NVIDIA CUDA
统一计算设备架构
编程指南

Version 1.1

11/29/2007

目　录

第 1 章　CUDA 简介 1

1.1　作为数据并行计算设备的图形处理器 1

1.2　CUDA：一种 GPU 计算的新架构 3

1.3　文档结构 6

第 2 章 编程模型 7

2.1　高度多线程协处理器 7

2.2　线程分批 7

2.2.1　线程块 7

2.2.2　线程块网格 8

2.3　内存模型 10

第 3 章　硬件实现 13

3.1　具有片上共享内存的一组 SIMD 多处理器 13

3.2　执行模型 14

3.3　计算能力 15

3.4　多个设备 16

3.5　显示模式切换 16

第 4 章　应用编程接口 17

4.1　C 编程语言扩展 17

4.2　语言扩展 17

4.2.1　函数类型限定符 18

4.2.2　变量类型限定符 19

4.2.3　执行配置 21

4.2.4　内置变量 21

4.2.5　使用 NVCC 编译 22

4.3　共用运行时组件 23

4.3.1　内置向量类型 23

4.3.2　数学函数 24

4.3.3　时间函数 24

4.3.4　纹理类型 24

4.4　设备运行时组件 26

4.4.1　数学函数 26

4.4.2 同步函数 26
4.4.3 类型转换函数 26
4.4.4 类型强制函数 27
4.4.5 纹理函数 27
4.4.6 原子函数 28
4.5 宿主运行时组件 28
4.5.1 常用概念 29
4.5.2 运行时 API 32
4.5.3 驱动程序 API 39
第 5 章 性能指南 47
5.1 指令性能 47
5.1.1 指令吞吐量 47
5.1.2 内存带宽 49
5.2 每块的线程数 62
5.3 宿主和设备之间的数据传送 63
5.4 纹理拾取与全局或常量内存读取 63
5.5 整体性能优化策略 64
第 6 章 矩阵乘法示例 67
6.1 概述 67
6.2 源码清单 69
6.3 源码攻略 71
6.3.1 `Mul()` 71
6.3.2 `Muld()` 71
附录 A 技术规格 73
A.1 通用规范 74
A.2 浮点标准 74
附录 B 数学函数 77
B.1 共用运行时组件 77
B.2 设备运行时组件 80
附录 C 原子函数 83
C.1 算术函数 83
C.1.1 `atomicAdd()` 83
C.1.2 `atomicSub()` 83
C.1.3 `atomicExch()` 83

C.1.4 atomicMin() 84
C.1.5 atomicMax() 84
C.1.6 atomicInc() 84
C.1.7 atomicDec() 84
C.1.8 atomicCAS() 84
C.2 位函数 85
C.2.1 atomicAnd() 85
C.2.2 atomicOr() 85
C.2.3 atomicXor() 85
附录 D 运行时 API 参考 87
D.1 设备管理 87
D.1.1 cudaGetDeviceCount() 87
D.1.2 cudaSetDevice() 87
D.1.3 cudaGetDevice() 87
D.1.4 cudaGetDeviceProperties() 88
D.1.5 cudaChooseDevice() 89
D.2 线程管理 89
D.2.1 cudaThreadSynchronize() 89
D.2.2 cudaThreadExit() 89
D.3 流管理 89
D.3.1 cudaStreamCreate() 89
D.3.2 cudaStreamQuery() 89
D.3.3 cudaStreamSynchronize() 89
D.3.4 cudaStreamDestroy() 89
D.4 事件管理 90
D.4.1 cudaEventCreate() 90
D.4.2 cudaEventRecord() 90
D.4.3 cudaEventQuery() 90
D.4.4 cudaEventSynchronize() 90
D.4.5 cudaEventDestroy() 90
D.4.6 cudaEventElapsedTime() 90
D.5 内存管理 91
D.5.1 cudaMalloc() 91
D.5.2 cudaMallocPitch() 91

D.5.3 cudaFree() 91
D.5.4 cudaMallocArray() 92
D.5.5 cudaFreeArray() 92
D.5.6 cudaMallocHost() 92
D.5.7 cudaFreeHost() 92
D.5.8 cudaMemset() 92
D.5.9 cudaMemset2D() 92
D.5.10 cudaMemcpy() 93
D.5.11 cudaMemcpy2D() 93
D.5.12 cudaMemcpyToArray() 94
D.5.13 cudaMemcpy2DToArray() 94
D.5.14 cudaMemcpyFromArray() 95
D.5.15 cudaMemcpy2DFromArray() 95
D.5.16 cudaMemcpyArrayToArray() 96
D.5.17 cudaMemcpy2DArrayToArray() 96
D.5.18 cudaMemcpyToSymbol() 96
D.5.19 cudaMemcpyFromSymbol() 96
D.5.20 cudaGetSymbolAddress() 97
D.5.21 cudaGetSymbolSize() 97
D.6 纹理参考管理 97
D.6.1 低层 API 97
D.6.2 高层 API 98
D.7 执行控制 100
D.7.1 cudaConfigureCall() 100
D.7.2 cudaLaunch() 100
D.7.3 cudaSetupArgument() 100
D.8 OpenGL 互操作性 100
D.8.1 cudaGLRegisterBufferObject() 100
D.8.2 cudaGLMapBufferObject() 101
D.8.3 cudaGLUnmapBufferObject() 101
D.8.4 cudaGLUnregisterBufferObject() 101
D.9 Direct3D 互操作性 101
D.9.1 cudaD3D9Begin() 101
D.9.2 cudaD3D9End() 101

D.9.3 cudaD3D9RegisterVertexBuffer() 101
D.9.4 cudaD3D9MapVertexBuffer() 101
D.9.5 cudaD3D9UnmapVertexBuffer() 102
D.9.6 cudaD3D9UnregisterVertexBuffer() 102
D.9.7 cudaD3D9GetDevice() 102
D.10 错误处理 102
D.10.1 cudaGetLastError() 102
D.10.2 cudaGetErrorString() 102
附录 E 驱动程序 API 参考 103
E.1 初始化 103
E.1.1 cuInit() 103
E.2 设备管理 103
E.2.1 cuDeviceGetCount() 103
E.2.2 cuDeviceGet() 103
E.2.3 cuDeviceGetName() 103
E.2.4 cuDeviceTotalMem() 104
E.2.5 cuDeviceComputeCapability() 104
E.2.6 cuDeviceGetAttribute() 104
E.2.7 cuDeviceGetProperties() 105
E.3 上下文管理 106
E.3.1 cuCtxCreate() 106
E.3.2 cuCtxAttach() 106
E.3.3 cuCtxDetach() 106
E.3.4 cuCtxGetDevice() 106
E.3.5 cuCtxSynchronize() 106
E.4 模块管理 106
E.4.1 cuModuleLoad() 106
E.4.2 cuModuleLoadData() 107
E.4.3 cuModuleLoadFatBinary() 107
E.4.4 cuModuleUnload() 107
E.4.5 cuModuleGetFunction() 107
E.4.6 cuModuleGetGlobal() 107
E.4.7 cuModuleGetTexRef() 108
E.5 流管理 108

E.5.1 cuStreamCreate() 108
E.5.2 cuStreamQuery() 108
E.5.3 cuStreamSynchronize() 108
E.5.4 cuStreamDestroy() 108
E.6 事件管理 108
E.6.1 cuEventCreate() 108
E.6.2 cuEventRecord() 108
E.6.3 cuEventQuery() 109
E.6.4 cuEventSynchronize() 109
E.6.5 cuEventDestroy() 109
E.6.6 cuEventElapsedTime() 109
E.7 执行控制 109
E.7.1 cuFuncSetBlockShape() 109
E.7.2 cuFuncSetSharedSize() 110
E.7.3 cuParamSetSize() 110
E.7.4 cuParamSeti() 110
E.7.5 cuParamSetf() 110
E.7.6 cuParamSetv() 110
E.7.7 cuParamSetTexRef() 110
E.7.8 cuLaunch() 110
E.7.9 cuLaunchGrid() 111
E.8 内存管理 111
E.8.1 cuMemGetInfo() 111
E.8.2 cuMemAlloc() 111
E.8.3 cuMemAllocPitch() 111
E.8.4 cuMemFree() 112
E.8.5 cuMemAllocHost() 112
E.8.6 cuMemFreeHost() 112
E.8.7 cuMemGetAddressRange() 112
E.8.8 cuArrayCreate() 113
E.8.9 cuArrayGetDescriptor() 114
E.8.10 cuArrayDestroy() 114
E.8.11 cuMemset() 114
E.8.12 cuMemset2D() 114

E.8.13 cuMemcpyHtoD() 115
E.8.14 cuMemcpyDtoH() 115
E.8.15 cuMemcpyDtoD() 115
E.8.16 cuMemcpyDtoA() 116
E.8.17 cuMemcpyAtoD() 116
E.8.18 cuMemcpyAtoH() 116
E.8.19 cuMemcpyHtoA() 116
E.8.20 cuMemcpyAtoA() 117
E.8.21 cuMemcpy2D() 117
E.9 纹理参考管理 119
E.9.1 cuTexRefCreate() 119
E.9.2 cuTexRefDestroy() 119
E.9.3 cuTexRefSetArray() 119
E.9.4 cuTexRefSetAddress() 120
E.9.5 cuTexRefSetFormat() 120
E.9.6 cuTexRefSetAddressMode() 120
E.9.7 cuTexRefSetFilterMode() 120
E.9.8 cuTexRefSetFlags() 121
E.9.9 cuTexRefGetAddress() 121
E.9.10 cuTexRefGetArray() 121
E.9.11 cuTexRefGetAddressMode() 121
E.9.12 cuTexRefGetFilterMode() 121
E.9.13 cuTexRefGetFormat() 122
E.9.14 cuTexRefGetFlags() 122
E.10 OpenGL 互操作性 122
E.10.1 cuGLInit() 122
E.10.2 cuGLRegisterBufferObject() 122
E.10.3 cuGLMapBufferObject() 122
E.10.4 cuGLUnmapBufferObject() 122
E.10.5 cuGLUnregisterBufferObject() 123
E.11 Direct3D 互操作性 123
E.11.1 cuD3D9Begin() 123
E.11.2 cuD3D9End() 123
E.11.3 cuD3D9RegisterVertexBuffer() 123

E.11.4 cuD3D9MapVertexBuffer() 123
E.11.5 cuD3D9UnmapVertexBuffer() 123
E.11.6 cuD3D9UnregisterVertexBuffer() 124
E.11.7 cuD3D9GetDevice() 124
附录 F 纹理拾取 125
F.1 最近点采样 126
F.2 线性过滤 127
F.3 查找表 128

图

图 1-1. CPU 和 GPU 的每秒浮点运算次数 1
图 1-2. GPU 将更多晶体管用于数据处理 2
图 1-3. 统一计算设备架构软件堆栈 3
图 1-4. *聚集*和*散布*存储器操作 4
图 1-5. 共享内存让数据更接近 ALU 5
图 2-1. 线程分批 9
图 2-2. 内存模型 11
图 3-1. 硬件模型 14
图 5-1. 已合并全局内存访问模式的示例 52
图 5-2. 未合并全局内存访问模式的示例 53
图 5-3. 未合并全局内存访问模式的示例 54
图 5-4. 无存储体冲突的共享内存访问模式示例 58
图 5-5. 无存储体冲突的共享内存访问模式示例 59
图 5-6. 有存储体冲突的共享内存访问模式示例 60
图 5-7. 有广播的共享内存读取访问模式示例 61
图 6-1. 矩阵乘法 68

第 1 章　CUDA 简介

1.1　作为数据并行计算设备的图形处理器

从图 1-1 可以看到，经过短短几年的发展，可编程图形处理器（GPU）就具备了超高的计算能力。在多个核心和高存储器带宽的配合下，最新的 GPU 成为了图形和非图形处理的超强工具。

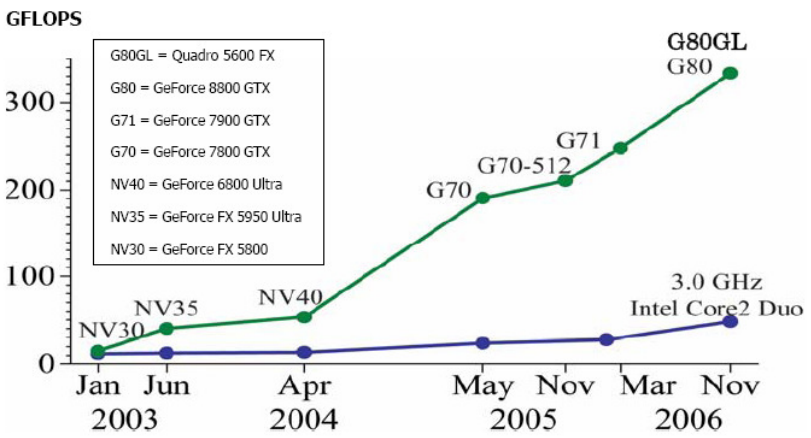


图 1-1. CPU 和 GPU 的每秒浮点运算次数

带来这种发展的主要原因是 GPU 专为计算密集型和高度并行的计算设计（而这正是图形渲染所需要的），因此 GPU 将更多的晶体管专用于数据处理，而非数据高速缓存（cache）和流控制（flow control），如图 1-2 所示。

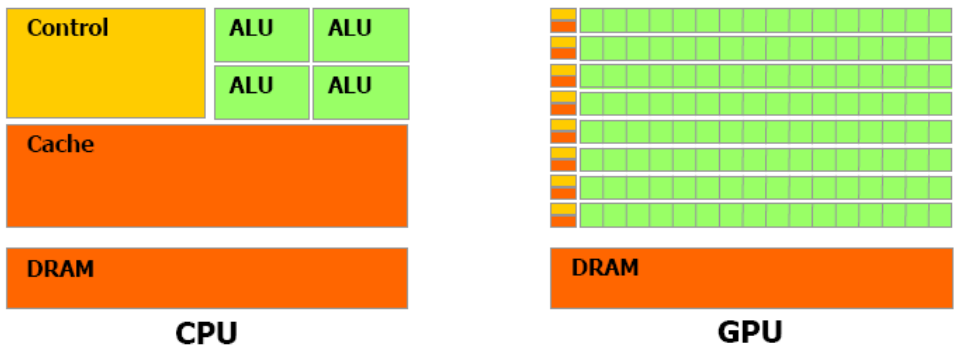


图 1-2. GPU 将更多晶体管用于数据处理

具体来说，GPU 很适合于解决数据并行（同一程序在许多数据元素上并行执行）的高运算密度（算术运算与存储器操作的比例）计算问题。因为在每个数据元素上执行同一程序，所以对复杂流控制的要求较低；又因为具有高运算密度的同一程序在大量数据元素上执行，也就可以通过大量的计算而非大量的数据高速缓存来隐藏访存延迟。

数据并行处理将数据元素映射到并行处理线程。处理大型数据集合（比如数组）的许多应用程序可以使用数据并行编程模型来加速计算。在 3D 渲染中，大量的像素和顶点集合被映射到并行线程。同样地，图像和媒体处理应用程序，比如渲染图像的后期处理、视频编码和解码、图像缩放、立体视觉、模式识别，可以将图像块和像素映射到并行处理线程。事实上，图像渲染和处理以外的许多计算方法也能通过数据并行处理来加速，其范围涵盖从一般的信号处理或物理模拟，到计算金融学或计算生物学等的诸多领域。

但是，在此之前，获得 GPU 中的所有计算能力并将其有效用于非图形应用程序中仍然不是件容易的事：

- GPU 只能通过图形 API 进行编程，从而把较高的学习曲线强加给初学者，并且图形 API 一般不太适用于非图形应用程序。
- GPU DRAM 可以使用一般方式来读取（即 GPU 程序可以从 DRAM 的任何部分*聚集*数据元素），但不能使用一般方式来写入（即 GPU 程序不能将消息*散布*到 DRAM 的任何部分），与 CPU 相比，其编程灵活性大大受限。
- 一些应用程序的瓶颈是 DRAM 存储器带宽，这就不能未充分利用 GPU 的计算能力。

本文档描述了一种创新性的硬件和编程模型，用以直接解决这些问题，并展示了 GPU 作为一种真正的通用数据并行计算设备的能力。

1.2 CUDA：一种 GPU 计算的新架构

CUDA（Compute Unified Device Architecture，统一计算设备架构），是一种新型的硬件和软件架构，它将 GPU 视为数据并行计算设备，在其上进行计算的分配和管理，而无需将其映射到图形 API。它可用于 GeForce 8 系列、Tesla 解决方案和一些 Quadro 解决方案（详细信息请参阅附录 A）。操作系统的多任务机制负责管理多个并发运行的 CUDA 应用程序和图形应用程序对 GPU 的访问。

CUDA 软件堆栈由几层组成，如图 1-3 所示：硬件驱动程序，应用编程接口（API）及其运行时（runtime），以及两个更高层的通用数学库 CUFFT 和 CUBLAS（这两个库会在单独的文档中介绍）。在硬件设计上，驱动程序层和运行时层是轻量级的，这样更能够达到高性能。

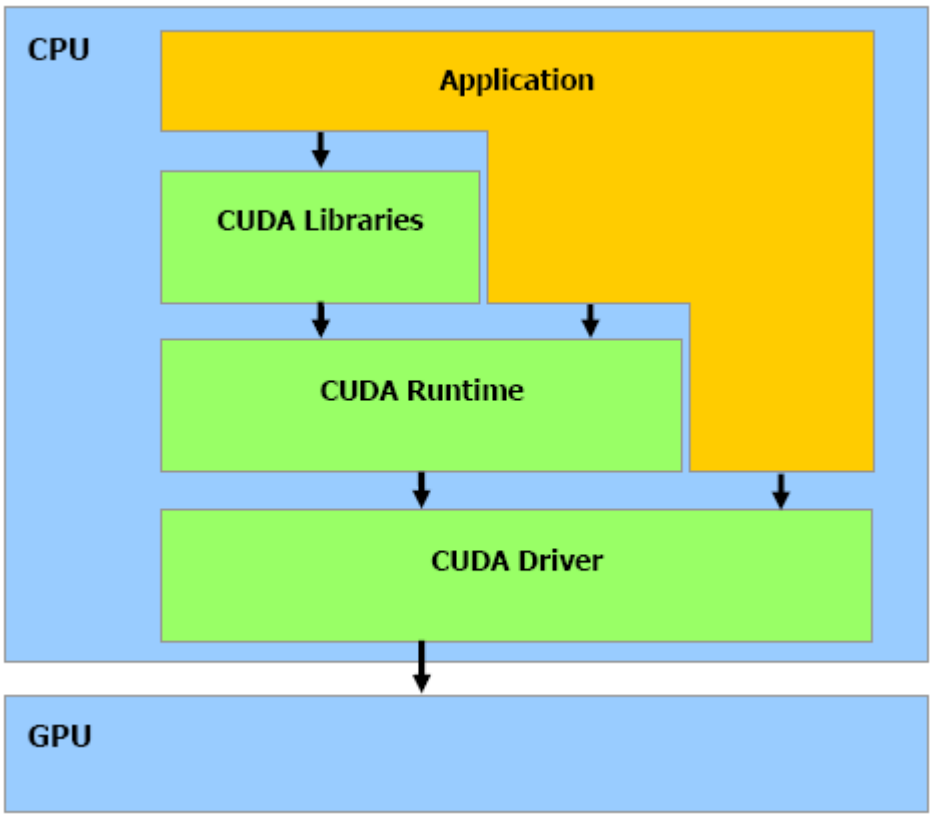


图 1-3. 统一计算设备架构软件堆栈

以给编程人员提供尽量低的学习曲线为目的，CUDA API 的语法由 C 语言语法扩展而来（参见第 4 章）。

如图 1-4 所示，CUDA 提供了通用的 DRAM 存储器寻址方式以实现更高的编程灵活性，这就是被称为聚集（gather）和散布（scatter）的存储器操作。从编程角度看，它们就是在 DRAM 的任何位置读取和写入数据的能力，与传统的 CPU 编程一样。

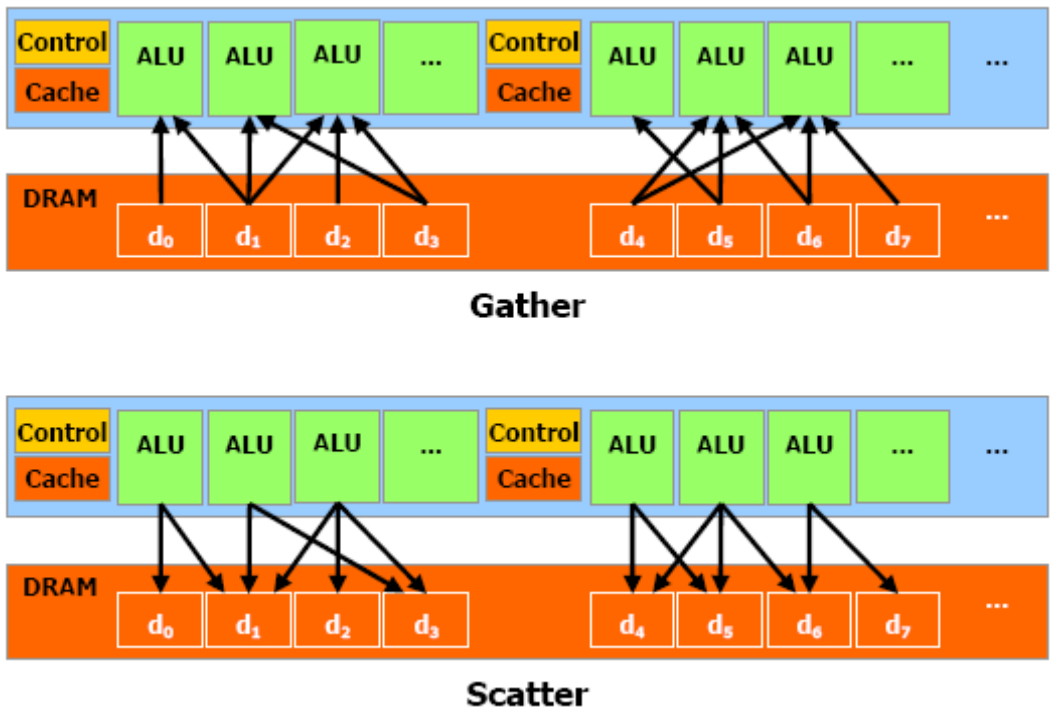


图 1-4. *聚集*和*散布*存储器操作

CUDA 提供了极高读写速度的并行数据高速缓存或者称其为片上共享内存（shared memory），线程可以使用它来互相共享数据（参见第 3 章）。如图 1-5 所示，应用程序可以利用它来最小化对 DRAM 的过取（overfetch）和轮询（round-trips），从而降低对 DRAM 内存带宽的依赖程度。

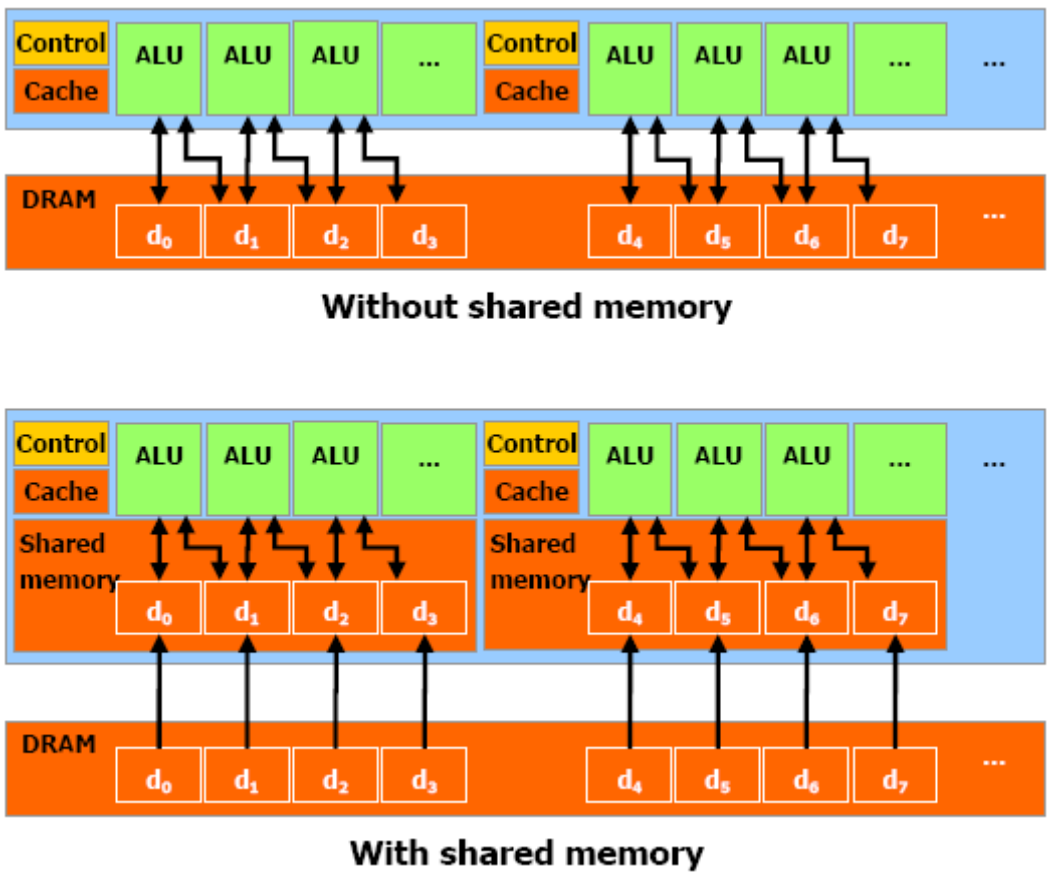


图 1-5. 共享内存让数据更接近 ALU

1.3 文档结构

本文档分为下列几章：

- 第 1 章包含对 CUDA 的一般性介绍。
- 第 2 章概述编程模型。
- 第 3 章介绍硬件实现。
- 第 4 章介绍 CUDA API 和运行时。
- 第 5 章提供一些有关如何达到最大性能的指南。
- 第 6 章通过分析一些简单示例的代码来进一步说明前几章的内容。
- 附录 A 给出各种设备的技术规格。
- 附录 B 列出 CUDA 支持的数学函数。
- 附录 C 列出 CUDA 支持的原子函数。
- 附录 D 是 CUDA 运行时 API 参考。
- 附录 E 是 CUDA 驱动程序 API 参考。
- 附录 F 给出有关纹理拾取一些细节。

第 2 章 编程模型

2.1 高度多线程协处理器

通过 CUDA 编程时，将 GPU 看作可以并行执行非常多个线程的*计算设备(compute device)*。它作为主 CPU（或被称为*宿主，host*）的协处理器运作。换句话说，在宿主上运行的应用程序中，数据并行的、计算密集型的部分被装载到计算设备上。

更准确地说，应用程序中，多次但在不同数据上独立执行的部分可以被独立放到在此设备上，用许多不同的线程去执行。要达到这种效果，可以将这样一个函数编译到设备的指令集中，并将编译后的程序（被称为*内核, kernel*）装载到设备上。

宿主和设备都保留自己的 DRAM，分别称为*宿主内存(host memory)*和*设备内存(device memory)*。用户可以通过优化的 API 调用将数据从一个 DRAM 复制到另一个 DRAM 中，该过程使用了设备的高性能直接内存访问（DMA）引擎。

2.2 线程分批

执行内核的一批线程组成线程块（block），再由线程块组成网格（grid），如 2.2.1 和 2.2.2 所述，并参见图 2-1。

2.2.1 线程块

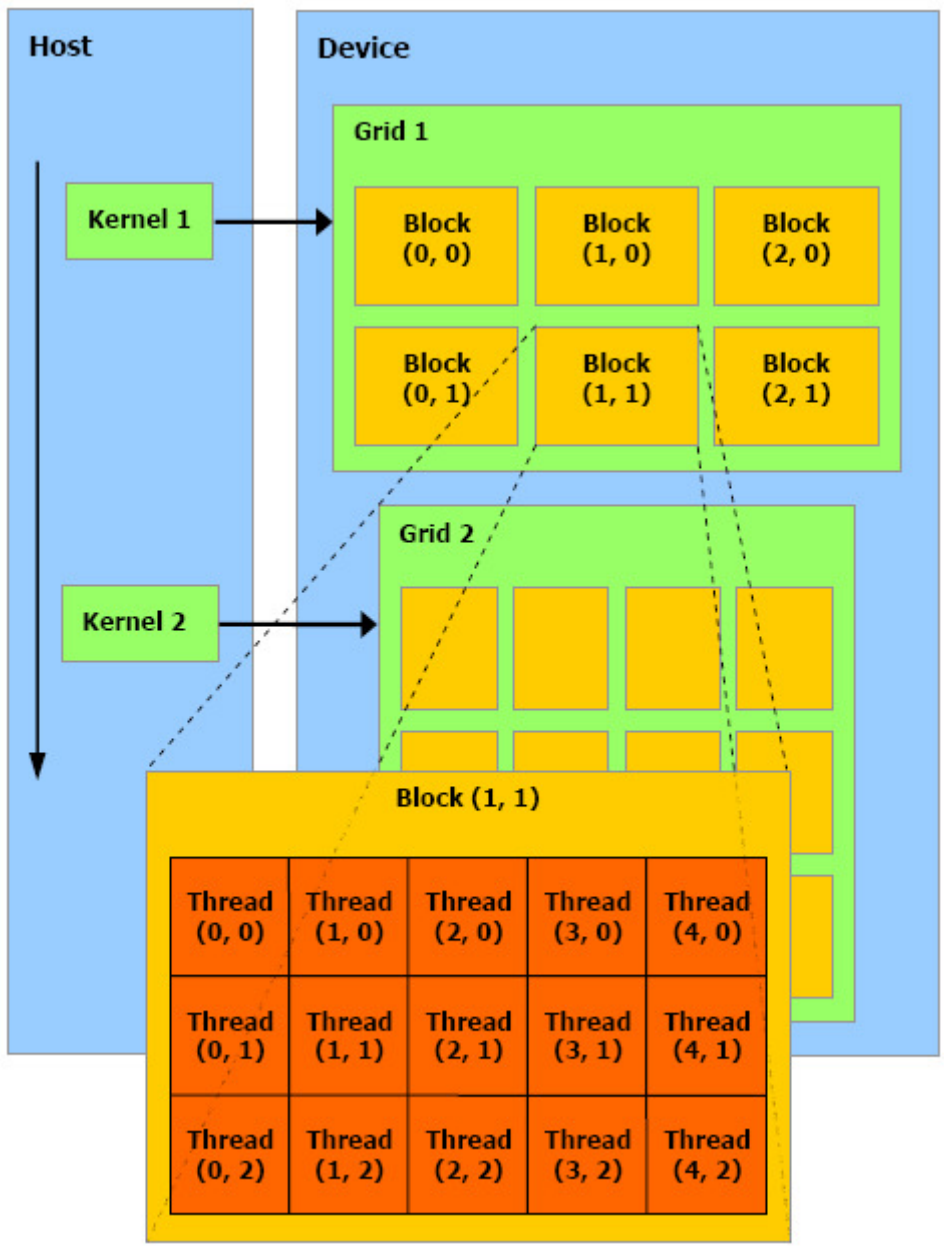
线程块是可以协同工作的一批线程，它们通过高速共享内存有效地共享数据，并同步其执行以协调访存。更准确地说，用户可以在内核中指定同步点，块中的线程在全部到达此同步点时挂起。

每个线程由*线程 ID(thread ID)*标识，这是块中的线程号。为了帮助基于线程 ID 的复杂寻址，应用程序还可以将块指定为任意大小的二维或三维数组，并使用 2 个或 3 个索引来标识每个线程。对于大小为(D_x,D_y)的二维块，索引为(x,y)的线程的线程 ID 为$(x+yD_x)$，对于大小为(D_x,D_y,D_z)的三维块，索引为(x,y,z)的线程的线程 ID 为$(x+yD_x+zD_xD_y)$。

2.2.2 线程块网格

块可以包含的最大线程数是有限制的。但是，执行相同内核的，具有相同维度和大小的块可以分批组合到一个块网格中，这样单个内核调用中启动的线程总数就可以变得很大。但这是以线程协作性的降低为代价的，因为同一网格中不同线程块中的线程不能互相通信和同步。此模型允许内核有效运行，而不必在具有不同并行能力的各种设备上重新编译：如果设备具有非常低的并行能力，则可以顺序运行网格的所有块，如果具有很高的并行能力，则可以并行运行网格的所有块，或者是多数情况下的二者组合使用。

每个块由*块 ID(block ID)*标识，这是网格中的块号。为了帮助基于块 ID 的复杂寻址，应用程序可以将网格指定为任意大小的二维数组，并使用 2 个索引来标识每个块。对于大小为(D_x,D_y)的二维块，索引为(x,y)的块的块 ID 为$(x+yD_x)$。



宿主执行一连串对设备的内核调用。每个内核作为一批线程来执行，若干个线程组成线程块，再由线程块组成网格。

图 2-1. 线程分批

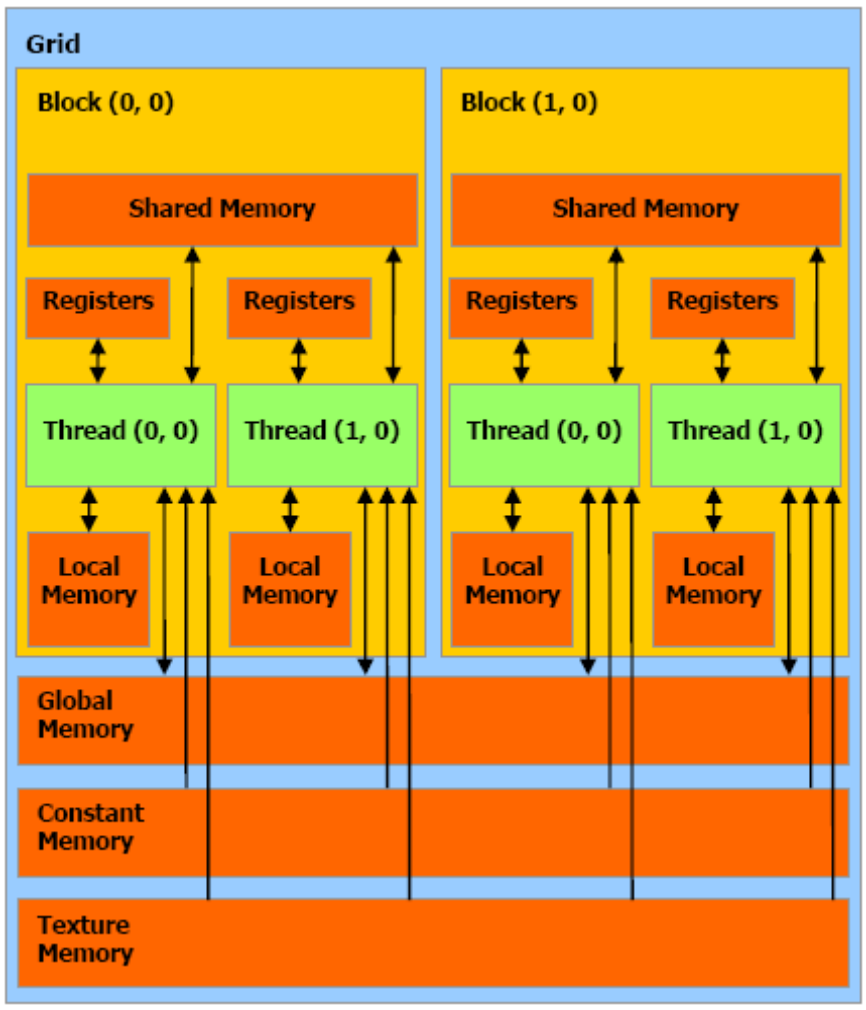
2.3 内存模型

在设备上执行的线程只能访问设备的 DRAM 和片上存储器，可访问的内存空间如下，示意图见图 2-2：

- 可读写每线程*寄存器*，
- 可读写每线程*本地内存*，
- 可读写每块*共享内存*，
- 可读写每网格*全局内存*，
- 只读每网格*常量内存*，
- 只读每网格*纹理内存*。

全局、常量和纹理内存空间可以通过宿主读或写，并可被相同应用程序的内核持续访问。

全局、常量和纹理内存空间对不同的内存使用方式进行了优化（参见 5.1.2.1、5.1.2.2 和 5.1.2.3）。纹理内存还为一些特定的数据格式提供不同的寻址模式和纹理过滤模式（参见 4.3.4）。



线程可以通过不同范围的一组内存空间来访问设备的 DRAM 和片上存储器。

图 2-2. 内存模型

第 3 章　硬件实现

3.1　具有片上共享内存的一组 SIMD 多处理器

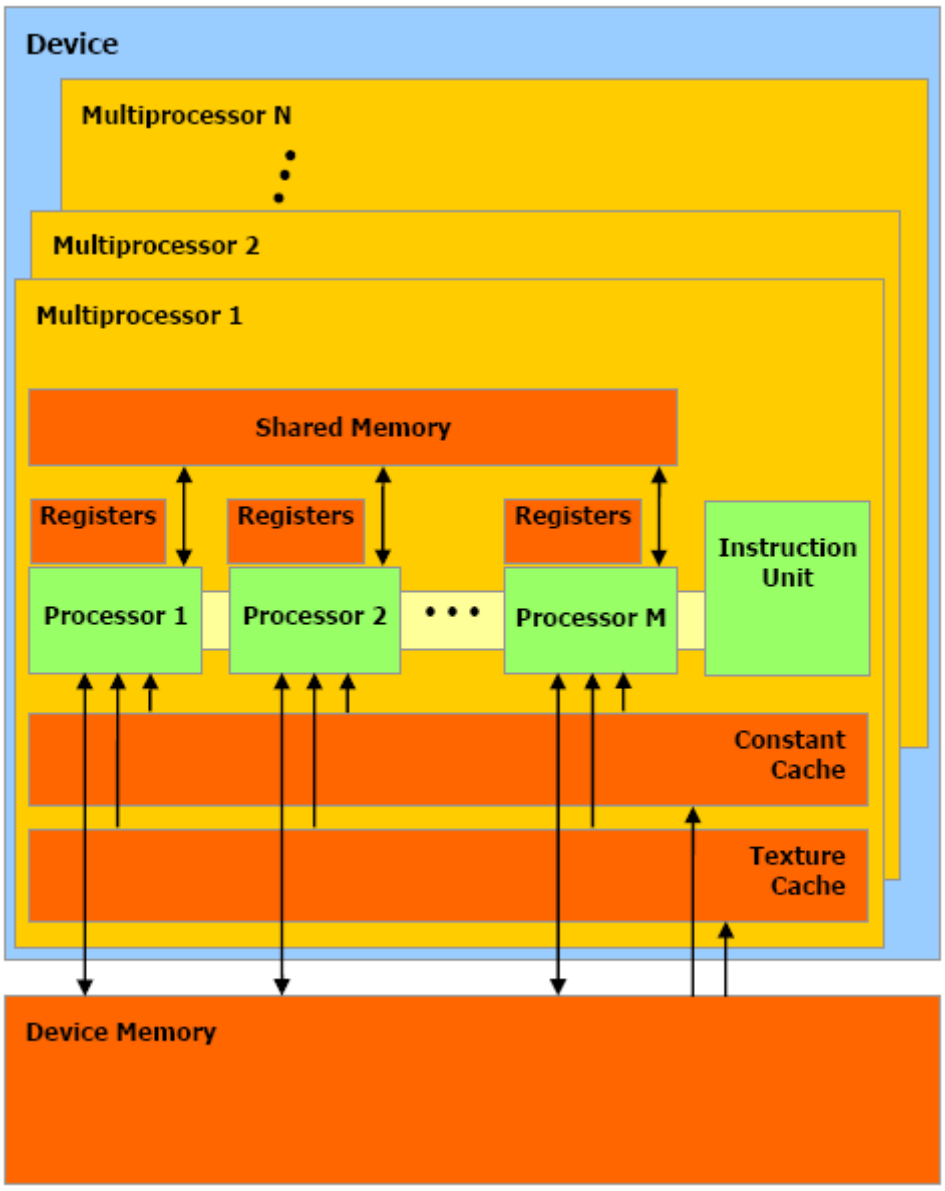
设备作为一组*多处理器(multiprocessors)*来实现，如图 3-1 所示。每个多处理器具有单指令多数据(SIMD)架构：在任何给定的时钟周期，多处理器中的每个处理器执行相同的指令，但操作不同的数据。

每个多处理器具有下列四种类型的片上存储器：

- 每个处理器有一组本地 32 位*寄存器*，
- 并行数据高速缓存或称为*共享内存(shared memory)*，由所有处理器共享并实现共享内存空间，
- 只读*常量高速缓存(constant cache)*，由所有处理器共享并加速从常量内存空间的读取，常量内存空间为设备内存的只读区域，
- 只读*纹理高速缓存(texture cache)*，由所有处理器共享并加速从纹理内存空间的读取，纹理内存空间为设备内存的只读区域。

本地和全局内存空间为设备内存的可读写区域，且无高速缓存。

每个多处理器通过*纹理单元(texture unit)*访问纹理高速缓存，其中纹理单元实现 2.3 一节提到的各种寻址模式和纹理过滤模式。



具有片上共享内存的一组 SIMD 多处理器。

图 3-1. 硬件模型

3.2 执行模型

线程块网格是通过调度块在多处理器上执行来在设备上执行的。每个多处理器一批接一批地处理块批次。一个块仅在一个多处理器内处理，所以存在于片上共享内存中的共享内存空间能够提供极高的访存速度。

因为多处理器上的寄存器和共享内存由块批次的所有线程瓜分，所以每个多处理器一批可以处理多少个块取决于给定内核内的每线程需要多少寄存器以及每块需要多少共享内存。如果某多处理器没有足够的可用寄存器或共享内存来处理至少一个块，则内核将无法将块分配给该多处理器执行。

在一个批次内并被一个多处理器处理的块被称为*活动(active)*块。每个活动块划分到被称为 *warp* 的 SIMD 线程组中：其中每个 warp 包含相同数量的线程，该数量被称为 *warp 大小*，并以 SIMD 方式由多处理器执行。活动 warp（比如所有活动块中的所有 warp）是分时的：*线程调度器（thread scheduler）*定期从一个 warp 切换到另一个 warp，以便最大化地利用多处理器的计算资源。*半 warp（half-warp）*是一个 warp 的第一半或第二半。

块划分为 warp 的方式始终相同；每个 warp 包含线程 ID 连续递增的线程，其中第一个 warp 从线程 0 开始递增。2.2.1 节介绍了线程 ID 与块中的线程索引如何相关联。

块中 warp 的发射（issue）顺序是不确定的，但其执行可以同步以协调全局或共享内存访问，如 2.2.1 所述。

线程块网格中块的发射顺序也是不确定的，且块之间没有同步机制，所以在网格执行期间，来自同一网格的两个不同块中的线程无法通过全局内存安全地互相通信。

如果某 warp 执行了非原子（non-atomic）指令对全局或共享内存中的相同位置进行写入，则此位置发生的串行化写入次数及其发生顺序是不确定的，但会保证其中一个写入成功。如果由 warp 里的多个线程执行原子（atomic）指令（参见 4.4.6）读取、修改并写入全局内存中的相同位置，则对此位置的每个读取、修改和写入都会发生，且全部都是串行化发生的，但发生的顺序是不确定的。

3.3 计算能力

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

具有相同主要修订号的设备具有相同的核心架构。附录 A 中列出的设备都是具有计算能力 1.x 的（其主要修订号为 1）。

次要修订号与核心架构的增量改进相对应，其中可能包括新功能。

各种计算能力的技术规格在附录 A 中给出。

3.4 多个设备

使用多个 GPU 作为 CUDA 设备的情况下，仅当这些 GPU 的类型相同时，应用程序才能保证工作。但是，如果系统处于 SLI 模式，则只有一个 GPU 可以用作 CUDA 设备，因为在驱动程序堆栈的最低层，所有 GPU 都熔合在一起。需要在控制面板中关闭 SLI 模式，CUDA 才能将每个 GPU 看作单独的设备。

3.5 显示模式切换

GPU 将一些 DRAM 内存分配给所谓的*主表面（primary surface）*，该主表面用于刷新用户当前查看的显示设备。当用户通过更改显示分辨率或位深（使用 NVIDIA 控制面板或 Windows 中的显示控制面板）启动显示*模式切换（mode switch）*时，主表面所需的内存量将随之变化。例如，如果用户将显示分辨率从 1280x1024x32-位更改为 1600x1200x32-位，系统必须为主表面分配 7.68MB 内存而非 5.24MB。（启用了反锯齿的全屏图形应用程序需要为主表面分配更多的显示内存。）在 Windows 中，可以启动显示模式切换的其它事件包括启动全屏 DirectX 应用程序、按下 Alt+Tab 从全屏 DirectX 应用程序切出或按下 Ctrl+Alt+Del 锁定计算机。

如果模式切换导致了主表面所需内存量的增加，系统可能不得不抽调本已指定给 CUDA 应用程序的内存分配给主表面使用，从而导致 CUDA 应用程序的崩溃。

第 4 章　应用编程接口

4.1　C 编程语言扩展

CUDA 编程接口的设计目标是为熟悉 C 编程语言的用户提供相对较低的学习曲线，以便更容易地编写在设备上执行的程序。

它包括：

- C 语言的最小扩展集合，如 4.2 所述，允许编程人员更关注于业务代码而非语言；
- 运行时库划分为：
  - 宿主组件，如 4.5 所述，在宿主上运行，提供函数以控制并访问宿主中的一个或多个计算设备；
  - 设备组件，如 4.4 所述，在设备上运行，并提供专用于设备的函数；
  - 共用组件，如 4.3 所述，提供内置的向量类型，以及宿主和设备代码中都支持的 C 标准库的一个子集。

必须强调的是，只有支持在设备上运行的 C 标准库中的函数才是共用组件提供的函数。

4.2　语言扩展

C 编程语言的扩展有四个部分：

- 函数类型限定符，用于指定函数是在宿主上还是在设备上执行，以及可以从宿主中还是设备中调用（参见 4.2.1）；
- 变量类型限定符，用于指定变量在设备上的内存位置（参见 4.2.2）；

- 新指令，用于指定来自宿主的内核如何在设备上执行（参见 4.2.3）；
- 四个内置变量，用于指定网格和块维度，以及块和线程索引（参见 4.2.4）。

包含这些扩展的每个源文件必须使用 CUDA 编译器 **nvcc** 编译，4.2.5 中有简要介绍。**nvcc** 的详细介绍可以参见单独的文档。

以上每个扩展都有一些限定条件，这些限定条件会在下文各节中描述。违反这些限制时，nvcc 将给出错误或警告，但一些违规无法被检测到。

4.2.1 函数类型限定符

4.2.1.1 `__device__`

`__device__`限定符声明某函数：

- 在设备上执行，
- 只能从设备中调用。

4.2.1.2 `__global__`

`__global__`限定符将某函数声明为内核。这种函数：

- 在设备上执行，
- 只能从宿主中调用。

4.2.1.3 `__host__`

`__host__`限定符声明某函数：

- 在宿主上执行，
- 只能从宿主中调用。

仅使用`__host__`限定符声明某函数，等价于不使用`__host__`、`__device__`或`__global__`限定符中的任何一个声明该函数；在这两种情况下，该函数仅为宿主编译。

但是，`__host__`限定符还可以与`__device__`限定符结合使用，此时，函数同时为宿主和设备编译。

4.2.1.4 限定条件

`__device__`和`__global__`函数不支持递归。

`__device__`和`__global__`函数不能在函数体内声明静态变量。

`__device__`和`__global__`函数不能具有可变个参数。

`__device__`函数不能取其地址；相反，`__global__`函数的函数指针则受支持。

`__global__`和`__host__`限定符不能结合使用。

`__global__`函数必须具有 `void` 返回类型。

对__global__函数的任何调用必须指定其执行配置，如 4.2.3 所述。

对__global__函数的调用是异步的，这意味着在设备完成其执行之前返回。

__global__函数参数当前通过共享内存传递给设备，并限制为 256 字节。

4.2.2 变量类型限定符

4.2.2.1 __device__

__device__限定符声明驻留在设备上的变量。

下面三节定义的其它类型限定符中，至多一个可以与__device__结合使用，以进一步指定变量属于哪个内存空间。如果不使用其中任何一个，则变量：

- 驻留在全局内存空间中，
- 具有应用程序的生命期，
- 可通过运行时库被网格的所有线程访问，也可以被宿主访问。

4.2.2.2 __constant__

__constant__限定符可以与__device__结合使用，声明变量：

- 驻留在常量内存空间中，
- 具有应用程序的生命期，
- 可通过运行时库被网格的所有线程访问，也可以被宿主访问。

4.2.2.3 __shared__

__shared__限定符可以与__device__结合使用，声明变量：

- 驻留在线程块的共享内存空间中，
- 具有块的生命期，
- 仅可被块内的所有线程访问。

线程内对共享变量的调用具有完全的顺序一致性，但多个线程间对该变量的调用顺序不严格一致。仅在__syncthreads()（参见 4.4.2）执行之后，来自其它线程的写入才能保证可见。除非变量被声明为不稳定的，否则只要满足__syncthreads()指令，编译器就可以优化对共享内存的读写。

将共享内存中的变量声明为外部数组，比如

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

数组大小在初始化时被确定（参见 4.2.3）。以此方式声明的所有变量在内存中的起始地址相同，所以必须通过偏移量显式地管理数组中变量的布局。例如，如果想要在动态分配的共享内存中建立

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

则可以使用下列方式声明和初始化数组：

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

4.2.2.4 限定条件

这些限定符不允许被用于 struct 和 union 成员、形参以及在宿主上执行的函数本地变量。

__shared__和__constant__变量含有默认的静态存储。

__device__、__shared__和__constant__变量不能使用 extern 关键字定义为外部变量。

__device__和__constant__变量仅在文件范围内生效。

__constant__变量不能从设备中赋值，只能在宿主中通过宿主运行时函数赋值（参见 4.5.2.3 和 4.5.3.6）。

__shared__变量不能在声明时进行初始化。

在设备代码中声明的、不带有其中任何一个限定符的自动变量一般驻留在寄存器中。但是，在一些情况下，编译器可能选择将其放置在本地内存中。这通常发生在将耗费太多寄存器空间的大型结构或数组，以及编译器无法确定其长度是否定长的数组身上。*ptx* 汇编代码（通过使用-ptx 或-keep 编译获得）能显示变量是否已经在第一个编译阶段被放置在本地内存中，此时此变量将使用.local 助记符声明并使用 ld.local 和 st.local 助记符访问。如果在第一个编译阶段，某变量没被放置在本地内存中，则后续编译阶段一旦发现该变量在目标架构中耗费了太多寄存器空间，仍可能将其放置到本地内存中。编译时可以通过--ptxas-options=-v 选项来检查本地内存使用（lmem）情况。

只要编译器能够解析在设备上执行的代码中的指针是指向共享内存空间还是全局内存空间，就支持这些指针，否则，就限制这些指针只能在全局内存空间中分配或声明。

在宿主上执行的代码中，试图析取指向全局或共享内存的指针将导致不确定的行为发生；同样，在设备上执行的代码中，试图析取指向宿主内存的指针也将导致不确定的行为发生。这通常会带来分段错误和应用程序终止。

仅能用在设备代码中获得__device__、__shared__或__constant__变量的地址。也仅能在宿主代码中通过 cudaGetSymbolAddress() 获得__device__或__constant__变量的地址，参见 4.5.2.3。

4.2.3 执行配置

对__global__函数的任何调用必须为此调用指定*执行配置（execution configuration）*。

执行配置用于定义在设备上执行函数的网格和块的维度，以及相关联的流（有关流的介绍，参见 4.5.1.5）。通过在函数名称和圆括号括起的参数列表之间插入<<< Dg, Db, Ns, S >>>形式的表达式，来定义执行配置，其中：

- Dg 的类型是 dim3（参见 4.3.1.2），用于指定网格的维度和大小，因此 Dg.x*Dg.y 等于要启动的块数；Dg.z 未使用；
- Db 的类型是 dim3（参见 4.3.1.2），用于指定每块的维度和大小，因此 Dg.x*Dg.y*Db.z 等于每块的线程数；
- Ns 的类型是 size_t，用于指定为此调用按块动态分配的共享内存中的字节数以及静态分配的内存；此动态分配的内存由声明为外部数组的任何一个变量使用，如 4.2.2.3 所述；Ns 是默认值为 0 的可选参数；
- S 的类型是 cudaStream_t，用于指定相关联的流；S 是默认值为 0 的可选参数。

例如，函数声明为

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

必须使用下列方式调用：

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

执行配置的参数在函数实参之前求值，并且与函数参数一样，直接通过共享内存传递给设备。

如果 Dg 或 Db 大于附录 A.1 中指定的设备允许的最大大小，或者如果 Ns 大于设备上可用的最大共享内存量减去静态分配、函数参数和执行配置所需的共享内存量，则函数调用将失败。

4.2.4 内置变量

4.2.4.1 gridDim

此变量标识网格的维度，类型为 dim3（参见 4.3.1.2）。

4.2.4.2 blockIdx

此变量标识网格中的块索引，类型为 uint3（参见 4.3.1.1）。

4.2.4.3 blockDim

此变量标识块的维度，类型为 dim3（参见 4.3.1.2）。

4.2.4.4 threadIdx

此变量标识块中的线程索引，类型为 uint3（参见 4.3.1.1）。

4.2.4.5 限定条件

- 不允许提取任何内置变量的地址。
- 不允许为任何内置变量赋值。

4.2.5 使用 NVCC 编译

nvcc 是用于简化 CUDA 代码编译过程的编译器驱动程序：它提供简单熟悉的命令行选项，并通过调用用于实现不同编译参数的工具集合来执行这些选项。

nvcc 的基本工作流包括将设备代码与宿主代码分离，并将设备代码编译为二进制形式或 *cubin* 对象。生成的宿主代码输出为可被另一个工具编译的 C 代码，或者输出为直接调用宿主编译器最近编译参数的对象代码。

应用程序要么忽略生成的宿主代码，而使用 CUDA 驱动程序 API（参见 4.5.3）加载并执行设备上的 *cubin* 对象，要么链接到生成的宿主代码，该宿主代码包括作为全局初始化数据数组的 *cubin* 对象，且包含从 4.2.3 所述的执行配置语法到必要 CUDA 运行时启动代码的转换，以便加载和启动每个已编译的内核（参见 4.5.2）。

编译器的前端按照 C++语法规则处理 CUDA 源文件。宿主代码完全支持 C++。但是，设备代码只完全支持 C++的 C 子集；像类、继承或基础块中的变量声明这种 C++的特有功能则不受支持。由于使用 C++语法规则的原因，空指针（比如 malloc()返回的值）不经过类型强制转换，不能赋值给非空指针。

有关 nvcc 工作流和命令选项的详细介绍，可参见单独的文档。

nvcc 引入了两个编译器指令，见下文所述。

4.2.5.1 __noinline__

默认情况下，__device__函数始终为内联。但是，__noinline__函数限定符暗示编译器尽量不内联函数。函数体与调用此函数的指令必须位于同一文件中。

对于带有指针参数的函数和含有长参数列表的函数，编译器将无视__noinline__限定符。

4.2.5.2 #pragma unroll

编译器会默认展开带有已知循环计数的小循环。而#pragma unroll 指令则可用于控制任何给定循环的展开。它必须放置在循环前，并只应用于此循环。它后面可以跟一个数字，用于指定循环必须展开多少次。

例如，在下列代码示例中循环将展开 5 次：

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

另外，编程人员应该确保展开将不影响程序的正确性（在上例中，如果 n 小于 5，则正确性可能受到影响）。

#pragma unroll 1 将禁止编译器展开循环。

如果在#pragma unroll 后面不指定任何数字，加之其循环计数是常数的情况下，循环将被完全展开，否则根本不展开。

4.3 共用运行时组件

共用运行时组件可以由宿主和设备函数使用。

4.3.1 内置向量类型

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

这些向量类型是从基本整数和浮点数类型派生而来的。作为结构体，其第 1、2、3、4 个分量分别可以通过字段 x、y、z 和 w 来访问。它们都具有 make_<type name>形式的构造函数；例如，

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

将创建类型为 int2、值为(x, y)的向量。

4.3.1.2 dim3 类型

此类型是基于 uint3 的整数向量类型，用于指定维度。定义 dim3 类型的变量时，未指定的分量将初始化为 1。

4.3.2 数学函数

表 B-1 包含当前支持的 C/C++标准库数学函数的完整列表，以及在设备上执行时各自的误差界。

在宿主代码中执行时，给定函数使用可用的 C 运行时实现。

4.3.3 时间函数

```
clock_t clock();
```

返回在每个时钟周期递增的计数值。

在内核开始和结束时取此计数器的值，求二者之差，并记录每个线程的结果，从而计量设备完全执行每个线程所用的时钟周期数，但这并非设备实际执行线程指令所用的时钟周期数。前者大于后者，因为线程是分时执行的。

4.3.4 纹理类型

CUDA 支持 GPU 上的一部分纹理硬件（它们原本是为图形处理而设计的）。从纹理内存而非全局内存读取数据具备几个性能优势，详见 5.4 节。

在内核中，使用名为*纹理拾取(texture fetches)*的设备函数读取纹理内存，如 4.4.5 所述。纹理拾取的第一个参数是一个名为*纹理参考(texture reference)*的对象。

纹理参考定义要拾取哪部分纹理内存。它必须通过宿主运行时函数（参见 4.5.2.6 和 4.5.3.9）绑定到一些内存区域（称为*纹理(texture)*），然后才能供内核使用。几个不同的纹理参考可以绑定到同一纹理或在内存中有所交叠。

纹理参考具有多个属性。其中之一是其维度，用于指定纹理是使用一个*纹理坐标(texture coordinate)*作为一维数组进行寻址，还是使用两个纹理坐标作为二维数组进行寻址。数组的元素称为纹理元素（*texel*，是“*texture element*”的简写）。

其它属性除了定义纹理拾取的输入和输出数据类型以外，还包括如何解释输入坐标，以及应执行什么处理。

4.3.4.1 纹理参考声明

纹理参考的一些属性不可变，而且必须在编译时已知；它们在声明纹理参考时被指定。纹理参考在文件范围内被声明为 `texture` 类型的变量：

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

其中：

- `Type` 指定拾取纹理时返回的数据类型；`Type` 限制为基本的整数和浮点数类型，以及 4.3.1.1 一节中定义的 1-、2-和 4-分量的向量类型之一。
- `Dim` 指定纹理参考的维度，等于 1 或 2；`Dim` 是可选参数，缺省值为 1；
- `ReadMode` 等于 `cudaReadModeNormalizedFloat` 或 `cudaReadModeElementType`；如果 `ReadMode` 为 `cudaReadModeNormalizedFloat`，且 `Type` 为 16-位或 8-位整数类型，则其值实际返回值为浮点数类型，即根据原整数类型的全范围进行归一化处理，结果被映射到[0.0, 1.0]区间（对于无符号整数）或[-1.0, 1.0]区间（对于有符号整数）；例如，值为 0xff 的无符号 8-位纹理元素返回值为 1；如果 `ReadMode` 为 `cudaReadModeElementType`，则不执行任何转换；`ReadMode` 是可选参数，默认为 `cudaReadModeElementType`。

4.3.4.2 运行时纹理参考属性

纹理参考的其它属性是可变的，可以在执行时通过宿主运行时（运行时 API 参见 4.5.2.6，驱动程序 API 参见 4.5.3.9）进行更改。它们指定寻址模式、纹理筛选以及纹理坐标是否归一化，详见下文。

默认情况下，使用[0, N)区间的浮点坐标实现纹理参考，其中 N 是与坐标相对应的维度中的纹理大小。例如，大小为 64×32 的纹理将分别在 x 和 y 维度上使用区间[0, 63]和[0, 31]中的坐标进行引用。归一化的纹理坐标将[0, N)区间映射为[0.0, 1.0)区间，因此，同一 64×32 纹理将在 x 和 y 维度上都使用范围[0, 1)中的归一化坐标来寻址。如果纹理坐标独立于纹理大小，那么归一化纹理坐标就成为了一些应用程序很自然的选择。

寻址模式定义纹理坐标超出范围时要执行的操作。使用非归一化纹理坐标时，超出范围[0, N)的纹理坐标将被夹合（clamp）：小于 0 的值设置为 0，大于等于 N 的值设置为 N-1。使用归一化纹理坐标时，默认采用 clamp 寻址模式：小于 0.0 或大于 1.0 的值设置到区间[0.0, 1.0)中。对于归一化坐标，也可以指定 wrap 寻址模式。当纹理包含周期性信息时，通常使用 wrap 寻址。wrap 寻址仅使用纹理坐标的小数部分；例如，1.25 当作 0.25 处理，-1.25 当作 0.75 处理。

仅为设置为返回浮点数数据的纹理执行线性纹理过滤。线性纹理过滤在相邻纹理元素之间执行低精度插值。启用线性纹理过滤时，将读取纹理拾取位置周围的纹理元素，并基于纹理坐标落入纹理元素之间的位置插值生成纹理拾取的返回值。对于一维纹理执行简单线性插值，对于二维纹理执行双线性插值。

附录 F 给出有关纹理拾取的更多详细信息。

4.3.4.3 线性内存中的纹理和 CUDA 数组中的纹理

纹理可以是线性内存或 CUDA 数组的任何区域（参见 4.5.1.2）。

在线性内存中分配的的纹理：

- 维度只能等于 1；
- 不支持纹理过滤；
- 只能使用非归一化的整数纹理坐标寻址；
- 寻址模式单一：越界的纹理访问将返回 0 值。

硬件在纹理基址上强制执行内存对齐。为对内存对齐进行抽象以便程序员更易使用，负责将纹理参考绑定到设备内存的函数会回传一个字节偏移量，该偏移量被应用到纹理拾取。由 CUDA 的分配例程返回的基指针符合此对齐约束，因此将已分配的指针传递到 `cudaBindTexture()/cuTexRefSetAddress()`，应用程序可以完全避免再设置偏移。

4.4 设备运行时组件

设备运行时组件只能在设备函数中使用。

4.4.1 数学函数

对于表 B-1 中的一些函数，设备运行时组件提供了一些精度略低但运行较快的版本；这些函数具有相同的名称，但加了前缀__（例如`__sin(x)`）。表 B-2 中列出这些内部函数及其各自的误差界。

编译器用一个选项（`-use_fast_math`）来强制将每个函数编译为其精度略低但运行较快的版本（如果存在的话）。

4.4.2 同步函数

```
void __syncthreads();
```

同步块中所有的线程。当所有线程都达到此同步点后，才继续执行后续代码。

`__syncthreads()`用于协调同一块内的线程间通信。当块中的一些线程访问共享或全局内存中的同一地址时，对于其中的一些内存访问，存在潜在的读后写、写后读或写后写的危害。这些危害可以通过同步这些访问之间的线程来避免。

`__syncthreads()`允许出现在条件代码中，但仅当条件在整个线程块中求值相同时才适用，否则代码执行可能暂挂或产生非预期的结果。

4.4.3 类型转换函数

下列函数后缀用于指定 IEEE-754 取整模式：

- `rn` 取整为最近的偶数，
- `rz` 向零取整，
- `ru` 向上取整（到正无穷大），
- `rd` 向下取整（到负无穷大）。

```
int __float2int_[rn,rz,ru,rd](float);
```

使用指定的取整模式将浮点参数转换为整数。

```
unsigned int __float2uint_[rn,rz,ru,rd](float);
```

使用指定的取整模式将浮点参数转换为无符号整数。

```
float __int2float_[rn,rz,ru,rd](int);
```

使用指定的取整模式将整数参数转换为浮点数。

```
float __uint2float_[rn,rz,ru,rd](unsigned int);
```

使用指定的取整模式将无符号整数参数转换为浮点数。

4.4.4 类型强制函数

```
float __int_as_float(int);
```

对整数参数执行浮点类型转换，保留值不变。例如，__int_as_float(0xC0000000)等于-2。

```
int __float_as_int(float);
```

对浮点参数执行整数类型转换，保留值不变。例如__float_as_int(1.0f)等于0x3f800000。

4.4.5 纹理函数

4.4.5.1 从设备内存取纹理

从设备内存取纹理时，使用tex1Dfetch()系列函数访问纹理；例如：

```
template<class Type>
Type tex1Dfetch(
   texture<Type, 1, cudaReadModeElementType> texRef,
   int x);

float tex1Dfetch(
   texture<unsigned char, 1, cudaReadModeNormalizedFloat> texRef,
   int x);

float tex1Dfetch(
   texture<signed char, 1, cudaReadModeNormalizedFloat> texRef,
   int x);

float tex1Dfetch(
   texture<unsigned short, 1, cudaReadModeNormalizedFloat> texRef,
   int x);

float tex1Dfetch(
```

```
   texture<signed short, 1, cudaReadModeNormalizedFloat> texRef,
   int x);
```

这些函数使用纹理坐标 x 对线性内存区域进行纹理拾取操作，然后绑定到纹理参考 texRef。这些方法不支持任何纹理过滤和寻址模式。对于整数类型，这些函数可以有选择地将整数转为 32 位浮点数。

除上述函数之外，还支持 2 元组和 4 元组；例如：

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

使用纹理坐标 x 对绑定到纹理参考 texRef 的线性内存进行纹理拾取操作。

4.4.5.2 从 CUDA 数组取纹理

从 CUDA 数组取纹理时，使用 tex1D()或 tex2D()访问纹理：

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

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

这些函数使用纹理坐标 x 和 y 拾取绑定到纹理参考 texRef 的 CUDA 数组。纹理参考的不变属性（编译时）和可变属性（运行时）的组合确定如何解释坐标、在纹理拾取期间执行何种处理、以及纹理拾取传递的返回值（参见 4.3.4.1 和 4.3.4.2）。

4.4.6 原子函数

原子函数（Atomic Functions）仅在计算能力 1.1 以上的设备中可用。附录 C 列出了这些设备。

原子函数在驻留于全局内存中的一个 32 位字上执行读-改-写原子操作。例如，atomicAdd()在全局内存中的某地址上读取一个 32 位字，为其加上一个整数，然后将结果写回同一地址。在保证执行时不受其它线程干扰这种意义上，此操作是原子的。换句话说，只有此操作完成之后，其它线程才可以访问此地址。

原子操作仅适用于 32 位有符号和无符号整数。

4.5 宿主运行时组件

宿主运行时组件只能由宿主函数使用。

它提供函数来处理：

- 设备管理，

- 上下文管理，
- 内存管理，
- 代码模块管理，
- 执行控制，
- 纹理参考管理，
- 与 OpenGL 和 Direct3D 的互操作性。

它由两套 API 组成：

- 名为 *CUDA 驱动程序 API* 的低层 API，
- 名为 *CUDA 运行时 API* 的高层 API（在 CUDA 驱动程序 API 之上实现）。

对于这两套 API，应用程序应该仅选择其中之一使用。

CUDA 运行时通过提供隐式初始化、上下文管理和模块管理使得设备代码管理变得容易。CUDA 运行时还负责通过 ncvv 生成 C 宿主代码（参见 4.2.5），因此链接到此代码的应用程序必须使用 CUDA 运行时 API。

相反，CUDA 驱动程序 API 需要更多的代码，更难于编程和调试，但是它提供较高级的控制，而且因为它仅处理 *cubin* 对象（参见 4.2.5），所以是独立于语言的。特别地，使用 CUDA 驱动程序 API 配置和启动内核比较困难，因为执行配置和内核参数必须使用显式函数调用来指定，而不是使用 4.2.3 中所述的执行配置语法来指定。另外，设备仿真（参见 4.5.2.7）不使用 CUDA 驱动程序 API。

CUDA 驱动程序 API 通过 `cuda` 动态库提供，并且它们所有的入口点都带有前缀 `cu`。

CUDA 运行时 API 通过 `cudart` 动态库提供，并且它们所有的入口点都带有前缀 `cuda`。

4.5.1 常用概念

4.5.1.1 设备

两种 API 都提供函数来枚举系统上可用的设备、查询其属性并选择其中之一执行内核（运行时 API 参见 4.5.2.2，驱动程序 API 参见 4.5.3.2）。

多个宿主线程可以在同一设备上执行设备代码，但在设计层面，一个宿主线程只能在一个设备上执行设备代码。因此，在多个设备上执行设备代码需要多个宿主线程。另外，通过一个宿主线程中的运行时创建的任何 CUDA 资源不能由其它宿主线程中的运行时使用。

4.5.1.2 内存

设备内存可以分配为*线性内存（linear memory）*或 *CUDA 数组（CUDA arrays）*。

线性内存在设备上以 32 位地址空间存在，因此单独分配的单元可以通过指针互相引用，二叉树是一个典型例子。

CUDA 数组为纹理拾取（参见 4.3.4）而优化，其内存布局是非透明的。CUDA 数组由组织成一维或二维数组的若干元素组成，每个元素具有 1、2 或 4 个分量，这些组件可以是有符号或无符号的 8、16 或 32-位整数、16-位浮点数（当前仅通过驱动程序 API 支持）或 32-位浮点数。CUDA

数组只能由内核通过纹理拾取来读取，且只能绑定到分量数一致的纹理参考。

线性内存和 CUDA 数组都可由宿主通过内存复制函数（如 4.5.2.3 和 4.5.3.6 所述）读取和写入。

宿主运行时还提供了函数用于分配和释放页面锁定的宿主内存，这与通过 `malloc()`分配的可分页宿主内存相反（运行时 API 参见 D.5.6 和 D.5.7，驱动程序 API 参见 E.8.5 和 E.8.6）。页面锁定内存的一个优点是如果宿主内存被分配为页面锁定，则宿主内存和设备内存之间的带宽较高（仅用于由分配该宿主内存的宿主线程执行的数据传送）。但是，页面锁定内存是稀有资源，所以早在可分页内存的分配之前，页面锁定内存的分配将可能失败。此外，分配太多的页面锁定内存会减少可用于操作系统分页的物理内存量，所以这将降低整体系统性能。

4.5.1.3 OpenGL 互操作性

OpenGL 缓冲对象（buffer object）可以映射到 CUDA 的地址空间中，从而允许 CUDA 读取由 OpenGL 写入的数据，或允许 CUDA 写入供 OpenGL 读取的数据。4.5.2.7 一节描述如何使用运行时 API 完成此操作，4.5.3.10 一节描述如何使用驱动程序 API 完成此操作。

4.5.1.4 Direct3D 互操作性

Direct3D 9.0 顶点缓冲（vertex buffer）可以映射到 CUDA 的地址空间，从而允许 CUDA 读取由 Direct3D 写入的数据，或允许 CUDA 写入供 Direct3D 读取的数据。4.5.2.8 一节描述如何使用运行时 API 完成此操作，4.5.2.8 一节描述如何使用驱动程序 API 完成此操作。

在同一时刻，CUDA 上下文仅能与一个 Direct3D 设备互操作，通过调用带括号的初始化/终止（begin/end）函数实现，详见 4.5.2.8 和 4.5.3.11。

CUDA 上下文和 Direct3D 设备必须在同一 GPU 上创建。这可以通过查询与 CUDA 设备相对应的 Direct3D 适配器来确保这一点，对于运行时 API 使用 `cudaD3D9GetDevice()`（参见 D.9.7），对于驱动程序 API 使用 `cuD3D9GetDevice()`（参见 E.11.7）。

必须使用 `D3DCREATE_HARDWARE_VERTEXPROCESSING` 标记来创建 Direct3D 设备。

目前 CUDA 还不支持：

- 除 Direct3D 9.0 之外的版本，
- 除顶点缓冲之外的 Direct3D 对象。

顺便提一句，当 Direct3D 和 CUDA 之间的负载均衡优先于互操作性时，`cuda3D9GetDevice()` 或 `cuD3D9GetDevice()` 还可以用于确保 Direct3D 和 CUDA 创建在不同的设备上。

4.5.1.5 异步并发执行

为了方便宿主和设备之间的并发执行，一些运行时函数是异步的：在设备完成请求的任务之前，控制就会返回到应用程序。这些函数包括：

- 内核里加了 `__global__` 限定符的函数或 `cuGridLaunch()` 和 `cuGridLaunchAsync()`；
- 执行内存复制并以 `Async` 为后缀的函数；
- 执行设备↔设备内存复制的函数；
- 设置内存的函数。

一些设备还可以在页面锁定宿主内存和设备内存之间执行复制的同时，并发执行内核函数。应用程序可以通过使用 `CU_DEVICE_ATTRIBUTE_GPU_OVERLAP` 调用 `cuDeviceGetAttribute()`来查询此功能（请分别参见 E.2.6）。对于 `cudaMallocPitch()`（参见 4.5.2.3）或 `cuMemAllocPitch()`（参见 4.5.3.6）分配的 CUDA 数组或 2D 数组，其内存复制还不能实现与内核函数的并发执行。

应用程序通过*流(streams)*管理并发。流是一个顺序执行的操作序列。另一方面，在同一时刻，不同的流之间可以不按顺序执行操作。

通过创建流对象，并将一个序列的内核启动和宿主↔设备内存复制的流参数设给此对象，可以定义流。4.5.2.4 描述如何使用运行时 API 完成此操作，4.5.3.7 介绍如何使用驱动程序 API 完成此操作。

对于任何内核启动、内存设置或内存复制，如果其流参数被指定为零，则仅当其所有先前的操作（包括属于流部分的操作）完成之后，该操作才能开始，而且在它完成之前，任何后续操作也都不能开始。

运行时 API 的 `cudaStreamQuery()`和驱动程序 API 的 `cuStreamQuery()`（请分别参见 D.3.2 和 E.5.2）可以检测流中所有先前的操作是否已经完成。运行时 API 的 `cudaStreamSynchronize()`和驱动程序 API 的 `cuStreamSynchronize()`（请分别参见 E.5.2 和 E.5.3）提供了一种方法，来显式地强制运行时在流中所有先前的操作完成之前等待。

同样地，使用运行时 API 的 `cudaThreadSynchronize()` 和驱动程序 API 的 `cuCtxSynchronize()`（请分别参见 D.2.1 和 E.3.5），应用程序可以强制运行时在所有先前的设备任务完成之前等待。为了避免不必要的速度降低，这些函数最适合用于计时，或用于隔离失败的启动或内存复制。

为让应用程序异步地记录程序中任何点的*事件（events）*，并查询这些事件被记录的时间，运行时还提供了一种方法来密切监控设备的进度并执行确准的计时。当事件之前的所有任务（或者给定流中的所有操作）都已完成时，记录此事件。4.5.2.5 描述如何使用运行时 API 完成此操作，4.5.3.8 描述如何使用驱动程序 API 完成此操作。

下列任意两个来自不同流的操作不能并发执行：页面锁定的宿主内存分配、设备内存分配、设备内存设置、设备↔设备内存复制或它们之间的事件记录。

在编程人员将CUDA_LAUNCH_BLOCKING环境变量设置为1的情况下，系统上运行的所有CUDA应用程序将全被禁止异步执行。此功能只提供用于调试目的，为让软件产品可靠运行，绝不要使用此功能。

4.5.2 运行时 API

4.5.2.1 初始化

运行时 API 没有任何显式初始化函数；第一次调用运行时函数时，运行时 API 即被初始化。当对运行时函数计时时，以及将第一次调用的错误代码解释到运行时中时，有必要紧记运行时 API 的初始化方式。

4.5.2.2 设备管理

章节 D.1 中的函数用于管理系统中的设备。

cudaGetDeviceCount()和 cudaGetDeviceProperties()提供一种用来枚举这些设备并检索其属性的方法。

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

cudaSetDevice()用来选择与宿主线程相关的设备：

```
cudaSetDevice(device);
```

在调用任何__global__函数或任何附录 D 中的函数之前必须选择一个设备。如果不执行显式调用 cudaSetDevice()，程序将自动选择设备 0，且任何后期的显式调用 cudaSetDevice()将不起作用。

4.5.2.3 内存管理

章节 D.5 中的函数用于分配和释放设备内存、访问在全局内存空间中声明的变量被分配的内存、和在宿主和设备内存之间传送数据。

使用 cudaMalloc()或 cudaMallocPitch()分配线性内存，使用 cudaFree()释放线性内存。

下列代码示例在线性内存中分配了包含 256 个浮点数元素的数组：

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

建议使用 cudaMallocPitch()进行 2D 数组的分配，该分配方式会对空间进行适当填补以满足 5.1.2.1 中描述的对齐要求，从而确保在访问行地址时或在 2D 数组和其它设备内存区域执行复制（使用 cudaMemcpy2D()函数）时获得最佳性能。返回的节距（pitch，即跨度）必须用于访问数组元素。下列代码示例分配浮点数值的 width×height 的 2D 数组，并显示如何在设备代码中循环遍历数组元素：

```
// host code
float* devPtr;
int pitch;
cudaMallocPitch((void**)&devPtr, &pitch,
                width * sizeof(float), height);
myKernel<<<100, 512>>>(devPtr, pitch);

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

CUDA 数组使用 cudaMallocArray()进行分配，使用 cudaFreeArray()进行释放。cudaMallocArray()的格式描述由 cudaCreateChannelDesc()创建。

下列代码示例分配一个 32-位浮点数元素的 width×height CUDA 数组：

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

cudaGetSymbolAddress()用于提取指向为全局内存空间中声明的变量分配的地址。已分配内存的大小通过 cudaGetSymbolSize()来获得。

D.5 一节列出用于在使用 cudaMalloc()分配的线性内存、使用 cudaMallocPitch()分配的线性内存、CUDA 数组和为全局或常量内存空间中声明的变量分配的内存之间复制内存的所有各种函数。

下列代码示例将 2D 数组复制到在上一代码示例中分配的 CUDA 数组：

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

下列代码示例将一些宿主内存数组复制到设备内存中：

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

下列代码示例将一些宿主内存数组复制到常量内存中：

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

4.5.2.4 流管理

D.3 一节中的函数用于创建和销毁流，并确定流的所有操作是否已经完成。

下列代码示例创建两个流：

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

下列代码示例定义了对这两个流的一系列操作：一个从宿主到设备的内存复制、一个内核启动和一个从设备到宿主的内存复制：

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

每个流将其部分输入数组 `hostPtr` 复制到设备内存中的数组 `inputDevPtr` 中，调用 `myKernel()`处理设备上的 `inputDevPtr`，并将结果 `outputDevPtr` 重新复制回到 `hostPtr` 的相应部分。使用两个流处理 `hostPtr` 允许一个流的内存复制与另一个流的内核执行同时发生。`hostPtr` 必须指向页面锁定的宿主内存，才能保证时间上的重叠：

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

最后调用 `cudaThreadSynchronize()`以确保在下一步处理之前所有流都已完成。

4.5.2.5 事件管理

D.4 一节中的函数用于创建、记录和销毁事件，并查询两个事件之间用去的时间。

下列代码示例创建两个事件：

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

这些事件可以对上一节的代码示例进行计时：

```
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);
```

4.5.2.6 纹理参考管理

D.6 一节的函数用于管理纹理参考。

由高层 API 定义的 `texture` 类型是一种从由低层 API 定义的 `textureReference` 类型中公共派生出来的结构，如下所示：

```
struct textureReference
{
  int                          normalized;
  enum cudaTextureFilterMode   filterMode;
  enum cudaTextureAddressMode  addressMode[2];
  struct cudaChannelFormatDesc  channelDesc;
}
```

- `normalized` 指定纹理坐标是否是归一化的；如果其值非 0，则纹理中的所有元素都使用区间`[0,1]`而非区间`[0,width-1]`或`[0,height-1]`中的纹理坐标来寻址，其中 `width` 和 `height` 是纹理大小；
- `filterMode` 指定过滤模式，即当拾取纹理时，如何基于输入纹理坐标来计算返回的值；`filterMode` 等于 `cudaFilterModePoint` 或 `cudaFilterModeLinear`；如果它为 `cudaFilterModePoint`，则返回值是纹理坐标最接近输入纹理坐标的纹理元素；如果它为 `cudaFilterModeLinear`，则返回值是纹理坐标最接近输入纹理坐标的两个（对于一维纹理）或四个（对于二维纹理）纹理元素的线性插值；`cudaFilterModeLinear` 仅在返回浮点类型值时有效；
- `addressMode` 指定寻址模式，即如何处理超出范围的纹理坐标；`addressMode` 是大小为 2 的数组，其第一个和第二个元素分别指定第一个和第二个纹理坐标的寻址模式；在寻址模式等于 `cudaAddressModeClamp` 的情况下，超出范围的纹理坐标将使用 clamp 寻址，等于 `cudaAddressModeWrap` 的情况下，超出范围的纹理坐标将使用 wrap 寻址；`cudaAddressModeWrap` 仅支持归一化的纹理坐标；
- `channelDesc` 描述拾取纹理时返回值的格式；`channelDesc` 具有下列类型：

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

其中，x、y、z 和 w 等于返回值的每个分量值，以位为单位，枚举值 f 可取：

- cudaChannelFormatKindSigned，如果这些分量为有符号整数类型，
- cudaChannelFormatKindUnsigned，如果这些分量为无符号整数类型，
- cudaChannelFormatKindFloat，如果这些分量为浮点数类型。

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

必须使用 cudaBindTexture()或 cudaBindTextureToArray()将纹理参考绑定到纹理之后，内核才可以使用纹理参考从纹理内存中读取数据。

下列代码示例将纹理参考绑定到 devPtr 指向的线性内存：

- 使用低层 API：

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

- 使用高层 API：

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

以下代码示例将纹理参考绑定到一个 CUDA 数组 cuArray：

- 使用低层 API：

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

- 使用高层 API：

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

将纹理绑定到纹理参考时指定的格式必须与声明纹理参考时指定的参数相匹配；否则，纹理拾取会导致不确定的结果。

cudaUnbindTexture()用于解除对纹理参考的绑定。

4.5.2.7 OpenGL 互操作性

D.8 一节中的函数用于控制与 OpenGL 的互操作性。

被映射的缓冲对象必须先在 CUDA 中注册。使用 cudaGLRegisterBufferObject()完成此操作：

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

注册之后，内核可以使用由 cudaGLMapBufferObject()返回的设备内存地址读取或写入缓冲对象：

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

使用 cudaGLUnmapBufferObject()解除映射，使用 cudaGLUnregisterBufferObject()解除注册。

4.5.2.8 Direct3D 互操作性

D.9 一节中的函数用于控制与 Direct3D 的互操作性。

与 Direct3D 的互操作性必须使用 cudaD3D9Begin()初始化，使用 cudaD3D9End()终止。

在这些调用之间，顶点对象必须注册到 CUDA 之后才能被映射。此操作使用 cudaD3D9RegisterVertexBuffer()来完成：

```
LPDIRECT3DVERTEXBUFFER9 vertexBuffer;
cudaD3D9RegisterVertexBuffer(vertexBuffer);
```

注册之后，内核可以使用由 cudaD3D9MapVertexBuffer()返回的设备内存地址读取或写入顶点缓冲：

```
LPDIRECT3DVERTEXBUFFER9 vertexBuffer;
float* devPtr;
cudaD3D9MapVertexBuffer((void**)&devPtr, vertexBuffer);
```

使用 cudaD3D9UnmapVertexBuffer()解除映射，使用 cudaD3D9UnregisterVertexBuffer()解除注册。

4.5.2.9 使用设备仿真模式调试

编程环境不包括对设备代码的任何原生调试支持，但提供了用于调试的设备仿真模式。在此模式下编译应用程序（使用-deviceemu 选项）时，设备代码在宿主上编译和运行，从而允许程序员使用宿主的原生调试支持来调试应用程序，就像此应用程序是宿主应用程序一样。预处理器宏__DEVICE_EMULATION__在此模式下被定义。应用程序的所有代码，包括使用的任何库，对于设备仿真或设备执行必须一致编译。将为设备仿真编译的代码与为设备执行编译的代码链接在一起将导致在初始化时返回 cudaErrorMixedDeviceExecution 运行时错误。

在设备仿真模式下运行应用程序时，编程模型由运行时仿真。对于线程块中的每个线程，运行时都在宿主上创建一个线程。编程人员必须确保：

- 宿主能够运行的最少线程数是每个线程块的最大线程数加上一个主线程。
- 有足够的内存可用于运行所有线程，已知每个线程需要 256KB 的堆栈。

设备仿真模式提供的许多功能使其成为一个非常有效的调试工具：

- 通过使用宿主的原生调试支持，编程人员可以利用调试器支持的所有功能，比如设置断点和监测数据。

- 因为设备代码编译后在宿主上运行，所以也可以使用那些原来不能在设备上运行的代码，比如到文件或到屏幕的输入和输出操作（`printf()`等）。
- 因为所有的数据驻留在宿主上，所以任何设备或宿主上专有的数据可以从设备或宿主代码上读取；同样地，任何设备或宿主函数可以从设备或宿主代码中调用。
- 如果错误使用了内部同步，则运行时将检测到死锁情况。

编程人员必须切记，设备仿真模式是在仿真设备，而非模拟设备。因此，设备仿真模式在查找算法错误时十分有用，但某些错误难以查找：

- 当网格中的多个线程可能同时访问某个内存位置时，则在设备仿真模式下运行的结果可能与在设备上的结果不同，因为在仿真模式下，线程顺序执行。
- 当在宿主上废弃一个指向全局内存的指针或在设备上废弃一个指向宿主内存的指针时，设备执行几乎肯定以一些不确定的方式失败，而设备仿真则可以生成正确的结果。
- 大多数时候，在设备上执行时与在设备仿真模式下的宿主上执行时，同一浮点计算将不会生成完全相同的结果。这是预期结果，因为一般说来，只需使用略有不同的编译器选项就能让同一浮点计算获得不同的结果，更不要说不同的编译器、不同的指令集或不同的架构了。

  特别地，一些宿主平台将单精度浮点计算的中间结果存储在更高精度的寄存器中，这可能造成与设备仿真模式下的精度有显著差异。当这种情况发生时，编程人员可以尝试下列任何方法（但不能保证可行）：

  - 将一些浮点变量声明为 volatile，以强制单精度存储；
  - 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()` 和 `_FPU_SETCW()`，或在 Windows 上使用 `_controlfp()`，以强制对某段代码进行单精度浮点计算，方法是在其前后使用指令

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

  或在该代码前面添加

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

  以存储控制字的当前值，并对其进行更改以强制尾数以 24-位存储，并在结尾处使用

```
_FPU_SETCW(originalCW);
```

  或

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

  以恢复原始控制字。

与计算设备（参见附录 A）不同，宿主平台通常还支持非归一化的数值。这可能导致设备仿真和

设备执行模式之间的结果显著不同，因为一些计算可能在一种情况下生成有限值的结果，而在另一种情况下生成无限值的结果。

4.5.3 驱动程序 API

驱动程序 API 是基于句柄的命令式 API：大多数对象通过不透明句柄来引用，这些句柄被指定到相应函数以处理对象。

CUDA 中的可用对象汇总在表 4-1 中。

表 4-1. CUDA 驱动程序 API 中的可用对象

对象	句柄	描述
设备	CUdevice	支持 CUDA 的设备
上下文	CUcontext	几乎相当于 CPU 进程
模块	CUmodule	几乎相当于动态库
函数	CUfunction	内核
堆内存	CUdeviceptr	指向设备内存的指针
CUDA 数组	CUarray	承载设备上一维或二维数据的不透明容器，通过纹理参考可读
纹理参考	CUtexref	描述如何解释纹理内存数据的对象

4.5.3.1 初始化

在调用附录 E 中的任何函数（参见 E.1）之前，需要使用 `cuInit()`进行初始化。

4.5.3.2 设备管理

E.2 中的函数用于管理系统中现有的设备。

`cuDeviceGetCount()`和 `cuDeviceGet()`提供了枚举这些设备的方法，E.2 中的其它函数用以获取其属性：

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

4.5.3.3 上下文管理

E.3 中的函数用于创建、附加和分离 CUDA 上下文。

CUDA 上下文类似于 CPU 进程。在计算 API 中执行的所有资源和操作都封装在 CUDA 上下文中，并且当上下文销毁时，系统将自动清除这些资源。除模块和纹理参考等对象之外，每个上下文还具有自己独立的 32-位地址空间。因此，不同 CUDA 上下文中的 `CUdeviceptr` 值引用不同的内存位置。

上下文具有与宿主线程一对一的对应关系。在同一时间，宿主线程只能有一个设备上下文。当宿主线程使用 `cuCtxCreate()` 创建上下文时，此上下文就成为该线程的当前上下文。

如果有效上下文不是某线程的当前上下文，则在上下文中操作的 CUDA 函数（不涉及设备仿真或上下文管理的大多数函数）将返回 `CUDA_ERROR_INVALID_CONTEXT`。

要简化在同一上下文中执行的第三方授权代码之间的互操作性，驱动程序 API 维护了一个使用计数，该计数根据给定上下文的每个独立客户机递增。例如，如果加载了三个库使用相同的 CUDA 上下文，则每个库必须调用 `cuCtxAttach()` 递增使用计数，并在库完成使用上下文时，调用 `cuCtxDetach()` 递减使用计数。当使用计数等于 0 时，则销毁上下文。对于大多数库，应用程序一般会在加载或初始化库之前就已经创建好了 CUDA 上下文；这样，应用程序可以使用其自己的启发式方法创建上下文，而库只需在传递给它的上下文上操作。

4.5.3.4 模块管理

E.4 中的函数用于加载和卸载模块，并获取模块中定义的变量或函数的句柄和指针。

模块是可动态加载的、包含设备代码和数据的包，类似于 Windows 中的 DLL，是 nvcc 的输出（参见 4.2.5）。所有符号（包括函数、全局变量和纹理参考）的名称在模块范围内维护，以便由不同第三方写好的模块可以在同一 CUDA 上下文中互操作。

下列代码示例加载模块并检索指向某个内核的句柄：

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

4.5.3.5 执行控制

E.7 中介绍的函数管理设备上内核的执行。`cuFuncSetBlockShape()` 设置给定函数的每块线程数，以及如何分配其线程 ID。`cuFuncSetSharedSize()` 设置函数的共享内存大小。`cuParam*()` 系列函数用于指定下一次调用 `cuLaunchGrid()` 或 `cuLaunch()` 启动内核时将提供给内核的参数。

```
cuFuncSetBlockShape(cuFunction, blockWidth, blockHeight, 1);
int offset = 0;
int i;
cuParamSeti(cuFunction, offset, i);
offset += sizeof(i);
```

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

4.5.3.6 内存管理

E.8 中的函数用于分配和释放设备内存，并在宿主和设备内存之间传送数据。

使用 `cuMemAlloc()`或 `cuMemAllocPitch()`分配线性内存，使用 `cuMemFree()`进行释放。

下列代码示例在线性内存中分配了包含 256 个浮点数元素的数组：

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

建议使用 `cuMemAllocPitch()`进行 2D 数组的分配，该分配方式会对空间进行适当填补以满足 5.1.2.1 中描述的对齐要求，从而确保在访问行地址时或在 2D 数组和其它设备内存区域执行复制（使用 `cuMemcpy2D()`）时获得最佳性能。返回的节距必须用于访问数组元素。下列代码示例分配浮点数值的 `width×height` 的 2D 数组，并显示如何在设备代码中循环遍历数组元素：

```
// host code
CUdeviceptr devPtr;
int pitch;
cuMemAllocPitch(&devPtr, &pitch,
                width * sizeof(float), height, 4);
cuModuleGetFunction(&cuFunction, cuModule, "myKernel");
cuFuncSetBlockShape(cuFunction, 512, 1, 1);
cuParamSeti(cuFunction, 0, devPtr);
cuParamSetSize(cuFunction, sizeof(devPtr));
cuLaunchGrid(cuFunction, 100, 1);

// device code
__global__ void myKernel(float* devPtr)
{
    for (int r = 0; r < height; ++r) {
        float* row = (float*)((char*)devPtr + r * pitch);
        for (int c = 0; c < width; ++c) {
            float element = row[c];
        }
    }
}
```

使用 `cuArrayCreate()`创建 CUDA 数组，使用 `cuArrayDestroy()`进行销毁。

下列代码示例分配了一个 32-位浮点数元素的 `width×height` 的 CUDA 数组：

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

E.8 一节列出用于在使用 cuMemAlloc()分配的线性内存、使用 cuMemAllocPitch()分配的线性内存、CUDA 数组之间复制内存的所有各种函数。下列示例代码将 2D 数组复制到上一代码示例中分配的 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);
```

下列代码示例将一些宿主内存数组复制到设备内存中：

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

4.5.3.7 流管理

E.5 中的函数用于创建和销毁流，并确定流的所有操作是否已经完成。

下列代码示例创建两个流：

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

下列代码示例定义了对这两个流的一系列操作：一个从宿主到设备的内存复制、一个内核启动和一个从设备到宿主的内存复制：

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

```
}
for (int i = 0; i < 2; ++i)
    cuMemcpyDtoHAsync(hostPtr + i * size, outputDevPtr + i * size,
                      size, stream[i]);
cudaCtxSynchronize();
```

每个流将其部分输入数组 hostPtr 复制到设备内存中的数组 inputDevPtr 中，通过调用 cuFunction 处理设备上的 inputDevPtr，并将结果 outputDevPtr 重新复制回到 hostPtr 的相应部分。使用两个流处理 hostPtr 允许一个流的内存复制与另一个流的内核执行同时发生。

hostPtr 必须指向页面锁定的宿主内存，才能保证时间上的重叠：

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

最后调用 cuCtxSynchronize()以确保在下一步处理之前所有流都已完成。

4.5.3.8 事件管理

E.6 中的函数用于创建、记录和销毁事件，并查询两个事件之间用去的时间。

下列代码示例创建两个事件：

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

这些事件可以对上一节的代码示例进行计时：

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

4.5.3.9 纹理参考管理

E.9 中的函数用于管理纹理参考。

必须使用 cuTexRefSetAddress()或 cuTexRefSetArray()将纹理参考绑定到纹理之后，内核才可以使用纹理参考从纹理内存中读取数据。

如果模块 cuModule 包含某个定义如下的纹理参考 texRef：

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

下列代码示例获取 texRef 的句柄：

```
CUtexref cuTexRef;
cuModuleGetTexRef(&cuTexRef, cuModule, "texRef");
```

下列代码示例将 texRef 绑定到由 devPtr 指向的一些线性内存：

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

下列代码示例将 texRef 绑定到 CUDA 数组 cuArray：

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

E.9 列出用于设置纹理参考的地址模式、过滤模式、格式和其它标识的各种函数。将纹理绑定到纹理参考时指定的格式必须与声明纹理参考时指定的参数相匹配；否则，纹理拾取会导致不确定的结果。

4.5.3.10 OpenGL 互操作性

E.10 中的函数用于控制与 OpenGL 的互操作性。

与 OpenGL 的互操作性必须使用 cuGLInit()进行初始化。

被映射的缓冲对象必须先在 CUDA 中注册。使用 cuGLRegisterBufferObject()完成此操作：

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

注册之后，内核可以使用由 cuGLMapBufferObject()返回的设备内存地址读取或写入缓冲对象：

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

使用 cuGLUnmapBufferObject()解除映射，使用 cuGLUnregisterBufferObject()解除注册。

4.5.3.11 Direct3D 互操作性

D.9 中的函数用于控制与 Direct3D 的互操作性。

与 Direct3D 的互操作性必须使用 cuD3D9Begin()进行初始化，使用 cuD3D9End()终止：

在这些调用之间，顶点对象必须注册到 CUDA 之后才能被映射。此操作使用 cuD3D9RegisterVertexBuffer()完成：

```
LPDIRECT3DVERTEXBUFFER9 vertexBuffer;
cuD3D9RegisterVertexBuffer(vertexBuffer);
```

注册之后，内核可以使用由 cuD3D9MapVertexBuffer()返回的设备内存地址读取或写入顶点对象：

```
LPDIRECT3DVERTEXBUFFER9 vertexBuffer;
```

```
CUdeviceptr devPtr;
int size;
cuD3D9MapVertexBuffer(&devPtr, &size, vertexBuffer);
```

使用 cuD3D9UnmapVertexBuffer()解除映射，使用 cuD3D9UnregisterVertexBuffer()解除注册。

第 5 章　性能指南

5.1　指令性能

对一个线程 warp 执行的指令，多处理器必须：

- 读取 warp 的每个线程的指令操作数，
- 执行指令，
- 写 warp 的每个线程的结果。

因此，有效的指令吞吐量取决于额定指令吞吐量以及内存延迟和带宽。它通过下列方式最大化：

- 尽量不使用低吞吐量的指令（参见 5.1.1），
- 最大化每种内存的可用内存带宽（参见 5.1.2），
- 允许线程调度器尽可能地将内存事务与数学计算同时执行，这需要：
  - 由线程执行的程序具有高算术密度，即每个内存操作对应更多算术操作；
  - 每个多处理器具有许多活动线程，详见 5.2。

5.1.1　指令吞吐量

5.1.1.1　算术指令

要发射 warp 的一个指令，多处理器花费：

- 4 个时钟周期，用于：浮点加、浮点乘、浮点乘-加、整数加、位操作、比较、求最小、求最大、类型转换指令；
- 16 个时钟周期，用于：倒数、平方根倒数、`__log(x)`（参见表 B-2）。

32-位整数乘法使用 16 个时钟周期，而`__mul24` 和`__umul24`（参见附录 B）提供了 4 个时钟周期的有符号和无符号 24 位整数乘法。但是，在将来的架构中，`__[u]mul24` 将比 32-位整数乘法慢，所以我们建议编写两个内核供应用程序在不同情况下调用，其中一个使用`__[u]mul24`，另一个使用一般的 32-位整数乘法。

整数除法和模操作特别昂贵，应该尽可能地避免，或者尽量替换为位操作：如果 n 是 2 的幂，则(i/n)等于(i>>log2(n))，(i%n)等于(i&(n-1))；如果 n 是文本型的，则编译器将执行这些转换。

其它函数使用更多时钟周期，因为它们实现为多个指令的组合。

浮点数平方根实现为平方根倒数再求倒数，而非平方根倒数再做乘法，所以它对于 0 和无穷大能够获得正确的结果。因此，它处理 1 个 warp 花费 32 个时钟周期。

浮点除法花费 36 个时钟周期，但__fdividef(x, y)提供了 20 个时钟周期的更快版本（参见附录 B）。

__sin(x)、__cos(x)、__exp(x)花费 32 个时钟周期。

有时候，编译器不得不额外插入转换指令，从而引入额外的执行周期。这种情况包括：

- 对操作数为 char 或 short 的函数操作，其操作数通常需要转换为 int，
- 在单精度浮点数计算中，将双精度浮点数常量（不使用任何类型后缀定义）作为输入值，
- 对表 B-1 中定义的数学函数的双精度版本，将单精度浮点数变量作为输入参数。

后两种情况可以通过下列方式避免：

- 对单精度浮点数常量，使用 f 后缀定义，比如 3.141592653589793f、1.0f、0.5f，
- 对数学函数的单精度版本，也使用 f 后缀定义，比如 sinf()、logf()、expf()。

对于单精度代码，我们强烈建议使用浮点类型和单精度数学函数。当在不支持原生双精度的设备（比如计算能力 1.x 的设备）上编译时，双精度类型默认降级为单精度，双精度数学函数映射为其单精度对应函数。但是对于将来支持双精度的设备，这些函数将映射为双精度实现。

5.1.1.2 控制流指令

任何流控制指令(if, switch, do, for, while)都会导致 warp 内的线程分流（diverge），即按照不同的执行路径执行，其结果是对有效指令吞吐量产生显著影响。如果线程分流发生，则不同的执行路径是串行化执行的，增加此 warp 执行的指令总数。当所有不同的执行路径都已完成时，线程才合流（converge）到同一执行路径。

当线程 ID 决定控制流指令时，要获得最佳性能，就应该写入控制条件，以便让尽量少的的 warp 产生分流。这种优化方式是可能实现的，因为如 3.2 所述，warp 在块中的分布具有确定性。比如，当控制条件仅取决于(threadIdx / WSIZE)时，这种优化方式就成立，其中 WSIZE 是 warp 大小。在这种情况下，没有任何 warp 会分流，因为控制条件与 warp 已完美对齐。

有时，编译器可以通过使用分支预测展开循环或优化 `if` 或 `switch` 语句，详细说明如下。在这些情况下，warp 绝不会分流。编程人员也可以使用`#pragma unroll` 指令控制循环展开（参见 4.2.5.2）。

当使用分支预测时，所有基于控制条件的指令都会被执行。而且，与每线程条件代码和基于控制条件的真/假*谓词（predicate）*相关每条指令都会被调度执行，但只有具有真谓词的指令将实际执行。具有假谓词的指令不写入结果，而且不取地址或读取操作数。

仅当由分支条件控制的指令数小于或等于特定临界值时，编译器才将分支指令替换为预测指令；如果编译器确定许多 warp 将会产生分流，则此临界值是 7，否则是 4。

5.1.1.3 内存指令

内存指令包括从共享或全局内存中读取或写入的任何指令。多处理器使用 4 个时钟周期来发射 warp 的一个内存指令。此外，当访问全局内存时，还有 400 到 600 个时钟周期的内存延迟。

例如，下列示例代码中的赋值操作符：

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

发射一个读取指令花费 4 个时钟周期，对共享内存的写入花费 4 个时钟周期，但最关键的是，从全局内存中读取浮点数会花费 400 到 600 个时钟周期。

如果在等待全局内存访问完成期间，线程调度器可以发射足够多的独立算术指令，则大部分全局内存访存延迟可以被隐藏掉。

5.1.1.4 同步指令

如果没有任何线程必须等待其它任何线程，则 1 个 warp 发射`__syncthreads` 将花费 4 个时钟周期。

5.1.2 内存带宽

每个内存空间的有效带宽主要取决于访存模式，详见下列小节。

因为设备内存与片上内存相比具有更高的延迟和更低的带宽，所以设备内存访问必须最小化。典型的编程模式是将来自设备内存的数据存储到共享内存中；换句话说，就是让块中的每个线程：

- 将设备内存中的数据加载到共享内存中，
- 与块的所有其它线程同步，以便每个线程可以安全读取由不同线程写入的某块共享内存，

- 处理共享内存中的数据，
- 如果必要的话，重新同步以确保共享内存已经由结果更新，
- 将结果写回到设备内存中。

5.1.2.1 全局内存

全局内存空间没有高速缓存，所以最重要的是按照正确的访问模式获得最大的内存带宽，尤其是已知对设备内存的访问有多昂贵时。

首先，设备能够在单个指令中将 32-位、64-位或 128-位字从全局内存读取到寄存器。用如下赋值方式：

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

编译到单个加载指令中，type 必须使得 sizeof(type)等于 4、8 或 16，且类型为 type 的变量必须对齐为 sizeof(type)个字节（也就是说，让其地址是 sizeof(type)的倍数）。

对于 4.3.1.1 一节中介绍的内置类型，比如 float2 或 float4，对齐将自动完成。

对于结构体，大小和对齐要求可以由编译器使用对齐指定符__align__(8)或__align__(16)强制执行，比如

```
struct __align__(8) {
    float a;
    float b;
};
```

或

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

对于大于 16 个字节的结构体，编译器生成多个加载指令。要确保它生成最少的指令，则这种结构体应使用__align__(16)定义，比如

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

会编译为两个 128-位加载指令，而非 5 个 32-位加载指令。

第二，在执行单个读取或写入指令期间，每个半 warp 中同时访问全局内存地址的每个线程应该进行排列，以便内存访问可以合并到单个邻近的、对齐的内存访问中。

更准确地说，在每个半 warp 中，线程号为 N 的线程应访问地址

```
HalfWarpBaseAddress + N
```

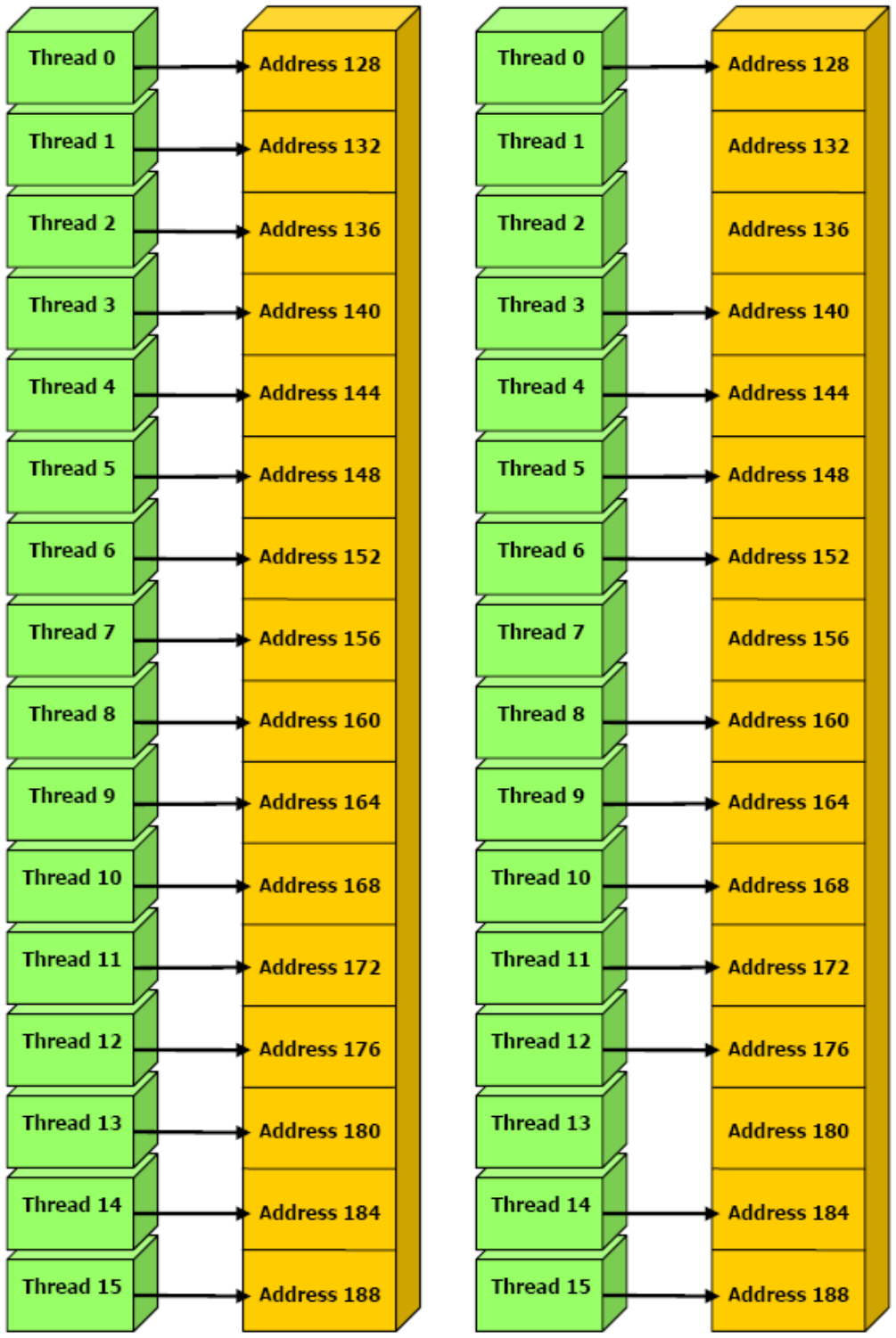
其中，类型为 type*和 type 的 HalfWarpBaseAddress 满足上述的大小和对齐要求。此外，HalfWarpBaseAddress 应对齐为 16*sizeof(type)个字节（比如，是 16*sizeof(type)的倍数）。对于驻留在全局内存中或由 D.5 或 E.8 中的内存分配例程之一返回的变量，其任何地址 BaseAddress 始终对齐为至少 256 个字节，所以为了满足内存对齐约束，HalfWarpBaseAddress-BaseAddress 应是 16*sizeof(type)的倍数。

注意，如果半 warp 满足上述所有要求，即使半 warp 的一些线程不实际访问内存，每线程内存访问也将合并。

与仅分别执行每个半 warp 的合并相对，我们建议执行整个 warp 的合并，因为对整个 warp 的访存适当合并将成为将来设备的必要条件。

图 5-1 显示了已合并访存的示例，而图 5-2 和图 5-3 显示了未合并访存的示例。

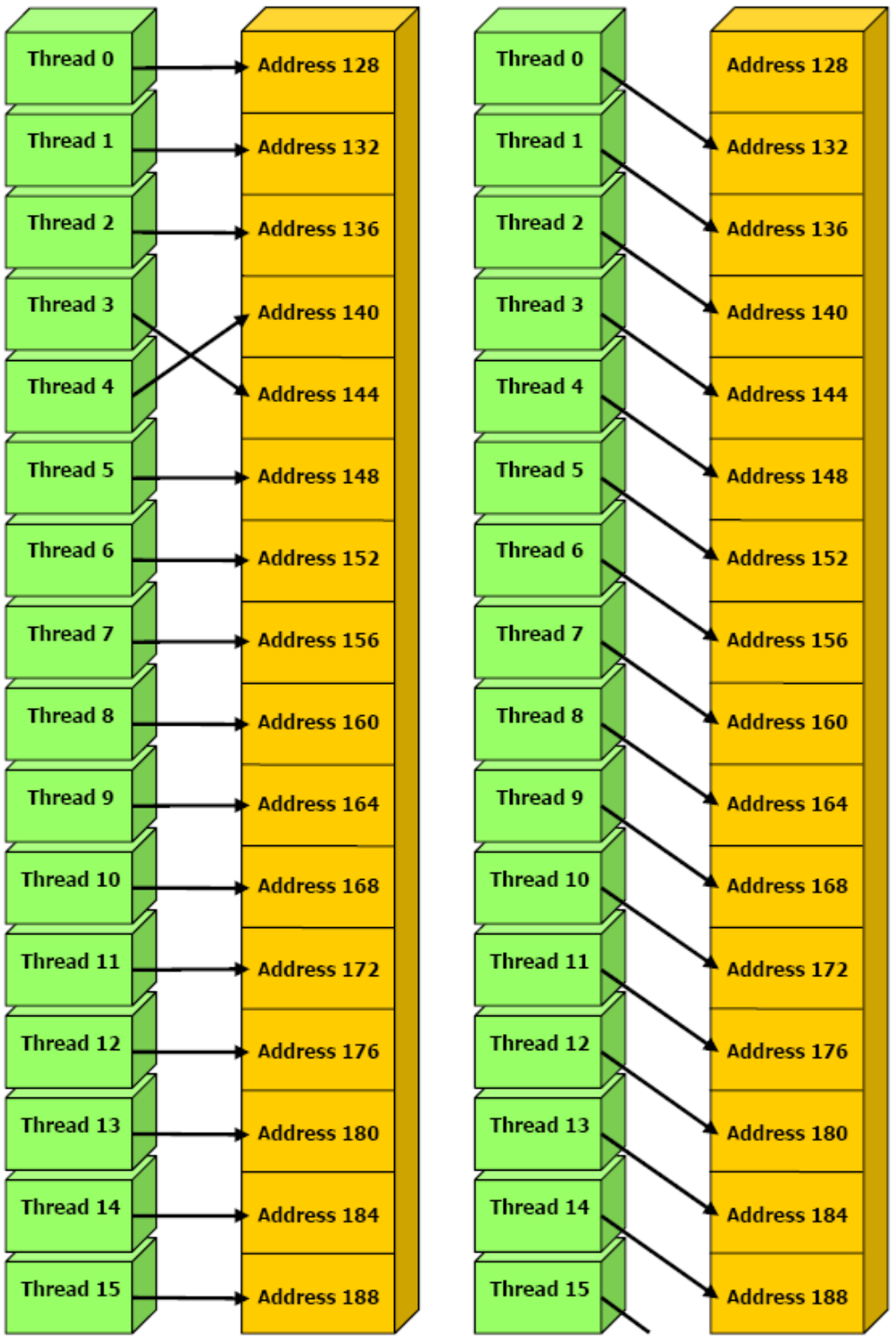
已合并 64-位访存提供了比已合并 32-位访存稍低的带宽，已合并 128-位访存提供了比已合并 32-位访存低很多的带宽。然而，当访存是 32 位时，尽管未合并访存的带宽比已合并访存的带宽低大约一个数量级，但当访存是 64-位时，仅低大约 4 倍，当访存是 128-位时，仅低大约 2 倍。



左：访问已合并的 `float` 内存。

右：访问已合并的 `float` 内存（被分流的 warp）。

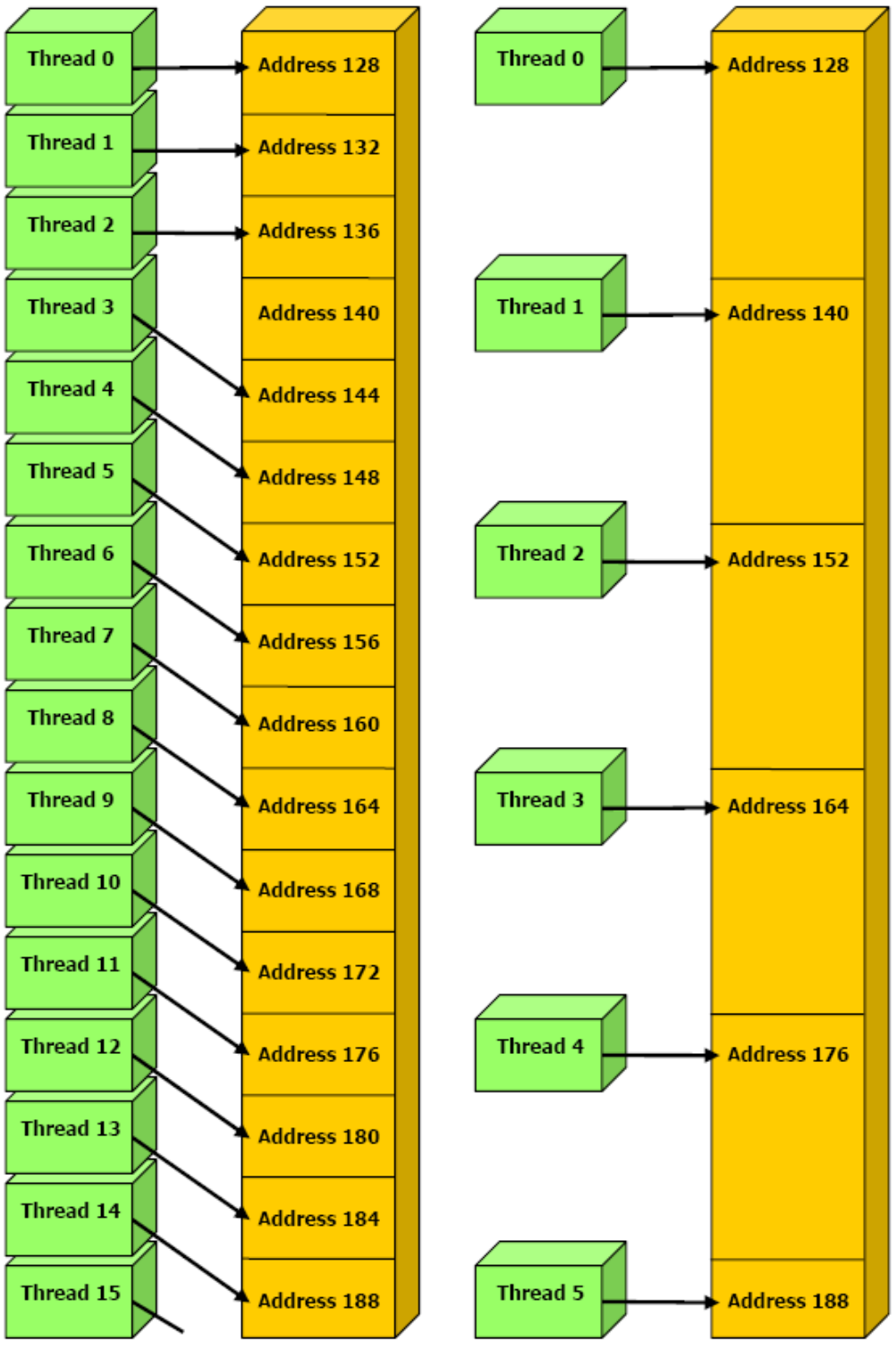
图 5-1. 已合并全局内存访问模式的示例



左：访问非顺序的 `float` 内存。

右：未对齐的初始地址。

图 5-2. 未合并全局内存访问模式的示例



左：访问不相连的 `float` 内存。

右：访问未合并的 `float3` 内存。

图 5-3. 未合并全局内存访问模式的示例

常见的全局内存访问模式是，线程 ID 为 tid 的每个线程访问位于地址 BaseAddress（类型为 type*）上的数组的一个元素，使用下列地址：

```
BaseAddress + tid
```

为合并访存，type 必须满足上述大小和对齐要求。特别地，这意味着，如果 type 是大于 16 个字节的结构体，则应分割为满足这些要求的多个结构体，而且数据应在内存中排列为这些结构体的多个数组，而非类型为 type*的单个数组。

另一个常见的全局内存访问模式是，当索引为(tx,ty)的每个线程访问位于地址 BaseAddress（类型为 type*、宽度为 width）上的 2D 数组的一个元素时，使用下列地址

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

在这种情况下，仅当满足下列条件，才能获得线程块的所有半 warp 的访存合并：

- 线程块的宽是半个 warp 大小的倍数；
- width 是 16 的倍数。

特别地，这意味着，如果宽度不是 16 的倍数的数组实际使用向上取整为最接近的 16 的倍数进行分配，且其行相应地进行填补，则此数组将获得较高效的访问。cudaMallocPitch()和 cuMemAllocPitch()函数及其相关的内存复制函数（参见 D.5 和 E.8）允许编程人员编写不依赖于硬件的代码来分配符合这些约束的数组。

5.1.2.2 常量内存

常量内存空间具有高速缓存，所以仅在高速缓存未命中时，才从设备内存中读取数据，否则仅花费读取常量高速缓存的时间。

对于半 warp 的所有线程，只要所有线程读取同一地址，则从常量内存中读取与从寄存器中读取一样快。访存花费的时间随读取不同地址的线程数目线性增减。与仅让每个半 warp 中的所有线程读取同一地址相对，我们建议让整个 warp 的所有线程读取同一地址，因为将来的设备将需要此操作来实现完全的快速读取。

5.1.2.3 纹理内存

纹理内存空间具有高速缓存，所以纹理拾取仅在高速缓存未命中时，才从设备内存中读取数据，否则仅花费读取纹理高速缓存的时间。纹理高速缓存针对 2D 空间局部性进行了优化，所以读取紧密相邻的纹理地址的同一 warp 的线程将达到最佳性能。此外，它还为具有恒定延迟的流式拾取而设计，比方说，高速缓存命中降低了 DRAM 带宽需求，但没有降低拾取延迟。

通过纹理拾取读取设备内存可能是从全局或常量内存中读取设备内存的有利备选方案，详见 5.4。

5.1.2.4 共享内存

因为位于芯片上，所以共享内存空间要比本地和全局内存空间快得多。实际上，对于 warp 的所有线程，只要在线程之间没有任何存储体冲突（bank conflict），访问共享内存就与访问寄存器一样快，详见下文。

为获得高内存带宽，共享内存被划分为同样大小的、可以同时访问的内存块，名为存储体（bank）。因此，由属于 *n* 个存储体的 *n* 个地址组成的任何内存读取或写入请求都可以同时获得服务，最后可获得的有效带宽是单个模块的带宽的 *n* 倍。

但是如果某个内存请求的两块地址落在同一存储体内，则会导致存储体冲突，访问也必须串行化。硬件将带有存储体冲突的内存请求按需分成许多单独的无冲突的请求，有效带宽就会减少数倍，该倍数与单独内存请求的数目相等。如果单独内存请求的数目为 *n*，则把此种情况称为 *n* 路（*n*-way）存储体冲突。

要获得最高性能，很有必要理解内存地址映射到存储体的方式，进而调度内存请求，最小化存储体冲突。

共享内存空间的存储体组织为：连续的 32-位字分配到连续的存储体中，每个存储体的带宽为 32-位每两个时钟周期。

对于计算能力 1.x 的设备，warp 大小为 32，存储体数为 16（参见 5.1）；将 warp 的共享内存请求划分为第一半 warp 的一个请求和第二半 warp 的一个请求。因此，属于第一半 warp 的线程和属于第二半 warp 的线程间不会发生任何存储体冲突。

一个常见的情况是每个线程从按线程 ID `tid` 索引的数据中使用某个跨度 `s` 来访问一个 32 位字：

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

这种情况下，只要 `s*n` 是存储体数 `m` 的倍数，或者同等地，只要 `n` 是 `m/d` 的倍数（`d` 是 `m` 和 `s` 最大公约数），则线程 `tid` 和 `tid+n` 将访问同一存储体。因此，仅当半 warp 大小小于或等于 `m/d` 时，才不会发生存储体冲突。对于计算能力 1.x 的设备，仅当 `d` 等于 1 时才不会发生任何存储体冲突，或者换句话说，因为 `m` 是 2 的幂次，所以仅当 `s` 是奇数时，才不会发生任何存储体冲突。

图 5-4 和图 5-5 显示了一些无冲突访存的示例，而图 5-6 显示了一些导致存储体冲突的访存示例。

值得一提的其它情况出现在每个线程访问小于或大于 32-位的元素时。例如，如果按下列方式访问 `char` 类型数组，则会发生存储体冲突。

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

这是因为 shared[0]、shared[1]、shared[2]和 shared[3]同属于一个存储体。但是，如果按下列方式访问同一数组，则不会发生任何存储体冲突：

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

对结构体而言，其编译后的内存请求与结构体成员数一样多，所以，下列代码：

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

将导致下列结果：

- 如果 type 定义如下，则结果为三个单独的无存储体冲突的内存读取

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

因为每个成员使用 3 个 32-位字的跨度来访问。

- 如果 type 定义如下，则结果为两个单独的有存储体冲突的内存读取

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

因为每个成员使用 2 个 32 位字的跨度来访问。

- 如果 type 定义如下，则结果为两个单独的有存储体冲突的内存读取

```
struct type {
       float f;
       char  c;
};
```

因为每个成员使用 5 个字节的跨度来访问。

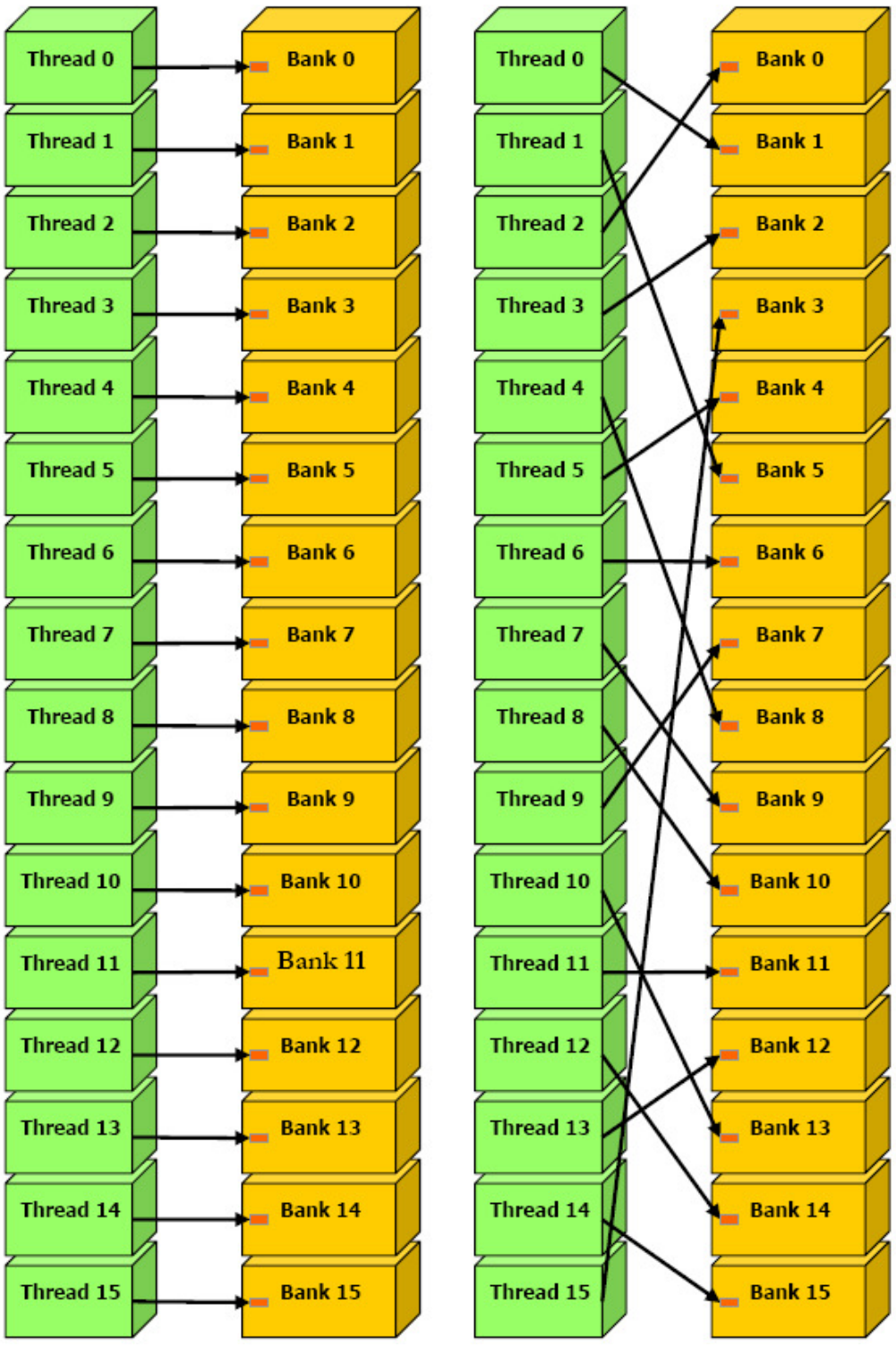
最后，共享内存还具有广播机制，当处理一个内存读取请求时，可以读取一个 32-位字并同时广播到多个线程。当半 warp 的多个线程从含有同一个 32-位字的地址读取时，这将减少存储体冲突的数目。更精确地说，对多个地址的内存读取请求由多个时间步完成（每两个时钟周期一步）每步处理一个这些地址的无冲突子集，直到所有地址都已完毕；在每一步，子集从尚未进行的剩余地址中构建，过程如下：

- 选择由剩余地址指向的其中一个字作为广播字，
- 将下列内容包括在子集中：
  - 位于广播字内的所有地址，
  - 指向每个存储体的剩余地址中的一个地址。

选择哪个字作为广播字，以及在每个周期选择哪个存储体地址都不是特定的。

常见的无冲突情况发生在半 warp 的所有线程从同一个 32-位字地址中读取时。

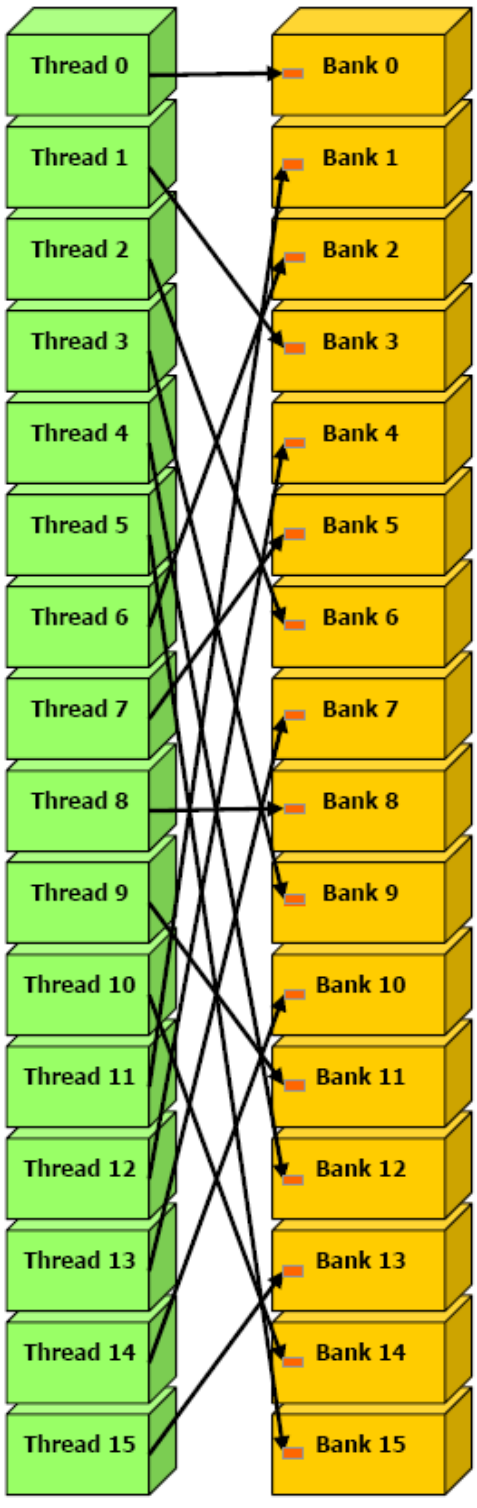
图 5-7 显示了一些涉及广播机制的内存读取访问的示例。



左：跨度为一个 32-位字的线性寻址

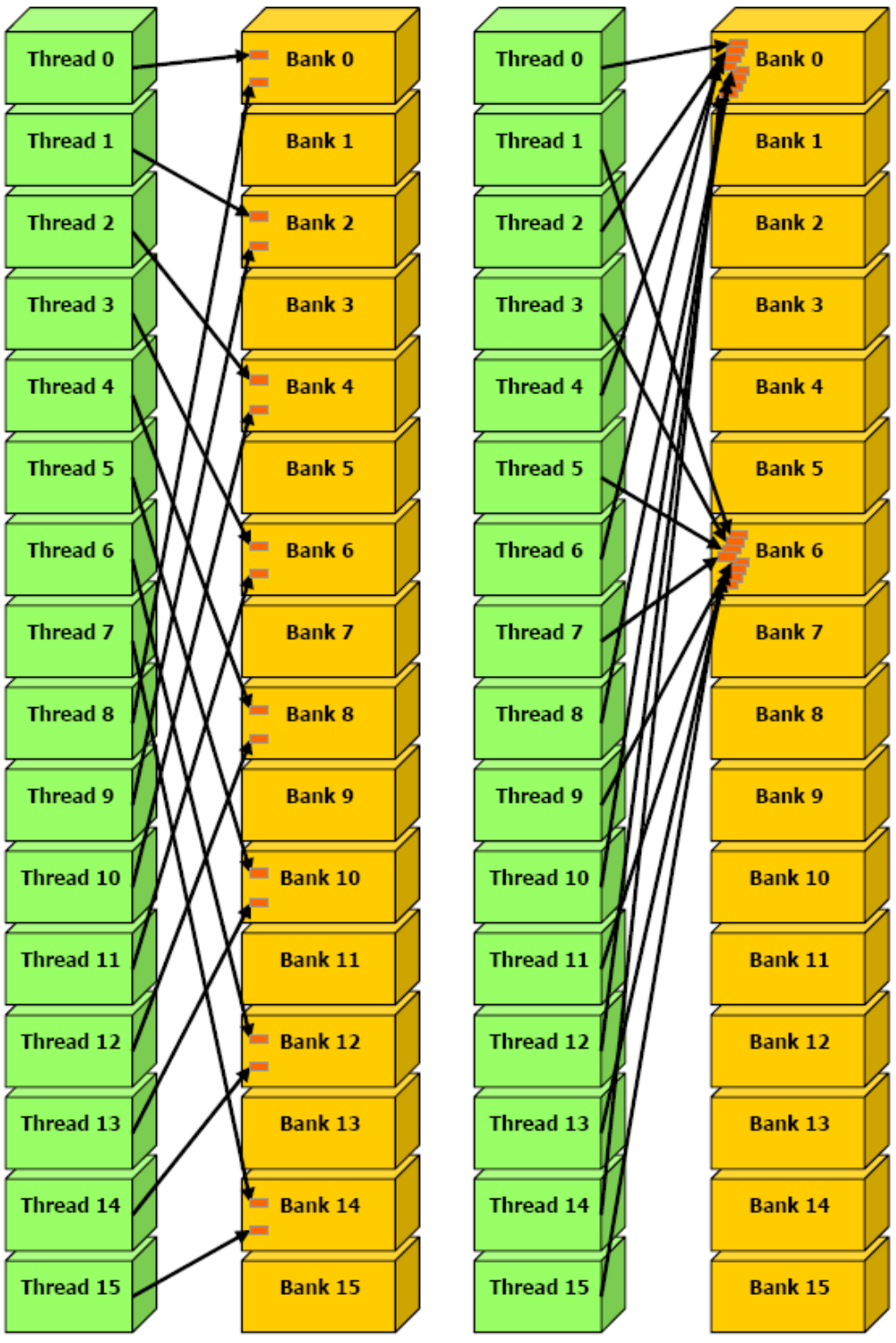
右：随机排列

图 5-4. 无存储体冲突的共享内存访问模式示例



跨度为 3 个 32-位字的线性寻址。

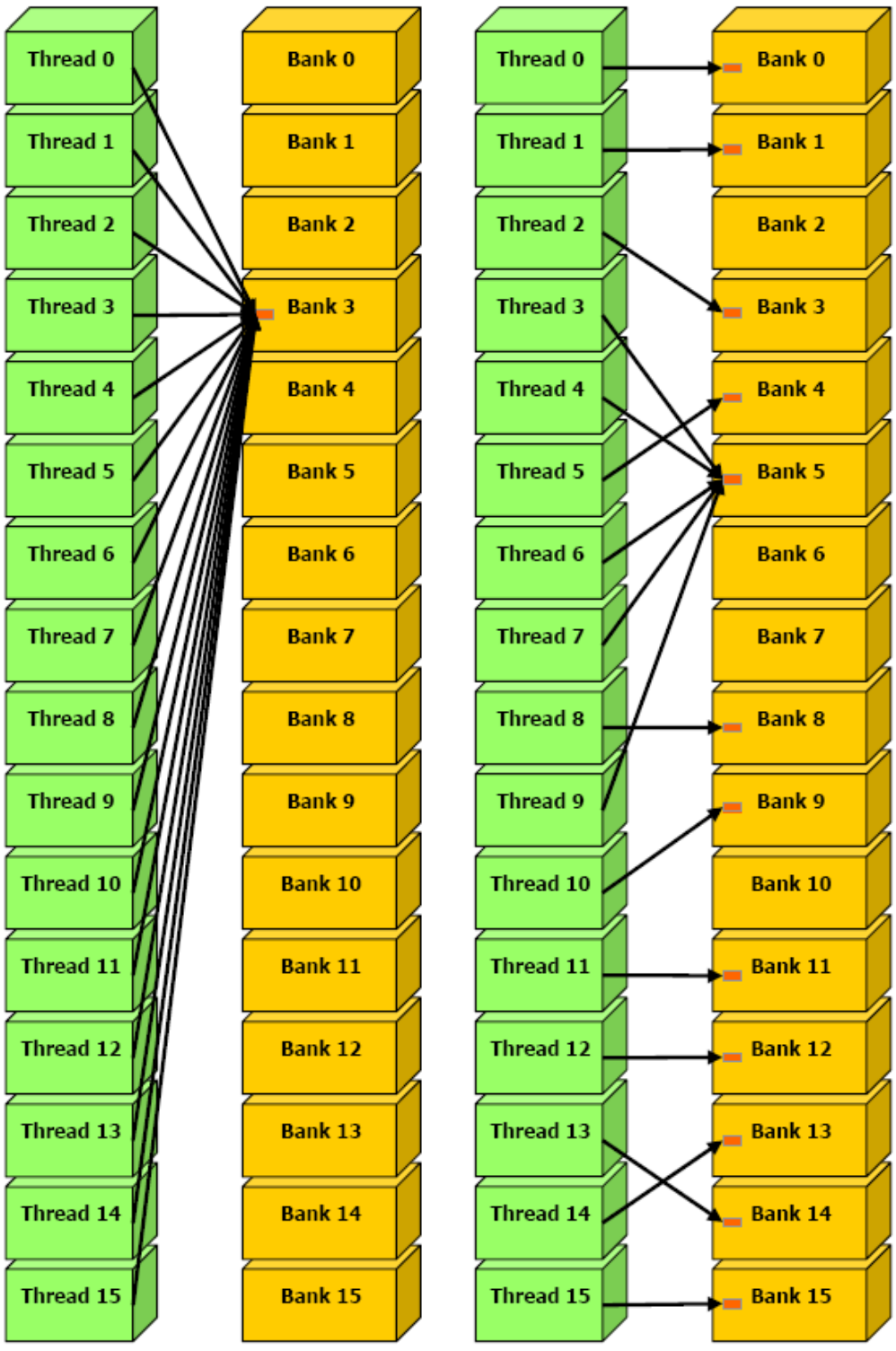
图 5-5. 无存储体冲突的共享内存访问模式示例



左：跨度为 2 个 32-位字的线性寻址将导致 2 路存储体冲突。

右：跨度为 8 个 32-位字的线性寻址将导致 8 路存储体冲突。

图 5-6. 有存储体冲突的共享内存访问模式示例



左：因为所有线程从同一个 32-位字地址读取，所以此访问模式是无冲突的。

右：如果在第一步期间选择“存储体 5”中的字作为广播字，则此访问模式不会导致任何冲突，否则会导致 2 路存储体冲突。

图 5-7. 有广播的共享内存读取访问模式示例

5.1.2.5 寄存器

通常，访问寄存器对于每条指令需要 0 个额外时钟周期，但是由于寄存器写后读依赖关系和寄存器存储体冲突，可能会发生延迟。

只要每个多处理器的活动线程达到 192 个，由写后读依赖关系导致的延迟就可以被隐藏进而忽略掉。

编译器和线程调度器尽可能用最佳方式调度指令，以避免寄存器存储体冲突。当每块中的线程数是 64 的倍数时，可以获得最佳效果。除了遵循此规则之外，应用程序无法直接控制寄存器存储体冲突。特别地，无需将数据打包为 `float4` 或 `int4` 类型。

5.2 每块的线程数

给定每网格的线程总数，设计每块的线程数或网格的块数时应该最大化可用计算资源的利用率。这意味着块的数目应该至少与设备中的多处理器的数目一样多。

进一步看，当每个多处理器仅运行一个线程块时，如果每块没有足够多的线程来掩盖加载延迟的话，则在线程同步期间和设备内存读取期间，每个多处理器将被强制进入空闲状态。因此，最好的方法是每个多处理器上存在两个或多个活动块，以允许等待的块和可以运行的块同时执行。要让这种情况发生，不仅块的数目至少应该是设备中多处理器数目的两倍，而且每块分配的共享内存量至多应该是每个多处理器可用共享内存总量的一半（参见 3.2）。更多线程块以管线方式在设备中流过，并在更大程度上分摊开销。

有了足够大数目的块，每块线程的数目应选择为 warp 大小的倍数，以避免使用未充满的 warp 而浪费计算资源，为 64 的倍数是较好的选择，其原因参见 5.1.2.5。为每块分配更多线程有利于有效的时间分片，但是每块的线程越多，每线程可用的寄存器就越少。如果内核编译后需要的寄存器数目大于执行配置所允许的上限，就可能会阻止内核继续调用。当使用 `--ptxas-options=-v` 选项编译时，可以得到内核编译后需要的寄存器数目（以及本地、共享和常量内存使用情况）。

对于计算能力 1.x 的设备，每线程可用的寄存器数等于：

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

其中，*R* 是每个多处理器的寄存器总数（参见附录 A），*B* 是每个多处理器的活动块数，*T* 是每块的线程数，*ceil(T, 32)*是 *T* 对 32 的最近倍数向上取整后的值。

每块最少含有 64 个线程，并且仅当每个多处理器有多个活动块时才有意义。每块有 192 或 256 个线程比较好，而且通常有足够的寄存器进行编译。

如果想将程序扩展到将来的设备，则每个网格的块数应该至少是 100；想有效利用未来数代的新设备，可以考虑每网格 1000 个块。

每个多处理器的活动 warp 数与活动 warp 的最大数目（参见附录 A）的比率称作多处理器的*占有率（occupancy）*。为了最大化占有率，编译器尽量最小化寄存器使用，而且编程人员需要小心选择执行配置。CUDA 软件开发工具包提供了一个电子表格以帮助编程人员基于共享内存和寄存器要求来选择线程块大小。

5.3 宿主和设备之间的数据传送

设备和设备内存之间的带宽比设备内存和宿主内存之间的带宽高得多。因此，用户应该争取最小化宿主和设备之间的数据传送，例如，将更多代码从宿主迁移到设备，即使这意味着要使用低并行计算来运行内核。中间数据结构可以在设备内存中创建，由设备操作，销毁，而且永远不要被宿主映射，或复制到宿主内存。

另外，由于每次传送都会有开销，所以将许多小的传送合成为一次大的传送要比单独执行每一个小传送要好得多。

最后，使用页面锁定内存时，可以在宿主和设备之间获得较高带宽，详见 4.5.1.2。

5.4 纹理拾取与全局或常量内存读取

与从全局或常量内存中读取相比，通过纹理拾取进行设备内存读取具有下列几个优点：

- 高速缓存，如果要被拾取的纹理在高速缓存中，则可以潜在地获得较高带宽；
- 不受访存模式的约束，而全局或常量内存读取则必须遵循相应访存模式才能获得好的性能（参见 5.1.2.1 和 5.1.2.2）；
- 寻址计算的延迟隐藏得更好，有时候会改善应用程序执行随机访问数据的性能；
- 打包的数据可以在单个操作中广播到多个独立变量中。
- 8-位和 16-位整数输入数据可以有选择地转化为[0.0,1.0]或[-1.0,1.0]区间内的 32 位浮点值（参见 4.3.4.1）。

如果纹理是 CUDA 数组（参见 4.3.4.2），则硬件提供了可能适用于不同应用程序的其它功能，尤其是图像处理：

功能	可用于……	限制
过滤	纹理单元之间快速的低精度插值	仅当纹理参考返回浮点数据时有效
归一化纹理坐标	独立于分辨率的编码	
寻址模式	边界情况的自动处理	只能用于归一化的纹理坐标

然而，在同一内核调用中，纹理高速缓存与全局内存写不保持一致，从而对已经在同一内核中通过全局写而写入的某个地址的纹理拾取将返回不确定的数据。换句话说，仅当此内存位置已经由先前的内核调用或内存复制更新过后，线程才可以通过纹理安全地读取该内存位置，而对于由同一内核调用的同一个或另一个线程更新过后，线程不能够通过纹理安全地读取该位置。这种相关性仅在对线性内存拾取时才存在，因为内核不能对 CUDA 数组实施写入。

5.5 整体性能优化策略

性能优化围绕三个基本策略：

- 最大化并行执行；
- 优化内存使用以获得最大内存带宽；
- 优化指令使用以获得最大指令吞吐量。

最大化并行执行的基础是优化算法结构以让数据尽可能地并行。在算法中，因为一些线程需要同步以互相共享数据，进而破坏了并行性，有两种情况：这些线程属于同一块，这种情况下使用`__syncthreads()`，并通过同一内核调用中的共享内存来共享数据；或者这些线程属于不同块，这种情况下，必须使用两个单独的内核调用通过全局内存来共享数据，一个内核调用写入全局内存，另一个从全局内存读取。

设计了算法的并行之后，应尽可能有效地将其映射到硬件。通过仔细选择每个内核调用的执行配置来完成此操作，详见 5.2。

应用程序还可以通过流的方式在设备上显式地并发执行，以获取较高级别的并行（如 4.5.1.5 所述），就象最大化宿主和设备之间的并发执行一样。

最优化内存使用首先应尽量不进行低带宽的数据传送。这意味着最小化宿主和设备之间的数据传送，详见 5.3，因为这要比在设备和全局内存之间的数据传送的带宽低得多。这也意味着通过最大化设备上共享内存的使用来最小化设备和全局内存之间的数据传送，详见 5.1.2。有时候，最好的优化甚至可能是通过简单地重新计算数据来避免数据传送，该方法十分有效。

如 5.1.2.1、5.1.2.2、5.1.2.3 和 5.1.2.4 所述，对于每种内存类型的不同访存模式，有效带宽可能会相差一个数量级。因此，优化内存使用的下一步是利用最佳的访存模式，尽量优化地组织内存访问。此优化对于全局内存访问尤其重要，因为全局内存访问的带宽很低，且其延迟是数百个时钟周期（参见 5.1.1.3）。另一方面，共享内存访问也常常值得优化，尤其是在具有高度的存储体冲突时。

对于优化指令使用，应该尽量不使用具有低吞吐量的算术指令（参见 5.1.1.1）。这包括在不影响最终结果时用精度换速度，比如使用硬件函数（intrinsic function）替代常规函数（硬件函数在表 B-2 中列出），或使用单精度而不使用双精度。由于设备的 SIMD 本质，所以要特别注意控制流指令，详见 5.1.1.2。

第 6 章　矩阵乘法示例

6.1　概述

计算两个维度分别为*(wA, hA)*和 *(wB, wA)* 的矩阵 *A* 和 *B* 的乘积 *C* 的任务以下列方式分为多个线程：

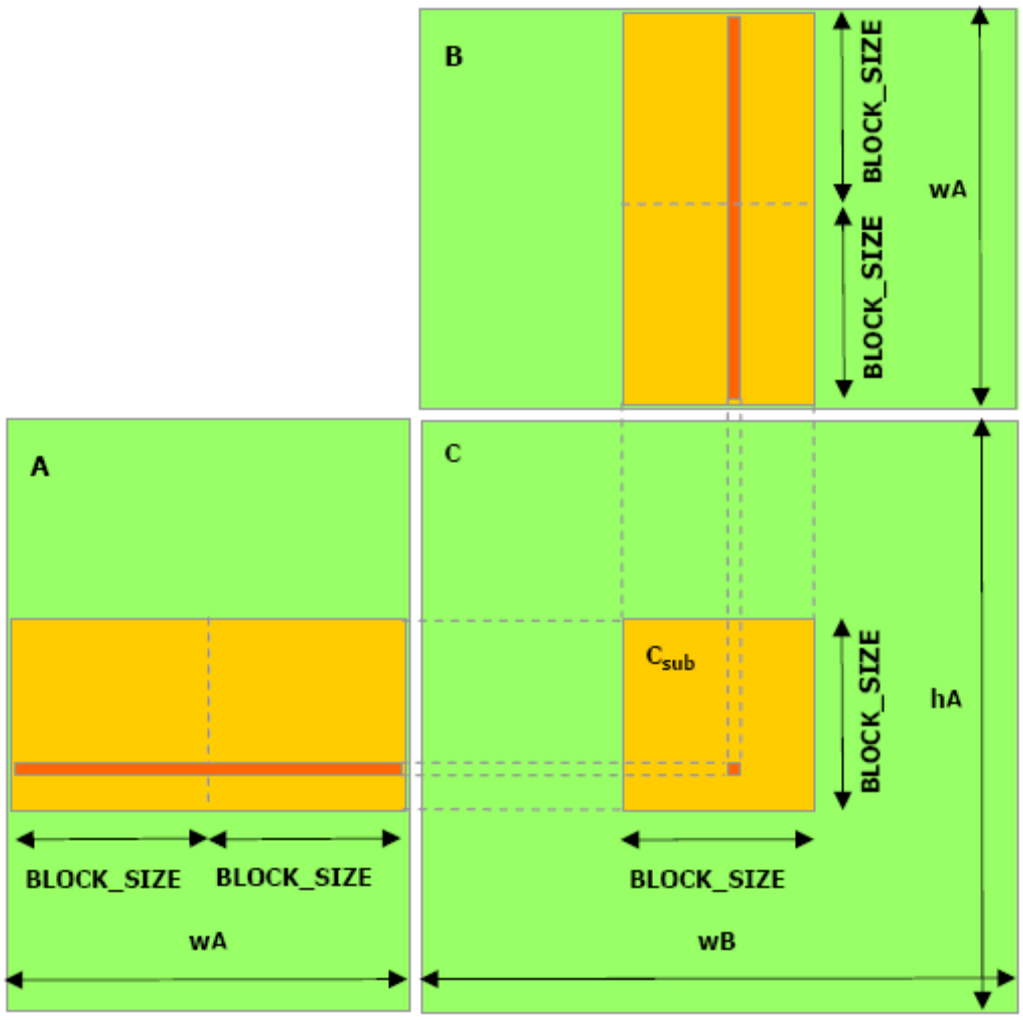
- 每个线程块负责计算 *C* 的一个子方阵 C_{sub} ；
- 块内的每个线程负责计算 C_{sub} 的一个元素。

选择 C_{sub} 的维度 *block_size* 等于 16，以便每块的线程数是 warp 大小的倍数（参见 5.2），且低于每块的最大线程数（参见附录 A）。

如图 6-1 所示，C_{sub} 等于两个矩形矩阵的乘积：*A* 的子矩阵（维度为*(wA, block_size)*）与 C_{sub} 具有相同的行索引，*B* 的子矩阵（维度为*(block_size, wA)*）与 C_{sub} 具有相同的列索引。为了适应设备的资源，这两个矩形矩阵可根据需要划分为许多维度为 *block_size* 的方阵，并且 C_{sub} 计算为这些方阵的乘积之和。其中每个乘积的执行过程是：首先使用每线程加载每个方阵的一个元素，将两个相应的方阵从全局内存加载到共享内存，然后让每个线程计算结果方阵的一个元素。每一线程将其中每个乘积的结果累计到寄存器中，执行完毕后，将结果写入全局内存。

通过以这种方式分块计算，我们可以有效利用快速的共享内存，并节省许多全局内存带宽，因为 *A* 和 *B* 仅从全局内存读取*(wA / block_size)*次。

然而，编写此示例是为了清楚地说明各种 CUDA 编程原则，并非是为一般的矩阵乘法提供高性能的计算方法，所以在实际应用中，不应按此示例编写矩阵相乘算法。



每一线程块计算 *C* 的一个子矩阵 C_{sub}。块内的每一线程计算 C_{sub} 的一个元素。

图 6-1. 矩阵乘法

6.2 源码清单

```
// Thread block size
#define BLOCK_SIZE 16

// Forward declaration of the device multiplication function
__global__ void Muld(float*, float*, int, int, float*);

// Host multiplication function
// Compute C = A * B
//   hA is the height of A
//   wA is the width of A
//   wB is the width of B
void Mul(const float* A, const float* B, int hA, int wA, int wB,
         float* C)
{
    int size;

    // Load A and B to the device
    float* Ad;
    size = hA * wA * sizeof(float);
    cudaMalloc((void**)&Ad, size);
    cudaMemcpy(Ad, A, size, cudaMemcpyHostToDevice);
    float* Bd;
    size = wA * wB * sizeof(float);
    cudaMalloc((void**)&Bd, size);
    cudaMemcpy(Bd, B, size, cudaMemcpyHostToDevice);

    // Allocate C on the device
    float* Cd;
    size = hA * wB * sizeof(float);
    cudaMalloc((void**)&Cd, size);

    // Compute the execution configuration assuming
    // the matrix dimensions are multiples of BLOCK_SIZE
    dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
    dim3 dimGrid(wB / dimBlock.x, hA / dimBlock.y);

    // Launch the device computation
    Muld<<<dimGrid, dimBlock>>>(Ad, Bd, wA, wB, Cd);

    // Read C from the device
    cudaMemcpy(C, Cd, size, cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(Ad);
    cudaFree(Bd);
    cudaFree(Cd);
}
```

```
// Device multiplication function called by Mul()
// Compute C = A * B
//   wA is the width of A
//   wB is the width of B
__global__ void Muld(float* A, float* B, int wA, int wB, float* C)
{
    // Block index
    int bx = blockIdx.x;
    int by = blockIdx.y;

    // Thread index
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // Index of the first sub-matrix of A processed by the block
    int aBegin = wA * BLOCK_SIZE * by;

    // Index of the last sub-matrix of A processed by the block
    int aEnd   = aBegin + wA - 1;

    // Step size used to iterate through the sub-matrices of A
    int aStep  = BLOCK_SIZE;

    // Index of the first sub-matrix of B processed by the block
    int bBegin = BLOCK_SIZE * bx;

    // Step size used to iterate through the sub-matrices of B
    int bStep  = BLOCK_SIZE * wB;

    // The element of the block sub-matrix that is computed
    // by the thread
    float Csub = 0;

    // Loop over all the sub-matrices of A and B required to
    // compute the block sub-matrix
    for (int a = aBegin, b = bBegin;
             a <= aEnd;
             a += aStep, b += bStep) {

        // Shared memory for the sub-matrix of A
        __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];

        // Shared memory for the sub-matrix of B
        __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

        // Load the matrices from global memory to shared memory;
        // each thread loads one element of each matrix
        As[ty][tx] = A[a + wA * ty + tx];
        Bs[ty][tx] = B[b + wB * ty + tx];

        // Synchronize to make sure the matrices are loaded
        __syncthreads();

        // Multiply the two matrices together;
        // each thread computes one element
        // of the block sub-matrix
        for (int k = 0; k < BLOCK_SIZE; ++k)
```

```
            Csub += As[ty][k] * Bs[k][tx];

        // Synchronize to make sure that the preceding
        // computation is done before loading two new
        // sub-matrices of A and B in the next iteration
        __syncthreads();
    }

    // Write the block sub-matrix to global memory;
    // each thread writes one element
    int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
    C[c + wB * ty + tx] = Csub;
}
```

6.3 源码攻略

源码包含下列两个函数：

- `Mul()`，作为`Muld()`的包装器的宿主函数。
- `Muld()`，在设备上执行矩阵乘法的内核。

6.3.1 Mul()

`Mul()`接受下列输入：

- 指向宿主内存中 *A* 和 *B* 的两个指针，
- *A* 的高度和宽度，*B* 的宽度，
- 指向应该写入宿主内存中 *C* 的指针。

`Mul()`执行下列操作：

- 使用`cudaMalloc()`将足够的全局内存分配给 *A*、*B* 和 *C*；
- 使用`cudaMemcpy()`将 *A* 和 *B* 从宿主内存复制到全局内存；
- 调用`Muld()`在设备上计算 *C*；
- 使用`cudaMemcpy()`将 *C* 从全局内存复制回宿主内存；
- 使用`cudaFree()`释放为 *A*、*B* 和 *C* 分配的全局内存。

6.3.2 Muld()

除了指针指向设备内存而非宿主内存之外，`Muld()`与`Mul()`具有相同的输入参数。

对于每个块，`Muld()`迭代处理所有需要计算 C_{sub} 的 *A* 和 *B* 的子矩阵。在每次迭代中，此函数：

- 将 *A* 的一个子矩阵和 *B* 的一个子矩阵从全局内存加载到共享内存中；
- 同步以确保两个子矩阵都由块内的所有线程完全加载；
- 计算两个子集的乘积并将其加到上一次迭代期间获得的乘积中；

- 再次同步以确保在开始下一次迭代之前两个子集的乘积已经完成。

一旦所有的子矩阵全部处理完毕，C_{sub} 也就计算完毕，`Muld()`将其写到全局内存中。

按照 5.1.2.1 和 5.1.2.4 所述，`Muld()`的编写原则是为了最大化内存性能。

假设 `wA` 和 `wB` 是 16 的倍数（如 5.1.2.1 所建议的），可以确保全局内存合并，因为 `a`、`b` 和 `c` 都是 `BLOCK_SIZE` 的倍数，`BLOCK_SIZE` 等于 16。

对于每个半 warp，也没有任何共享内存存储体冲突，所有线程的 `ty` 和 `k` 都是相同的，`tx` 在 0 到 15 之间变化，因此对于 `As[ty][tx]`、`Bs[ty][tx]`和 `Bs[k][tx]`的访存，每个线程都访问一个不同的存储体，而对于 `As[ty][k]`的访存，每个线程都访问同一个存储体。

附录 A　技术规格

具有 1.x 计算能力（参见 3.3）的所有设备都遵循本附录描述的技术规格。

原子函数仅可用于计算能力 1.1 的设备（参见 4.4.6）。

下表给出支持 CUDA 的所有设备的多处理器数目和计算能力：

	多处理器数目	计算能力
GeForce 8800 Ultra, 8800 GTX	16	1.0
GeForce 8800 GT	14	1.1
GeForce 8800M GTX	12	1.1
GeForce 8800 GTS	12	1.0
GeForce 8800M GTS	8	1.1
GeForce 8600 GTS, 8600 GT, 8700M GT, 8600M GT, 8600M GS	4	1.1
GeForce 8500 GT, 8400 GS, 8400M GT, 8400M GS	2	1.1
GeForce 8400M G	1	1.1
Tesla S870	4x16	1.0
Tesla D870	2x16	1.0
Tesla C870	16	1.0
Quadro Plex 1000 Model S4	4x16	1.0
Quadro Plex 1000 Model IV	2x16	1.0
Quadro FX 5600	16	1.0
Quadro FX 4600	12	1.0
Quadro FX 1700, FX 570, NVS 320M, FX 1600M, FX 570M	4	1.1
Quadro FX 370, NVS 290, NVS 140M, NVS 135M, FX 360M	2	1.1
Quadro NVS 130M	1	1.1

设备内存的时钟频率和总量可以使用运行时来查询（参见 4.5.2.2 和 4.5.3.2）。

A.1 通用规范

- 每线程块最多包含 512 线程；
- 线程块的 x 维、y 维和 z 维的最大大小分别是 512、512 和 64；
- 线程块网格的每个维的最大大小是 65535；
- warp 的大小是 32 个线程；
- 每个多处理器寄存器的数目是 8192；
- 每个多处理器可用的共享内存量是 16KB，分成 16 个存储体（参见 5.1.2.4）。
- 常量内存的总量是 64KB；
- 常量内存的高速缓存工作集是每个多处理器 8KB；
- 纹理内存的高速缓存工作集是每个多处理器 8KB；
- 每个多处理器的活动块的最大数目是 8；
- 每个多处理器的活动 warp 的最大数目是 24；
- 每个多处理器的活动线程的最大数目是 768；
- 对于绑定到一维 CUDA 数组的纹理参考，最大宽度是 2^{13}；
- 对于绑定到二维 CUDA 数组的纹理参考，最大宽度是 2^{16}，最大高度是 2^{15}；
- 对于绑定到线性内存的纹理参考，最大宽度是 2^{27}；
- 内核大小的限制为 200 万条原生指令；
- 每个多处理器由 8 个处理器组成，所以在 4 个时钟周期内多处理器能够处理 warp 的 32 个线程。

A.2 浮点标准

计算设备遵循单精度二进制浮点算术的 IEEE-754 标准，但存在下列不同：

- 加法和乘法一般被合并到单个乘加指令（FMAD）中；
- 除法是通过非标准的方式求倒数来实现的；
- 平方根是通过非标准的方式求平方根倒数来实现的；
- 对于加法和乘法，通过静态取整模式仅支持取整到最接近的偶数和向零取整；不支持直接向正/负无穷大取整；
- 不存在动态可配置的取整模式；
- 不支持非规格化数；浮点算术和比较指令在浮点操作之前将非规格化操作数转换为零；
- 下溢结果设为零；
- 没有用于检测浮点异常的机制，而且始终隐藏掩码浮点异常，但当异常发生时，掩码响应是符合标准的；

- Signaling NaN 不受支持。
- 包含一个或多个 NaN 输入的操作的结果是位模式为 0x7fffffff 的 Quiet NaN。注意，按照 IEEE-754R 标准，如果 `fminf()`、`fmin()`、`fmaxf()`或 `fmax()`的输入参数之一是 NaN，而另一个不是 NaN，则结果是非 NaN 的那个参数。

根据 IEEE-754 标准，当浮点值超出整数格式的范围时，从浮点值到整数值的转换保留为不确定的。对于计算设备，其做法是将其夹合到支持范围的终点。这和 x86 体系结构的做法是不同的。

附录 B　数学函数

B.1 中的函数由宿主和设备函数使用，而 B.2 中的函数仅在设备函数中使用。

B.1　共用运行时组件

下表 B-1 列出了 CUDA 运行时库支持的所有数学标准库函数。它还指出每个函数在设备上执行时的误差界。这些误差界还适用于当宿主不支持此函数时，在宿主上执行此函数的情况。这些误差界不保证在所有情况下均成立，因为虽进行了广泛测试但还没有穷尽所有设备。

加法和乘法是与 IEEE 兼容的，因此最大误差为 0.5 ulp。但是，加法和乘法通常合并到单个乘加指令(FMAD)中，从而截断了乘法的中间结果。

推荐的做法是将浮点操作数取整为整数，结果为浮点数应使用 `rintf()`，而非 `roundf()`。原因是 `roundf()`映射为 8 指令序列，而 `rintf()`映射为单个指令。`truncf()`、`ceilf()`和 `floorf()`也都映射为单个指令。

CUDA 运行时库还支持映射为单个指令的整数 `min()`和 `max()`。

表 B-1. 有最大 ULP 误差的数学标准库函数

函数	最大 ulp 误差
`x/y`	2（全范围）
`1/x`	1（全范围）
`1/sqrtf(x)` `rsqrtf(x)`	2（全范围）
`sqrtf(x)`	3（全范围）
`cbrtf(x)`	1（全范围）
`hypotf(x)`	3（全范围）
`expf(x)`	2（全范围）
`exp2f(x)`	2（全范围）
`exp10f(x)`	2（全范围）
`expm1f(x)`	1（全范围）
`logf(x)`	1（全范围）
`log2f(x)`	3（全范围）
`log10f(x)`	3（全范围）
`log1pf(x)`	2（全范围）

sinf(x)	2（全范围）
cosf(x)	2（全范围）
tanf(x)	4（全范围）
sincosf(x,sptr,cptr)	2（全范围）
asinf(x)	4（全范围）
acosf(x)	3（全范围）
atanf(x)	2（全范围）
atan2f(y,x)	3（全范围）
sinhf(x)	3（全范围）
coshf(x)	2（全范围）
tanhf(x)	2（全范围）
asinhf(x)	3（全范围）
acoshf(x)	4（全范围）
atanhf(x)	3（全范围）
powf(x,y)	16（全范围）
erff(x)	4（全范围）
erfcf(x)	8（全范围）
lgammaf(x)	6（外间距 -10.001 ... -2.264；内间距更大）
tgammaf(x)	11（全范围）
fmaf(x,y,z)	0（全范围）
frexpf(x,exp)	0（全范围）
ldexpf(x,exp)	0（全范围）
scalbnf(x,n)	0（全范围）
scalblnf(x,l)	0（全范围）
logbf(x)	0（全范围）
ilogbf(x)	0（全范围）
fmodf(x,y)	0（全范围）
remainderf(x,y)	0（全范围）
remquof(x,y,iptr)	0（全范围）
modff(x,iptr)	0（全范围）
fdimf(x,y)	0（全范围）
truncf(x)	0（全范围）

roundf(x)	0（全范围）
rintf(x)	0（全范围）
nearbyintf(x)	0（全范围）
ceilf(x)	0（全范围）
floorf(x)	0（全范围）
lrintf(x)	0（全范围）
lroundf(x)	0（全范围）
llrintf(x)	0（全范围）
llroundf(x)	0（全范围）
signbit(x)	不适用
isinf(x)	不适用
isnan(x)	不适用
isfinite(x)	不适用
copysignf(x,y)	不适用
fminf(x,y)	不适用
fmaxf(x,y)	不适用
fabsf(x)	不适用
nanf(cptr)	不适用
nextafterf(x,y)	不适用

B.2 设备运行时组件

表 B-2 列出仅在设备代码中支持的硬件函数。这些硬件函数的误差界是特定于 GPU 的。这些函数不太精确，但却是表 B-1 中一些函数的快速版本；它们具有相同名称，但加上了前缀__（比如__sinf(x)）。

__fadd_rz(x,y) 使用向零取整的取整模式计算浮点参数 x 和 y 的和。

__fmul_rz(x,y) 使用向零取整的取整模式计算浮点参数 x 和 y 的乘积。

普通浮点除法和__fdividef(x,y)有相同的精度，但对于 $2^{126} < y < 2^{128}$，__fdividef(x,y)的结果为零，而普通除法可以获得正确结果，精度见表 B-1。另外，对于 $2^{126} < y < 2^{128}$，如果 x 是无穷大，则__fdividef(x,y)返回 NaN（作为零乘无穷大的结果），而普通除法则返回无穷大。

__[u]mul24(x,y) 计算整数参数 x 和 y 的 24 个最低有效位数据的乘积，并返回结果的 32 个最低有效位。忽略 x 和 y 的 8 个最高位。

__[u]mulhi(x,y) 计算整数参数 x 和 y 的乘积，并传递 64-位结果的 32 个最高位。

__[u]mul64hi(x,y) 计算 64 位整数参数 x 和 y 的乘积，并传递 128-位结果的 64 个最高位。

如果 x 小于 0，则__saturate(x)返回 0，如果 x 大于 1，则返回 x。

__[u]sad(x,y,z)（绝对误差和）返回整数参数 z 与整数参数 x 和 y 之差绝对值的和。

__clz(x) 返回介于并包含 0 到 32 的数字，并且从整数参数 x 的最高位（例如 31 位）开始填充连续零。

__clzll(x) 返回介于并包含 0 到 64 的数字，并且在整数参数 x 的最高位（例如 63 位）开始填充连续零。

__ffs(x) 返回整数参数 x 中第一（最低）位的位置。最低位是位置 1。如果 x 是 0，则__ffs(x)返回 0。这与 Linux 函数 ffs 完全相同。

__ffsll(x) 返回 64-位整数参数 x 中第一（最低）位的位置。最低位是位置 1。如果 x 是 0，则__ffsll()返回 0。这与 Linux 函数 ffsll 完全相同。

表 B-2. CUDA 运行时库支持的硬件函数以及对于计算能力 1.x 的设备的各自误差界

函数	误差界
__fadd_rz(x,y)	与 IEEE 兼容。
__fmul_rz(x,y)	与 IEEE 兼容。
__fdividef(x,y)	如果 y 在[2^{-126}, 2^{126}]区间内，则最大 ulp 误差是 2。
__expf(x)	最大 ulp 误差是 2 + floor(abs(1.16 * x))。
__exp10f(x)	最大 ulp 误差是 2 + floor(abs(2.95 * x))。
__logf(x)	如果 x 在[0.5, 2]区间内，则最大绝对误差是 $2^{-21.41}$，否则，最大 ulp 误差是 3。
__log2f(x)	如果 x 在[0.5, 2]区间内，则最大绝对误差是 2^{-22}，否则，最大 ulp 误差是 2。
__log10f(x)	如果 x 在[0.5, 2]区间内，则最大绝对误差是 2^{-24}，否则，最大 ulp 误差是 3。
__sinf(x)	如果 x 在[-π, π]区间内，则最大绝对误差是 $2^{-21.41}$，否则会更大。
__cosf(x)	如果 x 在[-π, π]区间内，则最大绝对误差是 $2^{-21.19}$，否则会更大。
__sincosf(x,sptr,cptr)	与 sinf(x)和 cosf(x)一样。
__tanf(x)	其实现来自__sinf(x) * (1 / __cosf(x))。
__powf(x, y)	其实现来自 exp2f(y * __log2f(x))。
__mul24(x,y) __umul24(x,y)	不适用
__mulhi(x,y) __umulhi(x,y)	不适用
__int_as_float(x)	不适用
__float_as_int(x)	不适用
__saturate(x)	不适用
__sad(x,y,z) __usad(x,y,z)	不适用
__clz(x)	不适用
__ffs(x)	不适用

附录 C　原子函数

原子函数仅可用于设备函数中。

C.1　算术函数

C.1.1　atomicAdd()

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

读取位于全局内存中的地址 address 的 32-位字 old，计算(old + val)，并将结果存回全局内存的同一地址。这三个操作在一个原子事务处理中执行。函数返回 old。

C.1.2　atomicSub()

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

读取位于全局内存中的地址 address 的 32-位字 old，计算(old - val)，并将结果存回全局内存的同一地址。这三个操作在一个原子事务处理中执行。函数返回 old。

C.1.3　atomicExch()

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

读取位于全局内存中的地址 address 的 32-位字 old，并将 val 存回全局内存的同一地址。这两个操作在一个原子事务处理中执行。函数返回 old。

C.1.4 atomicMin()

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

读取位于全局内存中的地址 address 的 32-位字 old，计算 old 和 val 的最小值，并将结果存回全局内存的同一地址。这三个操作在一个原子事务处理中执行。函数返回 old。

C.1.5 atomicMax()

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

读取位于全局内存中的地址 address 的 32-位字 old，计算 old 和 val 的最大值，并将结果存回全局内存的同一地址。这三个操作在一个原子事务处理中执行。函数返回 old。

C.1.6 atomicInc()

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

读取位于全局内存中的地址 address 的 32-位字 old，计算 ((old >= val) ? 0 : (old+1))，并将结果存回全局内存的同一地址。这三个操作在一个原子事务处理中执行。函数返回 old。

C.1.7 atomicDec()

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

读取位于全局内存中的地址 address 的 32-位字 old，计算 (((old == 0) | (old > val)) ? val : (old-1))，并将结果存回全局内存的同一地址。这三个操作在一个原子事务处理中执行。函数返回 old。

C.1.8 atomicCAS()

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

读取位于全局内存中的地址 address 的 32-位字 old，计算 (old == compare ? val : old)，并将结果存回全局内存的同一地址。这三个操作在一个原子事务处理中执行。函数返回 old（比较并交换）。

C.2 位函数

C.2.1 atomicAnd()

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

读取位于全局内存中的地址 address 的 32-位字 old，计算(old & val)，并将结果存回全局内存的同一地址。这三个操作在一个原子事务处理中执行。函数返回 old。

C.2.2 atomicOr()

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

读取位于全局内存中的地址 address 的 32-位字 old，计算(old | val)，并将结果存回全局内存的同一地址。这三个操作在一个原子事务处理中执行。函数返回 old。

C.2.3 atomicXor()

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

读取位于全局内存中的地址 address 的 32-位字 old，计算(old ^ val)，并将结果存回全局内存的同一地址。这三个操作在一个原子事务处理中执行。函数返回 old。

附录 D　运行时 API 参考

运行时 API 有两个级别。

低层 API（cuda_runtime_api.h）是 C 风格的接口，不需要使用 nvcc 编译。

高层 API（cuda_runtime.h）是 C++风格的接口，构建于低层 API 之上。它使用重载、引用和默认参数包装了一些低层 API 例程。这些包装器可以在 C++代码中使用，并可以使用 C++编译器进行编译。高层 API 还具有一些特定于 CUDA 的包装器，其中包装了处理符号、纹理和设备函数的低级功能。这些包装器需要使用 nvcc，因为它们依赖于要由编译器生成的代码（参见 4.2.5）。例如 4.2.3 中描述的调用内核的执行配置语法，它仅可用于使用 nvcc 编译的源码中。

D.1　设备管理

D.1.1　cudaGetDeviceCount()

```
cudaError_t cudaGetDeviceCount(int* count);
```

在*count 中返回可供执行的计算能力大于或等于 1.0 的设备数。如果没有这样的设备，则 cudaGetDeviceCount()返回 1，设备 0 仅支持设备仿真模式，且其计算能力小于 1.0。

D.1.2　cudaSetDevice()

```
cudaError_t cudaSetDevice(int dev);
```

将 dev 记录为活动宿主线程执行设备代码所在的设备。

D.1.3　cudaGetDevice()

```
cudaError_t cudaGetDevice(int* dev);
```

在*dev 中返回活动宿主线程执行设备代码所在的设备。

D.1.4 cudaGetDeviceProperties()

```
cudaError_t cudaGetDeviceProperties(struct cudaDeviceProp* prop,
                                    int dev);
```

在*prop 中返回设备 dev 的属性。cudaDeviceProp 结构定义为：

```
struct cudaDeviceProp {
  char   name[256];
  size_t totalGlobalMem;
  size_t sharedMemPerBlock;
  int    regsPerBlock;
  int    warpSize;
  size_t memPitch;
  int    maxThreadsPerBlock;
  int    maxThreadsDim[3];
  int    maxGridSize[3];
  size_t totalConstMem;
  int    major;
  int    minor;
  int    clockRate;
  size_t textureAlignment;
};
```

其中：

- name 是用于标识设备的 ASCII 字符串；
- totalGlobalMem 是设备上可用的全局内存总量，单位为字节；
- sharedMemPerBlock 是每块可用的共享内存总量，单位为字节；
- regsPerBlock 是每块可用的寄存器总数；
- warpSize 是 warp 大小；
- memPitch 是 D.5 中的内存复制函数允许的最大节距，包括 cudaMallocPitch()（参见 D.5.2）分配的内存区域；
- maxThreadsPerBlock 是每块的最大线程数；
- maxThreadsDim[3]是每块每个维度的最大大小；
- maxGridSize[3]是网格的每个维度的最大大小；
- totalConstMem 是设备上可用的常量内存总量，单位为字节；
- major 和 minor 是定义设备计算能力的主要和次要修订号；
- clockRate 是时钟频率，单位为千赫；
- textureAlignment 是 4.3.4.3 中介绍的对齐要求；已经对齐为 textureAlign 个字节的纹理基址不再需要那种应用于纹理拾取的偏移。

D.1.5 cudaChooseDevice()

```
cudaError_t cudaChooseDevice(int* dev,
                             const struct cudaDeviceProp* prop);
```

在*dev 中返回其属性与*prop 最佳匹配的设备。

D.2 线程管理

D.2.1 cudaThreadSynchronize()

```
cudaError_t cudaThreadSynchronize(void);
```

阻塞，直到设备完成所有先前请求的任务为止。如果先前任务之一失败，则 cudaThreadSynchronize()返回错误。

D.2.2 cudaThreadExit()

```
cudaError_t cudaThreadExit(void);
```

显式地清除与调用宿主线程相关联的所有与运行时相关的资源。任何后续 API 调用会重新初始化运行时。cudaThreadExit()在宿主线程退出时隐式调用。

D.3 流管理

D.3.1 cudaStreamCreate()

```
cudaError_t cudaStreamCreate(cudaStream_t* stream);
```

创建流。

D.3.2 cudaStreamQuery()

```
cudaError_t cudaStreamQuery(cudaStream_t stream);
```

如果流中的所有操作都已完成，返回 cudaSuccess，否则，返回 cudaErrorNotReady。

D.3.3 cudaStreamSynchronize()

```
cudaError_t cudaStreamSynchronize(cudaStream_t stream);
```

阻塞，直到设备完成流中的所有操作为止。

D.3.4 cudaStreamDestroy()

```
cudaError_t cudaStreamDestroy(cudaStream_t stream);
```

销毁流。

D.4 事件管理

D.4.1 cudaEventCreate()

```
cudaError_t cudaEventCreate(cudaEvent_t* event);
```

创建事件。

D.4.2 cudaEventRecord()

```
cudaError_t cudaEventRecord(cudaEvent_t event, CUstream stream);
```

记录事件。如果 stream 非 0，则在流中的所有先前操作都已完成之后记录事件；否则，在 CUDA 上下文中的所有先前操作都已完成之后记录事件。因为此操作是异步的，所以必须使用 cudaEventQuery()和/或 cudaEventSynchronize()确定事件实际已经记录的时间。

如果 cudaEventRecord()已经调用，但事件尚未记录，则此函数返回 cudaErrorInvalidValue。

D.4.3 cudaEventQuery()

```
cudaError_t cudaEventQuery(cudaEvent_t event);
```

如果事件实际已经记录，返回 cudaSuccess，否则，返回 cudaErrorNotReady。如果 cudaEventRecord()尚未在此事件上调用，则此函数返回 cudaErrorInvalidValue。

D.4.4 cudaEventSynchronize()

```
cudaError_t cudaEventSynchronize(cudaEvent_t event);
```

阻塞，直到事件实际已经记录为止。如果 cudaEventRecord()尚未在此事件上调用，则此函数返回 cudaErrorInvalidValue。

D.4.5 cudaEventDestroy()

```
cudaError_t cudaEventDestroy(cudaEvent_t event);
```

销毁事件。

D.4.6 cudaEventElapsedTime()

```
cudaError_t cudaEventElapsedTime(float* time,
                                 cudaEvent_t start,
                                 cudaEvent_t end);
```

计算两个事件之间用去的时间（以毫秒计，分辨率约为 0.5 微秒）。如果任一事件尚未记录，则

此函数返回 cudaErrorInvalidValue。如果任一事件已经使用非 0 流记录，则此结果是不确定的。

D.5 内存管理

D.5.1 cudaMalloc()

```
cudaError_t cudaMalloc(void** devPtr, size_t count);
```

在设备上分配 count 个字节的线性内存，并在*devPtr 中返回指向已分配内存的指针。已分配内存相应对齐为任何种类的变量。分配的内存是未清除的。如果失败，则 cudaMalloc()返回 cudaErrorMemoryAllocation。

D.5.2 cudaMallocPitch()

```
cudaError_t cudaMallocPitch(void** devPtr,
                            size_t* pitch,
                            size_t widthInBytes,
                            size_t height);
```

在设备上分配至少 widthInBytes*height 个字节的线性内存，并在*devPtr 中返回指向已分配内存的指针。函数可以填补分配以确保任何给定行中的相应指针将继续满足对齐要求，以便当地址在行之间更新时进行内存合并（参见 5.1.2.1）。由 cudaMallocPitch()在*pitch 中返回的节距是分配的字节数宽度。节距的设计用途是作为单独的分配参数，用于计算 2D 数组中的地址。给定类型 T 的数组元素的行和列，地址计算为：

```
T* pElement = (T*)((char*)BaseAddress + Row * pitch) + Column;
```

对于 2D 数组的分配，建议编程人员考虑使用 cudaMallocPitch()执行节距分配。由于硬件中的节距对齐限制，如果应用程序将在设备内存的不同区域（不管是线性内存还是 CUDA 数组）之间执行 2D 内存复制，这时更应该使用此函数。

D.5.3 cudaFree()

```
cudaError_t cudaFree(void* devPtr);
```

释放由 devPtr 指向的内存空间，devPtr 必须已经由对 cudaMalloc()或 cudaMallocPitch()的调用返回。否则，如果 cudaFree(devPtr)以前已经调用，则返回错误。如果 devPtr 为 0，则不执行任何操作。如果失败，则 cudaFree()返回 cudaErrorInvalidDevicePointer。

D.5.4 cudaMallocArray()

```
cudaError_t cudaMallocArray(struct cudaArray** array,
                            const struct cudaChannelFormatDesc* desc,
                               size_t width, size_t height);
```

按照 desc（cudaChannelFormatDesc 结构）分配 CUDA 数组，并在*array 中返回新 CUDA 数组的句柄。cudaChannelFormatDesc 的介绍参见 4.3.4。

D.5.5 cudaFreeArray()

```
cudaError_t cudaFreeArray(struct cudaArray* array);
```

释放 CUDA 数组 array。如果 array 为 0，则不执行任何操作。

D.5.6 cudaMallocHost()

```
cudaError_t cudaMallocHost(void** hostPtr, size_t size);
```

分配 size 个字节的页面锁定的、可供设备访问的宿主内存。驱动程序跟踪使用此函数分配的虚拟内存范围，并自动加速对 cudaMemcpy*()等函数的调用。因为此内存可由设备直接访问，所以与使用 malloc()等函数获得的可分页内存相比，它在进行读取或写入时具有高得多的带宽。使用 cudaMallocHost()分配过量的内存可能降低系统性能，因为它降低了系统可用于分页的内存量。因此，最好节约使用此函数分配中转区进行宿主和设备之间的数据交换。

D.5.7 cudaFreeHost()

```
cudaError_t cudaFreeHost(void* hostPtr);
```

释放 hostPtr 指向的内存空间，hostPtr 必须已由先前对 cudaMallocHost()的调用返回。

D.5.8 cudaMemset()

```
cudaError_t cudaMemset(void* devPtr, int value, size_t count);
```

使用常量字节值 value 填充 devPtr 指向的内存区域的前 count 个字节。

D.5.9 cudaMemset2D()

```
cudaError_t cudaMemset2D(void* dstPtr, size_t pitch,
                         int value, size_t width, size_t height);
```

设置给指定值 value 一个 dstPtr 指向的矩阵（height 行，每行 width 个字节）。pitch 是由 dstPtr 指向的 2D 数组在内存中所占的字节数宽度，其中包括添加到每行尾的任何填补（参见 D.5.2）。当 pitch 已经由 cudaMallocPitch()传回时，此函数执行得最快。

D.5.10 cudaMemcpy()

```
cudaError_t cudaMemcpy(void* dst, const void* src,
                       size_t count,
                       enum cudaMemcpyKind kind);
cudaError_t cudaMemcpyAsync(void* dst, const void* src,
                            size_t count,
                            enum cudaMemcpyKind kind,
                            cudaStream_t stream);
```

将 count 个字节从由 src 指向的内存区域复制到由 dst 指向的内存区域，其中 kind 是 cudaMemcpyHostToHost、cudaMemcpyHostToDevice、cudaMemcpyDeviceToHost 或 cudaMemcpyDeviceToDevice 之一，指定复制的方向。内存区域不可以重叠。使用与复制方向不匹配的 dst 和 src 指针调用 cudaMemcpy()将导致不确定的行为。

cudaMemcpyAsync()是异步的，可以通过传递非零 stream 参数与流相关联。它仅适用于页面锁定的宿主内存，如果传递指向可分页内存的指针作为输入，则此函数返回错误。

D.5.11 cudaMemcpy2D()

```
cudaError_t cudaMemcpy2D(void* dst, size_t dpitch,
                         const void* src, size_t spitch,
                         size_t width, size_t height,
                         enum cudaMemcpyKind kind);
cudaError_t cudaMemcpy2DAsync(void* dst, size_t dpitch,
                              const void* src, size_t spitch,
                              size_t width, size_t height,
                              enum cudaMemcpyKind kind,
                              cudaStream_t stream);
```

将矩阵（height 行，每行 width 个字节）从由 src 指向的内存区域复制到由 dst 指向的内存区域，其中 kind 是 cudaMemcpyHostToHost、cudaMemcpyHostToDevice、cudaMemcpyDeviceToHost 或 cudaMemcpyDeviceToDevice 之一，指定复制的方向。dpitch 和 spitch 是由 dst 和 src 指向的 2D 数组在内存中的字节数宽度，其中包括添加到每行尾的任何填补（参见 D.5.2）。内存区域不可以重叠。使用与复制方向不匹配的 dst 和 src 指针调用 cudaMemcpy2D()将导致不确定的行为。如果 dpitch 和 spitch 大于允许的最大值（参见 D.1.4 中的 memPitch），则 cudaMemcpy2D()返回错误。

cudaMemcpy2DAsync()是异步的，可以通过传递非零 stream 参数与流相关联。它仅适用于页面锁定的宿主内存，如果传递指向可分页内存的指针作为输入，则此函数返回错误。

D.5.12 cudaMemcpyToArray()

```
cudaError_t cudaMemcpyToArray(struct cudaArray* dstArray,
                              size_t dstX, size_t dstY,
                              const void* src, size_t count,
                              enum cudaMemcpyKind kind);
cudaError_t cudaMemcpyToArrayAsync(struct cudaArray* dstArray,
                                   size_t dstX, size_t dstY,
                                   const void* src, size_t count,
                                   enum cudaMemcpyKind kind,
                                   cudaStream_t stream);
```

将 count 个字节从由 src 指向的内存区域复制到 CUDA 数组 dstArray 中从左上角（dstX, dstY）开始的位置，其中 kind 是 cudaMemcpyHostToHost、cudaMemcpyHostToDevice、cudaMemcpyDeviceToHost 或 cudaMemcpyDeviceToDevice 之一，指定复制的方向。

cudaMemcpyToArrayAsync()是异步的，可以通过传递非零 stream 参数与流相关联。它仅适用于页面锁定的宿主内存，如果传递指向可分页内存的指针作为输入，则此函数返回错误。

D.5.13 cudaMemcpy2DToArray()

```
cudaError_t cudaMemcpy2DToArray(struct cudaArray* dstArray,
                                size_t dstX, size_t dstY,
                                const void* src, size_t spitch,
                                size_t width, size_t height,
                                enum cudaMemcpyKind kind);
cudaError_t cudaMemcpy2DToArrayAsync(struct cudaArray* dstArray,
                                     size_t dstX, size_t dstY,
                                   const void* src, size_t spitch,
                                     size_t width, size_t height,
                                     enum cudaMemcpyKind kind,
                                     cudaStream_t stream);
```

将矩阵（height 行，每行 width 个字节）从由 src 指向的内存区域复制到 CUDA 数组 dstArray 中从左上角（dstX, dstY）开始的位置，其中 kind 是 cudaMemcpyHostToHost、cudaMemcpyHostToDevice、cudaMemcpyDeviceToHost 或 cudaMemcpyDeviceToDevice 之一，指定复制的方向。spitch 是由 src 指向的 2D 数组在内存中的字节数宽度，其中包括添加到每行尾的任何填补（参见 D.5.2）。如果 spitch 大于允许的最大值（参见 D.1.4 中的 memPitch），则 cudaMemcpy2D()返回错误。

cudaMemcpy2DToArrayAsync()是异步的，可以通过传递非零 stream 参数与流相关联。它仅适用于页面锁定的宿主内存，如果传递指向可分页内存的指针作为输入，则此函数返回错误。

D.5.14 cudaMemcpyFromArray()

```
cudaError_t cudaMemcpyFromArray(void* dst,
                                const struct cudaArray* srcArray,
                                size_t srcX, size_t srcY,
                                size_t count,
                                enum cudaMemcpyKind kind);
cudaError_t cudaMemcpyFromArrayAsync(void* dst,
                                 const struct cudaArray* srcArray,
                                     size_t srcX, size_t srcY,
                                     size_t count,
                                     enum cudaMemcpyKind kind,
                                     cudaStream_t stream);
```

将 count 个字节从 CUDA 数组 srcArray 的左上角(srcX, srcY)开始复制到 dst 指向的内存区域，其中 kind 是 cudaMemcpyHostToHost 、 cudaMemcpyHostToDevice 、cudaMemcpyDeviceToHost 或 cudaMemcpyDeviceToDevice 之一，指定复制的方向。

cudaMemcpyFromArrayAsync()是异步的，可以通过传递非零 stream 参数与流相关联。它仅适用于页面锁定的宿主内存，如果传递指向可分页内存的指针作为输入，则此函数返回错误。

D.5.15 cudaMemcpy2DFromArray()

```
cudaError_t cudaMemcpy2DFromArray(void* dst, size_t dpitch,
                                  const struct cudaArray* srcArray,
                                   size_t srcX, size_t srcY,
                                   size_t width, size_t height,
                                   enum cudaMemcpyKind kind);
cudaError_t cudaMemcpy2DFromArrayAsync(void* dst, size_t dpitch,
                                  const struct cudaArray* srcArray,
                                        size_t srcX, size_t srcY,
                                       size_t width, size_t height,
                                       enum cudaMemcpyKind kind,
                                        cudaStream_t stream);
```

将矩阵（height 行，每行 width 个字节）从 CUDA 数组 srcArray 的左上角（srcX, srcY）开始复制到由 dst 指向的内存区域，其中 kind 是 cudaMemcpyHostToHost、cudaMemcpyHostToDevice 、cudaMemcpyDeviceToHost 或 cudaMemcpyDeviceToDevice 之一，指定复制的方向。dpitch 是由 dst 指向的 2D 数组在内存中的字节数宽度，其中包括添加到每行尾的任何填补（参见 D.5.2）。如果 dpitch 大于允许的最大值（参见 D.1.4 中的 memPitch），则 cudaMemcpy2D()返回错误。

cudaMemcpy2DFromArrayAsync()是异步的，可以通过传递非零 stream 参数与流相关联。它仅适用于页面锁定的宿主内存，如果传递指向可分页内存的指针作为输入，则此函数返回错误。

D.5.16 cudaMemcpyArrayToArray()

```
cudaError_t cudaMemcpyArrayToArray(struct cudaArray* dstArray,
                                   size_t dstX, size_t dstY,
                                 const struct cudaArray* srcArray,
                                   size_t srcX, size_t srcY,
                                   size_t count,
                                   enum cudaMemcpyKind kind);
```

将 count 个字节从 CUDA 数组 srcArray 的左上角（srcX, srcY）开始复制到 CUDA 数组 dstArray 中的从左上角(dstX, dstY)开始的位置，其中 kind 是 cudaMemcpyHostToHost、cudaMemcpyHostToDevice、cudaMemcpyDeviceToHost 或 cudaMemcpyDeviceToDevice 之一，指定复制的方向。

D.5.17 cudaMemcpy2DArrayToArray()

```
cudaError_t cudaMemcpy2DArrayToArray(struct cudaArray* dstArray,
                                     size_t dstX, size_t dstY,
                                   const struct cudaArray* srcArray,
                                     size_t srcX, size_t srcY,
                                     size_t width, size_t height,
                                     enum cudaMemcpyKind kind);
```

将矩阵（height 行，每行 width 个字节）从 CUDA 数组 srcArray 的左上角（srcX, srcY）开始复制到 CUDA 数组 dstArray 中的从左上角（dstX, dstY）开始的位置，其中 kind 是 cudaMemcpyHostToHost、cudaMemcpyHostToDevice、cudaMemcpyDeviceToHost 或 cudaMemcpyDeviceToDevice 之一，指定复制的方向。

D.5.18 cudaMemcpyToSymbol()

```
template<class T>
cudaError_t cudaMemcpyToSymbol(const T& symbol, const void* src,
                               size_t count, size_t offset = 0,
                 enum cudaMemcpyKind kind = cudaMemcpyHostToDevice);
```

将 count 个字节从由 src 指向的内存区域复制到从符号 symbol 开始偏移 offset 字节指向的内存区域。内存区域不可以重叠。symbol 可以是一个驻留在全局或常量内存空间中的变量，也可以是一个字符串，用于命名驻留在全局或常量内存空间中的变量。kind 可以是 cudaMemcpyHostToDevice 或 cudaMemcpyDeviceToDevice。

D.5.19 cudaMemcpyFromSymbol()

```
template<class T>
cudaError_t cudaMemcpyFromSymbol(void *dst, const T& symbol,
                                 size_t count, size_t offset = 0,
                 enum cudaMemcpyKind kind = cudaMemcpyDeviceToHost);
```

将 count 个字节从以符号 symbol 开始偏移 offset 字节指向的内存区域复制到从由 dst 指

向的内存区域。内存区域不可以重叠。symbol 可以是一个驻留在全局或常量内存空间中的变量，也可以是一个字符串，用于命名驻留在全局或常量内存空间中的变量。kind 可以是 cudaMemcpyDeviceToHost 或 cudaMemcpyDeviceToDevice。

D.5.20 cudaGetSymbolAddress()

```
template<class T>
cudaError_t cudaGetSymbolAddress(void** devPtr, const T& symbol);
```

在*devPtr 中返回符号 symbol 在设备上的地址。symbol 可以是一个驻留在全局或常量内存空间中的变量，也可以是一个字符串，用于命名驻留在全局或常量内存空间中的变量。如果找不到 symbol，或者未在全局内存空间中声明 symbol，则*devPtr 不发生更改，并返回错误。如果失败，则 cudaGetSymbolAddress()返回 cudaErrorInvalidSymbol。

D.5.21 cudaGetSymbolSize()

```
template<class T>
cudaError_t cudaGetSymbolSize(size_t* size, const T& symbol);
```

在*size 中返回符号 symbol 的大小。symbol 可以是一个驻留在全局或常量内存空间中的变量，也可以是一个字符串，用于命名驻留在全局或常量内存空间中的变量。如果找不到 symbol，或者未在全局内存空间中声明 symbol，则*size 不发生更改，并返回错误。如果失败，则 cudaGetSymbolSize()返回 cudaErrorInvalidSymbol。

D.6 纹理参考管理

D.6.1 低层 API

D.6.1.1 cudaCreateChannelDesc()

```
struct cudaChannelFormatDesc
cudaCreateChannelDesc(int x, int y, int z, int w,
                      enum cudaChannelFormatKind f);
```

返回格式为 f 的通道描述符以及 x、y、z 和 w 分量的位数。cudaChannelFormatDesc 的介绍参见 4.3.4。

D.6.1.2 cudaGetChannelDesc()

```
cudaError_t cudaGetChannelDesc(struct cudaChannelFormatDesc* desc,
                               const struct cudaArray* array);
```

在*desc 中返回 CUDA 数组 array 的通道描述符。

D.6.1.3 cudaGetTextureReference()

```
cudaError_t cudaGetTextureReference(
```

```
                                    struct textureReference** texRef,
                                    const char* symbol);
```

在*texRef 中返回与由符号 symbol 定义的纹理参考相关联的结构。

D.6.1.4 cudaBindTexture()

```
cudaError_t cudaBindTexture(size_t* offset,
                            const struct textureReference* texRef,
                            const void* devPtr,
                        const struct cudaChannelFormatDesc* desc,
                            size_t size = UINT_MAX);
```

将由 devPtr 指向的内存区域的 size 个字节绑定到纹理参考 texRef。desc 描述在纹理拾取时如何解释内存。先前绑定到 texRef 的任何内存将解除绑定。

因为硬件强制在纹理基址上执行对齐，所以 cudaBindTexture()在*offset 中返回一个必须应用到纹理拾取的字节偏移，以便从所需内存中读取数据。此偏移必须除以纹理元素大小并传递给读取纹理的内核，以便这些内核可以应用于 tex1Dfetch()函数。如果设备内存指针从 cudaMalloc()中返回，则偏移一定为 0，NULL 也可以作为偏移参数传递。

D.6.1.5 cudaBindTextureToArray()

```
cudaError_t cudaBindTextureToArray(
                            const struct textureReference* texRef,
                            const struct cudaArray* array,
                            const struct cudaChannelFormatDesc* desc);
```

将 CUDA 数组 array 绑定到纹理参考 texRef。desc 描述在纹理拾取时如何解释内存。先前绑定到 texRef 的任何内存将解除绑定。

D.6.1.6 cudaUnbindTexture()

```
cudaError_t cudaUnbindTexture(
                            const struct textureReference* texRef);
```

对绑定到纹理参考 texRef 的纹理解除绑定。

D.6.1.7 cudaGetTextureAlignmentOffset()

```
cudaError_t cudaGetTextureAlignmentOffset(size_t* offset,
                            const struct textureReference* texRef);
```

在*offset 中返回偏移，该偏移是绑定纹理参考 texRef 时返回的。

D.6.2 高层 API

D.6.2.1 cudaCreateChannelDesc()

```
template<class T>
struct cudaChannelFormatDesc cudaCreateChannelDesc<T>();
```

返回格式与类型 T 匹配的通道描述符。类型 T 可以是 4.3.1.1 中的任意类型。3-分量类型被默认为 4-分量格式。

D.6.2.2 cudaBindTexture()

```
template<class T, int dim, enum cudaTextureReadMode readMode>
static __inline__ __host__ cudaError_t
cudaBindTexture(size_t* offset,
                const struct texture<T, dim, readMode>& texRef,
                const void* devPtr,
                const struct cudaChannelFormatDesc& desc,
                size_t size = UINT_MAX);
```

将由 devPtr 指向的内存区域的 size 个字节绑定到纹理参考 texRef。desc 描述在纹理拾取时如何解释内存。偏移参数是可选的，这和 D.6.1.4 介绍的低级 cudaBindTexture() 函数一样。先前绑定到 texRef 的任何内存将解除绑定。

```
template<class T, int dim, enum cudaTextureReadMode readMode>
static __inline__ __host__ cudaError_t
cudaBindTexture(size_t* offset,
                const struct texture<T, dim, readMode>& texRef,
                const void* devPtr,
                size_t size = UINT_MAX);
```

将由 devPtr 指向的内存区域的 size 个字节绑定到纹理参考 texRef。通道描述符从纹理参考类型中继承。偏移参数是可选的，这和 D.6.1.4 介绍的低级 cudaBindTexture() 函数一样。先前绑定到 texRef 的任何内存将解除绑定。

D.6.2.3 cudaBindTextureToArray()

```
template<class T, int dim, enum cudaTextureReadMode readMode>
static __inline__ __host__ cudaError_t
cudaBindTextureToArray(
                  const struct texture<T, dim, readMode>& texRef,
                  const struct cudaArray* cuArray,
                  const struct cudaChannelFormatDesc& desc);
```

将 CUDA 数组 array 绑定到纹理参考 texRef。desc 描述在纹理拾取时如何解释内存。先前绑定到 texRef 的任何 CUDA 数组将解除绑定。

```
template<class T, int dim, enum cudaTextureReadMode readMode>
static __inline__ __host__ cudaError_t
cudaBindTextureToArray(
                   const struct texture<T, dim, readMode>& texRef,
                   const struct cudaArray* cuArray);
```

将 CUDA 数组 array 绑定到纹理参考 texRef。通道描述符从 CUDA 数组中继承。先前绑定到 texRef 的任何 CUDA 数组将解除绑定。

D.6.2.4 cudaUnbindTexture()

```
template<class T, int dim, enum cudaTextureReadMode readMode>
static __inline__ __host__ cudaError_t
cudaUnbindTexture(const struct texture<T, dim, readMode>& texRef);
```

解除对绑定到纹理参考的 texRef 的绑定。

D.7 执行控制

D.7.1 cudaConfigureCall()

```
cudaError_t cudaConfigureCall(dim3 gridDim, dim3 blockDim,
                              size_t sharedMem = 0,
                              int tokens = 0);
```

指定要执行的设备调用的网格和块维度，这类似于 4.2.3 中介绍的执行配置语法。cudaConfigureCall()是基于堆栈的。每个调用在执行堆栈顶部压入数据。此数据包含网格和线程块的维度，以及调用的任何参数。

D.7.2 cudaLaunch()

```
template<class T> cudaError_t cudaLaunch(T entry);
```

在设备上启动函数 entry。entry 可以是一个在设备上执行的函数，也可以是一个字符串，用于命名在设备上执行的函数。entry 必须声明为__global__函数。在 cudaLaunch()之前必须调用 cudaConfigureCall()，因为此函数从执行栈中弹出由 cudaConfigureCall()压入的数据。

D.7.3 cudaSetupArgument()

```
cudaError_t cudaSetupArgument(void* arg,
                              size_t count, size_t offset);
template<class T> cudaError_t cudaSetupArgument(T arg,
                                                size_t offset);
```

将由 arg 指向的参数的 count 个字节存入从参数传递区域的开始位置偏移 offset 个字节(从 0 开始）处。参数存储在执行栈的顶部。在 cudaSetupArgument()之前必须调用 cudaConfigureCall()。

D.8 OpenGL 互操作性

D.8.1 cudaGLRegisterBufferObject()

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

注册 ID 为 bufferObj 的缓冲对象供 CUDA 访问。在 CUDA 可以映射缓冲对象之前，必须调用此函数。注册之后，除作为 OpenGL 的绘图命令之外，缓冲对象不能由任何 OpenGL 命令使用。

D.8.2 cudaGLMapBufferObject()

```
cudaError_t cudaGLMapBufferObject(void** devPtr,
                                  GLuint bufferObj);
```

将 ID 为 bufferObj 的缓冲对象映射到 CUDA 的地址空间中，并在*devPtr 中返回结果映射的基指针。

D.8.3 cudaGLUnmapBufferObject()

```
cudaError_t cudaGLUnmapBufferObject(GLuint bufferObj);
```

取消供 CUDA 访问的 ID 为 bufferObj 的缓冲对象的映射。

D.8.4 cudaGLUnregisterBufferObject()

```
cudaError_t cudaGLUnregisterBufferObject(GLuint bufferObj);
```

注销供 CUDA 访问的 ID 为 bufferObj 的缓冲对象。

D.9 Direct3D 互操作性

D.9.1 cudaD3D9Begin()

```
cudaError_t cudaD3D9Begin(IDirect3DDevice9* device);
```

初始化与 Direct3D 设备 device 的互操作。在 CUDA 可以映射 device 中的任何对象之前，必须调用此函数。然后，应用程序可以映射 Direct3D 设备的顶点缓冲，直到调用 cuD3D9End() 为止。

D.9.2 cudaD3D9End()

```
cudaError_t cudaD3D9End(void);
```

结束先前由 cuD3D9Begin()开始的 Direct3D 设备的互操作。

D.9.3 cudaD3D9RegisterVertexBuffer()

```
cudaError_t
cudaD3D9RegisterVertexBuffer(IDirect3DVertexBuffer9* VB);
```

注册供 CUDA 访问的顶点缓冲 VB。

D.9.4 cudaD3D9MapVertexBuffer()

```
cudaError_t cudaD3D9MapVertexBuffer(void** devPtr,
                                    IDirect3DVertexBuffer9* VB);
```

将顶点缓冲 VB 映射到当前 CUDA 上下文的地址空间中，并在*devPtr 中返回结果映射的基指针。

D.9.5 cudaD3D9UnmapVertexBuffer()

```
cudaError_t cudaD3D9UnmapVertexBuffer(IDirect3DVertexBuffer9* VB);
```

取消供 CUDA 访问的顶点缓冲 VB 的映射。

D.9.6 cudaD3D9UnregisterVertexBuffer()

```
cudaError_t
cudaD3D9UnregisterVertexBuffer(IDirect3DVertexBuffer9* VB);
```

注销供 CUDA 访问的顶点缓冲 VB。

D.9.7 cudaD3D9GetDevice()

```
cudaError_t
cudaD3D9GetDevice(int* dev, const char* adapterName);
```

在*dev 中返回与从 EnumDisplayDevices 或 IDirect3D9::GetAdapterIdentifier() 中获得的适配器名称 adapterName 相应的设备。

D.10 错误处理

D.10.1 cudaGetLastError()

```
cudaError_t cudaGetLastError(void);
```

返回从同一宿主线程中任何运行时调用返回的最后一个错误，并将其重置为 cudaSuccess。

D.10.2 cudaGetErrorString()

```
const char* cudaGetErrorString(cudaError_t error);
```

返回错误代码中的消息字符串。

附录 E　驱动程序 API 参考

E.1　初始化

E.1.1　cuInit()

```
CUresult cuInit(unsigned int Flags);
```

在调用驱动程序 API 中的其它任何函数之前，必须初始化驱动程序 API。当前，Flags 参数必须为 0。如果未调用 cuInit()，则驱动程序 API 中的任何函数将返回 CUDA_ERROR_NOT_INITIALIZED。

E.2　设备管理

E.2.1　cuDeviceGetCount()

```
CUresult cuDeviceGetCount(int* count);
```

在*count 中返回可供执行的计算能力大于或等于 1.0 的设备数。如果没有这样一个设备，则 cuDeviceGetCount()返回 1，设备 0 仅支持设备仿真模式，且其计算能力小于 1.0。

E.2.2　cuDeviceGet()

```
CUresult cuDeviceGet(CUdevice* dev, int ordinal);
```

在*dev 中返回给定序号在区间[0, cuDeviceGetCount()-1]中的设备句柄。

E.2.3　cuDeviceGetName()

```
CUresult cuDeviceGetName(char* name, int len, CUdevice dev);
```

在由 name 指向的 NULL 结尾的字符串中返回标识设备 dev 的 ASCII 字符串。len 指定返回的字符串的最大长度。

E.2.4 cuDeviceTotalMem()

```
CUresult cuDeviceTotalMem(unsigned int* bytes, CUdevice dev);
```

在*bytes 中返回设备 dev 上可用的内存总量，单位为字节。

E.2.5 cuDeviceComputeCapability()

```
CUresult cuDeviceComputeCapability(int* major, int* minor,
                                   CUdevice dev);
```

在*major 和*minor 中返回定义设备 dev 的计算能力的主要和次要修订号。

E.2.6 cuDeviceGetAttribute()

```
CUresult cuDeviceGetAttribute(int* value,
                              CUdevice_attribute attrib,
                              CUdevice dev);
```

在*value 中返回设备 dev 的属性 attrib，这些属性为整数值。支持的属性包括：

- CU_DEVICE_ATTRIBUTE_MAX_THREADS_PER_BLOCK：每块的最大线程数；
- CU_DEVICE_ ATTRIBUTE_MAX_BLOCK_DIM_X：块的最大 x 维度；
- CU_DEVICE_ATTRIBUTE_MAX_BLOCK_DIM_Y：块的最大 y 维度；
- CU_DEVICE_ATTRIBUTE_MAX_BLOCK_DIM_Z：块的最大 z 维度；
- CU_DEVICE_ATTRIBUTE_MAX_GRID_DIM_X：网格的最大 x 维度；
- CU_DEVICE_ATTRIBUTE_MAX_GRID_DIM_Y：网格的最大 y 维度；
- CU_DEVICE_ATTRIBUTE_MAX_GRID_DIM_Z：网格的最大 z 维度；
- CU_DEVICE_ATTRIBUTE_SHARED_MEMORY_PER_BLOCK：每块可用的共享内存总量，单位为字节；
- CU_DEVICE_ATTRIBUTE_TOTAL_CONSTANT_MEMORY：设备上可用的常量内存总量，单位为字节；
- CU_DEVICE_ATTRIBUTE_WARP_SIZE：warp 大小；
- CU_DEVICE_ATTRIBUTE_MAX_PITCH：E.8 中的内存复制函数允许的最大节距，包括通过 cuMemAllocPitch()（参见 E8.3）分配的内存区域；
- CU_DEVICE_ATTRIBUTE_REGISTERS_PER_BLOCK：每块可用的寄存器总数；
- CU_DEVICE_ATTRIBUTE_CLOCK_RATE：时钟频率，单位为千赫；
- CU_DEVICE_ATTRIBUTE_TEXTURE_ALIGNMENT：4.3.4.3 中提及的对齐要求；已经对齐为 textureAlign 个字节的纹理基址不再需要那种应用于纹理拾取的偏移；

- CU_DEVICE_ATTRIBUTE_GPU_OVERLAP：如果设备可以实现内核执行和复制内存（宿主和设备之间）的并发，则为 1，否则为 0。

E.2.7 cuDeviceGetProperties()

```
cudaError_t cuGetDeviceProperties(CUdevprop* prop,
                                  CUdevice dev);
```

在*prop 中返回设备 dev 的属性。CUdevprop 结构定义为：

```
typedef struct CUdevprop_st {
    int maxThreadsPerBlock;
    int maxThreadsDim[3];
    int maxGridSize[3];
    int sharedMemPerBlock;
    int totalConstantMemory;
    int SIMDWidth;
    int memPitch;
    int regsPerBlock;
    int clockRate;
    int textureAlign;
} CUdevprop;
```

其中：

- maxThreadsPerBlock 是每块的最大线程数；
- maxThreadsDim[3]是块的每个维度的最大大小；
- maxGridSize[3]是网格的每个维度的最大大小；
- sharedMemPerBlock 是每块可用的共享内存总量，单位为字节；
- totalConstantMemory 是设备上可用的常量内存总量，单位为字节；
- SIMDWidth 是 warp 大小；
- memPitch 是 E.8 中的内存复制函数允许的最大节距，包括 cuMemAllocPitch()（参见 E.8.3）分配的内存区域；
- regsPerBlock 是每块可用的寄存器总数；
- clockRate 是时钟频率，单位为千赫；
- textureAlign 是 4.3.4.3 中提及的对齐要求；已经对齐为 textureAlign 个字节的纹理基址不再需要那种应用于纹理拾取的偏移。

E.3 上下文管理

E.3.1 cuCtxCreate()

```
CUresult cuCtxCreate(CUcontext* pCtx, unsigned int Flags, CUdevice
dev);
```

为设备创建新上下文，并将其与调用线程相关联。当前，Flags 参数必须为 0。调用了 cuCtxCreate()之后，上下文使用计数（usage count）变为 1，该上下文完成使用之后，还必须调用 cuCtxDetach()递减上下文的使用计数。如果某个上下文已经是此线程的当前上下文，则此函数调用失败。

E.3.2 cuCtxAttach()

```
CUresult cuCtxAttach(CUcontext* pCtx, unsigned int Flags);
```

递增上下文的使用计数，并在*pCtx 中传回上下文句柄，当应用程序完成使用上下文之后，此句柄必须传递给 cuCtxDetach()。如果线程没有当前上下文，则 cuCtxAttach()调用失败。当前，Flags 参数必须为 0。

E.3.3 cuCtxDetach()

```
CUresult cuCtxDetach(CUcontext ctx);
```

递减上下文的使用计数，如果使用计数达到 0，则销毁上下文。上下文句柄必须是通过 cuCtxCreate()或 cuCtxAttach()传回的，并且必须是调用线程的当前上下文。

E.3.4 cuCtxGetDevice()

```
CUresult cuCtxGetDevice(CUdevice* device);
```

在*device 中返回当前上下文的设备的序号。

E.3.5 cuCtxSynchronize()

```
CUresult cuCtxSynchronize(void);
```

阻塞，直到设备已经完成所有先前请求的任务为止。如果先前任务之一失败，则 cuCtxSynchronize()返回错误。

E.4 模块管理

E.4.1 cuModuleLoad()

```
CUresult cuModuleLoad(CUmodule* mod, const char* filename);
```

从文件名为 filename 的文件加载相应模块 mod 到当前上下文中。CUDA 驱动程序 API 不会延迟分配模块所需的资源；如果无法分配模块所需的函数和数据的内存（常量和全局），则 cuModuleLoad()调用失败。文件应该是由 nvcc 输出的 *cubin* 文件（参见 4.2.5）。

E.4.2 cuModuleLoadData()

```
CUresult cuModuleLoadData(CUmodule* mod, const void* image);
```

从指针 image 中加载相应模块 mod 到当前上下文中。可以通过映射 *cubin* 文件获得该指针，将 *cubin* 文件作为文本字符串传递，或将 *cubin* 对象合并到可执行资源并使用操作系统调用（比如 Windows 的 FindResource()）来获得指针。

E.4.3 cuModuleLoadFatBinary()

```
CUresult cuModuleLoadFatBinary(CUmodule* mod, const void* fatBin);
```

从指针 fatBin 加载相应模块 mod 到当前上下文中。指针表示*多体系结构二进制（fat binary）*对象，此对象是不同 *cubin* 文件的集合，这些文件表示相同的设备代码，但针对不同的体系结构进行了编译和优化。编程人员还不能够在当前正式发布的 API 上构造和使用多体系结构二进制对象，因此此函数是此版本 CUDA 的一个内部函数。更多信息参见 nvcc 文档。

E.4.4 cuModuleUnload()

```
CUresult cuModuleUnload(CUmodule mod);
```

从当前上下文中卸载模块 mod。

E.4.5 cuModuleGetFunction()

```
CUresult cuModuleGetFunction(CUfunction* func,
                             CUmodule mod, const char* funcname);
```

在*func 中返回位于模块 mod 中的名称为 funcname 的函数的句柄。如果不存在具有此名称的函数，则 cuModuleGetFunction()返回 CUDA_ERROR_NOT_FOUND。

E.4.6 cuModuleGetGlobal()

```
CUresult cuModuleGetGlobal(CUdeviceptr* devPtr,
                           unsigned int* bytes,
                           CUmodule mod, const char* globalname);
```

在*devPtr 和*bytes 中返回位于模块 mod 中的名称为 globalname 的全局变量的基指针和大小。如果不存在具有此名称的变量，则 cuModuleGetGlobal() 返回 CUDA_ERROR_NOT_FOUND。两个参数 devPtr 和 bytes 是可选的。忽略其中的空值参数。

E.4.7 cuModuleGetTexRef()

```
CUresult cuModuleGetTexRef(CUtexref* texRef,
                           CUmodule hmod, const char* texrefname);
```

在*texref 中返回模块 mod 中名称为 texrefname 的纹理参考的句柄。如果不存在具有此名称的纹理参考，则 cuModuleGetTexRef()返回 CUDA_ERROR_NOT_FOUND。不要销毁此纹理参考句柄，因为它将在模块卸载时自动销毁。

E.5 流管理

E.5.1 cuStreamCreate()

```
CUresult cuStreamCreate(CUstream* stream, unsigned int flags);
```

创建流。当前，flags 必须为 0。

E.5.2 cuStreamQuery()

```
CUresult cuStreamQuery(CUstream stream);
```

如果流中的所有操作都已完成，则返回 CUDA_SUCCESS，否则，返回 CUDA_ERROR_NOT_READY。

E.5.3 cuStreamSynchronize()

```
CUresult cuStreamSynchronize(CUstream stream);
```

阻塞，直到设备完成流中的所有操作为止。

E.5.4 cuStreamDestroy()

```
CUresult cuStreamDestroy(CUstream stream);
```

销毁流。

E.6 事件管理

E.6.1 cuEventCreate()

```
CUresult cuEventCreate(CUevent* event, unsigned int flags);
```

创建事件。当前，flags 必须为 0。

E.6.2 cuEventRecord()

```
CUresult cuEventRecord(CUevent event, CUstream stream);
```

记录事件。如果 stream 非零，则在流中的所有先前操作都已完成之后记录事件；否则，在 CUDA 上下文中的所有先前操作都已完成之后记录事件。因为此操作是异步的，所以必须使用 cuEventQuery()和/或 cuEventSynchronize()来确定事件实际被记录的时间。

如果已经调用 cuEventRecord()，但事件尚未记录，则此函数返回 CUDA_ERROR_INVALID_VALUE。

E.6.3 cuEventQuery()

```
CUresult cuEventQuery(CUevent event);
```

如果事件实际已经被记录，则返回 CUDA_SUCCESS，否则，返回 CUDA_ERROR_NOT_READY。如果尚未在此事件上调用 cuEventRecord()，则此函数返回 CUDA_ERROR_INVALID_VALUE。

E.6.4 cuEventSynchronize()

```
CUresult cuEventSynchronize(CUevent event);
```

阻塞，直到事件实际已经被记录为止。如果尚未在此事件上调用 cuEventRecord()，则此函数返回 CUDA_ERROR_INVALID_VALUE。

E.6.5 cuEventDestroy()

```
CUresult cuEventDestroy(CUevent event);
```

销毁事件。

E.6.6 cuEventElapsedTime()

```
CUresult cuEventDestroy(float* time,
                        CUevent start, CUevent end);
```

计算两个事件之间用去的时间（单位为毫秒，分辨率约为 0.5 微秒）。如果其中一个事件尚未被记录，则此函数返回 CUDA_ERROR_INVALID_VALUE。如果使用非零流记录了任一事件，则结果是不确定的。

E.7 执行控制

E.7.1 cuFuncSetBlockShape()

```
CUresult cuFuncSetBlockShape(CUfunction func,
                             int x, int y, int z);
```

指定在启动由 func 给定的内核时创建的线程块的 X、Y 和 Z 维度。

E.7.2 cuFuncSetSharedSize()

```
CUresult cuFuncSetSharedSize(CUfunction func, unsigned int bytes);
```

通过 bytes 设置在启动由 func 给定的内核时将可用于每个线程块的共享内存量。

E.7.3 cuParamSetSize()

```
CUresult cuParamSetSize(CUfunction func, unsigned int numbytes);
```

通过 numbytes 设置函数 func 的函数参数所需的总大小（单位为字节）。

E.7.4 cuParamSeti()

```
CUresult cuParamSeti(CUfunction func,
                     int offset, unsigned int value);
```

设置下次调用与 func 对应的内核时将指定的整数参数。offset 是字节偏移。

E.7.5 cuParamSetf()

```
CUresult cuParamSetf(CUfunction func,
                     int offset, float value);
```

设置下次调用与 func 对应的内核时将指定的浮点参数。offset 是字节偏移。

E.7.6 cuParamSetv()

```
CUresult cuParamSetv(CUfunction func,
                     int offset, void* ptr,
                     unsigned int numbytes);
```

将任何数量的数据复制到与 func 对应的内核的参数空间中。offset 是字节偏移。

E.7.7 cuParamSetTexRef()

```
CUresult cuParamSetTexRef(CUfunction func,
                          int texunit, CUtexref texRef);
```

使得绑定到纹理参考 texRef 的 CUDA 数组或线性内存可供设备程序用作纹理。在此版本的 CUDA 中，纹理参考必须通过 cuModuleGetTexRef() 获得，texunit 参数必须设置为 CU_PARAM_TR_DEFAULT。

E.7.8 cuLaunch()

```
CUresult cuLaunch(CUfunction func);
```

在块维度是 1×1 的网格上调用内核 func。块包含上一次调用 cuFuncSetBlockShape() 的指

定的线程数。

E.7.9 cuLaunchGrid()

```
CUresult cuLaunchGrid(CUfunction func,
                      int grid_width, int grid_height);
CUresult cuLaunchGridAsync(CUfunction func,
                           int grid_width, int grid_height,
                           CUstream stream);
```

在块维度为 grid_width × grid_height 的网格上调用内核。每块包含上一次调用 cuFuncSetBlockShape()的指定的线程数。

可以通过传递非零 stream 参数将 cuLaunchGridAsync()关联到流。此函数仅适用于页面锁定的宿主内存，如果将可分页内存的指针作为输入，则此函数将返回错误。

E.8 内存管理

E.8.1 cuMemGetInfo()

```
CUresult cuMemGetInfo(unsigned int* free, unsigned int* total);
```

分别在*free 和*total 中返回可供 CUDA 上下文分配的空闲内存量和总内存量，单位为字节。

E.8.2 cuMemAlloc()

```
CUresult cuMemAlloc(CUdeviceptr* devPtr, unsigned int count);
```

在设备上分配 count 个字节的线性内存，并在*devPtr 中返回指向已分配内存的指针。已分配内存已经针对任何种类的变量进行了适当的对齐。内存是未清空的。如果 count 为 0，则 cuMemAlloc()返回 CUDA_ERROR_INVALID_VALUE。

E.8.3 cuMemAllocPitch()

```
CUresult cuMemAllocPitch(CUdeviceptr* devPtr,
                         unsigned int* pitch,
                         unsigned int widthInBytes,
                         unsigned int height,
                         unsigned int elementSizeBytes);
```

在设备上分配至少 widthInBytes*height 个字节的线性内存，并在*devPtr 中返回指向已分配内存的指针。此函数必须填补内存以确保任何给定行中的相应指针都将继续满足对齐要求，以便当地址在行之间更新时进行内存合并（参见 5.1.2.1）。elementSizeBytes 指定将在内存范围上执行的最大的读和写的大小。

elementSizeBytes 可以是 4、8 或 16（因为已合并的内存事务处理不能用于其它数据大小）。如果 elementSizeBytes 小于内核的实际读/写大小，则内核将正确运行，但速度可能会下降。由 cuMemAllocPitch()在*pitch 中返回的节距是分配的宽度（单位为字节）。作为与分配相独立的参数，节距用于计算 2D 数组中的地址。给定类型为 T 的数组元素的行和列，则地址计算为：

```
T* pElement = (T*)((char*)BaseAddress + Row * Pitch) + Column;
```

由 cuMemAllocPitch()返回的节距保证可以在所有情况下处理 cuMemcpy2D()。对于 2D 数组的分配，建议编程人员考虑使用 cuMemAllocPitch()执行节距分配。由于硬件中的对齐限制，如果应用程序将在设备内存的不同区域（不管是线性内存还是 CUDA 数组）之间执行 2D 内存复制，则更应该使用此函数。

E.8.4 cuMemFree()

```
CUresult cuMemFree(CUdeviceptr devPtr);
```

对于调用 cuMemMalloc()或 cuMemMallocPitch()返回的指针 devPtr，释放其指向的内存空间。

E.8.5 cuMemAllocHost()

```
CUresult cuMemAllocHost(void** hostPtr, unsigned int count);
```

分配 count 个字节的页面锁定的可由设备访问的宿主内存。驱动程序跟踪使用此函数分配的虚拟内存范围，并自动加速对 cuMemcpy()等函数的调用。因为此内存可由设备直接访问，所以与使用 malloc()等函数获得的可分页内存相比，它在进行读取或写入时具有高得多的带宽。使用 cuMemAllocHost()分配过量的内存可能降低系统性能，因为它降低了系统可用于分页的内存量。因此，最好节约使用此函数分配中转区进行宿主和设备之间的数据交换。

E.8.6 cuMemFreeHost()

```
CUresult cuMemFreeHost(void* hostPtr);
```

对于调用 cuMemAllocHost()返回的指针 hostPtr，释放其指向的内存空间。

E.8.7 cuMemGetAddressRange()

```
CUresult cuMemGetAddressRange(CUdeviceptr* basePtr,
                              unsigned int* size,
                              CUdeviceptr devPtr);
```

在*basePtr 和*size 中返回输入指针 devPtr（由 cuMemAlloc()或 cuMemAllocPitch()分配）的基址和大小。两个参数 basePtr 和 size 是可选的。忽略其中的空值参数。

E.8.8 cuArrayCreate()

```
CUresult cuArrayCreate(CUarray* array,
                       const CUDA_ARRAY_DESCRIPTOR* desc);
```

按照 CUDA_ARRAY_DESCRIPTOR 结构 desc 创建 CUDA 数组，并在*array 中返回新 CUDA 数组的句柄。CUDA_ARRAY_DESCRIPTOR 结构定义如下：

```
typedef struct {
    unsigned int Width;
    unsigned int Height;
    CUarray_format Format;
    unsigned int NumChannels;
} CUDA_ARRAY_DESCRIPTOR;
```

其中：

- Width 和 Height 是 CUDA 数组的宽度和高度（单位为元素数）；如果 Height 为 0，则 CUDA 数组为一维，否则为二维；
- NumChannels 指定每个 CUDA 数组元素的分量个数；可以是 1、2 或 4；
- Format 指定元素的格式；CUarray_format 定义如下：

```
typedef enum CUarray_format_enum {
    CU_AD_FORMAT_UNSIGNED_INT8  = 0x01,
    CU_AD_FORMAT_UNSIGNED_INT16 = 0x02,
    CU_AD_FORMAT_UNSIGNED_INT32 = 0x03,
    CU_AD_FORMAT_SIGNED_INT8    = 0x08,
    CU_AD_FORMAT_SIGNED_INT16   = 0x09,
    CU_AD_FORMAT_SIGNED_INT32   = 0x0a,
    CU_AD_FORMAT_HALF           = 0x10,
    CU_AD_FORMAT_FLOAT          = 0x20
} CUarray_format;
```

CUDA 数组描述的示例如下：

- 具有 2048 个浮点数的 CUDA 数组的描述；

```
CUDA_ARRAY_DESCRIPTOR desc;
desc.Format = CU_AD_FORMAT_FLOAT;
desc.NumChannels = 1;
desc.Width = 2048;
desc.Height = 1;
```

- 浮点数类型的 64×64 CUDA 数组的描述：

```
CUDA_ARRAY_DESCRIPTOR desc;
desc.Format = CU_AD_FORMAT_FLOAT;
desc.NumChannels = 1;
desc.Width = 64;
desc.Height = 64;
```

- 64-位元素（4x16-位半精度浮点数类型）的 width×height CUDA 数组的描述：

```
CUDA_ARRAY_DESCRIPTOR desc;
desc.FormatFlags = CU_AD_FORMAT_HALF;
desc.NumChannels = 4;
```

```
desc.Width = width;
desc.Height = height;
```

- 16-位元素（两个 8-位无符号字符）的 `width×height` CUDA 数组的描述：

```
CUDA_ARRAY_DESCRIPTOR arrayDesc;
desc.FormatFlags = CU_AD_FORMAT_UNSIGNED_INT8;
desc.NumChannels = 2;
desc.Width = width;
desc.Height = height;
```

E.8.9 cuArrayGetDescriptor()

```
CUresult cuArrayGetDescriptor(CUDA_ARRAY_DESCRIPTOR* arrayDesc,
                              CUarray array);
```

在`*arrayDesc` 中返回用于创建 CUDA 数组 `array` 的描述符。这对已经传递了 CUDA 数组的子例程很有用（但需要确定 CUDA 数组参数以用于验证或其它目的）。

E.8.10 cuArrayDestroy()

```
CUresult cuArrayDestroy(CUarray array);
```

销毁 CUDA 数组 `array`。

E.8.11 cuMemset()

```
CUresult cuMemsetD8(CUdeviceptr dstDevPtr,
                    unsigned char value, unsigned int count);
CUresult cuMemsetD16(CUdeviceptr dstDevPtr,
                     unsigned short value, unsigned int count);
CUresult cuMemsetD32(CUdeviceptr dstDevPtr,
                     unsigned int value, unsigned int count);
```

将 8-位、16-位或 32-位的内存范围 `count` 设置为指定的值 `value`。

E.8.12 cuMemset2D()

```
CUresult cuMemsetD2D8(CUdeviceptr dstDevPtr,
                      unsigned int dstPitch,
                      unsigned char value,
                      unsigned int width, unsigned int height);
CUresult cuMemsetD2D16(CUdeviceptr dstDevPtr,
                       unsigned int dstPitch,
                       unsigned short value,
                       unsigned int width, unsigned int height);
CUresult cuMemsetD2D32(CUdeviceptr dstDevPtr,
                       unsigned int dstPitch,
                       unsigned int value,
                       unsigned int width, unsigned int height);
```

将 8-位、16-位或 32-位的 2D 内存范围 `width` 设置为指定的值 `value`。`height` 指定要设置的行数，`dstPitch` 指定每行之间的字节数（参见 E.8.3）。当该节距已经由 `cuMemAllocPitch()`

传回时，这些函数的执行效率很高。

E.8.13 cuMemcpyHtoD()

```
CUresult cuMemcpyHtoD(CUdeviceptr dstDevPtr,
                      const void *srcHostPtr,
                      unsigned int count);
CUresult cuMemcpyHtoDAsync(CUdeviceptr dstDevPtr,
                           const void *srcHostPtr,
                           unsigned int count,
                           CUstream stream);
```

从宿主内存复制到设备内存。dstDevPtr 和 srcHostPtr 分别指定目标和来源的基址。count 指定要复制的字节数。

cuMemcpyHtoDAsync()是异步的，可以通过传递非零 stream 流参数将其关联到流。它仅适用于页面锁定的宿主内存，如果传递指向可分页内存的指针作为输入，则此函数返回错误。

E.8.14 cuMemcpyDtoH()

```
CUresult cuMemcpyDtoH(void* dstHostPtr,
                      CUdeviceptr srcDevPtr,
                      unsigned int count);
CUresult cuMemcpyDtoHAsync(void* dstHostPtr,
                           CUdeviceptr srcDevPtr,
                           unsigned int count,
                           CUstream stream);
```

从设备内存复制到宿主内存。dstHostPtr 和 srcDevPtr 分别指定来源和目标的基址。count 指定要复制的字节数。

cuMemcpyDtoHAsync()是异步的，可以通过传递非零 stream 流参数将其关联到流。它仅适用于页面锁定的宿主内存，如果传递指向可分页内存的指针作为输入，则此函数返回错误。

E.8.15 cuMemcpyDtoD()

```
CUresult cuMemcpyDtoD(CUdeviceptr dstDevPtr,
                      CUdeviceptr srcDevPtr,
                      unsigned int count);
```

从设备内存复制到设备内存。dstDevPtr 和 srcDevPtr 分别指定目标和来源的基址。count 指定要复制的字节数。

E.8.16 cuMemcpyDtoA()

```
CUresult cuMemcpyDtoA(CUarray dstArray,
                      unsigned int dstIndex,
                      CUdeviceptr srcDevPtr,
                      unsigned int count);
```

从设备内存复制到一维 CUDA 数组。dstArray 和 dstIndex 指定目标数据的 CUDA 数组句柄和开始索引。srcDevPtr 指定来源的基指针。count 指定要复制的字节数。

E.8.17 cuMemcpyAtoD()

```
CUresult cuMemcpyAtoD(CUdeviceptr dstDevPtr,
                      CUarray srcArray,
                      unsigned int srcIndex,
                      unsigned int count);
```

从一维 CUDA 数组复制到设备内存。dstDevPtr 指定目标的基指针，并且必须与 CUDA 数组元素自然对齐。srcArray 和 srcIndex 指定 CUDA 数组句柄和复制开始时数组元素的索引。count 指定要复制的字节数，并且必须可由数组元素大小整除。

E.8.18 cuMemcpyAtoH()

```
CUresult cuMemcpyAtoH(void* dstHostPtr,
                      CUarray srcArray,
                      unsigned int srcIndex,
                      unsigned int count);
CUresult cuMemcpyAtoHAsync(void* dstHostPtr,
                           CUarray srcArray,
                           unsigned int srcIndex,
                           unsigned int count,
                           CUstream stream);
```

从一维 CUDA 数组复制到宿主内存。dstHostPtr 指定目标的基址。srcArray 和 srcIndex 指定来源数据的 CUDA 数组句柄和开始索引。count 指定要复制的字节数。

cuMemcpyAtoHAsync()是异步的，可以通过传递非零 stream 流参数将其关联到流。它仅适用于页面锁定的宿主内存，如果传递指向可分页内存的指针作为输入，则此函数返回错误。

E.8.19 cuMemcpyHtoA()

```
CUresult cuMemcpyHtoA(CUarray dstArray,
                      unsigned int dstIndex,
                      const void *srcHostPtr,
                      unsigned int count);
CUresult cuMemcpyHtoAAsync(CUarray dstArray,
```

```
                                unsigned int dstIndex,
                                const void *srcHostPtr,
                                unsigned int count,
                                CUstream stream);
```

从宿主内存复制到一维 CUDA 数组。dstArray 和 dstIndex 指定目标数据的 CUDA 数组句柄和开始索引。srcHostPtr 指定来源的基址。count 指定要复制的字节数。

cuMemcpyHtoAAsync()是异步的，可以通过传递非零 stream 流参数将其关联到流。它仅适用于页面锁定的宿主内存，如果传递指向可分页内存的指针作为输入，则此函数返回错误。

E.8.20 cuMemcpyAtoA()

```
CUresult cuMemcpyAtoA(CUarray dstArray,
                      unsigned int dstIndex,
                      CUarray srcArray,
                      unsigned int srcIndex,
                      unsigned int count);
```

从一个一维 CUDA 数组复制到另一个一维 CUDA 数组。dstArray 和 srcArray 分别指定要复制的目标和来源 CUDA 数组的句柄。dstIndex 和 srcIndex 指定 CUDA 数组的目标和来源索引而非字节偏移，这些索引值位于 CUDA 数组的[0, Width-1]区间内。count 是要复制的字节数。CUDA 数组中元素的大小不需要具有相同的格式，但必须具有相同的大小；count 必须可由此大小整除。

E.8.21 cuMemcpy2D()

```
CUresult cuMemcpy2D(const CUDA_MEMCPY2D* copyParam);
CUresult cuMemcpy2DUnaligned(const CUDA_MEMCPY2D* copyParam);
CUresult cuMemcpy2DAsync(const CUDA_MEMCPY2D* copyParam,
                         CUstream stream);
```

按照 copyParam 中指定的参数执行 2D 内存复制。CUDA_MEMCPY2D 结构定义如下：

```
typedef struct CUDA_MEMCPY2D_st {

    unsigned int srcXInBytes, srcY;
    CUmemorytype srcMemoryType;
        const void *srcHost;
        CUdeviceptr srcDevice;
        CUarray srcArray;
        unsigned int srcPitch;

    unsigned int dstXInBytes, dstY;
    CUmemorytype dstMemoryType;
        void *dstHost;
        CUdeviceptr dstDevice;
        CUarray dstArray;
        unsigned int dstPitch;
```

```
    unsigned int WidthInBytes;
    unsigned int Height;
} CUDA_MEMCPY2D;
```

其中：

- srcMemoryType 和 dstMemoryType 分别指定来源和目标的内存类型；Cumemorytype_enum 定义如下：

```
typedef enum CUmemorytype_enum {
    CU_MEMORYTYPE_HOST = 0x01,
    CU_MEMORYTYPE_DEVICE = 0x02,
    CU_MEMORYTYPE_ARRAY = 0x03
} CUmemorytype;
```

  如果 srcMemoryType 是 CU_MEMORYTYPE_HOST，则 srcHost 和 srcPitch 指定来源数据的（宿主）基址和每行字节数。忽略 srcArray。

  如果 srcMemoryType 是 CU_MEMORYTYPE_DEVICE，则 srcDevice 和 srcPitch 指定来源数据的（设备）基址和每行字节数。忽略 srcArray。

  如果 srcMemoryType 是 CU_MEMORYTYPE_ARRAY，则 srcArray 指定来源数据的句柄。忽略 srcHost、srcDevice 和 srcPitch。

  如果 dstMemoryType 是 CU_MEMORYTYPE_HOST，则 dstHost 和 dstPitch 指定目标数据的（宿主）基址和每行字节数。忽略 dstArray。

  如果 dstMemoryType 是 CU_MEMORYTYPE_DEVICE，则 dstDevice 和 dstPitch 指定目标数据的（设备）基址和每行字节数。忽略 dstArray。

  如果 dstMemoryType 是 CU_MEMORYTYPE_ARRAY，则 dstArray 指定目标数据的句柄。忽略 dstHost、dstDevice 和 dstPitch。

- srcXInBytes 和 srcY 指定要复制的来源数据的基址。
  对于宿主指针，开始地址为

```
void* Start =
          (void*)((char*)srcHost+srcY*srcPitch + srcXInBytes);
```

  对于设备指针，开始地址为

```
CUdeviceptr Start = srcDevice+srcY*srcPitch+srcXInBytes;
```

  对于 CUDA 数组，srcXInBytes 必须可由数组元素大小整除。

- dstXInBytes 和 dstY 指定要复制的目标数据的基址。
  对于宿主指针，开始地址为

```
void* dstStart =
          (void*)((char*)dstHost+dstY*dstPitch + dstXInBytes);
```

  对于设备指针，开始地址为

```
CUdeviceptr dstStart = dstDevice+dstY*dstPitch+dstXInBytes;
```

对于 CUDA 数组，dstXInBytes 必须可由数组元素大小整除。

- WidthInBytes 和 Height 指定要执行的 2D 复制的宽度（单位为字节）和高度。任何节距必须大于或等于 WidthInBytes。

如果任何节距大于允许的最大值（参见 E.2.6 中的 CU_DEVICE_ATTRIBUTE_MAX_PITCH），则 cuMemcpy2D()返回错误。

cuMemAllocPitch()传回与 cuMemcpy2D()配合使用的节距。对于设备内的内存复制（设备↔设备、CUDA 数组↔设备、CUDA 数组↔CUDA 数组），如果节距不是 cuMemAllocPitch()计算出来的，cuMemcpy2D()会失败。而 cuMemcpy2DUnaligned()没有此限制，但当 cuMemcpy2D()将返回错误代码时，运行速度可能会显著降低。

cuMemcpy2DAsync()是异步的，可以通过传递非零 stream 参数与流相关联。它仅适用于页面锁定的宿主内存，如果传递指向可分页内存的指针作为输入，则此函数返回错误。

E.9 纹理参考管理

E.9.1 cuTexRefCreate()

```
CUresult cuTexRefCreate(CUtexref* texRef);
```

创建纹理参考并在*texRef 中返回其句柄。创建之后，应用程序必须调用 cuTexRefSetArray()或 cuTexRefSetAddress()将此参考与已分配内存相关联。当通过此纹理参考读取内存时，其它纹理参考函数用于指定要使用的格式和解释方法（寻址、过滤等）。要将纹理参考与给定函数的纹理序号相关联，应用程序应该调用 cuParamSetTexRef()。

E.9.2 cuTexRefDestroy()

```
CUresult cuTexRefDestroy(CUtexref texRef);
```

销毁纹理参考。

E.9.3 cuTexRefSetArray()

```
CUresult cuTexRefSetArray(CUtexref texRef,
                          CUarray array,
                          unsigned int flags);
```

将 CUDA 数组 array 绑定到纹理参考 texRef。此函数也将取代与该纹理参考相关联的任何先前的地址或 CUDA 数组状态。flags 必须设置为 CU_TRSA_OVERRIDE_FORMAT。先前绑定到 texRef 的任何 CUDA 数组都将解除绑定。

E.9.4 cuTexRefSetAddress()

```
CUresult cuTexRefSetAddress(unsigned int* byteOffset,
                            CUtexref texRef,
                            CUdeviceptr devPtr,
                            int bytes);
```

将线性地址范围绑定到纹理参考 texRef。此函数也将取代与该纹理参考相关联的任何先前的地址或 CUDA 数组状态。先前绑定到 texRef 的任何内存都将解除绑定。

因为硬件强制对纹理基址执行对齐要求，所以 cuTexRefSetAddress()在*byteOffset 中传回必须应用于纹理拾取的字节偏移，以便从所需内存中读取数据。此偏移必须除以纹理元素大小并传递给读取该纹理数据的内核，以便其可以应用于 tex1Dfetch()函数。

如果设备内存指针从 cuMemAlloc()中返回，则偏移保证为 0，且 NULL 可以作为 ByteOffset 参数传递。

E.9.5 cuTexRefSetFormat()

```
CUresult cuTexRefSetFormat(CUtexref texRef,
                           CUarray_format format,
                           int numPackedComponents);
```

指定要由纹理参考 texRef 读取的数据的格式。format 和 numPackedComponents 与 CUDA_ARRAY_DESCRIPTOR 结构的 Format 和 NumChannels 成员完全类似：它们指定每个分量的格式和每个数组元素的分量数。

E.9.6 cuTexRefSetAddressMode()

```
CUresult cuTexRefSetAddressMode(CUtexref texRef,
                                int dim, CUaddress_mode mode);
```

为纹理参考 texRef 的某维度指定寻址模式 mode。如果 dim 为 0，则将该寻址模式应用于纹理拾取所使用的函数的第一个参数（参见 4.4.5）；如果 dim 为 1，则应用于第二个，以此类推。CUaddress_mode 定义如下：

```
typedef enum CUaddress_mode_enum {
     CU_TR_ADDRESS_MODE_WRAP = 0,
     CU_TR_ADDRESS_MODE_CLAMP = 1,
     CU_TR_ADDRESS_MODE_MIRROR = 2,
} CUaddress_mode;
```

注意，如果 texRef 绑定到线性内存，则此调用不产生任何效果。

E.9.7 cuTexRefSetFilterMode()

```
CUresult cuTexRefSetFilterMode(CUtexref texRef,
                               CUfilter_mode mode);
```

指定通过纹理参考 texRef 读取内存时要使用的过滤模式 mode。CUfilter_mode_enum 定义如下：

```
typedef enum CUfilter_mode_enum {
    CU_TR_FILTER_MODE_POINT = 0,
    CU_TR_FILTER_MODE_LINEAR = 1
} CUfilter_mode;
```

注意，如果 texRef 绑定到线性内存，则此调用不产生任何效果。

E.9.8 cuTexRefSetFlags()

```
CUresult cuTexRefSetFlags(CUtexref texRef, unsigned int Flags);
```

指定可选标识以控制通过纹理参考返回数据的操作方式。有效的标识包括：

- CU_TRSF_READ_AS_INTEGER，禁止把纹理从整数数据转化为[0, 1]区间的浮点数据这种默认设置；
- CU_TRSF_NORMALIZED_COORDINATES，禁止纹理坐标落在[0, Dim)区间(Dim 是 CUDA 数组的宽度或高度）这种默认设置。相反，纹理坐标使用[0, 1.0)区间引用数组维度的整个宽度。

E.9.9 cuTexRefGetAddress()

```
CUresult cuTexRefGetAddress(CUdeviceptr* devPtr, CUtexref texRef);
```

在*devPtr 中返回要绑定到纹理参考 texRef 的基址，如果此纹理参考未绑定到任何设备内存范围，则返回 CUDA_ERROR_INVALID_VALUE。

E.9.10 cuTexRefGetArray()

```
CUresult cuTexRefGetArray(CUarray* array, CUtexref texRef);
```

在*array 中返回由纹理参考 texRef 绑定的 CUDA 数组，如果此纹理参考未绑定到任何 CUDA 数组，则返回 CUDA_ERROR_INVALID_VALUE。

E.9.11 cuTexRefGetAddressMode()

```
CUresult cuTexRefGetAddressMode(CUaddress_mode* mode,
                                CUtexref texRef,
                                int dim);
```

在*mode 中返回纹理参考 texRef 的 dim 维度的寻址模式。当前，dim 的有效值只有 0 和 1。

E.9.12 cuTexRefGetFilterMode()

```
CUresult cuTexRefGetFilterMode(CUfilter_mode* mode,
                               CUtexref texRef);
```

在*mode 中返回纹理参考 texRef 的过滤模式。

E.9.13 cuTexRefGetFormat()

```
CUresult cuTexRefGetFormat(CUarray_format* format,
                           int* numPackedComponents,
                           CUtexref texRef);
```

在*format 和*numPackedComponents 中返回绑定到纹理参考 texRef 的 CUDA 数组的分量格式和分量数。如果 format 或 numPackedComponents 为空，则将其忽略。

E.9.14 cuTexRefGetFlags()

```
CUresult cuTexRefGetFlags(unsigned int* flags, CUtexref texRef);
```

在*flags 中返回纹理参考 texRef 的标识。

E.10 OpenGL 互操作性

E.10.1 cuGLInit()

```
CUresult cuGLInit(void);
```

初始化 OpenGL 互操作。此函数必须在任何其它 OpenGL 互操作之前调用。如果所需的 OpenGL 驱动程序工具不可用，则此函数可能失败。

E.10.2 cuGLRegisterBufferObject()

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

注册供 CUDA 访问的 ID 为 bufferObj 的缓冲对象。在 CUDA 可以映射此缓冲对象之前，必须调用此函数。注册之后，除作为 OpenGL 绘图命令的数据来源之外，此缓冲对象不能由任何 OpenGL 命令使用。

E.10.3 cuGLMapBufferObject()

```
CUresult cuGLMapBufferObject(CUdeviceptr* devPtr,
                             unsigned int* size,
                             GLuint bufferObj);
```

将 ID 为 bufferObj 的缓冲对象映射到当前 CUDA 上下文的地址空间中，并在*devPtr 和 *size 中返回结果映射的基指针和大小。

E.10.4 cuGLUnmapBufferObject()

```
CUresult cuGLUnmapBufferObject(GLuint bufferObj);
```

取消供 CUDA 访问的 ID 为 bufferObj 的缓冲对象的映射。

E.10.5 cuGLUnregisterBufferObject()

```
CUresult cuGLUnregisterBufferObject(GLuint bufferObj);
```

注销供 CUDA 访问的 ID 为 bufferObj 的缓冲对象。

E.11 Direct3D 互操作性

E.11.1 cuD3D9Begin()

```
CUresult cuD3D9Begin(IDirect3DDevice9* device);
```

初始化与 Direct3D 设备 device 的互操作。在 CUDA 可以映射 device 中的任何对象之前，必须调用此函数。然后，此函数可以映射 Direct3D 设备拥有的顶点缓冲，直到调用 cuD3D9End() 为止。

E.11.2 cuD3D9End()

```
CUresult cuD3D9End(void);
```

结束与先前 cuD3D9Begin() 初始化的 Direct3D 设备的互操作。

E.11.3 cuD3D9RegisterVertexBuffer()

```
CUresult cuD3D9RegisterVertexBuffer(IDirect3DVertexBuffer9* VB);
```

注册供 CUDA 访问的 Direct3D 顶点缓冲 VB。

E.11.4 cuD3D9MapVertexBuffer()

```
CUresult cuD3D9MapVertexBuffer(CUdeviceptr* devPtr,
                               unsigned int* size,
                               IDirect3DVertexBuffer9* VB);
```

将 Direct3D 顶点缓冲 VB 映射到当前 CUDA 上下文的地址空间中，并在*devPtr 和*size 中返回结果映射的基指针和大小。

E.11.5 cuD3D9UnmapVertexBuffer()

```
CUresult cuD3D9UnmapVertexBuffer(IDirect3DVertexBuffer9* VB);
```

取消供 CUDA 访问的顶点缓冲 VB 的映射。

E.11.6 cuD3D9UnregisterVertexBuffer()

```
CUresult cuD3D9UnregisterVertexBuffer(IDirect3DVertexBuffer9* VB);
```

注销供 CUDA 访问的顶点缓冲 VB。

E.11.7 cuD3D9GetDevice()

```
cudaError_t
cuD3D9GetDevice(CUdevice* dev, const char* adapterName);
```

在*dev 中返回与从 EnumDisplayDevices 或 IDirect3D9::GetAdapterIdentifier() 中获得的适配器名称 adapterName 相应的设备。

附录 F　纹理拾取

本附录给出一系列公式，这些公式用于根据各种纹理参考属性（参见 4.3.4）计算 4.4.5 中的纹理函数的返回值。

绑定到纹理参考的纹理对于一维纹理表示为 N 个纹理元素的数组 T，对于二维纹理表示为 $N \times M$ 个纹理元素。它使用纹理坐标 x 和 y 来拾取。

纹理坐标必须落在 T 的寻址范围之内，才能用于寻址 T。寻址模式指定如何将超出范围的纹理坐标 x 重新映射到有效范围。如果 x 未归一化，则只支持 clamp 寻址模式，如果 $x<0$，则替换 x 为 0，如果 $N \leq x$，则替换 x 为 N-1。如果 x 已归一化：

- 在 clamp 寻址模式中，如果 $x<0$，则替换 x 为 0，如果 $1 \leq x$，则替换 x 为 1-1/N，
- 在 wrap 寻址模式中，替换 x 为 *frac(x)*，其中

  frac(x)=*x*-*floor(x)*，*floor(x)*是不大于 x 的最大整数。

在本附录的剩余部分中，x 和 y 是重新映射到 T 的有效寻址范围的非归一化纹理坐标。x 和 y 由归一化纹理坐标 $x\hat{}$ 和 $y\hat{}$ 获得：$x=N x\hat{}$，$y=M y\hat{}$。

F.1 最近点采样

在此过滤模式中，纹理拾取返回值为：

- 对于一维纹理，*tex(x)=T[i]*，
- 对于二维纹理，*tex(x,y)=T[i,j]*，

其中，*i=floor(x)*，*j=floor(y)*。

图 F-1 显示了 *N*=4 的一维纹理的最近点采样。

对于整数纹理，纹理拾取的返回值可以重映射到[0.0, 1.0]（参见 4.3.4.1）。

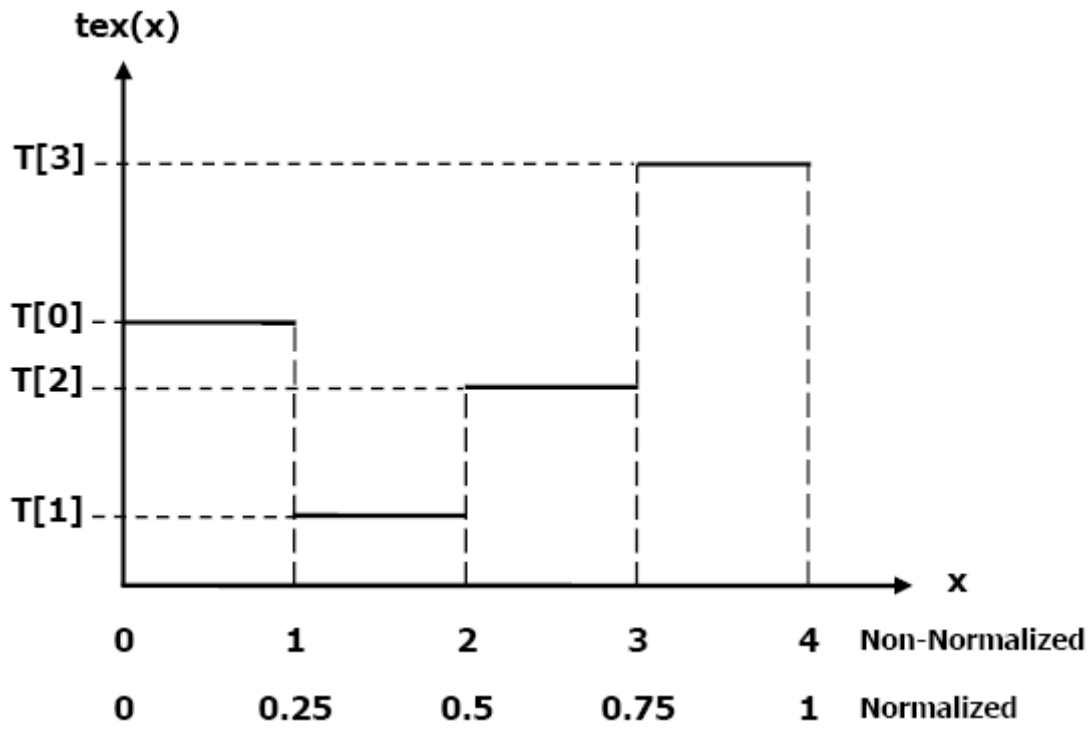


图 F-1. 四个纹理元素的一维纹理的最近点采样

F.2 线性过滤

在过滤模式（仅可用于浮点纹理）中，纹理拾取的返回值为：

- 对于一维纹理，$tex(x)=(1-\alpha)T[i]+\alpha T[i+1]$，
- 对于二维纹理，$tex(x,y)=(1-\alpha)(1-\beta)T[i,j]+\alpha(1-\beta)T[i+1,j]+(1-\alpha)\beta T[i,j+1]+\alpha\beta T[i+1,j+1]$，

其中：

- $i=floor(x_B)$， $\alpha=frac(x_B)$， $x_B=x-0.5$，
- $j=floor(y_B)$， $\alpha=frac(y_B)$， $y_B=y-0.5$。

α 和 β 使用包括 8 位小数值的 9-位定点格式存储。

图 F-2 显示了 $N=4$ 的一维纹理的最近点采样。

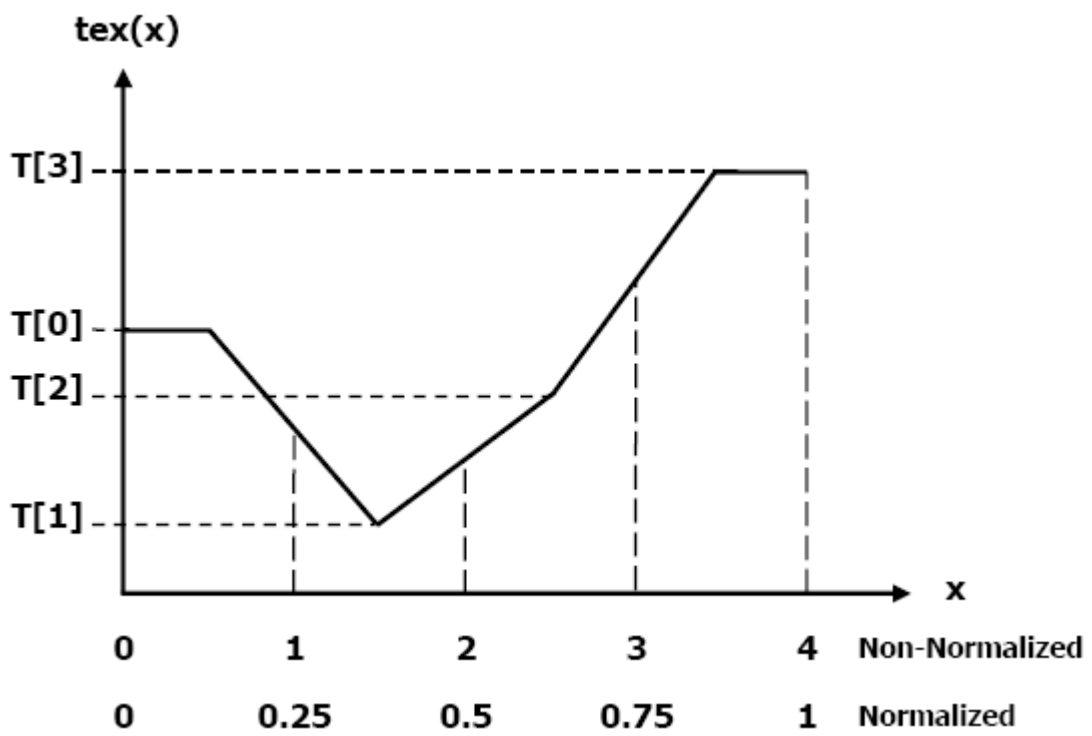


图 F-2. clamp 寻址模式中四个纹理元素的一维纹理的线性过滤

F.3 查找表

对于[0,*R*]区间内的 *x*，查找表 *TL*(*x*)实现为 $TL(x) = tex(\frac{N-1}{R}x + 0.5)$，从而确保 $TL(0) = T[0]$ 且 $TL(R) = T[N-1]$。

图 F-3 显示了从 *N*=4 的一维纹理中使用纹理过滤来实现 *R*=4 或 *R*=1 的查找表。

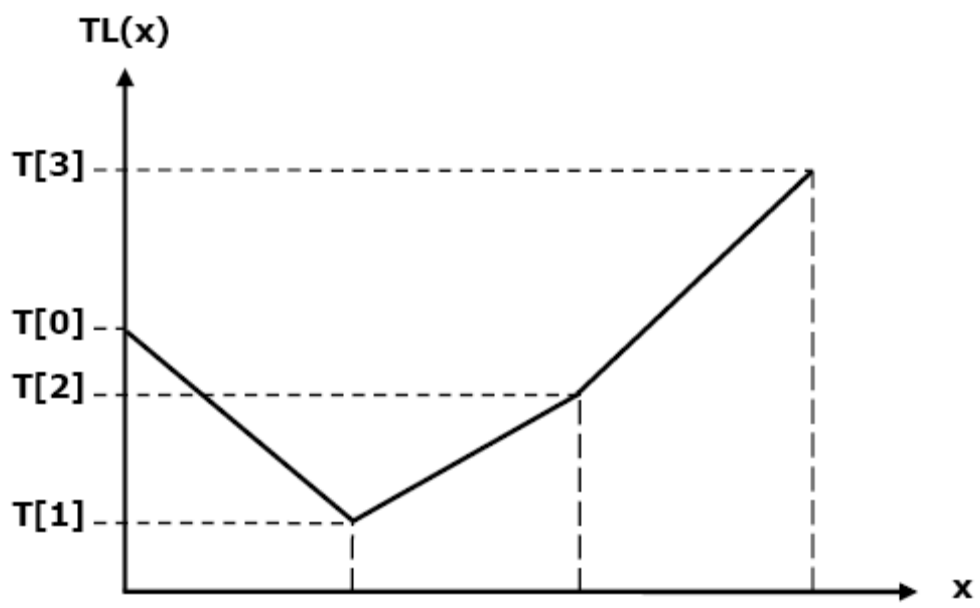


图 F-3. 使用线性过滤的一维查找表

致谢

刘伟峰先生对本指南中文版译稿全文进行了审校，在此表示衷心的感谢