NVIDIA

NVIDIA CUDA 编程指南

NVIDIA 技术文档（全译文）

Jan. 2008
Version 1.1

Chapter 1 介绍CUDA

1.1 作为一个并行数据计算设备的图形处理器单元仅仅几年的时间，可编程的图形处理器单元演变成为了一匹绝对的计算悍马，正如图1-1 所示。当极高的内存

带宽驱动多核处理器时，当今的GPU 为图型和非图型处理提供了难以置信的资源。

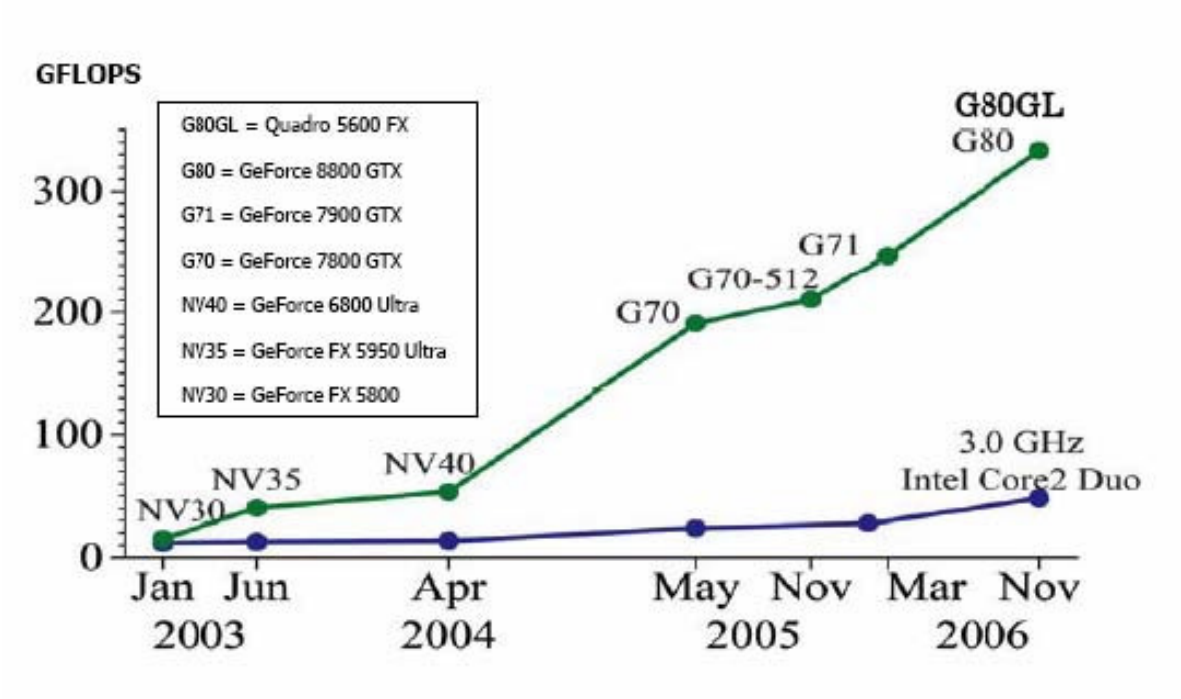


图1-1。CPU 和GPU 的每秒浮点运算

这个演变背后的主要原因是由于GPU 被设计用于高密度和并行计算，更确切地说是用于图形渲染。因此更多的晶体管被投入到数据处理而不是数据缓存和流量控制，图1-2 所示。

图1-2。 GPU 投入更多晶体管进行数据处理

更加具体地看，GPU 是特别适合于并行数据运算的问题－同一个程序在许多并行数据元素，并带有高运算密度（算术运算与内存操作的比例）。由于同一个程序要执行每个数据元素，降低了对复杂的流量控制要求；并且，因为它执行许多数据元素并且据有高运算密度，内存访问的延迟可以被忽略。

并行数据处理，意味着数据元素以并行线程处理。许多处理大量数据集，例如数组的应用程序可以使用一个并行数据的编程模型来加速计算。在3D 渲染上，大的像素集和顶点被映射到并行线程。同样，图像和媒体处理的应用程序例如着色的图像后处理，录像编码和解码，图像缩放比例，立体视觉，以及图像识别也可以映射图像块和像素到并行处理线程。实际上，在图像着色和处理领域外的许多算法同样可以通过并行数据处理得到加速，从一般信号处理或物理模拟到金融计算或者生物计算。

然而直到今天，尽管强大的计算能力包装进了GPU，而它对非图形应用的有效支持依然有限：
⌘ GPU 只能通过图型API 来编程，导致新手很难学习和非图形API 上很不充分的应用。
⌘ GPU DRAM 可以用一般方式下读取，GPU 程序可以从任何DRAM 部分收集数据元素。但不可写，在一般方式下的GPU 程序不能写入信息到DRAM 的任何部分，相比CPU 丧失了很多编程的灵活性。
⌘ 有些应用是由于DRAM 内存带宽而形成的瓶颈，未能充分利用GPU 的计算能力。

本文描述的是一个崭新的硬件和编程模型，它直接答复了这些问题并且展示了GPU 如何成为一个真正的通用并行数据计算设备。

1.2 CUDA： 一个在**GPU** 上计算的新架构CUDA（Compute Unified Device Architecture） 统一计算设备架构，在GPU 上发布的一个新的硬件和软件架
构，它不需要映射到一个图型API 便可在GPU 上管理和进行并行数据计算。从G80 系列和以后的型号都可以支持。操作系统的多任务机制通过几个CUDA 和图型应用程序协调运行来管理访问GPU。

CUDA 软件堆栈由几层组成，如图1-3 所示：一个硬件驱动程序，一个应用程序编程接口(API)和它的Runtime， 还有二个高级的通用数学库，CUFFT 和CUBLAS。硬件被设计成支持轻量级的驱动和Runtime 层面，因而提高性能。

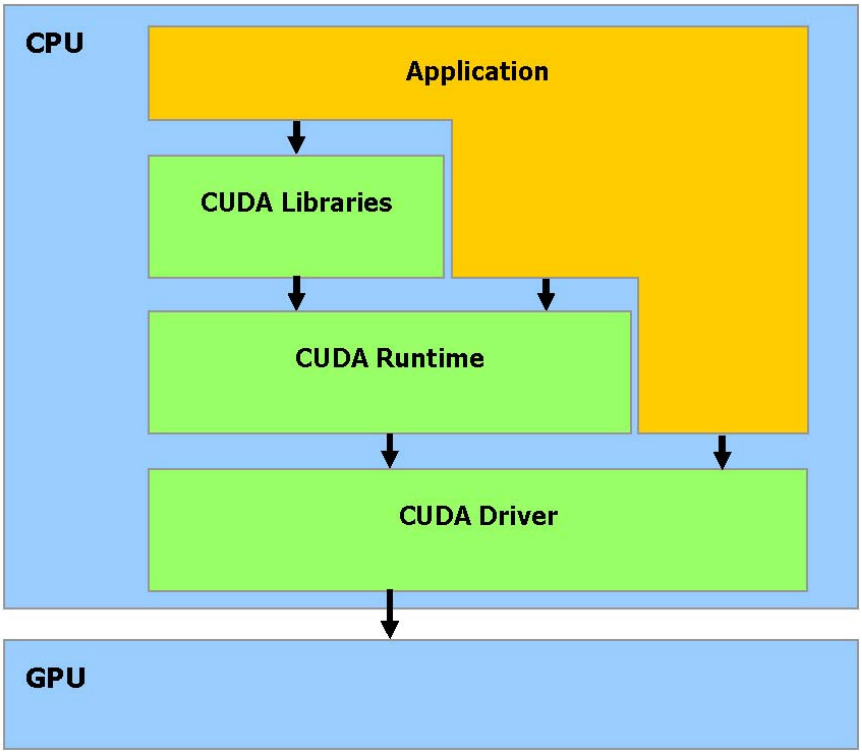


图1-3。统一计算设备架构的软件堆栈

CUDA API 更像是C 语言的扩展，以便最小化学习的时间。CUDA 提供一般DRAM 内存寻址方式：“发散”和“聚集”内存操作，如图1-4 所示。从而提供最大的编程灵活性。从编程的观点来看，它可以在DRAM的任何区域进行读写数据的操作，就像在CPU 上一样。

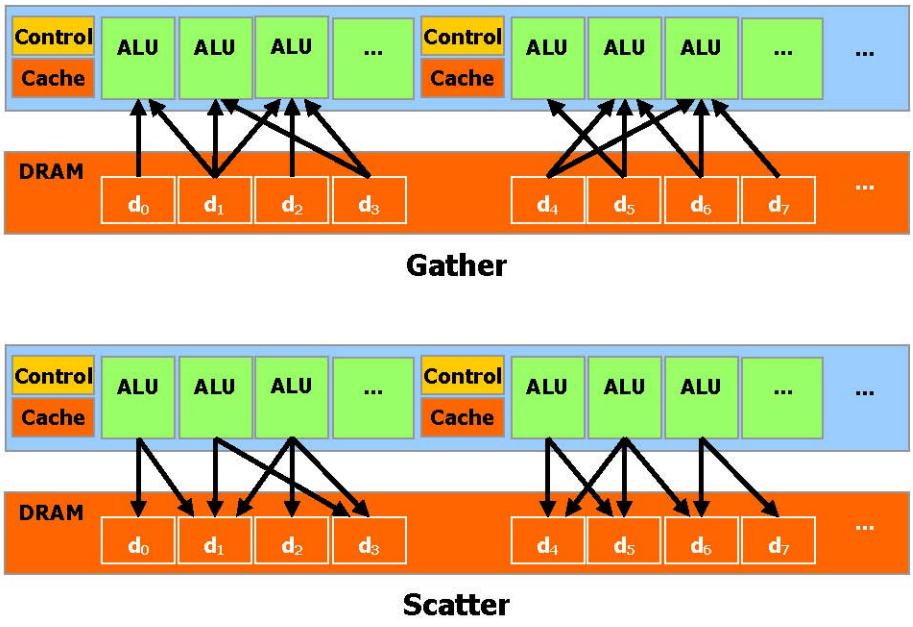
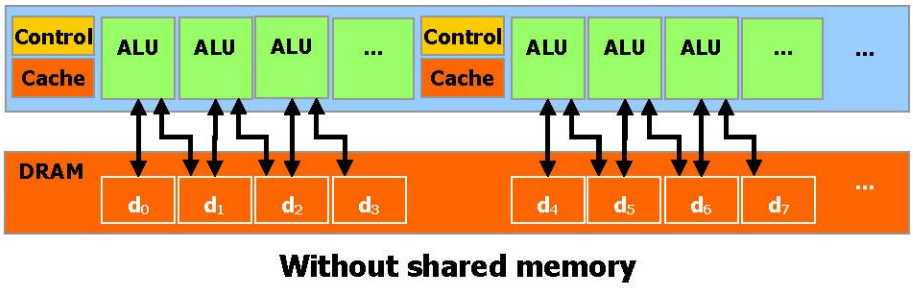
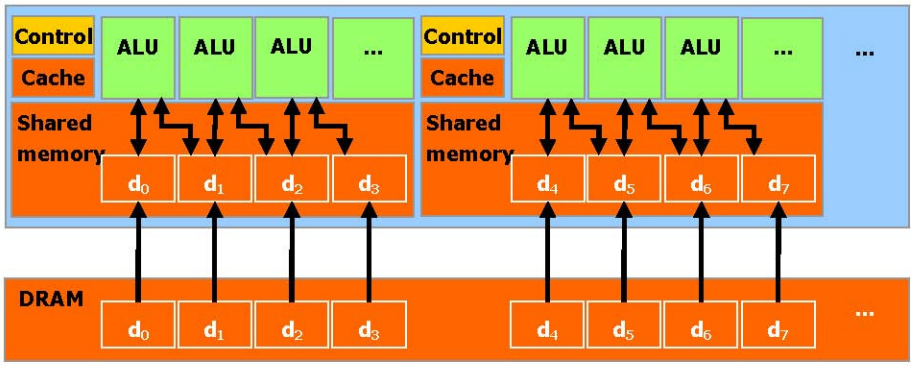


图1-4。 “聚集”和“发散”内存操作

CUDA 允许并行数据缓冲或者在On-chip 内存共享，可以进行快速的常规读写存取，在线程之间共享数据。如图1-5 所示，应用程序可以最小化数据到DRAM 的overfetch 和round-trips ，从而减少对DRAM 内存带宽的依赖。



Without shared memory



With shared memory

图1-5。共享内存使数据更接近ALU

Chapter 2 编程模型

2.1 一个超多线程协处理器

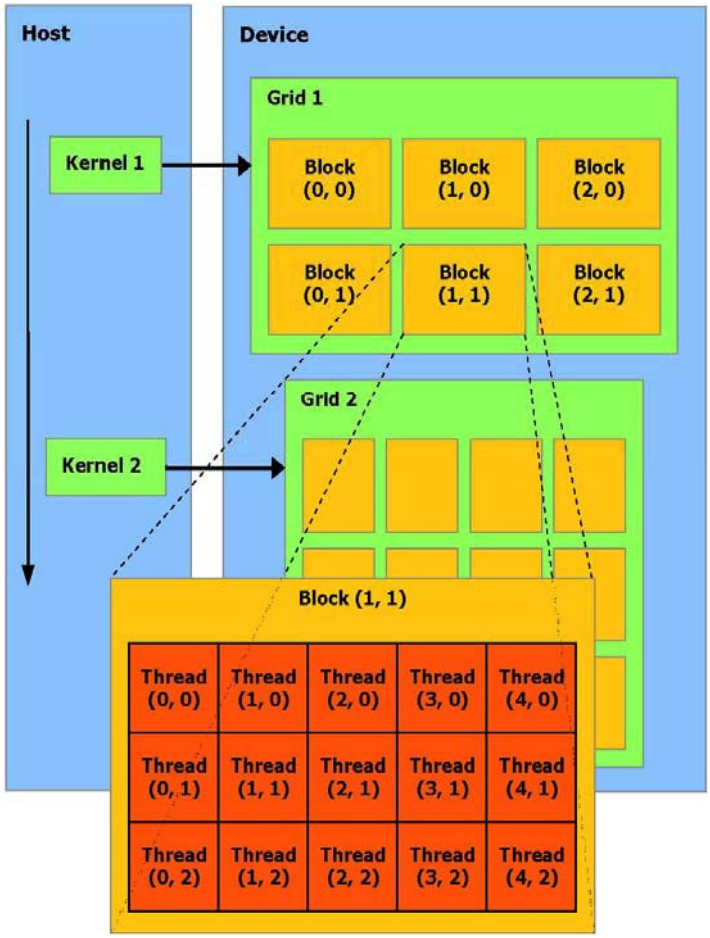
当通过CUDA 编译时，GPU 可以被视为能执行非常高数量并行线程的计算设备。它作为主CPU 的一个协处理器。换句话说，运行在主机上的并行数据和高密度计算应用程序部分，被卸载到这个设备上。

更准确地讲，一个被执行许多次不同数据的应用程序部分，可以被分离成为一个有很多不同线程在设备上执行的函数。达到这个效果，这个函数被编译成设备的指令集（kernel 程序），被下载到设备上。

主机和设备使用它们自己的DRAM，主机内存和设备内存。并可以通过利用设备高性能直接内存存取(DMA)的引擎（API）从一个DRAM 复制数据到其他DRAM。

2.2 线程批处理

线程批处理就是执行一个被组织成许多线程块的kernel，如图2-1 所示。



主机发送一个连续的kernel 调用到设备。每个kernel 作为一个由线程块组成的批处理线程来执行。

图2-1。线程批处理

2.2.1 线程块

一个线程块是一个线程的批处理，它通过一些快速的共享内存有效地分享数据并且在制定的内存访问中同步它们的执行。更准确地说，它可以在Kernel 中指定同步点，一个块里的线程被挂起直到它们所有都到达同步点。

每条线程是由它的线程ID 所确定，ID 是在块之内的线程编号。根据线程的ID 可以帮助进行复杂寻址，一个应用程序可以指定一个块作为一个二维或三维数组的任意大小，并且通过一个2 -或3-组件索引代替来指定每条线程。对于一个大小为 (D_x，D_y)二维块，线程的索引是(x, y)，这个线程ID 是(x + y D_x)。而对于一个三维的大小为 (D_x，D_y，D_z)的块，这个线程的索引是(x，y，z)， 线程的ID 是(x + y D_x + z D_xD_y)。

2.2.2 线程块栅格

一个块可以包含的线程最大数量是有限的。然而，执行同一个kernel 的块可以合成一批线程块的栅格，因此通过单一kernel 发送的请求的线程总数可以是非常巨大的。线程协作的减少会造成性能的损失，因为来自同一个栅格的不同线程块中的线程彼此之不间能通讯和同步。这个模式允许kernel 用不同的并行能力有效地运行在各种设备上而不用再编译：一个设备可以序列地运行栅格的所有块，如果它有非常少的并行特性，或者并行地运行，如果它有很多的并行的特性，或者通常是二者的组合。

. 每个块是由它的块ID 确定的，块的ID 是在栅格之内的块编号。根据块ID 可以帮助进行复杂寻址，一个应用程序可也以指定一个栅格作为任意大小的一个二维数组，并且通过一个2-组件索引替换来制定每个块。对于一个大小为 (D_x，D_y) 二维块，这个块的索引是(x，y)，块的ID 是(x + y D_x)。

2.3 内存模型

一条执行在设备上的线程，只允许通过如下的内存空间使用设备的DRAM 和On-Chip 内存，如图2-2 所示：

⌘ 读写每条线程的寄存器，

⌘ 读写每条线程的本地内存，

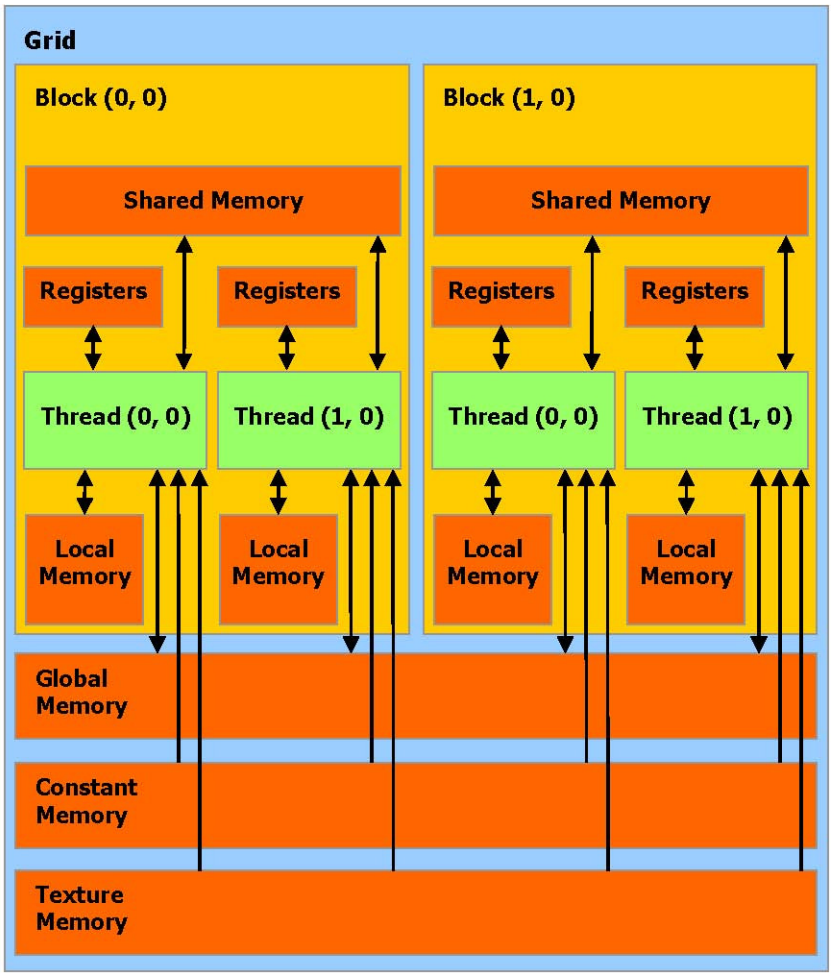
⌘ 读写每个块的共享内存，

⌘ 读写每个栅格的全局内存，

⌘ 只读每个栅格的常量内存，

⌘ 只读每个栅格的纹理内存。

全局，常量，和纹理内存空间可以通过主机或者同一应用程序持续的通过kernel 调用来完成读取或写入。全局，常量，和纹理内存空间对不同内存的用法加以优化。纹理内存同样提供不同的寻址模式，也为一些特殊的数据格式进行数据过滤。



线程通过一组不同范围的内存空间来使用设备的DRAM 和On-chip 内存。

图2-2。内存模型

Chapter 3 硬件实现

3.1 一组带有on-chip 共享内存的SIMD 多处理器

设备可以被看作一组多处理器，如图3-1 所示。每个多处理器使用单一指令，多数据架构(SIMD) ：在任何给定的时钟周期内，多处理器的每个处理器执行同一指令，但操作不同的数据。

每个多处理器使用四个以下类型的on-chip 内存：

- 每个处理器一组本地32 位寄存器，
- 并行数据缓存或共享内存，被所有处理器共享实现内存空间共享，
- 通过设备内存的一个只读区域，一个只读常量缓冲器被所有处理器共享，
- 通过设备内存的一个只读区域，一个只读纹理缓冲器被所有处理器共享，

本地和全局内存空间作为设备内存的读写区域，而不被缓冲。

每个多处理器通过纹理单元访问纹理缓冲器，它执行各种各样的寻址模式和数据过滤。

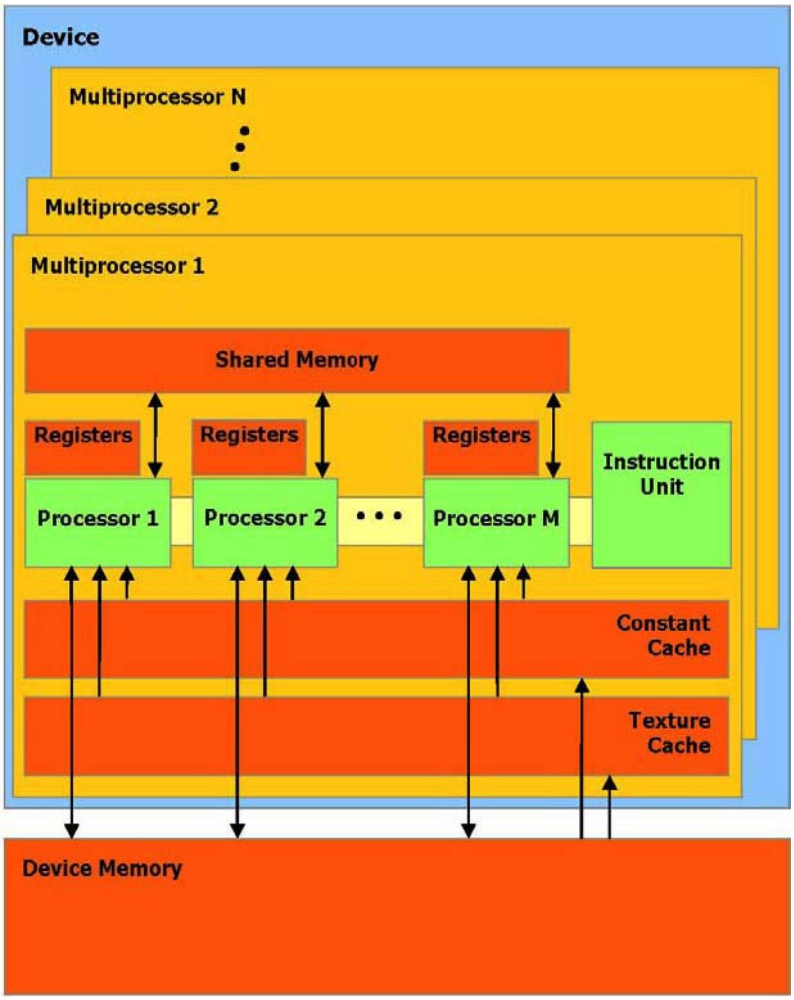


图3-1。硬件模型

3.2 执行模式

一个线程块栅格是通过多处理器规划执行的。每个多处理器一个接一个的处理块批处理。一个块只被一个多处理器处理，因此可以对驻留在on-chip 共享内存中的共享内存空间形成非常快速的访问。

一个批处理中每一个多处理器可以处理多少个块，取决于每个线程中分配了多少个寄存器和已知内核中每个时钟需要多少的共享内存，因为多处理器的寄存器和内存在所有的线程中是分开的。如果在至少一个块中，每个多处理器没有足够的寄存器或共享内存可用，那么内核将无法启动。

线程块在一个批处理中被一个多处理器执行，被称作active。每个active 块被划分成为SIMD 线程组，称为warps; 每一条这样的warp 包含数量相同的线程，叫做warp 大小，并且在SIMD 方式下通过多处理器执行; 线程调度程序周期性地从一条warp 切换到另一条warp，以达到多处理器计算资源使用的最大化。

块被划分成为warp 的方式总是相同的; 每条warp 包含连续的线程，线程索引从第一个warp 包含着的线程0 开始递增。

一个多处理器可以处理并发地几个块，通过划分在它们之中的寄存器和共享内存。更准确地说，每条线程可使用的寄存器数量，等于每个多处理器寄存器总数除以并发的线程数量，并发线程的数量等于并发块的数量乘以每块线程的数量。

在一个块内的warp 次序是未定义的，但通过协调全局或者共享内存的存取，它们可以同步的执行。如果一个通过warp 线程执行的指令写入全局或共享内存的同一位置，写的次序是未定义的。

在一个线程块栅格内的块次序是未定义的，并且在块之间不存在同步机制，因此来自同一个栅格的二个不同块的线程不能通过全局内存彼此安全地通讯。

3.3 计算兼容性

设备的计算兼容性由两个参数定义，主要版本号和次要版本号。设备拥有相同的主要版本号代表相同的核心架构。在附录A 中列出的设备全部是1.x（它们的主要版本号是1）。

次要版本号代表一些改进的核心架构，比如新的特性。不同计算兼容性的技术规格见附录A。

3.4 多设备

为一个应用程序使用多GPU 作为CUDA 设备，必须保证这些GPU 是一样的类型。如果系统工作在SLI 模式下，那么只有一个GPU 可以作为CUDA 设备，由于所有的GPU 在驱动堆栈中被底层的融合了。SLI 模式需要在控制面板中关闭，这样才能事多个GPU 作为CUDA 设备。

3.5 模式切换

GPU 指定一些DRAM 来存储被称作primary surface 的内容，这些内容被用于显示输出。如果用户改变显示的分辨率或者色深，那么primary surface 的存储需求量将改变。比如，如果用户将显示分辨率从1280x1024x32bit 到1600x1200x32bit ，系统必须指定7.68MB 的primary surface 而不在是5.24MB。（使全屏抗锯齿的应用程序需要更多的primary surface 空间）。另外，比如在Windows 中使用Alt+Tab 的切换，或者Ctrl+Alt+Del 的操作同样需要额外的primary surface 空间。

如果模式切换增加了primary surface 的内存空间，系统将占用CUDA 指定的内存空间，导致程序崩溃。

Chapter 4 应用程序编程接口（API）

4.1 一个C 语言的扩展

CUDA 编程接口的目标是为熟悉C 语言的用户提供一个相对简单的途径来编写设备执行程序。

它包括：

④ 一个小的C 语言扩展集，在第4.2 部分描述，允许程序员专注于在设备执行的原代码的部分;

④ 一个runtime 库分成：

- 一个主机组件，在第4.5 部分描述，它在主机上运行并且提供函数来控制和访问一个或多个计算设备;
- 一个设备组件，在第4.4 部分描述，它在设备运行并且提供特定设备的函数;
- 一个公共的组件，在第4.3 部分描述，它提供内置矢量类型和主机与设备编码都支持的C 标准库的一个子集。

应该强调的是，只有来自C 标准库的函数支持在设备上运行，是由公共Runtime 的组件提供的函数。

4.2 语言扩展

C 语言的扩展是四重的：

④ 函数类型限定句指定一个函数是否执行在主机或者执行在设备，和是否从主机或者从设备上调用(第4.2.1 部分);

④ 变量类型限定句指定设备上一个变量的内存位置(第4.2.2 部分);

④ 一个新的指令指定一个来自主机的kernel 如何在设备上执行 (第4.2.3 部分);

④ 四个内置的变量指定栅格和块的维数，还有块和线程的指标 (第4.2.4 部分)。

每个包含CUDA 语言扩展的源文件必须通过CUDA 编译器nvcc 编译，在第4.2.5 部分简要地加以描述。 nvcc 的一个详细的描述可以在一单独的文件中找到。

每一个扩展伴随的一些限定在每一节下面给予描述。 nvcc 将提供在一些违反这些限制时的一个错误或者一个警告，但有些违反无法被查出。

4.2.1 函数类型限定词

4.2.1.1 __device__

__device__限定词声明一个函数是：

④ 在设备上执行的，

④ 仅可从设备调用。

4.2.1.2 __global__

__global__限定词声明一个函数作为一个存在的kernel。这样的一个函数是：

④ 在设备上执行的，

④ 仅可从主机调用。

4.2.1.3 __host__

__host__限定词声明的函数是：

④ 在主机上执行的，

④ 仅可从主机调用。

它等于声明一个函数仅带有**__host__**限定词或者声明它没有任何**__host__**， **__device__**，或**__global__**限定词；在其他情况下这个函数仅为主机编译。

然而，**__host__** 限定词也可以用于与**__device__**限定词的组合，这种的情况下，这个函数是为主机和设备双方编译。

4.2.1.4 限定

__device__和**__global__**函数不支持递归。

__device__和**__global__**函数不能声明静态变量在它们体内。

__device__和**__global__**函数不能有自变量的一个变量数字。

__device__函数不可能取得它们的地址；另一方面，函数指向**__global__**函数是支持的。

不能一起使用__global__和__host__限定词。

__global__函数必须有void 的返回类型。任何调用到一个__global__函数必须指定它的执行配置，如在第4.2.3 部分所描述。对一个__global__函数的调用是同步的，意味着在设备执行完成前返回。

__global__函数参数目前是通过共享内存到设备的，并且被限制在256 个字节。

4.2.2 变量类型限定词

4.2.2.1 __device__

__device__限定词声明驻留在设备上的一个变量。

最多的一个其它类型限定词被定义在下面的三项里，可以与__device__一起共同用于进一步指定变量归属在哪些内存空间。如果它们都不存在，这个变量：

④ 驻留在全局内存空间，

④ 具有应用的生存期，

④ 从栅格内所有线程和从主机通过runtime 库是可访问的。

4.2.2.2 __constant__

__constant__限定词，与__device__一起随机使用，声明一变量：

④ 驻留在常量内存空间，

④ 具有应用的生存期，

④ 从栅格内所有线程和从主机通过runtime 库的是可访问的。

4.2.2.3 __shared__

__shared__限定词，与__device__一起选择使用，声明一个变量：

④ 驻留在线程块的共享内存空间中，

④ 具有块的生存期，

④ 只有块之内的所有线程是可访问的。

在线程中共享的变量有完全的顺序一致性。只有执行过一个__syncthreads（）函数，从其他线程的写才保证可见。除非变量被定义为可挥发的，否则只要前一个状态到达，编译器将自由的优化共享内存中的读写。

当声明一个在共享内存的变量作为一个外部数组时，例如

```
extern_shared_float shared[];
```

数组的大小是由发送时间(参见第4.2.3 部分)决定的。所有变量用这种方式声明的，开始于内存的同一个地址，因此在数组的变量布局必须通过offset(位移量)明确地加以控制。例如，如果你想要等于

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

在动态分配的共享内存，你可以用以下方式定义数组

```
extern __shared__ short 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

这些限定词不允许的一个函数之内的 `struct` 和 `union` 成员，形式参数和局部变量在主机上执行。

`__shared__`和**`__constant__`**变量隐含了静态存储。

`__device__`,`__shared__`和**`__constant__`**变量不能被用 extern 关键字定义为外部使用。

`__device__`和**`__constant__`**变量只允许在文件范围。

`__constant__`变量不能从设备上赋值，仅可以通过主机 runtime 函数从主机上赋值（参见 4.5.2.3 和 4.5.3.6）。

`__shared__`变量不能作为它们声明的一部分得到初始化。

一个不用任何限定词在设备码中声明的自变量通常驻留在一个寄存器中。尽管在某些情况下编译器可以选择在局部内存里安置它。通常在这个情况下，大型结构或者数组将消耗许多寄存器空间，并且编译器不能确定数组用常量数量建立了索引。ptx 汇编代码的检查(通过**`-ptx`** 或**`-keep`** 选项编译获得的)将指出，如果变量在第一个编译阶段期间被安置在局部内存，因此它将声明是使用**`.local`** 助记符或者使用**`ld.local`** 和 **`st.local`** 助记符访问的。假如它没有这样做，即使它们发现它为目标架构消耗太多寄存器空间，随后编译阶段仍然可以做出另外的决定。可以使用**`--ptxas-option=-v`** 生成局部内存使用量报告（**`lmem`**）。

执行在设备上的指针代码支持，当编译器可以解析它们是否指向全局内存空间或者局部内存空间。否则它们将被限制只指向寻址的内存或在全局内存中声明的空间。

解除一个指针在主机上执行的全局或共享内存中的代码，或者载设备上执行的主机内存代码，其结果产生一个未定义的行为，从而产生片断错误或者应用终止。

只能通过设备代码中的__device__,__shared__或__constant__变量来获取地址。__device__或__constant__变量的地址只能通过主机代码获得，cudaGetSymbolAddress()参见4.5.2.3 部分。

4.2.3 执行配置

所有**__global__**函数的调用必须指定执行配置。

执行配置定义了通常在设备执行的函数的栅格和块的维数，同样相关的stream（参见4.5.1.5 部分对streams 的描述）。它通过在函数名称和用括弧括起来的参数表之间插入表达式的形式**<<< Dg, Db, Ns, S>>>**来指定，如：

④ **Dg** 是类型**dim3**（参见4.3.1.2 部分）并且指定栅格的维数和大小，这样**Dg.x * Dg.y** 等于被发送的块的数量；

④ **Db** 是类型**dim3**（参见4.3.1.2 部分）并且指定每个块的维数和大小，这样**Db.x * Db.y * Db.z** 等于每个块的线程数量；

④ **Ns** 是类型**size_t** 并且指定在共享内存中的字节数量，这个共享内存是静态分配的内存之外的动态分配每个块的内存；这个动态分配的内存是被任何一个声明为外部数组的变量使用的，在4.2.2.3 部分会被涉及到；**Ns** 是一个默认为0 的可选参数。

④ **S** 是类型**cudaStream_t** 并且指定相关的stream；**S** 是一个默认为0 的可选数。

作为例子，函数被声明为

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

必须象这样调用：

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

执行配置的函数参数在调用前将被评估，且通过共享内存传至设备。

如果**Dg** 或**Db** 大于设备允许的最大值（参见附录A.1），或者**Ns** 的值大于（（设备共享内存的最大值）减去（共享内存中的静态分配的内存，函数参数，和执行配置））的值，函数将无法被调用。

4.2.4 内置变量

4.2.4.1 gridDim

这个变量是类型**dim3** （参见4.3.1.2 部分）并且包含栅格的维数。

4.2.4.2 blockIdx

这变量是类型**uint3** （参见4.3.1.1 部分）并且包含栅格之内的块索引。

4.2.4.3 blockDim

这变量是类型**dim3** （参见4.3.1.2 部分）并且包含在块的维数。

4.2.4.4 threadIdx

这变量是类型**uint3** （参见4.3.1.1 部分）并且包含块之内的线程索引。

4.2.4.5 限定

④ 内置变量不允许取得任何地址。

④ 不允许赋值到任何内置变量。

4.2.5 NVCC编译

nvcc 是编译CUDA 代码过程的编译器驱动程序的简称：它提供简单和熟悉的命令行选项，并且通过调用实施不同编译阶段汇集的工具来执行它们。

nvcc 的基本工作流程在于从主机代码中分离出设备代码，并且编译设备代码成为一个二进制格式的或*cubin* 对象。生成的主机代码输出，作为使用其他工具提交编译的C 代码，或者作为在最后编译阶段期间直接调用主机编译器的对象代码。

应用程序可以直接忽略生成的主机代码并使用CUDA 驱动程序API 加载在设备上的*cubin* 对象（参见第

4.5.3 部分），或者链接生成的主机代码，代码包括作为一个全局初始化的数据数组的*cubin* 对象，并且包含一个执行配置语法的转换，和进入必要的CUDA Runtime 的起始码（第4.2.3 部分），来加载和发送每个编译了的Kernel（参见第4.5.2 部分）。

编译器处理CUDA 源文件的前端部分完全遵照C++的语法。主机代码完全支持C++。但是设备代码只支持C++中的C 子集；在基本块中的C++的特性，比如：classes, inheritance， 或者变量的声明是不支持的。作为使用C++语法的结果，void 指针（例如，通过**malloc()**返回）在没有使用typecast 的情况下不能分配给non-void 的指针。

nvcc 的一个详细描述可在一个单独的文件中找到。下面介绍NVCC 两个编译器侦测。

4.2.5.1 __noinline__

默认下，**__device__**函数总是inline 的。**__noinline__**函数可以作为一个非inline 函数的提示。函数本身必须放在调用的文件中，编译器不能保证函数带有指针参数和函数带有大量参数表的**__noinline__**的限定词正常工作。

4.2.5.2 #pragma unroll

默认下，编译器为已知的行程计数展开小型循环。**#pragma unroll** 可以侦测和控制任何展开的循环。它必须放在这个循环之前，并只作用于这个循环。同时，可以通过一个参数指定循环可以展开多少次。

例如：

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

循环将展开5 次。请自行确定展开动作不会影响到程序的正确性。

如果**#pragma unroll** 后面没有附值，当行程计数为常数时，循环完全展开，否则不会展开。

4.3 公共Runtime 组件

公共的Runtime 的组件可同时被主机和设备函数使用。

4.3.1 内置矢量类型

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

这些矢量类型是源于基本的整型和浮点类型。它们是结构和第1，第2，第3，还有第4 个组件可通过域 x, y, z, 和 w 分别访问。它们全都带有一个来自格式**make_<type name>**的构造器函数；例如，

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

通过赋值**(x，y)**创建一个类型**int2** 的矢量。

4.3.1.2 dim3 类型

这个类型是基于**uint3** 的用于指定维数的整型矢量类型。当定义一个类型**dim3** 的变量时，所有剩余的非特指的组件初始化为1。

4.3.2 数学函数

在附录中的表B-1 中包含了一张当前支持的C/C++ 标准库数学函数的全面清单，随同它们一起的是在设备上执行时各自的错误范围。

在主机上执行时，一个给定的函数使用C runtime 执行。

4.3.3 时间函数

```
clock_t clock();
```

每个时钟周期递增下的计数器的返回值。

在 kernel 开始和的结束时采样这个计数器，取得这个二个采样的差，并且记录着每线程每时钟周期通过设备完全地执行线程取得的结果，而不是设备执行线程指令时实际花费的时钟周期数量。前着的数字是比后者更大是因为线程是被切成时间段的。

4.3.4 纹理类型

CUDA 支持硬件纹理渲染的一个子集，通过GPU 为图形使用纹理内存。通过纹理内存读取数据相比全局内存有很多性能上的优势（参见5.4 部分）。

纹理内存通过一个叫texture fetches 的设备函数从kernel 读取（参加4.4.5）。Texture fetch 的第一个参数指定一个叫texture referece 的对象。

Texture reference 定义纹理内存的哪一个部分被fetch 。在被kernel 使用之前，它必须通过主机的runtime 函数（参见部分4.5.2.6 和4.5.3.9） 绑定到一些内存区域。一些texture reference 也许绑定在同一个纹理下或者纹理映射的内存中。

Texture reference 有一些属性。其中的一个就是，它可以通过一个纹理坐标指定纹理是否使用一维数组寻址，或者通过两个纹理坐标指定纹理是否使用二维数组寻址。数组的元素被简称为texels，texture elements 的缩写。

另一个属性是，为纹理的fetch 定义输入输出数据类型。

4.3.4.1 Texture Reference 声明

一些texture reference 的属性是固定的，它们在声明texture reference 时被指定。一个texture reference 在文件范围作为类型**texture** 的一个变量被声明：

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

此时：

④ **Type** 指定的数据类型是在拾取纹理时返回的；**Type** 被限定在基本的整型和浮点类型和在4.3.1.1 部分定义的所有的矢量类型；

④ **Dim** 指定texture reference 的维数，它等于1 或2；**Dim** 的是默认为1 的一个可选自变量；

④ **ReadMode** 等于**cudaReadModeNormalizedFloat** 或**cudaReadModeElementType**；如果它是**cudaReadModeNormalizedFloat** 而且**Type** 是一个16-bit 或8-bit 的整型类型，实际上返回的值

将被看作浮点类型，unsigned 的整型类型被映射到[0.0，1.0]，signed 的整型类型被映射到[-1.0，1.0]； 例如，一个带有值`0xff` 的无符号的8-bit 纹理元素读作1；如果它是 **`cudaReadModeElementType`** ，将不执行转换；

`ReadMode` 是一个默认到**`cudaReadModeElementType`** 的可选自变量。

4.3.4.2 Runtime Texture Reference 属性

另外一些texture reference 的属性是不固定的，它们可以通过主机的runtime 改变(参见部分4.5.2.6 runtime API 和4.5.3.9 driver API)。它们可以指定纹理坐标是否是normalized ，寻址模式，和纹理过滤。

默认下，纹理通过浮点数坐标[0，N)引用，N 是关于坐标在空间上纹理的大小。例如，一个64x32 大小的纹理拥有坐标范围x 轴[0，63]和y 轴[0，31]。Normalized 的纹理通过坐标[0.0，1.0)引用，而不是[0，N)。因此，同样的64x32 纹理将被指向normalized 的坐标x 轴[0.0，1.0)和坐标y 轴[0.0，1.0)。Normalized 的纹理坐标天生适合一些应用程序，例如纹理坐标独立于纹理大小。

寻址模式定义了，当纹理坐标超出范围后会怎样。当使用unnormalized 的纹理坐标时，纹理坐标超出范围[0， N)时，小于0 的值被设成0，大于N 的值被设成N-1。。当使用normalized 的纹理坐标时，纹理坐标范围被限制在[0.0，1.0)。对于normalized 的纹理坐标，同样指定了"warp"寻址。Warp 寻址通常被用于，当纹理包含一个周期性的信号时。它只作用于纹理坐标的分数部分；例如，1.25 将被看作0.25，-1.25 将被看作0.75。

线性纹理过滤只能用在纹理被设置为返回浮点数据的情况下。它在邻近的texel 中执行一个低精度的插值。texel 周边的纹理拾取地址将被读取，并基于texel 所在的纹理坐标返回插值的纹理拾取值。简单的插值执行在一维纹理中，bilinear 插值执行在二维纹理中。

附录F 有关于纹理拾取的更多细节。

4.3.4.3 线性内存纹理操作对比CUDA 数组

一个纹理可以被划在线性的内存中或者一个CUDA 数组中（参见部分4.5.1.2）。

纹理分配在线性内存中：

④ 只有维数为1 时；

④ 不支持纹理过滤；

④ 只能使用non-normalized 纹理坐标寻址；

④ 不能支持不同的寻址模式：超出范围的纹理访问返回0。

4.4 设备Runtime 组件

设备runtime 的组件只能用于设备函数。

4.4.1 数学函数

对于表B-1 中的某些的函数，它们在设备Runtime 的组件中有低准确性而更快速的版本；它有相同的加__前缀（例如**__sin(x)**）。这些函数在表B-2 里列出。

编译器有一个选项（**-use_fast_math**）来强制每个函数编译到它的不太准确的副本。

4.4.2 同步函数

```
void __syncthreads();
```

在一个块内同步所有线程。一旦所有线程到达了这点，恢复正常执行。

__syncthreads()通常用于调整在相同块之间的线程通信。当在一个块内的有些线程访问相同的共享或全局内存时，对于有些内存访问潜在着read-after-write, write-after-read, 或者 write-after-write 的危险。这些数据危险可以通过同步线程之间的访问得以避免。

__syncthreads() 允许放在条件代码中，但只有当整个线程块有相同的条件贯穿时，否则代码执行可能被挂起或导致没想到的副作用。

4.4.3 类型转换函数

下面函数的后缀指定IEEE-754 的舍入模式：

rn 是求最近的偶数，

rz 是逼近零，

ru 是向上舍入（到正无穷），

rd 是向下舍入（到负无穷）。

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

用指定的舍入模式转换浮点参数到整型。

```
Unsignde int __float2unit_[rn,rz,ru,zd](float);
```

用指定的舍入模式转换浮点参数到无符号整型。

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

用指定的舍入模式转换整型参数到浮点数。

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

用指定的舍入模式转换无符号整型参数到浮点数。

4.4.4 Type Casting 函数

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

在整型自变量上执行一个浮点数的type cast，保持值不变。例如，**__int_as_float(0xC0000000)** 等于**-2**。

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

在浮点自变量上执行的一个整型的type cast ，保持值不变。例如，**__float_as_int (1.0f)** 等于**0x3f800000**。

4.4.5 纹理函数

4.4.5.1 设备内存纹理操作

设备内存中的纹理通过**tex1Dfetch()**函数访问，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);
```

这些函数通过纹理坐标**x** 拾取线性内存中绑定到texture reference **texRef** 的区域。对于整型来说，不允许纹理过滤和选择寻址模式。对于这些函数，可能需要将整型数升级到32-bit 浮点数。

下面的函数展示了2-和4-元组的支持：

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

通过纹理坐标**x** 拾取线性内存中绑定到texture reference **texRef** 的区域。

4.4.5.2 CUDA 数组纹理操作

从CUDA 数组中的纹理通过**tex1D()**或**tex2D()**函数访问：

```
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);
```

这些函数通过纹理坐标**x** 和**y** 拾取CUDA 数组中绑定到texture reference **texRef** 的区域。Texture reference
的编译时（固定的）和运行时（可变的）的属性决定了，坐标如何被解释，纹理拾取时将有哪些处理发生，和纹理拾取返回的值（参见部分4.3.4.1 和4.3.4.2）。

4.4.6 原子函数

只有计算兼容性为1.1 的设备才可以使用原子函数。它们在附录C 中列出。

原子函数在全局内存中的一个32-bit 字中执行一个读-修改-写的原子操作。例如，`atomicAdd()`在全局内存中的同一个地址读取一个32-bit 字，加一个整型进去，并写回结果到同一个地址。所谓“原子”就是保证操作不会干扰其它线程。在操作完成之前，其它线程也无法访问这个地址。

原子操作只能用于32-bit 有符号和无符号的整型数。

4.5 主机Runtime 组件

主机Runtime 的组件只能被主机函数使用。

它提供函数来处理：

④ 设备管理，

④ Context 管理，

④ 内存管理，

④ 编码模块管理，

④ 执行控制，

④ Texture reference 管理，

④ OpenGL 和Direct3D 的互用性。

它由二个API 组成：

④ 一个低级的API 调用CUDA 驱动程序API

④ 一个高级的API 调用的CUDA runtime API ，在CUDA 驱动程序API 之上运行的API。

这些API 是互相排斥：一个应用程序应该选择其中之一来使用。

CUDA runtime 通过提供固有的初始化，context 管理，和模块管理减轻了设备代码的管理。Nvcc 生成的C 主机代码基于CUDA runtime(参见第4.2.5 部分)，因此应用程序连接这个代码必须使用CUDA runtime API 。

相反， CUDA 驱动程序API 要求更多的代码，使编程和调试更加困难，但它提供更好的控制，并且是语言独立的，因为它只处理cubin 对象(参见第4.2.5 部分)。尤其是使用CUDA 驱动程序API 配置和启动Kernel 更加困难，因为执行配置和kernel 参数必须指定外在的函数调用，来替换执行配置语法（参见第4.2.3 部分）。同样的，设备仿真(参见4.5.2.9 部分)不能与CUDA 驱动程序API 一起工作。

CUDA 驱动程序API 通过**cuda** 动态库提供，所有它的进入点带有前缀**cu**。
CUDA runtime API 通过**cudart** 动态库提供，所有它的进入点带有前缀**cuda**。

4.5.1 公共概念

4.5.1.1 设备

两个API 都提供了函数来枚举在系统上可使用的设备，查询它们的属性，并选择它们中的一个来执行kernel （参见第4.5.2.2 部分和第4.5.3.2 部分）。

一些主机线程可以在同一设备上执行设备代码，但从设计角度上看，一个主机线程只能在一个设备上执行设备代码。因此，多主机线程需要在多个设备上执行设备代码。另外，任何在一个主机线程中通过runtime创建的CUDA 源文件不能被其它主机线程使用。

4.5.1.2 内存

设备内存可被分配到线性内存或者是CUDA 数组。
在设备上的线性内存使用32-bit 地址空间，因此单独分配的实体可以通过指针的互相引用，例如，在一个二元的树结构中。
CUDA 数组是针对纹理拾取优化的不透明的内存布局。它们是一维或二维的元素组成的，每个有1 个，2 个或者4 个组件，每个组件可以是有符号或无符号8-，16- 或32-bit 整型，16-位浮点(仅通过CUDA 驱动程序API 支持)，或32 位浮点。CUDA 数组只能通过kernel 纹理拾取读取。

通过主机的内存复制函，数线性内存和 CUDA 数组都是可读和可写的，参见 4.5.2.3 和 4.5.3.6 部分的描述。

不同于由**malloc()**函数分配的pageable 主机内存，主机runtime 同样提供可以分配和释放page-locked 主机内存的函数（参见附录D.5.6 和D.5.7，E.8.5 和E.8.6 部分）。如果主机内存被分配为page-locked ，使用page-locked 内存的优势是，主机内存和设备内存之间的带宽将非常高。但是，分配过多的 page-locked 内存将减少系统可用物理内存的大小，从而降低系统整体的性能。

4.5.1.3 OpenGL Interoperability

OpenGL 缓冲器对象可以被映射到CUDA 地址空间，使CUDA 能够读取被OpenGL 写入的数据，或者使CUDA 能够写入被OpenGL 消耗的数据。 4.5.2.7 部分描述了在runtime API 下如何使用，4.5.3.10 部分描述了驱动API 下如何使用。

4.5.1.4 Direct3D Interoperability

Direct3D 9.0 顶点缓冲器可以被映射到CUDA 地址空间，使CUDA 能够读取被Direct3D 写入的数据，或者使CUDA 能够写入被Direct3D 消耗的数据。4.5.2.8 部分描述了在runtime API 下如何使用，4.5.3.11 部分描述了驱动API 下如何使用。

一个CUDA context 每次只可以互用一个Direct3D 设备，通过把begin/end 函数括起来调用。参见4.5.2.8 和4.5.3.11 部分的描述。

CUDA context 和Direct3D 设备必须建立在同一个GPU 上。可以通过查询与CUDA 设备是否关联使用Direct3D的适配器来确保。对于runtime API 使用**cudaD3D9GetDevice()**（参见附录D.9.7），对于驱动API 使用**cuD3D9GetDevice()**（参见附录E.11.7）。

Direct3D 设备必须使用**D3DCREATE_HARDWARE_VERTEXPROCESSING** 标志创建。

CUDA 尚不支持：

④ 除Direct3D 9.0 之外的版本，

④ 除顶点缓冲器之外的Direct3D 对象。

cudaD3D9GetDevice()和**cuD3D9GetDevice()**同样可以确保Direct3D 设备和CUDA context 建立在不同的设备上，比如Direct3D 的loading balance 和CUDA 的over interoperability 。

4.5.1.5 异步的并发执行

为了促使主机和设备之间的并发执行，一些runtime 函数是异步的：在设备完成请求的任务之前，控制权将交还给应用程序。

④ Kernel 通过**__global__** 函数或**cuGridLaunch()**和**cuGridLaunchAsync()**启动；

④ 执行内存拷贝的函数需要后缀**Async**；

④ 执行设备到设备内存拷贝的函数；

④ 设置内存的函数。

一些设备也可以执行page-locked 主机内存和设备内存之间并发的拷贝。应用程序可以与**CU_DEVICE_ATTRIBUTE_GPU_OVERLAP**（参见E.2.6）一起通过**cuDeviceGetAttribute()**查询是否有这个功能。这个功能目前只支持内存拷贝，且是通过**cudaMallocPitch()**（参见4.5.2.3 部分）或**cuMemAllocPitch()**（参见4.5.3.6 部分）分配的不包括CUDA 数组或2D 数组的。

应用程序通过流（stream） 管理并发。一个流是有序执行的操作序列。另一方面，不同的流也许会遵循其它或并行的乱序执行。

一个流被定义为，通过建立流对象且指定它的流参数到kernel 的启动序列和主机到设备的内存拷贝。4.5.2.4 部分描述了在runtime API 下如何使用，4.5.3.7 部分描述了驱动API 下如何使用。

只有当全部的操作已完成的情况下：包括操作本事是流的一部分，和在完成之前没有随之的操作。一个带有零参数的流才开始，例如：任何kernel 的启动，内存的设置或内存的拷贝。

cudaStreamQuery()和 **cuStreamQuery()**提供一个方法使应用程序查询一个流中的全部操作是否完成了（分别参见 D.3.2 和 E.5.2 部分）。**cudaStreamSynchronize()** 和 **cuStreamSynchronize()**提供一个方法强制应用程序等待，直到流中的全部操作完成（分别参见 E.5.2 和 E.5.3 部分）。

同样的，**cudaThreadSynchronize()**和**cuThreadSynchronize()**应用程序可以强制runtime 等待，直到全部设备任务完成（分别参见D.2.1 和E.3.5 部分）。为了避免速度下降，这些函数最好用于时间目的的，孤立启动的或内存拷贝失败的情况。

runtime 同样提供一个方法更近的监控设备的进程。在精确的时间下，让应用程序异步的纪录程序中任意一点的事件，并可查询这些事件是何时被纪录的。4.5.2.5 部分描述了在runtime API 下如何使用，4.5.3.8 部分描述了驱动API 下如何使用。

不同流中的两个操作不能并发的执行，无论是page-locked 主机内存分配，设备内存的分配，设备内存的设置，设备到设备的内存拷贝，或它们之间的事件纪录。

程序员可以通过设置**CUDA_LAUNCH_BLOCKING** 环境变量为1，来全局的关闭异步执行，对于所有系统上的CUDA 应用程序。这个功能仅限debug 用途，不要用于增加软件产品的可靠性。

4.5.2 Runtime API

4.5.2.1 初始化

没有明确针对RuntimeAPI 的初始化函数；它在第一次Runtime 函数调用时初始化。需要注意的是，何时适时的调用Runtime 函数，和何时说明从第一次调用进入Runtime 的错误代码。

4.5.2.2 设备管理

附录D.1 的函数用来管理系统中的设备。

cudaGetDeviceCount()和**cudaGetDeviceProperties()**提供一个方法来枚举这些设备和获得它们的属性：

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

cudaGetDeviceProperties(&deviceProp,device);
}
```

cudaSetDevice()用来选择相关于主机线程的设备：

```
cudaSetDevice(device);
```

一个设备必须在任何**__global__**函数或者所有来自附录D 的任何函数调用之前选择；否则，device 0 自动地被选择，并且所有随后的设备选择将是无效的。

4.5.2.3 内存管理

附录D.5 的函数用来分配和释放设备内存，访问在全局内存中任意声明的变量分配的内存，和从主机内存到设备内存之间的数据传输。

线性内存通过**cudaMalloc()**或**cudaMallocPitch()**分配，通过**cudaFree()**释放。

下面的代码演示，在线性内存中分配一个256 个浮点元素的数组：

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

分配2D 数组建议使用**cudaMallocPitch()**， 从而保证访问行地址，或拷贝2D 数组到设备内存的其它区域的最佳性能。返回的pitch 必须用来访问数组元素。下面的代码演示，分配一个宽x 高带有浮点数的2D 数组，和在设备代码中如何循环数组元素：

```
// 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 数组通过**cudaMallocArray()**分配，通过**cudaFreeArray()**释放。**cudaMallocArray()**需要一

个格式的解释，通过**cudaCreateChannelDesc()**建立。

下面的代码演示，分配一个宽x 高带有一个32-bit 浮点数的2D 数组：

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

cudaGetSymbolAddress()用来获得指向全局内存中一个声明的变量分配的内存地址。分配内存的大小用cudaGetSymbolSize()取得。

附录D.5 列出了不同的函数用来拷贝内存，包括用**cudaMalloc()**分配的线性内存，用**cudaMallocPitch()** 分配的线性内存，CUDA 数组，全局变量分配的内存或常驻内存。

下面的代码演示，拷贝一个2D 数组到之前例子中分配的CUDA 数组：

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

下面的代码演示，拷贝一些主机内存数组到设备内存：

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

下面的代码演示，拷贝一些主机内存数组到常驻内存：

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

4.5.2.4 流管理

附录D.3 的函数用来创建和销毁流，并且判定一个流中的所有操作是否完成。

下面的代码演示，创建两个流：

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

下面的代码演示，每一个流被依次定义执行一次：从主机到设备的内存拷贝，kernel 启动，从设备到主机的内存拷贝。

```
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();
```

每个流拷贝部分输入数组**hostPtr** 到设备内存中的输入数组**inputDevPtr**， 通过调用**myKernel()**在设备上处理**inputDevPtr**， 并拷贝结果**outputDevPtr** 到**hostPtr** 的同一个部分。使用两个流处理**hostPtr** 允许内存拷贝从一个流覆盖到另一个流。对于任何覆盖，**hostPtr** 必须指向page-locked 的主机内存：

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

cudaThreadSynchronize()在最后被调用，以确保在继续处理之前全部的流已经完成。

4.5.2.5 事件管理

附录D.4 的函数用来创建，纪录和销毁事件，并可查询两个事件之间花费的时间。

下面的代码演示，建立两个事件：

```
cudaEvent_tstart,sto;
cudaEventCreate(&stat);
cudaEventCreate(&stop);
```

这些事件可以用来计算之前代码的时间：

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

4.5.2.6 Texture Reference 管理

附录D.6 的函数用来管理texture reference 。

纹理类型是由高级API 定义的公开的结构，texture Reference 类型是由低级API 定义的，例如：

```
struct textureReference
{
  int                    normalized;

   enum cudaTextureFilterMode   filterMode;

   enum cudaTextureAddressMode  addressMode[2];

   struct cudaChannelFormatDesc channelDesc;

}
```

④ **normalized** 指定纹理坐标是否是normalized； 如果它不是零，纹理中的全部元素拥有纹理坐标**[0,1]**， 而不是**[0,width-1]**或**[0,height-1]**，其中**width** 和**height** 是纹理的大小；

④ **filterMode** 指定过滤模式：基于输入的纹理坐标，计算纹理拾取的值是如何返回的；过滤模式等于**cudaFilterModePoint** 或**cudaFilterModeLinear**；如果是**cudaFilterModePoint**，返回的值等于最接近于输入纹理坐标的texel；如果是**cudaFilterModeLinear**，返回的值等于最接近于输入纹理坐标的，两个texel（对于一维纹理）或四个texel（对于二维纹理）的线性插值的结果；

cudaFilterModeLinear 需要返回值是浮点类型；

④ **addressMode** 指定寻址模式：如果控制超出范围的纹理坐标；**addressMode** 是一个拥有两个元素的数组，分别指定第一个纹理坐标的寻址模式和第二个纹理坐标的寻址模式；寻址模式等于**cudaAddressModeClamp** 或**cudaAddressModeWrap**； 如果是**cudaAddressModeClamp**，超出范围的纹理坐标被钳制到合法的范围；如果是**cudaAddressModeWrap**，超出范围的纹理坐标被覆盖到合法的范围；

cudaAddressModeWrap 仅支持normalized 的纹理坐标；

④ **channelDesc** 定义了，当拾取纹理时返回值的格式；参见下面代码：

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

其中，**x,y,z,**和**w** 是返回值每个部分的位数，**f** 是：

- **cudaChannelFormatKindSigned** 如果这些部分是有符号的整型，
- **cudaChannelFormatKindUnsigned** 如果它们是无符号的整型，
- **cudaChannelFormatKindFloat** 如果它们是浮点数。

normalized，**addressMode**，和**filterMode** 可以在主机代码中直接修改。它们只应用在绑定到CUDA 数组的texture reference。

在一个kernel 通过texture reference 读取纹理内存之前，texture reference 必须绑定到一个使用**cudaBindTexture()**或**cudaBindTextureToArray()**的纹理中。

下面的代码演示，绑定一个texture reference 到通过**devPtr** 指向的线性内存中：

④ 使用低级API：

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

④ 使用高级API：

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

下面的代码演示，绑定一个texture reference 到CUDA 数组cuArray 中：

④ 使用低级API：

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

④ 使用高级API：

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

绑定一个纹理到texture reference 的格式必须匹配声明texture reference 时的参数；否则，纹理拾取的结果将是未定义的。

cudaUnbindTexture()用来卸载texture reference 的绑定。

4.5.2.7 OpenGL Interoperability

附录D.8 的函数用来控制OpenGL 的互用性。

一个缓冲对象在被映射之前必须注册到CUDA。使用**cudaGLRegisterBufferObject()**：

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

注册以后，缓冲对象可以通过kenrel 使用设备内存读取或写入，内存设备的地址通过**cudaGLMapBufferObject()**返回：

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

卸载映射和注册通过，**cudaGLUnmapBufferObject()**和**cudaGLUnregisterBufferObject()**。

4.5.2.8 Direct3D Interoperability

附录D.9 的函数用来控制Direct3D 的互用性。

Direct3D 的互用性必须通过**cudaD3D9Begin()**来初始化，**cudaD3D9End()**来终止。

一个顶点对象在被映射之前必须注册到CUDA。使用**cudaD3D9RegisterVertexBuffer()**：

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

注册以后，缓冲对象可以通过kenrel 使用设备内存读取或写入，内存设备的地址通过**cudaD3D9MapVertexBuffer()**返回：

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

卸载映射和注册通过，**cudaD3D9UnmapVertexBuffer()**和**cudaD3D9UnregisterVertexBuffer()**。

4.5.2.9 使用设备仿真方式调试

编程环境不支持任何原生的调试运行在设备上的代码，但伴随而来一个针对调试目的的设备仿真模式。在这个方式(使用**-deviceemu** 选项)下编译应用程序时，设备代码被编译成针对运行在主机上的，允许开发人员使用主机原生的调试支持来调试应用程序。预处理器宏**__DEVICE_EMULATION__**在这个模式下被定义。对于当前应用程序的所有代码，包括任何被使用的库，必须被编译，无论对于设备仿真或设备执行。连接设备仿真或设备执行的代码所产生的运行时错误，可以通过**cudaErrorMixedDeviceExxcution** 返回。

当在设备仿真模式下运行应用程序时，编程模型通过runtime 被仿真。对于在线程块中的每条线程，runtime 在主机生成一条线程。开发人员需要确定：

④ 主机能够运行每个块线程的最大数量，加上一条主线程。

④ 足够的内存可用于运行所有线程，每条线程需要256 KB 堆栈。

通过设备仿真模式提供的许多特点使它成为一套非常有效的调试工具：

④ 通过使用主机原生的调试支持，开发人员可以使用调试器的所有特性，像设置断点和数据检查。

④ 由于设备代码被编译到主机上运行，这个代码可以把在设备不能运行的代码补充进来，像是对文件或者屏幕(**printf()**，等)的输入和输出操作。

④ 因为所有数据驻留在主机上，任何设备的或者主机特定的数据可以从设备或者主机代码读取；同样的，任何设备或主机的函数可以从设备或主机代码中调用。

④ 为防止同步的错误使用，runtime 监测死锁情况。

开发人员必须牢记设备仿真模式是模仿设备，不是模拟它。因此，设备仿真模式在发现算法错误上是非常有用的，但某些错误是很难发现

④ 当内存单元在同一时间被栅格之内的多条线程访问时，运行在设备仿真方式下时的结果与运行在设备上时的结果截然不同，因为在仿真模式下线程是顺序的执行。

④ 当解除一个指向主机上的全局内存或者指向设备上的主机内存的引用时，设备执行几乎肯定在一些未定义的方式上失败，反之，设备仿真可以产生正确的结果。

④ 在大多数情况下，在设备上执行和在主机上通过设备仿真模式执行，同一浮点计算将不会产生完全相同的结果。这在通常情况下是能预料到的，对于同一浮点计算，取得的不同结果是由不同的编译器选项形成的，更不用说不同的编译器，不同的指令组，或者不同的架构了。

特别是，一些主机平台在扩展精度的寄存器里存贮了单精确度浮点计算的中间结果，往往造成设备仿真模式下，精度上的很大不同。当出现这些情况时，开发人员可尝试以下方法，但任何一种方法都不能完全保证工作：

- 声明一些浮点变量因为不稳定而强制单一精确度存贮；
- 使用**gcc** 编译器的**-ffloat-storegcc** 选项；
- 使用Visual C++编译器的**/Op** 或**/fp** 选项；
- 对于Linux 使用**_FPU_GETCW()**和**_FPU_SETCW()**，对于Windows 使用**_controlfp()**

函数来强制一部分代码单一精度浮点计算

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

或

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

在一开始，存储控制字的当前值，并强制尾数以24 位存储

```
_FPU_SETCW(originalCW);
或
_controlfp(originalCW, 0xffffff);
```

在最后，恢复原是的控制字。

不同于计算设备(参见附录A)，主机平台通常也支持规格化的数字。这样可以导致在设备仿真和设备执行模式之间明显不同的结果，因为某些计算在一种情况下可能产生一个有限结果而在另外一个情况下可能产生一个无限结果。

4.5.3 驱动API

驱动程序API 是基于句柄的，命令式的API：多数对象通过不透明地句柄引用。

可用于CUDA 的对象在表4-1 加以概述。

表4-1。在CUDA 驱动API 上可用的对象

Object	Handle	Description
Device	CUdevice	CUDA-capable device
Context	N/A	Roughly equivalent to a CPU process
Module	CUmodule	Roughly equivalent to a dynamic library
Function	CUfunction	Kernel
Heap memory	CUdeviceptr	Pointer to device memory
CUDA Array	CUarray	Opaque container for 1D or 2D data on the device, readable via texture references
Texture reference	CUtexref	Object that describes how to interpret texture memory data

4.5.3.1 初始化

在其他的函数（附录E）被调用之前，需要使用**cuInit()**函数初始化。

4.5.3.2 设备管理

附录E.2 的函数用来管理当前系统中的设备。

cuDeviceGetCount()和**cuDeviceGet()**用来枚举这些设备，E.2 中的其他函数用来获得它们的属性：

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

4.5.3.3 Context管理

附录E.3 的函数用来创建，捆绑和分离CUDA context 。

一个CUDA context 是类似于一个CPU 处理。在计算API 之内执行的所有资源和行为被压缩在CUDA context

里，并且当context 被销毁时系统自动地清理这些资源。除了对象例如模块和纹理引用以外，

每个context 有它自己的独立的32bit 地址空间。因此，来自不同的CUDA context 的 **CUdeviceptr** 值引用不同的内存单元。

Context 在主机线程中有一个一一对应的机制。一个主机线程当前一次只可以有一个设备context。当一个context 被创建时**cuCtxCreate()**，它成为了当前调用的主机线程。

以一个context 操作的CUDA 函数(大多数函数不包括设备枚举或者context 管理) 将返回**CUDA_ERROR_INVALID_CONTEXT**，如果当前的线程不是一个合法的context。

为了促进运行在同一个context 中的第三方授权的代码之间的互用性，驱动API 提供一个由每个确定客户给定的context 的使用量计数器。例如，如果三个库被加载使用同一个CUDA context ，每个库必须调用**cuCtxAttach()**来增加使用量计数器，而且当库已经完成context 使用时，调用**cuCtxDetach()**减少使用量计数器。当使用量计数器为0 时context 就被销毁了。对大多数库来说，应用程序应该在加载或初始化库之前创建一个CUDA context; 那样，应用程序可以创建一个用于它自己的context，并且库仅简单的操作context 交给它的任务。

4.5.3.4 模块管理

附录E.4 的函数用来加载和卸载模块，并获得模块中的句柄，变量的指针或函数的定义。

模块是动态地可加载的包括设备代码和数据的压缩包，如同Windows 中的DLL， 它们通过 **nvcc** 输出。名称对于所有标志，包括函数，全局变量和纹理引用，在模块范围内提供，以便被独立第三方编写的模块在同一CUDA context 可以互用。

下面的代码演示，加载一个模块并为kernel 取得一个句柄：

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

4.5.3.5 执行控制

附录E.7 的函数用来管理在设备上一个kernel 的执行。**cuFuncSetBlockShape()**用来设置给定函数每个块中线程的数量，和线程ID 如何分配。**cuFuncSetSharedSize()**指定函数共享内存的大小。**cuParam*()** 函数集提供用于kernel 的参数，当下次kernel 启动时包含**culanuchGrid()**或**cuLanuch()**。

```
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(cuFuntion, gridWidth, gridHeight);
```

4.5.3.6 内存管理

附录E.8 的函数用来分配和释放设备内存，并从主机和设备内存之前传输数据。

线性内存通过**cuMemAlloc()**或**cuMemAllocPitch()**来分配，**cuMemFree()**来释放。

下面的代码演示，在线性内存中分配一个256 个浮点元素的数组：

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

分配2D 数组建议使用**cuMemMallocPitch()**，从而保证访问行地址，或拷贝2D 数组到设备内存的其它区域的最佳性能。返回的pitch 必须用来访问数组元素。下面的代码演示，分配一个宽x 高带有浮点数的2D 数组，和在设备代码中如何循环数组元素：

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

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

CUDA 数组通过**cuArrayCreate()**分配，通过**cuArrayDestroy()**释放。

下面的代码演示，分配一个宽x 高带有一个32-bit 浮点数的CUDA 数组：

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

附录E.8 列出了不同的函数用来拷贝内存，包括用**cuMemMalloc()**分配的线性内存，用

cuMemMallocPitch()分配的线性内存，CUDA 数组。

下面的代码演示，拷贝一个2D 数组到之前例子中分配的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);
```

下面的代码演示，拷贝一些主机内存数组到设备内存：

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

4.5.3.7 流管理

附录E.5 的函数用来创建和销毁流，并且判定一个流中的所有操作是否完成。

下面的代码演示，创建两个流：

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

下面的代码演示，每一个流被依次定义执行一次：从主机到设备的内存拷贝，kernel 启动，从设备到主机的内存拷贝。

```
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]);
cuCtxSynchronize();
```

每个流拷贝部分输入数组**hostPtr** 到设备内存中的输入数组**inputDevPtr**，通过调用**cuFunction** 在设备上处理**inputDevPtr**， 并拷贝结果**outputDevPtr** 到**hostPtr** 的同一个部分。使用两个流处理**hostPtr** 允许内存拷贝从一个流覆盖到另一个流。对于任何覆盖，**hostPtr** 必须指向page-locked 的主机内存：

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

cuCtxSynchronize()在最后被调用，以确保在继续处理之前全部的流已经完成。

4.5.3.8 事件管理

附录E.6 的函数用来创建，纪录和销毁事件，并可查询两个事件之间花费的时间。

下面的代码演示，建立两个事件：

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

这些事件可以用来计算之前代码的时间：

```
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 Texture Reference 管理

附录E.9 的函数用来管理texture reference 。

在一个kernel 通过texture reference 读取纹理内存之前，texture reference 必须绑定到一个使用**cuTexRefSetAddress()**或**cuTexRefSetArray()**的纹理中。

如果一个模块**cuModule** 中包含一些texture reference **texRef** 定义为

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

下面的代码演示，获得**texRef** 的句柄：

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

下面的代码演示，绑定一个texture reference 到通过**devPtr** 指向的线性内存中：

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

下面的代码演示，绑定一个texture reference 到CUDA 数组**cuArray** 中：

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

附录E.9 列出了各种函数用于，设定texture reference 的寻址模式，过滤模式，格式，和其他标志。绑定一个纹理到texture reference 的格式必须匹配声明texture reference 时的参数；否则，纹理拾取的结果将是未定义的。

4.5.3.10 OpenGL Interoperability

附录E.10 的函数用来控制OpenGL 的互用性。

OpenGL 的互用性必须通过**cuGLInit()**来初始化。

一个缓冲对象在被映射之前必须注册到CUDA。使用**cuGLRegisterBufferObject()**：

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

注册以后，缓冲对象可以通过kenrel 使用设备内存读取或写入，内存设备的地址通过**cuGLMapBufferObject()**返回：

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

卸载映射和注册通过，**cuGLUnmapBufferObject()**和**cuGLUnregisterBufferObject()**。

4.5.3.11 Direct3D Interoperability

附录E.11 的函数用来控制Direct3D 的互用性。

Direct3D 的互用性必须通过**cuD3D9Begin()**来初始化，**cuD3D9End()**来终止。

一个顶点对象在被映射之前必须注册到CUDA。使用**cuD3D9RegisterVertexBuffer()**：

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

注册以后，缓冲对象可以通过kenrel 使用设备内存读取或写入，内存设备的地址通过**cuD3D9MapVertexBuffer()**返回：

```
LPDIRECT3DVERTEXBUFFER9 vertexBuffer;
CUdeviceptr devPtr;
Int size;
cuD3D9MapVertexBuffer(&devPtr, &size, vertexBuffer);
```

卸载映射和注册通过，**cuD3D9UnmapVertexBuffer()**和**cuD3D9UnregisterVertexBuffer()**。

Chapter 5 性能指导

5.1 指令性能

处理一个warp 线程指令，一个多处理器必须：

- 读取每条warp 线程的指令运算域，
- 执行指令，
- 为每条warp 线程写出结果。

因此，有效的指令吞吐量取决于内存延迟时间和带宽：

- 将低吞吐量的指令使用减到最小(参见第5.1.1 部分)，
- 最大化的使用每个内存类别可用的内存带宽(参见第5.1.2 部分)，
- 允许线程调度程序尽量把内存处理与数学计算交叠起来，因此要求：
  - 线程执行的程序是高运算密度的，即每个内存操作有大量的算术运算;
  - 很多线程可以并行地运行，详述的在第5.2 部分。

5.1.1 指令吞吐量

5.1.1.1 算术指令

发布一个warp 指令，一个多处理器需要：

- 4 个时钟周期执行浮点相加，浮点相乘，浮点乘加，整数相加，逐字节操作，比较，最小，最大，类型变换指令；
- 16 个时钟周期执行倒数，倒数平方根，`__log(x)`(参见表B-2)。

32 位整数乘法需要16 个时钟周期，但`__mul24` 和`__umul24` (参见附录B)提供在4 个时钟周期内做有符号的和没有符号的24 位整数乘法。对于将来的架构，`__[u]mul24` 将会比32 位整数乘法慢，因此我们建议保留32 位整数乘法以便以后使用。

整数除法和模数操作是非常消耗资源的，应该避免或者只要可能就用逐位替换操作：如果 n 是 2 的指数，`(i/n)` 等于`(i>>log2(n))`和`(i%n)`等于`(i&(n-1))`；如果 `n` 是字母的，编译器将执行这些转换。其他函数占有更多时钟周期，他们作为几个指令的组合加以执行。

浮点平方根是作为一个倒数平方根跟随一个倒数被执行的，取代一个倒数平方根跟随一个乘法。因此一个warp 操作需要32 个时钟周期。

浮点除法需要36 个时钟周期，但**__fdividef(x，y)** 可以在20 个时钟周期完成(参见附录B)。**__sin(x)**，**__cos(x)**，**__exp(x)**需要32 时钟周期。

有时，编译器必须插入转换指令，引进额外的执行周期。这个情况是为：

④ 对于**char** 或者**short** 函数操作，操作数通常需要被转换成**int**，

④ 双精度浮点常量(没有任何类型后缀的情况下)是作为输入到单精确度浮点计算使用的。

④ 单精确度浮点变量是作为用于输入参数到在表B-1 里定义的双精度版本的数学函数使用的。

最后两个情况可以通过使用下面方法避免，：

④ 单精确度浮点常量，用**f** 后缀定义，例如**3.141592653589793f**， **1.0f**， **0.5f**，

④ 数学函数的单精确度版本，用**f** 后缀定义，例如**sinf()**， **logf()**， **expf()**。

对于单精度代码，我们特别推荐使用单精度数学函数。在为设备编译而没有原生双精度支持时，比如计算兼容性1.x 的产品，双精度数学函数在默认情况下被映射到他们等值的单精度。然而，在那些未来支持双精度的设备，这些函数将映射到双精度执行。

5.1.1.2 控制流指令

任何流控制指令(**if**，**switch**，**do**，**for**，**while**)可以通过同一warp 分流的线程促成对有效指令吞吐量极大冲击，那就是允许不同的执行路径。如果它发生了，不同的执行路径必须是序列的，并且为这个warp增加了执行指令总数。当所有的执行路径完成，线程聚集回同一个执行路径。

为了在流控制依赖线程 ID 的情况下得到最佳性能，控制条件应该被写为使分流的 warp 数量减到最小。这是可能的，因为 warp 分流的跨越块是确定的（如在第 3.2 部分提及的）。一个普通的例子是，当控制条件只依赖于(**threadIdx / WSIZE**)，这里 **WSIZE** 是 warp 的大小。在这种情况下，因为控制条件与 warp 完全结合而没有 warp 分流。

有时，编译器可能展开循环或者它可以通过改用分支断言优化出**if** 或者**switch** 语句，详述如下。在这些情况下， warp 不可能分流。开发人员同样可以通过**#pragma unroll** 指令控制循环的展开（参见4.2.5.2）。

当使用分支断言时没有一个依赖控制条件执行的指令得以省略。相反，它们之中的每一个指令与每线程条件代码或者断言联系在一起，根据控制条件建立了真或者假，虽然每一个指令得到执行的确定时间，仅带有真断言的指令实际上被执行。带有一个假断言的指令不写结果，并且同样不计算地址值也不读操作数。

只有当分支条件控制的指令数少于或等于某一确定的门限值时，编译器用断言指令替换一个分支指令：如果编译器确定这个条件可能生产很多分流的warp，这个门限值是7，否则它是4。

5.1.1.3 内存指令

内存指令包括所有读入或写出到共享或者全局内存的指令。一个多处理器需要4 个时钟周期发出一条warp内存指令。另外，当访问的全局内存时，内存延时时间需要400 个到600 个时钟周期。

作为例子，用以下样本代码演示运算符：

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

用4 个时钟周期从全局内存发布一个读操作，用4 个时钟周期发布一个写操作共享内存，而最重要的是从全局内存读取一个浮点要400 到600 个时钟周期。

在等待全局内存存取完成时，如果有足够的独立算术指令可以发布，很多这样的全局内存延时时间可以被线程调度程序隐藏起来。

5.1.1.4 同步指令

如果线程不必等待任何其他线程，**__syncthreads** 可以用4 个时钟周期发布warp。

5.1.2 内存带宽

每个内存空间的有效带宽极大的取决于内存存取模式，在以下分项详述。

由于设备内存比片上内存有更高的反应延时和更低的带宽，因此设备内存的存取应该减到最小。一个典型的编程模式是安排来自设备内存的数据进入共享内存；换句话说，进入一个块的每条线程：

④ 从设备内存加载数据到共享内存，

④ 与块的所有其他线程同步，以便每条线程可以安全地读取由不同线程写入的共享内存的存储单元，

④ 在共享内存中处理数据，

④ 如果需要再次同步确认共享内存已被结果更新，

④ 把结果写回到设备内存。

5.1.2.1 全局内存

全局内存空间没有被缓存，因此使用正确的存取模式来获得最大的内存带宽更为重要，尤其是如何存取昂贵的设备内存。

首先，设备有能力在一个单一指令下从全局内存读取32-bit，64-bit 或128-bit 字进入寄存器。分配如下：

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

编译一个单一加载指令，**type** 必须是**sizeof(type)**等于4， 8, 或者16 是这样的，而且变量的类型**type** 必须排列成字节 (即它们的地址等于**sizeof(type)**的倍数)。

队列要求4.3.1.1 部分的内置类型像**float2** 或**float4** 一样的自动完成的。

对于结构，大小和队列要求可以通过编译器强制使用队列指定的**__align__(8)**或**__align__(16)**，例如

```
struct __align(8)__ {
    float a;
    float b;
};
```

或

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

对于结构大于16 字节的，编译器生成几个加载指令。来保证它生成最低数量的指令，这样的结构应该用**__align__ (16)**定义，例如

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

被编译成为二个128-bit 加载指令而不是五个32-bit 加载指令。

其次，全局内存地址同时被每线程的一个half-warp 访问（执行读和写指令）时，应该排列好，以便内存的存取可以结合进入一个接近单一的，排列好的内存存取。它意味着在每一个half-warp 中，在half-warp 中的第**N** 个线程应该访问地址

```
    HalfWarpBaseAddress + N
```

这里，**HalfWarpBaseAddress** 是类型**type***，而**type** 应该符合之前讨论过的大小和队列的要求。

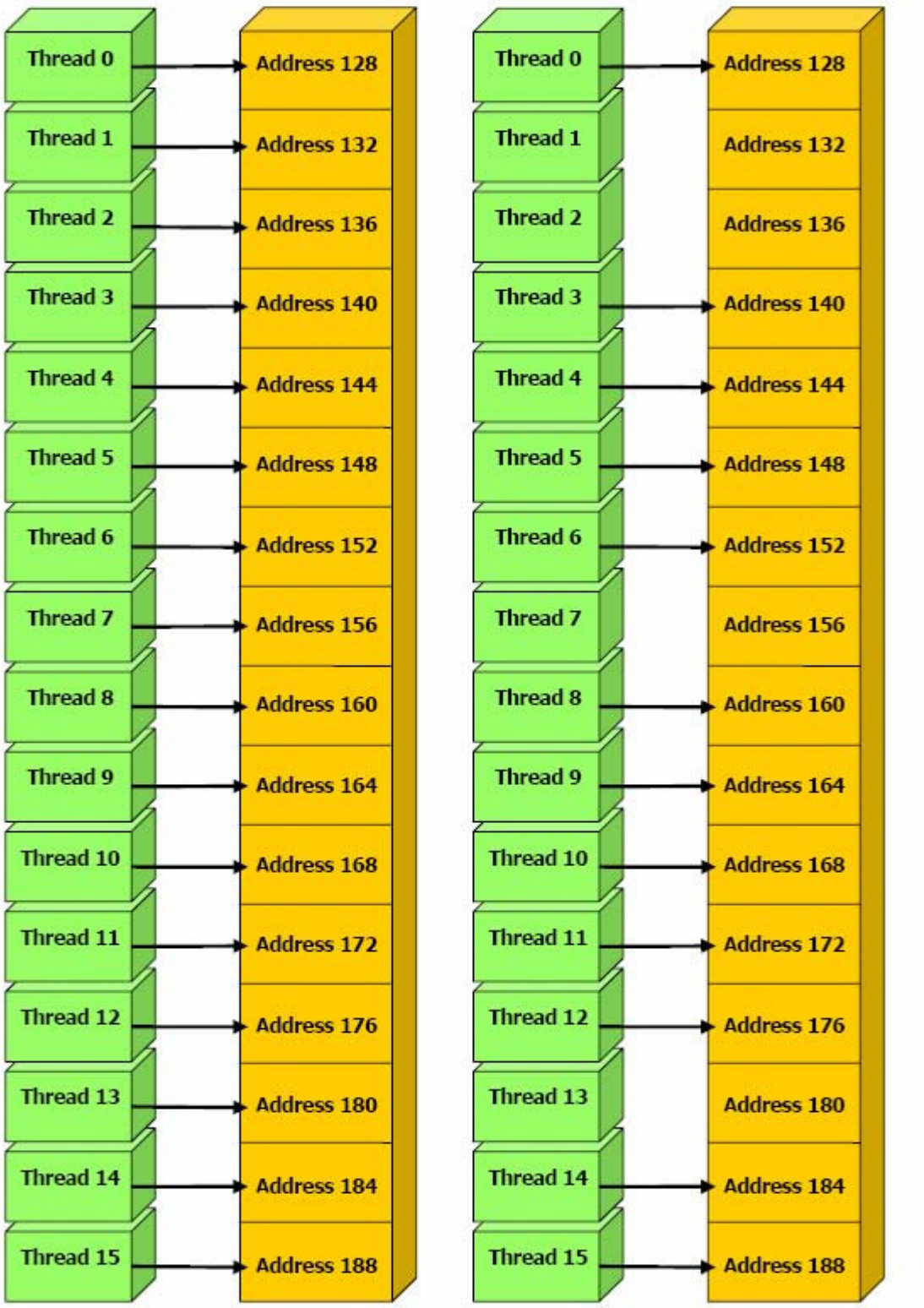
HalfWarpBaseAddress 应该排列成**16*sizeof(type)**字节，例如**16*sizeof(type)** 的倍数。任何一个驻留在全局内存**BaseAddress** 的变量的地址，或者一个来自D.5 或E.8 部分内存分配规则返回的地址，总是被排列成至少256 个字节，以此来满足内存队列的约束，**HalfWarpBaseAddress-BaseAddress** 应该是**16*sizeof(type)**的倍数。

注意，如果一个half-warp 满足了上面的所有需求，每线程的内存访问被联合了，即使half-warp 的一些线程实际上没有访问内存。

我们建议对于整个warp 满足这些需求，而不是分开的，半个半个的。因为未来的设备将默认为必要的要求。

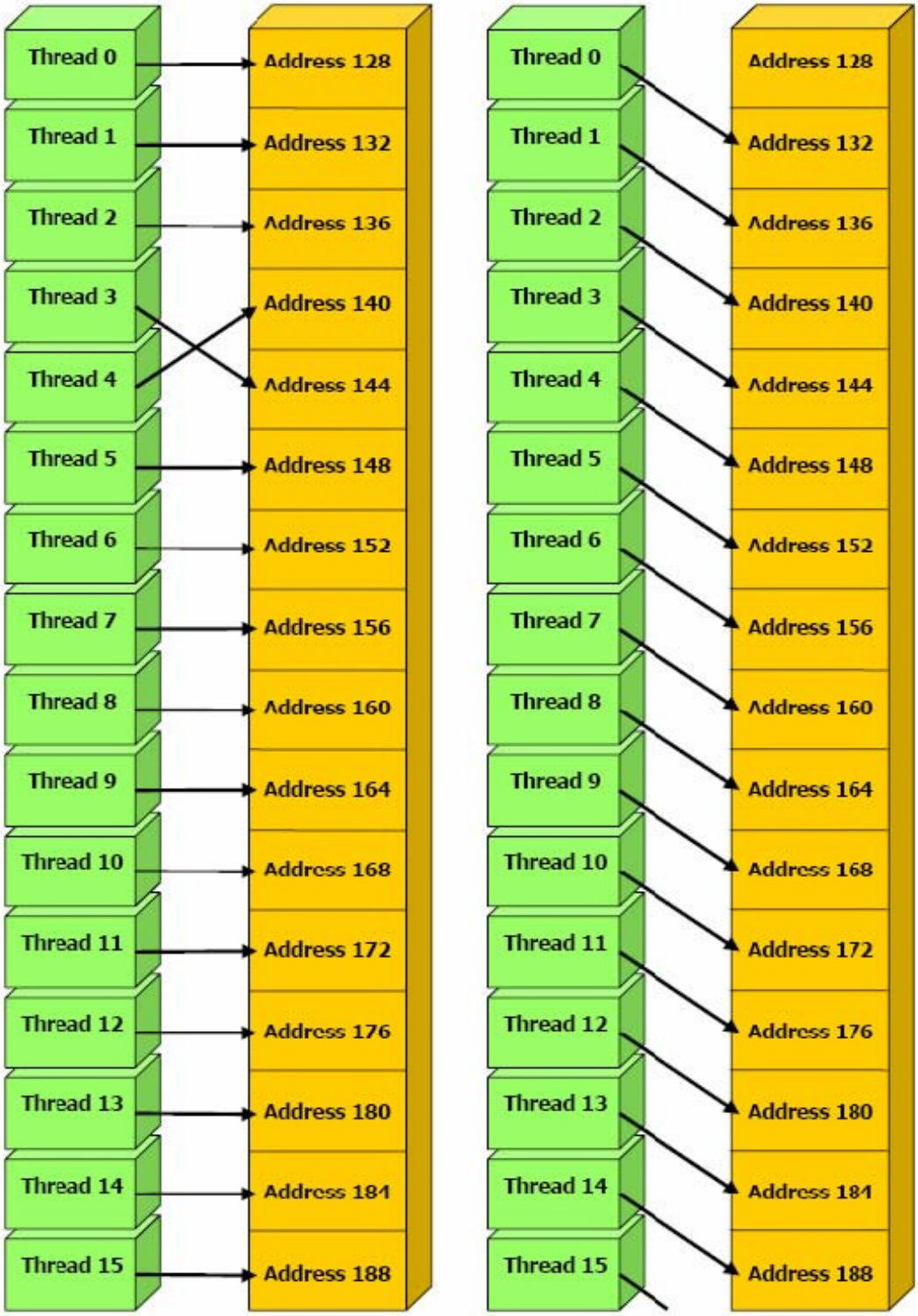
图5-1 展示了一些联合的内存访问的例子，图5-2 和图5-3 展示了一些未联合的内存访问的例子。

联合的64-bit 访问会比联合的32-bit 访问内存带宽低一点，联合的128-bit 访问会比联合的32-bit 访问内存带宽低很多。



Left: coalesced `float` memory access.
Right: coalesced `float` memory access (divergent warp).

图 5-1 联合的全局内存访问模式



Left: non-sequential `float` memory access.
Right: misaligned starting address.

图5-2 未联合的全局内存访问模式

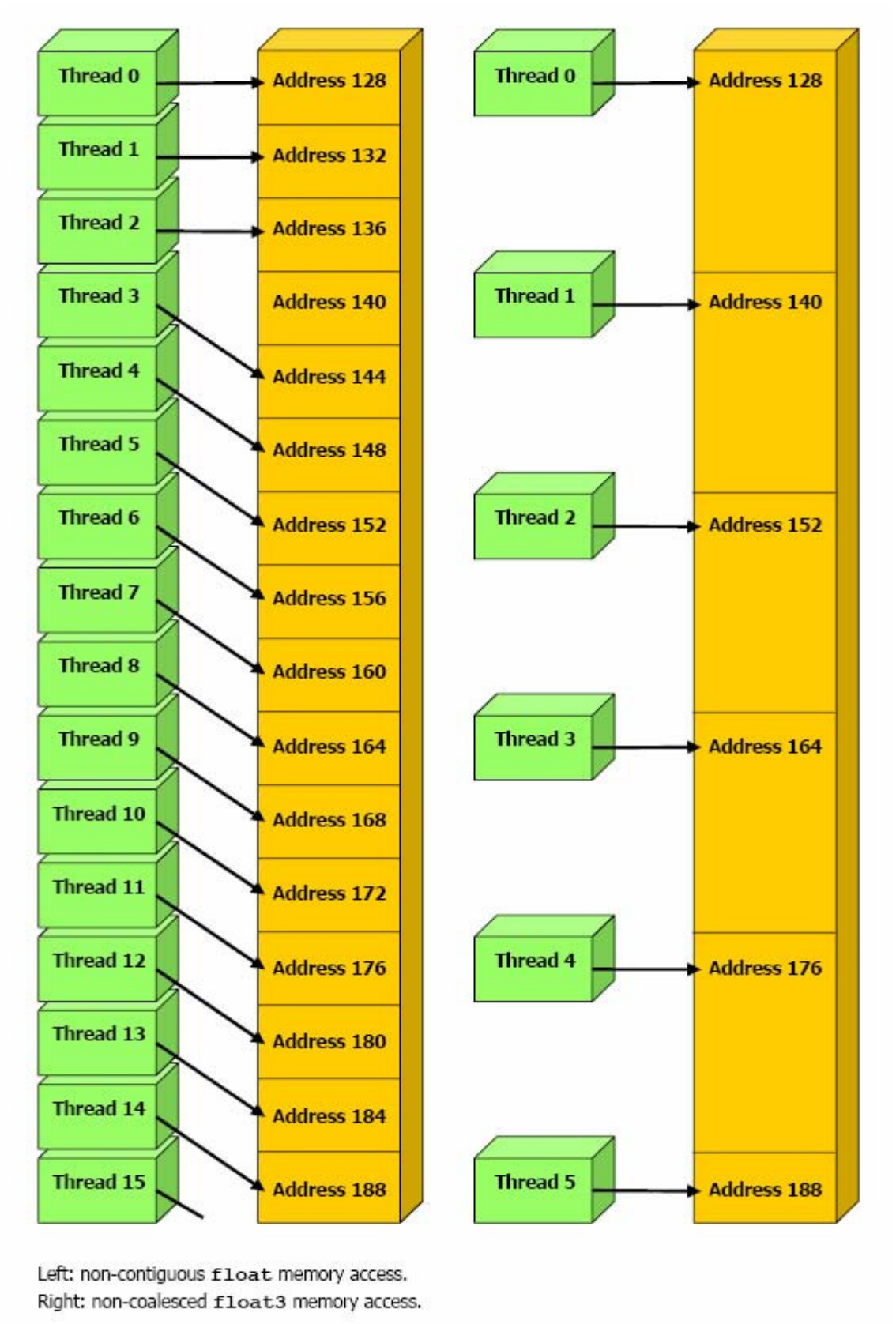


图 5-3 未联合的全局内存访问模式

一个公共的全局内存访问样式是，当每个带有线程ID **tid** 的线程访问位于一个数组的一个元素时，元素的地址位于类型**type*** 的**BaseAddress**，使用以下地址：

```
BaseAddress + tid
```

为了获得内存的联合，**type** 必须符合之前讨论过的大小和队列的要求。如果**type** 的结构大于16 字节，它应该被分成几个满足要求的结构，并且数据应该在内存中被划分成关于这些结构的几个数组，而不是一个类型**type***的单一数组。

另一个公共的全局内存访问样式是，当带有索引**(tx，ty)**的每条线程访问地址位于类型**type*** 的**BaseAddress** 和宽度**width** 的2D 数组的一个元素时使用以下地址：

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

在这样的情况下，获得half-warp 的所有块线程的内存结合，只有当：

④ 块线程的宽度是一半warp 大小的倍数;

④ **width** 是16 的倍数。

特别是，这意味着宽度不是16 的倍数的数组将被更高效地访问，如果它实际上分配是宽度被舍入到最接近16 的倍数而且它的列因此被填充了。**cudaMAllocPitch()**和**cuMemAllocPitch()**函数和在D.5 和E.8 部分描述的相关内存拷贝的函数，使开发人员能够编写非硬件独立的代码，来分配遵循这些约束的数组。

5.1.2.2 常量内存

常量内存空间被缓存了，因此，当缓存被错过时，一个常量内存的读操作消耗一个从设备内存的读操作，否则它的开销不过是常量缓存的一个读操作。

对于一个half-warp 的所有线程，只要所有线程读同一地址，常量缓存读操作和寄存器读操作一样快。所有线程读取不同地址时的开销量成线性。我们建议对于整个warp 中的所有线程读取同一地址，而不是只对每个half-warp 的所有线程，因为未来的设备将需要它做全速读取。

5.1.2.3 纹理内存

纹理内存空间被缓存了，因此，当缓存被错过时，一个纹理拾取操作消耗一个从设备内存的读操作，否则它的开销不过是纹理缓存的一个读操作。纹理缓存优化了2D 空间位置，同一条warp 读取纹理地址的线程靠拢在一起以便达到最佳性能。

通过纹理拾取读取设备内存相对于从全局或常驻内存读取设备内存有很多优势。

5.1.2.4 共享内存

由于它是在片上的，共享内存空间比局部和全局内存空间更快。实际上，对于一个warp 的所有线程，访问共享内存和访问寄存器一样快，只要在线程之间没有bank 冲突，详述如下。

为达到高内存带宽，共享内存空间被划分成大小相等的内存模块，称之为bank，它可以被同时访问。因此，所有内存读或写请求形成的n 个地址，它们集合在n 个不同的内存bank 中可以被同时访问，形成一个像单一模块带宽一样高的n 次有效带宽。

然而，如果内存请求的2 个地址集合在同一个内存bank 中，这里就存在bank 冲突而且访问必须被序列化。硬件将一个带有bank 冲突的内存请求分成许多无冲突的请求，有效带宽的减少等于被分割的内存请求数量。如果被分割的内存请求的数量是n，初始的内存请求应该有n 种bank 冲突。

为了获得最高性能，最重要的是了解内存地址是如何映射到内存bank 的，并有效安排内存请求，从而使bank 冲突最小化。

在共享内存空间情况下，bank 被组织成，比如连续的32-bit 字分配到连续的库，并且每个库有32 bits 带宽每个2 时钟周期。

对于计算兼容性 1.x 的设备，warp 的大小是 32，bank 的数量是 16 (参见第 5.1 部分)。所以，对于一个warp
的共享内存请求被分成：一个请求于warp 的前半部分，而另一个请求于同warp 的后半部分。

因此，在属于warp 前半部分的一个线程和属于同一个warp 的后半部的线程之间分不存在bank 的冲突。

一个典型情况，每个线程访问一个32-bit 字，从一个带有线程索引ID **tid** 和带有一些字长s 的数组：

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

在这种情况下，每当**s*n** 是bank **m** 数量的倍数或着相等，每当**n** 是**m/d** 的倍数，这里**d** 是**m** 和**s** 的最大公因子，线程**tid** 和**tid+n** 将访问同一个库。因此，只有当warp 大小的一半是小于或等于**m/d** 时，这里没有bank 冲突。对于计算兼容性1.x 的设备，只有当**d** 等于1时，换句话说，只有当**s** 是奇数时，因为**m** 是二的次方，这个转换没有bank 冲突。

图5-4 和图5-5 展示了一些没有冲突的内存访问，图5-6 展示了一些有bank 冲突的内存访问。

其他需要说明的情况是，当每条线程访问的一个元素小于或大于32 bits 时。例如，如果一个**char** 的数组以下列方式被访问，将存在bank 冲突：

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

因为**shared[0], shared[1], shared[2],** 和**shared[3]**属于同一个库。然而如果同一个数组用以下
方式访问，所有库将不存在冲突：

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

一个结构任务被编译成，内存请求数量和结构中成员数量一样多，例如：

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

不同的结果：
三个单独的内存读操作没有bank 冲突，如果**type** 被定义为

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

因为每名成员被一个长度为三个 32-bit 字访问；

二个单独的内存读操作带有bank 冲突，如果**type** 被定义成为，

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

因为每名成员被一个长度为二个32-bit 字访问。

二个单独的内存读操作带有bank 冲突，如果**type** 被定义成为，

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

因为每名成员被一个长度为五个字访问。

共享内存也有广播机制的特性，当响应一个内存读请求时，一个32-bit 字可以被读取并且同时广播到几个线程。当一个half-warp 的几条线程从同一个32-bit 字之内的一个地址读取时，有效地减少了bank 冲突的数量。更恰当地讲，一个内存请求被分为几个地址和几个步骤操作，每一步需要2 个时钟周期完成一个被划分成若干没有冲突的子寻址操作，直到整个请求完成，使用以下步骤：

④ 选择一个指向剩余地址的字作为广播字，

④ 在子集中包括：

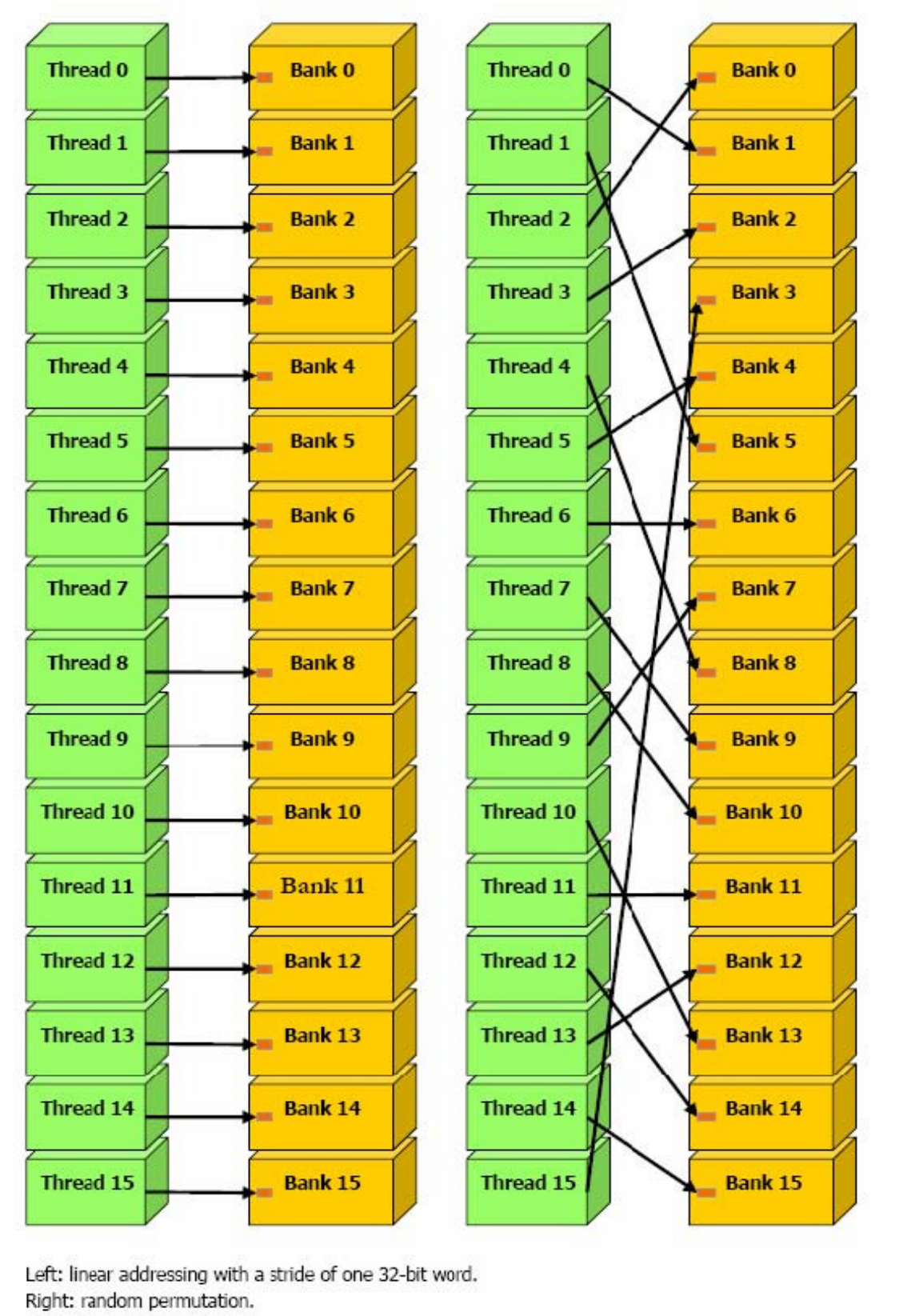
  ④ 所有地址在广播字之内，

  ④ 剩余的地址指出的每个bank 的一个地址。

哪个字被选择为广播字和哪个地址是被拾取对于每个bank 在每个周期是不确定的。

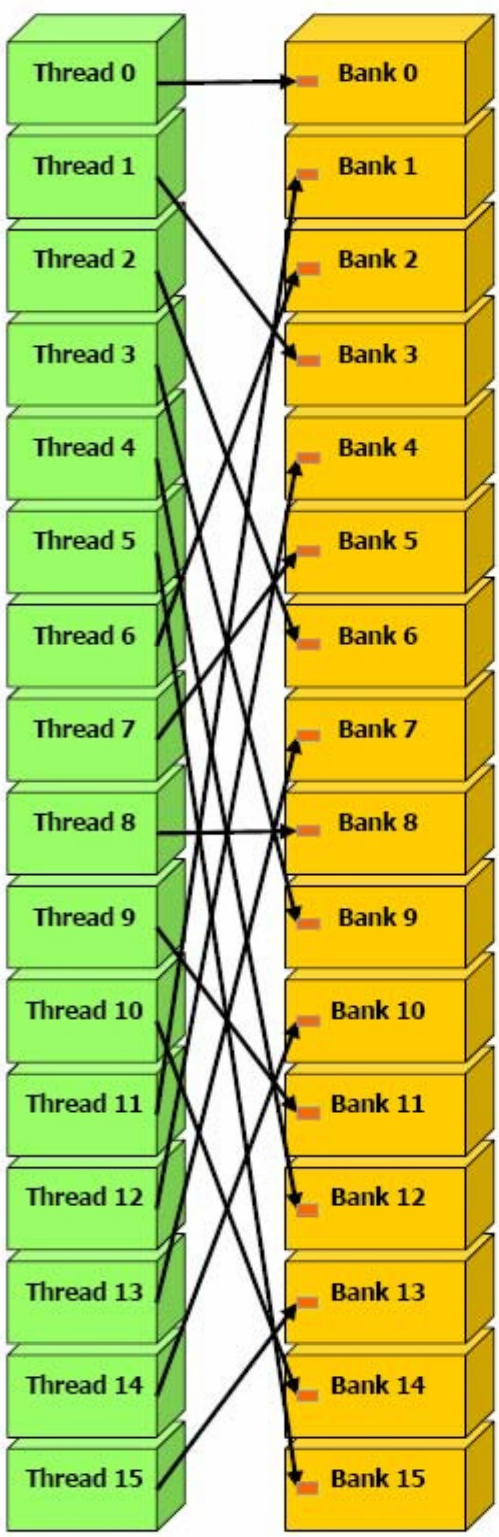
一个典型的无冲突的情况是，一个half-warp 的所有线程从同一个32-bit 字之内读取一个地址。

图5-7 展示一些带有广播机制的内存读取访问的例子



Left: linear addressing with a stride of one 32-bit word.
Right: random permutation.

图5-4 没有bank 冲突的共享内存访问模式



Linear addressing with a stride of three 32-bit words.

图 5-5 没有 bank 冲突的共享内存访问模式

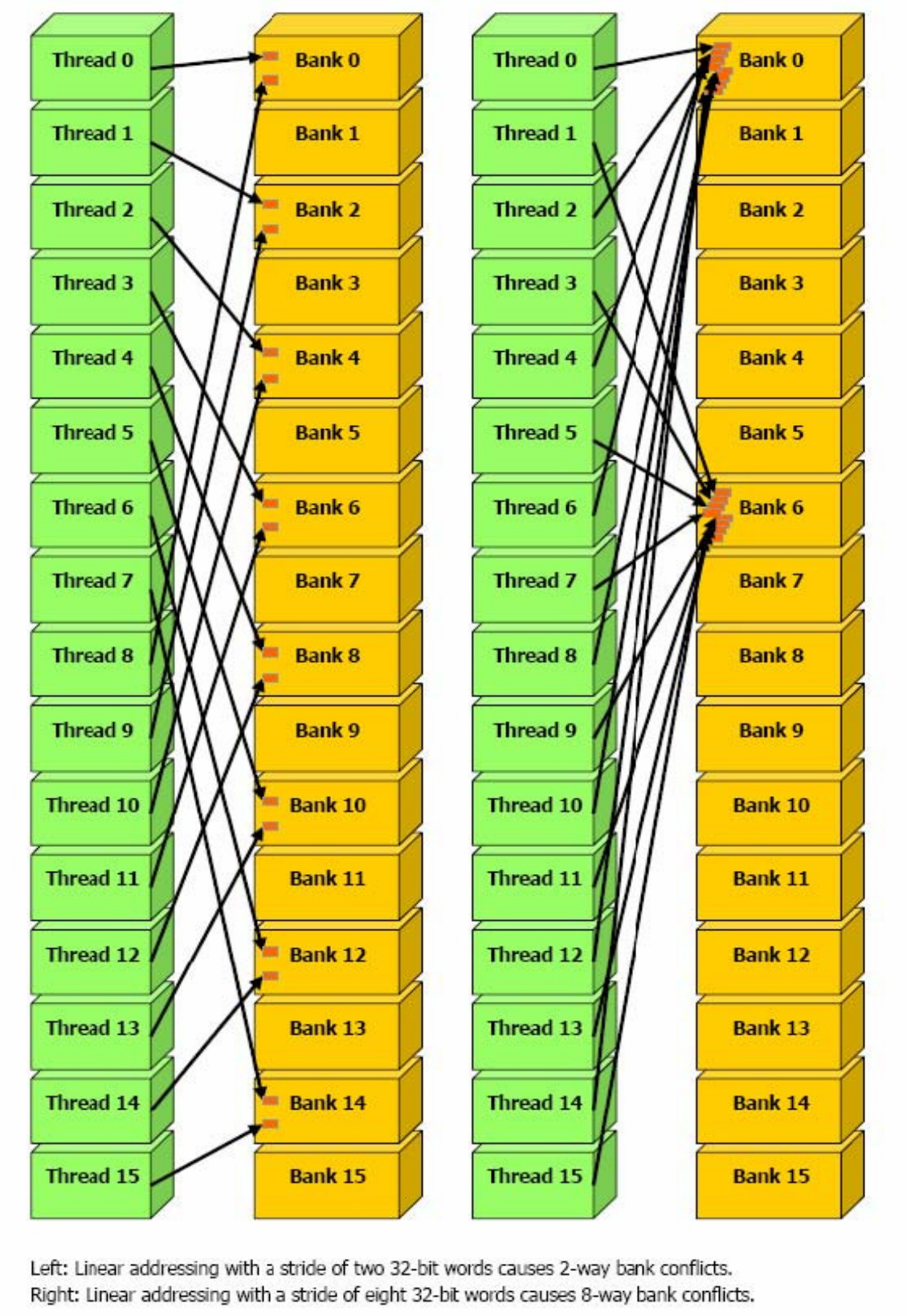
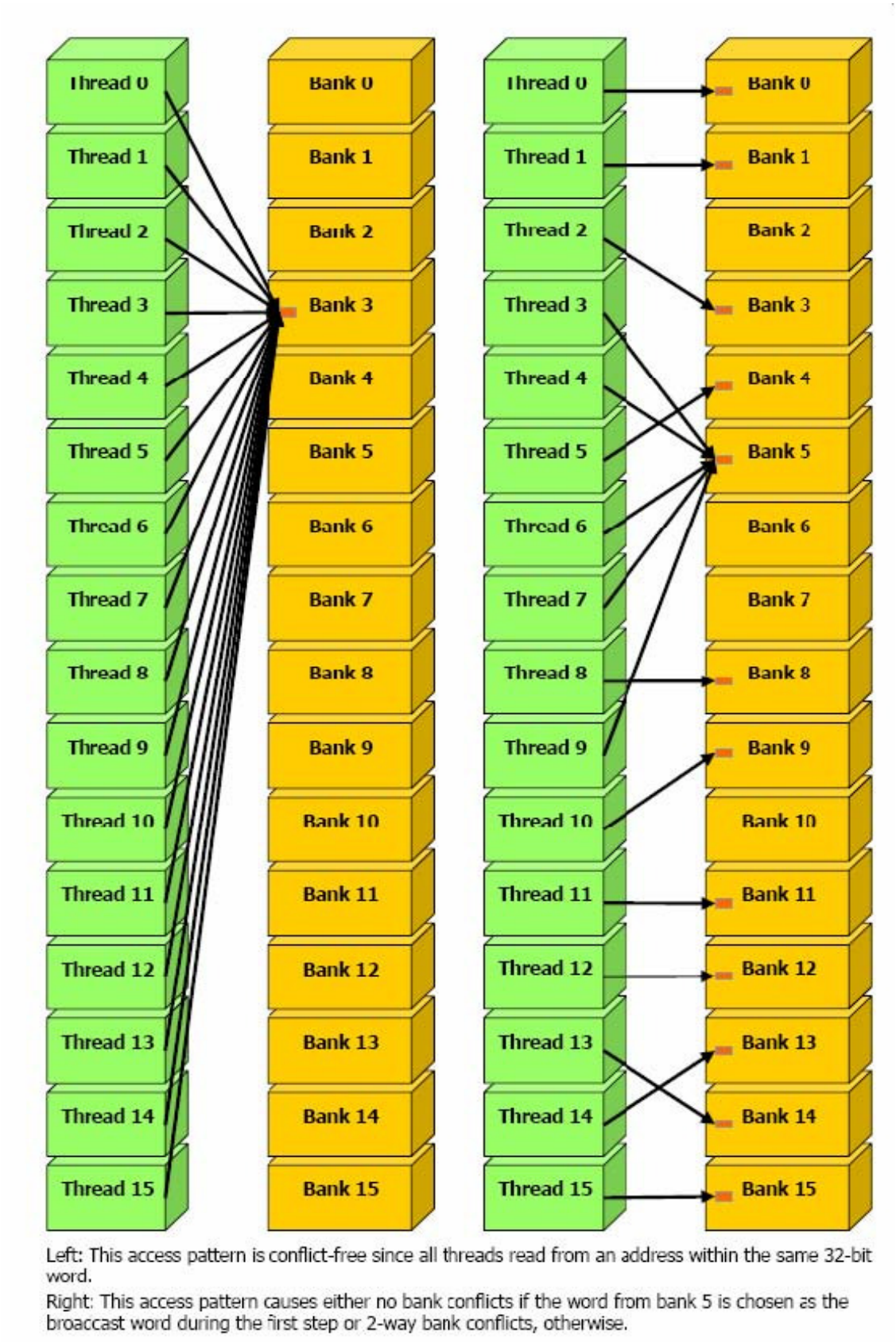


图 5-6 带有 bank 冲突的共享内存访问模式



Left: This access pattern is conflict-free since all threads read from an address within the same 32-bit word.

Right: This access pattern causes either no bank conflicts if the word from bank 5 is chosen as the broaccast word during the first step or 2-way bank conflicts, otherwise.

图5-7 带有广播机制的共享内存访问模式

5.1.2.5 寄存器

通常，每个指令访问一个寄存器是零附加时钟周期的，但延迟也许发生在寄存器read-after-write 和寄存器内存bank 冲突上。

被read-after-write 引入的延迟可以被忽略，一旦每台多处理器存在至少192 条并发线程。

编译器和线程的调度程序会尽可能的优化指令时间，避免寄存器内存bank 冲突。当每个块线程数是64 的倍数是达到最佳效果。另外，应用程序不直接控制这些bank 冲突。特别是，不需要把数据包装成**`float4`**或者**`int4`** 类型。

5.2 每个块的线程数量

确定每个栅格的线程总数，每个块的线程数量，或等效的块数量，从而最大化利用可用计算资源。这意味着，在设备上的多处理器数量至少应该和块的数量一样多。

此外，如果每个多处理器只执行一个块，在线程同步时，或设备内存读取时，将迫使多处理器闲置。因此在一个多处理器在建议同时运行两个或两个以上的块。并且，在设备上不仅要有至少两倍的块数量那么多的处理器，而且每个块分配的共享内存的总量应该最多是用于每个多处理器共享内存总量的一半。

如果块的数量足够多，每个块线程的数量应该选择成warp 大小的倍数以避免浪费计算资源，最好是64 的倍数，参见部分5.1.2.5。为每个块分配更多的线程对于时间分割更有效，但是为每个块分配越多的线程，每条线程可用的寄存器越少。如果kernel 编译的寄存器比执行配置允许的更多，这样也许可以防止kernel 随后的调用。寄存器被kernel 编译的数量可以通过**`--ptxas-options=-v`** 报告。

对于计算兼容性1.x 的设备，每个线程允许的寄存器数量等于

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

其中R 是附录A 中每个多处理器中寄存器的总数，B 是每个多处理器中活动的块数，T 是每个块中的线程数，$ceil(T,32)$ 是T 舍入最近的32 的倍数。

每个块最少应该有64 条线程并且每个多处理器应该有成倍的并发的块。每个块192 条或256 条线程是比较理想的，通常也有足够的寄存器进行编译。

如果要将代码用在未来的设备中，每个栅格的块的数量至少应该是100 个; 1000 个块将用在未来几代。

每个多处理器中活动的warp 数量与最大允许活动的warp 数量的比值被称作多处理器*occupancy*。为了达到最大的occupancy，编译器试图减小寄存器的使用量，同时开发人员也需要谨慎的使用执行配置。CUDA的软件开发工具包提供一个表格，辅助开发人员基于共享内存和寄存器的需求来选择线程块的大小。

5.3 主机与设备的数据传输

在设备到设备之间的内存带宽要比在设备到主机内存之间的带宽高很多。所以，应该力争使主机和设备之间数据传输减到最小。例如，移动更多的代码从主机到设备，即使这将导致更低的并行计算性。中间数据结构可以创建在设备中，并在设备中执行，最后在设备中销毁。

同样，批处理许多小的传输成为一个大传输永远比分开地逐个传输好很多。

最后，使用page-locked 内存将使主机到设备的内存带宽达到最大。

5.4 Texture Fetch 对比全局或常驻内存读取

通过Texture fetch 的内存读取相比从全局或常驻内存读取有几个优势：

④ 它们是被缓存的，如果它们在texture fetch 中将提供更高的带宽；

④ 它们不会像全局或常驻内存读取时受内存访问模式的约束（参见部分5.1.2.1 和5.1.2.2），从而获得更高的性能；

④ 寻址计算时的延迟更低，从而提高随机访问数据时的性能；

④ 在一个操作中，包装的数据可以通过广播到不同的变量中；

④ 8-bit 和16-bit 的整型输入数据可以被转换成在范围[0.0, 1.0]或[-1.0, 1.0]的浮点数（参见部分4.3.4.1）。

如果纹理是一个CUDA 数组（参见4.3.4.2），硬件提供其它有用的能力对于不同的应用程序，特别是图像处理：

Feature	Useful for...	Caveat
Filtering	Fast, low-precision interpolation between texels	Only valid if the texture reference returns floating-point data
Normalized texture coordinates	Resolution-independent coding	
Addressing modes	Automatic handling of boundary cases	Can only be used with normalized texture coordinates

但是，在同一个kernel 调用中，纹理缓存在全局内存写无法保持一致性，因此，任何纹理拾取到一个地址，将返回未定义的数据。换句话说，只有当内存地址被之前的kernel 调用或内存拷贝更新过，一个线程才能安全的读取纹理。但是，被同一个kernel 调用中的同一个或其他的线程更新过，是不安全的。当然，一个kernel 无法写入到一个CUDA 数组，因此，这只会发生在从线性内存拾取的情况下。

5.5 性能优化策略总结

性能优化包括三个基本策略：

④ 最大化并行执行；

④ 优化内存使用，以达到最大内存带宽；

④ 优化指令使用，以达到最大指令吞吐。

最大化并行执行通过构建有效的算法到达数据的最大并行。为了彼此共享数据，一些线程需要同步，从而打破了数据并行的机制。有两种情况：这些线程在同一个块中，在这种情况下，它们应该使用`__syncthreads()`， 并且通过同一kernel 调用中的共享内存共享数据；或者这些线程在不同的块中，在这种情况下，它们必须通过全局内存使用两个分开的kernel 请求来共享数据，一个用来从全局内存写，一个用来从全局内存读。

并行的算法同样需要尽可能有效的映射到硬件，这通过谨慎的执行配置达到（详细内容参见部分5.2）。

应用程序同样需要在高层最大化并行执行，这通过流在设备上的并发执行，最大化主机到设备的并发执行。

优化内存的使用通过尽量减低数据的传输。在部分5.3 中描述的，减小主机到设备的数据传输，由于设备到全局内存的带宽更低。在部分5.1.2 中描述的，通过最大化设备的共享内存，从而减小设备和全局内存之间的数据传输。有时，最佳的优化是通过简单的重新计算来避免任何的数据传输。

在部分5.1.2.1，5.1.2.2，5.1.2.3 和5.1.2.4 中描述的，依据不同的内存类型选择不同的访问方式。通过有效的组合内存的访问方式，从而优化内存的使用。对于全局内存的访问，优化全局内存的低带宽和上百时钟周期的延迟更重要。对于共享内存的访问，优化更针对于当它们有很高的bank 冲突时。

对于指令的优化，包括当最终结果不受影响时，如何权衡精度和速度。比如，使用本体（在表B-2 中列出）代替常规函数，或使用单精度代替双精度。控制流的指令需要特别注意，由于设备是SIMD 的特性。

Chapter 6 矩阵乘法的例子

6.1 概要

计算二个维度（*wA*，*hA*)和(*wB*，*wA*)的矩阵*A*和*B*的乘积*C*，分以下几步完成：

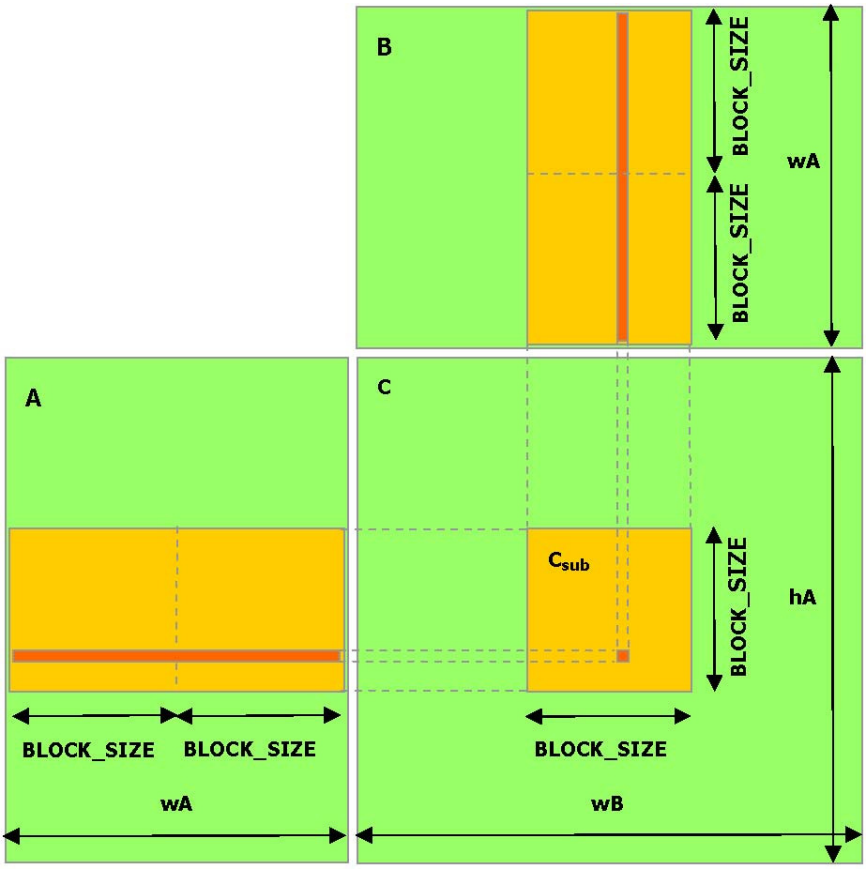
④ 每个线程块负责计算一个正方形的*C*的子矩阵C_{sub}

④ 在块之内的每条线程负责计算一个C_{sub}的元素。

C_{sub}维度*block_size* 等于16， 因此每个块线程的数量是warp 大小(第5.2 部分)的倍数而且低于每个块的昀大线程数(附录A)。

如图6-1 所示，C_{sub}等于二个矩形矩阵的乘积：维度(*wA*，*block_size*)的子矩阵*A*具有指向C_{sub}相同的行，维度(*block_size*，*wA*)的子矩阵*B*具有指向C_{sub}相同的列。为了适应设备的资源，需要把这二个矩形矩阵划分成维度*block_size* 的许多个正方形矩阵，并且C_{sub}是计算出来的这些方矩阵乘积的总和。每一个乘积是通过，加载二个对应的正方形矩阵从全局内存到共享内存，一条线程每次加载每个矩阵的一个元素，然后通过每个线程计算乘积的一个元素获得的。每条线程收集这些乘积的每个结果进入一个寄存器，最后把这些结果写到全局内存完成。

通过模块化计算这样的方式，我们利用了快速的共享内存而且节省了很多全局内存带宽，因为*A*和*B*从全局内存读取仅 (*wA / block_size*) 次。



每个线程块计算一个子矩阵C 的C_{sub}。每条在块之内线程计算C_{sub}的一个元素。

图6-1。矩阵乘法

6.2 源代码

```
    // 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
      dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
      dim3 dimGrid(wB / dimBlock.x, hA / dimBlock.y);

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
  // Multiply the two matrices together;
  // each thread computes one element of the block sub-matrix
  for (int k = 0; k < BLOCK_SIZE; ++k)
    Csub += As[ty][k] * Bs[k][tx];

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

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

6.3 源代码解释

源代码包含二个函数：

Mul()， 一个主机函数作为一个warpper 服务于**Muld()**；

Muld()， 在设备上执行矩阵乘法的一个kernel。

6.3.1 Mul()

Mul()用作输入：

④ 指向主机内存A 和B 元素的二个指针；

④ A 的高度和宽度，B 的宽度；

④ 指向主机内存应该写入C 的一个指针。

Mul()执行以下操作：

④ 使用**cudaMalloc()**来分配足够的全局内存存储*A*，*B*，和*C*；

④ 使用**cudaMemcpy()**从主机内存复制*A*和*B*到全局内存；

④ 调用**Muld()**来计算在设备上的*C*；

④ 使用**cudaMemcpy()**从全局内存复制*C*到主机内存；

④ 使用**cudaFree()**释放分配的全局内存*A*，*B*，和*C*。

6.3.2 Muld()

Muld()和**Mul()**有一样的输入，除了指针指向设备内存而不是主机内存。

对于每个块，**Muld()**迭代*A*和*B*的全部子矩阵来计算C_{sub}，每次迭代时：

④ 从全局内存加载一个*A*的子矩阵和一个*B*的子矩阵到共享内存；

④ 使用同步确认通过块之内的所有线程两个子矩阵得到完全地加载；

④ 计算二个子矩阵的乘积并且把它加到早先迭代获得的乘积里；

④ 再次同步，确认在开始下一个迭代之前二个子矩阵的乘积完成了。

一旦所有子矩阵得到处理，C_{sub}得到完全地计算并且**Muld()**把它写到全局内存中。

根据5.1.2.1 和5.1.2.4 部分，**Muld()**用来达到最大的内存性能。

如在5.1.2.1 部分建议的，假设***wA***和***wB***是16 的倍数，全局内存确定是聚集的，因为**a**，**b**，和**c** 是***BLOCK_SIZE***的倍数，***BLOCK_SIZE***等于16。

对于每个half-warp，共享内存也不存在bank 冲突，因为**ty** 和**k** 对于所有线程是相同的，并且**tx** 在0 和15 之间变化，因此，对于内存访问**As[ty][tx]**，**Bs[ty][tx]**和**Bs[k][tx]**，每条线程访问的是一个不同的bank，而对于内存访问**As[ty][k]**，每条线程访问的是同一个bank。

附录A 技术规格

这个附录介绍所有拥有计算兼容性（参见部分3.3）1.x 的设备技术规格。

原子函数只能应用于计算能力1.1 的设备（参见部分4.4.6）。

下表列出，所有支持CUDA 的设备，以及相对于该设备的多处理器数量和计算兼容性数值：

	Number of Multiprocessors	Compute Capability
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

设备的时钟频率和显存大小可以通过runtime 查询（参见部分4.5.2.2 和4.5.3.2）。

A.1 通用规格

④ 一个块最大线程数是512；

④ 一个线程块在x-，y-，和z-空间的最大大小分别是512，512 ，和64；

④ 一个线程块栅格的每个最大空间大小是65535；

④ Warp 的大小是32 个线程；

④ 每个多处理器的寄存器数量是8192；

④ 每个多处理允许的共享内存大小是16KB，被分为16 个bank；

④ 常驻内存大小是64KB；

④ 每个多处理器常驻内存可用的缓存是8KB；

④ 每个多处理器纹理内存可用的缓存是8KB；

④ 每个多处理器最大可用的块数量是8；

④ 每个多处理器最大可用的warp 数量是24；

④ 每个多处理器最大可用的线程数是768；

④ 对于绑定到一维CUDA 数组的texture reference，最大宽度是2^{13}

④ 对于绑定到二维CUDA 数组的texture reference，最大宽度是2^{15}，最大高度是2^{16}

④ 对于绑定到线性内存的texture reference ，最大宽度是2^{27}

④ Kernel 大小限制为2 百万个原生指令集；

④ 每个多处理器由8 个处理器构成，因此每个多处理器可以在四个时钟周期内处理一个32 个线程的warp。

A.2 浮点数标准

计算设备遵循单精度的二进制浮点数IEEE-754 标准，不同的是：

- 加法和乘法通常被合并成一个乘-加指令（FMAD）；
- 除法通过非标准兼容的倒数实现；
- 平方根通过非标准兼容的平方根倒数实现；
- 对于加法和乘法，只支持通过静态舍入模式实现的输入到最近偶数和输入到零；不支持直接舍入到正/负无穷；
- 没有动态配置的舍入模式；
- 不支持未归一化的数字；浮点算法和比较指令转换带有零优先级的未归一化操作数到浮点操作；
- Underflow 的结果被冲为零；
- 没有浮点异常的监测机制，浮点异常总是被mask 的；
- 不支持signaling NaN；
- 一个操作的结果包含一个或多个NaN，NaN 的位模式是0x7fffffff 。根据IEEE-754R 标准，一个输入到

fminf()，**fmin()**，**fmaxf()**，或**fmax()**的参数是NaN， 而不是其它，结果是一个非NaN 参数。当浮点数值超出IEEE-754 未定义的整型格式，浮点数转换成整型数。对于计算设备，仅钳制到支持的范围。这点不像x86 结构的习惯。

附录B 数学函数

B.1 中的函数可以被用作主机和设备函数，B.2 中的函数只能被用作设备函数。

B.1 公共**runtime** 组件表B-1 中列出了CUDA runtime 库中支持的标准数学函数。同时也报含每个函数的误差限，它们经过了密集的测试，但不保证所有的限都是绝对正确的。

加法和乘法为IEEE 兼容的，所以拥有最大误差0.5 ulp。它们通常被合并成一个乘-加指令（FMAD）。

我们建议求浮点数运算到整型时，使用**rintf()**，而不是**roundf()**。因为**roundf()**映射8 个指令序列，而**rintf()**只映射一个指令。**truncf()**，**ceilf()**，和**floorf()**同样也只映射一个指令。

CUDA runtime 库也支持整型的**min()**和**max()**，同样映射一个指令。

Table B-1. Mathematical Standard Library Functions with Maximum ULP Error

Function	Maximum ulp error
`x/y`	2 (full range)
`1/x`	1 (full range)
`1/sqrtf(x)` `rsqrtf(x)`	2 (full range)
`sqrtf(x)`	3 (full range)
`cbrtf(x)`	1 (full range)

Function	Maximum ulp error
`hypotf(x)`	3 (full range)
`expf(x)`	2 (full range)
`exp2f(x)`	2 (full range)
`exp10f(x)`	2 (full range)
`expm1f(x)`	1 (full range)
`logf(x)`	1 (full range)
`log2f(x)`	3 (full range)
`log10f(x)`	3 (full range)
`log1pf(x)`	2 (full range)
`sinf(x)`	2 (full range)
`cosf(x)`	2 (full range)
`tanf(x)`	4 (full range)
`sincosf(x,sptr,cptr)`	2 (full range)
`asinf(x)`	4 (full range)
`acosf(x)`	3 (full range)
`atanf(x)`	2 (full range)
`atan2f(y,x)`	3 (full range)
`sinhf(x)`	3 (full range)
`coshf(x)`	2 (full range)
`tanhf(x)`	2 (full range)
`asinhf(x)`	3 (full range)
`acoshf(x)`	4 (full range)
`atanhf(x)`	3 (full range)
`powf(x,y)`	16 (full range)
`erff(x)`	4 (full range)
`erfcf(x)`	8 (full range)
`lgammaf(x)`	6 (outside interval -10.001 ... -2.264; larger inside)
`tgammaf(x)`	11 (full range)
`fmaf(x,y,z)`	0 (full range)
`frexpf(x,exp)`	0 (full range)
`ldexpf(x,exp)`	0 (full range)
`scalbnf(x,n)`	0 (full range)
`scalblnf(x,l)`	0 (full range)
`logbf(x)`	0 (full range)
`ilogbf(x)`	0 (full range)
`fmodf(x,y)`	0 (full range)
`remainderf(x,y)`	0 (full range)
`remquof(x,y,iptr)`	0 (full range)
`modff(x,iptr)`	0 (full range)
`fdimf(x,y)`	0 (full range)
`truncf(x)`	0 (full range)
`roundf(x)`	0 (full range)

Function	Maximum ulp error
`rintf(x)`	0 (full range)
`nearbyintf(x)`	0 (full range)
`ceilf(x)`	0 (full range)
`floorf(x)`	0 (full range)
`lrintf(x)`	0 (full range)
`lroundf(x)`	0 (full range)
`llrintf(x)`	0 (full range)
`llroundf(x)`	0 (full range)
`signbit(x)`	N/A
`isinf(x)`	N/A
`isnan(x)`	N/A
`isfinite(x)`	N/A
`copysignf(x,y)`	N/A
`fminf(x,y)`	N/A
`fmaxf(x,y)`	N/A
`fabsf(x)`	N/A
`nanf(cptr)`	N/A
`nextafterf(x,y)`	N/A

B.2 设备runtime 组件

表B-2 列出了设备支持的固有函数。它们的误差限是GPU 特定的。虽然这些函数的精度更低，但是它们的速度比表B-1 中的一些函数快很多；它们拥有同样的前缀__（比如**__sinf(x)**）。

__fadd_rz(x,y) 使用舍入零的方式计算浮点参数**x** 和**y** 和。

__fmul_rz(x,y) 使用舍入零的方式计算浮点参数**x** 和**y** 乘积。

常规的浮点除法和**__fdividef(x,y)** 拥有同样的精度，但对于$2^{126} < y < 2^{128}$，**__fdividef(x,y)** 的结果为零，而常规的除法可以得到正确的结果。同样的，对于$2^{126} < y < 2^{128}$，如果**x** 是无穷大，**__fdividef(x,y)** 的结果是**NaN**（结果是无穷大乘于零），常规的除法返回无穷大。

__[u]mul24(x,y) 计算24 位最低有效位的整型参数**x** 和**y** 的乘积，并且给出32 位最低有效位的结果。**x** 和**y** 的8 位最高有效位被忽略。

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

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

__saturate(x) 如果**x** 小于0，返回0；如果**x** 大于1 返回1；如果**x** 在[0，1]之间，返回x。

__[u]sad(x,y,z)（Sum of Absolute Difference ）求整型参数**z** 与整型参数**x** 和**y** 差的绝对值的和。

__clz(x) 计算32 位整型参数x 的前导零。**__clzll(x)** 计算64 位整型参数x 的前导零。**__ffs(x)** 返回整型参数**x** 的第一个为1 位的位置。最低有效位的位置是1，如果**x** 是0，**__ffs()** 返回0。
这里和linux 函数ffs 是一样的。

__ffsll(x) 返回64 位整型参数**x** 的第一个为1 位的位置。最低有效位的位置是1，如果**x** 是0，**__ffsll()** 返回0。这里和Linux 函数**ffsll** 是一样的。

Table B-2. Intrinsic Functions Supported by the CUDA Runtime Library with Respective Error Bounds for Devices of Compute Capability 1.x

Function	Error bounds
`__fadd_rz(x,y)`	IEEE-compliant.
`__fmul_rz(x,y)`	IEEE-compliant.
`__fdividef(x,y)`	For `y` in $[2^{-126}, 2^{126}]$, the maximum ulp error is 2.
`__expf(x)`	The maximum ulp error is `2 + floor(abs(1.16 * x))`.
`__exp10f(x)`	The maximum ulp error is `2 + floor(abs(2.95 * x))`.
`__logf(x)`	For `x` in [0.5, 2], the maximum absolute error is $2^{-21.41}$, otherwise, the maximum ulp error is 3.
`__log2f(x)`	For `x` in [0.5, 2], the maximum absolute error is 2^{-22}, otherwise, the maximum ulp error is 2.
`__log10f(x)`	For `x` in [0.5, 2], the maximum absolute error is 2^{-24}, otherwise, the maximum ulp error is 3.
`__sinf(x)`	For `x` in $[-\pi, \pi]$, the maximum absolute error is $2^{-21.41}$, and larger otherwise.
`__cosf(x)`	For `x` in $[-\pi, \pi]$, the maximum absolute error is $2^{-21.19}$, and larger otherwise.
`__sincosf(x,sptr,cptr)`	Same as `sinf(x)` and `cosf(x)`.
`__tanf(x)`	Derived from its implementation as `__sinf(x) * (1 / __cosf(x))`.
`__powf(x, y)`	Derived from its implementation as `exp2f(y * __log2f(x))`.
`__mul24(x,y)` `__umul24(x,y)`	N/A
`__mulhi(x,y)` `__umulhi(x,y)`	N/A
`__int_as_float(x)`	N/A
`__float_as_int(x)`	N/A
`__saturate(x)`	N/A
`__sad(x,y,z)` `__usad(x,y,z)`	N/A
`__clz(x)`	N/A
`__ffs(x)`	N/A

附录C 原子函数

原子函数只能被设备函数使用。

C.1 算法函数

C.1.1 atomicAdd()

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

从全局内存中读取地址为**address** 的32-bit 字**old**，计算（**old + val**），将结果返回全局内存中的同一地址。这三个操作由一个原子操作执行。函数返回**old**。

C.1.2 atomicSub()

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

从全局内存中读取地址为**address** 的32-bit 字**old**，计算（**old - val**），将结果返回全局内存中的同一地址。这三个操作由一个原子操作执行。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);
```

从全局内存中读取地址为**address** 的32-bit 字**old**，存储**val** 返回全局内存中的同一地址。这二个操作由一个原子操作执行。函数返回**old**。

C.1.4 atomicMin()

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

从全局内存中读取地址为**address** 的32-bit 字**old**，计算**old** 和**val** 的最小值，将结果返回全局内存中的同一地址。这三个操作由一个原子操作执行。函数返回**old**。

C.1.5 atomicMax()

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

从全局内存中读取地址为**address** 的32-bit 字**old**，计算**old** 和**val** 的最大值，将结果返回全局内存中的同一地址。这三个操作由一个原子操作执行。函数返回**old**。

C.1.6 atomicInc()

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

从全局内存中读取地址为**address** 的32-bit 字**old**，计算**((old >= val) ? 0 : (old+1))**，将结果返回全局内存中的同一地址。这三个操作由一个原子操作执行。函数返回**old**。

C.1.7 atomicDec()

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

从全局内存中读取地址为**address**的32-bit 字**old**，计算**(((old ==0)| （old > val)) ? val : (old-1))**， 将结果返回全局内存中的同一地址。这三个操作由一个原子操作执行。函数返回**old**。

C.1.8 atomicCAS()

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

从全局内存中读取地址为**address** 的32-bit 字**old**，计算**(old == compare ? val : old)**，将结果返回全局内存中的同一地址。这三个操作由一个原子操作执行。函数返回**old**（比较和置换）。

C.2 位操作函数

C.2.1 atomicAnd()

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

unsigned int atomicAnd(unsigned int* address,

                       unsigned int val);
```

从全局内存中读取地址为**address** 的32-bit 字**old**，计算（**old & val**），将结果返回全局内存中的同一地址。这三个操作由一个原子操作执行。函数返回**old**。

C.2.2 atomicOr()

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

unsigned int atomicOr(unsigned int* address,

                      unsigned int val);
```

从全局内存中读取地址为**address** 的32-bit 字**old**，计算（**old | val**），将结果返回全局内存中的同一地址。这三个操作由一个原子操作执行。函数返回**old**。

C.2.3 atomicXor()

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

unsigned int atomicXor(unsigned int* address,

                       unsigned int val);
```

从全局内存中读取地址为**address** 的32-bit 字**old**，计算（**old ^ val**），将结果返回全局内存中的同一地址。这三个操作由一个原子操作执行。函数返回**old**。

附录D Runtime API Reference

Runtime API 有两个级别。

低级API（**cuda_runtime_api.h** ）是C 接口类型的，不需要nvcc 编译。

高级API（**cuda_runtime.h** ）是C++接口类型的，基于低级API 之上的，可直接使用C++代码，并被任何的

C++编译器编译。高级API 还有一些CUDA 特定的包，它们需要**nvcc** 编译。

D.1 设备管理

D.1.1 cudaGetDeviceCount()

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

返回计算兼容性大于等于1.0 的设备数量到指针***count**。如果没有相关设备，

cudaGetDeviceCount()返回1，device 0 仅支持设备仿真模式。

D.1.2 cudaSetDevice()

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

记录**dev** 作为设备在哪个活动的主机线程中执行设备代码。

D.1.3 cudaGetDevice()

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

返回设备在哪个活动的主机线程中执行设备代码到指针***dev**。

D.1.4 cudaGetDeviceProperties()

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

返回设备**dev** 的属性到指针***prop**。

cudaGetDeviceProp 的结构被定义为：

```
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：

④ **name** 是识别设备的字符串；

④ **totalGlobalMem** 是设备上全局内存的总byte 数；

④ **sharedMemPerBlock** 是每个块中共享内存的总byte 数；

④ **regsPerBlock** 是每个块中寄存器的总数；

④ **warpSize** 是warp 的大小

④ **memPitch** 是通过**cudaMallocPitch()**分配的内存空间中，允许内存拷贝函数的最大pitch；

④ **maxThreadsPerBlock** 是每个块中最大的线程数；

④ **maxThreadsDim[3]**是一个块中每个空间的最大大小；

④ **maxGridSize[3]**是一个栅格中每个空间的最大大小；

④ **totalConstMem** 是设备上常住内存的总byte 数；

④ **major** 和**minor** 是设备计算兼容性的主要次要版本；

④ **clockRate** 是时钟频率KHz；

④ **textureAlignment** 是对齐需求；纹理地址对齐到**textureAlignment** 字节的，不需要在texture fetch 中应用offset 。

D.1.5 cudaChooseDevice()

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

返回设备的哪些属性最匹配***prop** 到指针***dev**。

D.2 线程管理

D.2.1 cudaThreadSynchronize()

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

阻止直到设备上所有请求的任务执行完毕。**cudaThreadSynchronize()**返回一个错误，如果其中的一个任务失败。

D.2.2 cudaThreadExit()

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

清楚主机调用的线程中所有runtime 相关的资源。任何后来的API 将重新初始化runtime 。

D.3 流管理

D.3.1 cudaStreamCreate()

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

创建一个流。

D.3.2 cudaStreamQuery()

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

返回**cudasuccess**，如果所有流中的操作完成。返回**cudaErrorNotReady**，如果不是。

D.3.3 cudaStreamSyncronize()

```
cudaError_t cudaStreamSyncronize(cudaStream_t stream);
```

阻止直到设备上完成流中的所有操作。

D.3.4 cudaStreamDestroy()

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

销毁一个流。

D.4 事件管理

D.4.1 cudaEventCreate()

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

创建一个事件。

D.4.2 cudaEventRecord()

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

记录一个事件。如果stream 是非零的，当流中所有的操作完毕，事件被记录；否则，当CUDA context 中所有的操作完毕，事件被记录。由于这个操作是异步的，必须使用 **cudaEventQuery** 和/或**cudaEventSyncronize** 来决定何时事件被真的记录了。如果 **cudaEventRecord** 之前被调用了，并且事件还没有被记录，函数返回 **cudaErrorInvalidValue**。

D.4.3 cudaEventQuery()

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

返回**cudasuccess**，如果事件被真的记录了。返回**cudaErrorNotReady**，如果不是。如果**cudaEventQuery()**在这个事件中没有被调用，函数返回**cudaErrorInvalidValue**。

D.4.4 cudaEventSyncronize()

```
cudaError_t cudaEventSyncronize(cudaEvent_t event);
```

阻止直到事件被真的记录了。如果**cudaEventRecord()**在这个事件中没有被调用，函数返回**cudaErrorInvalidValue**。

D.4.5 cudaEventDestroy()

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

销毁一个事件。

D.4.6 cudaEventElapsedTime()

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

计算两个事件之间花费的时间（millisecond）。如果事件未被记录，函数返回**cudaErrorInvalidValue**。如果带有一个非零的stream 事件被记录，结果是未定义的。

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** 字节的线性内存，并返回分配内存的指针***devPtr**。分配的内存适合任何类型的变量。如果分配失败，**cudaMalloc()**返回**cudaErrorAlloction**。

D.5.2 cudaMallocPitch()

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

在设备上分配**widthInBytes*height** 字节的线性内存，并返回分配内存的指针***devPtr**。函数将确保在任何给出的行中对应的指针是连续的。pitch 返回的指针***pitch** 的是分配的宽度。Pitch 作为内存分配的一个分开的参数，用来计算2D 数组中的地址。一个给定行和列的类型**T** 的数组元素，地址等于：

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

对于2D 数组的分配，建议使用**cudaMallocPitch()**分配内存。由于pitch 对列限制受限于硬件，特别是当应用程序从设备内存的不同区域执行一个2D 的内存拷贝（无论是线性内存还是CUDA 数组）。

D.5.3 cudaFree()

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

释放被devPtr 指向的内存空间，devPtr 必须带有cudaMalloc()或cudaMallocPitch()的返回值。否则将返回一个错误，或者cudaFree(devPtr)已经在之前被使用过。如果devPtr 是0，不执行任何操作。如果cudaFree()调用失败，函数将返回cudaErrorInvalidDevicePointer。

D.5.4 cudaMallocArray()

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

根据**cudaChannelFormatDesc** 结构**desc** 分配一个CUDA 数组，并返回一个在***array** 的新CUDA 数组的句柄。

D.5.5 cudaFreeArray()

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

释放一个CUDA 数组**array**。如果**array** 是0，不执行任何操作。

D.5.6 cudaMallocHost()

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

分配一个**size** 字节可用于设备访问的page-locked 的主机内存。驱动程序追踪由这个函数分配的虚拟内存的范围，并自动加速函数的调用，例如**cudaMemcpy*()**。由于内存可以被设备直接访问，相比由例如函数**malloc()**分配的pagable 内存，拥有快很多的读写速度。但是，分配过多的page-locked 内存将减少系统可用物理内存的大小，从而降低系统整体的性能。

D.5.7 cudaFreeHost()

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

释放被**hostPtr** 指向的内存空间，**hostPtr** 必须带有**cudaMallocHost()**的返回值。

D.5.8 cudaMemSet()

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

使用常数值**value** 字节填充由**devPtr** 指向的内存空间内的前**count** 字节。

D.5.9 cudaMemSet2D()

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

设定由**dstPtr** 指向的一个矩阵中的值为**vaule**，**pitch** 是由**dstPtr** 指向的2D 数组中的内存宽度字节，2D 数组中每行的最后包含自动填充的数值（为保证效率的队列需求）。当pitch 是由**cudaMallocPitch()** 返回的值时，执行速度最快。

D.5.10 cudaMemcpy()

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

拷贝count 字节，从src 指向的内存区域到dst 指向的内存区域，kind 可以是cudaMemcpyHostToHost， cudaMemcpyHostToDevice，cudaMemcpyDeviceToHost，或cudaMemcpyDeviceToDevice 的拷贝方向。内存区域可能不会重叠。调用cudaMemcpy() 使用不匹配拷贝方向的指针src 和dst 将导致结果是未定义的。

cudaMemcpyAsync()是异步的，并且可以作为一个可选参数通过流使用。它只能应用于page-locked 的主机内存。如果使用一个指向pagable 的内存指针作为输入，函数将返回一个错误。

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);
```

拷贝一个矩阵，从src 指向的内存区域到dst 指向的内存区域，kind 可以是cudaMemcpyHostToHost， cudaMemcpyHostToDevice，cudaMemcpyDeviceToHost，或cudaMemcpyDeviceToDevice 的拷贝方向。dpitch 和spitch 是由dst 和src 指向的2D 数组中的内存宽度字节。内存区域可能不会重叠。调用cudaMemcpy2D()使用不匹配拷贝方向的指针src 和dst 将导致结果是未定义的。如果dpitch 和spitch 大于允许的最大值（参见附录D.1.4 中的memPitch），cudaMemcpy2D()将返回一个错误。

cudaMemcpy2DAsync()是异步的，并且可以作为一个可选参数通过流使用。它只能应用于page-locked 的主机内存。如果使用一个指向pagable 的内存指针作为输入，函数将返回一个错误。

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);
```

拷贝count 字节，从src 指向的内存区域到dstArray 指向的CUDA 数组，从数组的左上角（dstX, dstY） 开始，kind 可以是cudaMemcpyHostToHost， cudaMemcpyHostToDevice，cudaMemcpyDeviceToHost，或cudaMemcpyDeviceToDevice 的拷贝方向。

cudaMemcpyToArrayAsync()是异步的，并且可以作为一个可选参数通过流使用。它只能应用于page-locked 的主机内存。如果使用一个指向pagable 的内存指针作为输入，函数将返回一个错误。

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);
```

拷贝一个矩阵，从src 指向的内存区域到dstArray 指向的CUDA 数组，从数组的左上角（dstX, dstY） 开始，kind 可以是cudaMemcpyHostToHost， cudaMemcpyHostToDevice，cudaMemcpyDeviceToHost，或cudaMemcpyDeviceToDevice 的拷贝方向。spitch 是由src 指向的2D 数组中的内存宽度字节，2D 数组中每行的最后包含自动填充的数值（ 为保证效率的队列需求）。如果spitch 大于允许的最大值（参见附录D.1.4 中的memPitch），cudaMemcpy2DToArray()将返回一个错误。

cudaMemcpy2DToArrayAsync()是异步的，并且可以作为一个可选参数通过流使用。它只能应用于page-locked 的主机内存。如果使用一个指向pagable 的内存指针作为输入，函数将返回一个错误。

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);
```

拷贝count 字节，从srcArray 指向的CUDA 数组，从数组的左上角（srcX, srcY）开始，到dst 指向的内存区域，kind 可以是cudaMemcpyHostToHost， cudaMemcpyHostToDevice，cudaMemcpyDeviceToHost，或cudaMemcpyDeviceToDevice 的拷贝方向。

cudaMemcpy2DToArrayAsync()是异步的，并且可以作为一个可选参数通过流使用。它只能应用于page-locked 的主机内存。如果使用一个指向pagable 的内存指针作为输入，函数将返回一个错误。

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);
```

拷贝一个矩阵，从srcArray 指向的CUDA 数组，从数组的左上角（srcX, srcY）开始，到dst 指向的内存区域，kind 可以是cudaMemcpyHostToHost， cudaMemcpyHostToDevice，cudaMemcpyDeviceToHost，或cudaMemcpyDeviceToDevice 的拷贝方向。dpitch 是由dst 指向的2D 数组中的内存宽度字节，2D 数组中每行的最后包含自动填充的数值（ 为保证效率的队列需求）。如果dpitch 大于允许的最大值（参见附录D.1.4 中的memPitch），cudaMemcpy2DFromArray()将返回一个错误。

cudaMemcpy2DFromArrayAsync()是异步的，并且可以作为一个可选参数通过流使用。它只能应用于page-locked 的主机内存。如果使用一个指向pagable 的内存指针作为输入，函数将返回一个错误。

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);
```

拷贝**count** 字节，从**srcArray** 指向的CUDA 数组，从数组的左上角（**srcX**, **srcY**） 开始，到**dstArray** 指向的CUDA 数组，从数组的左上角（dstX, dstY），kind 可以是cudaMemcpyHostToHost， **cudaMemcpyHostToDevice**，**cudaMemcpyDeviceToHost**，或**cudaMemcpyDeviceToDevice** 的拷贝方向。

D.5.17 cudaMemcpy2DArrayToArray()

```
cudaError_t cudaMemcpyArrayToArray(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);
```

拷贝一个矩阵，从srcArray 指向的CUDA 数组，从数组的左上角（srcX, srcY）开始，到dstArray 指向的CUDA 数组，从数组的左上角（dstX, dstY），kind 可以是cudaMemcpyHostToHost， cudaMemcpyHostToDevice，cudaMemcpyDeviceToHost，或cudaMemcpyDeviceToDevice 的拷贝方向。

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);
```

拷贝**count** 字节，从**src** 指向的内存区域到由符号**symbol** 起始的**offset** 字节指向的内存区域。内存区域可能不会重叠。**symbol** 可以是在全局或常驻内存中的一个变量，也可以是在全局或常驻内存中的一个字符串。**kind** 可以是**cudaMemcpyHostToDevice**，或**cudaMemcpyDeviceToDevice** 的拷贝方向。

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);
```

拷贝**count** 字节，从由符号**symbol** 起始的**offset** 字节指向的内存区域到**dst** 指向的内存区域。内存区域可能不会重叠。**symbol** 可以是在全局或常驻内存中的一个变量，也可以是在全局或常驻内存中的一个字符串。**kind** 可以是**cudaMemcpyDeviceToHost**，或**cudaMemcpyDeviceToDevice** 的拷贝方向。

D.5.20 cudaGetSymbolAddress()

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

返回设备中符号**symbol** 的地址***devPtr**。**symbol** 可以是在全局或常驻内存中的一个变量，也可以是在全局或常驻内存中的一个字符串。如果**symbol** 没有被发现，或者**symbol** 没有在全局内存中被声明，***devPtr** 将无法改变，并返回一个错误。如果**cudaGetSymbolAddress()**调用失败，函数将返回**cudaErrorInvalidSymbol**。

D.5.21 cudaGetSymbolSize()

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

返回符号**symbol** 大小的地址***size**。**symbol** 可以是在全局或常驻内存中的一个变量，也可以是在全局或常驻内存中的命名一个变量的字符串。如果**symbol** 没有被发现，或者**symbol** 没有在全局内存中被声明，***size** 将无法改变，并返回一个错误。如果**cudaGetSymbolSize()**调用失败，函数将返回**cudaErrorInvalidSymbol**。

D.6 Texture Reference 管理

D.6.1 低级API

D.6.1.1 cudaCreateChannelDesc()

```
struct cudaChannelFormatDesc

cudaCreateChannelDesc(int x, int y, int z, int w,

                      enum cudaChannelFormatKind f);
```

返回一个通道描述符，其中，**x,y,z,**和**w** 是返回值每个部分的位数，**f** 是格式（参见4.5.2.6）。

D.6.1.2 cudaGetChannelDesc()

```
cudaError_t cudaGetChannelDesc(struct cudaChannelFormatDesc* desc,

                               const struct cudaArray* array);
```

返回CUDA 数组**Array** 的通道描述符的指针***desc**。

D.6.1.3 cudaGetTextureReference()

```
cudaError_t cudaGetTextureReference(struct textureReference** texRef,
                                    const char* symbol);
```

返回Texture reference 结构的指针***texRef**，Texture reference 是由**symbol** 定义的。

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);
```

绑定**devPtr** 指向的**size** 字节内存区域到texture reference **texRef**。**desc** 描述了从纹理拾取时内存如何被解释。任何之前被绑定到**texRef** 的内存将被解除绑定。

由于硬件对于基于纹理的地址有强制的队列需求，**cudaBindTexture()**返回的，***offset** 指向的offset 字节必须应用到纹理的fetch 中，从而读取所需的内存。这个offset 必须除以texel 的大小，并传递到kerenl 中，以便使用**tex1Dfetch()**函数。

D.6.1.5 cudaBindTextureToArray()

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

绑定CUDA 数组**array** 到texture reference **texRef** 。**desc** 描述了从纹理拾取时内存如何被解释。任何之前被绑定到**texRef** 的内存将被解除绑定。

D.6.1.6 cudaUnBindTexture()

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

解除绑定到texture reference **texRef** 的纹理。

D.6.1.7 cudaGetTextureAlignmentOffset()

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

返回***offset** 中的offset ， 当texture reference **texRef** 被绑定时。

D.6.2 高级API

D.6.2.1 cudaCreateChannelDesc()

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

返回一个格式匹配类型**T** 的通道描述符。**T** 可以是部分4.3.1.1 中的任何类型。

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);
```

绑定**devPtr** 指向的**size** 字节内存区域到texture reference **texRef**。**desc** 描述了从纹理拾取时内存如何被解释。offset 字节作为一个可选参数。任何之前被绑定到**texRef** 的内存将被解除绑定。

```
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);
```

绑定**devPtr** 指向的**size** 字节内存区域到texture reference **texRef**。通道描述符继承texture reference 的类型。offset 字节作为一个可选参数。任何之前被绑定到**texRef** 的内存将被解除绑定。

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);
```

绑定CUDA 数组**array** 到texture reference **texRef** 。**desc** 描述了从纹理拾取时内存如何被解释。任何之前被绑定到**texRef** 的CUDA 数组将被解除绑定。

```
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);
```

绑定CUDA 数组**array**到texture reference **texRef** 。通道描述符继承CUDA 数组。任何之前被绑定到**texRef** 的CUDA 数组将被解除绑定。

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);
```

解除绑定到texture reference **texRef** 的纹理。

D.7 执行控制

D.7.1 cudaConfigureCall()

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

指定在设备执行调用的栅格和块的维数，类似在部分4.2.3 中描述的执行控制。

cudaConfigureCall()是基于堆栈的。每个调用推动数据到一个执行堆栈顶端。这个数据包含栅格和块的维数，以及任何用于调用的参数。

D.7.2 cudaLaunch()

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

在设备上启动函数**entry**。**entry** 可以是执行在设备上的一个函数，也可以是命名一个函数的字符串。

entry 必须作为一个__global__函数声明。cudaLaunch()必须由一个到cudaConfigureCall()调用执行。

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);
```

从offset 0 开始，推动***arg** 指向的**count** 字节参数通过区域。参数存储在执行堆栈的顶端。

cudaSetupArgument()必须由一个到**cudaConfigureCall()**的调用执行。

D.8 OpenGL Interoperability

D.8.1 cudaGLRegisterBufferObject()

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

注册ID 为**bufferObj** 的缓冲对象。这个函数必须被调用后，CUDA 才能映射这个缓冲对象。一旦注册了，这个缓冲对象不能被任何OpenGL 命令使用，除非作为OpenGL drawing 命令的一个数据源。

D.8.2 cudaGLMapBufferObject()

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

映射ID 为**bufferObj** 的缓冲对象到CUDA 的地址空间中，并返回映射结果的基本指针到***devPtr**。

D.8.3 cudaGLUnMapBufferObject()

```
cudaError_t cudaGLUnMapBufferObject(GLuint bufferObj);
```

解除映射ID 为**bufferObj** 的缓冲对象。

D.8.4 cudaGLUnRegisterBufferObject()

```
cudaError_t cudaGLUnRegisterBufferObject(GLuint bufferObj);
```

解除注册ID 为**bufferObj** 的缓冲对象。

D.9 Direct3D Interoperability

D.9.1 cudaD3D9Begin()

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

为Direct3D 设备**device** 初始化Interoperability。这个函数必须被调用后，CUDA 才能从设备映射任何对象。应用程序接下来可以映射Direct3D 设备拥有的顶点缓冲，直到**cudaD3DEnd()**被调用。

D.9.2 cudaD3D9End()

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

去除Direct3D 设备**device** 的Interoperability。

D.9.3 cudaD3D9RegisterVertexBuffer()

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

注册顶点缓冲**VB**。

D.9.4 cudaD3D9MapVertexBuffer()

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

映射顶点缓冲**VB** 到当前的CUDA context 中，并返回映射结果的基本指针到***devPtr**。

D.9.5 cudaD3D9UnMapVertexBuffer()

```
cudaError_t cudaD3D9MapVertexBuffer(IDirect3DVertexBuffer9* VB);
```

解除映射顶点缓冲**VB**。

D.9.6 cudaD3D9UnRegisterVertexBuffer()

```
cudaError_t
cudaD3D9UnRegisterVertexBuffer(IDirect3DVertexBuffer9* VB);
```

解除注册顶点缓冲**VB**。

D.9.7 cudaD3D9GetDevice()

```
cudaError_t

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

返回相关于适配器名称的设备到指针*dev，适配器名称adapterName 由EnumDisplayDevices或IDirect3D9::GetAdapterIdentifier()获得。

D.10 错误处理

D.10.1 cudaGetLastError()

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

返回由同一主机线程中任何runtime 调用产生的最后的错误，并重置到**cudasuccess**。

D.10.2 cudaGetErrorString()

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

返回一个由错误代码产生的消息字符串。

附录E Driver API Reference

E.1 初始化

E.1.1 cuInit()

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

在其他的函数被调用之前，初始化驱动API 必须被调用。当前，**Flags** 参数必须是0。如果**cuInit()**没有被调用，其他驱动API 的函数将返回**CUDA_ERROR_NOT_INITIALIZED**。

E.2 设备管理

E.2.1 cuGetDeviceCount()

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

返回可用于执行的计算兼容性大于等于1.0 的设备数量到指针***count**。如果没有相关设备，**cuGetDeviceCount()**返回1，device 0 仅支持设备仿真模式。

E.2.2 cuDeviceGet()

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

返回原始范围在**[0, cuDeviceGetCount()-1]**的一个设备的句柄到指针***dev** 。

E.2.3 cuDeviceGetName()

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

返回一个识别设备**dev** 的ASCII 字符串到由**name** 指向的NULL-terminated 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);
```

返回在设备**dev** 上总的内存的字节数到指针***bytes**。

E.2.5 cuDeviceComputeCapability()

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

返回在设备**dev** 的计算兼容性的主要和次要版本到指针***major** 和***minor**。

E.2.6 cuDeviceGetAttribute()

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

返回设备**dev** 的特征的整数值**attrib** 到指针***value**。支持的特征是：

④ **CU_DEVICE_ATTRIBUTE_MAX_THREADS_PER_BLOCK**：每个块中最大的线程数；

④ **CU_DEVICE_ATTRIBUTE_MAX_BLOCK_DIM_X**：一个块的最大x-dimension；

④ **CU_DEVICE_ATTRIBUTE_MAX_BLOCK_DIM_Y**：一个块的最大y-dimension；

④ **CU_DEVICE_ATTRIBUTE_MAX_BLOCK_DIM_Z**：一个块的最大z-dimension；

④ **CU_DEVICE_ATTRIBUTE_MAX_GRID_DIM_X**：一个栅格的最大x-dimension；

④ **CU_DEVICE_ATTRIBUTE_MAX_GRID_DIM_Y**：一个栅格的最大y-dimension；

④ **CU_DEVICE_ATTRIBUTE_MAX_GRID_DIM_Z**：一个栅格的最大z-dimension

④ **CU_DEVICE_ATTRIBUTE_SHARED_MEMORY_PER_BLOCK**：每个块中共享内存的总byte数；

④ **CU_DEVICE_ATTRIBUTE_TOTAL_CONSTANT_MEMORY**：每个块中常驻内存的总byte数；

④ **CU_DEVICE_ATTRIBUTE_WARP_SIZE**：warp 的大小；

④ **CU_DEVICE_ATTRIBUTE_MAX_PITCH**：最大的pitch 数；

④ **CU_DEVICE_ATTRIBUTE_REGISTERS_PER_BLOCK**：每个块中寄存器的总数；

④ **CU_DEVICE_ATTRIBUTE_CLOCK_RATE**：时钟频率KHz；

CU_DEVICE_ATTRIBUTE_TEXTURE_ALIGNMENT：对齐需求；纹理地址对齐到**textureAlignment** 字节的，不需要在texture fetch 中应用offset

④ **CU_DEVICE_ATTRIBUTE_GPU_OVERLAP**：如果设备可以从主机到设备连续的拷贝内存，值是1， 否则是0。

E.2.7 cuDeviceGetProperties()

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

返回设备**dev** 的属性到指针***prop**。**CUdevprop** 的结构被定义为：

```
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：

④ **maxThreadsPerBlock** 是每个块中最大的线程数；

④ **maxThreadsDim[3]**是一个块中每个空间的最大大小；

④ **maxGridSize[3]**是一个栅格中每个空间的最大大小；

④ **sharedMemPerBlock** 是每个块中共享内存的总byte 数；

④ **totalConstantMemory** 是设备上常住内存的总byte 数；

④ **SIMDWidth** 是warp 的大小；

④ **memPitch** 是通过**cudaMallocPitch()**分配的内存空间中，允许内存拷贝函数的最大pitch；

④ **regsPerBlock** 是每个块中寄存器的总数；

④ **clockRate** 是时钟频率KHz；

④ **textureAlignment** 是对齐需求；纹理地址对齐到**textureAlignment** 字节的，不需要在texture fetch中应用offset 。

E.3 Context 管理

E.3.1 cuCtxCreate()

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

为设备创建一个新的context，并关联到调用的线程中。当前，**Flags** 参数必须是0。context 被创建时带有

使用计数器起始为1，并且**cuCtxCreate()**被调用完毕后，必须使用**cuCtxDetach()**。函数无法在线程中重复使用。

E.3.2 cuCtxAttach()

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

增加context 的使用计数器，并传回一个context 句柄到指针***pCtx**，当应用程序结束使用context，必须将其传递到**cuCtxDetach()**。如果当前线程中没有context，**cuCtxAttach()**调用失败。当前，**Flags** 参数必须是0。

E.3.3 cuCtxDetach()

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

减小context 的使用计数器，如果计数器为0，销毁context。

E.3.4 cuCtxGetDevice()

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

返回当前context 设备的顺序到指针***device**。

E.3.5 cuCtxSynchronize()

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

阻止直到设备上所有请求的任务执行完毕。**cuCtxSynchronize()**返回一个错误，如果其中的一个任务失败。

E.4 模块管理

E.4.1 cuModuleLoad()

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

取得一个文件名**filename**，并加载相关的模块**mod** 到当前的context 中。CUDA 驱动API 并不试图分配模块所需的资源；如果模块所需的内存和数据（常驻和全局）无法被分配，**cuModuleLoad()** 调用失败。文件通过**nvcc** 作为一个*cubin* 文件输出（参见部分4.2.5）。

E.4.2 cuModuleLoadData()

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

取得一个指针**image**，并加载相关的模块**mod** 到当前的context 中。指针可以通过映射一个*cubin* 文件获得，传递一个*cubin* 文件作为文本字符串，或者合并一个*cubin* 对象到可执行的资源中，并使用操作系统调用来获得指针，例如Windows 的**FindResource()** 。

E.4.3 cuModuleLoadFatBinary()

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

取得一个指针**fatBin**，并加载相关的模块**mod** 到当前的context 中。指针代表一个*fat binary* 对象，一个不同*cubin* 文件的集合，代表同样的设备代码，但为不同的架构编译和优化。目前没有整理好的API 用来构建和使用*fat binary* 对象，因此这个函数做为这个CUDA 版本的一个内部函数。

E.4.4 cuModuleUnload()

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

从当前的context 中卸载一个模块**mod**。

E.4.5 cuModuleGetFunction()

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

返回模块mod 中命名为funcname 的函数的句柄到指针*func。如果这个funcname 函数不存在，cuModuleGetFunction()返回CUDA_ERROR_NOT_FOUND。

E.4.6 cuModuleGetGlobal()

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

返回模块mod 中的基本指针到指针*devPtr，全局名称globalname 的大小到指针*bytes。如果名称globalname 的变量不存在，cuModuleGetGlobal()返回CUDA_ERROR_NOT_FOUND。参数devPtr 和bytes 为可选参数。如果其中的一个是空，它将被忽略。

E.4.7 cuModuleGetTexRef()

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

返回模块**mod** 中命名为**texrefname** 的texture reference 的句柄到指针***texref**。如果这个**texrefname** 的texture reference 不存在，**cuModuleGetTexRef()**返回**CUDA_ERROR_NOT_FOUND**。这个texture reference 句柄不需要销毁，因为当模块被卸载是，它将被销毁。

E.5 流管理

E.5.1 cuStreamCreate()

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

创建一个流。目前，**flags** 参数必须是0。

E.5.2 cuStreamQuery()

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

返回**CUDA_SUCCESS**，如果所有流中的操作完成。返回**CUDA_ERROR_NOT_READY**，如果不是。

E.5.3 cuStreamSynchronize()

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

阻止直到设备上所有请求的任务执行完毕。

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);
```

创建一个事件。目前，**flags** 参数必须是0。

E.6.2 cuEventRecord()

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

记录一个事件。如果**stream** 是非零的，当流中所有的操作完毕，事件被记录；否则，当CUDA context 中所有的操作完毕，事件被记录。由于这个操作是异步的，必须使用**cuEventQuery()**和/或**cuEventSyncronize()**来决定何时事件被真的记录了。如果**cuEventRecord** 之前被调用了，并且事件还没有被记录，函数返回**CUDA_ERROR_INVALID_VALUE**。

E.6.3 cuEventQuery()

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

返回CUDA_SUCCESS，如果事件被真的记录了。返回CUDA_ERROR_NOT_READY，如果不是。如果cuEventQuery()在这个事件中没有被调用，函数返回CUDA_ERROR_INVALID_VALUE。

E.6.4 cuEventSynchronize()

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

阻止直到事件被真的记录了。如果**cuEventRecord()**在这个事件中没有被调用，函数返回**CUDA_ERROR_INVALID_VALUE**。

E.6.5 cuEventDestroy()

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

销毁一个事件。

E.6.6 cuEventElapsedTime()

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

计算两个事件之间花费的时间（millisecond）。如果事件未被记录，函数返回**CUDA_ERROR_INVALID_VALUE**。如果带有一个非零的stream 事件被记录，结果是未定义的。

E.7 执行控制

E.7.1 cuFuncSetBlockShape()

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

指定线程块的**X**， **Y** 和**Z** 维度。

E.7.2 cuFuncSetSharedSize()

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

为每个线程指定**bytes** 大小的共享内存。

E.7.3 cuParamSetSize()

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

为函数设定参数的大小**numbytes**。

E.7.4 cuParamSeti()

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

为下次引入kernel 相关的**func** 时，设定一个整型参数。**offset** 是一个offset 字节。

E.7.5 cuParamSetf()

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

为下次引入kernel 相关的**func** 时，设定一个浮点参数。**offset** 是一个offset 字节。

E.7.6 cuParamSetv()

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

拷贝一个任意大小的数据到kernel 相关的**func** 的参数空间。**offset** 是一个offset 字节。

E.7.7 cuParamSetTexRef()

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

绑定CUDA 数组或线性内存到texture reference **texRef**，作为一个设备程序的纹理。在这个版本的CUDA 中，texture reference 必须通过**cuModuleGetTexRef()** 获得，并且**texunit** 参数必须设定到**CU_PARAM_TR_DEFAULT**。

E.7.8 cuLaunch()

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

引入kernel **func** 到一个1×1 的块栅格中。这个块包含由之前**cuFuncSetBlockShape()**指定的线程数量。

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);
```

引入kernel 到一个grid_width × rid_height grid 的块栅格中。这个块包含由之前cuFuncSetBlockShape()指定的线程数量。

cuLaunchGridAsync()是异步的，并且可以作为一个可选参数通过流使用。它只能应用于page-locked 的主机内存。如果使用一个指向pagable 的内存指针作为输入，函数将返回一个错误。

E.8 Memory 管理

E.8.1 cuMemGetInfo()

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

返回分配给CUDA context 的可用和总内存数到指针***free** 和***total**。

E.8.2 cuMemAlloc()

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

在设备上分配count 字节的线性内存，并返回分配内存的指针*devPtr。分配的内存适合任何类型的变量。如果**count** 是0，**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**widthInBytes*height** 字节的线性内存，并返回分配内存的指针***devPtr**。函数将确保在任何给出的行中对应的指针是连续的。**elementSizeBytes** 指定在内存中执行读和写的最大字节数。如果**elementSizeBytes** 小于kernel 实际读/写的大小，kernel 也能正确运行，但会降低速度。pitch 返回的指针***pitch** 的是分配的width in bytes 。Pitch 作为内存分配的一个分开的参数，用来计算2D 数组中的地址。一个给定行和列的类型**T** 的数组元素，地址等于：

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

由**cuMemAllocPitch()**返回的pitch 保证可以在任何情况下与**cuMemcpy2D()**一起工作。

对于2D 数组的分配，建议使用**cuMemAllocPitch()**分配内存。由于pitch 对列限制受限于硬件，特别是当应用程序从设备内存的不同区域执行一个2D 的内存拷贝（无论是线性内存还是CUDA 数组）。

E.8.4 cuMemFree()

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

释放被**devPtr** 指向的内存空间，**devPtr** 必须带有**cuMemAlloc()**或**cuMemAllocPitch()**的返回值。

E.8.5 cuMemAllocHost()

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

分配一个**count** 字节可用于设备访问的page-locked 的主机内存。驱动程序追踪由这个函数分配的虚拟内存的范围，并自动加速函数的调用，例如**cuMemcpy()**。由于内存可以被设备直接访问，相比由例如函数

malloc()分配的pagable 内存，拥有快很多的读写速度。但是，分配过多的page-locked 内存将减少系统可用物理内存的大小，从而降低系统整体的性能。

E.8.6 cuMemFreeHost()

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

释放被**hostPtr** 指向的内存空间，**hostPtr** 必须带有**cuMemAllocHost()**的返回值。

E.8.7 cuMemGetAddressRange()

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

返回包含输入指针**devPtr** 的**cuMemAlloc()**或**cuMemAllocPitch()**分配的基本地址到***basePtr**，分配的大小到***size**。两个参数都为可选参数。如果其中的一个是空，它将被忽略。

E.8.8 cuArrayCreate()

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

创建一个CUDA 数组根据**CUDA_ARRAY_DESCRIPTOR** 的结构**desc**，并返回一个句柄到这个新的CUDA 数组的地址到***array**。**CUDA_ARRAY_DESCRIPTOR** 的结构定义为：

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

其中：

④ **Width** 和 **Height** 是CUDA 数组的宽和高（在元素中）；如果**height** 是0，CUDA 数组是一维的，否则就是二维的；

④ **NumChannels** 指定每个CUDA 数组元素包含的组件数量；可以是1，2 或4；

④ Format 指定元素的格式；CUarray_format 定义为

```
typedefenumCUarray_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 数组描述的例子：

④ Description for a CUDA array of 2048 floats:

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

④ Description for a 64×64 CUDA array of floats:

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

④ Description for a **width×height** CUDA array of 64-bit, 4x16-bit float16's:

```
CUDA_ARRAY_DESCRIPTOR desc;
desc.FormatFlags = CU_AD_FORMAT_HALF;
desc.NumChannels = 4;
desc.Width = width;
desc.Height = height;
```

④ Description for a **width×height** CUDA array of 16-bit elements, each of which is two 8-bit unsigned chars:

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

E.8.9 cuArrayGetDescriptor()

```
CUresultcuArrayGetDescriptor(CUDA_ARRAY_DESC
                             CUarray array);
```

返回用来创建CUDA 数组**array** 的描述符的地址到***arrayDesc**。

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);
```

设定**count** 的内存范围8-，16-或32-bit 到指定的值**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);
```

设定**count** 的2D 内存范围8-，16-或32-bit 到指定的值**value**。**height** 指定行的数量，**dstPitch** 指定每行之间的字节数（参见部分E.8.3）。当pitch 是由**cuMemAllocPitch()** 返回的值时，执行速度最快。

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);
```

拷贝主机内存到设备内存。**dstDevPtr** 和**srcHostPtr** 分别指定目的基本地址和源基本地址。**count** 指定拷贝的字节数。

cuMemcpyHtoDAsync()是异步的，并且可以作为一个可选参数通过流使用。它只能应用于page-locked 的主机内存。如果使用一个指向pagable 的内存指针作为输入，函数将返回一个错误。

E.8.14 cuMemcpyDtoH()

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

拷贝设备内存到主机内存。**dstHostPtr** 和**srcDevPtr** 分别指定目的基本地址和源基本地址。**count** 指定拷贝的字节数。

cuMemcpyDtoHAsync() 是异步的， 并且可以作为一个可选参数通过流使用。它只能应用于page-locked 的主机内存。如果使用一个指向pagable 的内存指针作为输入，函数将返回一个错误。

E.8.15 cuMemcpyDtoD()

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

拷贝设备内存到设备内存。**dstDevPtr** 和**srcDevPtr** 分别指定目的基本地址和源基本地址。**count** 指定拷贝的字节数。

E.8.16 cuMemcpyDtoA()

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

拷贝设备内存到1D CUDA 数组。**dstArray** 和**dstIndex** 指定目的数据的CUDA 数组句柄和起始引索。**srcDevPtr** 指定源的基本指针。**count** 指定拷贝的字节数。

E.8.17 cuMemcpyAtoD()

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

拷贝1D CUDA 数组到设备内存。**dstDevPtr** 指定目的的基本指针，并且必须对齐到CUDA 数组元素。**srcArray**和**srcIndex**指定CUDA 数组句柄和数组元素的起始引索。

srcDevPtr 指定源的基本指针。**count** 指定拷贝的字节数，并且必须按数组元素的大小均匀的分配。

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);
```

拷贝1D CUDA 数组到主机内存。**dstHostPtr** 指定目的的基本指针。**srcArray** 和**srcIndex** 指定源数据的CUDA 数组句柄和起始引索。**count** 指定拷贝的字节数。**cuMemcpyAtoHAsync()**是异步的，并且可以作为一个可选参数通过流使用。它只能应用于page-locked 的主机内存。如果使用一个指向pagable 的内存指针作为输入，函数将返回一个错误。

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);
```

拷贝主机内存到1D CUDA 数组。**srcHostPtr** 指定目的的基本指针。**dstArray** 和**dstIndex** 指定目的数据的CUDA 数组句柄和起始引索。**count** 指定拷贝的字节数。

cuMemcpyHtoAAsync()是异步的，并且可以作为一个可选参数通过流使用。它只能应用于page-locked 的主机内存。如果使用一个指向pagable 的内存指针作为输入，函数将返回一个错误。

E.8.20 cuMemcpyAtoA()

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

拷贝一个1D CUDA 数组到另一个1D CUDA 数组。**dstArray** 和**srcArray** 分别指定目的CUDA 数组和源CUDA 数组的句柄。**dstIndex** 和**srcIndex** 分别指定目的CUDA 数组和源CUDA 数组的引索。这些值得范围是**[0, Width-1]**；它们不是字节offset 。**count** 指定拷贝的字节数。CUDA 数组中的元素不需要是相同的格式；但元素必须是同样大小，并且**count** 必须按数组元素的大小均匀的分配。

E.8.21 cuMemcpy2D()

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

根据copyParam 中指定的参数执行一个2D 内存拷贝。**CUDA_MEMCPY2D** 的结构定义为：

```
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：

④ **srcMemoryType** 和**dstMemoryType** 指定源内存和目的内存的类型；**Cumemorytype_enum** 被定义为：

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

如果**srcMemoryType**是**CU_MEMORYTYPE_HOST**, **srcHost**和**srcPitch**指定（主机）源数据的基本地址和每行的字节数。**srcArray**被忽略。

如果**srcMemoryType**是**CU_MEMORYTYPE_DEVICE**, **srcDevice**和**srcPitch**指定（设备）源数据的基本地址和每行的字节数。**srcArray**被忽略。

如果**srcMemoryType**是**CU_MEMORYTYPE_ARRAY**, **srcArray**指定（主机）源数据的句柄。**srcHost**, **srcDevice**和**srcPitch**被忽略。

如果**dstMemoryType**是**CU_MEMORYTYPE_HOST**, **dstHost**和**dstPitch**指定（主机）目的数据的基本地址和每行的字节数。**dstArray**被忽略。

如果**dstMemoryType**是**CU_MEMORYTYPE_DEVICE**, **dstDevice**和**dstPitch**指定（设备）目的数据的基本地址和每行的字节数。**dstArray**被忽略。

如果**dstMemoryType**是**CU_MEMORYTYPE_ARRAY**, **dstArray**指定（主机）目的数据的句柄。**dstHost**, **dstDevice**和**dstPitch**被忽略。

④ **srcXInBytes** 和**srcY** 指定源数据的基本地址。

对于主机指针, 起始地址为

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

对于设备指针, 起始地址为

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

对于CUDA数组，**srcXInBytes**必须按数组元素的大小均匀的分配。

④ **dstXInBytes** 和**dstY** 指定目的数据的基本地址。

对于主机指针, 起始地址为

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

对于设备指针, 起始地址为

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

对于CUDA数组，**dstXInBytes**必须按数组元素的大小均匀的分配。

④ **WidthInBytes** 和**Height** 指定2D 拷贝的宽度(in bytes)和高度。任何pitches 必须大于等于**WidthInBytes**.

如果pitch 超出最大允许值（参见**CU_DEVICE_ATTRIBUTE_MAX_PITCH** 部分E.2.6）**cuMemcpy2D()**将返回一个错误。**cuMemAllocPitch()**返回的pitch 总是可以和**cuMemcpy2D()**一起工作。在内部设备内存拷贝时（device↔device，CUDA array↔device，CUDA array↔CUDA array），如果pitch 不是由**cuMemAllocPitch()**计算得来，c**uMemcpy2D()**可能调用失败。**cuMemcpy2DUnaligned()**没有这个限制，但可能以极慢的速度运行，如果**cuMemcpy2D()**返回一个错误。

cuMemcpy2DAsync()是异步的，并且可以作为一个可选参数通过流使用。它只能应用于page-locked 的主机内存。如果使用一个指向pagable 的内存指针作为输入，函数将返回一个错误。

E.9 Texture Reference 管理

E.9.1 cuTexRefCreate()

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

创建一个texture reference ，并返回它的句柄到***texRef** 。一旦创建了， 应用程序必须调用**cuTexRefSetArray()**或**cuTexRefSetAddress()** 关联这个到分配的内存中。其它的texture reference 函数用来指定格式和解释（寻址， 过滤，等等），当这个texture reference 读取内存时。关联texture reference 到一个纹理，应用程序应该调用**cuParamSetTexRef()** 。

E.9.2 cuTexRefDestroy()

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

销毁texture reference 。

E.9.3 cuTexRefSetArray()

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

绑定CUDA 数组**array** 到texture reference **texRef** 。任何之前的地址或关联texture reference 的CUDA 数组被取代。**flags** 必须设定到**CU_TRSA_OVERRIDE_FORMAT**。任何之前绑定到**texRef** 的CUDA 数组将解除绑定。

E.9.4 cuTexRefSetAddress()

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

绑定线性地址范围到texture reference **texRef**。任何之前的地址或关联texture reference 的CUDA 数组被取代。任何之前绑定到**texRef** 的内存将解除绑定。

由于硬件对于基于纹理的地址有强制的队列需求，**cuTexRefSetAddress()**返回的，***offset** 指向offset 字节必须应用到纹理的fetch 中，从而读取所需的内存。这个offset 必须除以texel 的大小，并传递到kerenl 中，以便使用**tex1Dfetch()**函数。

如果设备内存的指针由**cuMemAlloc()**返回, offset 将是0 和NULL，并作为**ByteOffset** 的参数传递。

E.9.5 cuTexRefSetFormat()

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

指定被texture reference **texRef** 读取的数据格式。**format** 和**numPackedComponents** 完全类似于CUDA_ARRAY_DESCRIPTOR 的结构成员Format 和NumChannels：它们指定每个组件的格式和每个数组元素包含组件的数量。

E.9.6 cuTexRefSetAddressMode()

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

指定寻址模式**mod** 对于已知维度的texture reference **texRef**。如果**dim** 是零，寻址模式将使用函数的第一个参数拾取纹理（参见部分4.4.5)；如果**dim** 是1, 使用第二个参数, 依此类推。**CUaddress_mode** 被定义为：

```
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，这个调用不起作用。

E.9.7 cuTexRefSetFilterMode()

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

指定过滤模式**mode**，当通过texture reference **texRef** 读取内存时。**CUfilter_mode_enum** 被定义为：

```
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，这个调用不起作用。

E.9.8 cuTexRefSetFlags()

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

指定可选的flags 来控制通过texture reference 的数据的behavior 。可用的flag 有:

④ **CU_TRSF_READ_AS_INTEGER**, 抑制纹理在范围[0, 1]内转换整型数据到浮点数据;

④ **CU_TRSF_NORMALIZED_COORDINATES**, 抑制纹理的坐标范围[0, Dim)，其中Dim 是CUDA 数组的宽或高。取而代之的是，纹理坐标归一化为[0, 1.0)。

E.9.9 cuTexRefGetAddress()

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

返回绑定到texture reference **texRef** 的基本地址到指针***devPtr**，或者返回

CUDA_ERROR_INVALID_VALUE，如果texture reference 没有绑定到任何设备内存范围。

E.9.10 cuTexRefGetArray()

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

返回绑定到texture reference **texRef** 的CUDA 数组的地址到指针***array**，或者返回

CUDA_ERROR_INVALID_VALUE，如果texture reference 没有绑定到任何CUDA 数组。

E.9.11 cuTexRefGetAddressMode()

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

返回texture reference **texRef** 维度**dim** 的寻址模式到指针***mode**。目前，**dim** 的合法值为0 和1。

E.9.12 cuTexRefGetFilterMode()

```
CUresult cuTexRefGetFilterMode(CUfilter_mode* mode,
                               CUtexref texRef);
```

返回texture reference **texRef** 的过滤模式到指针***mode**。

E.9.13 cuTexRefGetFormat()

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

返回绑定到texture reference texRef 的CUDA 数组的组件的格式和数量到指针*format 和*numPackedComponents。如果format 或numPackedComponents 为空,它将被忽略。

E.9.14 cuTexRefGetFlags()

```
CUresult cuTexRefGetFlags(unsigned int* flags, CUtexref texRef);
```

返回texture reference **texRef** 的flag 到指针***flags**。

E.10 OpenGL Interoperability

E.10.1 cuGLInit()

```
CUresult cuGLInit(void);
```

初始化OpenGL interoperability 。它必须在任何其它OpenGL interoperability 函数使用之前被调用。如果所需的OpenGL 驱动设备无法使用，它将调用失败。

E.10.2 cuGLRegisterBufferObject()

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

注册ID 为**bufferObj** 的缓冲对象。这个函数必须被调用后，CUDA 才能映射这个缓冲对象。一旦注册了，这个缓冲对象不能被任何OpenGL 命令使用，除非作为OpenGL drawing 命令的一个数据源。

E.10.3 cuGLMapBufferObject()

```
CUresult cuGLMapBufferObject(CUdeviceptr* devPtr,
                             unsigned int* size,
                             GLuint bufferObj);
```

映射ID 为**bufferObj** 的缓冲对象到当前CUDA context 的地址空间中，并返回映射结果的基本指针和大小到指针***devPtr** 和***size**。

E.10.4 cuGLUnmapBufferObject()

```
CUresult cuGLUnmapBufferObject(GLuint bufferObj);
```

解除映射ID 为**bufferObj** 的缓冲对象。

E.10.5 cuGLUnregisterBufferObject()

```
CUresult cuGLUnregisterBufferObject(GLuint bufferObj);
```

解除注册ID 为**bufferObj** 的缓冲对象。

E.11 Direct3D Interoperability

E.11.1 cuD3D9Begin()

```
CUresult cuD3D9Begin(IDirect3DDevice9* device);
```

为Direct3D 设备**device** 初始化Interoperability。这个函数必须被调用后，CUDA 才能从设备映射任何对象。应用程序接下来可以映射Direct3D 设备拥有的顶点缓冲，直到**cuD3DEnd()** 被调用。

E.11.2 cuD3D9End()

```
CUresult cuD3D9End(void);
```

去除Direct3D 设备**device** 的Interoperability。

E.11.3 cuD3D9RegisterVertexBuffer()

```
CUresult cuD3D9RegisterVertexBuffer(IDirect3DVertexBuffer9* VB);
```

注册顶点缓冲**VB**。

E.11.4 cuD3D9MapVertexBuffer()

```
CUresult cuD3D9MapVertexBuffer(CUdeviceptr* devPtr,
                               unsigned int* size,
                               IDirect3DVertexBuffer9* VB);
```

映射顶点缓冲**VB** 到当前的CUDA context 中，并返回映射结果的基本指针和大小到指针***devPtr** 和***size**。

E.11.5 cuD3D9UnmapVertexBuffer()

```
CUresult cuD3D9UnmapVertexBuffer(IDirect3DVertexBuffer9* VB);
```

解除映射顶点缓冲**VB**。

E.11.6 cuD3D9UnregisterVertexBuffer()

```
CUresult cuD3D9UnregisterVertexBuffer(IDirect3DVertexBuffer9* VB);
```

解除注册顶点缓冲**VB**。

E.11.7 cuD3D9GetDevice()

```
cudaError_t

cuD3D9GetDevice(CUdevice* dev, const char* adapterName);
```

返回相关于适配器名称的设备到*dev，适配器名称adapterName 由EnumDisplayDevices 或IDirect3D9::GetAdapterIdentifier()获得。

附录F Texture Fetching

这个附录介绍一些方程式，这些方程式用来计算基于不同属性的texture reference（参见部分4.3.4）的纹理函数（参见部分4.4.5）的返回值。

绑定texture reference 的纹理可以表示成一个数组T ，一维纹理拥有N 个texel，二维纹理拥有$N \times M$ 个texel。使用纹理坐标x 和y 进行拾取。

一个纹理坐标必须符合T 的合法寻址范围。寻址模式指定一个超出范围的纹理坐标x 如何映射到合法的范围。如果x 是非归一化的，如果$x < 0$ ，钳制寻址模式将替换x 为0；如果$N \le x$ ， $N - 1$ 。如果是x 是归一化的：

④ 在钳制寻址模式中，如果$x < 0$ ，x 被替换为0；如果$1 \le x$ ，$1 - 1/N$ ，

④ 在warp 寻址模式中，x 被替换为$frac(x)$，其中$frac(x) = x - floor(x)$，$floor(x)$是不大于x 的最大的整数。

在余下的介绍中，纹理坐标x 和y 是非归一化的纹理坐标，映射到T 的合法寻址范围。x 和y 从归一化的纹理坐标$x\hat{}$和$y\hat{}$得来。比如：$x = Nx\hat{}$ 和$y = My\hat{}$。

F.1 Nearest-Point 采样

在这个过滤模式中，纹理拾取返回的值是：

④ $tex(x) = T[i]$ 对于一维纹理，

④ $tex(x,y) = T[i, j]$ 对于二维纹理，

其中i = x floor), j = y floor(y)

图F-1 展示了，当$N = 4$ 时，对于一维纹理的nearest-point 采样。

对于整型纹理，纹理拾取返回的值可以可选的映射到[0.0，1.0]（参见部分4.3.4.1）

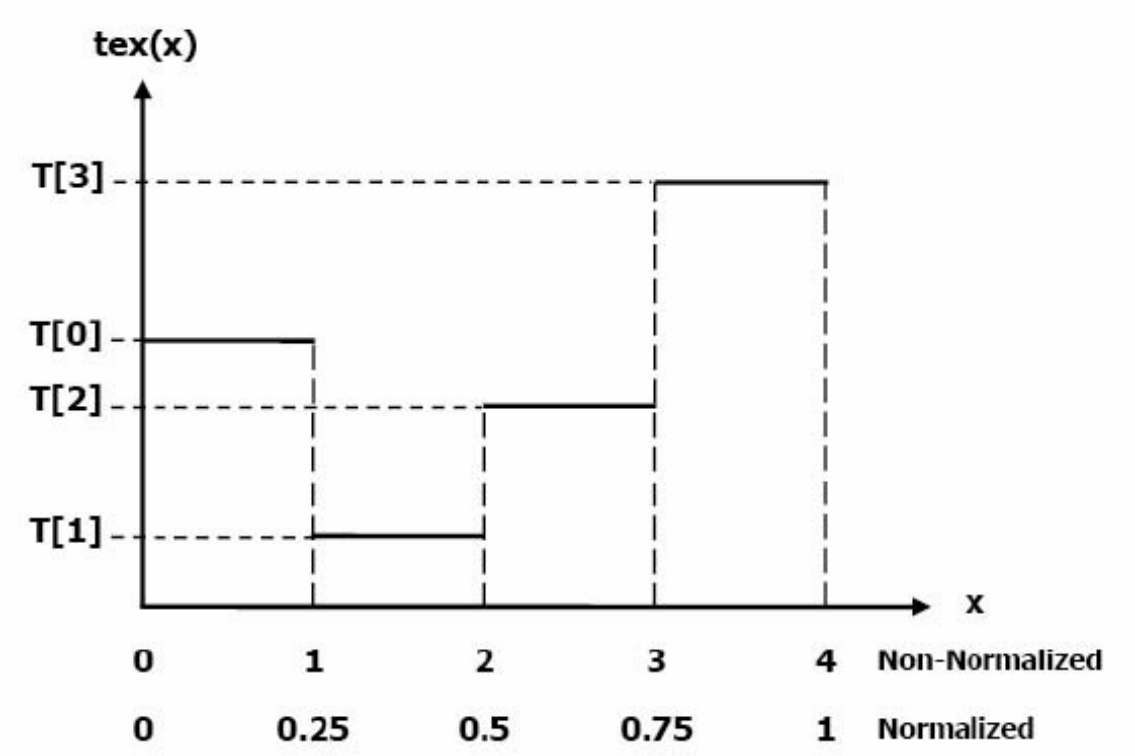


Figure F-1. Nearest-Point Sampling of a One-Dimensional Texture of Four Texels

F.2 线性过滤

在这个过滤模式中（仅支持浮点数纹理），纹理拾取返回的值是：

④ $tex(x) = (1-\alpha) i T] + \alpha i T + 1]$对于一维纹理，

④ $x\ tex(x) = (1-\alpha 1)(-\beta)\ i\ T,\ j] + \alpha(1-\beta)\ i\ T + ,1\ j] + (1-\alpha)\ \beta i\ T,\ j + 1] + \alpha\beta i\ T + ,1\ j + 1]$对于二维纹理，其中：

④ $i = floor(x_B), \alpha = x\ frac_B), x = x - ,5.0$

④ $j = floor(y), \beta = y\ frac), y = y - 5.0$

α and β are stored in 9-bit fixed point format with 8 bits of fractional value.

图F-2 展示了，当$N = 4$ 时，对于一维纹理的nearest-point 采样。

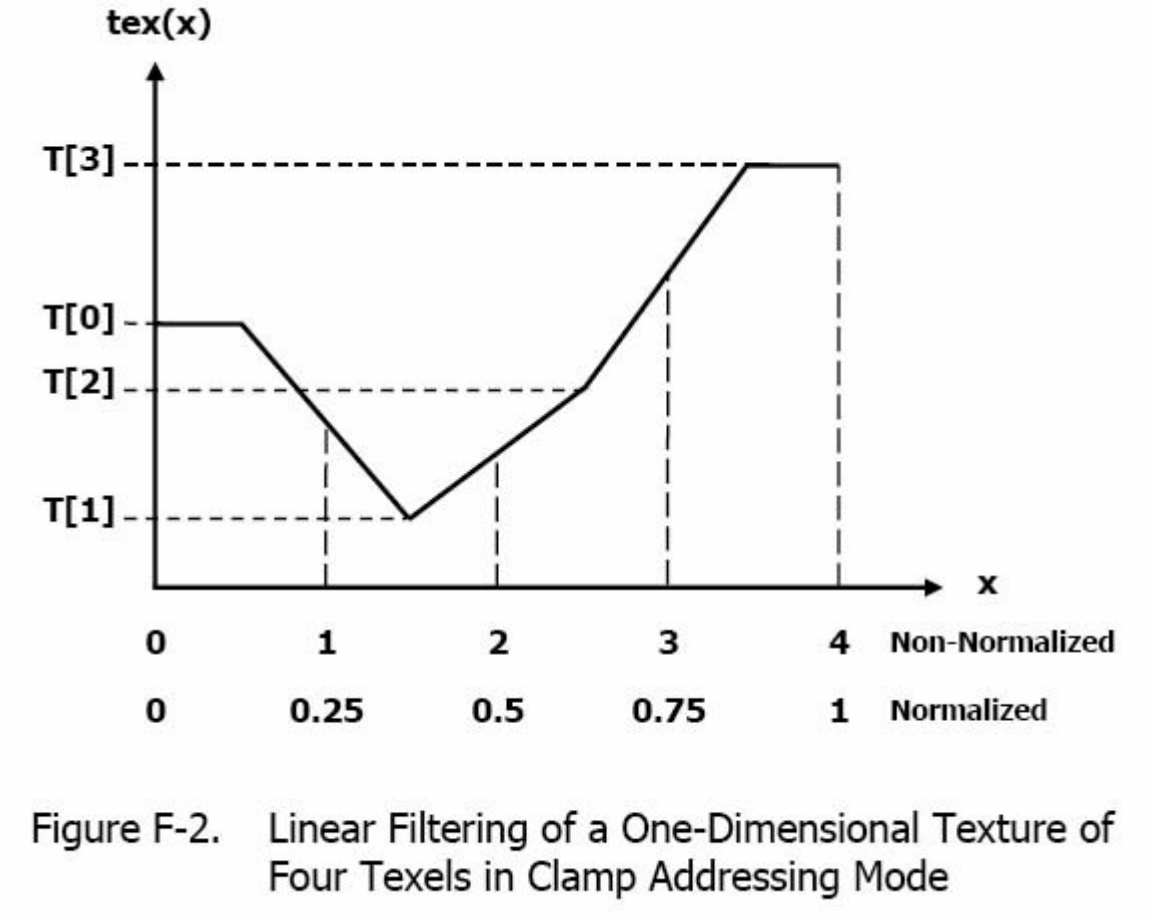


Figure F-2. Linear Filtering of a One-Dimensional Texture of Four Texels in Clamp Addressing Mode

F.3 查表法

A table lookup $TL(x)$ where x spans the interval $[0, R]$ can be implemented as $TL(x) = tex(\frac{N-1}{R}x + 0.5)$ in order to ensure that $TL(0) = T[0]$ and $TL(R) = T[N-1]$.

Figure F-3 illustrates the use of texture filtering to implement a table lookup with $R = 4$ or $R = 1$ from a one-dimensional texture with $N = 4$.

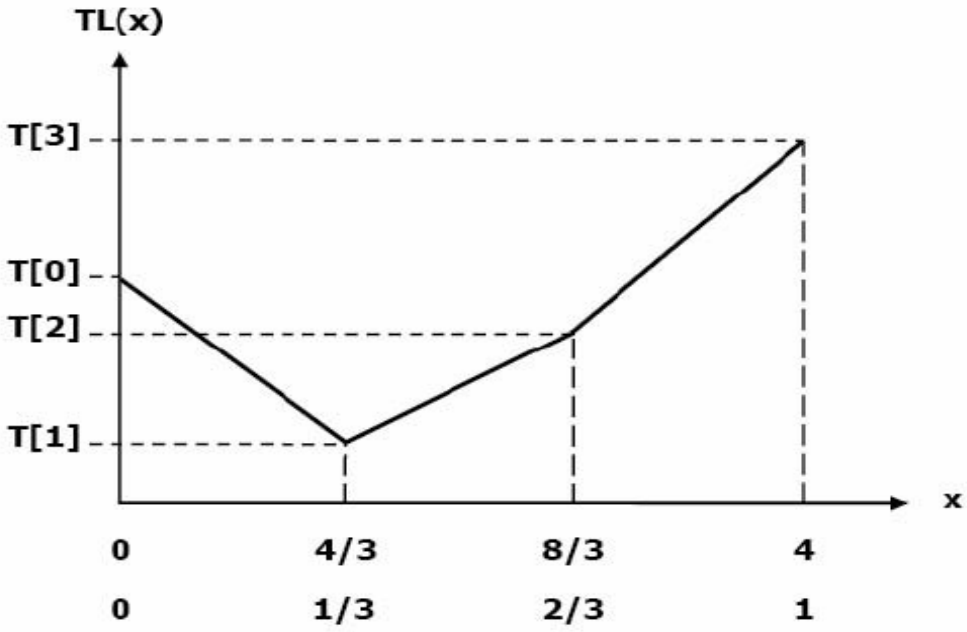


Figure F-3. One-Dimensional Table Lookup Using Linear Filtering