而分散，warp 块将连续执行所使用的各分支路径，而禁用未在此路径上的线程，完成所有路径时，线程重新汇聚到同一执行路径下。分支仅在 warp 块内出现，不同的 warp 块总是独立执行的——无论它们执行的是通用的代码路径还是彼此无关的代码路径。

SIMT 架构类似于 SIMD（单指令、多数据）向量组织方法，共同之处是使用单指令来控制多个处理元素。一项主要差别在于 SIMD 向量组织方法会向软件公开 SIMD 宽度（SIMD width），而 SIMT 指令指定单一线程的执行和分支行为。与 SIMD 向量机不同，SIMT 允许程序员为独立、标量线程编写线程级的并行代码，还允许为协同线程编写数据并行代码。为了确保正确性，程序员可忽略 SIMT 行为，但通过尽量减少一个 warp 块内的线程分支，即可实现显著的性能提升。在实践中，这与传统代码中的 cache 线（cache line）作用相似：在以正确性为目标进行设计时，可忽略 cache 线的大小，但如果以峰值性能为目标进行设计，在代码结构中就必须考虑其大小。另一方面，向量架构要求软件操作存储器合并，并手动管理分支。

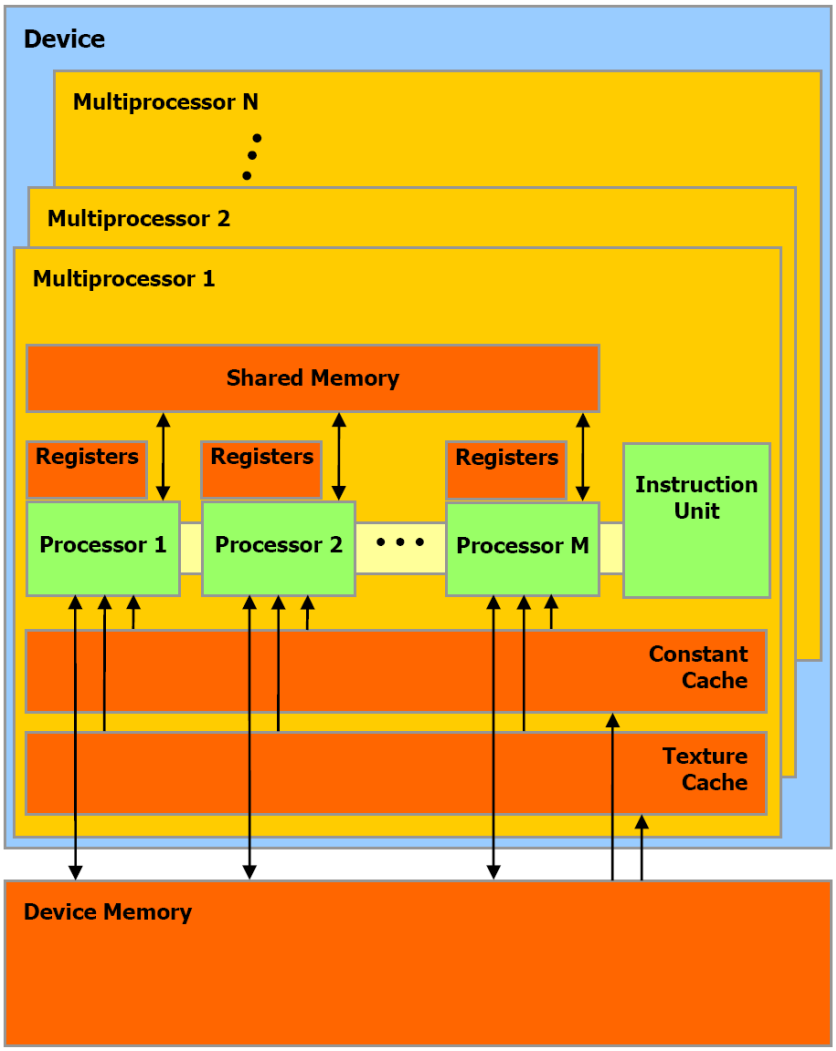
如图 3-1 所示，每个多处理器都有一个属于以下四种类型之一的片上存储器：

- 每个处理器上有一组本地 32 位寄存器；
- 并行数据缓存或共享存储器，由所有标量处理器核心共享，共享存储器空间就位于此处；
- 只读*常量缓存*（constant cache），由所有标量处理器核心共享，可加速从常量存储器空间进行的读取操作（这是设备存储器的一个只读区域）；
- 一个只读*纹理缓存*（texture cache），由所有标量处理器核心共享，加速从纹理存储器空间进行的读取操作（这是设备存储器的一个只读区域），每个多处理器都会通过实现不同寻址模型和数据过滤的*纹理单元*（texture unit）访问纹理缓存，相关内容请参见第 4.3.4 节。

本地和全局存储器空间是设备存储器的读/写区域，无缓存。

一个多处理器一次可处理的块数量取决于给定的内核中，每个线程需要多少个寄存器、每个块需要多少共享存储器，这是因为多处理器的寄存器和共享存储器被分配给一批块的所有线程。如果没有足够的寄存器或共享存储器可供多处理器用于处理至少一个块，内核会无法启动。一个多处理器可并发执行最多 8 个线程块。

如果 warp 块执行的非原子指令为 warp 块的多个线程写入全局或共享存储器中的同一位置，针对此位置的串行化写入操作的数量和这些写入操作所发生的顺序将无法确定，但其中一项操作必将成功。如果 warp 块执行原子指令来为 warp 块的多个线程读取、修改和写入全局存储器中的同一位置，则针对该位置的每一项读取、修改或写入操作都将发生，且均为串行化操作，但这些操作所发生的顺序无法确定。



具有片上共享存储器的一组 SIMT 多处理器

图 3-1. 硬件模型

3.2 多个设备

若要在多 GPU 系统上运行，且应用程序将多个 GPU 作为 CUDA 设备使用，则这些 GPU 必须具有相同的类型。但如果系统采用的是 SLI 模式，则仅有一个 GPU 可用作 CUDA 设备，因为所有 GPU 都将在驱动程序栈的最低级别融合。要使 CUDA 能够将各 GPU 视为独立设备，需要在 CUDA 的控制面板内关闭 SLI 模式。

3.3 模式切换

GPU 将部分 DRAM 存储器专门用于处理所谓的*主表面*（primary surface），它用于刷新用户所见的显示设备。当用户通过更改显示器的分辨率或位深度（使用 NVIDIA 控制面板或 Windows 的显示控制面板）发起模式切换时，主表面所需的存储器数量会随之改变。例如，如果用户将显示器的分辨率从 1280x1024x32 位更改为 1600x1200x32 位，系统必须为主表面分配 7.68 MB 的存储器，而不是 5.24 MB。（使用防锯齿设置运行的全屏图形应用程序可能需要为主表面分配更多显示存储器。）在 Windows 上，其他事件也可能会启动显示模式切换，包括启动全屏 DirectX 应用程序、按 Alt+Tab 键从全屏 DirectX 应用程序中切换出来或者按 Ctrl+Alt+Del 键锁定计算机。

如果模式切换增加了主表面所需的存储器数量，系统可能就必须挪用分配给 CUDA 应用程序的存储器，从而导致此类应用程序崩溃。

第 4 章 应用程序编程接口

4.1 C 编程语言的扩展

CUDA 编程接口的目标是为熟悉 C 编程语言的用户提供相对简单的途径，使之可轻松编写由设备执行的程序。

它包含：

- C 语言的最小扩展集，如 4.2 节所述，这允许程序员使源代码的某些部分可在设备上执行；
- 一个运行时库，可分割为：
  - 一个宿主（host）组件，如 4.5 节所述，运行在宿主上，提供函数来通过宿主控制和访问一个或多个计算设备；
  - 一个设备组件，如 4.4 节所述，运行在设备上，提供特定于设备的函数；
  - 一个通用组件，如 4.3 节所述，提供内置向量类型和 C 标准库的一个子集，宿主和设备代码都将支持此子集。

有必要强调，C 标准库中支持在设备上运行的函数只有通用运行时组件所提供的函数。

4.2 语言扩展

对 C 编程语言的扩展共有四重：

- 函数类型限定符（function type qualifier），指定函数是在宿主上还是设备上执行，以及函数是可通过宿主还是可通过设备调用（参见第 4.2.1 节）；
- 变量类型限定符（variable type qualifier），指定一个变量在设备上的存储器位置（参见第 4.2.2 节）；
- 一条新指令，指定如何通过宿主在设备上执行内核（参见第 4.2.3 节）；
- 四个内置变量，指定网格和块维度以及块和线程索引（参见第 4.2.4 节）。

包含这些扩展的所有源文件都必须使用 CUDA 编译器 nvcc 进行编译，4.2.5 节简单介绍了相关内容。关于 nvcc 的具体介绍将在其他文档中提供。

这些扩展均具有一些限制，下面几个小节将分别加以介绍。如果违背了这些限制，nvcc 将发出错误或警报信息，但有些违规情况无法检测到。

4.2.1 函数类型限定符

4.2.1.1 __device__

使用 __device__ 限定符声明的函数具有以下特征：

- 在设备上执行；
- 仅可通过设备调用。

4.2.1.2 __global__

使用 __global__ 限定符可将函数声明为内核。此类函数：

- 在设备上执行；
- 仅可通过宿主调用。

4.2.1.3 __host__

使用 __host__ 限定符声明的函数具有以下特征：

- 在宿主上执行；
- 仅可通过宿主调用。

仅使用 __host__ 限定符声明函数等同于不使用 __host__、__device__ 或 __global__ 限定符声明函数，这两种情况下，函数都将仅为宿主进行编译。

但 __host__ 限定符也可与 __device__ 限定符一起使用，此时函数将为宿主和设备进行编译。

4.2.1.4 限制

__device__ 和 __global__ 函数不支持递归。
__device__ 和 __global__ 函数的函数体内无法声明静态变量（static variable）。
__device__ 和 __global__ 函数的参数不得为变量。
__device__ 函数的地址无法获取，但支持 __global__ 函数的函数指针。
__global__ 和 __host__ 限定符无法一起使用。
__global__ 函数的返回类型必须为空（void）。
对 __global__ 函数的任何调用都必须按第 4.2.3 节介绍的方法指定其执行配置。
__global__ 函数的调用是异步的，也就是说它会在设备执行完成之前返回。
__global__ 函数参数将同时通过共享存储器传递给设备，且限制为 256 字节。

4.2.2 变量类型限定符

4.2.2.1 __device__

__device__ 限定符声明位于设备上的变量。

在接下来的三节中介绍的其他类型限定符中，最多只能有一种可与 __device__ 限定符一起使用，以更具体地指定变量属于哪个存储器空间。如果未出现其他任何限定符，则变量具有以下特征：

- 位于全局存储器空间中；
- 与应用程序具有相同的生命周期；
- 可通过网格内的所有线程访问，也可通过运行时库从宿主访问。

4.2.2.2 __constant__

__constant__ 限定符可选择与 __device__ 限定符一起使用，所声明的变量具有以下特征：

- 位于常量存储器空间中；
- 与应用程序具有相同的生命周期；

- 可通过网格内的所有线程访问，也可通过运行时库从宿主访问。

4.2.2.3 __shared__

__shared__ 限定符可选择与 __device__ 限定符一起使用，所声明的变量具有以下特征：

- 位于线程块的共享存储器空间中；
- 与块具有相同的生命周期；
- 尽可通过块内的所有线程访问。

只有在 __syncthreads()（参见第 4.4.2 节）的执行写入之后，才能保证共享变量对其他线程可见。除非变量被声明为 volatile 变量，否则只要之前的语句完成，编译器即可随意优化共享存储器的读写操作。

将共享存储器中的变量声明为外部数组时，例如：

```
extern __shared__ float shared[];
```

数组的大小将在启动时确定（参见第 4.2.3 节）。所有变量均以这种形式声明，在存储器中的同一地址开始，因此数组中的变量布局必须通过偏移显式管理。例如，如果一名用户希望在动态分配的共享存储器内获得与以下代码对应的内容：

```
short array0[128];
float array1[64];
int array2[256];
```

则应通过以下方法声明和初始化数组：

```
extern __shared__ char array[];
__device__ void func() // __device__ or __global__ function
{
    short* array0 = (short*)array;
     float* array1 = (float*)&array0[128];
     int* array2 = (int*)&array1[64];
}
```

4.2.2.4 限制

这些限定符不允许被用于在宿主上执行的函数内的 struct 和 union 成员的形参和局部变量。

__shared__ 和 __constant__ 变量具有隐含的静态存储（implied static storage）。

__device__、__shared__ 和 __constant__ 变量无法使用 extern 关键字定义为外部变量。

__device__ 和 __constant__ 变量仅允许在文件作用域内使用。

不可为设备或从设备指派 __constant__ 变量，仅可通过宿主运行时函数从宿主指派（参见第 4.5.2.3 节和第 4.5.3.6 节）。

shared 变量的声明中不可包含初始化。

在设备代码中声明、不带任何限定符的自动变量通常位于寄存器中。但在某些情况下，编译器可能选择将其置于本地存储器中。如果使用占用了过多寄存器空间的大型结构或数组，或者编译器无法确定其是否使用固定数量索引的数组，则往往会出现这种情况。检查 ptx 汇编代码（通过使用 –ptx 或 –keep 选项编译获得）即可在初次编译过程中确定一个变量是否位于本地存储器中，因为它将使用 .local 助记符声明，可使用 ld.local 和 st.local 助记符访问。如果不是这样，在后续编译阶段仍能确定是否占用了目标架构的过多寄存器空间。可通过使用 --ptxas- options =-v 选项编译来进行检查，这将报告本地存储器的使用情况（lmem）。

只要编译器能够确定在设备上执行的代码中的指针指向的是共享存储器空间还是全局存储器空间，此类指针即受支持，否则将仅限于指向在全局存储器空间中分配或声明的存储器。

如果取消在宿主上执行的代码中全局或共享存储器指针，或者在设备上执行的代码中宿主存储器指针的引用，将导致不确定的行为，往往会出现分区错误（segmentation fault）和应用程序终止。

通过获取 __device__、__shared__ 或 __constant__ 变量的地址而获得的地址仅可在设备代码中使用。通过 cudaGetSymbolAddress() （参见第 4.5.23 节）获取的 __device__ 或 __constant__ 变量的地址仅可在宿主代码中使用。

4.2.3 执行配置

对 __global__ 函数的任何调用都必须指定该调用的*执行配置*（execution configuration）。

执行配置定义将用于在该设备上执行函数的网格和块的维度，以及相关的流（有关流的内容将在第 4.5.1.5 节介绍）。可通过在函数名称和括号参数列表之间插入 <<<Dg, Db, Ns, s>>> 形式的表达式来指定，其中：

- Dg 的类型为 dim3（参见第 4.3.1.2 节），指定网格的维度和大小，Dg.x * Dg.y 等于所启动的块数量，Dg.z 无用；
- Db 的类型为 dim3（参见第 4.3.1.2 节），指定各块的维度和大小，Db.x * Db.y * Db.z 等于各块的线程数量；
- Ns 的类型为 size_t，指定各块为此调用动态分配的共享存储器（除静态分配的存储器之外），这些动态分配的存储器可供声明为外部数组的其他任何变量使用（参见第 4.2.2.3 节），Ns 是一个可选参数，默认值为 0；
- S 的类型为 cudaStream_t，指定相关流；S 是一个可选参数，默认值为 0。

举例来说，一个函数的声明如下：

```
__global__ void Func(float* parameter);
```

必须通过如下方法来调用此函数：

```
Func<<< Dg, Db, Ns >>>(parameter);
```

执行配置的参数将在实际函数参数之前被赋值，与函数参数一起通过共享存储器同时传递给设备。

如果 Dg 或 Db 大于设备允许的最大大小（参见附录 A.1.1），或 Ns 大于设备上可用的共享存储器最大值，或者小于静态分配、函数参数和执行配置所需的共享存储器数量，则函数将失败。

4.2.4 内置变量

4.2.4.1 gridDim

此变量的类型为 dim3（参见第 4.3.1.2 节），包含网格的维度。

4.2.4.2 blockIdx

此变量的类型为 uint3（参见第 4.3.1.1 节），包含网格内的块索引。

4.2.4.3 blockDim

此变量的类型为 dim3（参见第 4.3.1.2 节），包含块的维度。

4.2.4.4 threadIdx

此变量的类型为 uint3（参见第 4.3.1.1 节），包含块内的线程索引。

4.2.4.5 warpSize

此变量的类型为 int，包含以线程为单位的 warp 块大小。

4.2.4.6 限制

- 不允许试图获取任何内置变量的地址。
- 不允许为任何内置变量赋值。

4.2.5 使用 NVCC 进行编译

nvcc 是一种可简化 CUDA 代码编译过程的编译器驱动程序：它提供了简单、熟悉的命令行选项，通过调用实现不同编译阶段的工具集合来执行它们。

nvcc 的基本工作流在于将设备代码与宿主代码分离开来，并将设备代码编译为二进制形式或 cubin 对象。所生成的宿主代码将作为需要使用其他工具编译的 C 代码输出，或通过在最后一个编译阶段中调用宿主编译器直接作为对象代码输出。

应用程序可忽略所生成的宿主代码，使用 CUDA 驱动程序 API 在设备上加载并执行 cubin 对象，也可链接到所生成的宿主代码，其中包含 cubin 对象，其形式为全局初始化数据数组，包含将第 4.2.3 节所述执行配置语法转换为必要的 CUDA 运行启动代码的转换，目的在于加载和启动编译后的各内核（参见第 4.5.2 节）。

编译器的前端根据 C++ 语法规则处理 CUDA 源文件。宿主代码支持完整的 C++ 语法。但设备代码仅支持 C++ 的 C 子集，类、继承、基本块内的变量声明等 C++ 特殊特性不受支持。由于使用了 C++ 语法规则，因此若未经过强制类型转换，无法将空指针（例如 malloc() 所返回的空指针）指派给非空指针。

关于 nvcc 工作流和命令选项的详细说明将在其他文档中提供。

nvcc 引入了两个编译器指令，下面几节将加以介绍。

4.2.5.1 __noinline__

默认情况下，__device__ 函数总是内联的（inlined）。__noinline__ 函数限定符可用于指示编译器尽可能不要内联该函数。函数体必须位于所调用的同一个文件内。

如果函数具有指针参数或者具有较大的参数列表，则编译器不会遵从 __noinline__ 限定符。

4.2.5.2 #pragma unroll

默认情况下，编译器将展开具有已知行程计数（trip count）的小循环。#pragma unroll 指令可用于控制任何给定循环的展开操作。它必须紧接于循环之前，而且仅应用于该循环。可选择在其后接一个数字，指定必须展开多少次循环。

例如，在下面的代码示例中：

```
#pragma unroll 5
for (int i = 0; i < n; ++i)
```

循环将展开 5 次。程序员需要负责确保展开操作不会影响程序的正确性（在上面的示例中，如果 n 小于 5，则程序的正确性将受到影响）。

#pragma unroll 1 将阻止编译器展开一个循环。

如果在 #pragma unroll 后未指定任何数据，如果其行程计数为常数，则该循环将完全展开，否则将不会展

开。

4.3 通用运行时组件

宿主和设备函数均可使用通用运行时组件。

4.3.1 内置向量类型

4.3.1.1 char1、uchar1、char2、uchar2、char3、uchar3、char4、uchar4、short1、ushort1、short2、ushort2、short3、ushort3、short4、ushort4、int1、uint1、int2、uint2、int3、uint3、int4、uint4、long1、ulong1、long2、ulong2、long3、ulong3、long4、ulong4、float1、float2、float3、float4、double2

这些向量类型继承自基本整形和浮点类型。它们均为结构体，第 1、2、3、4 个分量分别可通过字段 x、y、z 和 w 访问。它们均附带形式为 make_<type name> 的构造函数，示例如下：

```
int2 make_int2(int x, int y);
```

这将创建一个类型为 int2 的向量，值为 (x, y)。

4.3.1.2 dim3 类型

此类型是一种整形向量类型，基于用于指定维度的 uint3。在定义类型为 dim3 的变量时，未指定的任何分量都将初始化为 1。

4.3.2 数学函数

B.1 节包含了当前支持的 C/C++ 标准库数学函数的完整列表，还分别给出了在设备上执行时的误差范围。在宿主代码中执行时，给定函数将在可用的前提下使用 C 运行时实现。

4.3.3 计时函数

```
clock_t clock();
```

在设备代码中执行时，返回随每一次时钟周期而递增的每个多处理器计数器的值。在内核启动和结束时对此计数器取样，确定两次取样的差别，然后为每个线程记录下结果，这为各线程提供一种度量方法，可度量设备为了完全执行线程而占用的时钟周期数，但不是设备在执行线程指令时而实际使用的时钟周期数。前一个数字要比后一个数字大得多，因为线程是分时的。

4.3.4 纹理类型

CUDA 支持 GPU 用于图形的纹理硬件子集，使之可访问纹理存储器。从纹理存储器而非全局存储器读取数据可带来多方面的性能收益，请参见第 5.4 节。

内核使用称为*纹理拾取*（texture fetch）的设备函数读取纹理存储器，请参见第 4.4.3 节。纹理拾取的第一个参数指定称为*纹理参考*（texture reference）的对象。

纹理参考定义拾取哪部分的纹理存储器。必须通过宿主运行时函数（参见第 4.5.2.6 和第 4.5.3.9 节）将其绑定到存储器的某些区域（即纹理），之后才能供内核使用。多个不同的纹理参考可绑定到同一个纹理，也可绑定到在存储器中存在重叠的纹理。

纹理参考有一些属性。其中之一就是其维度，指定纹理是使用一个*纹理坐标*（texture coordinate）将纹理作为一维数组寻址、使用两个纹理坐标作为二维数组寻址，还是使用三个纹理坐标作为三维数组寻址。数组的元素称为 texel，即“texture elements（纹理元素）”的简写。

其他属性定义纹理拾取的输入和输出数据类型，并指定如何介绍输入坐标、应进行怎样的处理。

4.3.4.1 纹理参考声明

纹理参考的部分属性是不变的，在编译时必须为已知，这些属性是在声明纹理参考时指定的。纹理参考在文件作用域内声明，形式为 texture 类型的变量：

```
texture<Type, Dim, ReadMode> texRef;
```

其中：

- Type 指定拾取纹理时所返回的数据类型；Type 仅限于基本整型、单精度浮点类型和第 4.3.1.1 节定义的 1 分量、2 分量 和 4 分量向量类型；
- Dim 指定纹理参考的维度，其值为 1、2 或 3；Dim 是一个可选的参数，默认值为 1；
- ReadMode 等于 cudaReadModeNormalizedFloat 或 cudaReadModeElementType；如果是 cudaReadModeNormalizedFloat，且 Type 为 16 位或 8 位整型类型，则值将作为浮点类型返回，对于所有整型数据而言，无符号整型将映射为 [0.0, 1.0]，有符号整型将映射为 [-1.0, 1.0]，例如，一个值为 0xff 的无符号 8 位纹理元素将被读取为 1；如果是 cudaReadModeElementType，则不执行任何转换操作；ReadMode 是一个可选的参数，默认值为 cudaReadModeElementType。

4.3.4.2 运行时纹理参考属性

纹理参考的其他属性是可变的，可通过宿主运行时在运行时更改（第 4.5.2.6 节介绍了运行时 API，第 4.5.3.9 介绍了驱动程序 API）。它们指定纹理坐标是否为归一化（normalized）的，以及寻址模式和纹理过滤，下面将介绍相关内容。

默认情况下，使用 [0, N) 范围内的浮点坐标引用纹理，其中的 N 是纹理在对应于坐标的维度中的大小。例如，有一个大小为 64x32 的纹理，在 x 和 y 维度引用此纹理时坐标分别处于 [0, 63] 和 [0, 31] 范围内。归一化的纹理坐标将在 [0.0, 1.0) 的范围内指定，而非 [0, N)，因此在归一化的坐标内，同一 64x32 纹理的寻址范围在 x 和 y 维度均为 [0, 1)。一般情况下，纹理坐标与纹理大小无关，规范化的纹理坐标通常足以满足一些应用程序的需求。

寻址模式定义在纹理坐标超出范围时将出现怎样的情况。在使用非归一化纹理坐标时，超出 [0, N) 范围的纹理坐标将被调整（clamp）：小于 0 的值被设置为 0，大于或等于 N 的值被设置为 N-1。在使用归一化纹理坐标时，默认寻址模式也是调整坐标：小于 0.0 或大于 1.0 的值将被调整到范围 [0.0, 1.0) 内。对于归一化坐标，也可指定“wrap”的寻址模式。Wrap 寻址往往在纹理包含周期信号时使用。它仅使用纹理坐标的一部分，例如，1.25 被视为 0.25，-1.25 被视为 0.75.

线性纹理过滤只能对配置为返回浮点数据的纹理进行。这将在相邻 texel 间执行低精度插值。在启用时，

位于纹理拾取位置周围的 texel 将被读取，纹理拾取的返回值将根据纹理坐标在 texel 间的位置进行插值。对于一维纹理执行简单的线性插值，而对于二维纹理则执行双线性插值。

附录 D 提供了关于纹理拾取的更多细节。

4.3.4.3 来自线性存储器的纹理与来自 CUDA 数组的纹理

纹理可以是线性存储器或 CUDA 数组的任意区域（参见第 4.5.1.2 节）。

在线性存储器内分配的纹理：

- 维度仅能为 1；
- 不支持纹理过滤；
- 仅可使用非归一化整型纹理坐标寻址；
- 不支持多种寻址模式：超出范围的纹理访问将返回零。

硬件会对纹理基址实施对齐要求。为了抽象这种来自程序员的对齐要求，绑定设备存储器上的纹理参考的函数将传回一个字节偏移，必须将其应用到纹理拾取，之后才能读取所需的存储器。CUDA 分配例程返回的基址指针符合这种对齐限制，因此应用程序可通过向 **`cudaBindTexture()`**/**`cuTexRefSetAddress()`** 传递所分配的指针来完全避免偏移。

4.4 设备运行时组件

设备运行时组件仅可用于设备函数。

4.4.1 数学函数

对于 B.1 节介绍的部分函数而言，设备运行时组件中存在准确性略低而速度更快的版本；其名称相同，但带有一个__前缀（如__sinf(x)）。B.2 节列举了这些内建（intrinsic function）函数，还列举了它们的对应误差范围。

编译器有一个 (-use_fast_math) 选项，用于强制要求所有函数编译其准确性略低的版本（如果存在）。

4.4.2 同步函数

```
void __syncthreads();
```

同步块中的所有线程。一旦所有线程均达到此同步点，才将继续执行后续代码。

__syncthreads() 用于协调同一个块内的线程间通信。在一个块内的某些线程访问共享或全局存储器中的相同地址时，部分访问操作可能存在写入后读取、读取后写入或写入后写入之类的风险。可通过在这些访问操作间同步线程来避免这些数据风险。

__syncthreads() 允许在条件代码中使用，但仅当条件估值在整个线程块中都相同时才允许使用，否则代码执行将有可能挂起，或者出现意料之外的副作用。

4.4.3 纹理函数

4.4.3.1 来自线性存储器的纹理

对于来自线性存储器的纹理，通过 tex1Dfetch() 系列函数访问纹理，示例如下：

```
template<class Type>
Type tex1Dfetch(
   texture<Type, 1, cudaReadModeElementType> texRef,
   int x);
float tex1Dfetch(
   texture<unsigned char, 1, cudaReadModeNormalizedFloat> texRef,
   int x);
float tex1Dfetch(
   texture<signed char, 1, cudaReadModeNormalizedFloat> texRef,
   int x);
float tex1Dfetch(
   texture<unsigned short, 1, cudaReadModeNormalizedFloat> texRef,
   int x);
float tex1Dfetch(
   texture<signed short, 1, cudaReadModeNormalizedFloat> texRef,
   int x);
```

这些函数会使用纹理坐标 x 拾取绑定到纹理参考 texRef 的线性存储器区域。不支持纹理过滤和寻址模式。对于整型来说，这些函数可选择将整型转变为单精度浮点类型。

除了上述函数以外，还支持 2 元组和 4 元组，示例如下：

```
float4 tex1Dfetch(
   texture<uchar4, 1, cudaReadModeNormalizedFloat> texRef,
   int x);
```

以上示例将使用纹理坐标 x 拾取绑定到纹理参考 texRef 的线性存储器。

4.4.3.2 来自 CUDA 数组的纹理

对于来自 CUDA 数组的纹理，可通过 tex1D()、tex2D()、tex3D() 访问纹理：

```
template<class Type, enum cudaTextureReadMode readMode>

Type tex1D(texture<Type, 1, readMode> texRef,
           float x);
template<class Type, enum cudaTextureReadMode readMode>
Type tex2D(texture<Type, 2, readMode> texRef,
           float x, float y);
template<class Type, enum cudaTextureReadMode readMode>
Type tex3D(texture<Type, 3, readMode> texRef,
           float x, float y, float z);
```

这些函数将使用纹理坐标 x、y 和 z 拾取绑定到纹理参考 texRef 的 CUDA 数组。纹理参考的不变（编译时）和可变（运行时）属性相互结合，共同确定坐标的解释方式、在纹理拾取过程中发生的处理以及纹理拾取所提供的返回值（参见第 4.3.4.1 和第 4.3.4.2 节）。

4.4.4 原子函数

原子函数（atomic function）对位于全局或共享存储器内的一个 32 位或 64 位字执行读取-修改-写入原子操作。例如，atomicAdd() 将在全局或共享存储器内的某个地址读取 32 位字，将其与一个整型相加，并将结果写回同一地址。之所以说这样的操作是原子的，是因为它可在不干扰其他线程的前提下执行。换句话说，在操作完成前，其他任何线程都无法访问此地址。

附录 C 列举了受支持的所有原子函数。如附录所述，并非所有设备都支持这些函数。具体来说，计算能

力为 1.0 的设备不支持任何原子函数。

原子操作仅适用于有符号和无符号整型（但 atomicExch() 是一个例外情况，它支持单精度浮点数）。

4.4.5 Warp vote 函数

只有计算能力为 1.2 或更高的设备支持 Warp vote 函数。

```
int __all(int predicate);
```

为 warp 块内的所有线程计算 predicate，当且仅当所有线程的 predicate 均非零时返回非零值。

```
int __any(int predicate);
```

为 warp 块内的所有线程计算 predicate，当且仅当任意线程的 predicate 非零时返回非零值。

4.5 宿主运行时组件

只有宿主函数才能使用宿主运行时组件。

它提供了具有以下功能的函数：

- 设备管理；
- 上下文管理；
- 存储器管理；
- 代码模块管理；
- 执行控制；
- 纹理参考管理；
- 与 OpenGL 和 Direct3D 的互操作性。

它包含两个 API：

- 一个称为 CUDA 驱动程序 API 的低级 API；
- 一个称为 CUDA 运行时 API 的高级 API，它是在 CUDA 驱动程序 API 的基础之上实现的。

这两个 API 是互斥的：一个应用程序仅能使用其中之一。

CUDA 运行时提供了隐式初始化、上下文管理和模块管理，从而简化了设备代码管理。nvcc 生成的 C 宿主代码基于 CUDA 运行时（请参见第 4.2.5 节），因此链接到此代码的应用程序必须使用 CUDA 运行时 API。

相反，CUDA 驱动程序 API 需要的代码数量更多，编程和调试更加困难，但提供了更出色的控制级别，此外还具有独立于语言的特点，因为它仅处理 cubin 对象（请参见第 4.2.5 节）。具体来说，使用 CUDA 驱动程序 API 配置和启动内核的难度更大，因为执行配置和内核参数必须通过显式函数调用来指定，而不能利用第 4.2.3 节介绍的执行配置语法。此外，设备模拟（请参见第 4.5.2.9 节）不适用于 CUDA 驱动程序 API。

CUDA 驱动程序 API 是通过 nvcuda 动态库提供的，其所有入口点都带有 cu 前缀。

CUDA 运行时 API 是通过 cudart 动态库提供的，其所有入口点都带有 cuda 前缀。

4.5.1 一般概念

4.5.1.1 设备

两种 API 都提供了枚举系统上可用设备、查询其属性、为内核执行选择一个设备的函数（运行时 API 的

相关内容请参见第 4.5.2.2 节，驱动程序 API 的相关内容请参见第 4.5.3.2 节）。

多个宿主线程可在同一个设备上执行设备代码，但根据设计，一个宿主线程只能在一个设备上执行设备代码。因而，需要多个宿主线程在多个设备上执行设备代码。此外，通过一个宿主线程的运行时创建的 CUDA 资源无法由来自其他宿主线程的运行时使用。

4.5.1.2 存储器

设备存储器可指派为*线性存储器*（linear memory）或 *CUDA 数组*（CUDA arrays）。

设备上的线性存储器位于 32 位地址空间内，因此，独立分配的实体可通过指针引用另外一个实体，比如，在二进制树内。

CUDA 数组的存储器布局是不透明，专为纹理拾取而优化（参见第 4.3.4 节）。它们可以是一维、二维或三维的，由元素组成，这些元素包含 1、2、4 个分量，这些分量可以是有符号或无符号的 8 位、16 位或 32 位整型，也可以是 16 位浮点（当前仅有驱动程序 API 支持）或 32 位浮点。CUDA 数组仅可由内核通过纹理拾取读取，仅可绑定到具有相同分量数的纹理参考。

宿主可通过第 4.5.2.3 和第 4.5.3.6 节介绍的存储器复制函数读取和写入线性存储器和 CUDA 数组。

宿主运行时还提供了函数来分配和释放分页锁定（page-locked）的宿主存储器—与 malloc() 分配的普通可分页存储器恰好相反。分页锁定存储器的优势之一在于，如果将宿主存储器指派为分页锁定存储器，宿主存储器和设备存储器之间的带宽较高——但仅针对分配宿主存储器的宿主线程所执行的数据传输。分页锁定的存储器是一种稀缺资源，因此分页锁定存储器中的分配将先于可分页存储器的分配而出错。此外，由于减少了操作系统可用于分页的物理存储器数量，分配过多的分页锁定存储器将降低整体系统性能。

4.5.1.3 OpenGL 互操作性

OpenGL 缓冲对象（buffer object）可映射到 CUDA 的地址空间，从而使 CUDA 能够读取 OpenGL 写入的数据或使 CUDA 能够写入数据供 OpenGL 使用。第 4.5.2.7 节描述了如何通过运行时 API 实现此目标，第 4.5.3.10 节描述了如何通过驱动程序 API 实现此目标。

4.5.1.4 Direct3D 互操作性

Direct3D 资源可映射到 CUDA 的地址空间，从而使 CUDA 能够读取 Direct3D 写入的数据，或者使 CUDA 能够写入数据供 Direct3D 使用。第 4.5.2.8 节描述了如何通过运行时 API 实现此目标，同一节还介绍了如何通过驱动程序 API 实现此目标。

对于可映射哪些资源的限制条件，请参见 **`cudaD3D9RegisterResource()`** 和 **`cuD3D9RegisterResource()`** 参考手册。

CUDA 上下文一次仅可与一个 Direct3D 设备互操作，CUDA 上下文和 Direct3D 设备必须是在同一个 GPU 上创建的。此外，Direct3D 设备在创建时必须使用 **`D3DCREATE_HARDWARE_VERTEXPROCESSING`** 标记。

Direct3D 互操作性目前仅支持 Direct3D 9.0。

4.5.1.5 异步并发执行

为了促进宿主和设备之间的并发执行，某些运行时函数是异步的：控制将在设备完成所请求的任务之前返回应用程序。此类函数包括：

- 通过 __global__ 函数或 cuLaunchGrid() 和 cuLaunchGridAsync() 启动的内核；
- 执行存储器复制和带有 Async 后缀的函数；

- 执行设备与设备间双向存储器复制的函数；
- 设置存储器的函数。

某些设备还可在分页锁定的宿主存储器和设备存储器之间执行复制，且与内核执行并发地执行此类复制操作。如果使用的是运行时 API，应用程序可通过调用 cudaGetDeviceProperties() 并检查 deviceOverlap 来查询此功能；如果使用的是驱动程序 API，则可通过使用 CU_DEVICE_ATTRIBUTE_GPU_OVERLAP 调用 cuDeviceGetAttribute() 来查询。此功能当前仅支持不涉及 CUDA 数组或通过 cudaMallocPitch()（请参见第 4.5.2.3 节）或 cuMemAllocPitch()（请参见第 4.5.3.6 节）分配的二维数组的存储器复制。

应用程序通过*流*（stream）管理并发。流就是按顺序执行的一系列操作。另一方面，不同的流可与其他流乱序执行其操作，也可并发执行。

流的定义方法是创建流对象，并将其指定为内核启动和宿主与设备间内存双向复制序列的流参数。第 4.5.2.4 节描述了如何通过运行时 API 实现此目标，第 4.5.3.7 节描述了如何通过驱动程序 API 实现此目标。

不带流参数或使用零作为流参数的任何内核启动、存储器设置或存储器复制函数都仅能在此前的操作完成后开始，包括作为流的一部分的操作，而在此类函数完成之前，无法启动任何后续操作。无流参数的内核启动，无 Async 后缀的存储器复制都将被指派给默认零流。

用于运行时 API 的 cudaStreamQuery() 和用于驱动程序 API 的 cuStreamQuery() 为应用程序提供了一种方法，使其能够了解一个流中的之前全部操作是否已完成。用于运行时 API 的 cudaStreamSnchronize() 和用于驱动程序 API 的 cuStreamSynchronize() 提供了一种方法，可显式强制要求运行时等待流中之前的所有操作完成。

类似地，用于运行时 API 的 cudaThreadSynchronize() 和用于驱动程序 API 的 cuCtxSynchronize() 使应用程序能够强制要求运行时等待所有流中的所有之前设备任务完成。为了避免不必要的减速，最好将这些函数用于计时或者隔离失败的启动或存储器复制操作。用于运行时 API 的 cudaStreamDestroy() 和用于驱动程序 API 的 cuStreamDestroy() 将等待给定流内之前的所有任务完成，之后再销毁流，并将控制返回给宿主线程。

运行时还提供了一种方法，可密切监控设备的进度并执行准确的计时，它允许应用程序异步记录程序内任意点的事件，并查询这些事件的实际记录事件。一个事件将在先于该事件的所有任务（也可以是给定流中的所有操作）均已完成时记录。零流中的事件将在设备完成来自所有流的之前任务/操作后记录。第 4.5.2.5 节将描述如何通过运行时 API 实现此目标，第 4.5.3.8 节将描述如何通过驱动程序 API 实现此目标。

如果宿主线程在来自不同流的两项操作之间调用了分页锁定的宿主存储器分配、设备存储器分配、设备存储器设置、设备与设备之间的双向存储器复制或零流的任何 CUDA 操作，则这两项操作无法并行执行。

程序员可将 **CUDA_LAUNCH_BLOCKING** 环境变量设置为 1，从而为系统上运行的所有 CUDA 应用程序全局禁用异步执行。此特性仅为调试提供，不应用于使生产软件更可靠运行之目的。

4.5.2 运行时 API

4.5.2.1 初始化

运行时 API 不存在显式初始化函数；它将在运行时函数被首次调用时初始化。需要牢记，在计时（timing）运行时调用发生时，或解释来自第一次运行时调用的错误码时，即进行初始化。

4.5.2.2 设备管理

cudaGetDeviceCount() 和 **cudaGetDeviceProperties()** 提供了一种方法，用于枚举这些设备并检索其属性：

```
int deviceCount;
cudaGetDeviceCount(&deviceCount);
```

```
int device;
for (device = 0; device < deviceCount; ++device) {
    cudaDeviceProp deviceProp;
    cudaGetDeviceProperties(&deviceProp, device);
}
```

cudaSetDevice() 用于选择与宿主线程相关的设备：

```
cudaSetDevice(device);
```

必须首先选择设备，之后才能调用 __global__ 函数或任何来自运行时 API 的函数。如果未通过显式调用 cudaSetDevice() 完成此任务，将自动选中设备 0，随后对 cudaSetDevice() 的任何显式调用都将无效。

4.5.2.3 存储器管理

线性存储器是使用 cudaMalloc() 或 cudaMallocPitch() 分配的，使用 cudaFree() 释放。

以下示例代码将在线性存储器中分配一个包含 256 个浮点元素的数组：

```
float* devPtr;
cudaMalloc((void**)&devPtr, 256 * sizeof(float));
```

建议在分配二维数组时使用 cudaMallocPitch()，因为它能确保合理填充已分配的存储器，满足第 5.1.2.1 节介绍的对齐要求，从而确保访问行地址或执行二维数组与设备存储器的其他区域之间的复制（使用 cudaMemcpy2D()）时获得最优性能。所返回的间距（或步幅）必须用于访问数组元素。以下代码示例将分配一个 widthxheight 的二维浮点值数组，并显示如何在设备代码中循环遍历数组元素：

```
// host code
float* devPtr;
int pitch;
cudaMallocPitch((void**)&devPtr, &pitch,
                width * sizeof(float), height);
myKernel<<<100, 512>>>(devPtr, pitch);
// device code
__global__ void myKernel(float* devPtr, int pitch)
{
    for (int r = 0; r < height; ++r) {
        float* row = (float*)((char*)devPtr + r * pitch);
        for (int c = 0; c < width; ++c) {
            float element = row[c];
        }
    }
}
```

CUDA 数组是使用 cudaMallocArray() 分配的，使用 cudaFreeArray() 释放。cudaMallocArray() 需要使用 cudaCreateChannelDesc() 创建的格式描述。

以下代码示例分配了一个 widthxheight 的 CUDA 数组，包含一个 32 位的浮点组件：

```
cudaChannelFormatDesc channelDesc =
                                    cudaCreateChannelDesc<float>();
cudaArray* cuArray;
cudaMallocArray(&cuArray, &channelDesc, width, height);
```

cudaGetSymbolAddress() 用于检索指向为全局存储器空间中声明的变量分配的存储器的地址。所分配存储器的大小是通过 cudaGetSymbolSize() 获取的。

参考手册列举了用于在 cudaMalloc() 分配的线性存储器、cudaMallocPitch() 分配的线性存储器、CUDA 数组和为全局或常量存储器空间中声明的变量分配的存储器之间复制存储器的所有函数。

下面的代码示例将二维数组复制到之前代码示例中分配的 CUDA 数组中：

```
cudaMemcpy2DToArray(cuArray, 0, 0, devPtr, pitch,
                    width * sizeof(float), height,
                    cudaMemcpyDeviceToDevice);
```

下面的代码示例将一些宿主存储器数组复制到设备存储器中：

```
float data[256];
int size = sizeof(data);
float* devPtr;
cudaMalloc((void**)&devPtr, size);
```

```
cudaMemcpy(devPtr, data, size, cudaMemcpyHostToDevice);
```

下面的代码示例将一些宿主存储器数组复制到常量存储器中：

```
__constant__ float constData[256];
float data[256];
cudaMemcpyToSymbol(constData, data, sizeof(data));
```

4.5.2.4 流管理

以下代码示例创建两个流：

```
cudaStream_t stream[2];
for (int i = 0; i < 2; ++i)
    cudaStreamCreate(&stream[i]);
```

这些流均通过以下代码示例定义为一个任务序列，包括一次从宿主到设备的存储器复制、一次内核启动、一次从设备到宿主的存储器复制：

```
for (int i = 0; i < 2; ++i)
    cudaMemcpyAsync(inputDevPtr + i * size, hostPtr + i * size,
                    size, cudaMemcpyHostToDevice, stream[i]);
for (int i = 0; i < 2; ++i)
    myKernel<<<100, 512, 0, stream[i]>>>
        (outputDevPtr + i * size, inputDevPtr + i * size, size);
for (int i = 0; i < 2; ++i)
    cudaMemcpyAsync(hostPtr + i * size, outputDevPtr + i * size,
                    size, cudaMemcpyDeviceToHost, stream[i]);
cudaThreadSynchronize();
```

两个流均会将其输入数组 hostPtr 的一部分复制到设备存储器的 inputDevPtr 数组中，通过调用 myKernel() 处理设备上的 inputDevPtr，并将结果 outputDevPtr 复制回 hostPtr 的相同部分。使用两个流处理 hostPtr 允许一个流的存储器复制与另外一个流的内核执行相互重叠。hostPtr 必须指向分页锁定的宿主存储器，这样才能同时执行：

```
float* hostPtr;
cudaMallocHost((void**)&hostPtr, 2 * size);
```

最后调用了 cudaThreadSynchronize()，目的是在进一步处理之前确定所有流均已完成。cudaStreamSynchronize() 可用于同步宿主与特定流，允许其他流继续在该设备上执行。通过调用 cudaStreamDestroy() 可释放流。

4.5.2.5 事件管理

下面的代码示例创建了两个事件：

```
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
```

这些事件可用于为上一节的代码示例计时，方法如下：

```
cudaEventRecord(start, 0);
for (int i = 0; i < 2; ++i)
    cudaMemcpyAsync(inputDev + i * size, inputHost + i * size,
                    size, cudaMemcpyHostToDevice, stream[i]);
for (int i = 0; i < 2; ++i)
    myKernel<<<100, 512, 0, stream[i]>>>
              (outputDev + i * size, inputDev + i * size, size);
for (int i = 0; i < 2; ++i)
    cudaMemcpyAsync(outputHost + i * size, outputDev + i * size,
                    size, cudaMemcpyDeviceToHost, stream[i]);
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
float elapsedTime;
cudaEventElapsedTime(&elapsedTime, start, stop);

cudaEventDestroy(start);
cudaEventDestroy(stop);
```

4.5.2.6 纹理参考管理

高级 API 定义的 texture 类型是一种公开继承自低级 API 定义的 textureReference 类型的纹理，如下：

```
struct textureReference
{
    int normalized;
    enum cudaTextureFilterMode filterMode;
    enum cudaTextureAddressMode addressMode[3];
    struct cudaChannelFormatDesc channelDesc;
}
```

- Normalized 指定纹理坐标是否为归一化形式；如果非零，纹理中的所有元素都将使用 [0, 1] 范围内的纹理坐标寻址，而非 **[0,width-1]**、**[0,height-1]** 或 **[0,depth-1]**，其中的 width、height 和 depth 是纹理大小；
- filterMode 指定过滤模式，即拾取根据输入的纹理坐标计算的纹理时如何返回值；filterMode 等于 cudaFilterModePoint 或 cudaFilterModeLinear；如果等于 cudaFilterModePoint，则所返回的值为纹理坐标最接近输入纹理坐标的 texel；如果等于 cudaFilterModeLinear，则所返回的值为纹理坐标最接近输入纹理坐标的两个（针对一维纹理）、四个（针对二维纹理）或八个（针对三维纹理）texel 的线性插值；

对于浮点型的返回值，cudaFilterModeLinear 是惟一的有效值。

- addressMode 指定寻址模式，表明如何处理超出范围的纹理坐标；addressMode 是一个大小为 3 的数组，其第一个、第二个和第三个元素分别指定第一个、第二个和第三个纹理坐标的寻址模式；寻址模式可等于 cudaAddressModeClamp，此时超出范围的纹理坐标将被调整（clamp）到有效范围之内，也可等于 cudaAddressModeWrap，此时超出范围的纹理坐标将被限定（wrap）到有效范围之内；

对于规范化的纹理坐标，仅支持 cudaAddressModeWrap。

- channelDesc 描述拾取纹理时所返回的值的格式；channelDesc 的类型如下：

```
struct cudaChannelFormatDesc {
    int x, y, z, w;
    enum cudaChannelFormatKind f;
};
```

其中 x、y、z 和 w 等于返回值各分量的位（bit）数，而 f 为：

- cudaChannelFormatKindSigned，在这些分量是有符号整型时；
- cudaChannelFormatKindUnsigned，在这些分量是无符号整型时；
- cudaChannelFormatKindFloat，在这些分量是浮点类型时。

normalized、addressMode 和 filterMode 可直接在宿主代码中修改。它们仅适用于绑定到 CUDA 数组的纹理参考。

在内核使用纹理参考从纹理存储器中读取之前，必须使用 cudaBindTexture() 或 cudaBindTextureToArray() 将纹理参考绑定到纹理。

以下代码示例将一个纹理参考绑定到 devPtr 指向的线性存储器：

- 使用低级 API：

```
texture<float, 1, cudaReadModeElementType> texRef;
textureReference* texRefPtr;
cudaGetTextureReference(&texRefPtr, “texRef”);
cudaChannelFormatDesc channelDesc =
cudaCreateChannelDesc<float>();
cudaBindTexture(0, texRefPtr, devPtr, &channelDesc, size);
```

- 使用高级 API：

```
texture<float, 1, cudaReadModeElementType> texRef;
cudaBindTexture(0, texRef, devPtr, size);
```

以下代码示例将一个纹理参考绑定到 CUDA 数组 cuArray：

- 使用低级 API：

```
texture<float, 2, cudaReadModeElementType> texRef;
textureReference* texRefPtr;
cudaGetTextureReference(&texRefPtr, “texRef”);
cudaChannelFormatDesc channelDesc;
```

```
cudaGetChannelDesc(&channelDesc, cuArray);
cudaBindTextureToArray(texRef, cuArray, &channelDesc);
```

- 使用高级 API：

```
texture<float, 2, cudaReadModeElementType> texRef;
cudaBindTextureToArray(texRef, cuArray);
```

将纹理绑定到纹理参考时指定的格式必须与声明纹理参考时指定的参数相匹配；否则纹理拾取的结果将无法确定。

cudaUnbindTexture() 用于解除纹理参考的绑定。

4.5.2.7 OpenGL 互操作性

首先必须将一个缓冲对象注册到 CUDA，之后才能进行映射。可通过 cudaGLRegisterBufferObject() 完成：

```
GLuint bufferObj;
cudaGLRegisterBufferObject(bufferObj);
```

注册完成后，内核即可使用 cudaGLMapBufferObject() 返回的设备存储器地址读取或写入缓冲对象：

```
GLuint bufferObj;
float* devPtr;
cudaGLMapBufferObject((void**)&devPtr, bufferObj);
```

解除映射是通过 cudaGLUnmapBufferObject() 完成的，可使用 cudaGLUnregisterBufferObject() 取消注册。

4.5.2.8 Direct3D 互操作性

Direct3D 互操作性要求在执行其他任何运行时调用之前通过 cudaD3D9SetDirect3DDevice() 指定 Direct3D 设备。

随后即可使用 cudaD3D9RegisterResource() 将 Direct3D 资源注册到 CUDA：

```
LPDIRECT3DVERTEXBUFFER9 buffer;
cudaD3D9RegisterResource(buffer, cudaD3D9RegisterFlagsNone);
LPDIRECT3DSURFACE9 surface;
cudaD3D9RegisterResource(surface, cudaD3D9RegisterFlagsNone);
```

cudaD3D9RegisterResource() 可能具有较高的开销，通常仅为每个资源调用一次。使用 **cudaD3D9UnregisterVertexBuffer()** 可取消注册。将资源注册到 CUDA 之后，即可在需要时分别使用 **cudaD3D9MapResources()** 和 **cudaD3D9UnmapResources()** 任意多次地映射和解除映射。内核可使用 **cudaD3D9ResourceGetMappedPointer()** 返回的设备存储器地址和 **cudaD3D9ResourceGetMappedSize()**、**cudaD3D9ResourceGetMappedPitch()** 及 **cudaD3D9ResourceGetMappedPitchSlice()** 返回的大小和间距信息来读取和写入已映射的资源。通过 Direct3D 访问已映射的资源将导致不确定的结果。

下面的代码示例使用 0 填充了一个缓冲区：

```
void* devPtr;
cudaD3D9ResourceGetMappedPointer(&devPtr, buffer);
size_t size;
cudaD3D9ResourceGetMappedSize(&size, buffer);
cudaMemset(devPtr, 0, size);
```

在下面的代码示例中，每个线程都访问大小为 **(width, height)** 的二维表面的一个像素，像素格式为 `float4`：

```
// host code
void* devPtr;
cudaD3D9ResourceGetMappedPointer(&devPtr, surface);
size_t pitch;
cudaD3D9ResourceGetMappedPitch(&pitch, surface);
dim3 Db = dim3(16, 16);
dim3 Dg = dim3((width+Db.x-1)/Db.x, (height+Db.y-1)/Db.y);
myKernel<<<Dg, Db>>>((unsigned char*)devPtr,
                     width, height, pitch);
// device code
```

```
__global__ void myKernel(unsigned char* surface,
                         int width, int height, size_t pitch)
{
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;
    if (x >= width || y >= height) return;
    float* pixel = (float*)(surface + y * pitch) + 4 * x;
}
```

4.5.2.9 使用设备模拟模式进行调试

对于在设备上运行的代码，编程环境未包含任何本地调试支持，但附带了一种设备模拟模式（device emulation mode）用于调试。在此模式中编译应用程序时（使用 –deviceemu 选项），设备代码将为宿主编译，并在宿主上运行，允许程序员使用宿主的本地调试支持来调试应用程序，就像调试宿主应用程序一样。预处理宏 __DEVICE_EMULATION__ 是在此模式中定义的。一个应用程序的所有代码，包括所用的全部库在内，都必须一致地为设备模拟或设备执行而编译。将为设备模拟编译的代码与为设备执行编译的代码相连接将导致初始化时返回以下运行时错误：

cudaErrorMixedDeviceExecution.

在设备模拟模式中运行应用程序时，运行时将模拟编程模型。对于线程块中的每一个线程，运行时都会在宿主上创建一个线程。程序员需要确保：

- 宿主能够运行每个块的最大线程数，此外还有一个针对主线程的线程。
- 有足够大的存储器可用于运行所有线程，已知每个线程都要占据堆栈中的 256 KB 空间。

设备模拟模式提供的众多特性使之成为一种独具优势的调试工具：

- 通过使用宿主的本地调试支持，程序员即可利用调试程序支持的所有特性，例如设置断点和检查数据。
- 由于设备代码被编译为在宿主上运行，因而可通过无法在设备上运行的代码扩展代码，如文件输入和输出操作或打印到屏幕（printf() 等）。
- 由于所有数据都位于宿主上，因而可从设备或宿主代码中读取任何特定于设备或特定于宿主的数据；类似地，可通过设备或宿主代码调用任何设备或宿主函数。
- 如果误用了同步函数，运行时将检测到死锁情况。

程序员必须牢记，设备模拟模式的目的在于模拟设备，而非仿真（simulating）。因而，设备模拟模式在查找算法错误时非常有用，但可能难以发现某些错误：

- 竞争条件在设备模拟模式中可能无法体现，因为同时执行的线程数量要比实际设备上少得多。
- 在解除指针对宿主上的全局存储器的引用或对设备上的宿主存储器的引用时，设备执行几乎必然以某种无法确定的方式失败，而设备模拟可能会得到正确的结果。
- 大多数时候，相同的浮点计算在设备上执行时所得到的结果，与在设备模拟模式中在宿主上执行时所得到的结果并不完全相同。这些情况是可以预见的，只要使用略有差异的编译器选项，相同的浮点计算就会得到不同的结果，更不用说不同的编译器、不同的指令集或不同的架构了。

具体来说，某些宿主平台会将单精度浮点计算的中间结果存储在一个扩展精度寄存器中，在设备模拟模式中运行时，这可能会造成准确性的显著差异。发生这种情况时，程序员可尝试以下方法，但没有任何一种方法能确保成功：

- 用 volatile 声明一些浮点变量，实施单精度存储；
- 使用 gcc 的 -ffloat-store 编译器选项；
- 使用 Visual C++ 编译器的 /Op 或 /fp 编译器选项；
- 在 Linux 上使用 _FPU_GETCW()、在 Windows 上使用 _controlfp()，使用以下代码包围目标代码，从而实施单精度浮点计算：

```
unsigned int originalCW;
_FPU_GETCW(originalCW);
unsigned int cw = (originalCW & ~0x300) | 0x000;
_FPU_SETCW(cw);
```

也可使用：

```
unsigned int originalCW = _controlfp(0, 0);
_controlfp(_PC_24, _MCW_PC);
```

在开头处，为了存储控制字的当前值并更改它以使尾数存储在 24 个位中，可使用：

```
_FPU_SETCW(originalCW);
```

或在结尾处使用以下代码，恢复原始控制字：

```
_controlfp(originalCW, 0xfffff);
```

同样，对于单精度浮点数，与计算设备不同（请参见附录 A），宿主平台通常支持反向规格化数（denormalized number）。这可能会导致设备模拟和设备执行模式下的结果存在显著差异，因为某些计算可能会在一种模式下产生有穷的结果，而在另一种模式下产生无穷的结果。

- 在设备模拟模式中，warp 块的大小等于 1。因而，warp vote 函数产生的结果将与设备执行模式中不同。

4.5.3 驱动程序 API

驱动程序 API 是一种基于句柄、命令式的 API：大多数对象都通过不透明的句柄引用，此类句柄可指定为操纵对象的函数。

表 4-1 汇总了 CUDA 中可用的对象。

表 4-1. CUDA 驱动程序 API 中可用的对象

对象	句柄	描述
设备	CUdevice	支持 CUDA 的设备
上下文	CUcontext	大致等同于 CPU 进程
模块	CUmodule	大致等同于动态库
函数	CUfunction	内核
堆存储器	CUdeviceptr	设备存储器的指针
CUDA 数组	CUarray	设备上一维或二维数据的不透明容器，可通过纹理参考读取
纹理参考	CUtexref	描述如何解释纹理存储器数据的对象

4.5.3.1 初始化

在驱动程序 API 中调用任何函数之前，都必须使用 cuInit() 进行初始化。

4.5.3.2 设备管理

cuDeviceGetCount() 和 **cuDeviceGet()** 提供了一种方法，可枚举这些设备和其他功能（详见参考手册），以便检索其属性：

```
int deviceCount;
cuDeviceGetCount(&deviceCount);
int device;
for (int device = 0; device < deviceCount; ++device) {
    CUdevice cuDevice;
    cuDeviceGet(&cuDevice, device);
    int major, minor;
    cuDeviceComputeCapability(&major, &minor, cuDevice);
}
```

4.5.3.3 上下文管理

CUDA 上下文类似于 CPU 进程。在驱动程序 API 中执行的所有资源和操作都封装在 CUDA 上下文内，在该上下文被销毁时，系统将自动清除这些资源。除了模块和纹理参考之类的对象之外，每个上下文还有自己独有的 32 位地址空间。因而，不同上下文的 Cudeviceptr 值将引用不同的存储器位置。

一个宿主线程一次只能有一个当前设备上下文。在使用 cuCtxCreate() 创建上下文时，它将成为调用方宿主线程的当前上下文。如果有效上下文未成为线程的当前上下文，在一个上下文中操作的 CUDA 函数（不涉及设备模拟或上下文管理的大多数函数）将返回 **CUDA_ERROR_INVALID_CONTEXT。**

每个宿主线程都有一个当前上下文堆栈。cuCtxCreate() 将新上下文压入堆栈顶端。可调用 cuCtxPopCurrent() 将上下文与宿主线程分离开来。随后此上下文将成为“浮动（floating）”上下文，可作为任意宿主线程的当前上下文压入。cuCtxPopCurrent() 还可恢复之前的当前上下文（如果存在）。

此外还会为每个上下文维护一个使用计数（usage count）。cuCtxCreate() 将通过使用计数 1 创建一个上下文。cuCtxAttach() 将使使用计数递增，而 cuCtxDetach() 则使之递减。在调用 cuCtxDetach() 或 cuCtxDestroy() 使使用计数为 0 时，上下文将被销毁。

使用计数促进了在同一上下文中操作的第三方授权代码之间的互操作。举例来说，如果载入了三个库，使用相同的上下文，则每个库都将调用 cuCtxAttach() 来递增使用计数，并在库不再使用该上下文时调用 cuCtxDetach() 来递减使用计数。对于大多数来说，应用程序将在载入或初始化库之前创建一个上下文，通过这种方式，应用程序即可使用自己的启发式方法来创建上下文，库只需在传递给它的上下文中操作即可。希望创建自己的上下文的库（其 API 客户端并不了解这种情况，并且可能已经创建或未创建自己的上下文）可使用 cuCtxPushCurrent() 和 cuCtxPopCurrent()，如图 4-1 所示。

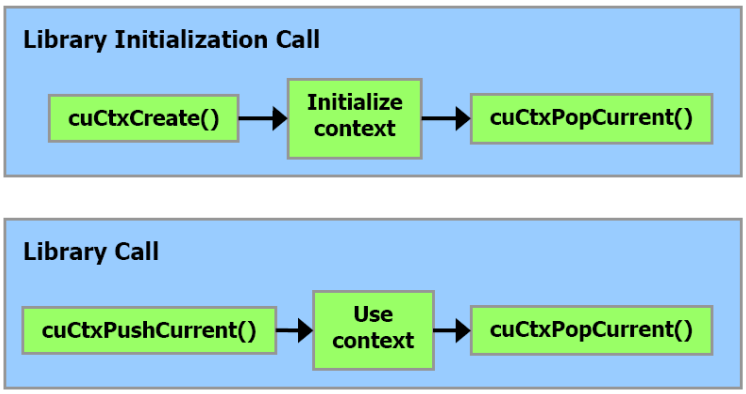


图 4-1. 库上下文管理

4.5.3.4 模块管理

模块（module）是可动态加载的设备代码和数据包，类似于 Windows 中的 DLL，是由 nvcc 输出的（请参见第 4.2.5 节）。所有符号的名称（包括函数、全局变量和纹理参考）均在模块范围内维护，从而使独立的第三方编写的模块可在相同的 CUDA 上下文中进行互操作。

下面的代码示例载入了一个模块并检索内核的句柄：

```
CUmodule cuModule;
cuModuleLoad(&cuModule, “myModule.cubin”);
CUfunction cuFunction;
cuModuleGetFunction(&cuFunction, cuModule, “myKernel”);
```

4.5.3.5 执行控制

cuFuncSetBlockShape() 为给定函数设置每个块的线程数，同时还会设置其 threadID 的分配方式。

cuFuncSetSharedSize() 为函数设置共享存储器的大小。

cuParam*() 系列函数用于指定在下一次调用 **cuLaunchGrid()** 或 **cuLaunch()** 来启动内核时为内核提供的参数。各 **cuParam*()** 函数的第二个参数指定参数在参数堆栈中的偏移。这个偏移量必须与参数类型的对齐要求相匹配。可通过使用 **__alignof()** 以一种可迁移的方式来完成此任务：

```
cuFuncSetBlockShape(cuFunction, blockWidth, blockHeight, 1);
```

```
int offset = 0;
int i;
cuParamSeti(cuFunction, offset, i);
offset += sizeof(i);
float f;
cuParamSetf(cuFunction, offset, f);
offset += sizeof(f);
char data[32];
cuParamSetv(cuFunction, offset, (void*)data, sizeof(data));
offset += sizeof(data);
cuParamSetSize(cuFunction, offset);
cuFuncSetSharedSize(cuFunction, numElements * sizeof(float));
cuLaunchGrid(cuFunction, gridWidth, gridHeight);
```

4.5.3.6 存储器管理

可使用 cuMemAlloc() 或 cuMemAllocPitch() 分配线性存储器，并使用 cuMemFree() 释放线性存储器。

下面的代码示例在线性存储器中分配了一个包含 256 个浮点元素的数组：

```
CUdeviceptr devPtr;
cuMemAlloc(&devPtr, 256 * sizeof(float));
```

建议在分配二维数组时使用 cudaMallocPitch()，因为它能确保合理填充已分配的存储器，满足第 5.1.2.1 节介绍的对齐要求，从而确保访问行地址或执行二维数组与设备存储器的其他区域之间的复制（使用 cudaMemcpy2D()）时获得最优性能。所返回的间距（或步幅）必须用于访问数组元素。以下代码示例将分配一个 widthxheight 的二维浮点值数组，并显示如何在设备代码中循环遍历数组元素：

```
// host code
CUdeviceptr devPtr;
int pitch;
cuMemAllocPitch(&devPtr, &pitch,
                width * sizeof(float), height, 4);
cuModuleGetFunction(&cuFunction, cuModule, “myKernel”);
cuFuncSetBlockShape(cuFunction, 512, 1, 1);
cuParamSeti(cuFunction, 0, devPtr);
cuParamSetSize(cuFunction, sizeof(devPtr));
cuLaunchGrid(cuFunction, 100, 1);
// device code
__global__ void myKernel(float* devPtr)
{
    for (int r = 0; r < height; ++r) {
        float* row = (float*)((char*)devPtr + r * pitch);
        for (int c = 0; c < width; ++c) {
            float element = row[c];
        }
    }
}
```

CUDA 数组是使用 cuArrayCreate() 创建的，使用 cuArrayDestroy() 销毁。

以下代码示例分配了一个 widthxheight 的 CUDA 数组，包含一个 32 位的浮点组件：

```
CUDA_ARRAY_DESCRIPTOR desc;
desc.Format = CU_AD_FORMAT_FLOAT;
desc.NumChannels = 1;
desc.Width = width;
desc.Height = height;
CUarray cuArray;
cuArrayCreate(&cuArray, &desc);
```

参考手册列举了用于在 cuMemAlloc() 分配的线性存储器、cuMemAllocPitch() 分配的线性存储器、CUDA 数组之间复制存储器的所有函数。下面的代码示例将二维数组复制到之前代码示例中分配的 CUDA 数组中：

```
CUDA_MEMCPY2D copyParam;
memset(&copyParam, 0, sizeof(copyParam));
copyParam.dstMemoryType = CU_MEMORYTYPE_ARRAY;
copyParam.dstArray = cuArray;
copyParam.srcMemoryType = CU_MEMORYTYPE_DEVICE;
```

```
copyParam.srcDevice = devPtr;
copyParam.srcPitch = pitch;
copyParam.WidthInBytes = width * sizeof(float);
copyParam.Height = height;
cuMemcpy2D(&copyParam);
```

下面的代码示例将一些宿主存储器数组复制到设备存储器中：

```
float data[256];
int size = sizeof(data);
CUdeviceptr devPtr;
cuMemAlloc(&devPtr, size);
cuMemcpyHtoD(devPtr, data, size);
```

4.5.3.7 流管理

下面的代码示例创建了两个流：

```
CUstream stream[2];
for (int i = 0; i < 2; ++i)
    cuStreamCreate(&stream[i], 0);
```

这些流均通过以下代码示例定义为一个任务序列，包括一次从宿主到设备的存储器复制、一次内核启动、一次从设备到宿主的存储器复制：

```
for (int i = 0; i < 2; ++i)
    cuMemcpyHtoDAsync(inputDevPtr + i * size, hostPtr + i * size,
                      size, stream[i]);
for (int i = 0; i < 2; ++i) {
    cuFuncSetBlockShape(cuFunction, 512, 1, 1);
    int offset = 0;
    cuParamSeti(cuFunction, offset, outputDevPtr);
    offset += sizeof(int);
    cuParamSeti(cuFunction, offset, inputDevPtr);
    offset += sizeof(int);
    cuParamSeti(cuFunction, offset, size);
    offset += sizeof(int);
    cuParamSetSize(cuFunction, offset);
    cuLaunchGridAsync(cuFunction, 100, 1, stream[i]);
}
for (int i = 0; i < 2; ++i)
    cuMemcpyDtoHAsync(hostPtr + i * size, outputDevPtr + i * size,
                      size, stream[i]);
cudaCtxSynchronize();
```

两个流均会将其输入数组 hostPtr 的一部分复制到设备存储器中的 inputDevPtr 数组中，通过调用 **cuFunction** 处理设备上的 inputDevPtr，并将结果 outputDevPtr 复制回 hostPtr 的相同部分。使用两个流处理 hostPtr 允许一个流的存储器复制与另外一个流的内核执行相互重叠。hostPtr 必须指向分页锁定的宿主存储器，这样才能同时执行：

```
float* hostPtr;
cuMemAllocHost((void**)&hostPtr, 2 * size);
```

最后调用了 **cuCtxSynchronize()**，目的是在进一步处理之前确定所有流均已完成。**cuStreamSynchronize()** 可用于同步宿主与特定流，允许其他流继续在该设备上执行。

4.5.3.8 事件管理

下面的代码示例创建了两个事件：

```
CUevent start, stop;
cuEventCreate(&start);
cuEventCreate(&stop);
```

这些事件可用于为上一节的代码示例计时，方法如下：

```
cuEventRecord(start, 0);
for (int i = 0; i < 2; ++i)
    cuMemcpyHtoDAsync(inputDevPtr + i * size, hostPtr + i * size,
```

```
                        size, stream[i]);
    for (int i = 0; i < 2; ++i) {
        cuFuncSetBlockShape(cuFunction, 512, 1, 1);
        int offset = 0;
        cuParamSeti(cuFunction, offset, outputDevPtr);
        offset += sizeof(outputDevPtr);
        cuParamSeti(cuFunction, offset, inputDevPtr);
        offset += sizeof(inputDevPtr);
        cuParamSeti(cuFunction, offset, size);
        offset += sizeof(size);
        cuParamSetSize(cuFunction, offset);
        cuLaunchGridAsync(cuFunction, 100, 1, stream[i]);
    }
    for (int i = 0; i < 2; ++i)
        cuMemcpyDtoHAsync(hostPtr + i * size, outputDevPtr + i * size,
                        size, stream[i]);
    cuEventRecord(stop, 0);
    cuEventSynchronize(stop);
    float elapsedTime;
    cuEventElapsedTime(&elapsedTime, start, stop);
    cuEventDestroy(start);
    cuEventDestroy(stop);
```

4.5.3.9 纹理参考管理

在内核使用纹理参考从纹理存储器中读取之前，必须使用 **cuTexRefSetAddress()** 或 **cuTexRefSetArray()**将纹理参考绑定到纹理。

如果模块 cuModule 包含定义如下的纹理参考 texRef：

```
texture<float, 2, cudaReadModeElementType> texRef;
```

则下面的代码示例将检索 texRef 的句柄：

```
CUtexref cuTexRef;
cuModuleGetTexRef(&cuTexRef, cuModule, “texRef”);
```

下面的代码示例将 texRef 绑定到 devPtr 指向的线性存储器：

```
cuTexRefSetAddress(NULL, cuTexRef, devPtr, size);
```

下面的代码示例将 texRef 绑定到 CUDA 数组 cuArray：

```
cuTexRefSetArray(cuTexRef, cuArray, CU_TRSA_OVERRIDE_FORMAT);
```

参考手册列举了用于设置寻址模式、过滤模式和其他针对纹理参考的标记的各种函数。在将纹理绑定到纹理参考时所指定的格式必须与声明纹理参考时指定的参数相匹配；否则纹理拾取的结果将无法确定。

4.5.3.10 OpenGL 互操作性

必须使用 cuGLInit() 初始化与 OpenGL 的互操作性。

首先必须将一个缓冲对象注册到 CUDA，之后才能进行映射。可通过 **cuGLRegisterBufferObject()** 完成：

```
GLuint bufferObj;
cuGLRegisterBufferObject(bufferObj);
```

注册完成后，内核即可使用 **cuGLMapBufferObject()** 返回的设备存储器地址读取或写入缓冲对象：

```
GLuint bufferObj;
CUdeviceptr devPtr;
int size;
cuGLMapBufferObject(&devPtr, &size, bufferObj);
```

解除映射是通过 **cuGLUnmapBufferObject()** 完成的，可使用 **cuGLUnregisterBufferObject()** 取消注册。

4.5.3.11 Direct3D 互操作性

Direct3D 互操作性要求在创建 CUDA 上下文时指定 Direct3D 设备。通过使用 **cuD3D9CtxCreate()** 而非 **cuCtxCreate()** 创建 CUDA 上下文即可实现此目标。。

随后即可使用 **cuD3D9RegisterResource()** 将 Direct3D 资源注册到 CUDA：

```
LPDIRECT3DVERTEXBUFFER9 buffer;
cuD3D9RegisterResource(buffer, CU_D3D9_REGISTER_FLAGS_NONE);
LPDIRECT3DSURFACE9 surface;
cuD3D9RegisterResource(surface, CU_D3D9_REGISTER_FLAGS_NONE);
```

cuD3D9RegisterResource() 可能具有较高的开销，通常仅为每个资源调用一次。使用 **cuD3D9UnregisterVertexBuffer()** 可取消注册。

将资源注册到 CUDA 之后，即可在需要时分别使用 **cuD3D9MapResources()** 和 **cuD3D9UnmapResources()**任意多次地映射和解除映射。内核可使用 **cuD3D9ResourceGetMappedPointer()** 返回的设备存储器地址和 **cuD3D9ResourceGetMappedSize()**、**cuD3D9ResourceGetMappedPitch()** 及 **cuD3D9ResourceGetMappedPitchSlice()** 返回的大小和间距信息来读取和写入已映射的资源。通过 Direct3D 访问已映射的资源将导致不确定的结果。

下面的代码示例使用 0 填充了一个缓冲区：

```
CUdeviceptr devPtr;
cuD3D9ResourceGetMappedPointer(&devPtr, buffer);
size_t size;
cuD3D9ResourceGetMappedSize(&size, buffer);
cuMemset(devPtr, 0, size);
```

在下面的代码示例中，每个线程都访问大小为 **(width, height)** 的二维表面的一个像素，像素格式为 float4：

```
// host code
CUdeviceptr devPtr;
cuD3D9ResourceGetMappedPointer(&devPtr, surface);
size_t pitch;
cuD3D9ResourceGetMappedPitch(&pitch, surface);
cuModuleGetFunction(&cuFunction, cuModule, “myKernel”);
cuFuncSetBlockShape(cuFunction, 16, 16, 1);
int offset = 0;
cuParamSeti(cuFunction, offset, devPtr);
offset += sizeof(devPtr);
cuParamSeti(cuFunction, 0, width);
offset += sizeof(width);
cuParamSeti(cuFunction, 0, height);
offset += sizeof(height);
cuParamSeti(cuFunction, 0, pitch);
offset += sizeof(pitch);
cuParamSetSize(cuFunction, offset);
cuLaunchGrid(cuFunction,
             (width+Db.x-1)/Db.x, (height+Db.y-1)/Db.y);
// device code
__global__ void myKernel(unsigned char* surface,
                         int width, int height, size_t pitch)
{
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;
    if (x >= width || y >= height) return;
    float* pixel = (float*)(surface + y * pitch) + 4 * x;
}
```

第 5 章 性能指南

5.1 指令性能

要为线程 warp 块处理指令，多处理器必须：

- 读取 warp 块中各线程的指令操作数；
- 执行指令；
- 为 warp 块中各线程写入结果。

因而，有效的指令吞吐量取决于额定指令吞吐量以及存储器延迟和带宽。可通过以下方法最大化指令吞吐量：

- 最小化吞吐量较低的指令的使用（参见第 5.1.1 节）；
- 最大化用于每一类存储器的可用存储器带宽的使用（参见第 5.1.2 节）；
- 允许线程调度程序尽可能地同时进行存储器事务处理与数学计算，这要求：
  - 线程执行的程序具有较高的运算强度（arithmetic intensity），也就是说，让每个存储器操作对应大量数学操作；
  - 每个多处理器上具有多个活动线程，如第 5.2 节所述。

5.1.1 指令吞吐量

5.1.1.1 数学指令

要为一个 warp 块发出一条指令，多处理器需占用：

- 4 个时钟周期，用于：
  - 单精度浮点加法、乘法和乘-加；
  - 整型加法；
  - 位运算、比较、最小值、最大值、类型转换指令；
- 16 个时钟周期，用于倒数、平方根倒数、__logf(x)（请参见第 B.2 节）。

32 位整型乘法占用 16 个时钟周期，但 __mul24 和 __umul24（请参见附录 B）可在 4 个时钟周期内提供有符号和无符号 24 位整型乘法。而在未来的架构中，__[u]mul24 将比 32 位整型乘法的速度更慢，因此我们建议您设计两套内核，一个使用 __[u]mul24，另一个使用普通的 32 位整型乘法，由应用程序合理调用。

整型除法和求模运算的成本极高，应尽可能避免使用，或在可能的情况下替换为位运算：如果 n 是 2 的幂，（`i/n`）等价于（`i>>log2(n)`），而（`i%n`）等价于（`i&(n-1)`）；如果 n 是常量，则编译器将执行这些转换。

其他函数要占用更多的时钟周期，因为它们是作为多种指令的混合体实现的。

单精度浮点平方根是作为倒数平方根再求倒数实现的，而非先求倒数平方根再求乘积，因为只有这样才能为 0 和无穷大给出正确结果。因而，一个 warp 块将占用 32 个时钟周期。
单精度浮点除法要占用 36 个时钟周期，但 __fdividef(x, y) 提供了速度更快的版本，只需占用 20 个时钟周期（请参见附录 B）。

`__sinf(x)`、`__cosf(x)`、`__expf(x)` 占用 32 个时钟周期。`sinf(x)`、`cosf(x)`、`tanf(x)`、`sincosf(x)`

的成本更高，如果 x 的绝对值大于 48039（参见 math_functions.h 了解更多细节），则成本还要更高（也就是说，速度降低约一个数量级）。此外，在这种情况下，参数约减代码使用本地存储器，这可能会进一步降低性能，因为本地存储器的延迟较高，也存在带宽问题（请参见第 5.1.2.2 节）。

有些时候，编译器必须插入转换指令，这又带来了额外的执行周期。可能导致此类情况的条件包括：

- 对 char 或 short 类型进行操作的函数，其操作数通常需要转换为 int；
- 双精度浮点常量（定义时不带任何类型后缀）用作单精度浮点计算的输入；
- 单精度浮点变量用作 B.1.2 节所定义的数学函数的双精度版本的输入参数。

可通过以下方法避免上述的后两种情况：

- 使用单精度浮点常量，在定义时带 f 后缀，如 **`3.141592653589793f`**、**`1.0f`**、**`0.5f`**；
- 使用数学函数的单精度版本，定义时同样带 `f` 后缀，如 `sinf()`、`logf()`、`expf()`。

对于单精度代码，强烈建议使用 float 类型和单精度数学函数。在为不具备本地双精度支持的设备编译时，如计算能力为 1.2 或更低的设备，应将双精度算法替换为单精度算法。

5.1.1.2 控制流指令

任何流控制指令（if、switch、do、for、while）都会导致同一 warp 块的线程分支，从而显著影响有效的指令吞吐量，也就是说，这些指令会导致线程采用不同的执行路径。如果出现这种情况，就必须序列化（serialize）不同的执行路径，因而增加了该 warp 块执行的指令总数。完成所有不同的执行路径时，线程将重新汇聚到同一执行路径。

为了在控制流以线程 ID 为依据的情况下获得最佳性能，应编写控制条件，最小化分支 warp 块的数量。这是完全可行的，因为 warp 块在块内的分布情况是确定的，如 3.1 节所述。以一个简单的情况为例，当控制条件仅依赖于 (threadIdx / WSIZE) 时就是如此，其中的 WSIZE 是 warp 块的大小。此时不会出现任何 warp 块分支，因为控制条件与 warp 块完美对齐。

有些时候，控制器可能会展开循环，或通过使用转移猜测（branch predication）来优化 if 或 switch 语句，下文将加以介绍。在这些情况下，不会有任何 warp 块分支。程序员还可使用 #pragma unroll 指令控制循环的展开（请参见第 4.2.5.2 节）。

在使用转移猜测时，依靠控制条件执行的任何指令都不会被跳过。而是分别与一个每线程条件代码或根据控制条件设置为 true 或 false 的谓词（predicate）相关联，尽管每一条指令都为执行而进行了调度，但只有谓词为 true 的指令才会被实际执行。带有 false 谓词的指令不会写入结果，也不会计算地址或读取操作数。

只有在分支条件控制的指令数量小于或等于特定阈值时，编译器才会使用猜测的指令替换分支指令：如果编译器确定出有可能产生大量分支 warp 块的条件，则此阈值为 7，否则为 4。

5.1.1.3 存储器指令

存储器指令包含读取或写入共享、本地或全局存储器的任何指令。只有在使用某些自动变量时才会发生本地存储器访问，详见第 4.2.2.4 节。

多处理器要占用 4 个时钟周期来为一个 warp 块发出一条存储器指令。在访问本地或全局存储器时，还存在额外的 400 到 600 个时钟周期的存储器延迟。

举例说明，注意以下示例代码中的赋值运算符：

```
__shared__ float shared[32];
__device__ float device[32];
shared[threadIdx.x] = device[threadIdx.x];
```

它将占用 4 个时钟周期来发送全局存储器的读取指令、4 个时钟周期来发送共享存储器的写入指令，但总共需要 400 到 600 个时钟周期来从全局存储器读取一个浮点数。

如果在等待全局存储器访问完成时有足够的独立算数指令可发出，则线程调度程序可隐藏这样的全局存储器延迟。

5.1.1.4 同步指令

如果没有任何线程需要等待其他线程，__syncthreads 将占用 4 个时钟周期来为一个 warp 块发出指令。

5.1.2 存储器带宽

各存储器空间的有效带宽主要取决于存储器访问模式，下面几节将介绍相关内容。

由于设备存储器比片上存储器的延迟更高、带宽更低，因此应尽量减少设备存储器访问。典型的编程模式是将来自设备存储器的数据转入共享存储器，也就是说，使一个块的所有线程：

- 将来自设备存储器的数据载入共享存储器；
- 与块中的其他所有线程同步，使各线程可安全地读取由其他线程写入的共享存储器位置；
- 在共享存储器中处理数据；
- 若有必要，再次进行同步，确保使用结果更新了共享存储器；
- 将结果写回设备存储器。

5.1.2.1 全局存储器

全局存储器空间不会被缓存，因而采用正确的访问模式来实现最大化的存储器带宽尤为重要，考虑到设备存储器访问的高昂成本时更是如此。

首先，设备能够通过一条指令将全局存储器中的 32 位、64 位 或 128 位读入寄存器。对于下面这样的赋值：

```
__device__ type device[32];
type data = device[tid];
```

将其编译为一条加载指令，type 必须使 sizeof(type) 等于 4、8 或 16，类型为 type 的变量必须与 sizeof(type) 字节一致（也就是说，它的地址必须是 sizeof(type) 的倍数）。

对于 4.3.1.1 节介绍的内置类型，如 float2 或 float4，对齐要求可自动满足。

对于结构体来说，可通过使用对齐说明符 **__align__(8)** 或 **__align__(16)** 来使编译器实施大小和对齐要求，举例如下：

```
struct __align__(8) {
    float a;
    float b;
};
或：
struct __align__(16) {
    float a;
    float b;
    float c;
};
```

对于超过 16 个字节的结构体来说，编译器会生成多条加载指令。为了确保生成的指令数量最少，应使用 **__align__(16)** 定义此类结构体，举例如下：

```
struct __align__(16) {
float a;
float b;
float c;
float d;
float e;
};
```

这将编译为 2 条 128 位的加载指令，而非 5 条 32 位的加载指令。

位于全局存储器中的变量的地址，或由驱动程序或运行时 API 的存储器分配例程返回的变量地址总是会对齐到至少 256 个字节。

其次，当半 warp 块中的线程同时进行的存储器访问（在单独一条读取或写入指令执行的过程中）可合并（coalesce）为一个存储器事务（memory transaction）时，全局存储器带宽的使用效率将达到最高。存储器事务的大小可为 32 字节（仅针对计算能力为 1.2 或更高的设备）、64 位或 128 位。

本节的后续内容将介绍根据设备的计算能力合并存储器访问的各种需求。如果半 warp 块能够满足这些需求，即便在 warp 块分支、半 warp 块的某些线程并未实际访问存储器的情况下，也可实现合并。

在下面的讨论中，全局存储器将被视为分区成大小等于 32、64 或 128 字节的部分，并且对齐到此大小。

在计算能力为 1.0 和 1.1 的设备上进行存储器合并

如果满足以下三个条件，半 warp 的所有线程进行的全局存储器访问都会合并为一个或两个存储器事务：

- 线程必须访问
  - 32 位字，得到一个 64 字节的存储器事务，
  - 或 64 位字，得到一个 128 字节的存储器事务，
  - 或 128 位字，得到两个 128 字节的存储器事务；
- 全部 16 个字必须位于大小等于存储器事务大小的同一个存储器段（segment）中（在访问 128 位字的情况下，位于存储器事务大小两倍的分段中）；
- 线程必须按顺序访问字：半 warp 块中的第 k 个线程必须访问第 k 个字。

如果半 warp 块不满足上述所有需求，将为各线程发出一个独立的存储器事务，而吞吐量将显著降低。

图 5-1 显示了接合后的存储器访问的示例，而图 5-2 和图 5-3 显示了未为计算能力是 1.0 或 1.1 的设备进行存储器合并的存储器访问示例。

存储器合并后的 64 位访问带宽比存储器合并后的 32 位访问相比略低，而存储器合并后的 128 位访问则比存储器合并后的 32 位访问带宽低出许多。然而，同为 32 位访存时，尽管未存储器合并的访问的带宽比存储器合并后的访问带宽要低一个数量级，但同为 64 位情况下，前者带宽比后者低 4 倍，128 位情况下，前者带宽比后者低 2 倍。

在计算能力为 1.2 或更高的设备上进行存储器合并

只要所有半 warp 块的所有线程访问的字位于大小满足以下条件的同一个存储器段内，所有线程进行的全局存储器访问都会合并为一个或两个存储器事务：

- 如果所有线程都访问 8 位字，则为 32 字节；
- 如果所有线程都访问 16 位字，则为 64 字节；
- 如果所有线程都访问 32 位或 64 位字，则为 128 字节。

半 warp 块所请求的任何地址模式都会实现存储器合并，包括多个线程访问同一个地址的模式。这与具有较低计算能力的设备的情况截然不同，在那种情况下，线程需要串行地访问字。

如果半 warp 块在 n 个不同的存储器段内对字进行寻址，则发出 n 个存储器事务（每个事务针对一个存储器段），而计算能力较低的设备将在 n 大于 1 时发出 16 个事务。具体来说，如果线程访问 128 位字，则至少发出两个存储器事务。

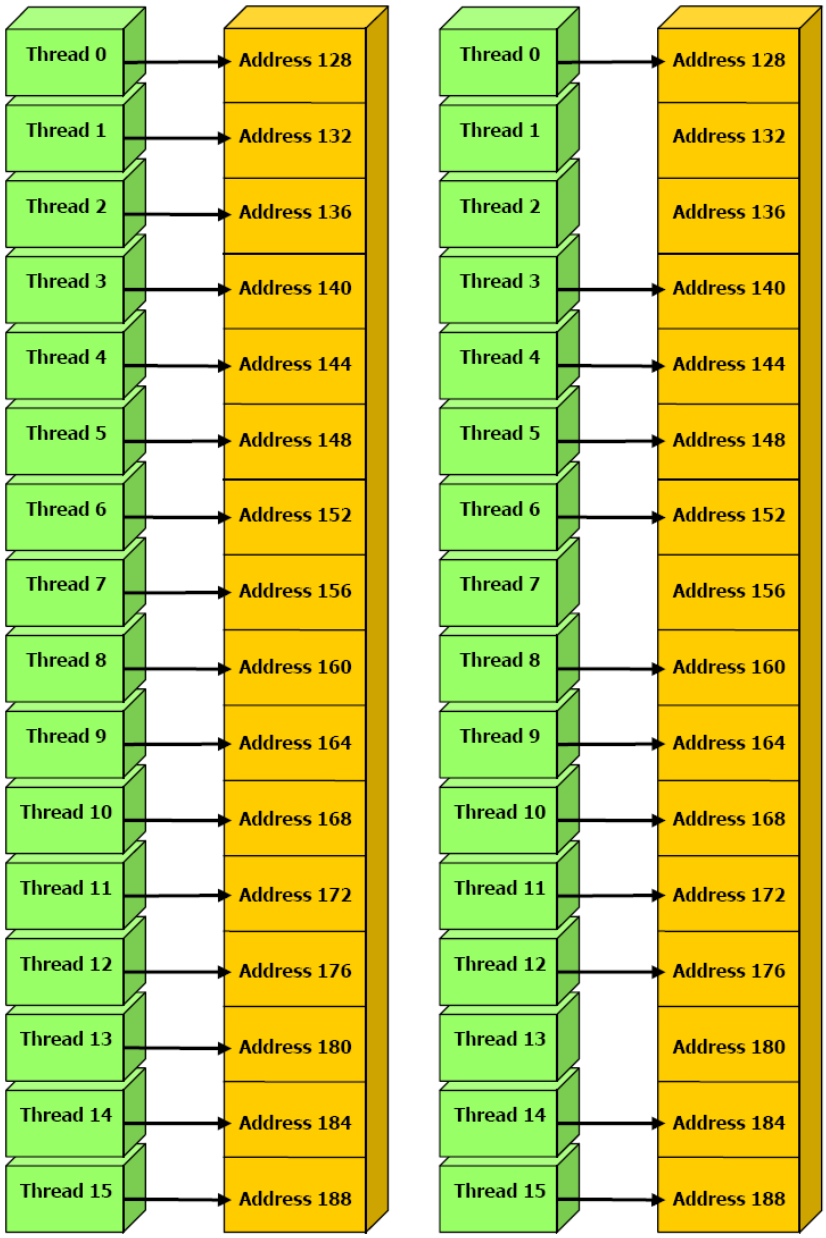
在存储器事务中，无用的字也会被读取，这将浪费带宽。为减少浪费，硬件将自动发出包含所请求的字的最小的存储器事务。举例来说，如果所请求的全部字都位于一个 128 字节分区的一半，则发出一个 64 位的事务。

更精确地来说，以下协议用于为半 warp 块发出存储器事务：

- 查找包含编号最小的活动线程所请求的地址的存储器段。对于 8 位数据来说，段大小是 32 字节，对于 16 位数据是 64 字节，对于 32 位、64 位和 128 位数据是 128 字节。
- 查找请求位于同一存储器段内的地址的其他所有活动线程。
- 尽可能减小事务的大小：
  - 如果事务大小为 128 字节，而且仅使用了下半或上半，则将事务大小缩减为 64 字节；

- 如果事务大小为 64 字节，而且仅使用了下半或上半，则将事务大小缩减为 32 字节。
- 执行事务，并将得到服务的线程标记为不活动（inactive）。
- 重复上述操作，直至为半 warp 块中的所有线程提供了服务。

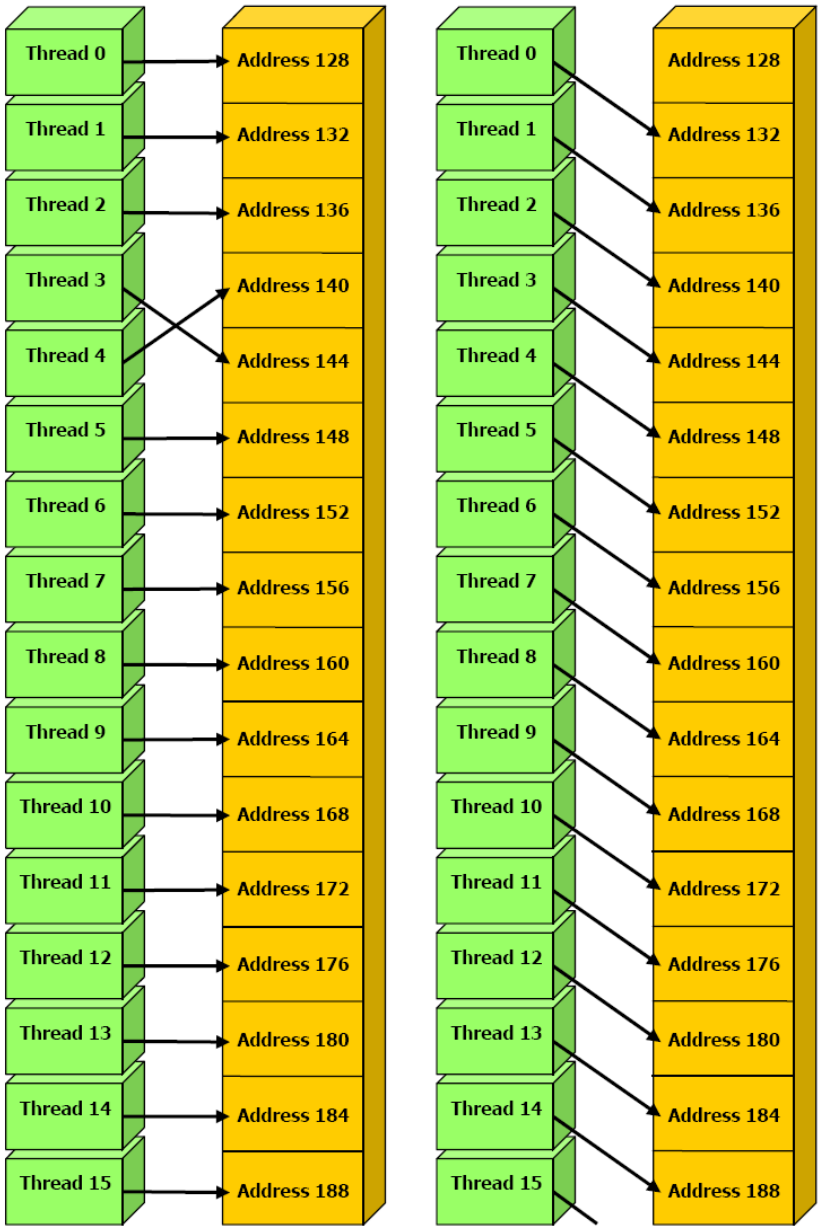
图 5-4 展示了计算能力为 1.2 或更高的设备的全局存储器访问示例。



左侧：存储器合并后的 float 存储器访问，得到一个存储器事务。

右侧：存储器合并后的 float 存储器访问（分支 warp），得到一个存储器事务。

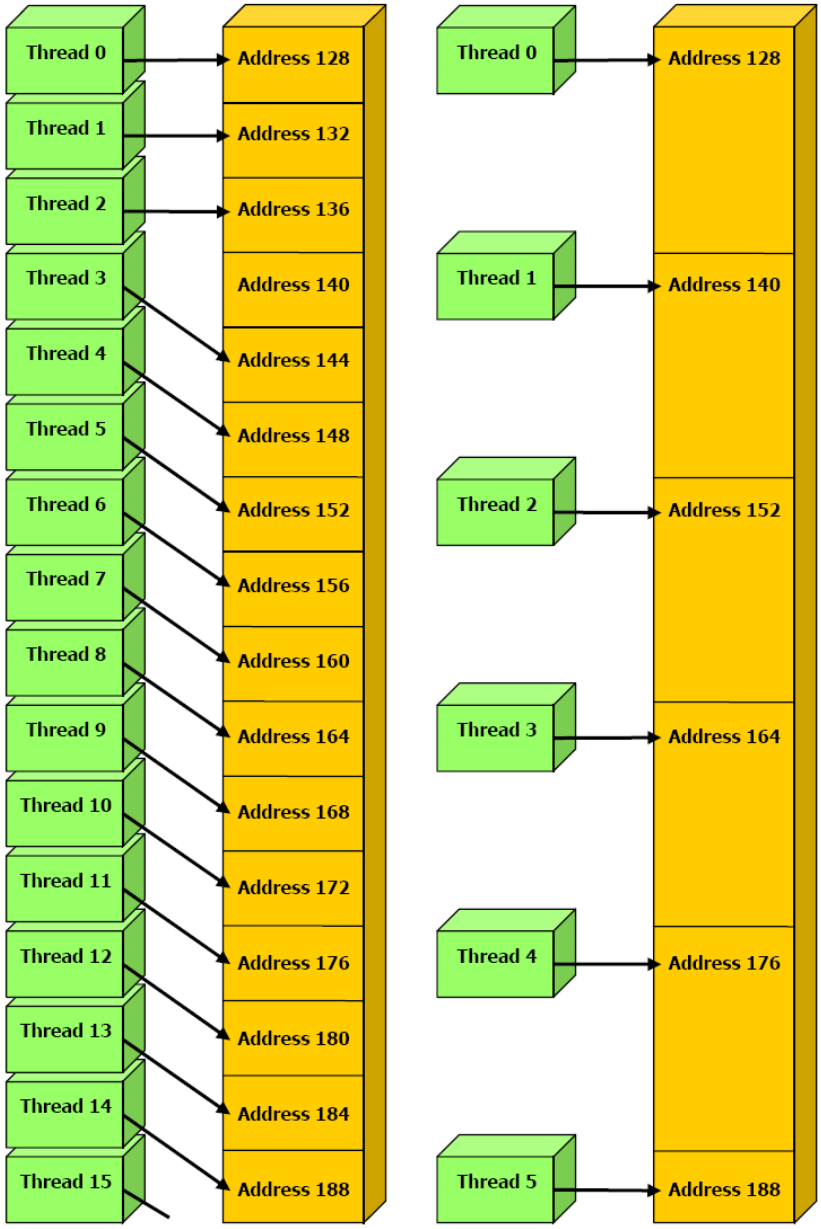
图 5-1. 存储器合并后的存储器访问模式示例



左侧：非连续的 float 存储器访问，得到 16 个存储器事务。

右侧：使用未对齐的起始地址，得到 16 个存储器事务

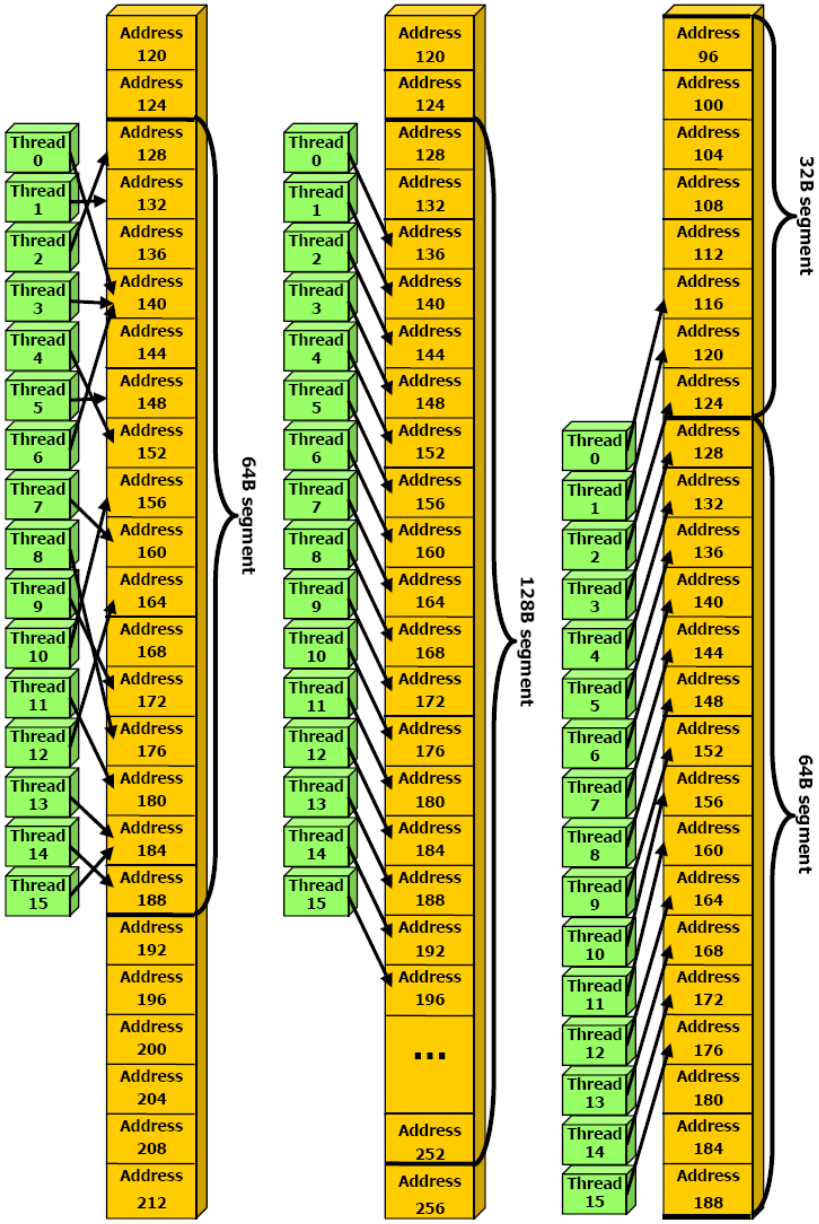
图 5-2. 未为计算能力是 1.0 或 1.1 的设备进行存储器合并的全局存储器访问模式示例



左侧：不相邻的 float 存储器访问，得到 16 个存储器事务。

右侧：不相邻的 float3 存储器访问，得到 16 个存储器事务。

图 5-3. 未为计算能力是 1.0 或 1.1 的设备进行存储器合并的全局存储器访问模式示例



左侧：在一个 64 字节的存储器段内的随机 float 存储器访问，得到一个存储器事务。

中间：未对齐的 float 存储器访问，得到一个事务。

右侧：未对齐的 float 存储器访问，得到两个事务。

图 5-4. 计算能力为 1.2 或更高的设备的全局存储器访问示例

一般访问模式

全局存储器的一般访问模式（Common Access Pattern）是指：线程 ID 为 tid 的每个线程使用以下地址访问类型为 type*、位于地址 BaseAddress 处的一个数组的一个元素：

```
BaseAddress + tid
```

为了实现存储器合并，type 必须满足上文介绍的大小和对齐要求。具体来说，这也就意味着，如果 type 是一个大于 16 字节的结构体，就应将其分割为多个满足这些需求的结构体，而数据应作为此类结构体的多个数组存储于存储器之中，而不是单独一个 type* 类型的数组。

另外一种全局存储器一般访问模式是索引为 (tx, ty) 的各线程使用以下地址访问类型为 type*、位于地址 BaseAddress 处、宽度为 width 的二维数组中的一个元素：

```
BaseAddress + width * ty + tx
```

在这种情况下，只有满足了以下条件，才能为线程块的所有半 warp 块实现存储器合并：

- 线程块的宽度是半 warp 块大小的倍数；
- Width 是 16 的倍数。

具体来说，这也就意味着，宽度不是 16 的倍数的数组在分配时的宽度越接近 16 的倍数，并且行得到了相应的填充（padding），那么对于该数组的访问就越有效率。参考手册中介绍的 cudaMallocPitch() 和 cuMemAllocPitch() 函数和相关的存储器复制函数使程序员能够编写不依赖于硬件的代码，以分配符合这些限制条件的数组。

5.1.2.2 本地存储器

本地存储器（local memory）访问仅由某些自动变量分配，如第 4.2.2.4 节所述。与全局存储器空间相似，本地存储器空间不会被缓存，因此本地存储器的访问成本与全局存储器一样高。但由于它们在定义上是基于每个线程的，因而访问总是会合并。

5.1.2.3 常量存储器

常量存储器空间会被缓存，因此常量存储器的读取仅需在缓存丢失时读取一次设备存储器，否则只需读取常量缓存即可。

对于半 warp 块的所有线程来说，只要所有线程都读取同一个地址，从常量缓存读取的速度就与读取寄存器的速度一样快。成本随着所有线程读取的不同地址的数量增加而线性增加。我们建议，使整个 warp 块的所有线程都读取相同的地址，而不仅仅是保证各半 warp 块中的所有线程都读取相同的地址，因为未来的设备可能要求全速读取。

5.1.2.4 纹理存储器

纹理存储器空间会被缓存，因此纹理拾取仅需在缓存丢失时读取一次设备存储器，否则只需读取纹理缓存即可。纹理缓存已为二维空间位置而优化，因此读取相邻纹理地址的同一个 warp 块的线程将实现最高性能。此外，它设计用于以固定的延迟执行流式拾取（streaming fetch），也就是说，一次缓存命中将减少 DRAM 带宽要求，但不涉及拾取延迟。

与从全局存储器或常量存储器读取设备存储器的方法相比，通过纹理拾取读取设备存储器可能是一种更有优势的替代方法，详见第 5.4 节。

5.1.2.5 共享存储器

由于共享存储器位于芯片上，因而共享存储器空间比本地和全局存储器空间的速度都要快得多。实际上，对于 warp 块中的所有线程来说，只要线程间不存在存储体冲突（bank conflict），访问共享存储器的速度就与访问寄存器一样快，下面将详细介绍相关内容。

为了获得较高的存储器带宽，共享存储器被划分为多个大小相等的存储器模块，称为存储体（bank），这些存储体可同时访问。因此，对落入 n 个不同存储体的 n 个地址的任何存储器读取或写入请求都可同时实现，得到更高效的带宽，可达到单独一个模块的带宽的 n 倍。

但若一个存储器请求的两个地址落入同一个存储体内，就会出现存储体冲突，访问必须串行化。硬件会在必要时将存在存储体冲突的存储器请求分割为多个不冲突的请求，此时有效带宽将降低为原带宽除以分离后的存储器请求的数量。如果分离后的存储器请求数量为 n，就可以说初始存储器请求导致了 n 路

（n-way）存储体冲突。

为了获得最大化的性能，有必要理解存储器地址如何映射到存储体以调度存储器请求，以最小化存储体冲突。

对于共享存储器空间，存储体采用了这样一种组织方式：为连续的存储体分配连续的 32 位字，每个存储体的带宽都是 32 位/2 个时钟周期。

对于计算能力为 1.x 的设备，warp 块的大小是 32，存储体的数量为 16（参见第 5.1 节）；warp 块的共享存储器请求将分割为一个针对 warp 块上半部分的请求和一个针对 warp 块下半部分的请求。因而，属于 warp 块第一部分的线程和属于 warp 块第二部分的线程之间不可能出现存储体冲突。

一种常见的情况就是各线程访问数组中的 32 位字，使用线程 ID tid 进行索引，步幅（stride）为 s：

```
__shared__ float shared[32];
float data = shared[BaseIndex + s * tid];
```

在本例中，只要 s*n 是存储体 m 的倍数，或者说只要 n 是 m/d 的倍数（其中 d 是 m 和 s 的最大公约数），线程 tid 和 tid+n 访问的就是同一个存储体。因而，只有在 warp 块的一半大小小于等于 m/d 时，才不会存在存储体冲突。对于计算能力是 1.x 的设备，可以说只有在 d 等于 1 时，或者说只有在 s 是奇数时，才不会存在存储体冲突，因为 m 是 2 的幂。

图 5-5 和图 5-6 展示了无冲突存储器访问的示例，图 5-7 展示了导致存储体冲突的存储器访问示例。

其他值得注意的情况还包括在各线程访问小于或大于 32 位的元素时。举例来说，如果通过以下方式访问 char 数组，则将出现存储体冲突：

```
__shared__ char shared[32];
char data = shared[BaseIndex + tid];
```

举例来说，由于 **shared[0]**、**shared[1]**、**shared[2]** 和 **shared[3]** 属于同一个存储体。因此不存在存储体冲突，但若通过以下方式访问同一个数组：

```
char data = shared[BaseIndex + 4 * tid];
```

double 数组将存在 2 路存储体冲突：

```
__shared__ double shared[32];
double data = shared[BaseIndex + tid];
```

由于存储器请求将编译为两个独立的 32 位请求。在本例中避免存储体冲突的方法之一就是将 double 操作数分割为两部分，如以下示例代码所示：

```
__shared__ int shared_lo[32];
__shared__ int shared_hi[32];
double dataIn;
shared_lo[BaseIndex + tid] = __double2loint(dataIn);
shared_hi[BaseIndex + tid] = __double2hiint(dataIn);
double dataOut =
                __hiloint2double(shared_hi[BaseIndex + tid],
                                 shared_lo[BaseIndex + tid]);
```

但这种做法并非总是能够提高性能，在未来的架构中可能表现更差。

结构体赋值将在必要时编译为针对结构体中各成员的多个存储器请求，因此，以下代码：

```
__shared__ struct type shared[32];
struct type data = shared[BaseIndex + tid];
```

将得到以下结果：

- 如果 type 定义如下，则进行三次无存储体冲突的存储器读取：

```
struct type {
   float x, y, z;
};
```

  这是因为每个成员都是使用三个 32 位字作为步幅访问的。

- 如果 type 定义如下，则进行两次有存储体冲突的存储器读取：

```
struct type {
   float x, y;
};
```

  这是因为每个成员都是使用两个 32 位字作为步幅访问的。

- 如果 type 定义如下，则进行两次有存储体冲突的存储器读取：

```
struct type {
```

```
    float f;
    char c;
};
```

这是因为每个成员都是使用 5 个字节作为步幅访问的。

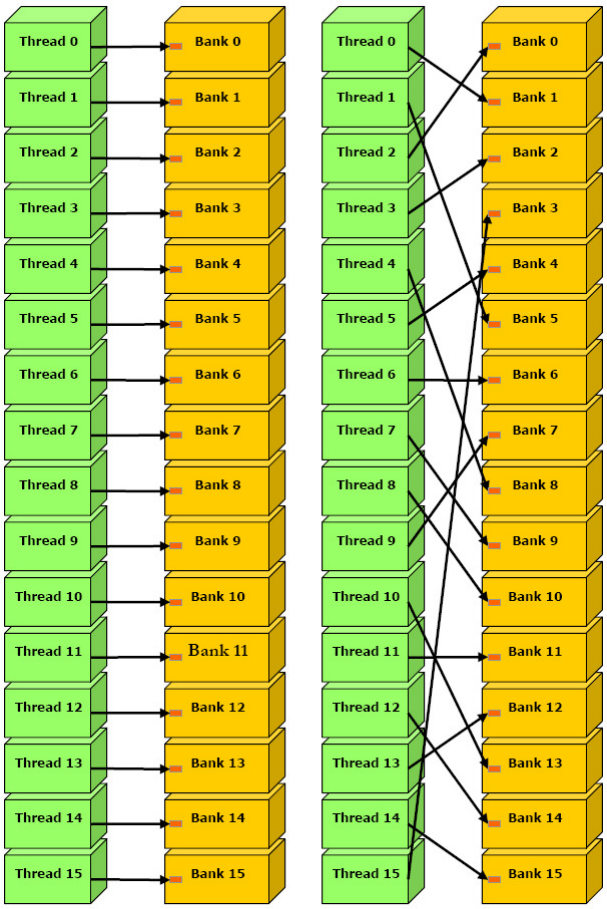
最终，共享存储器也会运用一种广播（broadcas）机制，从而能够在对一个 32 位字的存储器读取请求提供服务时同步将其读取并广播给多个线程。在半 warp 块的多个线程读取同一个 32 位字内的地址时，这会减少存储体冲突的数量。更精确地来说，包含多个地址的存储器读取请求是通过几个步骤逐渐完成的——每两个时钟周期一步（step），每步为一个无存储体冲突的地址子集（subset）提供服务，直至为所有地址提供服务；在每个步骤中都通过以下步骤在尚未被服务的地址中建立子集：

- 选择剩余地址指向的一个字作为广播字；
- 子集中包括：
  - 广播字内的所有地址；
  - 剩余地址指向的各存储体的地址。

不指定在每个周期选择哪个字作为广播字以及为各存储体选择哪个地址。

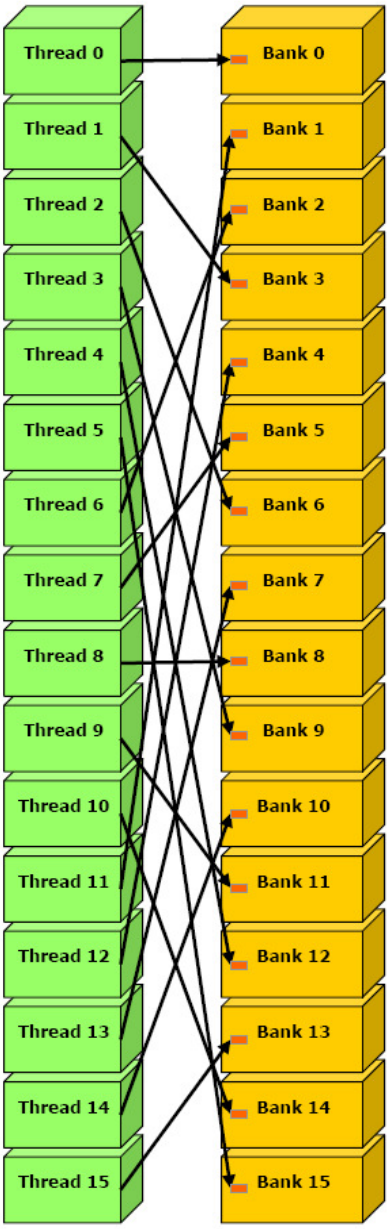
一种常见的无冲突情况就是半 warp 块中的所有线程都读取同一个 32 位字内的地址。

图 5-8 展示了包括广播机制在内的存储器读取访问的示例。



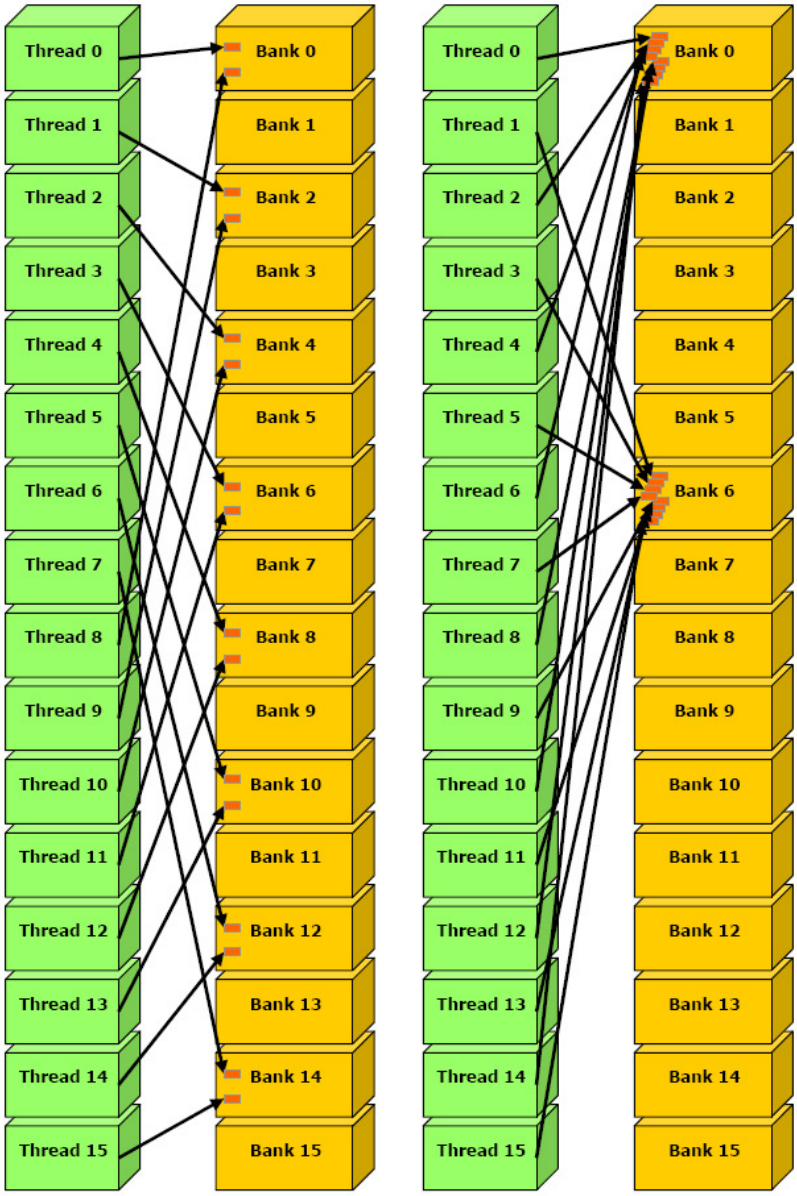
左侧：步幅为一个 32 位字的线性寻址。
右侧：随机排列（random permutation）。

图 5-5. 无存储体冲突的共享存储器访问模式示例



步幅为三个 32 位字的线性寻址。

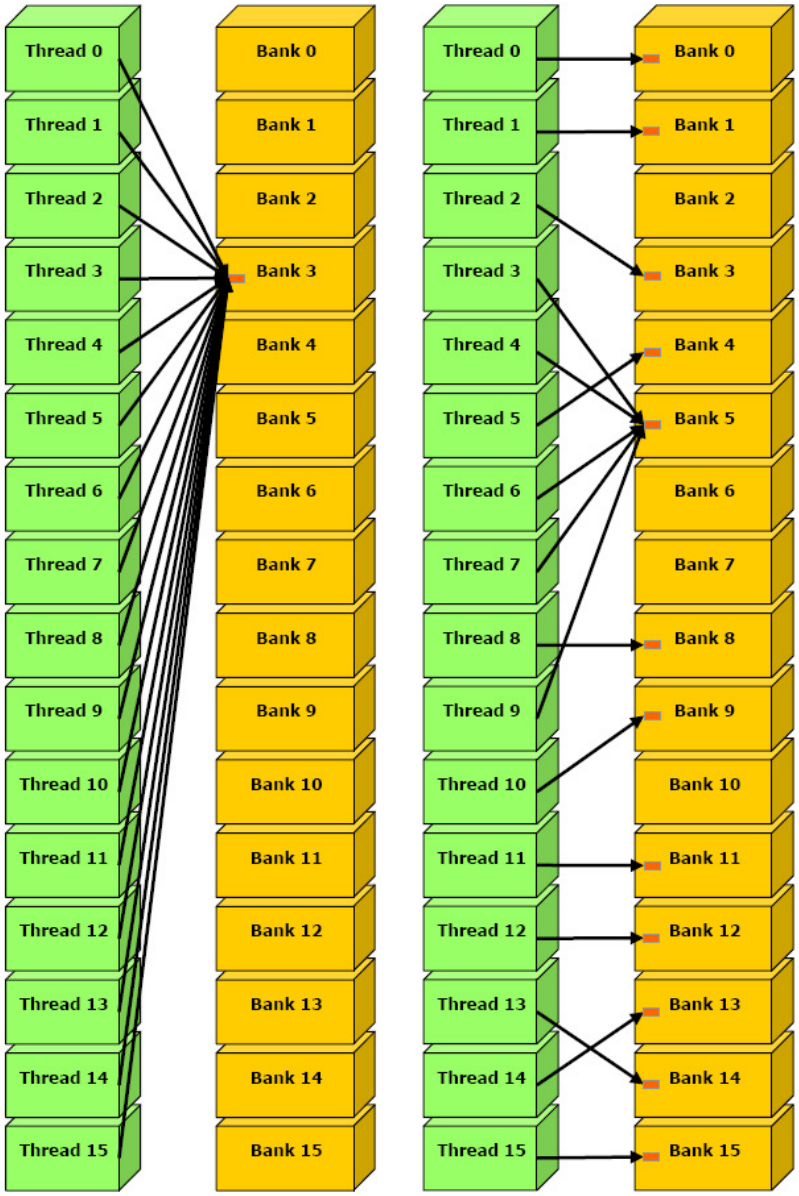
图 5-6. 无存储体冲突的共享存储器访问模式示例



左侧：步幅为 2 个 32 位字的线性寻址将导致二路存储体冲突。

右侧：步幅为 8 个 32 位字的线性寻址将导致八路存储体冲突。

图 5-7. 有存储体冲突的共享存储器访问模式示例



左侧：这种访问模式不存在冲突，因为所有线程都读取同一个 32 位字内的地址。

右侧：这种访问模式要么不会导致存储体冲突（如果存储体 5 内的字在第一个步中被选作广播字），要么会导致二路存储体冲突。

图 5-8. 使用广播机制的共享存储器读取访问模式示例

5.1.2.6 寄存器

一般而言，访问寄存器不会给每条指令带来额外的时钟周期，但寄存器的写入后读取依赖关系和寄存器的存储体冲突可能会导致延迟。

如果每个多处理器至少有 192 个活动线程，写入后读取依赖关系带来的延迟可忽略不计。

编译器和线程调度程序将尽可能以最优方式调度指令，避免出现寄存器的存储体冲突。当每个块的线程数量是 64 的倍数时，这种方法的效果最优。除了遵从此规则之外，应用程序无法直接控制这些存储体冲突。具体来说，无需将数据转换成 float4 或 int4 类型。

5.2 每个块的线程数量

给定每个网格的线程总数、每个块的线程总数（网格包含的块数），应选择最大化可用计算资源的利用率。这意味着块的数量至少应与设备中的多处理器数量相同。

此外，每个多处理器仅运行一个块将使多处理器在线程同步过程中出现空闲，如果每个块中没有足够的线程可容纳负载延迟，则在设备存储器读取过程中也会出现空闲。因而，最好允许在每个多处理器上存在两个或多个活动块，从而允许等待的块与运行的块间存在重叠。为此，块的数量至少应为设备中多处理器数量的二倍，而且为每个块所分配的共享存储器也至少应为每个多处理器可用的共享存储器总量的一半（请参见第 3.1 节）。如果有更多的线程块以流水线的形式通过设备，则开销将进一步降低。

如果希望网格有能力扩展到未来的设备，每个网格的块数量至少应为 100；1000 个块将能够扩展到此后数代的设备上。

在块的数量足够多后，每个块的线程数量应为 warp 块大小的倍数，避免因 warp 块填充不足而造成的计算资源浪费，最好使线程数量是 64 的倍数，详见第 5.1.2.6 节所述。

为每个块分配更多线程可实现有效的时间分割，但每个块的线程越多，每个线程可用的寄存器就越少。如果内核编译时使用了超过执行配置允许数量的寄存器，这可能会妨碍内核调用。

对于计算能力为 1.x 的设备来说，每个线程可用的寄存器数量等于：

$$\frac{R}{B \times ceil(T,32)}$$

其中 R 是附录 A 中给出的各多处理器的寄存器总数，B 是每个多处理器的活动块数量，T 是每个块的线程数，ceil(T, 32) 是将 T 四舍五入为最接近的 32 的倍数。

如果在编译时使用 –ptxas-options=-v 选项，编译器将报告内核编译所使用的寄存器数量（以及本地、共享和常量存储器使用情况）。请注意，每个 double 或 long long 变量都为本地支持这些类型的设备使用两个寄存器，也就是计算能力为 1.2 或更高的设备对应于 long long，计算能力为 1.3 或更高的设备对应于 double。但与计算能力较低的设备相比，计算能力为 1.2 或更高的设备的每个多处理器的寄存器数量都是计算能力低于 1.2 的设备的二倍。

每个块最少有 64 个线程，这仅在每个多处理器有多个活动块时才是有意义的。每个块 192 或 256 个线程是更理想的情况，允许使用足够的寄存器进行编译。

每个多处理器活动 warp 块数量与活动 warp 块最大数量（如附录 A 所示）的比率称为多处理器占用率。为最大化占用率，编译器会尝试最小化寄存器的使用，程序员需要谨慎选择执行配置。CUDA Software Development Kit 提供了一份电子表格，可帮助程序员根据共享存储器和寄存器要求选择线程块的大小。

5.3 宿主和设备间的数据传输

设备和设备存储器之间的带宽比设备存储器和宿主存储器之间的带宽高得多。因而，应尽力最小化宿主和设备之间的数据传输，例如，将更多代码从宿主迁移到设备——即便这意味着以并行性更低的计算方式运行内核。设备存储器中可能会创建中间数据结构，由设备操作，而且在销毁时不会被宿主映射，也不会复制到宿主存储器。

此外，考虑到与每次传输相关的开销，将多个较小的传输操作整合为一次较大的成批传输操作的性能总是优于分别进行每一次传输。

最后，在使用分页锁定存储器时，宿主和设备间的带宽更高，如第 4.5.1.2 节所述。

5.4 纹理拾取与全局或常量存储器读取的对比

与读取全局或常量存储器相比，通过纹理拾取实现设备存储器读取有着一些优势：

- 它们是缓存的，因而假如纹理拾取中存在数据局部性（locality），纹理拾取将表现出更高的带宽；
- 它们不受制于存储器访问模式的限制，而全局或常量存储器读取必须遵从此类限制以获得较好的性能（请参见第 5.1.2.1 节和 5.1.2.3 节）；
- 寻址计算的延迟得到了更好的隐藏，可能改进执行随机数据访问的应用程序的性能；
- 打包的数据可通过一次操作广播给不同的变量；
- 8 位和 16 位整型输入数据可转换为 [0.0, 1.0] 或 [-1.0, 1.0] 区间内的 32 位浮点值（请参见第 4.3.4.1 节）。

如果纹理是一个 CUDA 数组（请参见第 4.3.4.2 节），硬件提供的其他功能可能对不同的应用程序非常有用，特别是图像处理：

特性	适用场景	提示
过滤	Texel 之间的快速、低精度的插值	仅在纹理参考返回浮点数据时有效
归一化的纹理坐标	独立于分辨率的编码	
寻址模式	自动处理边界情况	仅适用于归一化的纹理坐标

但在同一个内核调用中，纹理缓存不会在全局存储器写入方面保持连贯性，因此如果纹理拾取针对的是通过同一内核调用的全局写入操作写入的地址，则将返回不确定的数据。换句话说，仅在存储器位置已由之前的内核调用或存储器复制更新时，线程才能通过纹理安全读取某些存储器位置，但如果已由同一内核调用的同一线程或其他线程更新，则并非如此。二者仅在使用无法写入 CUDA 数组的内核从线性存储器拾取时才有所相关。

5.5 整体性能优化策略

性能优化是以三项基本策略为基础演化出来的：

- 最大化并行执行；
- 优化存储器利用率，实现最大化的存储器带宽；
- 优化指令利用率，实现最大化的指令吞吐量。

最大化并行执行的第一步就是结构化算法，尽可能地寻找到更多的数据并行性。当算法的并行性因某些线程需要同步以彼此共享数据而被打破时，可能出现两种情况：这些线程可能属于同一个块，此时应使用__syncthreads()，在同一个内核调用中同步共享存储器共享数据；如果属于不同的块，此时必须使用两个独立的内核调用通过全局存储器共享数据，一个用于写入全局存储器，一个用于读取全局存储器。

一旦算法的并行性被找到，就需要尽可能有效地在硬件实现。这是通过谨慎选择各内核调用的执行配置完成的，详细内容请参见第 5.2 节。

应用程序还应在较高的层面最大化并行执行，通过流在设备上显式寻找并发执行，如第 4.5.1.5 节所述，应用程序还应最大化宿主和设备间的并发执行。

优化存储器利用率的第一步是最小化低带宽的数据传输。这也就意味着最小化宿主和设备之间的数据传输，如第 5.3 节所述，因为此类传输比设备和全局存储器之间的数据传输带宽低得多。这也意味着通过最大化设备共享存储器的使用来尽可能地减少设备和全局存储器之间的数据传输，如第 5.1.2 节所述。有时，最佳优化方法可能是首先避免进行任何数据传输，直接在需要的位置重新计算数据。

如第 5.1.2.1、5.1.2.3、5.1.2.4 和 5.1.2.5 节所述，根据各类存储器的访问模式的不同，有效带宽可能会出现数量级的变化。优化存储器利用率的下一步骤是根据最优的存储器访问模式，尽可能以最优方式组织存

储器访问。这种优化对于全局存储器访问尤为重要，因为全局存储器的带宽较低，其延迟可能达到数百个时钟周期（请参见第 5.1.1.3 节 ）。另一方面，通常只有在共享存储器访问中存在严重的存储体冲突时才进行共享存储器访问优化。

对于指令利用率的优化，应最小化低吞吐量的数学指令的使用（请参见第 5.1.1.1 节）。这包括权衡计算精度，在不影响最终结果的前提下提高速度，例如使用内建函数而非一般函数（B.2 节列举了内建函数）、使用单精度而非双精度。由于设备的 SIMT 特性，应特别注意控制流指令，如第 5.1.1.2 节所述。

第 6 章 矩阵乘法示例

6.1 概述

计算维度分别为 *(wA, hA)* 和 *(wB, wA)* 的矩阵 A 和矩阵 B 的乘积 C，此任务将以如下方法划分为几个线程：

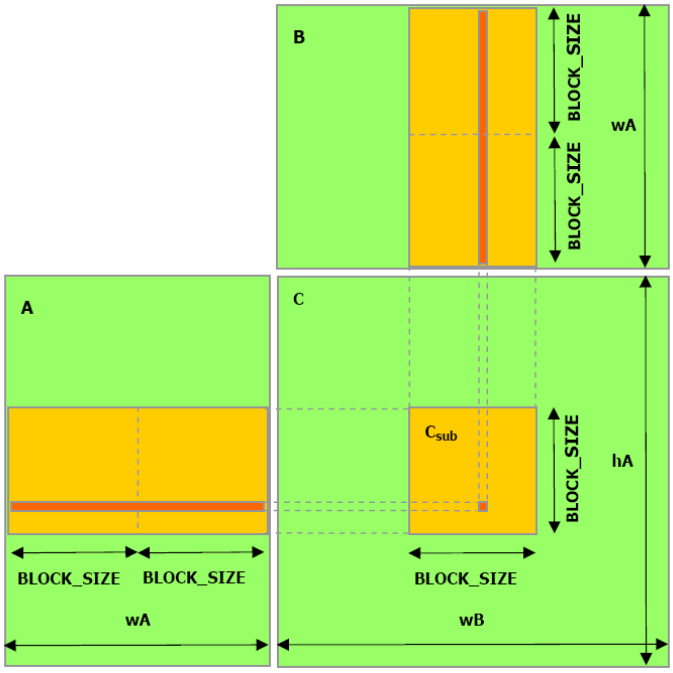
- 每个线程块均负责计算 C 的一个子方阵 C_{sub}；
- 块内的每个线程负责计算 C_{sub} 的一个元素。

C_{sub} 的维度 block_size 选为 16，使每个块的线程数是 warp 块大小的倍数（请参见第 5.2 节），同时保证不超过每个块的最大线程数（附录 A）。

如图 6-1 所示，C_{sub} 等于以下两个长方矩阵的乘积：维度为 *(wA, block_size)* 的 A 的子阵，它的行索引与 C_{sub} 相同；维度为 *(block_size, wA)* 的 B 的子阵，它与 C_{sub} 具有相同的列索引。为了适应设备的资源，这两个长方矩阵被分割为多个维度为 block_size 的方阵，通过计算这些方阵的乘积和来计算 C_{sub}。每个乘积首先都从全局存储器将两个对应的方阵载入共享存储器，使用一个线程载入各矩阵的一个元素，然后由各线程来计算乘积的一个元素。各线程将所有这些乘积的结果汇总到一个寄存器中，完成后，将结果写入全局存储器。

通过这种方法安排计算，我们就利用了速度较快的共享存储器，节省了大量全局存储器带宽，这是因为从全局存储器中读取 A 和 B 的次数仅为 *(wA / block_size)*。

而在编写本示例时，我们仅以展示各种 CUDA 编程原则为目标，所以在实际应用中，请探索比本示例性能更高的通用矩阵乘法实现。



每个线程块计算 C 的一个子矩阵 C_{sub}。块内的每个线程计算 C_{sub} 的一个元素。

图 6-1. 矩阵乘法

6.2 源代码清单

```
// Thread block size
#define BLOCK_SIZE 16
// Forward declaration of the device multiplication function
__global__ void Muld(float*, float*, int, int, float*);
// Host multiplication function
// Compute C = A * B
//   hA is the height of A
//   wA is the width of A
//   wB is the width of B
void Mul(const float* A, const float* B, int hA, int wA, int wB,
         float* C)
{
    int size;
    // Load A and B to the device
    float* Ad;
    size = hA * wA * sizeof(float);
    cudaMalloc((void**)&Ad, size);
    cudaMemcpy(Ad, A, size, cudaMemcpyHostToDevice);
    float* Bd;
    size = wA * wB * sizeof(float);
    cudaMalloc((void**)&Bd, size);
    cudaMemcpy(Bd, B, size, cudaMemcpyHostToDevice);
    // Allocate C on the device
    float* Cd;
    size = hA * wB * sizeof(float);
    cudaMalloc((void**)&Cd, size);
    // Compute the execution configuration assuming
    // the matrix dimensions are multiples of BLOCK_SIZE
    dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
    dim3 dimGrid(wB / dimBlock.x, hA / dimBlock.y);
    // Launch the device computation
    Muld<<<dimGrid, dimBlock>>>(Ad, Bd, wA, wB, Cd);
    // Read C from the device
    cudaMemcpy(C, Cd, size, cudaMemcpyDeviceToHost);
    // Free device memory
    cudaFree(Ad);
    cudaFree(Bd);
    cudaFree(Cd);
}
// Device multiplication function called by Mul()
// Compute C = A * B
// wA is the width of A
// wB is the width of B
__global__ void Muld(float* A, float* B, int wA, int wB, float* C)
{
    // Block index
    int bx = blockIdx.x;
    int by = blockIdx.y;
    // Thread index
    int tx = threadIdx.x;
    int ty = threadIdx.y;
    // Index of the first sub-matrix of A processed by the block
    int aBegin = wA * BLOCK_SIZE * by;
    // Index of the last sub-matrix of A processed by the block
    int aEnd = aBegin + wA - 1;
    // Step size used to iterate through the sub-matrices of A
    int aStep = BLOCK_SIZE;
    // Index of the first sub-matrix of B processed by the block
    int bBegin = BLOCK_SIZE * bx;
    // Step size used to iterate through the sub-matrices of B
    int bStep = BLOCK_SIZE * wB;
    // The element of the block sub-matrix that is computed
    // by the thread
    float Csub = 0;
    // Loop over all the sub-matrices of A and B required to
    // compute the block sub-matrix
```

```
    for (int a = aBegin, b = bBegin;
             a <= aEnd;
             a += aStep, b += bStep) {
        // Shared memory for the sub-matrix of A
        __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
        // Shared memory for the sub-matrix of B
        __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
        // Load the matrices from global memory to shared memory;
        // each thread loads one element of each matrix
        As[ty][tx] = A[a + wA * ty + tx];
        Bs[ty][tx] = B[b + wB * ty + tx];
        // Synchronize to make sure the matrices are loaded
        __syncthreads();
        // Multiply the two matrices together;
        // each thread computes one element
        // of the block sub-matrix
        for (int k = 0; k < BLOCK_SIZE; ++k)
        Csub += As[ty][k] * Bs[k][tx];
        // Synchronize to make sure that the preceding
        // computation is done before loading two new
        // sub-matrices of A and B in the next iteration
        __syncthreads();
    }
    // Write the block sub-matrix to global memory;
    // each thread writes one element
    int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
    C[c + wB * ty + tx] = Csub;
}
```

6.3 源代码说明

源代码包含两个函数：

- Mul()，一个宿主函数，用作 Muld() 的包装器（wrapper）；
- Muld()，一个内核，在设备上执行矩阵乘法。

6.3.1 Mul()

Mul() 可接受的输入：

- 指向 A 和 B 的元素的两个宿主存储器指针；
- A 的高度和宽度以及 B 的宽度；
- 指向 C 写入位置的宿主存储器指针。

Mul() 执行以下操作：

- 使用 cudaMalloc() 分配足够的全局存储器来存储 A、B 和 C；
- 使用 cudaMemcpy() 将 A 和 B 从宿主存储器复制到全局存储器；
- 调用 Muld() 在设备上计算 C；
- 使用 cudaMemcpy() 将 C 从全局存储器复制到宿主存储器；
- 使用 cudaFree() 释放为 A、B 和 C 分配的全局存储器。

6.3.2 Muld()

Muld() 可接受的输入与 Mul() 相同，但指针指向设备存储器，而非宿主存储器。

对于每一个块，Muld() 将循环 A 和 B 中计算 C_{sub} 所需的全部子阵。在每次循环中：

- 将一个 A 的子阵和一个 B 的子阵从全局存储器载入共享存储器；
- 同步以确保两个子阵均已被块内的所有线程完全载入；

- 计算两个子阵的乘积，并将其与上一次循环获得的乘积相加；
- 再次进行同步，以确保两个子阵的乘积运算已完成，之后才开始下一次循环。

处理所有子阵之后，C_{sub} 计算完成，Muld() 会将结果写入全局存储器。

Muld() 的编写目的是为了最大化存储器性能，请参见第 5.1.2.1 节和第 5.1.2.5 节。

实际上，假设 wA 和 wB 是 16 的倍数，如第 5.1.2.1 节所述，此时将能够确保全局存储器合并，因为 a、b 和 c 都是 BLOCK_SIZE（等于 16）的倍数。

这里也不存在任何存储体冲突，因为在每个半 warp 块中，ty 和 k 对于所有线程都是相同的，tx 在 0 至 15 之间，因此每个线程在进行 **`As[ty][tx]`**、**`Bs[ty][tx]`** 和 **`Bs[k][tx]`** 存储器访问时都会访问不同的存储体，而在进行 **`As[ty][k]`** 存储器访问时访问相同的存储体。

附录 A 技术规范

A.1 一般规范

计算设备的一般规范取决于其计算能力（请参见第 2.5 节）。

下面的内容描述了与各种计算能力相关联的技术规范和特性。若无其他说明，给定计算能力的规范与略低计算能力的规范相同。类似地，计算能力向下兼容。

下表列出了所有支持 CUDA 的设备的多处理器数量和计算能力：

	多处理器数量	计算能力
GeForce GTX 280	30	1.3
GeForce GTX 260	24	1.3
GeForce 9800 GX2	2×16	1.1
GeForce 9800 GTX	16	1.1
GeForce 8800 Ultra, 8800 GTX	16	1.0
GeForce 8800 GT	14	1.1
GeForce 9600 GSO, 8800 GS, 8800M GTX	12	1.1
GeForce 8800 GTS	12	1.0
GeForce 9600 GT, 8800M GTS	8	1.1
GeForce 9500 GT, 8600 GTS, 8600 GT, 8700M GT, 8600M GT, 8600M GS	4	1.1
GeForce 8500 GT, 8400 GS, 8400M GT, 8400M GS	2	1.1
GeForce 8400M G	1	1.1
Tesla S1070	4×30	1.3
Tesla C1060	30	1.3
Tesla S870	4×16	1.0
Tesla D870	2×16	1.0
Tesla C870	16	1.0
Quadro Plex 1000 Model S4	4×16	1.0
Quadro Plex 1000 Model IV	2×16	1.0
Quadro FX 5600	16	1.0
Quadro FX 3700	14	1.1
Quadro FX 3600M	12	1.1
Quadro FX 4600	12	1.0
Quadro FX 1700, FX 570, NVS 320M, FX 1600M, FX 570M	4	1.1
Quadro FX 370, NVS 290, NVS 140M, NVS 135M, FX 360M	2	1.1
Quadro NVS 130M	1	1.1

可使用运行时查询时钟频率和设备存储器总量（请参见第 4.5.2.2 节和第 4.5.3.2 节）。

A.1.1 计算能力 1.0 的规范

每个块的最大线程数为 512；

一个线程块的 x、y 和 z 维的最大规格分别为 512、512 和 64；

线程块网格各维度的最大规格为 65535；

- Warp 块的大小是 32 个线程；
- 每个多处理器的寄存器数量是 8192；
- 每个多处理器可用的共享存储器数量是 16KB，组织为 16 个存储体（请参见第 5.1.2.5 节）；
- 常量存储器的总量是 64KB；
- 常量存储器的缓存工作区是每个多处理器 8KB；
- 纹理存储器的缓存工作区介于每个多处理器 6 到 8KB 之间；
- 每个多处理器的最大活动块数是 8；
- 每个多处理器的最大活动 warp 块数是 24；
- 每个多处理器的最大活动线程数是 768；
- 对于绑定到一维 CUDA 数组的纹理参考，最大宽度为 2^{13}；
- 对于绑定到二维 CUDA 数组的纹理参考，最大宽度为 2^{16}，最大高度为 2^{15}；
- 对于绑定到三维 CUDA 数组的纹理参考，最大宽度为 2^{11}，最大高度为 2^{11}，最大深度为 2^{11}；
- 对于绑定到线性存储器的纹理参考，最大宽度为 2^{27}；
- 内核大小限制为 200 万 PTX 指令；
- 每个多处理器都由 8 个处理器组成，因此一个多处理器能够在 4 个时钟周期内处理一个 warp 块的 32 个线程。

A.1.2 计算能力 1.1 的规范

- 支持在全局存储器的 32 位字上操作的原子函数（请参见第 4.4.4 节）。

A.1.3 计算能力 1.2 的规范

支持在共享存储器中操作的原子函数以及在全局存储器的 64 位字上操作的原子函数（请参见第 4.4.4 节）；

- 支持 warp vote 函数（请参见第 4.4.5 节）；
- 每个多处理器的寄存器数量是 16384；
- 每个多处理器的最大活动 warp 块数量是 32；
- 每个多处理器的最大活动线程数是 1024。

A.1.4 计算能力 1.3 的规范

- 支持双精度浮点数。

A.2 浮点标准

所有计算设备都符合针对二进制浮点算术的 IEEE-754 标准，但有以下例外：

- 不存在动态可配置的舍入模式（rounding mode）；但大多数操作都支持 IEEE 舍入模式，可通过设备函数公开；
- 不存在检测发生浮点异常的机制，所有操作的行为就像 IEEE-754 异常被掩藏一样，并且这种隐藏的表达方式与 IEEE-754 遇到异常事件时一样；同样，尽管支持 SNaN（Signaling NaN） 编码，但实际上不支持 Signaling；
- 绝对值和负数不符合 IEEE-574 在 NaN 方面的规定，它们将按原样传递；

- 对于单精度浮点数：
  - 加法和乘法往往结合为一条乘-加指令（FMAD），它将截取乘法的中间结果；
  - 除法是通过倒数、以非标准的方式实现的；
  - 平方根是以非标准的方式通过倒数平方根实现的；
  - 对于加法和乘法，仅支持通过静态舍入模式实现舍入到最近的偶数（round-to-nearest-even）和向零舍入（round-towards-zero），不支持向正负无穷大方向的舍入；
  - 不支持反向规格化数（denormalized number）；浮点算术和比较指令会在浮点操作之前将操作数反向规格化为零；
  - 下溢的结果将被刷新为零；
  - 涉及一个或多个输入 NaN 的操作的结果是位模式 0x7fffffff 的 QNaN（Quiet NaN），请加以注意：
- 仅对于双精度浮点数：
  - 倒数、除法和平方根运算中惟一支持的 IEEE 舍入模式是舍入到最近的偶数

根据 IEEE-754R 标准，如果 **`fminf()`**、**`fmin()`**、**`fmaxf()`** 或 **`fmax()`** **的输入参数之一是 `NaN`**，但其他参数不是，则结果为非 NaN 参数。

IEEE-754 并未定义在浮点值超出整型格式的范围的情况下，浮点值到整型值的转换方式。对于计算设备而言，采用将数限定在所支持的范围尾端的方法。这与 x86 架构的处理方式不同。

附录 B 标准数学函数

B.1 节中列举的函数可由宿主和设备函数使用，但 B.2 节中的函数仅能在设备函数中使用。

B.1 通用运行时组件

这一节介绍 CUDA 运行时库支持的所有数学标准库（mathematical standard library）函数。此外还指定了各函数在设备上执行时的误差范围。当函数在宿主上执行，而宿主并未提供此函数时，误差范围同样适用。这里列举的误差范围经过了广泛的测试，但并非详尽无遗，所以很难保证正确无误。

B.1.1 单精度浮点函数

加法和乘法是符合 IEEE 的，因此误差最多为 0.5 ulp。但在设备上，编译器往往将其整合为一个单独的乘-加指令（FMAD），截取乘法的中间结果。可通过使用 _fadd_rn() 和 _fmul_rn() 函数来避免这样的整合（请参见第 B.2 节）。

要将单精度浮点操作数舍入为整型，而使结果为单精度浮点数，推荐的方法是使用 rintf() 而非 roundf()。原因在于 roundf() 会映射到设备上的一个 8 指令序列，而 rintf() 只映射到一条指令。truncf()、ceilf() 和 floorf() 也均可映射到一条指令。

表 B-1. 数学标准库函数及其最大 ULP 误差

最大误差表示为正确舍入的单精度结果和 CUDA 库函数返回的结果之间以 ulp 计算的差的绝对值。

函数	最大 ulp 错误
x+y	0（IEEE-754 舍入到最近的偶数） （在合并入 FMAD 时除外）
x*y	0（IEEE-754 舍入到最近的偶数） （在合并入 FMAD 时除外）
x/y	2（全范围（full range））
1/x	1 (全范围)
1/sqrtf(x) rsqrtf(x)	2 (全范围)
sqrtf(x)	3 (全范围)
cbrtf(x)	1 (全范围)
hypotf(x,y)	3 (全范围)
expf(x)	2 (全范围)
exp2f(x)	2 (全范围)
exp10f(x)	2 (全范围)
expm1f(x)	1 (全范围)
logf(x)	1 (全范围)
log2f(x)	3 (全范围)
log10f(x)	3 (全范围)
log1pf(x)	2 (全范围)
sinf(x)	2 (全范围)
cosf(x)	2 (全范围)
tanf(x)	4 (全范围)

sincosf(x,sptr,cptr)	2 (全范围)
asinf(x)	4 (全范围)
acosf(x)	3 (全范围)
atanf(x)	2 (全范围)
atan2f(y,x)	3 (全范围)
sinhf(x)	3 (全范围)
coshf(x)	2 (全范围)
tanhf(x)	2 (全范围)
asinhf(x)	3 (全范围)
acoshf(x)	4 (全范围)
atanhf(x)	3 (全范围)
powf(x,y)	7 (全范围)
erff(x)	4 (全范围)
erfcf(x)	8 (全范围)
lgammaf(x)	6 (外间距（outside interval）-10.001 ... -2.264; 内间距更大)
tgammaf(x)	11 (全范围)
fmaf(x,y,z)	0 (全范围)
frexpf(x,exp)	0 (全范围)
ldexpf(x,exp)	0 (全范围)
scalbnf(x,n)	0 (全范围)
scalblnf(x,l)	0 (全范围)
logbf(x)	0 (全范围)
ilogbf(x)	0 (全范围)
fmodf(x,y)	0 (全范围)
remainderf(x,y)	0 (全范围)
remquof(x,y,iptr)	0 (全范围)
modff(x,iptr)	0 (全范围)
fdimf(x,y)	0 (全范围)
truncf(x)	0 (全范围)
roundf(x)	0 (全范围)
rintf(x)	0 (全范围)
nearbyintf(x)	0 (全范围)
ceilf(x)	0 (全范围)
floorf(x)	0 (全范围)
lrintf(x)	0 (全范围)
lroundf(x)	0 (全范围)
llrintf(x)	0 (全范围)
llroundf(x)	0 (全范围)
signbit(x)	N/A
isinf(x)	N/A
isnan(x)	N/A
isfinite(x)	N/A
copysignf(x,y)	N/A
fminf(x,y)	N/A
fmaxf(x,y)	N/A
fabsf(x)	N/A
nanf(cptr)	N/A
nextafterf(x,y)	N/A

B.1.2 双精度浮点函数

下面列举的误差仅适用于为具有本地双精度支持的设备编译的情况。在为无此类支持的设备编译时（如计算能力为 1.2 或更低的设备），double 类型将默认地降低为 float，双精度数学函数将映射为其单精度版本。

要将双精度浮点操作数舍入为整型，而使结果为双精度浮点数，推荐的方法是使用 rint()，而不是 round()。原因在于 round() 会映射到设备上的一个 8 指令序列，而 rint() 仅映射到一条指令。Trunc()、ceil() 和 floor() 均可映射到一条指令。

表 B-2. 数学标准库函数及其最大 ULP 错误

最大误差表示为正确舍入的双精度结果和 CUDA 库函数返回的结果之间以 ulp 计算的差的绝对值。

函数	最大 ulp 错误
x+y	0 （IEEE-754 舍入到最近的偶数）
x*y	0 （IEEE-754 舍入到最近的偶数）
x/y	0 （IEEE-754 舍入到最近的偶数）
1/x	0 （IEEE-754 舍入到最近的偶数）
sqrt(x)	0 （IEEE-754 舍入到最近的偶数）
rsqrt(x)	1 (全范围)
cbrt(x)	1 (全范围)
hypot(x,y)	2 (全范围)
exp(x)	1 (全范围)
exp2(x)	1 (全范围)
exp10(x)	1 (全范围)
expm1(x)	1 (全范围)
log(x)	1 (全范围)
log2(x)	1 (全范围)
log10(x)	1 (全范围)
log1px(x)	1 (全范围)
sin(x)	2 (全范围)
cos(x)	2 (全范围)
tan(x)	3 (全范围)
sincos(x,sptr,cptr)	2 (全范围)
asin(x)	2 (全范围)
acos(x)	2 (全范围)
atan(x)	3 (全范围)
atan2(y,x)	3 (全范围)
sinh(x)	1 (全范围)
cosh(x)	1 (全范围)
tanh(x)	1 (全范围)
asinh(x)	2 (全范围)
acosh(x)	2 (全范围)
atanh(x)	2 (全范围)
pow(x,y)	2 (全范围)
erf(x)	2 (全范围)
erfc(x)	6 (全范围)
lgamma(x)	4 (外间距-11.0001 ... -2.2637; 内间距更大)
tgamma(x)	7 (全范围)
fma(x,y,z)	0 （IEEE-754 舍入到最近的偶数）
frexp(x,exp)	0 (全范围)
ldexp(x,exp)	0 (全范围)

scalbn(x,n)	0 (全范围)
scalbln(x,l)	0 (全范围)
logb(x)	0 (全范围)
ilogb(x)	0 (全范围)
fmod(x,y)	0 (全范围)
remainder(x,y)	0 (全范围)
remquo(x,y,iptr)	0 (全范围)
modf(x,iptr)	0 (全范围)
fdim(x,y)	0 (全范围)
trunc(x)	0 (全范围)
round(x)	0 (全范围)
rint(x)	0 (全范围)
nearbyint(x)	0 (全范围)
ceil(x)	0 (全范围)
floor(x)	0 (全范围)
lrint(x)	0 (全范围)
lround(x)	0 (全范围)
llrint(x)	0 (全范围)
llround(x)	0 (全范围)
signbit(x)	N/A
isinf(x)	N/A
isnan(x)	N/A
isfinite(x)	N/A
copysign(x,y)	N/A
fmin(x,y)	N/A
fmax(x,y)	N/A
fabs(x)	N/A
nan(cptr)	N/A
nextafter(x,y)	N/A

B.1.3 整型函数

CUDA 运行时库支持整型 min(x, y) 和 max(x, y)，它们可映射到设备上的一条指令。

B.2 设备运行时组件

这一节列举了仅在设备代码中支持的内建函数。这些函数中包括第 B.1 节所列函数的精确度较低但速度更快的版本；它们具有相同的名称，另外加上 __ 前缀（如__sinf(x)）。

使用 _rn 后缀的函数将使用舍入到最近的偶数模式操作；

使用 _rz 后缀的函数将使用向零方向舍入模式操作；

使用 _ru 后缀的函数将使用向上舍入（向正无穷大）模式；

使用 _rz 后缀的函数将使用向下舍入（向负无穷大）模式。

不同于将一种类型转换为另一种类型的类型转换函数（如 _int2float_rn），类型强制转换函数直接对一个参数执行类型强制转换，而保留其值不变。例如：

`__int_as_float(0xC0000000)` 等于 **`-2`**，**`__float_as_int(1.0f)`** 等于 **`0x3f800000`**。

B.2.1 单精度浮点函数

__fadd_rn() 和 __fmul_rn() 映射为加法和乘法操作，且编译器不会把他们并入 FMAD 中。与此相比，“*”和“+”运算符生成的加法和乘法一般都将被并入 FMAD。

普通浮点除法和 _fdividef(x, y) 具有相同的精确度，但在 $2^{126} < \mathbf{y} < 2^{128}$ 时，__fdivedef(x, y) 的结果为 0，而普通除法将在表 B-1 列举的精确度内提供正确的结果。同样，在 $2^{126} < \mathbf{y} < 2^{128}$ 时，如果 x 是无穷大，__ fdivedf(x, y) 将得到结果 NaN（无穷大乘以 0 的结果），而普通除法将返回无穷大。

__[u]mul24(x, y) 计算整型参数 x 和 y 的 24 个最低有效位的乘积，提供 32 个最低有效位的结果。x 和 y 的 8 个最高有效位将被忽略。

__[u]mulhi(x, y) 计算整型参数 x 和 y 的乘积，提供 64 位结果中的 32 个最高有效位。

__[u]mul64hi(x, y) 计算 64 位整型参数 x 和 y 的乘积，提供 128 位结果中的 64 个最高有效位。

__saturate(x) 将在 x 小于 0 时返回 0，在 x 大于 1 时返回 1，否则返回 x。

__[u]sad(x, y, z)（绝对差之和）返回整型参数 z 和整型参数 x 与 y 的差的绝对值之和。

__clz(x) 返回整型参数 x 从最高有效位（即第 31 位）开始的连续 0 位的数量，介于 0 和 32 之间（包括 0 和 32）。

__clzll(x) 返回 64 位整型参数 x 从最高有效位（即第 63 位）开始的连续 0 位的数量，介于 0 和 64 之间（包括 0 和 64）。

__ffs(x) 返回整型参数 x 中第一个（最低有效）位组的位置。最低有效位是位置 1。如果 x 为 0，_ffs() 将返回 0。请注意，此函数等同于 Linux 函数 ffs。

__ffsll(x) 返回 64 位整型参数 x 中的第一个（最低有效）位组的位置。最低有效位是位置 1。如果 x 为 0，__ffsll() 将返回 0。请注意，此函数等同于 Linux 函数 ffsll。

表 B-3 CUDA 运行时库支持的单精度浮点内建函数及其误差范围

函数	误差范围
`__fadd_[rn,rz](x,y)`	符合 IEEE
`__fmul_[rn,rz](x,y)`	符合 IEEE
`__fdividef(x,y)`	如果 y 在 $[2^{-126}, 2^{126}]$ 区间内，则最大 ulp 误差为 2.
`__expf(x)`	最大 ulp 误差为 `2 + floor(abs(1.16 * x))`。
`__exp10f(x)`	最大 ulp 误差为 `2 + floor(abs(2.95 * x))`。
`__logf(x)`	如果 x 在 [0.5, 2] 区间内，则最大绝对误差为 $2^{-21.41}$，否则最大 ulp 误差为 3。
`__log2f(x)`	如果 x 在 [0.5, 2] 区间内，则最大绝对误差为 2^{-22}，否则最大 ulp 误差为 2。
`__log10f(x)`	如果 x 在 [0.5, 2] 区间内，则最大绝对误差为 2^{-24}，否则最大 ulp 误差为 3。
`__sinf(x)`	如果 x 在 [-π, π] 区间内，则最大绝对误差为 $2^{-21.41}$，否则更大。
`__cosf(x)`	如果 x 在 [-π, π] 区间内，则最大绝对误差为 $2^{-21.19}$，否则更大。
`__sincosf(x,sptr,cptr)`	与 sinf(x) 和 cosf(x) 相同。
`__tanf(x)`	继承自以下实现：`__sinf(x) * (1 / __cosf(x))`。
`__powf(x, y)`	继承自以下实现：`exp2f(y * __log2f(x))`。
`__mul24(x,y)` `__umul24(x,y)`	N/A
`__mulhi(x,y)` `__umulhi(x,y)`	N/A

`__int_as_float(x)`	N/A
`__float_as_int(x)`	N/A
`__saturate(x)`	N/A
`__sad(x,y,z)` `__usad(x,y,z)`	N/A
`__clz(x)`	N/A
`__ffs(x)`	N/A
`__float2int_[rn,rz,ru,rd]`	N/A
`__float2uint_[rn,rz,ru,rd]`	N/A
`__int2float_[rn,rz,ru,rd]`	N/A
`__uint2float_[rn,rz,ru,rd]`	N/A

B.2.2 双精度浮点函数

_dadd_rn() 和 _dmul_rn() 映射为加法和乘法运算，且编译器不会把他们并入 FMAD 运算。与此相比，“*”和“+”运算符生成的加法和乘法一般都将被并入 FMAD。

表 B-4. CUDA 运行时库支持的双精度浮点内建函数及其误差范围

函数	误差范围
__dadd_[rn,rz,ru,rd](x,y)	符合 IEEE
__dmul_[rn,rz,ru,rd](x,y)	IEEE-compliant.
__fma_[rn,rz,ru,rd](x,y,z)	IEEE-compliant.
__double2float_[rn,rz](x)	N/A
__double2int_[rn,rz,ru,rd](x)	N/A
__double2uint_[rn,rz,ru,rd](x)	N/A
__double2ll_[rn,rz,ru,rd](x)	N/A
__double2ull_[rn,rz,ru,rd](x)	N/A
__int2double_rn(x)	N/A
__uint2double_rn(x)	N/A
__ll2double_[rn,rz,ru,rd](x)	N/A
__ull2double_[rn,rz,ru,rd](x)	N/A
__double_as_longlong(x)	N/A
__longlong_as_double(x)	N/A
__double2hiint(x)	N/A
__double2loint(x)	N/A
__hiloint2double(x, ys)	N/A

B.2.3 整型函数

_popc(x) 返回在 32 位整型参数 x 的二进制表示中设置为 1 的位数。

_popcll(x) 返回在 64 位整型参数 x 的二进制表示中设置为 1 的位数。

附录 C 原子函数

原子函数仅可用于设备函数之中，仅对计算能力为 1.1 或更高的设备可用。

操作共享存储器和 64 位字的原子函数只能用于计算能力为 1.2 或更高的设备。

C.1 数学函数

C.1.1 atomicAdd()

```
int atomicAdd(int* address, int val);
unsigned int atomicAdd(unsigned int* address,
                       unsigned int val);
unsigned long long int atomicAdd(unsigned long long int* address,
                                 unsigned long long int val);
```

读取位于全局或共享存储器中地址 address 处的 32 位或 64 位字 old，计算 (old + val)，并将结果存储在存储器的同一地址中。这三项操作在一次原子事务中执行。该函数将返回 old。

只有全局存储器支持 64 位字。

C.1.2 atomicSub()

```
int atomicSub(int* address, int val);
unsigned int atomicSub(unsigned int* address,
                       unsigned int val);
```

读取位于全局或共享存储器中地址 address 处的 32 位字 old，计算 (old - val)，并将结果存储在存储器的同一地址中。这三项操作在一次原子事务中执行。该函数将返回 old。

C.1.3 atomicExch()

```
int atomicExch(int* address, int val);
unsigned int atomicExch(unsigned int* address,
                        unsigned int val);
unsigned long long int atomicExch(unsigned long long int* address,
                                  unsigned long long int val);
float atomicExch(float* address, float val);
```

读取位于全局或共享存储器中地址 address 处的 32 位或 64 位字 old，并将 val 存储在存储器的同一地址中。这两项操作在一次原子事务中执行。该函数将返回 old。

只有全局存储器支持 64 位字。

C.1.4 atomicMin()

```
int atomicMin(int* address, int val);
unsigned int atomicMin(unsigned int* address,
                       unsigned int val);
```

读取位于全局或共享存储器中地址 address 处的 32 位字 old，计算 old 和 val 的最小值，并将结果存储在存储器的同一地址中。这三项操作在一次原子事务中执行。该函数将返回 old。

C.1.5 atomicMax()

```
int atomicMax(int* address, int val);
unsigned int atomicMax(unsigned int* address,
                       unsigned int val);
```

读取位于全局或共享存储器中地址 address 处的 32 位字 old，计算 old 和 val 的最大值，并将结果存储在存储器的同一地址中。这三项操作在一次原子事务中执行。该函数将返回 old。

C.1.6 atomicInc()

```
unsigned int atomicInc(unsigned int* address,
                       unsigned int val);
```

读取位于全局或共享存储器中地址 address 处的 32 位字 old，计算 **((old >= val) ? 0 : (old+1))**，并将结果存储在存储器的同一地址中。这三项操作在一次原子事务中执行。该函数将返回 old。

C.1.7 atomicDec()

```
unsigned int atomicDec(unsigned int* address,
                       unsigned int val);
```

读取位于全局或共享存储器中地址 address 处的 32 位字 old，计算 **(((old == 0) | (old > val)) ? val : (old-1))**，并将结果存储在存储器的同一地址中。这三项操作在一次原子事务中执行。该函数将返回 old。

C.1.8 atomicCAS()

```
int atomicCAS(int* address, int compare, int val);
unsigned int atomicCAS(unsigned int* address,
                       unsigned int compare,
                       unsigned int val);
unsigned long long int atomicCAS(unsigned long long int* address,
                                 unsigned long long int compare,
                                 unsigned long long int val);
```

读取位于全局或共享存储器中地址 address 处的 32 位或 64 位字 old，计算 **(old == compare ? val : old)**，并将结果存储在存储器的同一地址中。这三项操作在一次原子事务中执行。该函数将返回 old（比较并交换）。

只有全局存储器支持 64 位字。

C.2 位逻辑函数

C.2.1 atomicAnd()

```
int atomicAnd(int* address, int val);
unsigned int atomicAnd(unsigned int* address,
                       unsigned int val);
```

读取位于全局或共享存储器中地址 address 处的 32 位字 old，计算 **(old & val)**，并将结果存储在存储器的同一地址中。这三项操作在一次原子事务中执行。该函数将返回 old。

C.2.2 atomicOr()

```
int atomicOr(int* address, int val);
unsigned int atomicOr(unsigned int* address,
                      unsigned int val);
```

读取位于全局或共享存储器中地址 address 处的 32 位字 old，计算 **(old | val)**，并将结果存储在存储器的同一地址中。这三项操作在一次原子事务中执行。该函数将返回 old。

C.2.3 atomicXor()

```
int atomicXor(int* address, int val);
unsigned int atomicXor(unsigned int* address,
                       unsigned int val);
```

读取位于全局或共享存储器中地址 address 处的 32 位字 old，计算 **(old ^ val)**，并将结果存储在存储器的同一地址中。这三项操作在一次原子事务中执行。该函数将返回 old。

附录 D 纹理拾取

本附录提供了用于根据纹理参考的不同属性计算第 4.4.3 节介绍的纹理函数的返回值的公式（请参见第 4.3.4 节）。

绑定到纹理参考的纹理表示为数组 T，包含针对一维纹理的 N 个 texel，或者针对二维纹理的 $N \times M$ 个 texel，或者针对三维纹理的 $N \times M \times L$ 个 texel。它将使用纹理坐标 x、y 和 z 进行拾取。

纹理坐标必须位于 T 的有效寻址范围内，之后才能用于寻址 T。寻址模式指定了如何将超出范围的纹理坐标映射到有效范围内。如果 x 是非归一化的，则仅支持调整寻址模式，如果 $x < 0$，则将 x 替换为 0，如果 $N \le x$，则替换为 N-1。如果 x 是归一化的：

- 在 clamp 寻址模式内，如果 $x < 0$，x 将替换为 0，如果 $1 \le x$，则替换为 1-1/N；
- 在 wrap 寻址模式中，x 将被替换为 frac(x)，其中 frac(x) = x-floor(x)，floor(x) 是不大于 x 的最大整数。

在下面的内容中，x、y 和 z 是非归一化的纹理坐标，已重新映射到 T 的有效寻址范围内。x、y、z 继承自归一化纹理坐标 x^ 、y^ 和 z^，即 x = Nx^、y = My^、z = Lz^。

D.1 最近点采样

在这种过滤模式中，纹理拾取返回的值如下：

- 对于一维纹理是 $tex(x) = T[i]$
- 对于二维纹理是 $tex(x, y) = T[i, j]$
- 对于三维纹理是 $tex(x, y, z) = T[i, j, k]$

其中 i = floor(x) 、j = floor(y)、k = floor(z)。

图 F-1 展示了一维纹理的最近点采样，N =4。

对于整型纹理来说，纹理拾取的返回值可重新映射到 [0.0, 1.0]（请参见第 4.3.4.1 节）。

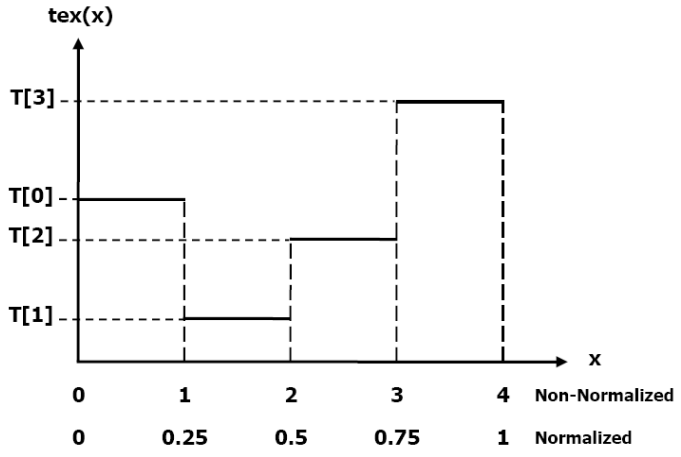


图 F-1. 包含 4 个 texel 的一维纹理的最近点采样

D.2 线性过滤

在这种仅对浮点纹理可用的过滤模式中，纹理拾取的返回值如下：

- 对于一维纹理是 tex(x) = (1– α)T[i] + α T[i +1]
- 对于二维纹理是 tex(x, y) = (1– α)(1– β)T[i, j] + α (1– β)T[i +1, j] + (1– α) β T[i, j +1] + α β T[i +1, j +1]
- 对于三维纹理是

$$tex(x,y,z) =$$
$$(1-\alpha)(1-\beta)(1-\gamma)T[i,j,k]+\alpha(1-\beta)(1-\gamma)T[i+1,j,k]+$$
$$(1-\alpha)\beta(1-\gamma)T[i,j+1,k]+\alpha\beta(1-\gamma)T[i+1,j+1,k]+$$
$$(1-\alpha)(1-\beta)\gamma T[i,j,k+1]+\alpha(1-\beta)\gamma T[i+1,j,k+1]+$$
$$(1-\alpha)\beta\gamma T[i,j+1,k+1]+\alpha\beta\gamma T[i+1,j+1,k+1]$$

其中：

- $i = floor(x_B)$, α = $frac(x_B)$, $x_B = x - 0.5$；
- $j = floor(y_B)$, β = $frac(y_B)$, $y_B = y - 0.5$；
- $k = floor(z_B)$, γ = $frac(z_B)$, $z_B = z - 0.5$。

α、β和γ 存储在 9 位的定点格式中，有 8 位 fractional 值。

图 F-2 展示了一维纹理的最近点采样，N = 4。

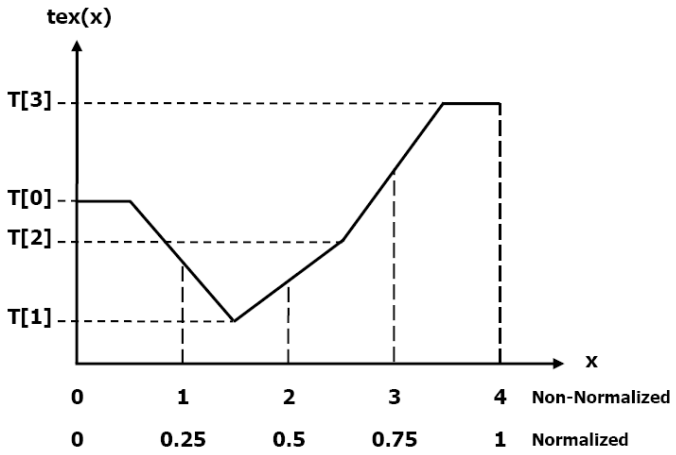


图 F-2. 压缩寻址模式中有 4 个 Texel 的一维纹理的线性过滤

D.3 表查找

表查找函数 TL(x) 可实现为 $TL(x) = tex(\frac{N-1}{R}x + 0.5)$ ，其中 x 处于 [0, R] 区间内，这样即可确保 $TL(0) = T[0]$ 且 $TL(R) = T[N-1]$。

图 F-3 展示了利用纹理过滤来实现表查找，其中 R=4 或 R=1，来自 N=4 的一维纹理。

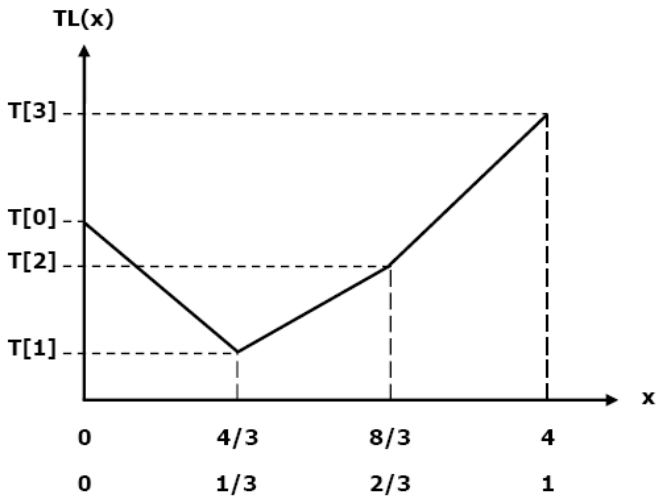


图 F-3. 使用线性过滤的一维表查找

注意：

所有 NVDIA 设计规范、参考设计、文件、绘图、诊断程序、列表和其他文档（统称为“材料”）均按“原样”提供。NVIDIA 不对此材料提供任何明示或暗示的保证和担保，不保证材料的无侵害性、适销性或针对特定目的的适用性。

我们尽可能保证信息的准确和可靠。但 NVIDIA 不为使用此类信息的后果承担责任，也不为使用此类信息可能导致的侵害第三方专利或其他权利的后果负责。本材料未通过任何隐含的方式或其他方式授予对 NVIDIA Corporation 专利或权利权的许可。本文档提到的规范可能随时更改，恕不另行通知。本文档提供的是最新内容，取代之前提供的所有信息。若无 NVIDIA Corporation 的明确书面许可，不得将 NVIDIA Corporation 的产品用作生命保障设备或系统的关键组件。

商标

NVIDIA、NVIDIA 徽标、Geforce、Tesla 和 Quadro 都是 NVIDIA Corporation 的商标或注册商标。其他公司或产品名称可能是其对应公司的商标。

版权

本文档整合了早先一份资料的部分内容：Scalable Parallel Programming with CUDA, in ACM Queue, VOL 6, No. 2 (March/April 2008), © ACM, 2008. http://mags.acm.org/queue/20080304/?u1=texterity"

致谢

刘伟峰先生对本指南中文版译稿全文进行了审校，在此表示衷心的感谢。

NVIDIA Corporation