CUDA 编程指南 3.0 中文版

译者：风辰
QQ:304128534
E-mail:ily152832912@gmail.com

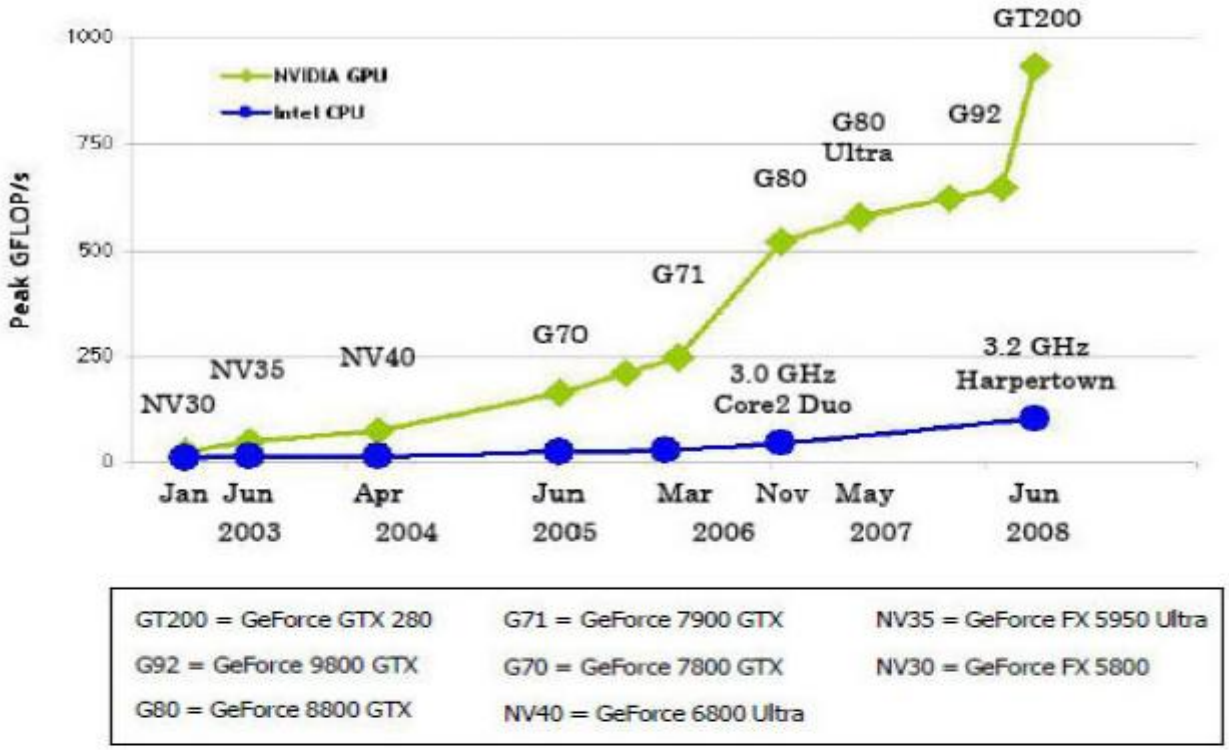
本文档依据 2 月 9 号的英文文档翻译，所以和 2 月 20 的会有些不同，请大家指出，兄弟会尽快更正。谢谢！

目录

第一章 导论

1.1 从图形处理到通用并行计算

市场对实时、高清晰度的三维图形具有无法满足的要求，由于这种要求的推动，可编程图像处理器（GPU）已经演化成高并行度，多线程，拥有强大计算能力和极高存储器带宽的多核处理器，如图 1-1 所示



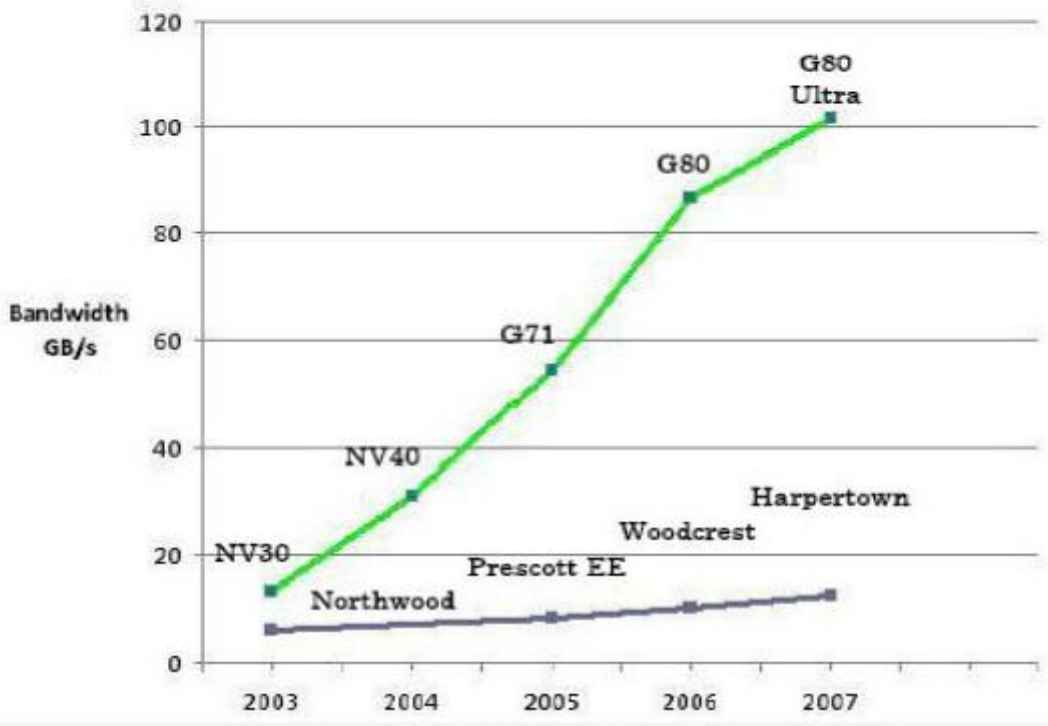


图1-1. CPU和GPU每秒浮点操作数和存储器带宽

GPU 和 CPU 的浮点计算能力差异的原因是：GPU 是特别为计算密集，高并行度计算（如同图像渲染）设计的，因此将更多的晶体管用于数据处理而不是数据缓存和流控，如图 1-2 所示。

图1-2. GPU将更多的晶体管用于数据处理

特别地，GPU 非常适合处理那些能够表示为数据并行计算（同一程序在多个数据上并行执行）的问题，数据并行计算的算术计算密度（算术操作和存储器操作的比例）非常高。由于同一程序在每个元素上执行，因此对复杂流控的要求非常少，更因为在多个元素上执行和高计算密度，访存延迟可以被计算隐藏，因此用不着大的数据缓存。

数据并行处理将数据元素映射到并行处理的线程上。很多处理大的数据集的应用可以使用数据并行处理模型加速。三维图像渲染处理中，大量的像素和顶点被映射到并行线程。类似地，图像和多媒体处理应用、图像渲染的后处理、视频编解码、图像缩放、立体视觉和模式识别能够将图像块和像素映射到并行处理的线程。实际上，除了图像渲染和处理领域，还有很多算法已被数据并行处理加速，从普通信号处理或物理模拟到计算金融或计算生物学。

1.2 CUDA™：一种通用并行计算架构

2006 年 11 月，英伟达推出了 CUDA™，一种新的并行编程模型和指令集架构的通用计算架构，CUDA 能够利用英伟达 GPU 的并行计算引擎比 CPU 更高效的解决许多复杂计算任务。

CUDA 包含一个让开发者能够使用 C 作为高级编程语言的软件环境。如图 1-3 所示，其它的语言和应用编程接口（API）也被支持，如 CUDA FORTRAN，OpenCL 和 Direct Compute。

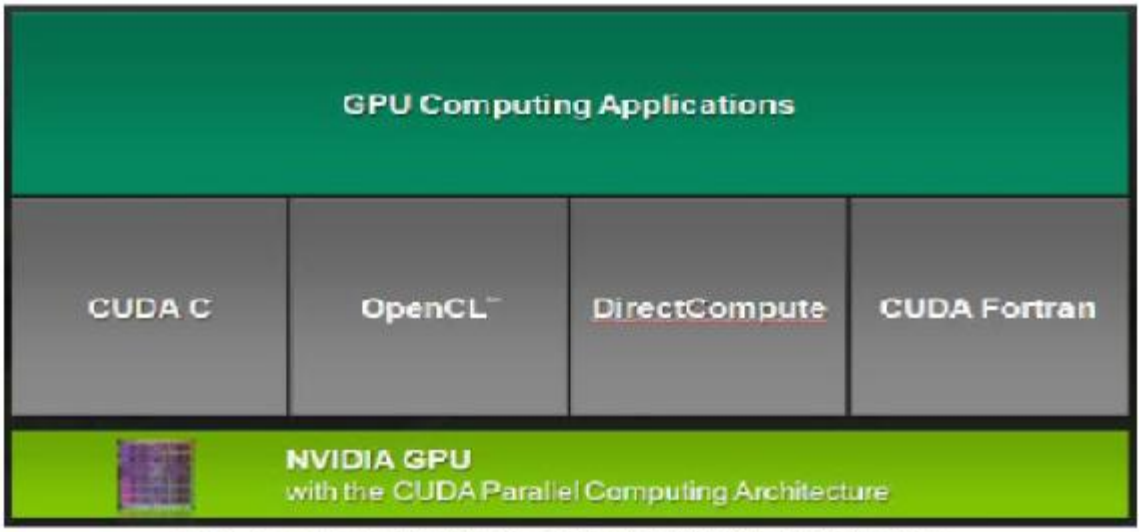


图1-3. CUDA被设计为支持多种语言或编程接口

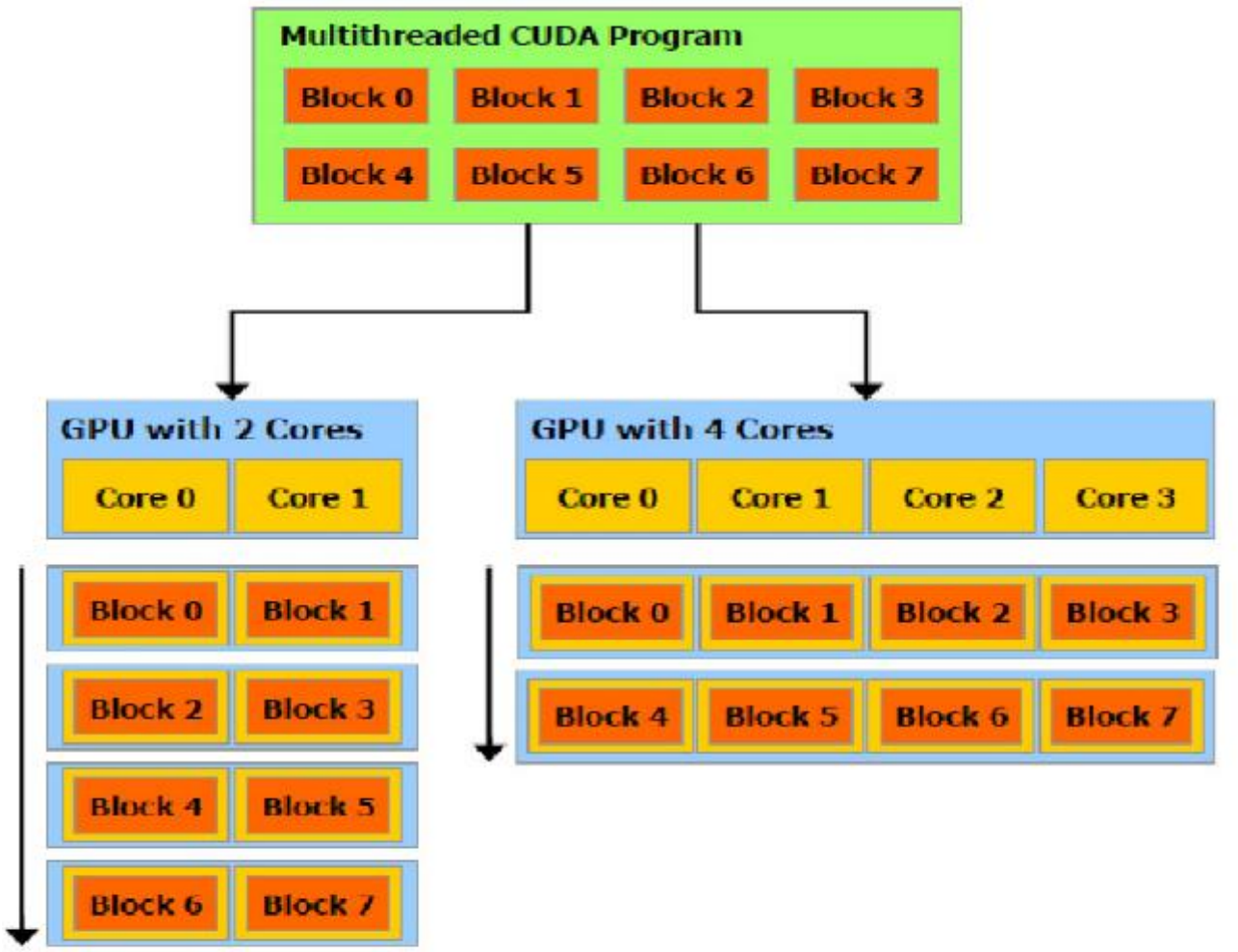
1.3 一种可扩展的编程模型

多核 CPU 和众核 GPU 的出现，意味着主流处理器芯片现在已经是并行系统了。更进一步的说，他们的并行度将继续以摩尔定律扩展。面临的挑战是开发透明的扩展并行度以利用不断增加的处理器核心数的应用软件，更像三维图形应用在不同数目核心的 GPU 上，透明的扩展他们的并行度一样。

设计 CUDA 并行编程模型是为了在克服这种挑战的同时，使得熟悉标准编程语言（如 C）的程序员保持一个比较低的学习曲线。

CUDA 核心包含三个重点抽象：线程组层次、共享存储器和栅栏同步，这些被作为一个最小的语言扩展集简单呈现（expose）给程序员。

这些抽象提供了细粒度数据并行度和线程并行度，嵌套在粗粒度数据并行和任务并行中。他们引导程序员将问题划分为可以被多个块内线程独立并行处理的粗粒度子问题，而每个子问题又被分为可以被一个块内线程并行协作处理的更小的片段。这种分解通过在处理子问题的时候允许线程协作，保持了语言的表达性，同时保证了自动可扩展性。事实上，每个块可被调度到可用处理器核心的任意一个上，以任何顺序，并行或者串行执行，这使得已编译好的 CUDA 程序能够在任意核心的 GPU 上执行，如图 1-4 所示，只有运行时系统需要知道物理处理器的数量。



一个多线程程序被划分为多个线程块，块之间彼此无关，独立执行，因此一个拥有多核心的GPU花费的时间自动地比少核心的短.

图1-4. 自动可扩展性

这种可扩展的编程模型允许 CUDA 架构通过简单的缩放处理器的数量和存储器分区的数量来满足市场不同层次的需求：从高性能发烧友级精视 GPU 和专业级的 Quadro 和 Tesla 计算产品到多种便宜，主流的精视 GPU（参看附录 A 关于支持 CUDA 的 GPU 列表）

1.4 文档结构

本文档包括以下各章

- 第一章、CUDA 基本介绍
- 第二章、CUDA 编程模型要点
- 第三章、编程接口描述
- 第四章、硬件实现描述
- 第五章、性能优化指南
- 附录 A、支持 CUDA 的 GPU 列表
- 附录 B、CUDA C 扩展的详细说明
- 附录 C、CUDA 支持的数学函数列表
- 附录 D、设备支持的 C++结构列表
- 附录 E、nvcc 支持的关键字和指令列表
- 附录 F、更详细的纹理存取说明
- 附录 G、给出各种设备的技术规范，更多架构详细说明

第二章 编程模型

本章引入了 CUDA 编程模型背后的主要概念，方式是概述它们是怎样使用 C 语言表示的。更多的关于 CUDA C 的描述在 3.2 节。

本章使用的向量相加例子的完整代码和下一个例子可在 SDK 中的 vectorAdd 代码样本中找到。

2.1 内核

CUDA 通过允许程序员定义称为内核的 C 函数扩展了 C，内核调用时会被 N 个 CUDA 线程执行 N 次（注：这句话要好好理解，其实每个线程只执行了一次），这和普通的 C 函数只执行一次不同。

内核使用__global__声明符定义，使用一种新<<<...>>>执行配置语法指定执行某一指定内核调用的线程数（参看附录 B.13）。每个执行内核的线程拥有一个独一无二的线程 ID，可以通过内置的 threadIdx 变量在内核中访问。

下面的样本代码将两个长度为 N 的向量 A 和 B 相加，并将结果存入向量 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，N 个线程中的每一个执行 VecAdd()的一次成对加法。

2.2 线程层次

为简便起见，threadIdx 是一个有 3 个部件的向量，所以线程可以使用一维，二维，三维索引标识，形成一维，二维，三维的线程块。这提供了一种自然的方式来调用作用在域内元素上的计算，如向量，矩阵，体元(volume)。

线程索引和线程 ID 直接相关：对于一维的块，它们相同；对于二维长度为（Dx,Dy）的块，线程索引为（x,y)的线程 ID 是(x+yDx)；对于三维长度为(Dx,Dy,Dz)的块，索引为（x,y,z)的线程 ID 为（x+yDx+zDxDy）。

下面的例子代码将两个长度为 N*N 的矩阵 A 和 B 相加，然后将结果写入矩阵 C。

```
// 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);
}
```

由于块内的所有线程必须存在于同一个处理器核心中且共享该核心有限的存储器资源，因此，一个块内的线程数目是有限的。在目前的 GPU 上，一个线程块可以包含多达 512 个线程。

然而，一个内核可被多个同样大小的线程块执行，所以总的线程数等于每个块内的线程数乘以线程块数。

线程块被组织成一维或二维的线程网格，如图 2-1 所示。一个网格内的线程块数往往由被处理的数据量而不是系统的处理器数决定，前者往往远超后者。

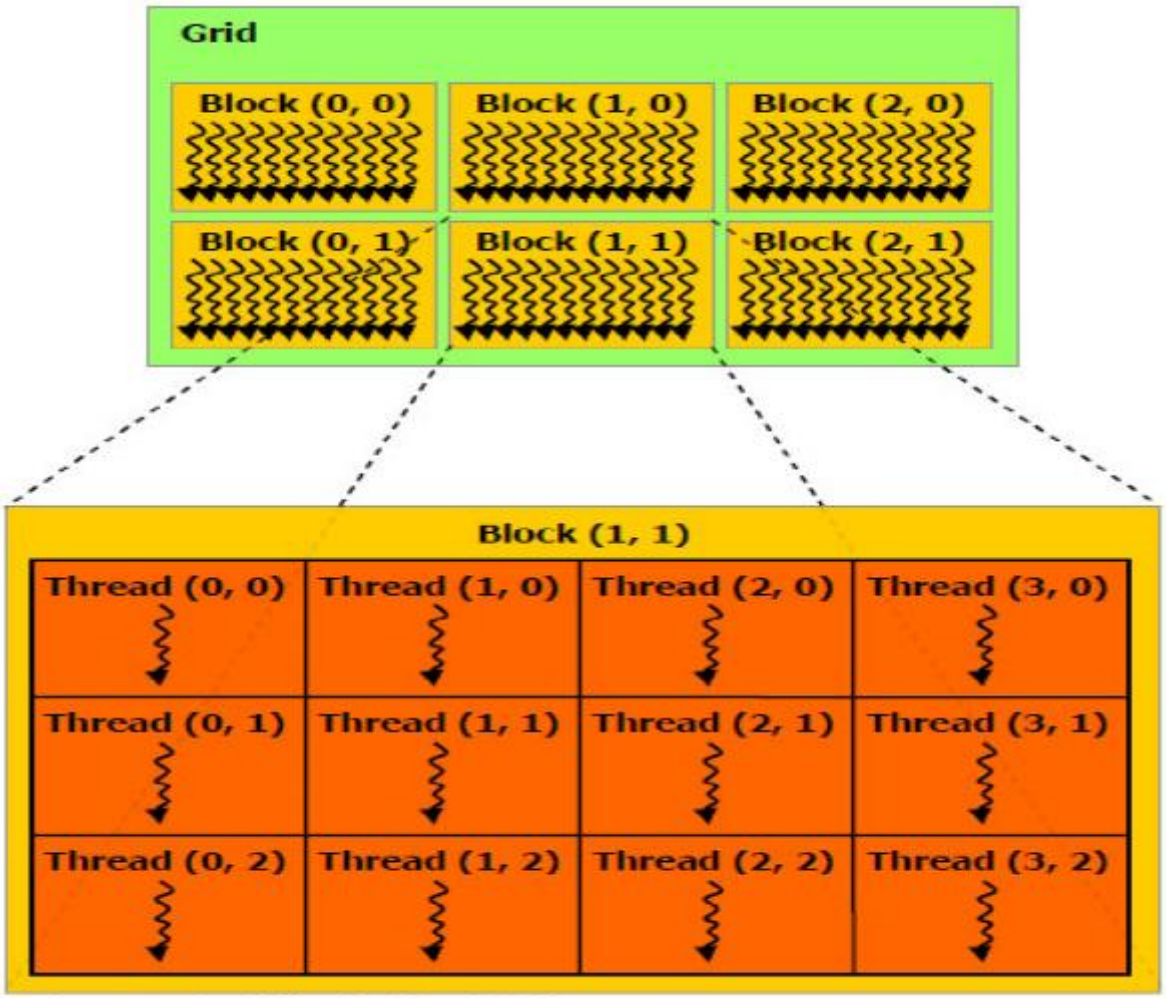


图2-1. 网格内线程块

线程块内线程数和网格内线程块数由<<<...>>>语法确定，参数可以是整形或者 dim3 类

型。二维的块或网格的尺寸可以以和上一个例子相同的方式指定。

网格内的每个块可以通过可在内核中访问的一维或二维索引唯一确定，此索引可通过内置的 blockIdx 变量获得。块的尺寸(dimension)可以在内核中通过内置变量 blockDim 访问。

为了处理多个块，扩展前面的 MatAdd()例子后，代码成了下面的样子。

```
// 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);
}
```

一个长度为 16*16（256 线程）的块，虽然是特意指定，但是常见。像以前一样，创建了内有足够的块的网格，使得一个线程处理一个矩阵元素。为简便起见，此例假设网格每一维上的线程数可被块内对应维上的线程数整除，尽管这并不常见。

线程块必须独立执行。而且能够以任意顺序，串行或者并行执行。这种独立性要求使得线程块可以以任何顺序在任意数目核心上调度，如图 1-4 所示，保证了程序员能够写出能够随核心数目扩展的代码（enabling programmers to write code that scales with the number of cores）。

块内线程可通过共享存储器和同步执行协作，共享存储器可以共享数据，同步执行可以协调存储器访问。更精确一点说，可以在内核中调用__syncthreads()内置函数指明同步点；__syncthreads()起栅栏的作用，在其调用点，块内线程必须等待，直到所以线程都到达此点才能向前执行。节 3.2.2 给出了一个使用共享存储器的例子。

为了能有效协作，共享存储器要求是靠近每个处理器核心的低延迟存储器（更像 L1 缓存），而且__syncthreads()要是轻量级的。

2.3 存储器层次

在执行期间，CUDA 线程可能访问来自多个存储器空间的数据，如图 2-2 所示。每个线程有私有的本地存储器。每个块有对块内所有线程可见的共享存储器，共享存储器的生命期和块相同。所有的线程可访问同一全局存储器。

另外还有两种可被所有线程访问的只读存储器：常量和纹理存储器空间。全局，常量和纹理存储器空间为不同的存储器用途作了优化（参看 5.3.2.1 节,5.3.2.4 节和 5.3.2.5 节）。纹理存储器还为一些特殊数据格式提供了不同的寻址模式和数据滤波（参看 3.2.4 节）。

在同一应用发射的内核之间，全局，常量和纹理存储器空间是持久的。

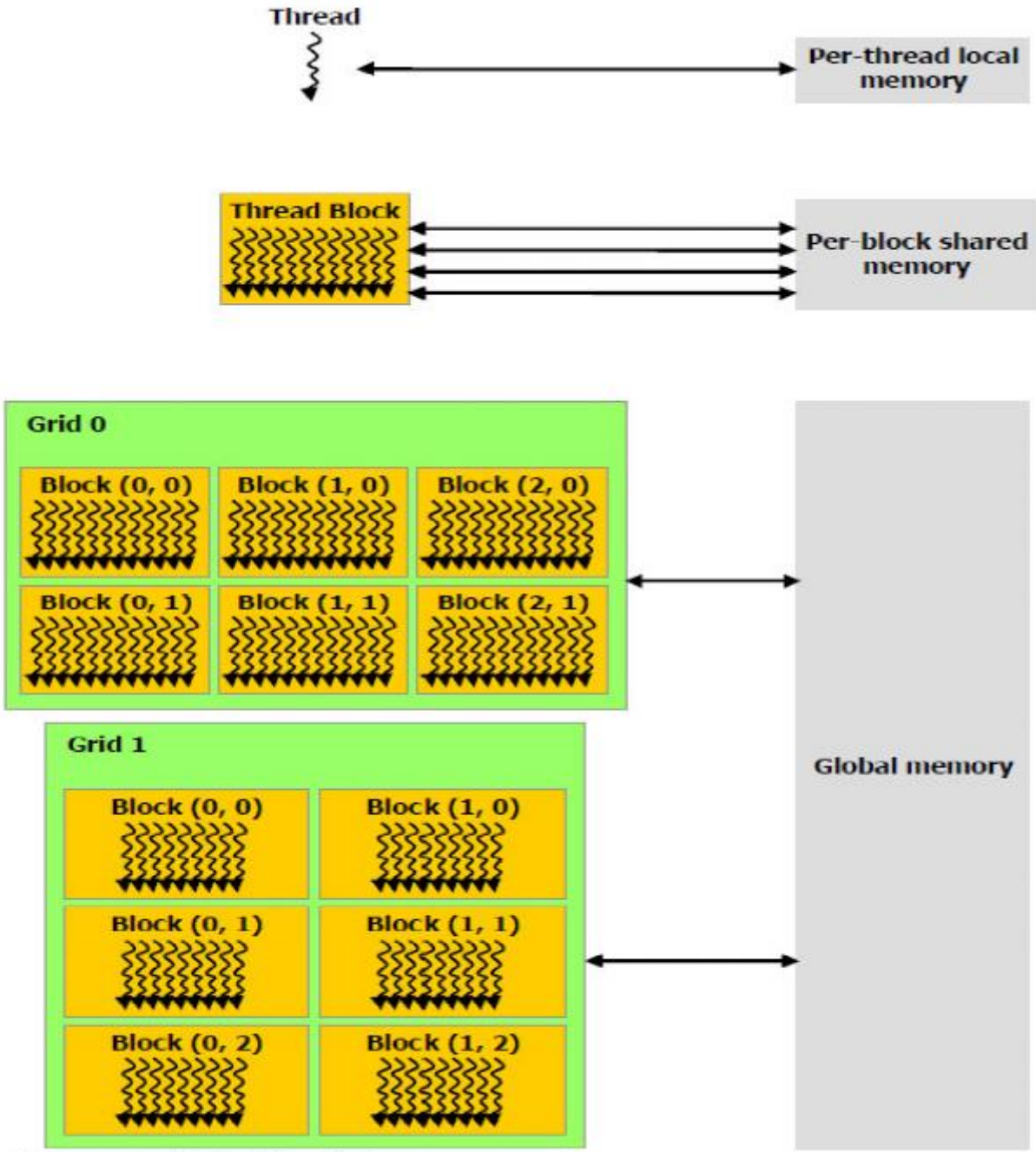
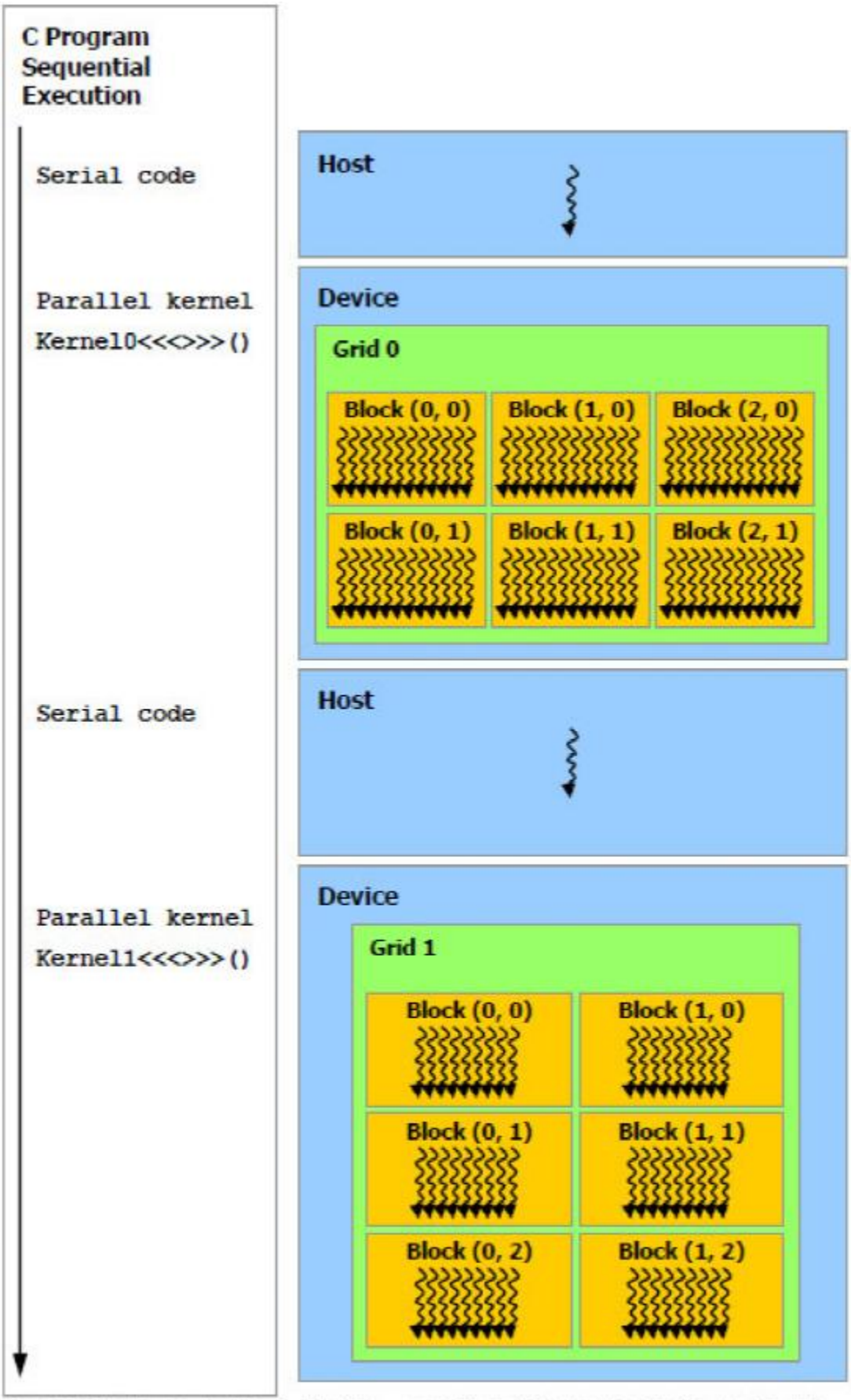


图2-2. 存储器层次

2.4 异构编程

如图 2-3 所示，CUDA 编程模型假设 CUDA 线程在物理上独立的设备上执行，设备作为主机的协处理器，主机运行 C 程序。例如，内核在 GPU 上执行，而 C 程序的其它部分在 CPU 上执行就是这种模式。

CUDA 编程模型同时假设主机和设备各自都维护着自己独立的 DRAM 存储器空间，各自被称为主机存储器空间和设备存储器空间。因此，程序通过调用 CUDA 运行时，来管理对内核可见的全局、常量和纹理存储器空间（参看第三章）。这包括设备存储器分配和释放，也包括在主机和设备间的数据传输。



串行代码在主机上执行，而并行代码在设备上执行

图2-3. 异构编程

2.5 计算能力

设备的计算能力由主修订号和次修订号定义。

主修订号相同的设备基于相同的核心架构。Fermi 架构的主修订号为 2。以前的设备的计算能力都是 1.x（它们的主修订号为 1）。

次修订号对应着对核心架构的增量提升，也可能包含了新特性。

附录 A 列出了所有支持 CUDA 的设备，包括它们的计算能力。附录 G 给出了各计算能力设备的技术规范。

第三章 编程接口

目前可用两种接口写 CUDA 程序：CUDA C 和 CUDA 驱动 API。一个应用典型的只能使用其中一种，但是遵守 3.4 节描述的限制时，可以同时使用两种。

CUDA C 将 CUDA 编程模型作为 C 的最小扩展集展示出来。任何包含某些扩展的源文件必须使用 nvcc 编译，nvcc 的概要在 3.1 节。这些扩展允许程序员像定义 C 函数一样定义内核和在每次内核调用时，使用新的语法指定网格和块的尺寸。

CUDA 驱动 API 是一个低层次的 C 接口，它提供了从汇编代码或 CUDA 二进制模块中装载内核，检查内核参数，和发射内核的函数。二进制和汇编代码通常可以通过编译使用 C 写的内核得到。

CUDA C 包含运行时 API，运行时 API 和驱动 API 都提供了分配和释放设备存储器、在主机和内存间传输数据、管理多设备的系统的函数等等。

运行时 API 是基于驱动 API 的，初始化、上下文和模块管理都是隐式的，而且代码更简明。CUDA C 也支持设备模拟，这有利于调试（参见节 3.2.8）。

相反，CUDA 驱动 API 要求写更多的代码，难于编程和调试，但是易于控制且是语言无关的，因为它处理的是二进制或汇编代码。

3.2 节接着第二章介绍 CUDA C。也引入了 CUDA C 和驱动 API 共有的概念：线性存储器、CUDA 数组、共享存储器、纹理存储器、分页锁定主机存储器、设备模拟、异步执行和与图形学 API 互操作。3.3 节会介绍有关这些概念的知识和描述它们在驱动 API 中是怎样表示的。

3.1 用 nvcc 编译

内核可以使用 PTX 编写，PTX 就是 CUDA 指令集架构，PTX 参考手册中描述了 PTX。通常 PTX 效率高于像 C 一样的高级语言。无论是使用 PTX 还是高级语言，内核都必须使用 nvcc 编译成二进制代码才能在设备在执行。

nvcc 是一个编译器驱动，简化了 C 或 PTX 的编译流程：它提供了简单熟悉的命令行选项，同时通过调用一系列实现了不同编译步骤的工具集来执行它们。本节简介了 nvcc 的编译流程和命令选项。完整的描述可在 nvcc 用户手册中找到。

3.1.1 编译流程

nvcc 可编译同时包含主机代码（有主机上执行的代码）和设备代码（在设备上执行的代码）的源文件。nvcc 的基本流程包括分离主机和设备代码并将设备代码编译成汇编形式（PTX）或/和二进制形式（cubin 对象）。生成的主机代码要么被输出为 C 代码供其它工具编译，要么在编译的最后阶段被 nvcc 调用主机编译器输出为目标代码。

应用能够：

- 要么在设备上使用 CUDA 驱动 API 装载和执行 PTX 源码或 cubin 对象（参见 3.3 节）同时忽略生成的主机代码（如果有）；
- 要么链接到生成的主机代码；生成的主机代码将 PTX 代码和/或 cubin 对象作为已初始化的全局数据数组导入，还将 2.1 节引入的<<<…>>>语法转化为必要的函数调用以加载和发射每个已编译的内核。

应用在运行时装载的任何 PTX 代码被设备驱动进一步编译成二进制代码。这称为即时编

译。即时编译增加了应用装载时间，但是可以享受编译器的最新改进带来的好处。也是当前应用能够在未来的设备上运行的唯一方式，细节参见 3.1.4 节。

3.1.2 二进制兼容性

二进制代码是由架构确定的。生成 cubin 对象时，使用编译器选项-code 指定目标架构：例如，用-code=sm_13 编译时，为计算能力 1.3 的设备生成二进制代码 。二进制兼容性保证向后兼容，但不保证向前兼容，也不保证跨越主修订号兼容。换句话说，为计算能力为 X.y 生成的 cubin 对象只能保证在计算能力为 X.z 的设备上执行，这里，z>=y。

3.1.3 PTX 兼容性

一些 PTX 指令只被高计算能力的设备支持。例如，全局存储器上的原子指令只在计算能力 1.1 及以上的设备上支持；双精度指令只在 1.3 及以上的设备上支持。将 C 编译成 PTX 代码时，-arch 编译器选项指定假定的计算能力。因此包含双精度计算的代码，必须使用“-arch=sm_13”（或更高计算能力）编译，否则双精度计算将被降级为单精度计算。

为某些特殊计算能力生成的 PTX 代码始终能够被编译成相等或更高计算能力设备上的二进制代码。

3.1.4 应用兼容性

为了在特定计算能力的设备上执行代码，应用加载的二进制或 PTX 代码必须满足如 3.1.2 节和 3.1.3 节说明的计算能力兼容性。特别地，为了能在将来更高计算能力（不能产生二进制代码）的架构上执行，应用必须装载 PTX 代码并为那些设备即时编译。

CUDA C 应用中嵌入的 PTX 和二进制代码，由-arch 和-code 编译器选项或-gencode 编译器选项控制，详见 nvcc 用户手册。例如，

```
nvcc x.cu
   -gencode arch=compute_10,code=sm_10
   –gencode arch=compute_11,code=\’compute_11,sm_11\’
```

嵌入与计算能力 1.0 兼容的二进制代码（第一个-gencode 选项）和 PTX 和与计算能力 1.1 兼容的二进制代码（第二个-gencode 选项）。

生成的主机代码在运行时自动选择最合适的代码装载并执行，对于上面例子，将会是：

- 1.0 二进制代码为计算能力 1.0 设备，
- 1.1 二进制代码为计算能力 1.1,1.2,1.3 的设备，
- 通过为计算能力 2.0 或更高的设备编译 1.1PTX 代码获得的二进制代码。

例如，x.cu 可有一个使用原子指令的优化代码途径，只能支持计算能力 1.1 或更高的设备。__CUDA_ARCH__宏可以基于计算能力用于不同的代码途径。它只为设备代码定义。例如，当使用“arch=compte_11”编译时，__CUDA_ARCH__等于 110。

使用驱动 API 的应用必须将代码编译成分立的文件，且在运行时显式装载和执行最合适的文件。

nvcc 用户手册为-arch,-code 和-gencode 编译器选项列出了多种简写。如“arch=sm_13”是“arch=compute_13 code=compute_13,sm_13”的简写（等价于“-gencode arch=compute_13,code=\’ compute_13,sm_13\’ ”）。

3.1.5 C++兼容性

编译器前端依据C++语法规则处理CUDA源文件。主机代码完整支持C++。设备代码只完整支持C++的一个子集，详见附录D。由于使用了C++的语法规则，空指针（如malloc()的返回值）不能赋值给非空指针，必须转型后才能赋值。

nvcc也支持特定的关键字和指令，详见附录E。

3.2 CUDA C

CUDA C为熟悉C语言的用户提供了一个简单途径，让他们能够轻易的写出能够在设备上执行的程序。

CUDA C包含了一个C语言的最小扩展集和一个运行时库。语言核心扩展在第二章已经介绍了。本节继续介绍运行时。所有扩展的完整的描述可在附录B找到，CUDA运行时的完整描述可在CUDA参考手册中找到。

cudart动态库是运行时的实现，它所有的入口点前缀都是cuda。

运行时没有显式的初始化函数；在初次调用运行时函数（更精确地，不在参考手册中设备和版本管理节中的任何函数）时初始化。在计算运行时函数调用时间和解析初次调用运行时产生的错误码时必须牢记这点。

一旦运行时在主机线程中初始化，在主机线程中通过一些运行时函数调用分配的任何资源（存储器，流，事件等）只在当前主机线程的上下文中有效。因此只有在这个主机线程中调用的运行时函数（存储器拷贝，内核发射等）才能操作这些资源。这是因为CUDA上下文（参见3.3.1节）作为初始化的一部分建立且成为主机线程的当前上下文，且不能成为其它主机线程的当前上下文。

在多设备的系统中，内核默认在0号设备上执行，详见3.2.3节。

3.2.1 设备存储器

正如2.4节所提到的，CUDA编程模型假定系统包含主机和设备，它们各有自己独立的存储器。内核不能操作设备存储器，所以运行时提供了分配，释放，拷贝设备存储器和在设备和主机间传输数据的函数。

设备存储器可被分配为线性存储器或CUDA数组。

CUDA数组是不透明的存储器层次，为纹理获取做了优化。它们的细节在3.2.4节。

计算能力1.x的设备，其线性存储器存在于32位地址空间内，计算能力2.0的设备，其线性存储器存在于40位地址空间内，所以独立分配的实体能够通过指针引用，如，二叉树。

典型地，线性存储器使用cudaMalloc()分配，通过cudaFree()释放，使用cudaMemcpy()在设备和主机间传输。在2.1节的向量加法代码中，向量要从主机存储器复制到设备存储器：

```
// Device code
__global__ void VecAdd(float* A, float* B, float* C, int N) {
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < N)
        C[i] = A[i] + B[i];
}
// Host code
int main(){
    int N = ...;
    size_t size = N * sizeof(float);
    // Allocate input vectors h_A and h_B in host memory
    float* h_A = (float*)malloc(size);
    float* h_B = (float*)malloc(size);
    // Initialize input vectors
    ...
    // Allocate vectors in device memory
    float* d_A;
    cudaMalloc((void**)&d_A, size);
    float* d_B;
    cudaMalloc((void**)&d_B, size);
    float* d_C;
    cudaMalloc((void**)&d_C, size);
    // Copy vectors from host memory to device memory
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
    // Invoke kernel
    int threadsPerBlock = 256;
    int blocksPerGrid = (N + threadsPerBlock – 1) / threadsPerBlock;
    VecAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, N);
    // Copy result from device memory to host memory
    // h_C contains the result in host memory
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
    // Free device memory
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
    // Free host memory
    ...
}
```

线性存储器也可以通过 cudaMallocPitch()和 cudaMalloc3D()分配。在分配 2D 或 3D 数组的时候，推荐使用，因为这些分配增加了合适的填充以满足 5.3.2.1 节的对齐要求，在按行访问时或者在二维数组和设备存储器的其它区域间复制（用 cudaMemcpy2D()和 cudaMemcpy3D()函数）时，保证了最佳性能。返回的步长（pitch,stride)必须用于访问数组元素。下面的代码分配了一个尺寸为 width*height 的二维浮点数组，同时演示了怎样在设备代码中遍历数组元素。

```
// 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];
        }
    }
}
```

下面的代码分配了一个尺寸为 width*height*depth 的三维浮点数组，同时演示了怎样在设备代码中遍历数组元素。

```
// Host code
cudaPitchedPtr devPitchedPtr;
cudaExtent extent = make_cudaExtent(64, 64, 64);
cudaMalloc3D(&devPitchedPtr, extent);
MyKernel<<<100, 512>>>(devPitchedPtr, extent);
// Device code
__global__ void MyKernel(cudaPitchedPtr devPitchedPtr, cudaExtent extent) {
    char* devPtr = devPitchedPtr.ptr;
    size_t pitch = devPitchedPtr.pitch;
    size_t slicePitch = pitch * extent.height;
    for (int z = 0; z < extent.depth; ++z) {
        char* slice = devPtr + z * slicePitch;
        for (int y = 0; y < extent.height; ++y) {
        float* row = (float*)(slice + y * pitch);
        for (int x = 0; x < extent.width; ++x) {
            float element = row[x];
        }
    }
}
```

参考手册列出了在 cudaMalloc()分配的线性存储器，cudaMallocPitch()或 cudaMalloc3D()分配的线性存储器，CUDA 数组，和为声明在全局存储器和常量存储器空间分配的存储器之间拷贝的所有各种函数。

下面的例子代码复制了一些主机存储器数组到常量存储器中：

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

为声明在全局存储器空间的变量分配的存储器的地址，可以使用 cudaGetSymbolAddress()函数检索到。分配的存储器的尺寸可以通过 cudaGetSymbolSize()函数获得。

3.2.2 共享存储器

共享存储器使用__shared__限定词分配，详见 B.2 节。

正如在 2.2 节提到的，共享存储器应当比全局存储器更快，详见 5.3.2.3 节。任何用访问共享存储器取代访问全局存储器的机会应当被发掘，如下面的矩阵相乘例子展示的那样。

下面的代码是矩阵相乘的一个直接的实现，没有利用到共享存储器。每个线程读入 A 的一行和 B 的一列，然后计算 C 中对应的元素，如图 3-1 所示。这样，A 读了 B.width 次，B 读了 A.height 次。

```
// Matrices are stored in row-major order:
// M(row, col) = *(M.elements + row * M.width + col)
typedef struct {
    int width;
    int height;
    float* elements;
} Matrix;
// Thread block size
#define BLOCK_SIZE 16
// Forward declaration of the matrix multiplication kernel
__global__ void MatMulKernel(const Matrix, const Matrix, Matrix);
// Matrix multiplication - Host code
// Matrix dimensions are assumed to be multiples of BLOCK_SIZE
void MatMul(const Matrix A, const Matrix B, Matrix C) {
    // Load A and B to device memory
    Matrix d_A;
    d_A.width = A.width; d_A.height = A.height;
    size_t size = A.width * A.height * sizeof(float);
    cudaMalloc((void**)&d_A.elements, size);
    cudaMemcpy(d_A.elements, A.elements, size,
    cudaMemcpyHostToDevice);
    Matrix d_B;
    d_B.width = B.width; d_B.height = B.height;
    size = B.width * B.height * sizeof(float);
    cudaMalloc((void**)&d_B.elements, size);
    cudaMemcpy(d_B.elements, B.elements, size,
    cudaMemcpyHostToDevice);
    // Allocate C in device memory
    Matrix d_C;
    d_C.width = C.width; d_C.height = C.height;
    size = C.width * C.height * sizeof(float);
    cudaMalloc((void**)&d_C.elements, size);
    // Invoke kernel
    dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
    dim3 dimGrid(B.width / dimBlock.x, A.height / dimBlock.y);
    MatMulKernel<<<dimGrid, dimBlock>>>(d_A, d_B, d_C);
    // Read C from device memory
    cudaMemcpy(C.elements, Cd.elements, size, cudaMemcpyDeviceToHost);
    // Free device memory
    cudaFree(d_A.elements);
    cudaFree(d_B.elements);
    cudaFree(d_C.elements);
}
// Matrix multiplication kernel called by MatMul()
__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C) {
    // Each thread computes one element of C
    // by accumulating results into Cvalue
    float Cvalue = 0;
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    for (int e = 0; e < A.width; ++e)
    Cvalue += A.elements[row * A.width + e]
    * B.elements[e * B.width + col];
    C.elements[row * C.width + col] = Cvalue;
}
```

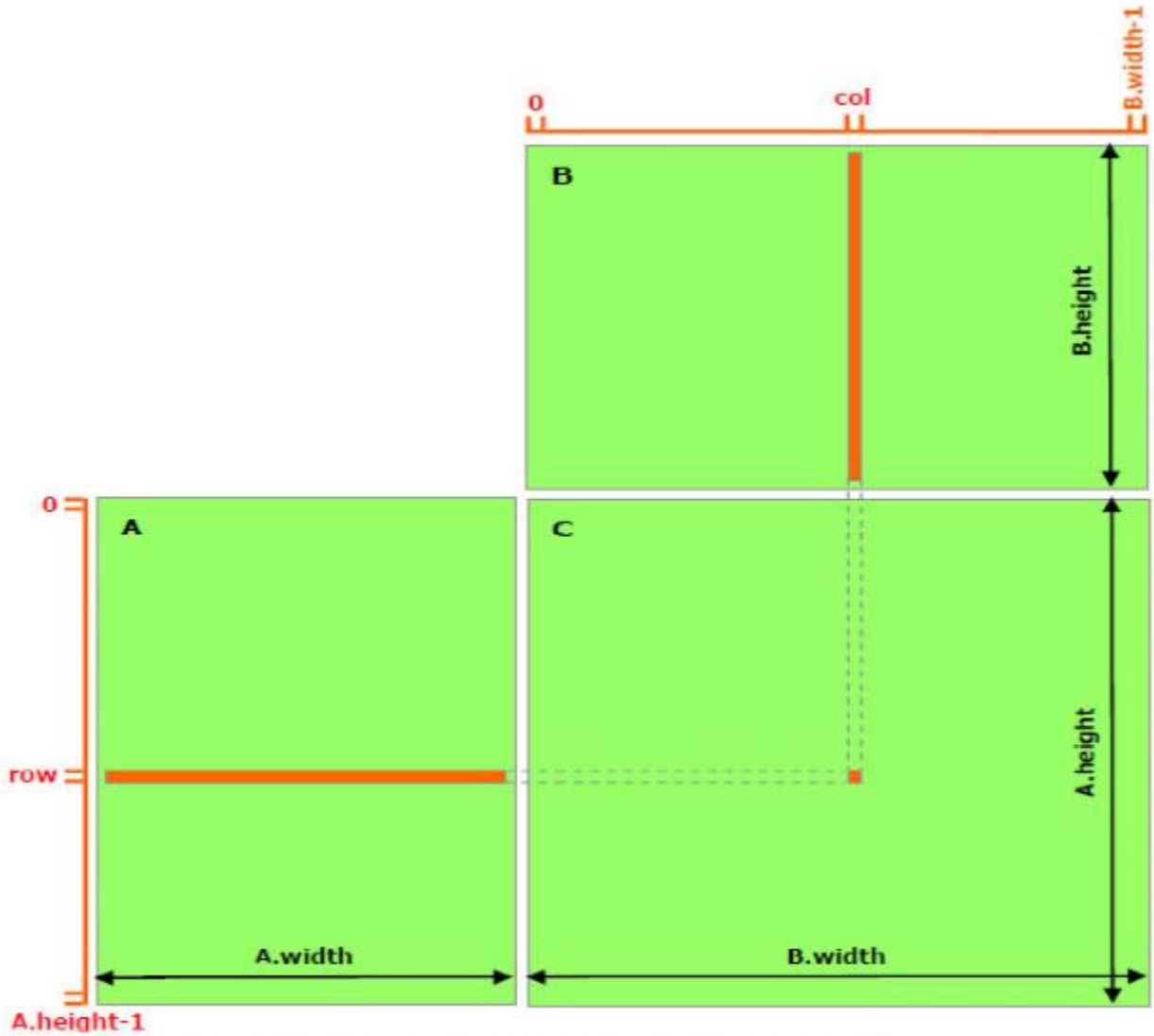


图3-1. 没有使用共享存储器的矩阵相乘

下面的例子代码利用了共享存储器实现矩阵相乘。本实现中，每个线程块负责计算一个小方阵 Csub，Csub 是 C 的一部分，而块内的每个线程计算 Csub 的一个元素。如图 3-2 所示。Csub 等于两个长方形矩阵的乘积：A 的子矩阵尺寸是（A.width,block_size），行索引与 Csub 相同，B 的子矩阵的尺寸是（block_size,A.width），列索引与 Csub 相同。为了满足设备的资源，两个长方形的子矩阵分割为尺寸为 block_size 的方阵，Csub 是这些方阵积的和。每次乘法的计算是这样的，首先从全局存储器中将二个对应的方阵载入共享存储器中，载入的方式是一个线程载入一个矩阵元素，然后一个线程计算乘积的一个元素。每个线程积累每次乘法的结果并写入寄存器中，结束后，再写入全局存储器。

采用这种将计算分块的方式，利用了快速的共享存储器，节约了许多全局存储器带宽，因为在全局存储器中，A 只被读了（B.width/block_size）次同时 B 读了（A.height/block_size）次。

前面代码中的 Matrix 类型增加了一个 stride 域，这样子矩阵能够用同样的类型有效表示。__device__函数（见 B.1.1 节）用于读写元素和从矩阵中建立子矩阵。

```
// Matrices are stored in row-major order:
// M(row, col) = *(M.elements + row * M.stride + col)
typedef struct {
    int width;
    int height;
    int stride;
    float* elements;
} Matrix;
// Get a matrix element
__device__ float GetElement(const Matrix A, int row, int col) {
    return A.elements[row * A.stride + col];
}
// Set a matrix element
__device__ void SetElement(Matrix A, int row, int col, float value) {
    A.elements[row * A.stride + col] = value;
}
// Get the BLOCK_SIZExBLOCK_SIZE sub-matrix Asub of A that is
// located col sub-matrices to the right and row sub-matrices down
// from the upper-left corner of A
__device__ Matrix GetSubMatrix(Matrix A, int row, int col) {
    Matrix Asub;
    Asub.width = BLOCK_SIZE;
    Asub.height = BLOCK_SIZE;
    Asub.stride = A.stride;
    Asub.elements = &A.elements[A.stride * BLOCK_SIZE * row + BLOCK_SIZE * col];
    return Asub;
}
// Thread block size
#define BLOCK_SIZE 16
// Forward declaration of the matrix multiplication kernel
__global__ void MatMulKernel(const Matrix, const Matrix, Matrix);
// Matrix multiplication - Host code
// Matrix dimensions are assumed to be multiples of BLOCK_SIZE
void MatMul(const Matrix A, const Matrix B, Matrix C) {
    // Load A and B to device memory
    Matrix d_A;
    d_A.width = d_A.stride = A.width; d_A.height = A.height;
    size_t size = A.width * A.height * sizeof(float);
    cudaMalloc((void**)&d_A.elements, size);
    cudaMemcpy(d_A.elements, A.elements, size,
    cudaMemcpyHostToDevice);
    Matrix d_B;
    d_B.width = d_B.stride = B.width; d_B.height = B.height;
    size = B.width * B.height * sizeof(float);
    cudaMalloc((void**)&d_B.elements, size);
    cudaMemcpy(d_B.elements, B.elements, size,
    cudaMemcpyHostToDevice);
    // Allocate C in device memory
    Matrix d_C;
    d_C.width = d_C.stride = C.width; d_C.height = C.height;
    size = C.width * C.height * sizeof(float);
    cudaMalloc((void**)&d_C.elements, size);
    // Invoke kernel
    dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
    dim3 dimGrid(B.width / dimBlock.x, A.height / dimBlock.y);
    MatMulKernel<<<dimGrid, dimBlock>>>(d_A, d_B, d_C);
```

```
        // Read C from device memory
        cudaMemcpy(C.elements, d_C.elements, size,
        cudaMemcpyDeviceToHost);
        // Free device memory
        cudaFree(d_A.elements);
        cudaFree(d_B.elements);
        cudaFree(d_C.elements);
}
// Matrix multiplication kernel called by MatMul()
__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C) {
        // Block row and column
        int blockRow = blockIdx.y;
        int blockCol = blockIdx.x;
        // Each thread block computes one sub-matrix Csub of C
        Matrix Csub = GetSubMatrix(C, blockRow, blockCol);
        // Each thread computes one element of Csub
        // by accumulating results into Cvalue
        float Cvalue = 0;
        // Thread row and column within Csub
        int row = threadIdx.y;
        int col = threadIdx.x;
        // Loop over all the sub-matrices of A and B that are
        // required to compute Csub
        // Multiply each pair of sub-matrices together
        // and accumulate the results
        for (int m = 0; m < (A.width / BLOCK_SIZE); ++m) {
        // Get sub-matrix Asub of A
        Matrix Asub = GetSubMatrix(A, blockRow, m);
        // Get sub-matrix Bsub of B Matrix Bsub = GetSubMatrix(B, m, blockCol);
          // Shared memory used to store Asub and Bsub respectively
          __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
        __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
        // Load Asub and Bsub from device memory to shared memory
          // Each thread loads one element of each sub-matrix
        As[row][col] = GetElement(Asub, row, col);
        Bs[row][col] = GetElement(Bsub, row, col);
        // Synchronize to make sure the sub-matrices are loaded
          // before starting the computation
          __syncthreads();
          // Multiply Asub and Bsub together
          for (int e = 0; e < BLOCK_SIZE; ++e)
        Cvalue += As[row][e] * Bs[e][col];
        // 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 Csub to device memory
        // Each thread writes one element
          SetElement(Csub, row, col, Cvalue);
}
```

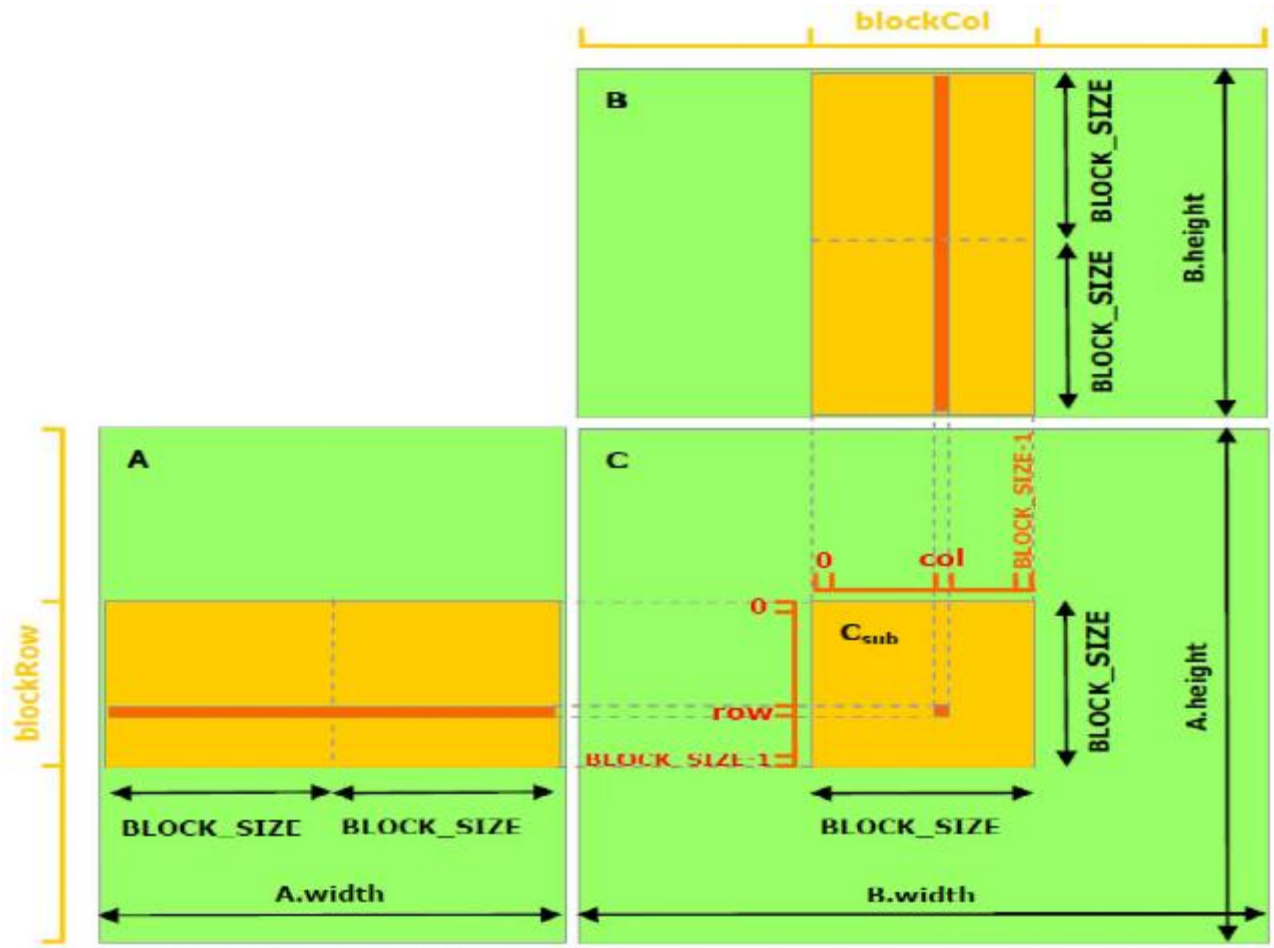


图3-2. 使用共享存储器的矩阵相乘

3.2.3 多设备

主机系统上可以有多个设备。可以枚举这些设备，也可以查询他们的属性，可以选择它们中的一个执行内核。

多个主机线程可以在同一个设备上执行设备代码，但是设计成在某个既定时间，一个主机线程只能在一个设备上执行设备代码。这样，多个主机线程在多个设备上执行设备代码。在某个主机线程内，使用 CUDA 运行时创建的任何 CUDA 资源不能被其它线程使用。

下面的例子代码枚举了系统中的所有设备同时检索了它们的属性。也确定了支持 CUDA 的设备的数目。

```
int deviceCount;
cudaGetDeviceCount(&deviceCount);
int device;
for (device = 0; device < deviceCount; ++device) {
    cudaDeviceProp deviceProp;
    cudaGetDeviceProperties(&deviceProp, device);
    if (dev == 0) {
        if (deviceProp.major == 9999 && deviceProp.minor == 9999)
            printf("There is no device supporting CUDA.\n");
        else if (deviceCount == 1)
            printf("There is 1 device supporting CUDA\n");
    else
        printf("There are %d devices supporting CUDA\n", deviceCount);
    }
}
```

默认情况下，只要一个非运行时设备管理函数调用（例外参见 3.6 节），主机线程隐式的使用 0 号设备。可以通过调用 cudaSetDevice()函数来启用其它的设备。一旦设备启用，无论是显式的还是隐式的对 cudaSetDevice()的调用都会失败，除非调用了 cudaThreadExit()。cudaThreadExit()清理所有与主机调用线程相关的运行相关的资源。随后的运行时 API 调用将重新初始化运行时。

3.2.4 纹理存储器

CUDA 支持纹理硬件的一个子集，GPU 为图形使用这个子集访问纹理存储器。如 5.3.2.5 节所示，从纹理存储器而不是全局存储器中读数据有许多性能好处。

如 B.8 节所示，在内核中，调用纹理获取设备函数读纹理存储器。纹理获取的第一个参数指定的对象称为纹理参考。

纹理参考定义了被获取的纹理存储器部分。如 3.2.4.3 节所述，纹理参考在被内核使用之前，必须使用运行时函数绑定到存储器的某个区域，这个区域称为纹理。多种不同的纹理参考可能绑定到同一纹理或者绑定到存储器重叠的纹理。

纹理参考有许多属性。其中之一就是维数，维数指定纹理是作为一维的数组使用一个纹理坐标、二维数组使用两个纹理坐标、还是三维数组使用三维坐标来寻址。数组的元素称为 texels，是纹理参考元素的简称。

其它属性定义纹理获取的输入输出数据类型，也包括怎样解释输入坐标和要做那些处理。

纹理可以是线性存储器的任何一个区域或者一个 CUDA 数组。

CUDA 数组是为纹理获取优化的不透明的存储器层次。它们可以是一维的，二维的或三维的，也可由多个元素组成，每个元素可有 1，2 或 4 个组件，这些组件可能是有符号或无符号 8，16 或 32 位整形，16 位浮点（目前只在驱动 API 中支持），或 32 位浮点。CUDA 数组只能在内核中通过纹理获取读取，且只能绑定到和已打包的组件数目相同的纹理参考。

2.2.4.1 纹理参考声明

纹理参考的一些属性不可变并且在编译时必须知道；它们在声明纹理参考时指定。纹理参考必须在文件域内声明，变量类型为 texture；

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

其中：

- Type 指定纹理获取时的返回的数据类型，Type 限制为基本的整形和单精度浮点型和 B.3.1 节定义的 1,2 和 4 个组件的向量类型的任何一个。
- Dim 指定纹理参考的维数，且等于 1,2 或 3；Dim 是可选的，默认为 1；
- ReadMode 等于 cudaReadModeNormalizedFloat 或 cudaReadModeElementType；如果它是 cudaReadModeNormalizedFloat 且 Type 是 16 位或者 8 位整形，实际返回值是浮点类型，对于无符号整型，整形全范围被映射到[0.0，1.0]，对于有符号整型，映射成[-1.0，1.0]；例如，无符号八位值为 0xff 的纹理元素映射为 1；如果 ReadMode 是 cudaReadModeElementType，不会进行转换；ReadMode 是个可选参数，默认为 cudaReadModeElementType。

3.2.4.2 运行时纹理参考属性

纹理参考的其它属性是可变的，并且能够在运行时通过主机运行时改变。这些属性指定

纹理坐标是否归一化、寻址模式和纹理滤波，细节如下。

默认情况下，纹理使用[0，N）范围内的浮点坐标引用，其中 N 是坐标对应维度的尺寸。例如，尺寸为 64*32 的纹理可引用的坐标范围是 x 维[0，63]和 y 维[0，31]。归一化的纹理坐标范围指定为[0.0，1.0）而不是[0，N），所以同样的 64*32 纹理的归一化坐标 x 维和 y 维可寻址范围都是[0,1)。归一化的纹理坐标天然的符合某些应用的要求，如果为了让纹理坐标独立于纹理尺寸，就更可取了。

寻址模式定义了当纹理坐标越界时发生了什么了。当使用非归一化纹理坐标时，纹理坐标在[0，N）范围之外的被钳位（clamp)：小于 0 的设置为 0 而大于等于 N 的设置为 N-1。钳位也是使用归一化纹理坐标时默认的寻址模式：小于 0.0 或大于 1.0 钳位到[0.0，1.0）范围。对于归一化坐标，也可以指定为循环寻址模式。一般在纹理有周期性信号时使用循环模式。循环模式只使用纹理坐标的小数部分；如 1.25 和 0.25 等同，-1.25 和 0.75 等同。

线性纹理滤波只能对返回值为浮点型的纹理配置起作用。它在周围的纹理元素点上执行低精度插值。如果启用滤波，纹理获取点周围的点被读取，纹理获取点的返回值基于那些纹理坐标落入那些元素中间的元素进行插值。对于一维的纹理进行简单的线性插值，而二维纹理使用双线性插值。

附录 F 给出了纹理获取的细节。

3.2.4.3 纹理绑定

如参考手册中所解释的，运行时 API 有一个低级的 C 风格的接口和一个高级的 C++风格的接口。texture 类型是在高级 API 中定义的一个结构体，公有继承自在低级 API 中定义的 textrueReference 类型。textureReference 定义如下：

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

- normalized 指定纹理坐标是否归一化；如果非零，纹理中所有元素可寻址的纹理坐标范围是[0，1]，而不是[0，width-1]，[0，height-1]，或[0，depth-1]，其中 width, height 和 depth 是纹理尺寸。
- filterMode 指定滤波模式，即纹理获取时，如何根据输入的纹理坐标计算返回值；filterMode 等于 cudaFilterModePoint 或 cudaFilterModeLinear；如果是 cudaFilterModePoint，则所返回的值为纹理坐标最接近输入纹理坐标的纹理元素；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）纹理元素的线性插值； 对于浮点型的返回值，cudaFilterModeLinear 是惟一的有效值。
- addressMode 指定寻址模式，即如何处理越界的纹理坐标；addressMode 是一个尺寸为 3 的数组，其第一个、第二个和第三个元素各自指定第一个、第二个和第三个纹理坐标的寻址模式；寻址模式可等于 cudaAddressModeClamp，此时越界的纹理坐标将被钳位到有效范围之内，也可等于 cudaAddressModeWrap，此时越界的纹理坐标将被环绕到有效范围之内；cudaAddressModeWrap 仅支持归一化的纹理坐标 。
- channelDesc 描述获取纹理时返回值的格式；channelDesc 类型定义如下：

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

其中

- cudaChannelFormatKindSigned，如果这些组件是有符号整型；
- cudaChannelFormatKindUnsigned，如果这些组件是无符号整型；
- cudaChannelFormatKindFloat，如果这些组件是浮点类型。

normalized、addressMode 和 filterMode 可直接在主机代码中修改。

在内核中使用纹理参考从纹理存储器中读取数据之前，必须使用 cudaBindTexture() 或 cudaBindTextureToArray() 将纹理参考绑定到纹理。cudaUnbindTexture()用于解绑定纹理参考。

下面的代码将纹理参考绑定到 devPtr 指针指向的线性存储器：

■ 使用低级 API：

```
texture<float, 2, cudaReadModeElementType> texRef;
textureReference* texRefPtr;
cudaGetTextureReference(&texRefPtr, “texRef”);
cudaChannelFormatDesc channelDesc =
cudaCreateChannelDesc<float>();
cudaBindTexture2D(0, texRefPtr, devPtr, &channelDesc, width, height, pitch);
```

■ 使用高级 API

```
texture<float, 2, cudaReadModeElementType> texRef;
cudaChannelFormatDesc channelDesc =
cudaCreateChannelDesc<float>();
cudaBindTexture2D(0, texRef, devPtr, &channelDesc, width, height, pitch);
```

下面的代码将纹理绑定到 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);
```

声明纹理参考时指定的参数必须与将纹理绑定到纹理参考时指定的格式匹配；否则纹理获取的结果没有定义。

下面的代码在内核中应用了一些简单的转换。

```
// 2D float texture
texture<float, 2, cudaReadModeElementType> texRef;
// Simple transformation kernel
__global__ void transformKernel(float* output,
      int width, int height, float theta) {
      // Calculate normalized texture coordinates
      unsigned int x = blockIdx.x * blockDim.x + threadIdx.x;
      unsigned int y = blockIdx.y * blockDim.y + threadIdx.y;
      float u = x / (float)width;
      float v = y / (float)height;
      // Transform coordinates
      u -= 0.5f; v -= 0.5f;
      float tu = u * cosf(theta) – v * sinf(theta) + 0.5f;
      float tv = v * cosf(theta) + u * sinf(theta) + 0.5f;
      // Read from texture and write to global memory
      output[y * width + x] = tex2D(tex, tu, tv);
}
// Host code
int main()
{
      // Allocate CUDA array in device memory
      cudaChannelFormatDesc channelDesc = cudaCreateChannelDesc(
            32, 0, 0, 0, cudaChannelFormatKindFloat);
      cudaArray* cuArray;
      cudaMallocArray(&cuArray, &channelDesc, width, height);
// Copy to device memory some data located at address h_data
// in host memory
      cudaMemcpyToArray(cuArray,0,0,h_data,size,
            cudaMemcpyHostToDevice);
// Set texture parameters
      texRef.addressMode[0] = cudaAddressModeWrap;
      texRef.addressMode[1] = cudaAddressModeWrap;
      texRef.filterMode = cudaFilterModeLinear;
      texRef.normalized = true;
// Bind the array to the texture
      cudaBindTextureToArray(texRef, cuArray, channelDesc);
// Allocate result of transformation in device memory
      float* output;
      cudaMalloc((void**)&output, width * height * sizeof(float));
// Invoke kernel
      dim3 dimBlock(16, 16);
      dim3 dimGrid((width + dimBlock.x – 1) / dimBlock.x,
            (height + dimBlock.y – 1) / dimBlock.y);
      transformKernel<<<dimGrid, dimBlock>>>(output, width,
            height, angle);
// Free device memory
      cudaFreeArray(cuArray);
      cudaFree(output);
}
```

3.2.5 分页锁定主机存储器

运行时提供了分配和释放分页锁定主机存储器（也称为 pinned）的函数 cudaHostAlloc()和 cudaFreeHost()，分页锁定主机存储器与常规的使用 malloc()分配的可分页的主机存储器不同。

使用分页锁定主机存储器有许多优点：

- 如 3.2.6 节提到的，在某些设备上，设备存储器和分页锁定主机存储器间数据拷贝可与内核执行并发进行；
- 在一些设备上，分页锁定主机内存可映射到设备地址空间，减少了和设备间的数据拷贝，详见 3.2.5.3 节；
- 在有前端总线的系统上，如果主机存储器是分页锁定的，主机存储器和设备存储器间的带宽会高些，如果再加上 3.2.5.2 节所描述的写结合（write-combining）的话，带宽会更高。

然而分页锁定主机存储器是稀缺资源，所以可分页内存分配得多的话，分配会失败。另外由于减少了系统可分页的物理存储器数量，分配太多的分页锁定内存会降低系统的整体性能。

SDK 中的 simple zero-copy 例子中有分页锁定 API 的详细文档。

3.2.5.1 可分享存储器（portable memory）

一块分页锁定存储器可被任何主机线程使用，但是默认的情况下，只有分配它的线程可以使用它。为了让所有线程可以使用它，可以在使用 cudaHostAlloc()分配时传入 cudaHostAllocPortable 标签。

3.2.5.2 写结合存储器

默认情况下，分页锁定主机存储器是可缓存的。可以在使用 cudaHostAlloc()分配时传入 cudaHostAllocWriteCombined 标签使其被分配为写结合的。写结合存储器没有一级和二级缓存资源，所以应用的其它部分就有更多的缓存可用。另外写结合存储器在通过 PCI-e 总线传输时不会被监视（snoop），这能够获得高达 40%的传输加速。

从主机读取写结合存储器极其慢，所以写结合存储器应当只用于那些主机只写的存储器。

3.2.5.3 被映射存储器

在一些设备上，在使用 cudaHostAlloc()分配时传入 cudaHostAllocMapped 标签可分配一块被映射到设备地址空间的分页锁定主机存储器。这块存储器有两个地址：一个在主机存储器上，一个在设备存储器上。主机指针是从 cudaHostAlloc（）返回的，设备指针可通过 cudaHostGetDevicePointer（）函数检索到，可以使用这个设备指针在内核中访问这块存储器。

从内核中直接访问主机存储器有许多优点：

- 无须在设备上分配存储器，也不用在这块存储器和主机存储器间显式传输数据；数据传输是在内核需要的时候隐式进行的。
- 无须使用流（参见 3.2.6.4 节）重叠数据传输和内核执行；数据传输和内核执行自动重叠。

由于被映射分页锁定存储器在主机和设备间共享，应用必须使用流或事件（参见 3.2.6 节）来同步存储器访问以避免任何潜在的读后写，写后读，或写后写危害。

一块分页锁定存储器可同时分配为被映射的和可分享的（见 3.2.5.1 节），这种情况下，每个要映射这块存储器的主机线程必须调用 cudaHostGetDevicePointer()检索设备指针，因为每个主机线程持有的设备指针一般不同。

为了在给定的主机线程中能够检索到被映射分页锁定存储器的设备指针，必须在调用任何 CUDA 运行时函数前调用 cudaSetDeviceFlags()，并传入 cudaDeviceMapHost 标签。否

则，cudaHostGetDevicePointer()将会返回错误。

如果设备不支持被映射分页锁定存储器，cudaHostGetDevicePointer()将会返回错误。

应用可能会查询设备是否支持映射分页锁定主机存储器，可以使用函数cudaGetDeviceProperties()检查 canMapHostMemory 属性。

注意：从主机和其它设备的角度看，操作被映射分页锁定存储器的原子函数（5.4.3 和 B.10 节）不是原子的。

3.2.6 异步并发执行

3.2.6.1 主机和设备间异步执行

为了易于使用主机和设备间的异步执行，一些函数是异步的：在设备完全完成任务前，控制已经返回给主机线程了。它们是：

- 内核发射；
- 存储器拷贝函数中带有 Async 后缀的；
- 设备间数据拷贝函数；
- 设置设备存储器的函数；

程序员可通过将 CUDA_LAUNCH_BLOCKING 环境变量设置为 1 来全局禁用所有运行在系统上的应用的异步内核发射。提供这个特性只是为了调试，永远不能作为使软件产品运行得可靠的方式。

当应用通过 CUDA 调试器或 CUDA profiler 运行时，所有的内核发射都是同步的。

3.2.6.2 数据传输和内核执行重叠

一些计算能力 1.1 或更高的设备可在内核执行时，在分页锁定存储器和设备存储器之间拷贝数据。应用可以通过调用 cudaGetDeviceProperties()函数检查 deviceOverlap 属性查询这种能力。这种能力目前只支持不涉及 CUDA 数组和使用 cudaMallocPitch()分配的二维数组的存储器拷贝（参见 3.2.1 节）。

3.2.6.3 并发内核执行

一些计算能力 2.0 的设备可并发执行多个内核。应用可以通过调用cudaGetDeviceProperties()函数检查 concurrentKernels 属性以查询这种能力。

设备最大可并发执行的内核数目是 4。

来自不同 CUDA 上下文的内核不能并发执行。

使用许多纹理或大量本地存储器的内核和其它内核并发执行的可能性比较小。

3.2.6.4 流

应用通过流管理并发。流是一系列顺序执行的命令。另外，流之间相对无序的或并发的执行它们的命令；这种行为是没有保证的，而且不能作为正确性的的保证（如内核间的通信没有定义）。

可以通过创建流对象来定义流，且可指定它作为一系列内核发射和设备主机间存储器拷贝的流参数。下面的代码创建了两个流且在分页锁定存储器中分配了一个名为 hostPtr 的浮点数组。

```
cudaStream_t stream[2];
for (int i = 0; i < 2; ++i)
    cudaStreamCreate(&stream[i]);
float* hostPtr;
cudaMallocHost((void**)&hostPtr, 2 * size);
```

下面的代码定义的每个流是一个由一次主机到设备的传输，一次内核发射，一次设备到主机的传输组成的系列。

```
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();
```

每个流将它的 hostPtr 输入数组的部分拷贝到设备存储器数组 inputdevPtr，调用 MyKernel()内核处理 inputDevPtr，然后将结果 outputDevPtr 传输回 hostPtr 同样的部分。使用两个流处理 hostPtr 允许一个流的传输和另一个流的执行重叠。为了使用重叠 hostPtr 必须指向分页锁定主机存储器。

最后调用的 cudaThreadSynchronize()保证所有的流在进一步执行前已经完成。这个函数强制运行时等待所有流中的任务都完成。cudaStreamSynchronize()强制运行时等待某个流中的任务都完成。可用于同步主机和特定流，同时允许其它流继续执行。cudaStreamQuery()用于查询流中的所有之前的命令是否已经完成。为了避免不必要的性能损失，这些函数最好用于计时或隔离失败的发射或存储器拷贝。

调用 cudaStreamDestroy()来释放流。

```
for (int i = 0; i < 2; ++i)
cudaStreamDestroy(stream[i]);
```

cudaStreamDestroy()等待指定流中所有在前的任务完成，然后释放流并将控制权返回给主机线程。

如果是下面情况，来自不同流的两个命令也不能并发，或分页锁定主机存储器分配，设备存储器分配，设备存储器设置，设备之间拷贝，或主机线程在 0 号流中调用的任何它们之间的 CUDA 命令（包含没有指定任何流参数的内核发射和设备和主机间存储器拷贝）。

任何需要依赖检测以确定内核发射是否完成的操作会阻塞 CUDA 上下文中后面任何流中所有的内核发射直至被检测的内核发射完成。需要依赖检测的操作包括同一个流中的一些其它类似被检查的发射的命令和流中的任何 cudaStreamQuery()调用。因此，应用应当遵守这些指导以提升潜在的内核并发执行：

- 所有独立操作应当在依赖操作之前发出，
- 任何类型同步尽量延后。

G.4.1 节描述的一级缓存和共享存储器的配置切换为所有未完成的内核发射插入了一个设备端同步栅栏。

3.2.6.5 事件

通过在应用的任意点上异步地记载事件和查询事件是否真正被记载，运行时提供了精密地监测设备运行进度的方式，事件同时也提供精确计时功能。当事件记载点前面，事件指定

的流中的所有任务或者命令全部完成时，事件被记载。只有记载点之前所有的流中的任务/命令都已完成，0 号流的事件才会记载。

下面的代码创建了两个事件：

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

按下面的方式，这些事件可用于计算上节代码的运行时间：

```
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);
```

可用这种方式销毁事件：

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

3.2.6.6 同步调用

直到设备真正完成任务，同步函数调用的控制权才会返回给主机线程。在主机线程执行任何其它 CUDA 调用前，通过调用 cudaSetDeviceFlags()并传入指定标签（参见参考手册）可以指定主机线程的让步，阻塞，或自旋状态。

3.2.7 图形学互操作性

一些 OpenGL 和 Direct3D 的资源可被映射到 CUDA 地址空间，要么使 CUDA 可以读 OpenGL 或 Direct3D 写的数据，要么使 CUDA 写数据供 OpenGL 或 Direct3D 消费。

资源必须先在 CUDA 中注册，才能被 3.2.7.1 和 3.2.7.2 提到的函数映射。这些函数返回一个指向 cudaGraphicsResource 类型结构体的 CUDA 图形资源。资源注册是潜在高消耗的，因此通常每个资源只注册一次。可以使用 cudaGraphicsUnregisterResource()解注册 CUDA 图形资源。

一旦资源被注册到 CUDA，就可以按需要被任意次的映射和解映射，映射和解映射使用 cudaGraphicsMapResources()和 cudaGraphicsUnmapResources()。可以使用 cudaGraphicsResourceSetMapFlags()来指定资源用处（只读，只写），CUDA 驱动可以据此优化资源管理。

可以获得 cudaGraphicsResourceGetMappedPointer()为缓冲区返回的设备地址空间和 cudaGraphicsSubResourceGetMappedArray()为 CUDA 数组返回的设备地址空间，内核通过读写这些空间读写被映射资源。

通过 OpenGL 或 Direct3D 访问被映射到 CUDA 的 OpenGL 或 Direct3D 的资源，其结果未定义。

3.2.7.1 和 3.2.7.2 节给出了每种图形 API 的特性和一些代码例子。

3.2.7.1 OpenGL 互操作性

和 OpenGL 互操作要求在其它任何运行时函数调用前，使用 cudaGLSetGLDevice()指定 CUDA 设备。注意 cudaSetDevice()和 cudaGLSetDevice()是相互排斥的。

可以被映射到 CUDA 地址空间的 OpenGL 资源有 OpenGL 缓冲区、纹理和渲染缓存对象。

使用 cudaGraphicsGLRegisterBuffer()注册缓冲对象。在 CUDA 中，缓冲对象表现为设备指针，因此可以在内核中读写或通过 cudaMemcpy()调用。

纹理或渲染缓存对象使用 cudaGraphicsGLRegisterImage()注册。在 CUDA 中，它们表现为 CUDA 数组，可绑定到纹理参考，可被内核读写或通过 cudaMemcpy2D()调用。cudaGraphicsGLRegisterImage()使用内置的 float 类型（例如，GL_RGBA_FLOAT32）和非归一化整数（例如 GL_RGBA8UI）支持所有纹理格式。请注意，由于 GL_RGBA8UI 是 OpenGL3.0 纹理格式，只能被着色器写，不能被固定功能的管线写。

下面的代码使用内核动态的修改一个存储在顶点缓冲对象中的二维 width*height 顶点网格。

```
GLuint positionsVBO;
struct cudaGraphicsResource* positionsVBO_CUDA;

int main()
{
    // Explicitly set device
    cudaGLSetGLDevice(0);
    // Initialize OpenGL and GLUT
    ...
    glutDisplayFunc(display);
    // Create buffer object and register it with CUDA
    glGenBuffers(1, positionsVBO);
    glBindBuffer(GL_ARRAY_BUFFER, &vbo);
    unsigned int size = width * height * 4 * sizeof(float);
    glBufferData(GL_ARRAY_BUFFER, size, 0, GL_DYNAMIC_DRAW);
    glBindBuffer(GL_ARRAY_BUFFER, 0);
    cudaGraphicsGLRegisterBuffer(&positionsVBO_CUDA, positionsVBO,
    cudaGraphicsMapFlagsWriteDiscard);
    // Launch rendering loop
    glutMainLoop();
}
```

```
void display()
{
    // Map buffer object for writing from CUDA
    float4* positions;
    cudaGraphicsMapResources(1, &positionsVBO_CUDA, 0);
    size_t num_bytes;
    cudaGraphicsResourceGetMappedPointer((void**)&positions, &num_bytes,
    positionsVBO_CUDA));
    // Execute kernel
    dim3 dimBlock(16, 16, 1);
    dim3 dimGrid(width / dimBlock.x, height / dimBlock.y, 1);
    createVertices<<<dimGrid, dimBlock>>>(positions, time, width, height);
    // Unmap buffer object
    cudaGraphicsUnmapResources(1, &positionsVBO_CUDA, 0);
    // Render from buffer object
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glBindBuffer(GL_ARRAY_BUFFER, positionsVBO);
    glVertexPointer(4, GL_FLOAT, 0, 0);
    glEnableClientState(GL_VERTEX_ARRAY);
    glDrawArrays(GL_POINTS, 0, width * height);
    glDisableClientState(GL_VERTEX_ARRAY);
    // Swap buffers
    glutSwapBuffers();
    glutPostRedisplay();
}
void deleteVBO()
{
    cudaGraphicsUnregisterResource(positionsVBO_CUDA);
    glDeleteBuffers(1, &positionsVBO);
}
__global__ void createVertices(float4* positions, float time, unsigned int width, unsigned int height)
{
    unsigned int x = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned int y = blockIdx.y * blockDim.y + threadIdx.y;
    // Calculate uv coordinates
    float u = x / (float)width;
    float v = y / (float)height;
    u = u * 2.0f - 1.0f;
    v = v * 2.0f - 1.0f;
    // calculate simple sine wave pattern
    float freq = 4.0f;
    float w = sinf(u * freq + time) * cosf(v * freq + time) * 0.5f;
    // Write positions
    positions[y * width + x] = make_float4(u, w, v, 1.0f);
}
```

在 Windows 系统上和对于 Quadro 显卡，可以用 cudaWGLGetDevice()检索关联到 wglEnumGpusNV()返回的句柄的 CUDA 设备。Quadro 显卡与 OpenGL 的互操作性能比 GeForce 和 Tesla 要好。在一个多 GPU 的系统中，在 Quadro GPU 上运行 OpenGL 渲染，在其它的 GPU 进行 CUDA 计算。

3.2.7.2 Direct3D 互操作性

Direct3D 互操作性支持 Direct3D 9，Direct3D 10，和 Direct3D 11。

一个 CUDA 上下文一次只能和一个 Direct3D 设备互操作，且 CUDA 上下文和 Direct3D 设

备必须在同一个 GPU 上创建，而且 Direct3D 设备必须使用 D3DCREATE_HARDWARE_VERTEXPROCESSING 标签创建。

和 Direct3D 的互操作性要求：在任何其它的运行时函数调用前，使用 cudaD3D9SetDirect3DDevice()，cudaD3D10SetDirect3DDevice() 和 cudaD3D11SetDirect3DDevice()指定 Direct3D 设备。可用 cudaD3D9GetDevice(),cudaD3D10GetDevice() 和 cudaD3D11GetDevice()检索关联到一些适配器的 CUDA 设备。

可以被映射到 CUDA 地址空间的 Direct3D 资源有 Direct3D 缓冲区，纹理和表面。可以使用 cudaGraphicsD3D9RegisterResource()， cudaGraphicsD3D10RegisterResource()和 cudaGraphicsD3D11RegisterResource()注册这些资源。

下面的代码使用内核动态的修改一个存储在顶点缓冲对象中的二维 width*height 网格顶点。

Direct3D 9 版本

```
IDirect3D9* D3D;
IDirect3DDevice9* device;
struct CUSTOMVERTEX {
    FLOAT x, y, z;
    DWORD color;
};
IDirect3DVertexBuffer9* positionsVB;
struct cudaGraphicsResource* positionsVB_CUDA;

int main() {
    // Initialize Direct3D
    D3D = Direct3DCreate9(D3D_SDK_VERSION);
    // Get a CUDA-enabled adapter
    unsigned int adapter = 0;
    for (; adapter < g_pD3D->GetAdapterCount(); adapter++) {
        D3DADAPTER_IDENTIFIER9 adapterId;
        g_pD3D->GetAdapterIdentifier(adapter, 0, &adapterId);
        int dev;
        if (cudaD3D9GetDevice(&dev, adapterId.DeviceName) == cudaSuccess)
            break;
    }

    // Create device
    ...
    D3D->CreateDevice(adapter, D3DDEVTYPE_HAL, hWnd,
    D3DCREATE_HARDWARE_VERTEXPROCESSING, &params, &device);
    // Register device with CUDA
    cudaD3D9SetDirect3DDevice(device);
    // Create vertex buffer and register it with CUDA
    unsigned int size = width * height * sizeof(CUSTOMVERTEX);
    device->CreateVertexBuffer(size, 0, D3DFVF_CUSTOMVERTEX,
                               D3DPOOL_DEFAULT,&positionsVB, 0);
    cudaGraphicsD3D9RegisterResource(&positionsVB_CUDA, positionsVB,
        cudaGraphicsRegisterFlagsNone);
    cudaGraphicsResourceSetMapFlags(positionsVB_CUDA,
cudaGraphicsMapFlagsWriteDiscard);
    // Launch rendering loop
    while (...) {
    ...
        Render();
...
    }
}
```

```
void Render() {
    // Map vertex buffer for writing from CUDA
    float4* positions;
    cudaGraphicsMapResources(1, &positionsVB_CUDA, 0);
    size_t num_bytes;
    cudaGraphicsResourceGetMappedPointer((void**)&positions, &num_bytes,
        positionsVB_CUDA));
    // Execute kernel
    dim3 dimBlock(16, 16, 1);
    dim3 dimGrid(width / dimBlock.x, height / dimBlock.y, 1);
    createVertices<<<dimGrid, dimBlock>>>(positions, time, width, height);
    // Unmap vertex buffer
    cudaGraphicsUnmapResources(1, &positionsVB_CUDA, 0);
    // Draw and present
    ...
}

void releaseVB() {
    cudaGraphicsUnregisterResource(positionsVB_CUDA);
    positionsVB->Release();
}

__global__ void createVertices(float4* positions, float time, unsigned int width, unsigned int height) {
    unsigned int x = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned int y = blockIdx.y * blockDim.y + threadIdx.y;
    // Calculate uv coordinates
    float u = x / (float)width;
    float v = y / (float)height;
    u = u * 2.0f - 1.0f;
    v = v * 2.0f - 1.0f;
    // Calculate simple sine wave pattern
    float freq = 4.0f;
    float w = sinf(u * freq + time) * cosf(v * freq + time) * 0.5f;
    // Write positions
positions[y * width + x] = make_float4(u, w, v, __int_as_float(0xff00ff00));
}
```

Direct3D 10 版本：

```
ID3D10Device* device;
struct CUSTOMVERTEX {
    FLOAT x, y, z;
    DWORD color;
};
ID3D10Buffer* positionsVB;
struct cudaGraphicsResource* positionsVB_CUDA;
int main() {
    // Get a CUDA-enabled adapter
    IDXGIFactory* factory;
    CreateDXGIFactory(__uuidof(IDXGIFactory), (void**)&factory);
    IDXGIAdapter* adapter = 0;
    for (unsigned int i = 0; !adapter; ++i) {
        if (FAILED(factory->EnumAdapters(i, &adapter))
            break;
        int dev;
        if (cudaD3D10GetDevice(&dev, adapter) == cudaSuccess)
            break;
        adapter->Release();
    }
```

```
        factory->Release();
        // Create swap chain and device
        D3D10CreateDeviceAndSwapChain(adapter, D3D10_DRIVER_TYPE_HARDWARE, 0,
                D3D10_CREATE_DEVICE_DEBUG, D3D10_SDK_VERSION, &swapChainDesc,
                &swapChain, &device);
        adapter->Release();
        // Register device with CUDA
        cudaD3D10SetDirect3DDevice(device);
        // Create vertex buffer and register it with CUDA
        unsigned int size = width * height * sizeof(CUSTOMVERTEX);
        D3D10_BUFFER_DESC bufferDesc;
        bufferDesc.Usage = D3D10_USAGE_DEFAULT;
        bufferDesc.ByteWidth = size;
        bufferDesc.BindFlags = D3D10_BIND_VERTEX_BUFFER;
        bufferDesc.CPUAccessFlags = 0;
        bufferDesc.MiscFlags = 0;
        device->CreateBuffer(&bufferDesc, 0, &positionsVB);
        cudaGraphicsD3D10RegisterResource(&positionsVB_CUDA, positionsVB,
              cudaGraphicsRegisterFlagsNone);
        cudaGraphicsResourceSetMapFlags(positionsVB_CUDA, cudaGraphicsMapFlagsWriteDiscard);
        / Launch rendering loop
        while (...) {
              Render();
        }
}
void Render() {
        // Map vertex buffer for writing from CUDA
        float4* positions;
        cudaGraphicsMapResources(1, &positionsVB_CUDA, 0);
        size_t num_bytes;
        cudaGraphicsResourceGetMappedPointer((void**)&positions, &num_bytes,
              positionsVB_CUDA));
        // Execute kernel
        dim3 dimBlock(16, 16, 1);
        dim3 dimGrid(width / dimBlock.x, height / dimBlock.y, 1);
        createVertices<<<dimGrid, dimBlock>>>(positions, time, width, height);
        // Unmap vertex buffer
        cudaGraphicsUnmapResources(1, &positionsVB_CUDA, 0);
        // Draw and present
}
void releaseVB() {
        cudaGraphicsUnregisterResource(positionsVB_CUDA);
        positionsVB->Release();
}
__global__ void createVertices(float4* positions, float time, unsigned int width, unsigned int height) {
        unsigned int x = blockIdx.x * blockDim.x + threadIdx.x;
        unsigned int y = blockIdx.y * blockDim.y + threadIdx.y;
        // Calculate uv coordinates
        float u = x / (float)width;
        float v = y / (float)height;
        u = u * 2.0f - 1.0f;
        v = v * 2.0f - 1.0f;
        // Calculate simple sine wave pattern
        float freq = 4.0f;
        float w = sinf(u * freq + time) * cosf(v * freq + time) * 0.5f;
        // Write positions
        positions[y * width + x] = make_float4(u, w, v, __int_as_float(0xff00ff00));
}
```

Direct3D 11 版本：

```
ID3D11Device* device;
struct CUSTOMVERTEX {
    FLOAT x, y, z;
    DWORD color;
};
ID3D11Buffer* positionsVB;
struct cudaGraphicsResource* positionsVB_CUDA;

int main() {
    // Get a CUDA-enabled adapter
    IDXGIFactory* factory;
    CreateDXGIFactory(__uuidof(IDXGIFactory), (void**)&factory);
    IDXGIAdapter* adapter = 0;
    for (unsigned int i = 0; !adapter; ++i) {
        if (FAILED(factory->EnumAdapters(i, &adapter))
            break;
        int dev;
        if (cudaD3D11GetDevice(&dev, adapter) == cudaSuccess)
            break;
        adapter->Release();
}
    factory->Release();
    // Create swap chain and device
    …
    sFnPtr_D3D11CreateDeviceAndSwapChain(adapter, D3D11_DRIVER_TYPE_HARDWARE, 0,
        D3D11_CREATE_DEVICE_DEBUG, featureLevels, 3, D3D11_SDK_VERSION,
        &swapChainDesc, &swapChain, &device, &featureLevel, &deviceContext);
    adapter->Release();
    // Register device with CUDA
    cudaD3D11SetDirect3DDevice(device);
    // Create vertex buffer and register it with CUDA
    unsigned int size = width * height * sizeof(CUSTOMVERTEX);
    D3D11_BUFFER_DESC bufferDesc;
    bufferDesc.Usage = D3D11_USAGE_DEFAULT;
    bufferDesc.ByteWidth = size;
    bufferDesc.BindFlags = D3D11_BIND_VERTEX_BUFFER;
    bufferDesc.CPUAccessFlags = 0;
    bufferDesc.MiscFlags = 0;
    device->CreateBuffer(&bufferDesc, 0, &positionsVB);
    cudaGraphicsD3D11RegisterResource(&positionsVB_CUDA,
    positionsVB,
    cudaGraphicsRegisterFlagsNone);
    cudaGraphicsResourceSetMapFlags(positionsVB_CUDA,
    cudaGraphicsMapFlagsWriteDiscard);
    // Launch rendering loop
    while (...) { ...
        Render(); ...
    }
}

void Render() {
    // Map vertex buffer for writing from CUDA
    float4* positions;
    cudaGraphicsMapResources(1, &positionsVB_CUDA, 0);
    size_t num_bytes;
    cudaGraphicsResourceGetMappedPointer((void**)&positions, &num_bytes,
        positionsVB_CUDA));
```

```
    // Execute kernel
    dim3 dimBlock(16, 16, 1);
    dim3 dimGrid(width / dimBlock.x, height / dimBlock.y, 1);
    createVertices<<<dimGrid, dimBlock>>>(positions, time, width, height);
    // Unmap vertex buffer
    cudaGraphicsUnmapResources(1, &positionsVB_CUDA, 0);
    // Draw and present
    ...
}

void releaseVB() {
    cudaGraphicsUnregisterResource(positionsVB_CUDA);
    positionsVB->Release();
}
__global__ void createVertices(float4* positions, float time, unsigned int width, unsigned int height){
    unsigned int x = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned int y = blockIdx.y * blockDim.y + threadIdx.y;
    // Calculate uv coordinates
    float u = x / (float)width;
    float v = y / (float)height;
    u = u * 2.0f - 1.0f;
    v = v * 2.0f - 1.0f;
    // Calculate simple sine wave pattern
    float freq = 4.0f;
    float w = sinf(u * freq + time)* cosf(v * freq + time) * 0.5f;
    // Write positions
    positions[y * width + x] =make_float4(u, w, v, __int_as_float(0xff00ff00));
}
```

3.2.8 错误处理

所有的运行时函数都返回错误码，但对于异步函数（参见 3.2.6 节），由于会在任务结束前返回，因此错误码不能报告异步调用的错误；错误码只报告在任务执行之前的错误，典型的错误有关参数有效性；如果异步调用出错，错误将会在后面某个无关的函数调用中出现。

唯一能够检查异步调用出错的方式是通过在异步调用函数后面使用 cudaThreadSynchronize()同步（或使用 3.2.6 节介绍的其它同步机制），然后检查 cudaThreadSynchronize()的返回值。

运行时为每个主机线程维护着一个初始化为 cudaSuccess 的错误变量，每次错误发生（可以是参数不正确或异步错误）时，该变量会被错误码重写。cudaGetLastError()会返回这个变量，并将它重新设置为 cudaSuccess。

内核发射不返回任何错误码，所以应当在内核发射后立刻调用 cudaGetLastError()检测发射前错误。为保证 cudaGetLastError()返回的错误值不是由于内核发射前错误导致的，必须保证运行时错误变量在内核发射前被设置为 cudaSuccess，可以通过在内核发射前调用 cudaGetLastError()实现。内核发射是异步的，因此为了检测异步错误，应用必须在内核发射和 cudaGetLastError()之间同步。

注意 cudaStreamQuery()可能返回 cudaErrorNotReady，而由于 cudaEventQuery()没有考虑错误，因此不会被 cudaGetLastError()报告。

3.2.9 使用设备模拟模式调试

CUDA-GDB 可以用于调试计算能力大于 1.0 的设备（参看 CUDA-GDB 用户手册获得支持平台）。编译器和运行时也支持模拟模式调试，模拟模式可以在没有 CUDA 支持的设备上运行。当使用模拟模式编译应用时（使用-deviceemu 选项），设备代码为主机编译，并在主机上运行，允许程序员使用主机上的本地调试器，就好像是主机程序一样。预处理宏 __DEVICE_EMULATION__是为这种模式定义的。使用了任何库的所有应用代码必须一致的用设备模拟模式或设备执行模式编译。链接设备模拟模式编译的代码和设备执行模式编译的代码会返回下面的运行时初始化错误：cudaErrorMixedDeviceExecution。

在设备模拟模式下运行应用，运行时模拟编程模型。对于块内的每个线程，运行时在主机上建立一个线程。程序员要确定：

- 主机能够运行每个块的最大线程数，外加一主线程。
- 足够的可用存储器用于运行所有线程，已知每个线程都要占据 256 KB 栈空间。

设备模拟模式提供的许多特性让他成为一种有效的调试工具：

- 通过使用主机的本地调试支持，程序员可利用调试器支持的所有特性，例如断点设置和数据检查。
- 由于设备代码编译后在主机上运行，故可调用无法在设备上运行的代码扩展功能，如文件输入和输出操作或输出到屏幕（printf() 等）。
- 由于所有数据都位于主机上，因而可从设备或主机代码可读取任何特定于设备或主机的数据；类似地，设备和主机可调用任何设备或主机函数。
- 万一误用了内置同步函数，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。因此，设备模拟模式在查找算法错误时非常有用，但是很难发现某些错误：

- 竞争条件在设备模拟模式中体现的可能很小，因为同时执行的线程数量要比实际运行在设备上的少得多。
- 在主机上解引用指向全局存储器的指针或在设备上解引用指向主机存储器的指针时，设备执行几乎必然以某种没有定义的方式失败，而设备模拟能得到正确的结果。
- 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得到的结果并不完全相同。这是可预见的，因为一般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。

特别地，某些主机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，这潜在地导致精确性的显著差异。当发生这种情况时，程序员可尝试下面的任何一种方法，但没有任何一种方法能确保成功：

- 声明一些易变的浮点变量，强制单精度存储；
- 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()，使用这些代码环绕原代码的一部分，强制使用单精度浮点计算：

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

或

```
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，使用下面代码，重置原始控制字：

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

另外，对于单精度浮点数，计算能力 1.1 的设备不支持非正规数（参见附录 G），而主机平台通常支持。这可能会导致设备模拟模式和设备执行模式的结果显著不同，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，束的大小等于 1（参见的 4.1 节了解束的定义）。因而，束表决（warp vote）函数的结果将与设备执行模式不同。

3.3 驱动 API

驱动 API 是基于句柄的，命令式的：大多数对象通过不透明的句柄引用，通过将此句柄指定给函数以操作对象。

驱动 API 可用的对象总结如下表 3-1

表 3-1. 驱动 API 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，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

驱动 API 的实现在 nvcuda 动态库中，其所有的入口点前缀为 cu。

在调用任何其它驱动 API 前必须用 cuInit()初始化驱动 API。然后必须创建一个 CUDA 上下文，该上下文关联到特定设备并成为主机线程的当前上下文，详见 3.3.1 节。

在 CUDA 上下文中，内核必须显式的作为 PTX 或二进制对象被主机代码加载，参见 3.3.2 节。用 C 写的内核必须独立的编译成 PTX 或二进制对象。发射内核使用的 API 入口点可参见 3.3.3 节。

任何想要在未来的设备架构上运行的应用必须加载 PTX，而不是二进制代码。这是因为二进制是架构相关的，因此和未来的架构不兼容，而 PTX 代码可被驱动在加载时编译成二进制代码。

下面是 2.1 节例子的驱动 API 实现：

```
int main() {
    int N = ...;
    size_t size = N * sizeof(float);
    // Allocate input vectors h_A and h_B in host memory
    float* h_A = (float*)malloc(size);
    float* h_B = (float*)malloc(size);
    // Initialize input vectors
    ...
    // Initialize
    cuInit(0);
    // Get number of devices supporting CUDA
    int deviceCount = 0;
    cuDeviceGetCount(&deviceCount);
    if (deviceCount == 0) {
        printf("There is no device supporting CUDA.\n");
        exit (0);
    }
    // Get handle for device 0
    CUdevice cuDevice;
    cuDeviceGet(&cuDevice, 0);
    // Create context
    CUcontext cuContext;
    cuCtxCreate(&cuContext, 0, cuDevice);
    // Create module from binary file
    CUmodule cuModule;
    cuModuleLoad(&cuModule, “VecAdd.ptx”);
    // Allocate vectors in device memory
    CUdeviceptr d_A; cuMemAlloc(&d_A, size);
    CUdeviceptr d_B; cuMemAlloc(&d_B, size);
    CUdeviceptr d_C; cuMemAlloc(&d_C, size);
    // Copy vectors from host memory to device memory
    cuMemcpyHtoD(d_A, h_A, size);
    cuMemcpyHtoD(d_B, h_B, size);
    // Get function handle from module
    CUfunction vecAdd;
    cuModuleGetFunction(&vecAdd, cuModule, "VecAdd");
    // Invoke kernel
    #define ALIGN_UP(offset, alignment) \
        (offset) = ((offset) + (alignment) – 1) & ~((alignment) – 1)
    int offset = 0; void* ptr;
    ptr = (void*)(size_t)A;
    ALIGN_UP(offset, __alignof(ptr));
    cuParamSetv(vecAdd, offset, &ptr, sizeof(ptr));
    offset += sizeof(ptr);
    ptr = (void*)(size_t)B;
    ALIGN_UP(offset, __alignof(ptr));
    cuParamSetv(vecAdd, offset, &ptr, sizeof(ptr));
    offset += sizeof(ptr);
    ptr = (void*)(size_t)C;
    ALIGN_UP(offset, __alignof(ptr));
    cuParamSetv(vecAdd, offset, &ptr, sizeof(ptr));
    offset += sizeof(ptr);
    ALIGN_UP(offset, __alignof(N));
    cuParamSeti(vecAdd, offset, N);
    offset += sizeof(N);
    cuParamSetSize(vecAdd, offset);
    int threadsPerBlock = 256;
    int blocksPerGrid = (N + threadsPerBlock – 1) / threadsPerBlock;
    cuFuncSetBlockShape(vecAdd, threadsPerBlock, 1, 1);
    cuLaunchGrid(vecAdd, blocksPerGrid, 1);
    ...
}
```

全部代码可在 SDK 中的 vectorAddDrv 例子中找到。

3.3.1 上下文

CUDA 上下文类似于 CPU 的进程。所有资源和在驱动程序 API 中执行的操作都封装在 CUDA 上下文内，在销毁上下文时，系统将自动清理这些资源。除了模块和纹理参考之类的对象外，每个上下文都有自己不同的 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空间。

一个主机线程在某时只能有一个当前设备上下文。当使用 cuCtxCreate() 创建上下文时，它将成为主机调用线程的当前上下文。如果有效上下文不是线程的当前上下文，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。

每个主机线程都有一个当前上下文堆栈。cuCtxCreate() 将新上下文压入栈顶。可调用 cuCtxPopCurrent() 分离上下文与主机线程。随后此上下文将成为“游魂（floating）”上下文，可作为任意主机线程的当前上下文入栈。cuCtxPopCurrent() 还可重建之前的当前上下文（如果有）。

此外还会为每个上下文维护一个引用计数（usage count）。cuCtxCreate()创建一个将引用计数为 1 的上下文。cuCtxAttach()递增计数，而 cuCtxDetach() 则递减。当调用 cuCtxDetach()时计数为 0 或 cuCtxDestroy()，上下文将被销毁。

引用计数有利于同一上下文中第三方授权代码间的互操作。比如，如果载入了三个使用相同上下文的库，则每个库都将调用 cuCtxAttach() 来递增计数，并在库不再使用该上下文时调用 cuCtxDetach() 递减计数。对大多数库来说，应用应当在载入或初始化库之前创建一个上下文，通过这种方式，应用可使用自己的启发式（heuristics）方法来创建上下文，库只需在传递给它的上下文中简单操作。希望创建自己的上下文的库（其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 3-3 所示。

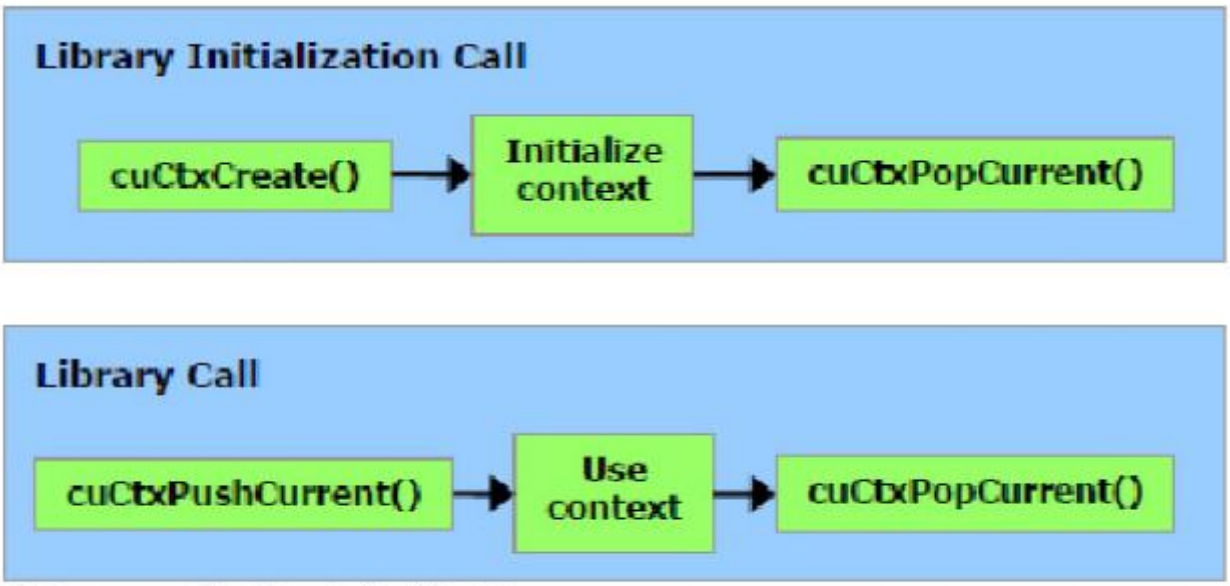


图3-3.库上下文管理

3.3.2 模块

模块是可动态加载的设备代码和数据包，和 Windows 中的 DLL 类似，是由 nvcc 输出的（参见 3.1 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，所以独立的第三方编写的模块可在同一 CUDA 上下文中配合。
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.ptx”);
CUfunction myKernel;
cuModuleGetFunction(&myKernel, cuModule, “MyKernel”);
```

下[illegible]

```
#define ERROR_BUFFER_SIZE 100
CUmodule cuModule;
CUptxas_option options[3];
void* values[3];
char* PTXCode = “some PTX code”;
options[0] = CU_ASM_ERROR_LOG_BUFFER;
values[0] = (void*)malloc(ERROR_BUFFER_SIZE);
options[1] = CU_ASM_ERROR_LOG_BUFFER_SIZE_BYTES;
values[1] = (void*)ERROR_BUFFER_SIZE;
options[2] = CU_ASM_TARGET_FROM_CUCONTEXT;
values[2] = 0;
cuModuleLoadDataEx(&cuModule, PTXCode, 3, options, values);
for (int i = 0; i < values[1]; ++i) {
    // Parse error string here
}
```

3.3.3 内核执行

cuFuncSetBlockShape() 为指定函数设置每个块的线程数，同时确定其 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发射内核时为内核提供的参数，调用 cuLaunchGrid() 或 cuLaunch()发射内核。

各 cuParam*() 函数的第二个参数指定参数在参数堆栈中的偏移。这个偏移量必须与设备代码中对参数类型的对齐要求相匹配。设备代码中内置向量类型的对齐要求列在表 B-1 中。对于所有其它基本类型，设备代码和主机代码的对齐要求一致，且可通过使用 _alignof()获得。唯一的例外是，当主机编译器将双精度和 long long（或者 64 位机的 long）对齐在单字边界而非双字边界（例如使用 gcc 的-mno-align-double 编译选项）时，因为在设备代码中，这些类型永远以双字对齐。另外，CUdeviceptr 是整形，但代表指针，所以它的对齐要求是__alignof(void*)。下面的代码使用宏调整每个参数的偏移以满足对齐要求。

```
#define ALIGN_UP(offset, alignment) \
      (offset) = ((offset) + (alignment) – 1) & ~((alignment) – 1)
int offset = 0;

int i;
ALIGN_UP(offset, __alignof(i));
cuParamSeti(cuFunction, offset, i);
offset += sizeof(i);

float4 f4;
ALIGN_UP(offset, 16); // float4's alignment is 16
cuParamSetv(cuFunction, offset, &f4, sizeof(f4));
offset += sizeof(f4);

char c;
ALIGN_UP(offset, __alignof(c));
cuParamSeti(cuFunction, offset, c);
offset += sizeof(c);

float f;
ALIGN_UP(offset, __alignof(f));
cuParamSeti(cuFunction, offset, f);
offset += sizeof(f);

CUdeviceptr dptr;
// void* should be used to determine CUdeviceptr's alignment
void* ptr = (void*)(size_t)dptr;
ALIGN_UP(offset, __alignof(ptr));
cuParamSetv(cuFunction, offset, &ptr, sizeof(ptr));
offset += sizeof(ptr);

float2 f2;
ALIGN_UP(offset, 8); // float2's alignment is 8
cuParamSetv(cuFunction, offset, &f2, sizeof(f2));
offset += sizeof(f2);
cuParamSetSize(cuFunction, offset);

cuFuncSetBlockShape(cuFunction, blockWidth, blockHeight, 1);
cuLaunchGrid(cuFunction, gridWidth, gridHeight);
```

结构体的对齐要求等于它的域的最大对齐要求。包含内置向量类型，CUdeviceptr，或非对齐双精度和 long long 的对齐要求在主机和设备中可能不同。这种结构体的填充也不同。例如，下面的结构体在主机上根本不加填充，但在设备中会在 f 后加上 12 个字节的填充，因为对 f4 域的对齐要求是 16。

```
typedef struct {
      float f;
      float4 f4;
} myStruct;
```

myStruct 类型的任何参数必须独立调用 cuParam*()传递，例如：

```
myStruct s;
int offset = 0;
cuParamSetv(cuFunction, offset, &s.f, sizeof(s.f));
offset += sizeof(s.f);
ALIGN_UP(offset, 16); // float4's alignment is 16
cuParamSetv(cuFunction, offset, &s.f4, sizeof(s.f4));
offset += sizeof(s.f4);
```

3.3.4 设备存储器

线性存储器使用 cuMemAlloc()或 cuMemAllocPitch()分配，使用 cuMemFree()释放。下面是 3.2.1 节代码的驱动 API 版本：

```
// Host code
int main() {
    // Initialize
    if (cuInit(0) != CUDA_SUCCESS)
        exit (0);
    // Get number of devices supporting CUDA
    int deviceCount = 0;
    cuDeviceGetCount(&deviceCount);
    if (deviceCount == 0) {
        printf("There is no device supporting CUDA.\n");
        exit (0);
    }
    // Get handle for device 0
    CUdevice cuDevice = 0; cuDeviceGet(&cuDevice, 0);
     Create context
    CUcontext cuContext; cuCtxCreate(&cuContext, 0, cuDevice);
    // Create module from binary file
    CUmodule cuModule; cuModuleLoad(&cuModule, “VecAdd.ptx”);
    // Get function handle from module
    CUfunction vecAdd;
    cuModuleGetFunction(&vecAdd, cuModule, "VecAdd");
    // Allocate vectors in device memory
    size_t size = N * sizeof(float);
    CUdeviceptr d_A; cuMemAlloc(&d_A, size);
    CUdeviceptr d_B; cuMemAlloc(&d_B, size);
    CUdeviceptr d_C; cuMemAlloc(&d_C, size);
    // Copy vectors from host memory to device memory
    // h_A and h_B are input vectors stored in host memory
    cuMemcpyHtoD(d_A, h_A, size);
    cuMemcpyHtoD(d_B, h_B, size);
    // Invoke kernel
    #define ALIGN_UP(offset, alignment) \
        (offset) = ((offset) + (alignment) – 1) & ~((alignment) – 1)
    int offset = 0;
    void* ptr; ptr = (void*)(size_t)d_A;
    ALIGN_UP(offset, __alignof(ptr));
    cuParamSetv(vecAdd, offset, &ptr, sizeof(ptr));
    offset += sizeof(ptr);
    ptr = (void*)(size_t)d_B;
    ALIGN_UP(offset, __alignof(ptr));
    cuParamSetv(vecAdd, offset, &ptr, sizeof(ptr));
    offset += sizeof(ptr);
    ptr = (void*)(size_t)d_C;
    ALIGN_UP(offset, __alignof(ptr));
    cuParamSetv(vecAdd, offset, &ptr, sizeof(ptr));
    offset += sizeof(ptr);
    cuParamSetSize(VecAdd, offset);
    int threadsPerBlock = 256;
    int blocksPerGrid = (N + threadsPerBlock – 1) / threadsPerBlock;
    cuFuncSetBlockShape(vecAdd, threadsPerBlock, 1, 1);
    cuLaunchGrid(VecAdd, blocksPerGrid, 1);
    // Copy result from device memory to host memory
    cuMemcpyDtoH(h_C, d_C, size);
    // Free device memory
    cuMemFree(d_A); cuMemFree(d_B); cuMemFree(d_C);
}
```

线性存储器也可使用 cuMemAllocPitch()分配，尤其在分配二维数组时推荐使用，因为它保证分配满足 5.3.2.1 节描述的对齐要求，因此在按行访问或在二维数组和设备存储器的其它区域拷贝（cuMemcpy2D()）时可获得最佳性能。返回的步长必须用于访问数组元素。下面的例子分配了一个尺寸为 width*height 的二维浮点数组并在设备代码中遍历数组元素：

```
// Host code (assuming cuModule has been loaded)
CUdeviceptr devPtr;
int pitch;
cuMemAllocPitch(&devPtr, &pitch, width * sizeof(float), height, 4);
CUfunction myKernel;
cuModuleGetFunction(&myKernel, cuModule, “MyKernel”);
void* ptr = (void*)(size_t)devPtr;
cuParamSetv(myKernel, 0, &ptr, sizeof(ptr));
cuParamSetSize(myKernel, sizeof(ptr));
cuFuncSetBlockShape(myKernel, 512, 1, 1);
cuLaunchGrid(myKernel, 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];
        }
    }
}
```

下

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

参考手册中列出了所有在使用 cuMemAlloc()，cuMemAllocPitch()分配的线性存储器和 CUDA 数组间拷贝的函数。

下面的代码将一个二维的数组复制到上面代码分配的 CUDA 数组

```
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);
```

下面的代码将一些主机存储器数组复制到常量存储器：

```
__constant__ float constData[256];
float data[256];
CUdeviceptr devPtr;
unsigned int bytes;
cuModuleGetGlobal(&devPtr &,bytes,cuModule, “constData”);
cuMemcpyHtoD(devPtr, data, bytes);
```

3.3.5 共享存储器

下面的代码是 3.2.2 节代码的驱动版本。注意 Matrix 类型在主机中的声明必须不同，因为设备指针的是 CUdeviceptr 类型的句柄。

下面的例子，共享存储器是在内核中静态分配的，不同于使用 cuFuncSetSharedSize() 在运行时动态分配。

```
// Matrices are stored in row-major order:
// M(row, col) = *(M.elements + row * M.stride + col)
typedef struct {
    int width; int height; int stride;
    #ifdef __CUDACC__
        float* elements;
    #else
    CUdeviceptr elements;
    #endif
} Matrix;
// Matrix multiplication - Host code
// Matrix dimensions are assumed to be multiples of BLOCK_SIZE
void MatMul(const Matrix A, const Matrix B, Matrix C){
    // Load A and B to device memory
    Matrix d_A;
    d_A.width = d_A.stride = A.width; d_A.height = A.height;
    size_t size = A.width * A.height * sizeof(float);
    cuMemAlloc(&d_A.elements, size);
    cuMemcpyHtoD(d_A.elements, A.elements, size);
    Matrix d_B;
    d_B.width = d_B.stride = B.width; d_B.height = B.height;
    size = B.width * B.height * sizeof(float);
    cuMemAlloc(&d_B.elements, size);
    cuMemcpyHtoD(d_B.elements, B.elements, size);
    // Allocate C in device memory
    Matrix d_C;
    d_C.width = d_C.stride = C.width; d_C.height = C.height;
    size = C.width * C.height * sizeof(float);
    cuMemAlloc(&d_C.elements, size);
    // Invoke kernel (assuming cuModule has been loaded)
    CUfunction matMulKernel;
    cuModuleGetFunction(&matMulKernel, cuModule, "MatMulKernel");
    #define ALIGN_UP(offset, alignment) \
        (offset) = ((offset) + (alignment) – 1) & ~((alignment) – 1)
    int offset = 0;
    void* ptr; ptr = (void*)(size_t)d_A; ALIGN_UP(offset, __alignof(ptr));
    cuParamSetv(matMulKernel, offset, &ptr, sizeof(ptr));
    offset += sizeof(ptr);
    ptr = (void*)(size_t)d_B; ALIGN_UP(offset, __alignof(ptr));
    cuParamSetv(matMulKernel, offset, &ptr, sizeof(ptr));
    offset += sizeof(ptr);
    ptr = (void*)(size_t)d_C; ALIGN_UP(offset, __alignof(ptr));
    cuParamSetv(matMulKernel, offset, &ptr, sizeof(ptr));
    offset += sizeof(ptr);
    cuParamSetSize(matMulKernel, offset);
    cuFuncSetBlockShape(matMulKernel, BLOCK_SIZE, BLOCK_SIZE, 1);
    cuLaunchGrid(matMulKernel, B.width/dimBlock.x, A.height/dimBlock.y);
    // Read C from device memory
    cuMemcpyDtoH(C.elements, d_C.elements, size);
    // Free device memory
    cuMemFree(d_A.elements); cuMemFree(d_B.elements); cuMemFree(d_C.elements);
}
```

3.3.6 多设备

cuDevieGetCount()和 cuDeviceGet()提供了一种枚举系统中设备的方法，其它的一些函数（参见参考手册）可检索设备的属性。

```
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);
}
```

3.3.7 纹理存储器

使用 cuTexRefSetAddress()将线性存储器绑定到纹理，而 cuTexRefSetArray()将 CUDA 数组绑定到纹理。

如果 cuModule 模块包含某个纹理参考 texRef 定义如

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

下面的代码检索 texRef 的句柄：

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

下面的代码将 texRef 绑定到 devPtr 指向的线性存储器：

```
CUDA_ARRAY_DESCRIPTOR desc;
cuTexRefSetAddress2D(cuTexRef, &desc, devPtr, pitch);
```

下面的代码将 texRef 绑定到 CUDA 数组 cuArray：

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

参考手册中列出了多种用于设置纹理参考的寻址模式，滤波模式，格式，和其它的标签的函数。纹理绑定时指定的纹理参考格式必须和声明纹理参考时指定的格式匹配，否则纹理获取结果未定义。

下面的代码是 3.2.4.3 节代码的驱动 API 版本：

```
// Host code
int main() {
    // Allocate CUDA array in device memory
    CUarray cuArray;
    CUDA_ARRAY_DESCRIPTOR desc;
    desc.Format = CU_AD_FORMAT_FLOAT;
    desc.NumChannels = 1; desc.Width = width; desc.Height = height;
    cuArrayCreate(&cuArray, &desc);
    // Copy to device memory some data located at address h_data
    // in host memory
    CUDA_MEMCPY2D copyParam;
    memset(&copyParam, 0, sizeof(copyParam));
    copyParam.dstMemoryType = CU_MEMORYTYPE_ARRAY;
    copyParam.dstArray = cuArray;
    copyParam.srcMemoryType = CU_MEMORYTYPE_HOST;
    copyParam.srcHost = h_data;
    copyParam.srcPitch = width * sizeof(float);
    copyParam.WidthInBytes = copyParam.srcPitch;
    copyParam.Height = height;
    cuMemcpy2D(&copyParam);
    // Set texture parameters
    CUtexref texRef;
    cuModuleGetTexRef(&texRef, cuModule, "texRef"));
    cuTexRefSetAddressMode(texRef, 0, CU_TR_ADDRESS_MODE_WRAP);
    cuTexRefSetAddressMode(texRef, 1, CU_TR_ADDRESS_MODE_WRAP);
    cuTexRefSetFilterMode(texRef, CU_TR_FILTER_MODE_LINEAR);
    cuTexRefSetFlags(texRef, CU_TRSF_NORMALIZED_COORDINATES);
    cuTexRefSetFormat(texRef, CU_AD_FORMAT_FLOAT, 1);
    // Bind the array to the texture
    cuTexRefSetArray(texRef, cuArray, CU_TRSA_OVERRIDE_FORMAT);
    // Allocate result of transformation in device memory
    CUdeviceptr output;
    cuMemAlloc((void**)&output, width * height * sizeof(float));
    // Invoke kernel (assuming cuModule has been loaded)
    CUfunction transformKernel;
    cuModuleGetFunction(&transformKernel,
    cuModule, "transformKernel");
    #define ALIGN_UP(offset, alignment) \
        (offset) = ((offset) + (alignment) – 1) & ~((alignment) – 1)
    int offset = 0;
    void* ptr = (void*)(size_t)output;
    ALIGN_UP(offset, __alignof(ptr));
    cuParamSetv(transformKernel, offset, &ptr, sizeof(ptr));
    offset += sizeof(ptr);      ALIGN_UP(offset, __alignof(width));
    cuParamSeti(transformKernel, offset, width);
    offset += sizeof(width);        ALIGN_UP(offset, __alignof(height));
    cuParamSeti(transformKernel, offset, height);
    offset += sizeof(height);       ALIGN_UP(offset, __alignof(angle));
    cuParamSetf(transformKernel, offset, angle);
    offset += sizeof(angle);
    cuParamSetSize(transformKernel, offset));
    cuParamSetTexRef(transformKernel, CU_PARAM_TR_DEFAULT, texRef);
    cuFuncSetBlockShape(transformKernel, 16, 16, 1);
    cuLaunchGrid(transformKernel, (width + dimBlock.x – 1) / dimBlock.x,
        (height + dimBlock.y – 1) / dimBlock.y);
    // Free device memory
    cuArrayDestroy(cuArray); cuMemFree(output);
}
```

3.3.8 分页锁定主机存储器

可以使用 cuMemHostAlloc()分配分页锁定主机存储器，此时传入的标签不能相互冲突：

- CU_MEMHOSTALLOC_PORTABLE 分配在 CUDA 上下文间可分享的存储器（见 3.2.5.1 和 3.2.5.2）；
- CUD_MEMHOSTALLOC_WRITECOMBINED 分配写结合主机存储器（参见 3.2.5.2 节）；
- CU_MEMHOSTALLOC_DEVICEMAP 分配被映射分页锁定主机存储器（参见 3.2.5.3 节）；

使用 cuMemFreeHost()释放分页锁定主机存储器。

使用 CU_CTX_MAP_HOST 标签创建上下文才支持使用被映射分页锁定存储器，可全长 cuMemHostGetDevicePointer()取得被映射内存的设备指针。

应用可以使用 cuDeviceGetAttribute()函数查询 CU_DEVICE_ATTRIBUTE_CAN_MAP_HOST_MEMORY 属性可以确定设备是否支持被映射主机分页锁定内存。

3.3.9 异步并发执行

应用可以使用 cuDeviceGetAttribute()函数查询 CU_DEVICE_ATTRIUTE_GPU_OVERLAP 属性确定设备是否支持分页锁定主机存储器和设备存储器间数据拷贝与内核执行重叠。

应用可以通过 cuDeviceGetAttribute()函数查询 CU_DEVICE_ATTRIBUTE_CONCURRENT_KERNELS 属性确定设备是否支持多内核并发执行。

3.3.9.1 流

驱动 API 像运行时 API 一样提供了管理流的函数。下面的代码是 3.2.6.4 节代码的驱动版本。

```
CUstream stream[2];
for (int i = 0; i < 2; ++i)
    cuStreamCreate(&stream[i], 0);
float* hostPtr;
cuMemAllocHost((void**)&hostPtr, 2 * size);
for (int i = 0; i < 2; ++i)
    cuMemcpyHtoDAsync(inputDevPtr+i*size,hostPtr+i*size,size,stream[i]);
for (int i = 0; i < 2; ++i) {
    #define ALIGN_UP(offset, alignment) \
        (offset) = ((offset) + (alignment) - 1) & ~((alignment) - 1)
    int offset = 0;
void* ptr;
ptr = (void*)(size_t)outputDevPtr;
ALIGN_UP(offset, __alignof(ptr));
cuParamSetv(cuFunction, offset, &ptr, sizeof(ptr));
offset += sizeof(ptr);
ptr = (void*)(size_t)inputDevPtr;
ALIGN_UP(offset, __alignof(ptr));
cuParamSetv(cuFunction, offset, &ptr, sizeof(ptr));
offset += sizeof(ptr); ALIGN_UP(offset, __alignof(size));
cuParamSeti(cuFunction, offset, size);
offset += sizeof(int);
cuParamSetSize(cuFunction, offset);
cuFuncSetBlockShape(cuFunction, 512, 1, 1);
cuLaunchGridAsync(cuFunction, 100, 1, stream[i]);
}
for (int i = 0; i < 2; ++i)
    cuMemcpyDtoHAsync(hostPtr + i * size, outputDevPtr + i * size, size, stream[i]);
cuCtxSynchronize();
for (int i = 0; i < 2; ++i)
    cuStreamDestroy(&stream[i]);
```

3.3.9.2 事件管理

驱动 API 像运行时 API 一样提供了管理事件的函数。下面的代码是 3.2.6.5 节代码的驱动版本。

```
CUevent start, stop;
cuEventCreate(&start);
cuEventCreate(&stop);
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) {
    #define ALIGN_UP(offset, alignment) \
        (offset) = ((offset) + (alignment) – 1) & ~((alignment) – 1)
    int offset = 0;
    void* ptr;
    ptr = (void*)(size_t)outputDevPtr;
    ALIGN_UP(offset, __alignof(ptr));
    cuParamSetv(cuFunction, offset, &ptr, sizeof(ptr));
    offset += sizeof(ptr);
    ptr = (void*)(size_t)inputDevPtr;
    ALIGN_UP(offset, __alignof(ptr));
    cuParamSetv(cuFunction, offset, &ptr, sizeof(ptr));
    offset += sizeof(ptr);
    ALIGN_UP(offset, __alignof(size));
    cuParamSeti(cuFunction, offset, size);
    offset += sizeof(size);
    cuParamSetSize(cuFunction, offset);
    cuFuncSetBlockShape(cuFunction, 512, 1, 1);
    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);
```

3.3.9.3 同步调用

在一同步函数调用时，主机线程是让步、阻塞还是自旋，可通过调用 cuCtxCreate()并指定标签（参见手册）确定。

3.3.10 图形学互操作性

驱动 API 像运行时 API 一样提供了函数以管理图形学互操作性。

资源在使用 3.3.10.1 节和 3.3.10.2 节的函数映射之前，必须注册到 CUDA。这些函数返回一个 CUgraphicsResource 类型的 CUDA 图形资源。注册资源是高消耗的，因此典型地每种

资源只注册一次。使用 cuGraphicsunregisterResource()解注册 CUDA 图形资源。

一旦资源注册到 CUDA，只要需要可无数次使用 cuGraphicsMapResources()和 cuGraphicsUnmapResource()映射和解映射。可以使用 cuGraphicsResourceSetMapFlags()指定用处（只读，只写），以便 CUDA 驱动优化资源管理。

cuGraphicsResourceGetMappedPointer()为缓冲区返回设备存储器地址，cuGraphicsSubResourceGetMappedArray()为 CUDA 数组返回设备存储器地址，内核可以使用这些地址读写被映射资源。

通过 OpenGL 或 Direct3D 访问被映射到 CUDA 的资源，其结果未定义。

3.3.10.1 节和 3.3.10.2 节给出了每种图形 API 的特性和一些例子。

3.3.10.1 OpenGL 互操作性

OpenGL 互操作性要求使用 cuGLCtxCreate()而不是 cuCtxCreate()创建 CUDA 上下文。

可以被映射到 CUDA 地址空间的 OpenGL 资源有 OpenGL 缓冲区，纹理，和沉浸缓冲对象。使用 cugraphicsGLRegisterBuffer()注册缓冲区对象。使用 cuGraphicsGLRegisterimage()注册纹理或沉浸缓冲区对象。限制和 3.2.7.1 节一致。

下面代码是 3.2.7.1 节代码的驱动版本。

```
CUfunction createVertices;
GLuint positionsVBO;
struct cudaGraphicsResource* positionsVBO_CUDA;
int main() {
    // Initialize driver API
    ...
    // Get handle for device 0
    CUdevice cuDevice = 0;
    cuDeviceGet(&cuDevice, 0);
    // Create context
    CUcontext cuContext;
    cuGLCtxCreate(&cuContext, 0, cuDevice);
    // Create module from binary file
    CUmodule cuModule;
    cuModuleLoad(&cuModule, “createVertices.ptx”);
    // Get function handle from module
    cuModuleGetFunction(&createVertices,
    cuModule, "createVertices");
    // Initialize OpenGL and GLUT
    ...
    glutDisplayFunc(display);
    // Create buffer object and register it with CUDA
    glGenBuffers(1, positionsVBO);
    glBindBuffer(GL_ARRAY_BUFFER, &vbo);
    unsigned int size = width * height * 4 * sizeof(float);
    glBufferData(GL_ARRAY_BUFFER, size, 0, GL_DYNAMIC_DRAW);
    glBindBuffer(GL_ARRAY_BUFFER, 0);
    cuGraphicsGLRegisterBuffer(&positionsVBO_CUDA,
        positionsVBO, cudaGraphicsMapFlagsWriteDiscard);
    // Launch rendering loop
    glutMainLoop();
}
```

```
void display() {
    // Map OpenGL buffer object for writing from CUDA
    CUdeviceptr positions;
    cuGraphicsMapResources(1, &positionsVBO_CUDA, 0);
    size_t num_bytes;
    cuGraphicsResourceGetMappedPointer((void**)&positions,
          &num_bytes, positionsVBO_CUDA));
    // Execute kernel
    #define ALIGN_UP(offset, alignment) \
        (offset) = ((offset) + (alignment) – 1) & ~((alignment) – 1)
    int offset = 0;
    void* ptr = (void*)(size_t)positions;
    ALIGN_UP(offset, __alignof(ptr));
    cuParamSetv(createVertices, offset, &ptr, sizeof(ptr));
    offset += sizeof(ptr);
    ALIGN_UP(offset, __alignof(time));
    cuParamSetf(createVertices, offset, time);
    offset += sizeof(time);
    ALIGN_UP(offset, __alignof(width));
    cuParamSeti(createVertices, offset, width);
    offset += sizeof(width);
    ALIGN_UP(offset, __alignof(height));
    cuParamSeti(createVertices, offset, height);
    offset += sizeof(height);
    cuParamSetSize(createVertices, offset);
    int threadsPerBlock = 16;
    cuFuncSetBlockShape(createVertices, threadsPerBlock, threadsPerBlock, 1);
    cuLaunchGrid(createVertices, width / threadsPerBlock, height / threadsPerBlock);
    // Unmap buffer object
    cuGraphicsUnmapResources(1, &positionsVBO_CUDA, 0);
    // Render from buffer object
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glBindBuffer(GL_ARRAY_BUFFER, positionsVBO);
    glVertexPointer(4, GL_FLOAT, 0, 0);
    glEnableClientState(GL_VERTEX_ARRAY);
    glDrawArrays(GL_POINTS, 0, width * height);
    glDisableClientState(GL_VERTEX_ARRAY);
    // Swap buffers
    glutSwapBuffers();
    glutPostRedisplay();
}
void deleteVBO(){
    cuGraphicsUnregisterResource(positionsVBO_CUDA);
    glDeleteBuffers(1, &positionsVBO);
}
```

在 Windows 系统上和对于 Quadro 显卡，cuWGLGetDevice()可以检索由 wglEnumGpusNV()返回的句柄关联的 CUDA 设备。

3.3.10.2 Direct3D 互操作性

和 Direct3D 的互操作性要求在建立 CUDA 上下文时指定 Direct3D 设备，可以使用 cuD3D9CtxCreate()（resp. cuD3D10CtxCreate()）而非 cuCtxCreate()做到。

能够映射到 CUDA 地址空间的 Direct3D 资源包括 Direct3D 缓冲区、纹理和表面。使用 cuGraphicsd3d9RegisterResource()、cuGraphicsd3d19RegisterResource()和 cuGraphicsD3D11RegisterResource()注册这些资源。

下面的例子是 3.2.7.2 节代码的驱动版本。

Direct3D 9 版本：

```
IDirect3D9* D3D;
IDirect3DDevice9* device;
struct CUSTOMVERTEX {
    FLOAT x, y, z;
    DWORD color;
};
IDirect3DVertexBuffer9* positionsVB;
struct cudaGraphicsResource* positionsVB_CUDA;

int main(){
    // Initialize Direct3D
    D3D = Direct3DCreate9(D3D_SDK_VERSION);
    // Get a CUDA-enabled adapter
    unsigned int adapter = 0;
    for (; adapter < g_pD3D->GetAdapterCount(); adapter++) {
        D3DADAPTER_IDENTIFIER9 adapterId;
        g_pD3D->GetAdapterIdentifier(adapter, 0, &adapterId);
        int dev;
        if (cuD3D9GetDevice(&dev, adapterId.DeviceName)== cudaSuccess)
            break;
    }
    // Create device
    ...
    D3D->CreateDevice(adapter, D3DDEVTYPE_HAL, hWnd,
        D3DCREATE_HARDWARE_VERTEXPROCESSING, &params, &device);
    // Initialize driver API
    ...
    // Create context
    CUdevice cuDevice;
    CUcontext cuContext;
    cuD3D9CtxCreate(&cuContext, &cuDevice, 0, &device);
    // Create module from binary file
    CUmodule cuModule;
    cuModuleLoad(&cuModule, “createVertices.ptx”);
    // Get function handle from module
    cuModuleGetFunction(&createVertices, cuModule, "createVertices");
    // Create vertex buffer and register it with CUDA
    unsigned int size = width * height * sizeof(CUSTOMVERTEX);
    device->CreateVertexBuffer(size, 0, D3DFVF_CUSTOMVERTEX,
        D3DPOOL_DEFAULT, &positionsVB, 0);
    cuGraphicsD3D9RegisterResource(&positionsVB_CUDA,positionsVB,
        cudaGraphicsRegisterFlagsNone);
    cuGraphicsResourceSetMapFlags(positionsVB_CUDA,cudaGraphicsMapFlagsWriteDiscard);
    // Launch rendering loop
    while (...) {
        ...
        Render();
        ...
    }
}
```

```
void Render() {
    // Map vertex buffer for writing from CUDA
    float4* positions;
    cuGraphicsMapResources(1, &positionsVB_CUDA, 0);
    size_t num_bytes;
    cuGraphicsResourceGetMappedPointer((void**)&positions, &num_bytes,
    positionsVB_CUDA));
    // Execute kernel
    #define ALIGN_UP(offset, alignment) \
        (offset) = ((offset) + (alignment) – 1) & ~((alignment) – 1)
    int offset = 0;
    void* ptr = (void*)(size_t)positions;
    ALIGN_UP(offset, __alignof(ptr));
    cuParamSetv(createVertices, offset, &ptr, sizeof(ptr));
    offset += sizeof(ptr);
    ALIGN_UP(offset, __alignof(time));
    cuParamSetf(createVertices, offset, time);
    offset += sizeof(time);
    ALIGN_UP(offset, __alignof(width));
    cuParamSeti(createVertices, offset, width);
    offset += sizeof(width);
    ALIGN_UP(offset, __alignof(height));
    cuParamSeti(createVertices, offset, height);
    offset += sizeof(height);
    cuParamSetSize(createVertices, offset);
    int threadsPerBlock = 16;
    cuFuncSetBlockShape(createVertices, threadsPerBlock, threadsPerBlock, 1);
    cuLaunchGrid(createVertices, width / threadsPerBlock, height / threadsPerBlock);
    // Unmap vertex buffer
    cuGraphicsUnmapResources(1, &positionsVB_CUDA, 0);
    // Draw and present
    ...
}
void releaseVB(){
    cuGraphicsUnregisterResource(positionsVB_CUDA);
    positionsVB->Release();
}
```

Direct3D 10 版本：

```
ID3D10Device* device;
struct CUSTOMVERTEX {
    FLOAT x, y, z;
    DWORD color;
};
ID3D10Buffer* positionsVB;
struct cudaGraphicsResource* positionsVB_CUDA;
int main() {
    // Get a CUDA-enabled adapter
    IDXGIFactory* factory;
    CreateDXGIFactory(__uuidof(IDXGIFactory), (void**)&factory);
    IDXGIAdapter* adapter = 0;
    for (unsigned int i = 0; !adapter; ++i) {
        if (FAILED(factory->EnumAdapters(i, &adapter))
            break;
        int dev;
        if (cuD3D10GetDevice(&dev, adapter) == cudaSuccess)
            break;
        adapter->Release();
    }
    factory->Release();
    // Create swap chain and device
    ...
    D3D10CreateDeviceAndSwapChain(adapter, D3D10_DRIVER_TYPE_HARDWARE, 0,
    D3D10_CREATE_DEVICE_DEBUG, D3D10_SDK_VERSION, &swapChainDesc &swapChain,
    &device);
    adapter->Release();
    // Initialize driver API
    ...
    // Create context
    CUdevice cuDevice;
    CUcontext cuContext;
    cuD3D10CtxCreate(&cuContext, &cuDevice, 0, &device);
    // Create module from binary file
    CUmodule cuModule;
    cuModuleLoad(&cuModule, “createVertices.ptx”);
    // Get function handle from module
    cuModuleGetFunction(&createVertices, cuModule, "createVertices");
    // Create vertex buffer and register it with CUDA
    unsigned int size = width * height * sizeof(CUSTOMVERTEX);
    D3D10_BUFFER_DESC bufferDesc;
    bufferDesc.Usage = D3D10_USAGE_DEFAULT;
    bufferDesc.ByteWidth = size;
    bufferDesc.BindFlags = D3D10_BIND_VERTEX_BUFFER;
    bufferDesc.CPUAccessFlags = 0;
    bufferDesc.MiscFlags = 0;
    device->CreateBuffer(&bufferDesc, 0, &positionsVB);
    cuGraphicsD3D10RegisterResource(&positionsVB_CUDA, positionsVB,
    cudaGraphicsRegisterFlagsNone);
    cuGraphicsResourceSetMapFlags(positionsVB_CUDA, cudaGraphicsMapFlagsWriteDiscard);
    // Launch rendering loop
    while (...) {
        ...
        Render();
          ...
    }
}
```

```
void Render() {
    // Map vertex buffer for writing from CUDA
    float4* positions;
    cuGraphicsMapResources(1, &positionsVB_CUDA, 0);
    size_t num_bytes;
    cuGraphicsResourceGetMappedPointer((void**)&positions, &num_bytes,
    positionsVB_CUDA));
    // Execute kernel
    #define ALIGN_UP(offset, alignment) \
    (offset) = ((offset) + (alignment) – 1) & ~((alignment) – 1)
    int offset = 0;
    void* ptr = (void*)(size_t)positions;
    ALIGN_UP(offset, __alignof(ptr));
    cuParamSetv(createVertices, offset, &ptr, sizeof(ptr));
    offset += sizeof(ptr);
    ALIGN_UP(offset, __alignof(time));
    cuParamSetf(createVertices, offset, time);
    offset += sizeof(time);
    ALIGN_UP(offset, __alignof(width));
    cuParamSeti(createVertices, offset, width);
    offset += sizeof(width);
    ALIGN_UP(offset, __alignof(height));
    cuParamSeti(createVertices, offset, height);
    offset += sizeof(height);
    cuParamSetSize(createVertices, offset);
    int threadsPerBlock = 16;
    cuFuncSetBlockShape(createVertices, threadsPerBlock, threadsPerBlock, 1);
    cuLaunchGrid(createVertices, width / threadsPerBlock, height / threadsPerBlock);
    // Unmap vertex buffer
    cuGraphicsUnmapResources(1, &positionsVB_CUDA, 0);
    // Draw and present
    ...
}
void releaseVB(){
    cuGraphicsUnregisterResource(positionsVB_CUDA);
    positionsVB->Release();
}
```

Direct3D 11 版本：

```
ID3D11Device* device;
struct CUSTOMVERTEX {
    FLOAT x, y, z;
    DWORD color;
};
ID3D11Buffer* positionsVB;
struct cudaGraphicsResource* positionsVB_CUDA;
int main() {
    // Get a CUDA-enabled adapter
    IDXGIFactory* factory;
    CreateDXGIFactory(__uuidof(IDXGIFactory), (void**)&factory);
    IDXGIAdapter* adapter = 0;
    for (unsigned int i = 0; !adapter; ++i) {
        if (FAILED(factory->EnumAdapters(i, &adapter))
            break;
        int dev;
        if (cuD3D11GetDevice(&dev, adapter) == cudaSuccess)
            break;
        adapter->Release();
    }
    factory->Release();
    // Create swap chain and device
    ...
    sFnPtr_D3D11CreateDeviceAndSwapChain(adapter,
    D3D11_DRIVER_TYPE_HARDWARE, 0, D3D11_CREATE_DEVICE_DEBUG,
    featureLevels, 3, D3D11_SDK_VERSION, &swapChainDesc, &swapChain, &device,
    &featureLevel, &deviceContext);
    adapter->Release();
    // Initialize driver API
    ...
    // Create context
    CUdevice cuDevice;
    CUcontext cuContext;
    cuD3D11CtxCreate(&cuContext, &cuDevice, 0, &device);
    // Create module from binary file
    CUmodule cuModule;
    cuModuleLoad(&cuModule, “createVertices.ptx”);
    // Get function handle from module
    cuModuleGetFunction(&createVertices, cuModule, "createVertices");
    // Create vertex buffer and register it with CUDA
    unsigned int size = width * height * sizeof(CUSTOMVERTEX);
    D3D11_BUFFER_DESC bufferDesc;
    bufferDesc.Usage = D3D11_USAGE_DEFAULT;
    bufferDesc.ByteWidth = size;
    bufferDesc.BindFlags = D3D10_BIND_VERTEX_BUFFER;
    bufferDesc.CPUAccessFlags = 0;
    bufferDesc.MiscFlags = 0;
    device->CreateBuffer(&bufferDesc, 0, &positionsVB);
    cuGraphicsD3D11RegisterResource(&positionsVB_CUDA, positionsVB,
    cudaGraphicsRegisterFlagsNone);
    cuGraphicsResourceSetMapFlags(positionsVB_CUDA,
    cudaGraphicsMapFlagsWriteDiscard);
    // Launch rendering loop
    while (...){
        ...
        Render();
        ...
    }
}
```

```
void Render() {
    // Map vertex buffer for writing from CUDA
    float4* positions;
    cuGraphicsMapResources(1, &positionsVB_CUDA, 0);
    size_t num_bytes;
    cuGraphicsResourceGetMappedPointer((void**)&positions, &num_bytes,
    positionsVB_CUDA));
     // Execute kernel
    #define ALIGN_UP(offset, alignment) \
    (offset) = ((offset) + (alignment) – 1) & ~((alignment) – 1)
    int offset = 0;
    void* ptr = (void*)(size_t)positions;
    ALIGN_UP(offset, __alignof(ptr));
    cuParamSetv(createVertices, offset, &ptr, sizeof(ptr));
    offset += sizeof(ptr);
    ALIGN_UP(offset, __alignof(time));
    cuParamSetf(createVertices, offset, time);
    offset += sizeof(time);
    ALIGN_UP(offset, __alignof(width));
    cuParamSeti(createVertices, offset, width);
    offset += sizeof(width);
    ALIGN_UP(offset, __alignof(height));
    cuParamSeti(createVertices, offset, height);
    offset += sizeof(height);
    cuParamSetSize(createVertices, offset);
    int threadsPerBlock = 16;
    cuFuncSetBlockShape(createVertices, threadsPerBlock, threadsPerBlock, 1);
    cuLaunchGrid(createVertices, width / threadsPerBlock, height / threadsPerBlock);
    // Unmap vertex buffer
    cuGraphicsUnmapResources(1, &positionsVB_CUDA, 0);
    // Draw and present
    ...
}
void releaseVB(){
    cuGraphicsUnregisterResource(positionsVB_CUDA);
    positionsVB->Release();
}
```

3.3.11 错误处理

所有的驱动函数返回错误码，但是对于异步函数（参见 3.2.6 节），错误码并不能报告设备上发生的任何异步错误，因为函数会在设备完成任务前返回；错误码只能报告发生在任务调用前的错误，典型地有关参数正确性；如果异步错误出现，他将会被后面无关的函数调用报告。

唯一能够检查异步错误的方式是在异步函数调用后调用 cuCtxSynchronize()（或 3.3.9 节描述的其它同步机制），然后检查 cuCtxSynchronize()返回的错误码。

3.4 运行时 API 和驱动 API 的互操作性

在满足本节的限制的前提下，应用可以混合使用运行时 API 和驱动 API。

如果上下文是使用驱动 API 创建并成当前上下文的，随后的运行时调用将使用这个上下文而不是新建一个。

如果运行时已经隐式初始化（如 3.2 节提到的），可以使用 cuCtxAttach()检索初始化间创建的上下文，在随后的驱动 API 调用中可使用它。

设备存储器可使用任何一种 API 分配和释放。CUdeviceptr 也可以转型为规则的指针，反之亦然。

```
CUdeviceptr devPtr;
float* d_data;
// Allocation using driver API
cuMemAlloc(&devPtr, size);
d_data = (float*)devPtr;
// Allocation using runtime API
cudaMalloc((void**)&d_data, size);
devPtr = (CUdeviceptr)(size_t)d_data;
```

特别地，这意味着使用驱动 API 编写的应用能够调用运行时 API 编写的库（如 CUFFT，CUBLAS，....）。

手册中设备和版本管理节的函数可互换使用。

下面的特性不支持在运行时 API 和驱动 API 中混合使用。

- 设备模拟，
- 通过 cuCtxPopCurrent()和 cuCtxPushCurrent()管理上下文栈（参见 3.3.1 节）。

3.5 版本和互操作性

在开发 CUDA 应用时，开发人员应该关注两种版本号：计算能力描述了基本规范和计算设备特性（见 2.5 节），CUDA 驱动 API 版本描述了驱动 API 和运行时 API 支持的特性。

驱动 API 的版本在驱动头文件中定义为 CUDA_VERSION。开发人员可用其检查其应用是否要比当前版本更新的驱动。这非常重要，因为驱动 API 是向后兼容的，这意味着特定版本编译的应用、插件和库（包括 C 运行时）能够在以后发布的驱动上工作，如图 3-4 所示。但是驱动 API 不是向前兼容的，这意味着特定版本编译的应用、插件和库（包括 C 运行时）不能够在以前发布的驱动上工作。

要注意混合版本是不支持的；尤其：

- 一个系统上所有应用、插件和库必须使用同一版本的 CUDA 驱动 API，因为一个系统上只能安装一种版本的 CUDA 驱动。
- 应用使用的所有插件和库必须使用同一版本的运行时。
- 应用使用的所有插件和库必须同一版本的任何使用了运行时的库（如 CUFFT，CUBLAS）。

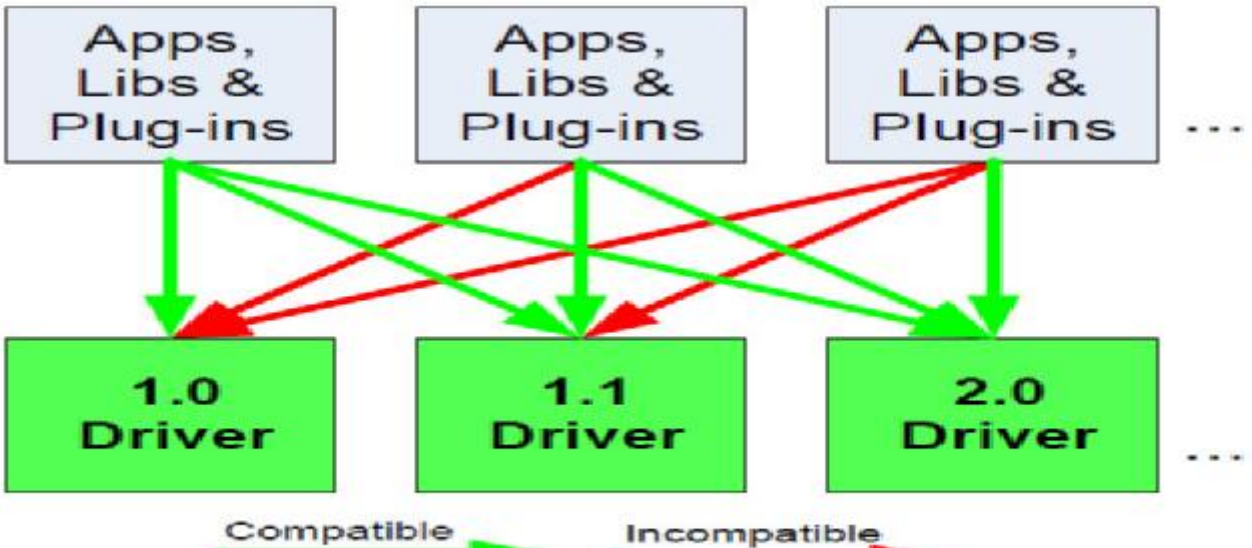


图3-4. 驱动API向后兼容，不向前兼容

3.6 计算模式

在 Linux 上运行的 Tesla 解决方案，可以使用 NVIDIA 的系统管理接口（nvidia-smi）设置系统上任何设备的计算模式的为下面的三种之一，nvidia-smi 是一个作为 Linux 驱动一部分发布的工具。

- 默认模式：多个主机线程可同时使用设备（使用运行时调用 cudaSetDevice()，或使用驱动 API 时将关联到设备的上下文作为当前上下文）。
- 排它模式：在任何时候，只能有一个主机线程使用设备。
- 禁止模式：不允许任何主机线程使用设备。

特别地，这意味着使用运行时且没有显式调用 cudaSetDevice()的主机线程可能不被关联到 0 号设备，如果 0 号设备刚好工作在禁止模式或在排它模式下工作但被其它设备使用。cudaSetValidDevice()可基于设备优先级列表设置一个设备。

应用可调用 cudaGetDeviceProperties()并查询 computeMode 属性或调用 cuDeviceGetAttribute()查询 CU_DEVICE_COMPUTE_MODE 属性以查询设备的计算模式。

3.7 模式切换

GPU 将部分 DRAM 存储器用于所谓的主表面（primary surface），它用于刷新显示器，用户查看显示器的输出。当用户通过更改显示器的分辨率或位深度（使用 NVIDIA 的控制面板或 Windows 的显示控制面板）初始化模式切换时，主表面所需的存储器数量随之改变。例如，如果用户将显示器的分辨率从 1280*1024*32 位更改为 1600*1200*32 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可能需要为主表面分配更多存储器用于显示）。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 应用的存储器，因此模式切换可能导致任何调用 CUDA 运行时的应用崩溃并返回无效上下文错误。

第四章 硬件实现

CUDA 架构是围绕一个可扩展的多线程流多处理器（SMs）阵列构建的。当主机上的 CUDA 程序调用内核网格，网格内块枚举并分发到有可用执行资源的多处理器上。线程块内线程在一个多处理器上并发执行且多个块可在一个流多处理器上并发执行。线程块终止时，便在空闲多处理器上发射新块。

流多处理器设计为能同时并发执行上百线程。为了管理如此多的线程，多处理器采用了一种称为 SIMT（单指令，多线程）的独一无二的架构，这在 4.1 节描述。为了最大化利用功能单元，流多处理器使用如 4.2 节所描述的硬件多线程利用线程级并行，超越单线程内的指令级并行（指令通过管线输送，但不像 CPU 核心一样顺序执行并没有分支预测和猜测执行）。

4.1 节和 4.2 节描述了所有设备都相同的流多处理器架构特色。G.3.1 节和 G.4.1 节分别提供了计算能力 1.1 和 2.0 的特性。

4.1 SIMT 架构

多处理器以 32 个为一组创建、管理、调度和执行并行线程，这 32 个线程组称为束（warps）。束内包含的不同线程从同一程序地址开始，但它们有自己的指令地址计数器和寄存器状态，因此可自由分支和独立执行。束这个术语来源于纺织（weaving）这第一种并行线程技术。半束（half-warp）是束的前一半或后一半。四分之一束是指第一、第二、第三或第四个，束的四分之一。

当多处理器得到一个或多个块执行，它会将块分割成束以执行，束被束调度器调度。块分割成束的方式总是相同的；束内线程是连续的，递增线程 ID，第一个束包含线程 0。2.2 节给出了块内线程 ID 和线程索引的关系。

束每次执行一个相同的指令，所以如果束内所有 32 个线程在同一条路径上执行的话，会达到最高效率。如果由于数据依赖条件分支导致束分岔，束会顺序执行每个分支路径，而禁用不在此路径上的线程，直到所有路径完成，线程重新汇合到同一执行路径。分支岔开只会在同一束内发生；不同的束独立执行不管它们是执行相同或不同的代码路径。

在使用单指令控制多处理元素这点上，SIMT 架构类似 SIMD（单指令，多数据）向量组织方法。重要的不同在于 SIMD 组织方法会向应用暴露 SIMD 宽度，而 SIMT 指定单线程的执行和分支行为。与 SIMD 向量机相反，SIMT 允许程序员为独立标量线程编写线程级并行代码，也为协调线程编写数据并行代码。为了正确性，程序员可忽略 SIMT 行为；然而只要维护束内线程很少分支的代码就可显著提升性能。实践中，这类似于传统代码中缓存线的角色：以正确性为目标进行设计时，可忽略缓存线尺寸，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。另外，向量架构要求软件将负载合并成向量，并手动管理分支。

如果一个束执行非原子指令为多个线程写入全局存储器或共享存储器的同一位置，串行写入位置变量的数目依赖于设备的计算能力（参见 G.3.2 节、G.3.3 节和 G.4.3 节）且那个线程最后写入无法确定。

如果束执行的原子指令（参见 B.10 节）为束内多个线程读、修改和写入全局存储器的同一位置，每次读、修改和写入都会串行执行，但是他们执行的顺序没有定义。

4.2 硬件多线程

流多处理器处理的每个束的执行上下文（程序计数器，寄存器等）在束的生存期内被维护在片上。从一个执行上下文切换到另一个执行上下文没有消耗，而且在每个指令发射间，

束调度器选择所有线程已准备好执行的束（活动束）并且向这些线程发射下个指令。

特别地，每个多处理器有一组 32 位的寄存器，这些寄存器被束分割而并行数据缓存或共享存储器在块内分割。

多处理器能够为一个内核常驻和同时处理的块和束的数量依赖于内核使用的寄存器和共享存储器数量和多处理器拥有的寄存器和共享存储器总量。同时每个多处理器有一个最大的常驻块数量和常驻束数量。这些限制包括多处理器上可用寄存器和共享存储器的数量是设备计算能力的函数，且其值在附录 G 中给出。如果多处理器没有足够的可用寄存器或共享存储器以处理至少一个块，内核将会发射失败。

块内总束数量 W_{block} 如下：

$$W_{block}=ceil(T/W_{size},1)$$

- T 是块内线程数，
- W_{size} 是束尺寸，等于 32，
- ceil(x,y)等于 x 向上取到 y 的整数倍

分配给一个块的总寄存器数量 R_{block} 如下：

对于计算能力 1.x 的设备：

$$R_{block}=ceil(ceil(W_{block},G_W)*W_{size}*R_k,G_T)$$

对于计算能力 2.0 的设备：

$$R_{block}=ceil(R_k*W_{size},G_T)*W_{block}$$

- G_W 是束分配粒度，等于 2（只是计算能力 1.x），
- R_k 是内核使用的寄存器数，
- G_T 是线程分配的粒度，计算能力 1.0 和 1.1 的设备上等于 256，计算能力 1.2 和 1.3 的设备上等于 512，计算能力 2.0 的设备上等于 64.

块内总共享存储器数量 S_{block} 如下：

$$S_{block}=ceil(S_k,G_S)$$

- S_k 是内核使用的共享存储器总量，以字节为单位，
- G_S 是共享存储器分配粒度，对于计算能力 1.x 的设备，其值为 512，而对于计算能力 2.0 的设备其值为 128

4.3 多设备

在多 GPU 的系统上，所有支持 CUDA 的设备可通过 CUDA 驱动 API 和运行时 API 像独立设备一样访问。但是如果系统在 SLI 模式下，要作下面的考虑。

首先，在一个 GPU 上的一个 CUDA 设备的存储器分配也会消耗其它 GPU 的存储器，由于这一点，分配会比预想的早点失败。

其次，当一个 Direct3D 应用运行在 SLI 交替帧渲染模式，应用创建的 Direct3D 设备可用于 CUDA-Direct3D 互操作性（如，在使用运行时 API 时，传一个参数到 cudaD3D[9|10]SetDirect3DDevice()），但是从这些 Direct3D 设备中的一个一次只能创建一个 CUDA 设备。这个 CUDA 设备只能在一个 SLI 配置下的 GPU 上执行 CUDA 任务。结果，真正的互操作性只在 GPU 内 Direct3D 资源间（注意：在交替帧渲染模式下，Direct3D 资源必须从一个 GPU 存储器复制到其它在 SLI 配置下的 GPU 存储器）拷贝发生。在某些情况下这并

不是要求的行为且应用可能要禁止使用 CUDA-Direct3D 互操作性 API 且使用现在的 CUDA 和 Direct3DAPI 手动拷贝 CUDA 工作的输出到 Direct3D 资源。

第五章 性能优化指南

5.1 总体性能优化策略

性能优化始终围绕三个基本策略：

- 最大化并行执行以获得最大利用率；
- 优化存储器使用以获得最大存储器吞量；
- 优化指令使用获得最大指令吞吐量。

对于应用的某个特定部分，什么策略会产生最好的性能收益依赖于这些部分的性能限制；如，优化存储器访问限制的内核的指令使用不会产生明显的效果。优化努力应当不变地被测试到和监视到的性能限制所定向，如使用 CUDA profiler。另外比较某个特定内核浮点操作吞吐量或存储器吞吐量（更有意义）与对应的理论峰值吞吐量的差异可指出性能提升空间。

5.2 最大化利用率

尽量展现并行性并且有效的将并行性映射到系统的各个组件上以保证它们大部分时间都在工作，为了最大利用率应用应当以这为准则构建。

5.2.1 应用层次

在高层次上，应用应当通过使用异步函数调用和 3.2.6 节描述的流来最大化主机，设备和连接主机和设备的总线的并行执行。应当把每个处理器最擅长的任务分配给它：串行工作分配给主机；并行工作分配给设备。

对于并行工作，在算法中，由于某些线程为了与其它线程共享数据而同步导致并行性中断的点，有两种情况：或那些线程属于同一个块，这种情况下，它们只要使用 __syncthreads()和在同一个内核调用中使用共享存储器共享数据，或属于不同的块，这种情况下，它们必须使用两个不同的内核调用以通过全局存储器共享数据，一个内核将数据写入全局存储器，另一个从全局存储器中读。第二种情况优化得比较差，因为它增加了额外的内核调用消耗和全局存储器通信量。为了减弱这种影响，应当以一种将线程间通信局限在一个块的方式将算法映射到 CUDA 编程模型。

5.2.2 设备层次

在低层次，应用应当最大化设备内多处理器间的并行执行。

对于计算能力 1.x 的设备，一次只允许一个在内核在设备上执行，所以内核至少要发射的块数要和多处理器数一样多。

对于计算能力 2.0 的设备，多个内核可在设备上并发执行，因此可以使用流来发射多个内核并发执行（参见 3.2.6 节）来获得最大利用率。

5.2.3 多处理器层次

在一个更低的层次，应用应当最大化多处理器内部的各种功能单元的并行执行。

如 4.2 节所描述的，GPU 多处理器依赖线程级并行来最大化其功能单元的利用。利用率

和常驻束数量直接相关。在每次指令发射时，束调度器选择一个已准备好执行的束并将下个指令发射给束内的活动线程。束准备执行下一条指令花费的时钟周期数称为延迟，如果在延迟期间，每个时针周期内，束调度器有一些指令可为某些束发射就可获得完全的利用，或换句话说，当每个束的延迟可被其它束完全隐藏。要多少条指令才能隐藏延迟依赖指令吞吐量。例如，使用基本的单精度浮点算术运算指令隐藏 L 个时钟周期的延迟（已调度到 CUDA 核心上）。

- 对于计算能力 1.x 要 L/4（最近取整）条指令，因为多处理器每四个时针周期为每束发射一条该指令，如 G.3.1 节。
- 对于计算能力 2.0 要 L/2（最近取整）条指令，因为多处理器每两个时钟周期为两个束发射两条指令，如 G.4.1 节。

束没有准备好执行它下条指令的最常见原因是指令的输入操作为没有准备好。

如果所有输入操作数是寄存器，延迟是因为寄存器依赖，如，一些输入操作数是由一些还没有完成的在前面的指令写的。在紧接着的寄存器依赖的情况下（一些输入操作数是由前面的指令写的），延迟等于前面指令的执行时间且在此期间束调度器必须为不同的束调度指令。指令不同，执行时间也会变化，但是典型地大约 22 个时钟周期，等于计算能力 1.x 设备上的 6 个束，等于计算能力 2.0 设备上的 11 个束。

如果某些输入操作数在片下存储器中，延迟更高：400 到 800 时针周期。在如此高的延迟期保持束调度器繁忙要求的束数依赖于内核代码；一般，如果没有片下存储器的操作数的指令（如，大多数时候算术指令）数与片下存储器的操作的指令比率小的话（这个比率经常称为程序的运算密度），要求更多束。比如，如果比率是 10，为了隐藏大约 600 个时钟周期的延迟，对于计算能力 1.x 的设备大约要 15 个束，对于计算能力 2.0 的设备大约 30 个束。

另一个束没有准备好执行下一条指令的原因是在等待一些存储器栅栏（见 b.5 节）或同步节（见 B.6 节）。一个同步点能强制多处理器空闲，因为许多束要等待同一块内其它束完成同步点之前的指令。一个多处理器上有多个常驻块能够减少这种情况下的闲置，因为不同块内的束不用在同步点等待。

对于给定的内核调用，一个多处理器上的块和束数目依赖调用时的执行配置（参见 B.13 节），多处理器的存储器资源，和如 4.2 节描述的内核资源要求。为了帮助程序员基于寄存器和共享存储器要求选择合适的线程块尺寸，CUDA 软件开发工具包提供了一个称为 CUDA 占有率计算器的电子表格，在这里，占有率定义为常驻块数目和最大常驻块数目之比（附录 G 为各种计算能力给出了数据）。

内核使用的寄存器数目对常驻束数目有显著影响。例如，计算能力 1.2 的设备，如果内核使用了 16 个寄存器且每个块有 512 个线程且要求非常少的共享存储器，这样多处理器可常驻两个块，因为它们要求 2*512*16 个寄存器，这匹配多处理器的可用寄存器数目。但是如果只要内核多使用一个寄存器，就只有一个块常驻多处理器上了，因为两个块要求 2*512*7 个寄存器，这大于多处理器的可用寄存器数目。因此编译器试图在保证寄存器溢出（原文应当是这样，是否错误？）的前提下最小化寄存器使用和指令数目。可以使用-maxrregcount 编译选项或如 B.14 节描述的发射绑定控制寄存器使用。

一个块内需要的总共享存储器数目等于静态分配的和动态分配的共享存储器之和，另外在计算能力 1.x 的设备上，还包括传输内核参数需要的共享存储器用量（参见 B.1.4 节）。

寄存器、本地存储器、共享存储器和常量存储器的使用可在编译时使用—ptxas-options=-v 选项报告。注意每个双精度变量（在支持本地双精度的设备上，如计算能力 1.2 或更高的设备）和每个 long long 变量使用两个寄存器。但是计算能力 1.2 或以上的寄存器总量是低计算能力的至少两倍。

如果可能每个块的线程数应当是束尺寸的整数倍以避免因为束占用不足而浪费计算资源。

如上面提到地，对于一个给定的内核，执行配置对性能的影响依赖于内核代码。推荐实验后确定。应用也基于寄存器文件和共享存储器尺寸参数化执行配置，这依赖于设备的计算能力，也依赖于多处理器的数目和设备存储器带宽，所有的这些都可以可以使用运行时 API 和驱动 API 查询。

5.3 最大化存储器吞吐量

最大化应用的总体存储器吞吐量的第一步是最小化低带宽的数据传输。

这意味着要最小化主机存储器和设备存储器间的数据传输，详见 5.3.1 节，因为这带宽比全局存储器和设备之间数据传输的带宽要少。

也意味着通过最大化片上存储器使用最小化全局存储器和设备间的数据传输：如共享存储器和缓存（如计算能力 2.0 的设备上的 L1/L2 缓存，所有设备上的纹理缓存和常量缓存）。

共享存储器等价于用户管理的缓存：应用显式的分配和访问它。如 3.2.2 节所描述的，一个常用的编程模式是将来自设备存储器的数据存储到共享存储器，换句话说，让块内的每个线程：

- 从设备存储器中加载数据到共享存储器，
- 同步块内的其它线程以便每个线程能够安全读其它线程写的数据，
- 在共享存储器内处理数据，
- 如果需要的话再次同步以保证以结果更新了共享存储器，
- 将结果写回设备存储器

对于一些应用（如对于那些访问全局存储器是数据依赖的），传统的硬件管理的缓存更适用于发掘数据局部性。如 G.4.1 节所提到的，对于计算能力 2.0 的设备，同一片上存储器用于 L1 和共享存储器，至于在每个内核调用时为 L1 和共享存储器各分配多少则可以配置。

内核的存储器访问吞吐量依赖对于每种类型存储器的访问模式，其性能可相差几个数量级。最大化存储器吞吐量的下一步是依据 5.3.2.1 节，5.3.2.3 节，5.3.2.4 节和 5.3.2.5 节描述的最优化存储器访问模式重新组织存储器访问。对于全局存储器来说，这种优化非常重要，因为全局存储器的带宽比较低，所以没有优化全局存储器访问对性能有更大的影响。

5.3.1 主机和设备的数据传输

应用应当尽力最小化主机和设备间的数据传输。达到这种目标的一种方法是将更多的代码从主机上移到设备上，即使这意味着内核运行低并行性的计算。中间数据结构可能在设备上建立，被设备操作和销毁而用不着被主机映射或复制到主机存储器。

另外由于每次传输的消耗，将多次小的传输组合成一次大的传输会比单独传输更好。

对于有前端总线的系统，使用 3.2.5 节描述的分页锁定主机存储器进行主机和设备间的数据传输会获得更好的性能。

另外，在使用被映射主机存储器时（参见 3.2.5.3 节），没有必要分配任何设备存储器和显式在主机和设备间传输数据。数据传输会在每次内核访问映射存储器时隐式进行。为了最大化性能，这些存储器访问必须像全局存储器一样满足合并访问要求（参见 5.3.2.1 节）。假定被映射主机存储器只被读写一次，使用被映射主机存储器性能会提高。

在集成系统上，设备存储器和主机存储器在物理上是相同的，任何设备和主机间的数据传输都是多余的，此时应当使用被映射主机存储器。应用可以调用 cudaGetDeviceProperties()检查 integrated 属性或调用 cuDeviceGetAttribute()检查 CU_DEVICE_ATTRIBUTE_INTEGRATED 属性来查询设备是不是集成的。

5.3.2 设备存储器访问

访问可寻址空间（如，全局，本地，共享，常量或纹理存储器）的指令可能要被多次重新发射，这依赖于在一个束内线程访问的地址的分布。对于每种存储器这些分布如何影响指令吞吐量是由存储器种类决定的，这在下节描述。例如，对于全局存储器，作为一个通用的准则，地址越分散，吞吐量越小。

5.3.2.1 全局存储器

全局存储器在设备存储器中且设备存储器通过 32，64 或 128 个字节存储器通信访问。这些存储器访问是天然对齐的：只有 32，64 或 128 字节的设备存储器片段是对齐到它自身的尺寸，它们可以在一次通信中被读写。

当一个束执行一条访问全局存储器的指令时，它会合并束内线程的存储器的访问成一次或多次，这依赖于每个线程访问的字的尺寸和线程间存储器地址分布。一般，通信越多，除线程访问的字外转移的不用的字就越多，相应的降低了指令吞吐量。如，如果每个线程访问 4 字节产生了 32 字节的通信，吞吐量要除以 8.

要多少次通信和对吞吐量的最终影响都随计算能力变化。对于计算能力 1.0 和 1.1 的设备，线程间地址的分布满足合并访问的要求是非常严格的。更高计算能力的设备上就宽松了许多。对于计算能力 2.0 的设备，存储器通信是缓存的，挖掘了数据局部性以减少合并访问对性能的影响。G.3.2 和 G.4.2 节给出了全局存储器访问如何被各种计算能力的设备处理的细节。

为了最大化全局存储器吞吐量，最大化合并访问非常重要：

- 遵从 G.3.2 和 G.4.2 节的最优访问模式，
- 使用满足尺寸和对齐要求的数据类型，详见 5.3.2.1.1 节
- 在某些情况下，填充数据，如 5.3.2.1.2 节描述的访问二维数组。

5.3.2.1.1 尺寸和对齐要求

全局存储器指令支持读写长度为 1、2、4、8 或 16 字节的字。任何对全局存储器的数据访问（通过变量或指针）编译成一次单独的全局存储器指令当且仅当数据类型的尺寸为 1、2、4、8 或 16 字节并且数据是天然对齐的（如地址是尺寸的倍数）。

如果尺寸和对齐要求没有满足，访问被编译成交叉访问的多条指令，这不能完全合并。因此建议对全局存储器中的数据使用满足要求的数据类型。

B.3.1 节的像 float2 或 float4 这样的内置数据类型自动满足对齐要求。

对于结构体，其尺寸和对齐要求可使用对齐修饰符__align__(8)或 __align__(16)被编译器保证。如

```
struct __align__(8) {
    float x;
    float y;
};
or
struct __align__(16) {
    float x;
    float y;
    float z;
};
```

任何全局存储器中的变量的地址或驱动 API 或运行时 API 中的存储器分配例程返回的地址总是对齐到 256 字节。

读没有天然对齐的 8 字节或 16 字节的字会产生错误的结果（偏离一些字），所以要特别注意保证任何值或数组的起始地址对齐。一个典型的容易忽视的例子是使用一些自定义的全局存储器分配模式，如将多个数组的分配作为一次大的分配并为各数组划分分配的存储器，这种情况下，每个数组的起始地址是偏离块的起始地址的。

5.3.2.1.2 二维数组

一个常见的全局存储器访问模式是各个索引为(tx,ty)的线程使用下面的地址访问类型为 type*（满足 5.3.2.1.1 节的要求）、位于地址 BaseAddress、宽为 width 的二维数组的一个元素：

BaseAddress + width * ty + tx

为了全部满足合并访问，width 和块的宽度必须是束的整数倍（或对于计算能力 1.x 的设备是半束的整数倍）。

特别地，这意味着宽度不是束的整数倍的数组，在分配空间的时候行向上填充到最近的束的整数倍时，会更有效。手册中的 cudaMallocPitch()和 cudaMemAllocPitch()函数及相关的存储器复制函数保证程序员写出硬件无关的代码以分配服从这些限制的数组。

5.3.2.2 本地存储器

本地存储器访问仅对某些自动变量发生，如 B.2.5 节所述。编译器可能放入本地存储器的自动变量是：

- 不能确定是用常数量索引的数组，
- 可能消耗太多寄存器空间的大结构或数组，
- 如果内核使用了超过可用的寄存器的任何变量（也称为寄存器溢出）。

检查 PTX 汇编代码（编译时使用-ptx 或-keep 选项获得）可以分辨，如果一个变量在编译的第一阶段被放入本地存储器，会使用.local 助记符声明，使用 ld.local 和 st.local 访问。即使此时没有放入本地存储器中，在随后的编译阶段，如果觉得对于目标架构它消耗了太多的寄存器，也会被放入本地存储器中：使用 cuobjdump 检查 cubin 对象会分辨是不是这种情况。另外编译时使用—ptxas-options=-v 选项，编译器会报告内核总的本地存储器用量（lmem）。注意一些数学函数有的实现路径也可能访问了本地存储器。

本地存储器存在于设备存储器空间，所有本地存储器访问延迟像全局存储器一样高，带宽和全局存储器一样低，且服从于 5.3.2.1 节描述的存储器合并访问要求。本地存储器的组织使得连续的线程 ID 访问连续的 32 位字。只要束内所有线程访问同一相对地址（如数组变量的同一索引，结构体变量的同一成员）访问就可完全合并。

在计算能力 2.0 的设备上，本地存储器访问总是以和全局存储器访问（参见 G.4.2 节）同样的方式被缓存在 L1 和 L2。

5.3.2.3 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，对于束内所有线程来说，只要线程间不存在存储体冲突，访问共享存储器的速度与访问寄存器一样快，细节如下。

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），存储体可同步被访问。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步实现，整体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，访问必须序列化。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 路存储体冲突。

为了获得最大化的性能，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就很重要。这些在 G.3.3 和 G.4.3 节为计算能力 1.x 和计算能力 2.0 的分别描述。

5.3.2.4 常量存储器

常量存储器空间存在于设备存储器并被缓存到常理缓存中，参见 G.3.1 和 G.4.1 节。

对于计算能力 1.x 设备，一个束的常量存储器请求首先被分成两个请求，每半束一个，独立发射。

一个请求然后分成多个请求，使得它们的地址在初始访问中是不同，吞吐量减少了一个等于分立请求的数量的因子。

最终的请求等于常量缓存的吞吐量，如果缓存命中的话，否则等于设备存储器的吞吐量。

5.3.2.5 纹理存储器

纹理存储器空间存在于设备存储器中，并被缓存到纹理缓存，因此纹理获取仅需在缓存丢失时读取一次设备存储器，否则只需花费一次纹理缓存读取。纹理缓存已为二维空间位置而优化，因此读取二维相邻纹理地址的同一个束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获取，一次缓存命中将减少 DRAM 带宽要求，但不涉及获取延迟。

与从全局存储器或固定存储器读取设备存储器的方法相比，通过纹理获取读取设备存储器可能是一种更有优势的替代方法。

- 如果存储器读取不符合全局或常量存储器读取模式，纹理存储器会得到好性能（参见 5.3.2.1 和 5.3.2.4 节），如果在纹理获取的时候存在局部性，能得到更高的带宽（这不像计算能力 2.0 的设备那样明显，假使全局存储器读取被缓存）；
- 地址计算在内核外由特定单元处理；
- 在一个操作中包装的数据可能广播到独立的变量；
- 8 位和 16 位整形输入数据可能转化为 32 位浮点值，范围为[0.0，1.0]或[-1.0，1.0]（参见 3.2.4.1 节）。

然而，在同一内核调用，纹理缓存并不像全局存储器一样保持一致性，所以任何某个地址的纹理获取刚被同一内核写过，其结果没有定义。换句话说，线程能够读纹理存储器位置，只要这个位置被前面的内核调用或存储器复制更新过，但是并不是被当前线程更新或同一内核内其它线程更新。这只和从线性存储器中获取相关，因为任何情况下内核不能写 CUDA 数组。

5.4 最大化指令吞吐量

为了最大化指令吞吐量，应用应当：

- 最小化吞吐量较低的指令的使用；这包括为速度牺牲精度，如果不影响最终结果的话，

如使用内置函数取代常规函数（内置函数列在 C.2 节），单精度取代双精度，或将非正规数刷为 0。

- 最小化由流控指令引起的束分支（参见第 5.4.2 节）；
- 减少指令数目，例如，如 5.4.3 节所描述的只要可能就优化同步点或如 E.3 节所描述的使用 restrict 修饰指针。

本节，吞吐量以每时钟周期每个多处理器操作数目给定。对于尺寸为 32 的束，一条指令导致 32 个操作。

因此，如果 T 是每个时钟周期操作数目，指令吞吐量就是每 32/T 时钟周期一个指令。

尽管吞吐量是对一个多处理器而言。它们可以乘以设备中的多处理器个数以获得总个设备的吞吐量。

5.4.1 算术指令

表 5-1 给出了各种计算能力的设备其硬件本地支持的算术指令吞吐量。对于计算能力 2.0 的设备，每个时钟周期两个不同的束执行一半操作（参见 G.4.1 节）。

表 5-1 本地算术指令吞吐量（每时钟周期每多处理器）

	计算能力 1.x	计算能力 2.0
32-bit floating-point add, multiply, multiply-add	8	32
64-bit floating-point add, multiply, multiply-add	1	16
32-bit integer add, logical operation, shift, compare	8	32
24-bit integer multiply (__[u]mul24(x,y))	8	多条指令
32-bit integer multiply, multiply-add, sum of absolute difference	多条指令	32
32-bit floating-point reciprocal, reciprocal square root, base-2 logarithm (__log2f), base-2 exponential (exp2f),sine (__sinf), cosine (__cosf)	2	4
Type conversions	8	32

其它指令和函数基于本地指令实现。对于计算能力 1.x 和 2.0 的设备其实现可能不同，而不同的编译器版本编译后的本地指令数量也可能波动。对于已编译函数，可能有多条代码路径，这依赖于输入。可以使用 cuobjdump 检查 cubin 对象中的一个特定实现。

一些函数的实现已经在 CUDA 头文件中（math_functions.h，device_functions.h）。

一般，代码使用-ftz=true（非正规化数刷到 0）编译比使用-ftz=false 编译的性能要高。同样的代码使用-prec-div=false（低精度除法）编译的性能比-prec-div=true 编译的性能高，和代码使用-prec-sqrt=false（低精度平方根）编译比使用-prec-sqrt=true 编译性能要高。nvcc 用户手册更详细说明了这些标签。

单精度浮点加和内置乘

__fadd_r[d,u], __fmul_r[d,u], and __fmaf_r[n,z,d,u]（参见 C.2.1节)，对于计算能力1.x的设备，编译成几十个指令，但对于计算能力2.0的设备映射成单条本地指令

单精度浮点除

__fdividef(x, y) (参见C.2.1节)提供了比除法操作符快的单精度浮点除法

单精度浮点倒数平方根

为了保存IEEE-754的助记符，当倒数和平方根都是近似的时候（比如-prec-div=false

和-prec-sqrt=false），编译器将 1.0/sqrtf()优化为 rsqrtf()，所以应当在需要的时候显式的调用rsqrtf()。

单精度浮点平方根

单精度浮点平方根实现为倒数平方根的倒数而不是倒数平方根的乘法，因此对0和无穷能得到正确的结果。因此，它的吞吐量是计算能力1.x的每时钟周期1个操作，对于计算能力2.0每时钟周期2个操作。

正弦和余弦

sinf(x), cosf(x), tanf(x), sincosf(x),和对应双精度指令比较昂贵，如果x的值大的话会更为昂贵。

更精确地，参数归约代码（参见math_functions.h的实现）包含两个称为快速路径和慢路径的代码。

快速路径用于参数足够小且本质上只包含一些乘加操作。慢路径用于参数大且包含长计算要求以获得整个参数范围内的正确结果。

目前，三角任何函数的参数归约代码在参数小于48039.0f时为单精度函数选择快路径，小于2147483648.0时为双精度选择快路径。

因为慢路径相比快路径要更多的寄存器，故将中间结果放到本地存储器中试图减少寄存器的用量，由于本地存储器的高延迟和低带宽（参见5.3.2.2节），可能影响性能。目前，为单精度使用了28字节的本地存储器，而双精度使用了44个。但具体的数量可能会改变。

由于慢路径使用了长计算量和本地存储器，三角任何函数的吞吐量慢路径比快路径相差一个数量级。

整数算术

在计算能力1.x的设备上，32位整数乘法是使用非本地支持的乘法指令实现的。24位整形乘法是通过__[u]mul24内置指令（参见C.2.3节）本地支持的。在指令限制的内核中，只要可能就使用__[u]mul24取代32位乘法操作符，性能一般会得到提升的。但是如果使用__[u]mul24阻止了编译器优化，也可能得到相反的效果。

在计算能力2.0的设备上，32位整数乘法是本地支持的，而24位不是。__[u]mul24使用多条指令使用，因此不应当使用。

整数除法和模余非常昂贵，只要可能就应当避免使用或用位操作符取代：如果n是2的乘方，（i/n）等于(i>>log2(n))，而(i%n)等于(i & (n-1))；如果n是字面量，编译器会自动实现转换。

__brev、__brevll、__popc和__popcll（参见C.2.3节），对于计算能力1.x编译成几十个指令，但是对于计算能力2.0__brev和__popc映射成一条指令，而__brell和__popcll就几条指令。

__clz、__clzll、__ffs和__ffsll（参见C.2.3节）编译成的指令数在计算能力2.0的设备上比计算能力1.x的要少。

类型转换

有时，编译器会插入类型转换指令，这增加了执行周期。下面的就是这种情况：

- 函数操作的变量的类型为char或short，这的操作数一般会被转化为int。
- 双精度浮点常量（如，没有后缀的类型常量定义）用作单精度计算的输入。

后面一种情况可以使用单精度浮点常量避免，使用后缀f定义，如3.141592653589793f, 1.0f, 0.5f。

5.4.2 控制流指令

任何流控制指令（if、switch、do、for、while）都会导致同一束的线程分支（如走向不同的执行路径），从而显著影响有效指令吞吐量。如果出现这种情况，不同的执行路径必须序列化，因而增加了为该束执行的指令总数。当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。

为了在控制流依赖线程 ID 的情况下获得最佳性能，应控制条件以最小化分支束的数量。这是可行的，因为束在块内的分布情况是确定的，如 4.1 节所提到的。一个简单的例子就是，当控制条件仅依赖于 (threadIdx / WSIZE) 时，其中的 WSIZE 是束的大小，这种情况下，不会出现任何束内分支，因为控制条件与束完美对齐。

有些时候，编译器可能会展开循环或使用分支谓词来优化 if 或 switch 语句，下面将详细说明。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 E.2 节）。

在使用分支谓词时，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，编译器才会使用有谓词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束的条件，则此阈值为 7，否则为 4。

5.4.3 同步指令

对于计算能力 1.x 的设备，__syncthreads()是每个时钟周期 8 个操作，对于计算能力 2.0 的设备每时钟周期 16 个操作。

注意；__syncthreads()能通过强制某些多处理器闲着从而影响性能，细节见 5.2.3 节。

由于束一次执行一条常用指令，束内的线程是隐式同步的，这可用于忽略 __syncthreads()以提升性能。

例如，下面的代码，为了获得预想结果（如，对于 i>0,result[i]=2*myArray[i]），__syncthreads()调用一个也不能少。没有同步，再次引用 myArray[tid]都可能返回 2 或者初始化值，这由存储器读是发生在存储器写之前还是之后决定。

```
// myArray is an array of integers located in global or shared
// memory
__global__ void MyKernel(int* result) {
int tid = threadIdx.x;
...
int ref1 = myArray[tid];
__syncthreads();
myArray[tid + 1] = 2;
__syncthreads();
int ref2 = myArray[tid];
result[tid] = ref1 * ref2;
...
}
```

然而，在下面稍微改了点的代码中，线程保证属于同一束，所以不用任何

__syncthreads().

```
// myArray is an array of integers located in global or
shared
// memory
__global__ void MyKernel(int* result) {
int tid = threadIdx.x;
...
if (tid < warpSize) {
int ref1 = myArray[tid];
myArray[tid + 1] = 2;
int ref2 = myArray[tid];
result[tid] = ref1 * ref2;
}
...
}
```

简单去掉__syncthreads()是不够的；myArray 必须声明为 volatile 如 B.2.4 节所描述。

附录 A 支持 CUDA 的 GPU

表 A-1 列出了所有支持 CUDA 的设备，它们的计算能力，多处理器数目和 CUDA 核心的数目。

另外，核心频率和设备存储器总量可以使用驱动 API 或运行时 API 查询（参见手册）。

表 A-1.支持 CUDA 的设备、计算能力、多处理器数和 CUDA 核心数			
	计算能力	多处理器数量	CUDA 核心数量
GeForce GTX 295	1.3	2*30	2*240
GeForce GTX285,GTX 280	1.3	30	240
GeForce GTX 260	1.3	24	192
GeForce 9800 GX2	1.1	2*16	2*128
GeForce GTS 250,GTS 150, 9800 GTX,9800GTX+,8800 GTS 512	1.1	16	128
GeForce 8800 Ultra, 8800 GTX	1.0	16	128
GeForce 9800 GT,8800 GT,GTX 280M,9800M GTX	1.1	14	112
GeForce GT240	1.2	12	96
GeForce GT 130,9600 GSO, 8800 GS, 8800M GTX,GTX260M,9800M GT	1.1	12	96
GeForce 8800 GTS	1.0	12	96
GeForce 9600 GT, 8800M GTS,9800M GTS	1.1	8	64
GeForce GT 220	1.2	6	48
GeForce 9700M GT	1.1	6	48
GeForce GT 120,9500 GT, 8600 GTS, 8600 GT,9700M GT,9650M GS,9600M GT,9600M GS,9500M GS,8700M GT, 8600M GT, 8600M GS	1.1	4	32
GeForce 210	1.2	2	16
GeForce G100,8500 GT, 8400 GS, 8400M GT,9500M G,9300M G,8400M GS,9400 mGPU,9300 mGPU,8300 mGPU,8200 mGPU,8100 mGPU	1.1	2	16
GeForce 9300M GS,9200M GS,9100M G,8400M G,	1.1	1	8
Tesla S1070	1.3	4*30	4*240
Tesla C1060	1.3	30	240
Tesla S870	1.0	4*16	4*128
Tesla D870	1.0	2*16	2*128
Tesla C870	1.0	16	128

Quadro Plex 2200 D2	1.3	2*28	2*240
Quadro Plex 2100 D4	1.1	4*14	4*112
Quadro Plex 2100 Model S4	1.0	4*16	4*128
Quadro Plex 1000 Model IV	1.0	2*16	28128
Quadro FX 5800	1.3	30	240
Quadro FX 4800	1.3	24	192
Quadro FX 4700 X2	1.1	2*14	2*112
Quadro FX 3700M,FX 3800M	1.1	16	128
Quadro FX 5600	1.0	16	128
Quadro FX 3700	1.1	14	112
Quadro FX 2800M	1.1	12	96
Quadro FX 4600	1.0	12	96
Quadro FX 1800M	1.2	9	72
Quadro FX 3600M	1.1	8	64
Quadro FX 880M	1.2	6	48
Quadro FX 2700M	1.1	6	48
Quadro FX 1700, FX 570, NVS 320M, FX 1700M,FX 1600M,FX 770M,FX 570M	1.1	4	32
Quadro FX 380M	1.2	2	16
Quadro FX 370, NVS 290, NVS 140M, NVS 135M, FX 360M	1.1	2	16
Quadro FX 370M,NVS 130M	1.1	1	8

附录 B C 语言扩展

B.1 函数类型限定符

函数类型限定符指定函数是在主机上执行还是在设备上执行和是从主机上调用还是从设备上调用。

B.1.1 _device_

使用 _device_ 限定符声明的函数：

- 在设备上执行；
- 仅可通过设备调用。

B.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；
- 只能通过主机调用。

B.1.3 _host_

使用 _host_ 限定符声明的函数：

- 在主机上执行；
- 仅可通过主机调用。

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，函数都将仅为主机进行编译。

但 _host_ 限定符也可与 _device_ 限定符一起使用，此时函数将为主机和设备分别进行编译。

B.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。
device 和 _global_ 函数不得使用可变参数列表。
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。
对 _global_ 函数的任何调用都必须按第 B.13 节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 函数参数将传递给设备：

- 计算能力 1.x 的使用共享存储器传递，且大小限制为 256 字节，
- 计算能力 2.0 的通过常量存储器传递，且其大小限制为 4K 字节。

B.2 变量类型限定符

变量类型限定符指定变量在设备上的存储位置。

B.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，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；
- 网格内的所有线程都可访问，主机也可通过运行时库访问。

B.2.2 _constant_

constant 限定符可与 _device_ 限定符一起使用，此时__device__是可选的，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；
- 网格内的所有线程都可访问，主机也可通过运行时库访问。

B.2.3 _shared_

shared 限定符可与 _device_ 限定符一起使用，此时__device__是可选的，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；
- 只可通过块内的所有线程访问。

将共享存储器中的变量声明为动态数组时，例如：

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

数组的大小将在发射时确定（参见 B.13 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：
则应通过以下方法声明和初始化数组：

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

在动态分配的共享存储器，必须以下面的方式声明和初始化：

```
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];
}
```

要注意指针要对齐到它指向的类型，所以下面的代码不能工作，因为 array1 没有对齐到 4 字节。

```
extern __shared__ char array[];
__device__ void func() // __device__ or __global__ function
{
short* array0 = (short*)array;
float* array1 = (float*)&array0[127];
}
```

表 B-1 列出了所有内置类型的对齐要求。

B.2.4 volatile

只有在执行__threadfence_block()、__threadfence()或__syncthreads()（参见 B.5 和 B.6 节）后，在此之前对全局存储器或共享存储器写入的才能保证对其它线程可见。只要满足这个要求，编译器可任意优化对全局存储器或共享存储器的读写。例如，下面的例子，第一次引用 myArray[tid]编译成一次全局或共享存储器读指令，但是第二次并非如此，因为编译器简单的重用前一次读的结果。

```
// myArray is an array of non-zero integers
// located in global or shared memory
  __global__ void MyKernel(int* result) {
int tid = threadIdx.x;
int ref1 = myArray[tid] * 1;
myArray[tid + 1] = 2;
int ref2 = myArray[tid] * 1;
result[tid] = ref1 * ref2;
}
```

因此，线程 tid 中 ref2 可能不因为线程 tid-1 用 2 重写了 myArray[tid]而等于 2。

这种行为可以使用 volatile 关键字改变：如果全局存储器或共享存储器中的变量被声明为 volatile，编译器假定它的值可能在任何时候被其它线程改变，因此每次对它的引用都会被编译成一条实际的读存储器指令。

注意即使 myArray 被声明为 volatile 也不能保证在线程 tid 中等于 2，因为 tid 可能在线程 tid-1 重写 myArray 之前就已经读了。需要使用 5.4.3 节提到的同步。

B.2.5 限制

不允许为在主机上执行的函数内的 struct 和 union 成员、形参和局部变量使用这些限定符。

shared 和 _constant_ 变量具有隐含的静态存储。

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，仅可通过主机运行时函数从主机指派（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。

如果在主机上执行的代码中解引用全局或共享存储器指针，或者在设备上执行的代码中解引用主机存储器指针，将结果没有定义，往往会出现分区错误和应用程序终止。

通过取 _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_ 变量的地址仅可在主机代码中使用。

B.3 内置变量类型

B.3.1

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 个组件分别可通过字段 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)。

在主机代码中，这些向量类型的对齐要求等于它们的基本类型的对齐要求。但是设备上的情况有时会有不同。详见表 B-1。

表B-1.内置类型的对齐要求	
Type	**Alignment**
char1, uchar1	1
char2, uchar2	2
char3, uchar3	1
char4, uchar4	4
short1, ushort1	2
short2, ushort2	4
short3, ushort3	2
short4, ushort4	8
int1, uint1	4
int2, uint2	8
int3, uint3	4
int4, uint4	16
long1, ulong1	Same as int1 or longlong1 depending on platform
char1, uchar1	1
char2, uchar2	2
long2, ulong2	Same as int2 or longlong2 depending on platform
long3, ulong3	Same as int3 depending on platform
long4, ulong4	Same as int4 depending on platform
longlong1	8
longlong2	16
float1	4
float2	8
float3	4
float4	16
double1	8

B.3.2 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，未指定的任何组件都将初始化为 1。

B.4 内置变量

内置类型指定块和网格的尺寸及块和线程索引，它们只在在设备上执行的函数内有效。

B.4.1 gridDim

此变量的类型为 dim3（参见 B.3.2 节），包含网格的维度。

B.4.2 blockIdx

此变量的类型为 uint3（参见 B.3.1 节），包含网格内的块索引。

B.4.3 blockDim

此变量的类型为 dim3（参见 B.3.2 节），包含块的维度。

B.4.4 threadIdx

此变量的类型为 uint3（参见第 B.3.1 节），包含块内的线程索引。

B.4.5 warpSize

此变量的类型为 int，包含以线程为单位的 warp 块大小。

B.4.6 限制

- 不允许取任何内置变量的地址。
- 不允许为任何内置变量赋值。

B.5 存储器栅栏函数

```
void    threadfence_block();
```

等待直到在此函数调用前的所有全局存储器和共享存储器访问对块内所有线程可见。

```
void    threadfence();
```

等待直到在此函数调用前的所有全局存储器和共享存储器访问对下列线程可见：

- 对于共享存储器，对块内所有线程，
- 对于全局存储器，对设备上的所有线程可见。

```
void __threadfence_system();
```

等待直到在此函数调用前的所有全局存储器和共享存储器访问对下列线程可见：

- 对于共享存储器，对块内所有线程可见，
- 对于全局存储器，对设备上的所有线程可见，
- 分页锁定主机存储器，对主机线程可见（参见 3.2.5.3 节）。

__threadfence_system()只支持计算能力 2.0 的设备。

一般，当一线程发射一系列特殊顺序的写存储器指令，其它线程看到的写存储器效果顺序不同，__threadfence_block()，__threadfence()和__threadfence_system()可用于保证顺序。

一个用处是当线程消费其它线程生产的数据时，就如下面的代码描述的一个内核在一次调用中计算一个 N 个数的数组的和。每个块先计算数组的一部分并将结果存储到全局存储器。当所有的块都完成后，最后一个块从全局存储器中读取部分和并将它们加和得到最终结果。为了决定那个块最后完成，每个块原子地递增计数器以通知计算完成并存储部分和（参见 B.10 节了解原子函数）。最后一个块是得到计数器的值为 gridDim.x-1 的块。如果在存储部分和和递增计数器值时没有放置栅栏，计数器值可能在部分和存储之前增加了，这样最后一个块可能在部分和没有更新就开始读取部分和了。

```
__device__ unsigned int count = 0;
__shared__ bool isLastBlockDone;
__global__ void sum(const float* array, unsigned int N, float* result)
{
    // Each block sums a subset of the input array
    float partialSum = calculatePartialSum(array, N);
    if (threadIdx.x == 0) {
        // Thread 0 of each block stores the partial sum to global memory
        result[blockIdx.x] = partialSum;
        // Thread 0 makes sure its result is visible to
        // all other threads
        __threadfence();
        // Thread 0 of each block signals that it is done
        unsigned int value = atomicInc(&count, gridDim.x);
        // Thread 0 of each block determines if its block is
        // the last block to be done
        isLastBlockDone = (value == (gridDim.x - 1));
    }
    // Synchronize to make sure that each thread reads
    // the correct value of isLastBlockDone
    __syncthreads();
    if(isLastBlockDone){
        // The last block sums the partial sums
        // stored in result[0 .. gridDim.x-1]
        float totalSum = calculateTotalSum(result);
        if (threadIdx.x == 0) {
            // Thread 0 of last block stores total sum
            // to global memory and resets count so that
            // next kernel call works properly
            result[0] = totalSum;
            count = 0;
        }
    }
}
```

B.6 同步函数

__syncthreads()

等待直到块内所有线程达到此同步点并且在此点之前所有的共享存储器和全局存储器访问对块内所有线程可见。

_syncthreads() 用于协调同一个块的线程之间的通信。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。

计算能力 2.0 的设备还支持下面三种版本的__syncthreads()。

int __syncthreads_count(int predicate);

等价于包含另外特性的__syncthreads()，该特性就是为块内所有线程计算 predicate 并返回 predicate 值不是 0 的线程数目。

int __syncthreads_and(int predicate);

等价于包含另外特性的__syncthreads()，该特性就是为块内所有线程计算 predicate 并当且仅当所有线程的 predicate 值不是 0 才返回非零值。

int __syncthreads_or(int predicate);

等价于包含另外特性的__syncthreads()，该特性就是为块内所有线程计算 predicate 并只要有一个线程的 predicate 值不是零，返回的就是非零值。

B.7 数学函数

C.1 节列出了目前在设备代码中支持的所有标准 C/C++库数学函数以及它们各自的误差上限。当在主机代码中执行时，如果可用的话，主机会使用 C 运行时的实现。

对于 C.1 节的某些函数，在设备运行时组件中，有一个精确度差点但更快的版本；名字相同但加了__前缀（例如__sinf(x)）。这些内置函数列在了 C.2 节，包括它们各自的误差上限。

有一个编译器选项（-use_fast_math)强制表 B-2 中的每个函数编译成它们的内置版本。除了精度减弱外，在某些特别的情况下，需要特殊处理。一个更强健的方法是选择性的在能够提升性能而特性的改变（如精度减弱和特殊情况处理）又能够忍受的地方使用内置版本。

表 B-2.受-use_fast_math 影响的函数

Operator/Function	**Device Function**
x/y	**__fdividef(x,y)**
sinf(x)	**__sinf(x)**
cosf(x)	**__cosf(x)**
tanf(x)	**__tanf(x)**
sincosf(x,sptr,cptr)	**__sincosf(x,sptr,cptr)**
logf(x)	**__logf(x)**
log2f(x)	**__log2f(x)**
log10f(x)	**__log10f(x)**

B.8 纹理函数

对于纹理函数，纹理参考的可变与不可变属性综合决定了怎样解释纹理坐标，在纹理获取期间做了那些处理，同时返回值由纹理获取传递。不可变的属性在 3.2.4.1 节描述。可变的属性在 3.2.4.2 节描述。纹理获取在附录描述。

附录 G 列出了不同计算能力的最大的纹理宽、高和深度。

B.8.1 tex1Dfetch()

```
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);
```

使用整形坐标获取绑定到纹理参考 texRef 的线性存储器。不支持纹理过滤和寻址模式。对于整形，这些函数可选的将整形提升到单精度浮点型。

除了上面的函数，也支持二元组和四元组；例如：

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

使用纹理坐标 x 获取绑定到 texRef 的线性存储器区域。

B.8.2 tex1D()

```
template<class Type, enum cudaTextureReadMode readMode> Type tex1D(texture<Type, 1,
readMode> texRef, float x);
```

使用浮点纹理坐标 x 获取绑定到纹理参考 texRef 的 CUDA 数组。

B.8.3 tex2D()

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

使用纹理坐标 x 和 y 获取绑定到纹理参考 texRef 的 CUDA 数组或线性存储器区域。

B.8.4 tex3D()

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

使用纹理坐标 x, y 和 z 获取绑定到 texRef 的 CUDA 数组。

B.9 时间函数

```
clock_t clock();
```

在设备代码中执行时，返回每个多处理器计数器的值，此计数器随每一次时钟周期而递增。在内核发射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。

B.10 原子函数

原子函数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。说操作是原子的，是因为它可在不受其他线程干扰的前提下执行。换句话说，在操作完成前，其他任何线程都无法访问此地址。

原子操作仅适用于有符号和无符号整型，对于计算能力 2.0 的设备，atomAdd()例外，对于所有的设备 atomExch()例外，它们能够作用于单精度浮点数。

原子指令只能在设备代码中使用，而且只在计算能力 1.1 或更高上可用。

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。

注意：从主机或设备的观点来看，作用在被映射分页锁定存储器（参见 3.2.5.3 节）上的原子函数不是原子的。

B.10.1 数学函数

B.10.1.1 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);
float atomicAdd(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，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。

浮点版本的 atomAdd()只支持计算能力 2.0 的设备。

B.10.1.2 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。

B.10.1.3 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。

B.10.1.4 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。

B.10.1.5 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。

B.10.1.6 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。

B.10.1.7 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。

B.10.1.8 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。

B.10.2 位逻辑函数

B.10.2.1 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。

B.10.2.2 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。

B.10.2.3 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。

B.11 束表决（warp vote）函数

只有计算能力为 1.2 或更高的设备支持束表决（参见 4.1 节了解束的定义）函数。

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

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

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

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

```
unsigned int __ballot(int predicate);
```

为束内所有线程计算 predicate 值，并返回一个整数，如果束内第 N 个线程的 predicate 值为非零，则该整数的第 N 位为 1。计算能力为 2.0 的设备支持。

B.12 取样计数器函数

每个多处理器有一组十六个硬件计数器，应用可以调用__prof_trigger()函数使用一条指令递增计数器。

```
void __prof_trigger(int counter);
```

索引为 counter 的每个多处理器的硬件计数器每束增加 1，8 号和 15 号计数器保留，应用不能使用。

第一个多处理器的 0，1，...，7 号计数器值可通过 CUDA profiler 取得，方式是在 profiler.conf 文件中列出 prof_trigger_00，prof_trigger_01，...，prof_trigger_07，等等（详见 profiler 手册）。在每次内核调用前，所有的计数器重置（注意当应用通过 CUDA 调试器或 CUDA profiler 运行时，所有的发射都是同步的）。

B.13 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。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（参见 3.3.9.1 节了解流的详细内容）。

使用驱动 API 时，执行配置通过一系列驱动函数调用指定，详见 3.3.3 节。

使用运行时 API（参见 3.2 节）时，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 B.3.2 节），指定网格的维度和大小，Dg.x * Dg.y 等于所发射的块数量，Dg.z 必须等于 1；
- Db 的类型为 dim3（参见 B.3.2 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动态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求值，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值，为设备指定的最大值参看附录 G，或 Ns 大于设备上可用的共享存储器最大值减去静态分配、函数参数（为计算能力 1.x）和执行配置所需的共享存储器数量，则函数将失败。

B.14 发射绑定

如 5.2.3 节详细讨论的那样，内核使用的寄存器越少，常驻的线程和线程块可能就越多，这能够提升性能。

因此，编译器会试探性的在保证寄存器溢出（参见 5.3.2.2 节）的前提下最小化寄存器使用和最小化指令数量。应用可以以发射绑定的形式提供额外的信息给编译器以辅助这种试探，发射绑定在定义内核函数时使用__launch_bounds__()限定符指定。

```
__global__ void __launch_bounds__(maxThreadsPerBlock, minBlocksPerMultiprocessor)
MyKernel(...){
  ...
}
```

- maxThreadsPerBlock 指定内核发射时块内的最大线程数；它被编译成.maxntid PTX 指令；
- minBlocksPerMultiprocessor 是可选的，其指定每个多处理器最小常驻块数量；它被编译成.minnctapersm PTX 指令。

如果指定了发射绑定，编译器从它们得到内核使用的寄存器的上限 L，以保证 minBlocksPerMultiprocessor 个块，每个块内 maxThreadsPerBlock 线程能常驻多处理器（参见 4.2 节了解内核使用的寄存器数量和块分配的寄存器数量之间的关系）。编译可以用以下方式优化寄存器的使用。

- 如果初始的寄存器用量超过 L，编译器会减少它到等于或小于 L，经常以使用本地存储器和/或增加指令数目为代价。
- 如果初始的寄存器用量小于 L，
  - 如果指定了 maxThreadsPerBlock 但 minBlocksPerMultiprocessors 没有，编译器使用;maxThreadsPerBlock 为 n 和 n+1 个常驻块确定寄存器用量限度（例，5.2.3 节的例子，少使用一个寄存器就为多一个常驻块提供了空间），然后像没有指定发射绑定一样应用相似的试探；
  - 如果 minBlocksPerMultiprocessor 和 maxThreadsPerBlock 都指定，编译器可能增加寄存器使用以减少指令数量和更好的隐藏单线程指令延迟。

如果每个块线程数量超过了 maxThreadsPerBlock，内核发射失败。

为既定内核优化发射绑定依据主架构修订号变化。例子代码展示了怎样使用 3.1.4 节的 __CUDA_ARCH 宏解决这个问题。

```
#define THREADS_PER_BLOCK 256
#if __CUDA_ARCH__ >= 200
    #define MY_KERNEL_MAX_THREADS (2 * THREADS_PER_BLOCK)
    #define MY_KERNEL_MIN_BLOCKS 3
#else
    #define MY_KERNEL_MAX_THREADS THREADS_PER_BLOCK
    #define MY_KERNEL_MIN_BLOCKS 2
#endif
// Device code
__global__ void __launch_bounds__(MY_KERNEL_MAX_THREADS,
MY_KERNEL_MIN_BLOCKS) MyKernel(...) {
  ...
}
```

通常，使用最大块内线程数量（__launch_bounds__()的第一个参数指定）调用MyKernel，在执行配置时，倾向使用MY_KERNEL_MAX_THREADS作为块内线程数：

```
// Host code MyKernel<<<blocksPerGrid, MY_KERNEL_MAX_THREADS>>>(...);
```

但是这不会工作，因为如3.1.4节提到的__CUDA_ARCH并没有在主机代码中指定，所以即使__CUDA_ARCH大于200，MyKernel也会以每块256个线程发射。块内线程数应当以下面的方式确定：

- 或者在编译时使用不依赖__CUDA_ARCH__的宏，如

```
// Host code
MyKernel<<<blocksPerGrid, THREADS_PER_BLOCK>>>(...);
```

- 或者在运行时基于计算能力

```
// Host code
cudaGetDeviceProperties(&deviceProp, device);
int threadsPerBlock = (deviceProp.major >= 2 ?
    2 * THREADS_PER_BLOCK : THREADS_PER_BLOCK);
MyKernel<<<blocksPerGrid, threadsPerBlock>>>(...);
```

使用ptxas-options=-v编译器选项可以报告寄存器用量。常驻块数量可从CUDA profiler中给出的占用率（参见5.2.3节了解占用率的定义）得到。

对于__global__函数的寄存器用量也可以使用;-maxrregcount编译器选项控制。对于发射绑定的函数，—maxrregcount值会被忽略。

附录 C 数学函数

C.1 节中列举的函数主机和设备函数都可使用，但 C.2 节中的函数仅能在设备函数中使用。

注意浮点函数是重载的，所以对于给定的函数有三种原型：

(1) 双精度 函数名(double)，如 double log(double)
(2) 单精度 函数名(float)，如 float log(float)
(3) 单精度 函数名 f(float)，如 float logf(float)

这意味着，特别地，传入单精度参数通常返回单精度结果（上面的变体(2)和(3)）。

C.1 标准函数

这一节列出了所有设备代码支持的数学标准库函数。此外还指定了各函数在设备上执行时的误差范围。当函数在主机上执行，而主机并未提供此函数时，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，但并不彻底，所以不能保证没有偏差。

C.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。但在计算能力 1.x 的设备上，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 C.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() 均可映射到一条指令。

表 C-1. 数学标准库函数及其最大 ULP 错误

最大错误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 就近舍入到最近偶数） （在计算能力 1.x 设备上，加法合并入 FMAD 时除外）
x*y	0（IEEE-754 就近舍入到最近偶数） （在合并入 FMAD 时除外）
x/y	计算能力>=2 的设备，使用-prec-div=true 编译时，0；其它，2（整个范围）
1/x	计算能力>=2 的设备，使用-prec-div=true 编译时，0；其它，1 (整个范围)
1/sqrtf(x) rsqrtf(x)	2 (整个范围)，只有当它被编译器转化为 rsqrtf(x) 才应用到 1/sqrtf(x)
sqrtf(x)	计算能力>=2 的设备，使用-prec-div=true 编译时，0；其它，3 (整个范围)
cbrtf(x)	1 (整个范围)
rcbrtf(x)	2（整个范围）
hypotf(x,y)	3 (整个范围)
expf(x)	2 (整个范围)
exp2f(x)	2 (整个范围)
exp10f(x)	2 (整个范围)

expm1f(x)	1 (整个范围)
logf(x)	1 (整个范围)
log2f(x)	3 (整个范围)
log10f(x)	3 (整个范围)
log1pf(x)	2 (整个范围)
sinf(x)	2 (整个范围)
cosf(x)	2 (整个范围)
tanf(x)	4 (整个范围)
sincosf(x,sptr,cptr)	2 (整个范围)
asinf(x)	4 (整个范围)
acosf(x)	3 (整个范围)
atanf(x)	2 (整个范围)
atan2f(y,x)	3 (整个范围)
sinhf(x)	3 (整个范围)
coshf(x)	2 (整个范围)
tanhf(x)	2 (整个范围)
asinhf(x)	3 (整个范围)
acoshf(x)	4 (整个范围)
atanhf(x)	3 (整个范围)
powf(x,y)	8 (整个范围)
erff(x)	3 (整个范围)
erfcf(x)	6 (整个范围)
erfinvf(x)	5(整个范围)
erfcinvf(x)	7(整个范围)
lgammaf(x)	6 (outside interval -10.001 ... -2.264; larger inside)
tgammaf(x)	11 (整个范围)
fmaf(x,y,z)	0 (整个范围)
frexpf(x,exp)	0 (整个范围)
ldexpf(x,exp)	0 (整个范围)
scalbnf(x,n)	0 (整个范围)
scalblnf(x,l)	0 (整个范围)
logbf(x)	0 (整个范围)
ilogbf(x)	0 (整个范围)
fmodf(x,y)	0 (整个范围)
remainderf(x,y)	0 (整个范围)
remquof(x,y,iptr)	0 (整个范围)
modff(x,iptr)	0 (整个范围)
fdimf(x,y)	0 (整个范围)
truncf(x)	0 (整个范围)
roundf(x)	0 (整个范围)
rintf(x)	0 (整个范围)
nearbyintf(x)	0 (整个范围)
ceilf(x)	0 (整个范围)
floorf(x)	0 (整个范围)
lrintf(x)	0 (整个范围)
lroundf(x)	0 (整个范围)
llrintf(x)	0 (整个范围)
llroundf(x)	0 (整个范围)
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

C.1.2 双精度浮点函数

下面列举的错误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 C-2. 数学标准库函数及其最大 ULP 错误

最大错误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 就近舍入）
x*y	0 (IEEE-754 舍入到最近偶数)
x/y	0 (IEEE-754 舍入到最近偶数)
1/x	0 (IEEE-754 舍入到最近偶数)
sqrt(x)	0 (IEEE-754 舍入到最近偶数)
rsqrt(x)	1 (整个范围)
cbrt(x)	1 (整个范围)
rcbrt(x)	1(整个范围)
hypot(x,y)	2 (整个范围)
exp(x)	1 (整个范围)
exp2(x)	1 (整个范围)
exp10(x)	1 (整个范围)
expm1(x)	1 (整个范围)
log(x)	1 (整个范围)
log2(x)	1 (整个范围)
log10(x)	1 (整个范围)
log1px(x)	1 (整个范围)
sin(x)	2 (整个范围)
cos(x)	2 (整个范围)
tan(x)	2 (整个范围)
sincos(x,sptr,cptr)	2 (整个范围)
sinpi(x)	2(整个范围)
asin(x)	2 (整个范围)
acos(x)	2 (整个范围)
atan(x)	2 (整个范围)
atan2(y,x)	2 (整个范围)
sinh(x)	1 (整个范围)
cosh(x)	1 (整个范围)

tanh(x)	1 (整个范围)
asinh(x)	2 (整个范围)
acosh(x)	2 (整个范围)
atanh(x)	2 (整个范围)
pow(x,y)	2 (整个范围)
erf(x)	2 (整个范围)
erfc(x)	5 (整个范围)
erfinv(x)	8(整个范围)
erfcinv(x)	8(整个范围)
lgamma(x)	4 (outside interval -11.0001 ... -2.2637; larger inside)
tgamma(x)	8 (整个范围)
fma(x,y,z)	0 (IEEE-754 round-to-nearest-even)
frexp(x,exp)	0 (整个范围)
ldexp(x,exp)	0 (整个范围)
scalbn(x,n)	0 (整个范围)
scalbln(x,l)	0 (整个范围)
logb(x)	0 (整个范围)
ilogb(x)	0 (整个范围)
fmod(x,y)	0 (整个范围)
remainder(x,y)	0 (整个范围)
remquo(x,y,iptr)	0 (整个范围)
modf(x,iptr)	0 (整个范围)
fdim(x,y)	0 (整个范围)
trunc(x)	0 (整个范围)
round(x)	0 (整个范围)
rint(x)	0 (整个范围)
nearbyint(x)	0 (整个范围)
ceil(x)	0 (整个范围)
floor(x)	0 (整个范围)
lrint(x)	0 (整个范围)
lround(x)	0 (整个范围)
llrint(x)	0 (整个范围)
llround(x)	0 (整个范围)
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

C.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。

C.2 内置函数

这一节列举了仅在设备代码中支持的内置函数。这些函数中包括第 C.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（_sinf(x)）。

使用 _rn 前缀的函数将使用就近舍入到偶数模式操作；

使用 _rz 前缀的函数将使用向零舍入模式操作；

使用 _ru 前缀的函数将使用上舍入（向正无穷大）模式；

使用 _rd 前缀的函数将使用下舍入（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**。

C.2.1 单精度浮点函数

编译器从未将_fadd_rn() 和 _fmul_rn() 映射的加法和乘法操作并入 FMAD 中。与此相比，“*”和“+”运算符生成的加法和乘法将频繁并入 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 C-3 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。

_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。

__float2ll_[rn,rz,ru,rd](x)（__float2ull_[rn,rz,ru,rd](x)）用指定的 IEEE-754 舍入模式将单精度浮点 x 转化为 64 位有符号（无符号）整数。

表 B-3　CUDA 运行时库支持的单精度浮点内部函数及其误差范围

函数	误差范围
__fadd_[rn,rz,ru,rd](x,y)	符合 IEEE
__fmul_[rn,rz,ru,rd](x,y)	符合 IEEE
__fmaf_[rn,rz,ru,rd](x,y,z)	符合 IEEE
__frcp_[rn,rz,ru,rd](x)	符合 IEEE
__fsqrt_[rn,rz,ru,rd](x)	符合 IEEE
__fdiv_[rn,rz,ru,rd](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 错误为 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 错误为 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 错误为 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

C.2.2 双精度浮点函数

编译器从未将_dadd_rn() 和 _dmul_rn() 映射的加法和乘法运算操作并入 FMAD 。与此相比，“*”和“+”运算符生成的加法和乘法将频繁并入 FMAD。

表 C-4. CUDA 运行时库支持的双精度浮点内部函数及其误差范围

函数	误差范围
__dadd_[rn,rz,ru,rd](x,y)	符合 IEEE
__dmul_[rn,rz,ru,rd](x,y)	符合 IEEE
__fma_[rn,rz,ru,rd](x,y,z)	符合 IEEE
__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
__ddiv_[rn,rz,ru,rd](x,y)(x,y)	符合 IEEE，要求计算能力不小于于 2.0
__drcp_[rn,rz,ru,rd](x)	符合 IEEE，要求计算能力不小于于 2.0
__dsqrt_[rn,rz,ru,rd](x)	符合 IEEE，要求计算能力不小于于 2.0
__double2float_[ru,rd](x)	N/A

C.2.3 整型函数

__[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。

__[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。

_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。
__brev(x) 反转 32 位无符号整数 x，如结果的第 N 位对应 x 的 31-N 位
__brevll(x)反转 64 位无符号 long long 整数 x，如结果的第 N 位对应 x 的 63-N 位

附录 D C++语言特性

在设备代码中，CUDA 支持下列的 C++语言构造：

- 多态
- 默认参数
- 运算符重载
- 命名空间
- 函数模板
- 计算能力 2.0 的设备支持类

这些 C++构造参考“The C++ Programing Langue"的指定实现。这些构造在.cu 文件中的主机，设备和内核函数中都可以使用。本编程指南中前面详细说明的任何限制如缺乏对递归的支持依旧有效。

下面的子节为各种构造提供了例子。

D.1 多态

通常，多态是指一种定义函数或操作符在不同的上下文中行为不同的能力。这也被称为函数（或运算符）重载。

用实践中的术语，这意味着在同一作为域内定义两个不同的函数只要它们有不同的函数签名。这意味着这两个函数要么有不同数量的参数或者不同类型的参数。当编译器在多个有关函数中选定任何一个作为实现时，就说符合函数签名。

由于隐式的转型，编译器可能有多个匹配的调用，此时应用 C++语言标准的匹配规则。实践中这意味着如果有多个匹配的话，编译器会寻找最接近的匹配。

例：下面的代码在 CUDA 中有效：

```
__device__ void f(float x)
{// do something with x
}
__device__ void f(int i)
{
// do something with i
}
__device__ void f(double x, double y)
{
// do something with x and y
}
```

D.2 默认参数

有了上一节描述的多态和函数签名匹配规则支持，这里为函数参数支持默认值就变得可能了。

例子：

```
__device__ void f(float x = 0.0f)
{
// do something with x
}
```

内核或其它的设备函数现在可以用两种方式调用这个版本的 f：

```
f();
// or
float x = /* some value */;
f(x);
```

默认参数只能作用于既定函数的后 n 个参数。

D.3 运算符重载

运算符重载允许程序员为新类型定义运算符。C++能够重载的运算符有：+、-、*、/、+=、&、[]等。

例子：下面的代码在 CUDA 中有效，实现了两个 uchar4 向量的+操作：

```
__device__ uchar4 operator+ (const uchar4& a, const uchar4& b) {
uchar4 r;
r.x = a.x + b.x;
  ...
return r;
}
```

新操作符可以像这样使用：

```
uchar4 a, b, c;
a = b = /* some initial value */;
c = a + b;
```

D.4 命名空间

C++中的命名空间允许建造一个可视空间层次。此命名空间内的所有符号能够不用任何附加语法在此命名空间内使用。

命名空间可用于解决名字冲突（两个不同的符号使用同一名字），这经常发生在使用来自不同源的多个函数库时。

例子：下面代码在两个分立的命名空间（“nvidia”和“other”）定义了两个“f()”函数。

```
namespace nvidia {
    __device__ void f(float x){
        /* do something with x */ ;
    }
}
namespace other {
    __device__ void f(float x){
        /* do something with x */ ;
    }
}
```

这函数可以通过全限定的名字在任何地方使用：

```
nvidia::f(0.5f);
```

一个命名空间内的符号可被导入另一个命名空间：

```
using namespace nvidia;
f(0.5f);
```

D.5 函数模板

函数模板是一种元编程形式，它允许编写数据类型无关的泛型函数。CUDA 支持 C++标准的全部扩展，包括以下概念：

- 隐式模板参数推导。
- 显式实例化。
- 模板特化。

例子：

```
template <T>
__device__ bool f(T x)
{ return /* some clever code that turns x into a bool here */ }
```

这个函数将将任务数据类型的 x 转化为 bool，只要函数体内的代码能够为变量 x 的实际类型编译。

f()可以用两种方式调用：

```
int x = 1;
bool result = f(x);
```

第一种调用依赖编译器隐式推断的 T 的正确函数类型的能力。在这个例子中，编译器推断 T 为 int 且实例化 f<int>(x)。

另一种类型的调用是通过显式实例化，如下：

```
bool result = f<double>(0.5);
```

函数模板可能特化成：

```
template <T>
__device__ bool f(T x)
{ return false; }
template <>
__device__ bool
f<int>(T x)
{ return true; }
```

这种情况 T 的实现展示了 int 类型特化为返回真，所有其它类型将会被更通用的模板捕捉且返回假。

完全的匹配规则（为隐式推断模板参数）和多态匹配函数应用 C++标准指定的规则。

D.6 类

为计算能力 2.0 和更高的设备编译的代码可以使用 C++类，只要没有虚函数（这个限制将会在未来的版本中去除）。

没有虚函数的类有两种常见用途：

- 小数据类型的聚集体。中，像像素（r,g,b,a）等数据类型，二维和三维点，向量等。
- 函数对象。因为设备函数指针不支持所以不能将函数作为模板参数。此时可以使用函数对象（看下面的代码）。

D.6.1 像素数据类型

下面的例子是为 RGBA 像素定义的数据类型，其每个通道是八位深度：

```
class PixelRGBA{
public:
__device__
PixelRGBA(): r_(0), g_(0), b_(0), a_(0)
{ ; }
__device__
PixelRGBA(unsigned char r, unsigned char g, unsigned char b, unsigned char a = 255):
r_(r), g_(g), b_(b), a_(a)
{ ; }
// other methods and operators left out for sake of brevity
private:
unsigned char r_, g_, b_, a_;
friend PixelRGBA operator+(const PixelRGBA &, const PixelRGBA &);
};
__device__
PixelRGBA operator+(const PixelRGBA & p1, const PixelRGBA & p2)
{
return PixelRGBA(p1.r_ + p2.r_, p1.g_ + p2.g_, p1.b_ + p2.b_, p1.a_ + p2.a_);
}
```

其它的设备代码现在可以使用这新类型，某人可能希望这样：

```
PixelRGBA p1, p2;
// [...] initialization of p1 and p2 here
PixelRGBA p3 = p1 + p2;
```

D.6.2 函数对象

下面的例子演示怎样将函数对象作为函数模板的参数以实现一系列向量算术运算。下面是为浮点加减法建立的两个函数对象：

```
class Add{
public:
    __device__
    float operator() (float a, float b)
    const {
        return a + b;
    }
};
class Sub
{
public:
    __device__
    float operator() (float a, float b)
    const
    {
        return a - b;
    }
};
```

下面的特化了的内核利用函数对象实现了浮点向量上的操作：

```
// Device code
template<class O>
__global__
void
    VectorOperation(const float * A, const float * B, float * C, unsigned int N, O op)
{
    unsigned int iElement = blockDim.x * blockIdx.x + threadIdx.x;
    if (iElement < N)
    {
        C[iElement] = op(A[iElement], B[iElement]);
    }
}
```

为了获得向量加法 VectorOperation 内核现在可以像这样发射：

```
// Host code
VectorOperation<<<blocks, threads>>>(v1, v2, v3, N, Add());
```

附录 E nvcc 特性

E.1 _noinline_

默认情况下，_device_ 函数总是内联的。_noinline_ 函数限定符暗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。

对于计算能力 1.x 的设备，如果函数有指针参数或者较大的参数列表，则编译器会否决 _noinline_ 限定符。

E.2 #pragma unroll

默认情况下，编译器将展开具有已知行程计数的小循环。#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。可选择性的在其后接一个数字，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。

E.3 __restrict__

nvcc 通过__restrict__关键字支持受限的（restricted）指针。

C99 中引入受限的指针是为了缓解 C 风格代码中的指针别名问题，别名阻碍了从代码重组到子表达式删除等各种优化。

下面是一个别名问题的例子，使用受限指针能够帮助编译器减少指令数：

```
void foo(const float* a, const float* b, float* c)
{
    c[0] = a[0] * b[0];
    c[1] = a[0] * b[0];
    c[2] = a[0] * b[0] * a[1];
    c[3] = a[0] * a[1];
    c[4] = a[0] * b[0];
    c[5] = b[0];
    ...
}
```

在 C 风格代码中，指针 a,b 和 c 可能有别名，所以任何通过 c 的写操作都可能修改 a 或 b 的元素。这意味着为了保证功能正确性，编译器不能将 a[0]和 b[0]载入寄存器相乘，然后将结果存入 c[0]和 c[1]，因为如果说 a[0]和 c[0]是同一个位置的话，从抽象执行模型来看结果可能不一样。因此编译器不能利用常见的子表达式。类似地，编译器不能只是重排 c[4]为 c[0]和 c[1]的计算，因为前面 C[3]的写入可能改变 C[4]的读入。

通过将 a,b 和 c 指定为受限的指针，程序员断言这些指针是没有别名的，这意味着对 c 写入不会重写 a 或 b 的元素。这将函数原型改成下面的样子：

```
void foo(const float* __restrict__ a, const float* __restrict__ b, float* __restrict__ c);
```

注意为了编译器优化所有的指针参数需要被声明为受限地。增加了__restrict__关键字，编译器现在能够任意重排和使用子表达式删除，同时保证功能一致。

```
void foo(const float* __restrict__ a, const float* __restrict__ b, float* __restrict__ c)
{
    float t0 = a[0];
    float t1 = b[0];
    float t2 = t0 * t2;
    float t3 = a[1];
    c[0] = t2;
    c[1] = t2;
    c[4] = t2;
    c[2] = t2 * t3;
    c[3] = t0 * t3;
    c[5] = t1;
...
}
```

这减少了访存次数和计算量。这同时也增加了寄存器压力，使用寄存器以缓存负载和公共子表达式。

因为对于很多 CUDA 代码来说，寄存器压力是一个重要的问题，使用受限指针产生的负作用就是可能减小了占用率，这可能降低性能。

附录 F 纹理获取

本附录提供了用于根据纹理参考的不同属性（参见 3.2.4 节）计算 B.8 节介绍的纹理函数的返回值的公式。

绑定到纹理参考的纹理表示为数组 T，对于一维纹理，它有 N 个纹理元素，或者针对二维纹理它有 $N \times M$ 个元素，或者针对三维纹理它有 $N \times M \times L$ 个元素。它将使用纹理坐标 x、y 和 z 获取。

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越界纹理坐标重新映射到有效范围内。如果 x 是非归一化的，则仅支持钳位寻址模式，如果 $x < 0$，则将 x 替换为 0，如果 $N < x$，则替换为 N-1。如果 x 是归一化的：

- 使用钳位寻址模式时，如果 x < 0，x 将替换为 0，如果 1<x，则替换为 1-1/N；
- 在循环寻址模式中，x 将被替换为 frac(x)，其中 frac(x) = x-floor(x)，floor(x) 是最大不大于 x 的整数。

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 xˆ 、yˆ 和 zˆ，即 x = Nxˆ、y = Myˆ、z = Lzˆ。

F.1 最近点取样

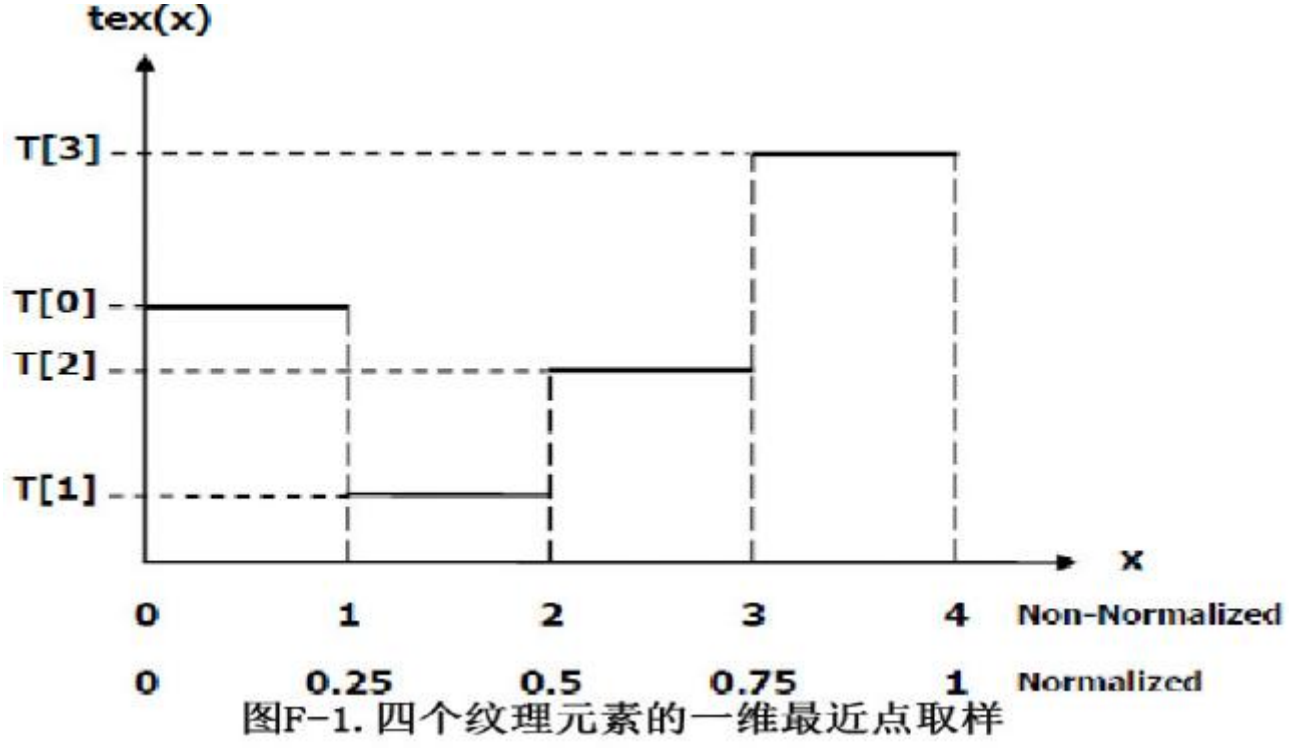
在这种过滤模式中，纹理获取返回的值如下：

- 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取样，N =4。

对于整型纹理来说，纹理获取的返回值可重新映射到 [0.0, 1.0]（参见 3.2.4.1 节）。



图F-1. 四个纹理元素的一维最近点取样

F.2 线性滤波

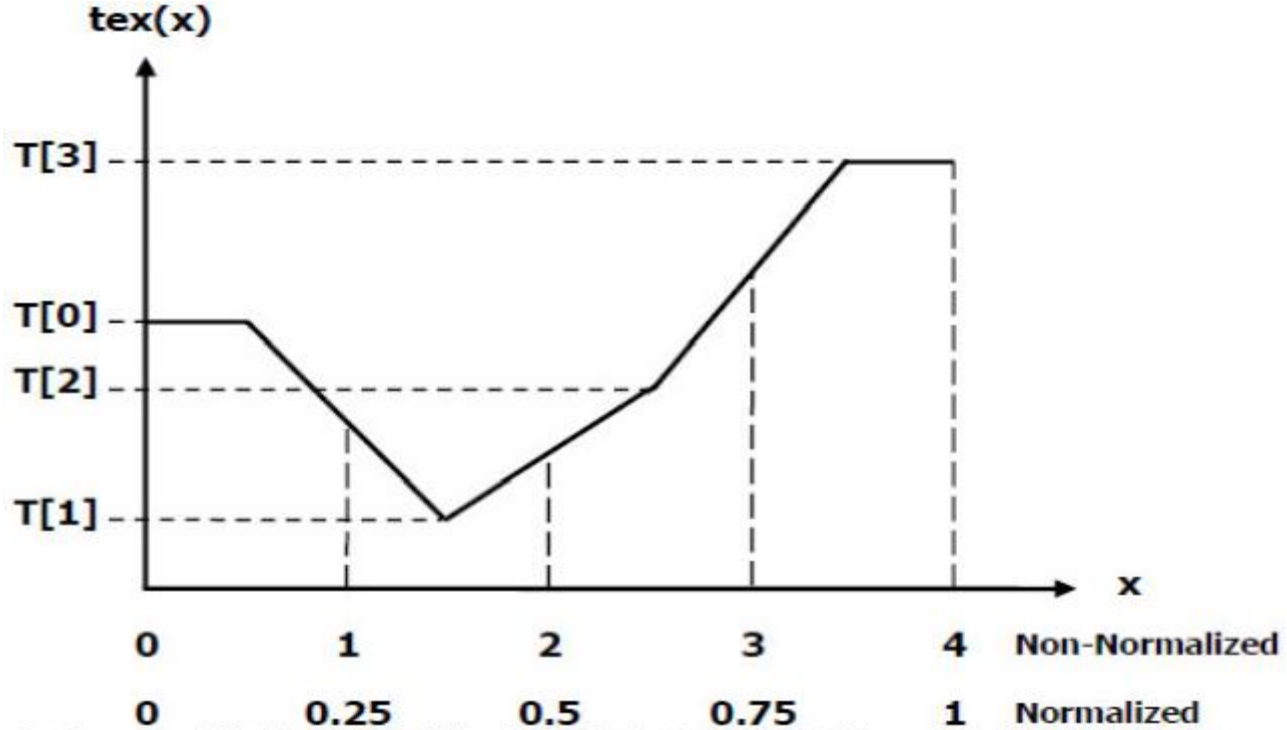
在这种仅对浮点纹理可用的过滤模式中，纹理获取的返回值如下：

- 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)βT[i,j+1]+α βT[i+1,j+1]
- 对于三维纹理是 tex(x,y,z) = (1−α)(1−β)(1−γ)T[i,j,k]+α(1−β)(1−γ)T[i+1,j,k]+(1−α)(1−β)γT[i,j,k+1]+α(1−β)γT[i+1,j,k+1]+(1−α)β γT[i,j+1,k+1]+α β γT[i+1,j+1,k+1]

其中：

- $i = floor(xB)$, $\alpha = frac(xB)$, $xB = x - 0.5$;
- $j = floor(yB)$, $\beta = frac(yB)$, $yB = y - 0.5$;
- $k = floor(zB)$, $\gamma = frac(zB)$, $zB = z - 0.5$。

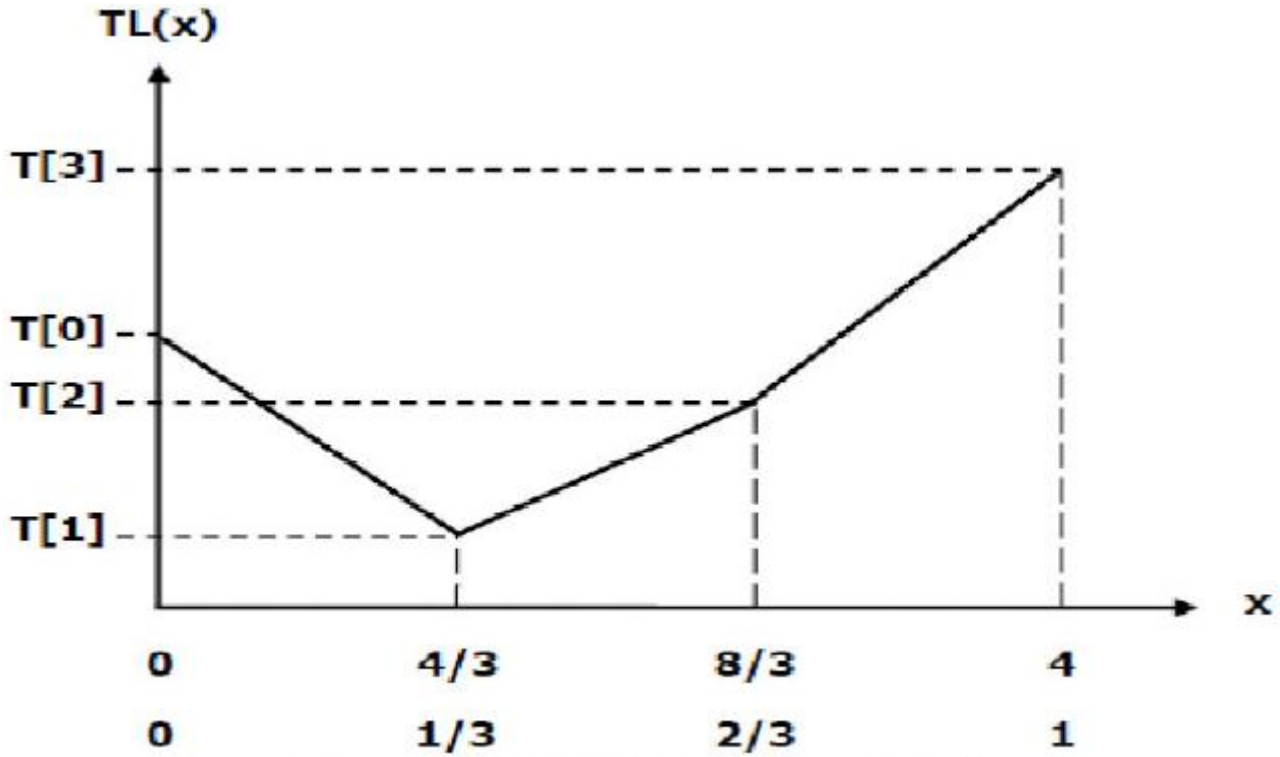
α、β和γ 存储在 9 位的定点格式中，有 8 位分数值（所以 1.0 是准确表示的）。
图 F-2 展示了一维纹理的线性取样，N = 4。



图F-2. 钳位寻址模式下四个元素的一维纹理线性滤波

F.3 查找表

查找表函数 TL(x) 可实现为 $TL(x) = tex(\frac{N-1}{R}x + 0.5)$，其中 x 处于 [0, R] 的间距内，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. 线性滤波下一维纹理查表

附录 G 计算能力

设备的计算能力决定了它大部分规范和特性（参见 2.5 节）
G.1 节给出了各计算能力设备的特性和技术规范。
G.2 节回顾了与 IEEE 浮点标准的符合程度。
G.3 节和 G.4 节分别给出了计算能力 1.x 和 2.0 的设备更多架构细节。

G.1 特性和技术规范

<table>
<tr><td></td><td colspan="5">计算能力</td></tr>
<tr><td>支持特性</td><td>1.0</td><td>1.1</td><td>1.2</td><td>1.3</td><td>2.0</td></tr>
<tr><td>作用在全局存储器上 32 位字的整形原子函数（参见 B.10 节）</td><td>否</td><td colspan="4">是</td></tr>
<tr><td>作用在全局存储器上 64 位字的整形原子函数（参见 B.10 节）</td><td colspan="2" rowspan="3">否</td><td colspan="3" rowspan="3">是</td></tr>
<tr><td>作用在共享存储器上 32 位字的整形原子函数（参见 B.10 节）</td></tr>
<tr><td>束表决函数（参见 B.11 节）</td></tr>
<tr><td>双精度浮点数</td><td colspan="3">否</td><td colspan="2">是</td></tr>
<tr><td>作用在全局和共享存储器上 32 位字的浮点原子加函数（参见 B.10 节）</td><td colspan="4" rowspan="4">否</td><td rowspan="4">是</td></tr>
<tr><td>__ballot()（参见 B.11）</td></tr>
<tr><td>__theadfence_system()（参见 B.5 节）</td></tr>
<tr><td>__syncthreads_count(),
__syncthreads_and(),
syncthreads_or()（参见 B.6）</td></tr>
<tr><td>网格的最大 x 或 y 维尺寸</td><td colspan="5">65535</td></tr>
<tr><td>块内最大线程数</td><td colspan="4">512</td><td>1024</td></tr>
<tr><td>块最大 x 或 y 维尺寸</td><td colspan="4">512</td><td>1024</td></tr>
<tr><td>块最大 z 维尺寸</td><td colspan="5">64</td></tr>
<tr><td>束尺寸</td><td colspan="5">32</td></tr>
<tr><td>多处理器最大常驻块数量</td><td colspan="5">8</td></tr>
<tr><td>多处理器最大常驻束数量</td><td>24</td><td colspan="2">32</td><td colspan="2">48</td></tr>
<tr><td>多处理器最大常驻线程数量</td><td>768</td><td colspan="2">1024</td><td colspan="2">1536</td></tr>
<tr><td>多处理器的 32 位寄存器数量</td><td>8K</td><td colspan="2">16K</td><td colspan="2">32K</td></tr>
<tr><td>多处理器最大共享存储器数量</td><td colspan="4">16KB</td><td>48KB</td></tr>
<tr><td>共享存储器存储体数量</td><td colspan="4">16</td><td>32</td></tr>
<tr><td>每线程的最大本地存储器数量</td><td colspan="4">16KB</td><td>512KB</td></tr>
<tr><td>常量存储器尺寸</td><td colspan="5">64KB</td></tr>
<tr><td>每个多处理器常量缓存数量</td><td colspan="5">8KB</td></tr>
<tr><td>多处理器纹理缓存数量</td><td colspan="5">设备相关，6KB 到 8KB</td></tr>
<tr><td>绑定到 CUDA 数组的一维纹理参考最大宽度</td><td colspan="4">8192</td><td>32768</td></tr>
<tr><td>绑定到线性纹理的一维纹理参考最大</td><td colspan="5">2^{27}</td></tr>
</table>

宽度		
绑定到线性存储器或 CUDA 数组的二维纹理参考的最大宽和高	65536*32768	65536*65536
绑定到线性存储器或 CUDA 数组的三维纹理参考的最大宽、高和深度	2048*2048*2048	4096*4096*4096
内核的最大指令数量	2 百万	

G.2 浮点标准

所有计算设备对于二进制浮点算术服从 IEEE 754-2008 标准，有下列偏差：

- 没有动态可配置的舍入模式，但是大多数操作支持多种 IEEE 舍入模式，这通过设备内置指令的方式展现；
- 没有检测浮点异常发生的机制且所有的异常操作行为像 IEEE-754 异常一样总是被屏蔽，并像 IEEE-754 定义的一样如果异常发生就传递屏蔽的响应；同样的原因，支持 SNaN 解码，它们并不通知且被无声的处理；
- 包含一个或多个 NaN 输入的单精度浮点操作的结果是 NaN，其位模式为 0x7fffffff；
- 对于 NaN，双精度浮点绝对值和求负不符合 IEEE-754 标准；它们的结果就是自身；
- 对于计算能力 1.x 的设备上的单精度浮点数：
  - 不支持非规格化数；浮点算术和比较指令在操作之前将非规格化操作数转化为 0；
  - 下溢的结果刷为 0；
  - 一些指令是不服从 IEEE 的：
    - 加法和乘法经常被组合成乘加指令（FMAD），它截取（也就是说没有舍入）乘法的中间尾数；
    - 除法以非标准的方式通过倒数实现；
    - 平方根以非标准的方式通过倒数平方根实现；
    - 对于加法和乘法，只有舍入到最近偶数和向零舍入通过静态舍入得到支持；不支持直接舍入到正负无穷；

  为了缓和这些限制，通过下列内置方式提供符合 IEEE 的软（因此更慢）实现（参见 C.2.1 节）：
    - __fmaf_r[n,z,u,d](float,float,float)：IEEE 舍入模式的单精度积和乘加，
    - __frcp_r[n,z,u,d](float)：IEEE 舍入模式的单精度倒数，
    - __fdiv_r[n,z,u,d](float,float)：IEEE 舍入模式的单精度除法，
    - __fsqrt_r[n,z,u,d](float)：IEEE 舍入模式的单精度平方根，
    - __fadd_r[n,z,u,d](float,float)：IEEE 直接舍入模式的单精度加法，
    - __fmul_r[u,d](float,float)：IEEE 直接舍入模式的单精度乘法

- 对于计算能力 1.x 的设备上的双精度浮点数：
  - 舍入到最近偶数是唯一支持倒数，除法和平方根的 IEEE 舍入模式。

当为没有本地双精度浮点支持的的设备上编译时，也就是说，计算能力 1.2 或更低的设备，每个双精度变量转化为单精度浮点格式（但依旧保持 64 位长度）且双精度算术降级为单精度算术。

对于计算能力 2.0 或更高的设备，代码必须使用-ftz=false，-prec-div=true 和-prec-sqrt=true 以保证符合 IEEE（这是默认设置，详细的编译选项参见 nvcc 用户手册）；使

用-ftz=true,-prec-div=false 和-prec-sqrt=false 编译的代码更接近为计算能力 1.x 设备生成的代码。

加法和乘法经常被组合成一个单独的乘加指令：
- 为计算能力 1.0 的设备生成单精度的 FMAD
- 为计算能力 2.0 的设备生成单精度的 FFMA

如上所示，FMAD 截取加法使用之前的尾数。另一方面，FFMA 是符合 IEEE-754(2008)的积和乘加指令，所以在加法中使用全宽度的乘积且在产生最终结果中有一次舍入。而 FFMA 相比 FMAD 通常有更好数值特性，从 FMAD 切换到 FFMA 在数值上结果上能产生极小的变化但是极少导致最终结果的大误差。

依据 IEEE-754R 标准，如果对 fminf(),fmin(),fmaxf()或 fmx()中之一而言，输入参数是 NaN，但是其它的不是，结果是非 NaN 参数。

IEEE-754 没有定义当将浮点数转化为整数时，值超出整数格式表示的范围时的行为。对于计算设备，其行为是将结果钳位到支持的范围的最后一个值，这和 x86 不一样。

G.3 计算能力 1.x

G.3.1 架构

对于计算能力 1.x，一个多处理器包含：
- 8 个 CUDA 核心用于整数和单精度浮点算术操作，
- 1 个双精度浮点操作单元用于双精度浮点算术操作（计算能力 1.3），
- 2 个特殊函数单元用于单精度浮点超越函数（这些单元同时也处理单精度浮点乘法），
- 1 个束调度器。

为了为束内所有线程执行一条指令，束调度器发射指令必须花费：
- 为整数和单精度浮点算术指令 4 个时钟周期，
- 为双精度浮点算术指令 32 个时钟周期，
- 为单精度超越指令 16 个时钟周期。

每个多处理器有一个被所有功能单元共享的只读的常量缓存，加速来自常量存储器空间的读操作，常量存储器在设备存储器中。

多处理器组成纹理处理集群（Texture Processor Cluster，TPC）。每个 TPC 中多处理器数目是：
- 对于计算能力 1.0 和 1.1 的设备是 2，
- 对于计算能力 1.2 和 1.3 的设备是 3.

每个 TPC 有一个被所有多处理器共享的只读纹理缓存，加速来自纹理存储器空间的读操作，纹理存储器在设备存储器中。每个多处理器通过纹理单元来访问纹理缓存，纹理单元实现了如 3.2.4 节提到的多种寻址模式和数据滤波。

全局存储器和本地存储器在设备存储器中且没有缓存。

G.3.2 全局存储器

来自一个束的一个全局存储器请求被分成两个存储器请求，每个对应半束，独立发射。G.3.2.1 和 G.3.2.2 节描述了半束线程的存储器访问如何被合并成一次或多次存储器事务，这

依赖于设备计算能力。图 G-1 显示了一些基于计算能力的全局存储器访问和对应存储器事务的例子。

最终的存储器事务贡献了存储器吞吐量。

G.3.2.1 计算能力 1.0 和 1.1 的设备

为了合并访问，半束的存储器请求必须满足下列条件：

- 线程读的字的长度必须是 4,8 或 16 字节，
- 如果长度是：
  - 4,所有的 16 个字必须在同一 64 字节段中，
  - 8，所有的 16 个字必须在同一 128 字节段中，
  - 16，前 8 个字必须在同一 128 字节段中，后 8 个字必须在随后的 128 字节段中；
- 线程必须顺序的访问字：半束中第 k 个线程访问第 k 个字。

如果半束满足这些要求，如果线程访问的字的长度分别为 4，8，16 字节，分别发射一个 64 字节存储器事务，一个 128 字节存储器事务或 2 个 128 字节存储器事务。即使束产生了分支也会合并，也就是说，一些活动线程并没有真正访问存储器。

如果半束不满足这些条件，16 个单独的 32 位存储器事务被发射。

G.3.2.2 计算能力 1.2 和 1.3 的设备

线程可以以任意顺序访问任意字，包括同一字，为半束寻址的每个段发射一次存储器事务。这和计算能力 1.0 和 1.1 的设备要求线程顺序访问字且只有半束寻址单一段时才能合并相反。

更准确地，使用下面的协议决定半束内线程必要的存储器事务：

- 找出最小编号活动线程寻址的存储器片段。段的长度由线程访问的字的长度决定：
  - 1 字节的字 32 字节
  - 2 字节的字 64 字节
  - 4，8，16 字节的字 128 字节
- 找出其它地址在同一段内的活动线程
- 减小事务长度，如果可能：
  - 如果事务是 128 字节且只有下半部分或上半部分被使用，减小事务到 64 字节；
  - 如果事务是 64 字节（原始的或者从 128 字节减小后的）且只有上半部分或下半部分被使用，减小事务到 32 字节。
- 执行事务且标记线程为非活动的。
- 重复直到半束内所有线程得到服务。

G.3.3 共享存储器

共享存储器被组织成 16 个存储体使得相邻 32 位字被分配到相邻的存储体。每个存储体每两个时钟周期有 32 位带宽。

束的一次共享存储器访问被分成再次存储器访问，每次为半束独立发射。因此前半束的线程和后半束的线程不可能出现存储体冲突。

如果束执行非原子指令为束内多个线程写共享存储器的同一位置，半束里只有一个线程执行写操作且那个线程执行最后的写操作没有定义。

G.3.3.1 32 位步长访问

每个线程以线程 ID tid 为索引，以 s 为步长从数组中访问一个 32 位字是一个常见的模式：

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

这种情况下，当 s*n 是存储体数的倍数时，线程 tid 和 tid+n 访问同一存储体，或等价地，当 n 是 16/d 的整数倍时，其中 d 是 16 和 n 的最小公约数。因此，只有半束长度小于等于 16/d 时没有存储体冲突，此时只有 d=1，也就是说 s 是奇数。

图 G-2 为计算能力 2.0 的设备展示的一些按步长访问的例子。这些例子对于计算能力 1.x 的设备同样有效，但是存储体数目是 16 而不是 32.

G.3.3.2 32 位广播访问

共享存储器有个特性是广播机制，因此当响应读请求的时候，一个 32 位字能够被读取并同时广播给多个线程。当多个线程读同一 32 位字地址时，减少了存储体冲突的数量。更精确地，由多个地址组成的一次存储体读请求的响应由多个步骤组成，每一个步骤响应一次没有冲突的访问，直至所有的请求被响应；在每一步骤，通过下列过程从剩下的地址中建立访问子集：

- 从尚未访问的地址所指向的字中，选择一个作为广播字；
- 子集包括
  - 在广播字内的所有地址，
  - 剩下地址指向的存储体中，每个存储体（不包括广播大存储体）的一个地址。

在每个周期内，那个字被选为广播字和为每个存储体选择的地址没有定义。

一个常见的没有存储体冲突的例子是当半束内所有线程从同一 32 位字的一个地址中读时。

图 G-3 展示了有关广播机制的一些存储体读访问的例子。这些例子对于计算能力 1.x 的设备同样有效，但是存储体数目是 16 而非 32.

G.3.3.3 8 位和 16 位访问

8 位和 16 位访问典型地会产生存储体冲突。例如，如果一个 char 数组以下面的方式访问就会有存储体冲突：

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

因为 shared[0],shared[1],shared[2]和 shared[3]在同一存储体中。如果以下面的方式访问，就没有存储体冲突：

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

G.3.3.4 大于 32 位访问

每个线程大于 32 位访问会被拆成 32 位的访问，这典型的会产生存储体冲突。
例如，如下的对 double 数组的访问会产生 2 路存储体冲突：

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

由于存储器请求被编译成 2 个独立的步长为 2 的 32 位访问。一种避免存储体冲突的方

式是将 double 操作数拆成两个，就像下面的代码一样：

```
__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]);
```

这可能并不问题能提高性能，在计算能力 2.0 的设备上这确实会降低性能。

对于结构体同样有效。如下面的代码：

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

导致：

- 三次独立的没有存储体冲突的访问，如果 type 定义为：

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

因为每个成员以奇数步长被访问，步长是 3 个 32 位字：

- 两个独立的有存储体冲突的访问，如果 type 定义为

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

因为每个成员以偶数步长被访问，步长是 2 个 32 位字。

G.4 计算能力 2.0

G.4.1 架构

对于计算能力 2.0 的设备，一个多处理器包含：

- 32 个 CUDA 核心用于整形和浮点算术操作，
- 4 个特殊函数单元用于单精度浮点超越函数，
- 2 个束调度器。

在每个指令发射时间，第一个束调度器为某个奇数 ID 的束发射一条指令且第二个束调度器为某个偶数 ID 的束发射一条指令。唯一的例外是一个束调度器发射一个双精度指令，此时，另一个调度器不能发射任何指令。

一个束调度器只能为 CUDA 核心的一半发射一条指令。为了为束内所有线程执行一条指令，束调度器必须发射指令超过：

- 为整数或浮点算术指令花费 2 个时钟周期，
- 为双精度浮点算术运算花费 4 个时钟周期，
- 为单精度超越指令花费 8 个时钟周期。

每个多处理器也有一个只读的一致的缓存，该缓存被所有功能单元共享，且能够加速从常量存储器空间的读取，常量存储器在设备存储器上。

每个多处理器有一级缓存，所有多处理器共享二级缓存，二者都用于缓存全局或本地的

访问，包括临时寄存器溢出。缓存行为（如读是缓存在 L1 和 L2 还是只缓存在 L2）能够基于访问使用读写指令修饰语部分配置。

同一片上存储器同时用于 L1 和共享存储器：可以配置为 48KB 的共享存储器和 16KB L1 缓存（默认设置）或者 16KB 共享存储器和 48KB L1 缓存，设置使用 cudaFuncSetCacheConfig()函数或 cuFuncSetCacheConfig()函数：

```
// Device code
__global__ void MyKernel() { ... }
 // Host code // Runtime API
// cudaFuncCachePreferShared: shared memory is 48 KB
 // cudaFuncCachePreferL1: shared memory is 16 KB
// cudaFuncCachePreferNone: no preference
cudaFuncSetCacheConfig(MyKernel, cudaFuncCachePreferShared)
// Driver API
// CU_FUNC_CACHE_PREFER_SHARED: shared memory is 48 KB
// CU_FUNC_CACHE_PREFER_L1: shared memory is 16 KB
// CU_FUNC_CACHE_PREFER_NONE: no preference
CUfunction myKernel;
cuFuncSetCacheConfig(myKernel, CU_FUNC_CACHE_PREFER_SHARED)
```

多处理器组成图形处理器单元（GPC）。每个 GPC 包含 4 个多处理器。

每个多处理器有一个只读纹理缓存以加速读取纹理存储器空间，纹理存储器在设备存储器上。通过纹理单元访问纹理缓存，纹理单元实现了多种寻址模式和数据滤波，这些在 3.2.4 节说明了。

G.4.2 全局存储器

全局存储器访问被缓存。使用编译器选项-dlcm 标签，在编译时配置是在 L1 和 L2 都缓存（-Xptxas -dlcm=ca）或只在 L2 中缓存（-Xptxas -dlcm=cg）。

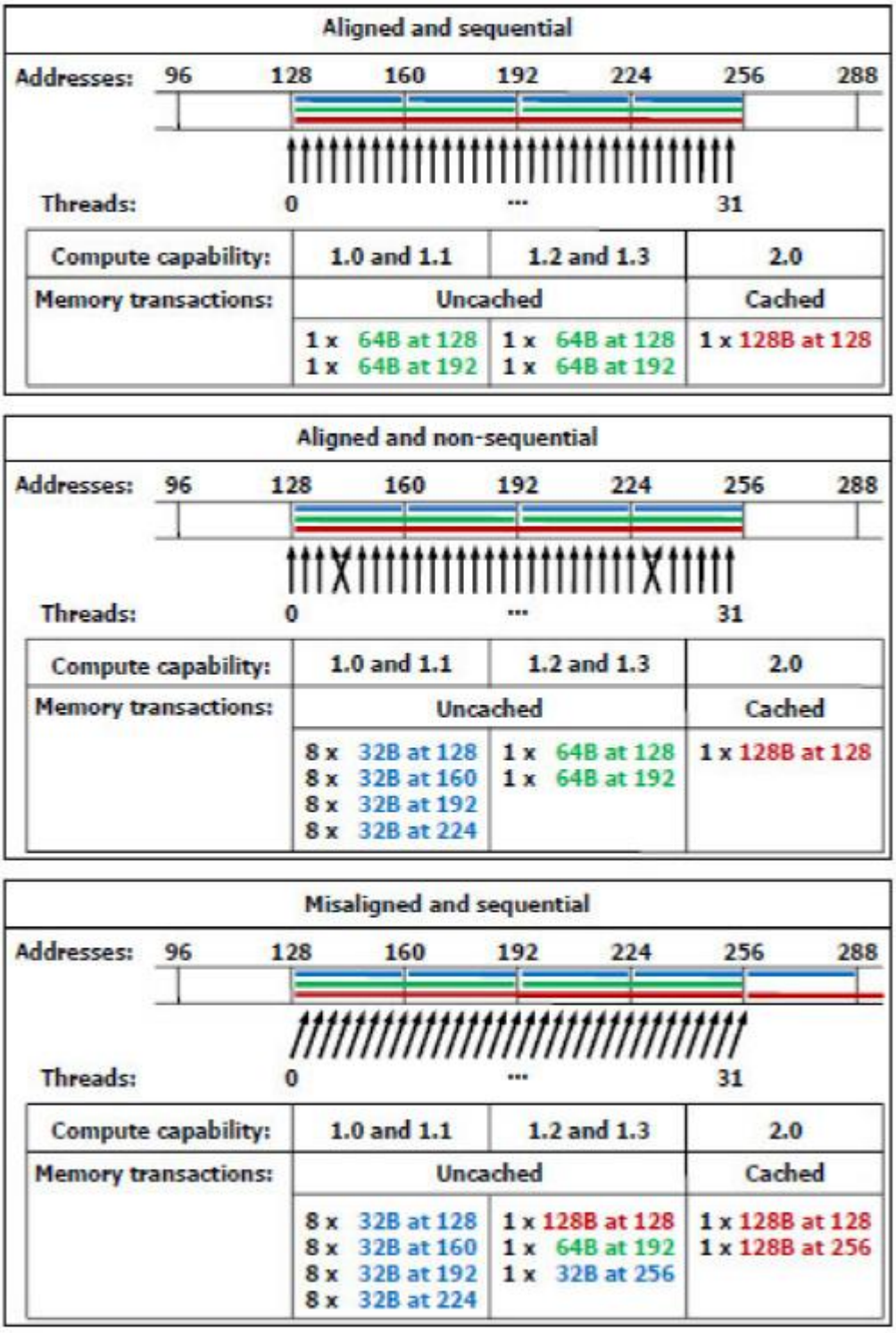
L1 或 L2 的缓存线是 128 字节且映射到设备存储器中一个 128 字节对齐的段。

如果每个线程访问的字的尺寸大于 4 字节，束的存储器访问首先被拆成独立的 128 字节的存储器请求独立发射：

- 如果尺寸是 8 字节，拆成两个请求，每个请求负责半束，
- 如果尺寸是 16 字节，拆成四个请求，每个请求负责四分之一束。

每个存储器请求分解成缓存线请求独立发射。如果缓存命中，请求的吞吐量就是 L1 或 L2 的吞吐量，否则吞吐量是设备存储器的吞吐量。

如果束执行非原子指令为束内多个线程写入全局存储器的同一位置，只有一个线程进行了写操作且那个线程执行没有定义。



图G-1. 束访问全局存储器例子，每线程4字节和基于计算能力的存储器事务

G.4.3 共享存储器

共享存储器有 32 个存储体，存储体的组织使得相邻的 32 位字被分配到相邻的存储体中。每个存储体带宽为每 2 个时钟周期 32 位字。因此不像低计算能力的设备，前半束内的线程和后半束的线程可能发生存储体冲突。

如果多个线程访问 32 位字的任何字节，如果多个字节属于同一存储体，就发生了存储体冲突。如果多个线程访问同一 32 位字的任何字节，不会发生存储体冲突：对于读，字会广播给请求线程（不像计算能力 1.x 的设备，多个字可在一次事务中广播）；对于写，每个字节只会读线程中的一个写（那个线程执行没有定义）。

特别地，这意味着不像计算能力 1.x 的设备，如下的方式访问 char 数组没有存储体冲突：

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

G.4.3.1 32 位步长访问

一个常见的访问模式是每个线程以线程 ID 作为索引，s 作为步长来访问来自数组的一个 32 位字：

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

在这个例子中，当 s*n 是存储体数量的倍数时，线程 tid 和 tid+n 访问同一存储体或者等价地，当 n 是 32/d 的倍数时，其中 d 是 32 和 s 的最大公约数。因此只有束尺寸小于等于 32/d，d 只有等于 1，也是说 s 是奇数。

图 G-2 展示某些步长访问的例子。

G.4.3.2 大于 32 位访问

64 位和 128 位访问 被特殊处理以最小化存储体冲突，细节如下。

其它大于 32 位的访问被拆成 32 位，64 位或 128 位访问。下面的代码：

```
struct type { float x, y, z; };
__shared__ struct type shared[32];
struct type data = shared[BaseIndex + tid];
```

导致三个独立的没有存储体冲突的 32 位读，因为每个成员以三个 32 位字为步长被访问。

64 位访问

对于 64 位访问，存储体冲突只发生在半束中的两个或多个线程访问同一存储体的不同地址。

不像计算能力 1.x 的设备，像下面的方式访问 double 数组不会产生存储体冲突：

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

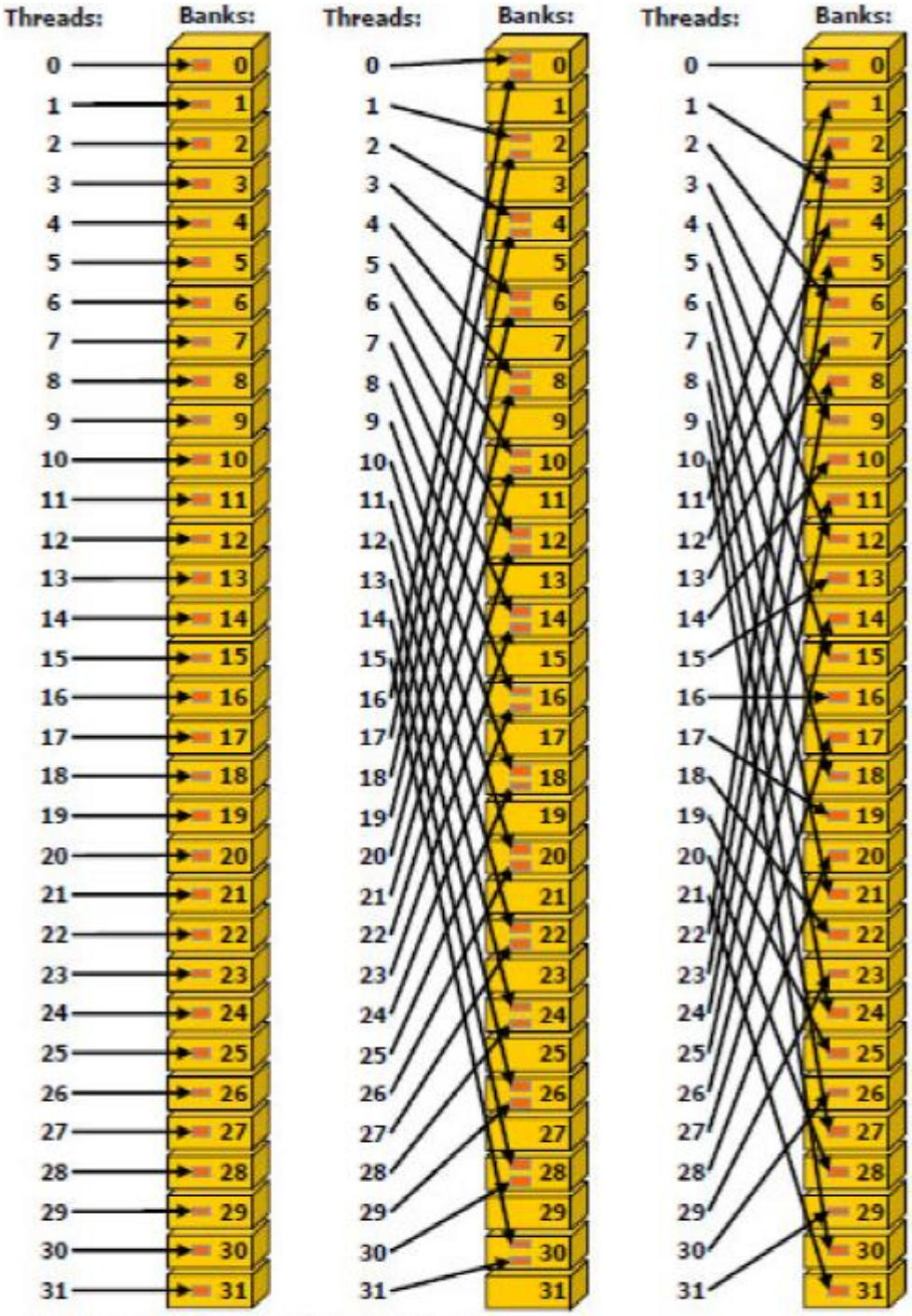
128 位访问

大多数 128 位访问会引起 2 路存储体冲突，即使四分这一束没有两个线程访问同一存储体中的不同地址。因此为了确定存储体冲突的数目，必须加 1 到四分之一束中属于同一存储体的访问不同地址的数种数。

G.4.4 常量存储器

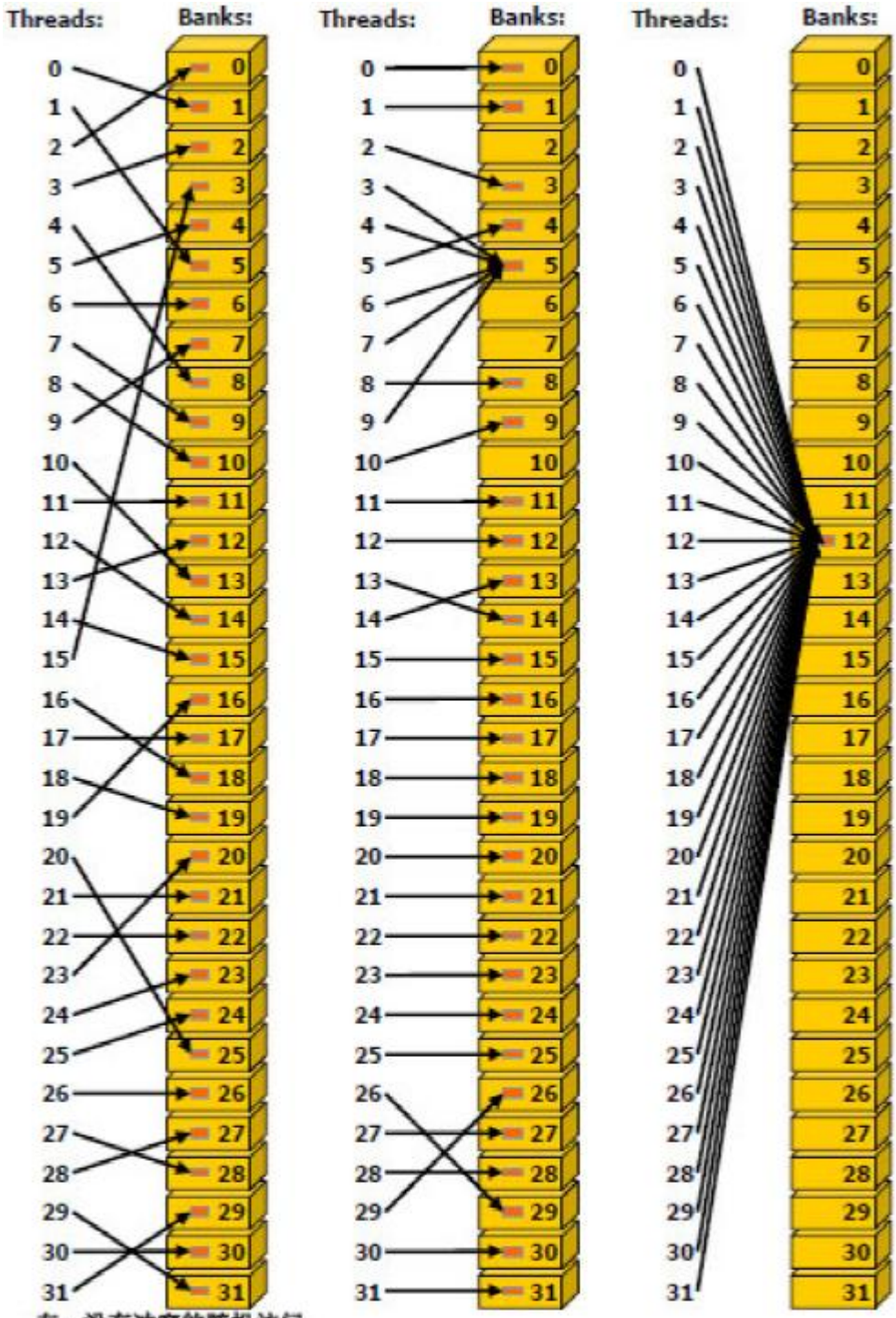
除了常量存储器空间得到所有计算能力设备的支持外（__constant__声明的变量存储位置），计算能力 2.0 的设备支持 LDU 指令，编译器使用 LDU 指令装载变量：

- 指向全局存储器，
- 在内核中只读（程序员可以使用 const 关键字保证这一点），
- 不依赖线程 ID.



左：步长为32位字的线性访存（没有冲突）
中：步长为2个32位字的线性访存（2路冲突）
右：步长为3个32位字的线性访存（没有冲突）

图G-2. 计算能力2.0共享存储器步长访问



左：没有冲突的随机访问
中：没有冲突，即使多个线程访问存储体5
右：没有冲突的广播（所有线程访问同一个字）

图G-3. 不常见的共享存储器访问

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 专利或专利权的许可。本文档提供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"