

# CUDA编程指南5.0中文版

风辰



# 目 录

目录 .....	i
第一章 导论 .....	1
1.1 从图形处理到通用并行计算 .....	1
1.2 CUDA <sup>TM</sup> : 一种通用并行计算架构 .....	3
1.3 一种可扩展的编程模型 .....	3
1.4 文档结构 .....	4
第二章 编程模型 .....	7
2.1 内核 .....	7
2.2 线程层次 .....	8
2.3 存储器层次 .....	11
2.4 异构编程 .....	11
2.5 计算能力 .....	11
第三章 编程接口 .....	15
3.1 用nvcc编译 .....	15
3.1.1 编译流程 .....	16
3.1.1.1 离线编译 .....	16
3.1.1.2 即时编译 .....	16
3.1.2 二进制兼容性 .....	17
3.1.3 PTX兼容性 .....	17
3.1.4 应用兼容性 .....	18
3.1.5 C/C++兼容性 .....	19
3.1.6 64位兼容性 .....	19
3.2 CUDA C运行时 .....	19

3.2.1	初始化 .....	20
3.2.2	设备存储器 .....	20
3.2.3	共享存储器 .....	24
3.2.4	分页锁定主机存储器 .....	32
3.2.4.1	可分享存储器(portable memory) .....	34
3.2.4.2	写结合存储器 .....	34
3.2.4.3	被映射存储器 .....	34
3.2.5	异步并发执行 .....	35
3.2.5.1	主机和设备间异步执行 .....	35
3.2.5.2	数据传输和内核执行重叠 .....	36
3.2.5.3	并发内核执行 .....	36
3.2.5.4	并发数据传输 .....	36
3.2.5.5	流 .....	37
3.2.5.6	事件 .....	41
3.2.5.7	同步调用 .....	42
3.2.6	多设备系统 .....	42
3.2.6.1	枚举设备 .....	42
3.2.6.2	设备指定 .....	42
3.2.6.3	流和事件行为 .....	43
3.2.6.4	p2p存储器访问 .....	44
3.2.6.5	p2p存储器复制 .....	45
3.2.6.6	统一虚拟地址空间 .....	45
3.2.6.7	错误检查 .....	46
3.2.7	调用栈 .....	47
3.2.8	纹理和表面存储器 .....	47
3.2.8.1	纹理存储器 .....	47
3.2.8.2	表面存储器(surface) .....	60
3.2.8.3	CUDA 数组 .....	65

3.2.8.4	读写一致性 .....	66
3.2.9	图形学互操作性 .....	66
3.2.9.1	OpenGL互操作性 .....	67
3.2.9.2	Direct3D互操作性 .....	70
3.2.9.3	SLI (速力) 互操作性 .....	82
3.3	版本和兼容性 .....	82
3.4	计算模式 .....	83
3.5	模式切换 .....	84
3.6	Windows上的Tesla计算集群模式 .....	85
<b>第四章</b>	<b>硬件实现 .....</b>	<b>87</b>
4.1	SIMT 架构 .....	87
4.2	硬件多线程 .....	88
<b>第五章</b>	<b>性能指南 .....</b>	<b>91</b>
5.1	总体性能优化策略 .....	91
5.2	最大化利用率 .....	91
5.2.1	应用层次 .....	91
5.2.2	设备层次 .....	92
5.2.3	多处理器层次 .....	92
5.3	最大化存储器吞吐量 .....	94
5.3.1	主机和设备的数据传输 .....	95
5.3.2	设备存储器访问 .....	96
5.3.2.1	全局存储器 .....	96
5.3.2.2	本地存储器 .....	98
5.3.2.3	共享存储器 .....	99
5.3.2.4	常量存储器 .....	100
5.3.2.5	纹理和表面存储器 .....	100
5.4	最大化指令吞吐量 .....	100

5.4.1	算术指令 .....	101
5.4.2	控制流指令 .....	104
5.4.3	同步指令 .....	105
<b>附录 A 支持CUDA的GPU .....</b>		<b>107</b>
<b>附录 B C语言扩展 .....</b>		<b>109</b>
B.1	函数类型限定符 .....	109
B.1.1	__device__ .....	109
B.1.2	__global__ .....	109
B.1.3	__host__ .....	109
B.1.4	__noinline__ 和 __forceinline__ .....	110
B.2	变量类型限定符 .....	110
B.2.1	__device__ .....	111
B.2.2	__constant__ .....	111
B.2.3	__shared__ .....	112
B.2.4	__restrict__ .....	113
B.3	内置变量类型 .....	115
B.3.1	char1、uchar1、char2、uchar2、char3、uchar3、char4、 uchar4、short1、ushort1、short2、ushort2、short3、ushort3、 short4、ushort4、int1、uint1、int2、uint2、int3、uint3、 int4、uint4、long1、ulong1、long2、ulong2、long3、ulong3、 long4、ulong4、float1、float2、float3、float4、double2 ...	115
B.3.2	dim3类型 .....	115
B.4	内置变量 .....	115
B.4.1	gridDim .....	115
B.4.2	blockIdx .....	115
B.4.3	blockDim .....	117
B.4.4	threadIdx .....	117
B.4.5	warpSize .....	117

B.5 存储器栅栏函数 .....	117
B.6 同步函数 .....	119
B.7 数学函数 .....	120
B.8 纹理函数 .....	120
B.8.1 纹理对象函数 .....	120
B.8.1.1 tex1Dfetch() .....	120
B.8.1.2 tex1D() .....	121
B.8.1.3 tex2D() .....	121
B.8.1.4 tex3D() .....	121
B.8.1.5 tex1DLayered() .....	121
B.8.1.6 tex2DLayered() .....	122
B.8.1.7 texCubemap() .....	122
B.8.1.8 texCubemapLayered() .....	122
B.8.1.9 tex2Dgather() .....	123
B.8.2 纹理参考函数 .....	123
B.8.2.1 tex1Dfetch() .....	123
B.8.2.2 tex1D() .....	124
B.8.2.3 tex2D() .....	124
B.8.2.4 tex3D() .....	125
B.8.2.5 tex1DLayered() .....	125
B.8.2.6 tex2DLayered() .....	125
B.8.2.7 texCubemap() .....	125
B.8.2.8 texCubemapLayered() .....	126
B.8.2.9 tex2Dgather() .....	126
B.9 表面函数(surface) .....	126
B.9.1 表面对象函数 .....	127
B.9.1.1 surf1Dread() .....	127
B.9.1.2 surf1Dwrite() .....	127

B.9.1.3	surf2Dread()	127
B.9.1.4	surf2Dwrite()	128
B.9.1.5	surf3Dread()	128
B.9.1.6	surf3Dwrite()	128
B.9.1.7	surf1DLayeredread()	129
B.9.1.8	surf1DLayeredwrite()	129
B.9.1.9	surf2DLayeredread()	129
B.9.1.10	surf2DLayeredwrite()	130
B.9.1.11	surfCubemapread()	130
B.9.1.12	surfCubemapwrite()	131
B.9.1.13	surfCubemapLayeredread()	131
B.9.1.14	surfCubemapLayeredwrite()	131
B.9.2	表面引用API	132
B.9.2.1	surf1Dread()	132
B.9.2.2	surf1Dwrite()	132
B.9.2.3	surf2Dread()	132
B.9.2.4	surf2Dwrite()	133
B.9.2.5	surf3Dread()	133
B.9.2.6	surf3Dwrite()	133
B.9.2.7	surf1DLayeredread()	134
B.9.2.8	surf1DLayeredwrite()	134
B.9.2.9	surf2DLayeredread()	135
B.9.2.10	surf2DLayeredwrite()	135
B.9.2.11	surfCubemapread()	135
B.9.2.12	surfCubemapwrite()	136
B.9.2.13	surfCubemapLayeredread()	136
B.9.2.14	surfCubemapLayeredwrite()	137
B.10	时间函数	137



B.11 原子函数 .....	137
B.11.1 数学函数 .....	138
B.11.1.1 atomicAdd() .....	138
B.11.1.2 atomicSub() .....	139
B.11.1.3 atomicExch() .....	139
B.11.1.4 atomicMin() .....	140
B.11.1.5 atomicMax() .....	140
B.11.1.6 atomicInc() .....	140
B.11.1.7 atomicDec() .....	141
B.11.1.8 atomicCAS() .....	141
B.11.2 位逻辑函数 .....	141
B.11.2.1 atomicAnd() .....	141
B.11.2.2 atomicOr() .....	142
B.11.2.3 atomicXor() .....	142
B.12 束表决 (warp vote) 函数 .....	142
B.13 束洗牌函数 .....	143
B.13.1 概览 .....	143
B.13.2 在束内广播一个值 .....	144
B.13.3 计算8个线程的前缀和 .....	145
B.13.4 束内求和 .....	146
B.14 取样计数器函数 .....	146
B.15 断言 .....	147
B.16 格式化输出 .....	148
B.16.1 格式化符号 .....	149
B.16.2 限制 .....	149
B.16.3 相关的主机端API .....	150
B.16.4 例程 .....	151
B.17 动态全局存储器分配 .....	152

B.17.1 堆存储器分配 .....	153
B.17.2 与设备存储器API的互操作 .....	154
B.17.3 例程 .....	154
B.17.3.1 每个线程的分配 .....	154
B.17.3.2 每个线程块的分配 .....	155
B.17.3.3 在内核启动之间持久的分配 .....	156
B.18 执行配置 .....	159
B.19 启动绑定 .....	160
B.20 #pragma unroll .....	162
B.21 SIMD 视频指令 .....	163
<b>附录 C 数学函数 .....</b>	<b>165</b>
C.1 标准函数 .....	165
C.1.1 单精度浮点函数 .....	165
C.1.2 双精度浮点函数 .....	168
C.2 内置函数 .....	171
C.2.1 单精度浮点函数 .....	172
C.2.2 双精度浮点函数 .....	172
<b>附录 D C++语言支持 .....</b>	<b>175</b>
D.1 代码例子 .....	175
D.1.1 数据类 .....	175
D.1.2 派生类 .....	176
D.1.3 类模板 .....	177
D.1.4 函数模板 .....	178
D.1.5 函子类 .....	178
D.2 限制 .....	180
D.2.1 预处理符号 .....	180
D.2.2 限定符 .....	180

D.2.2.1 设备存储器限定符 .....	180
D.2.2.2 Volatile限定符 .....	182
D.2.3 指针 .....	182
D.2.4 运算符 .....	183
D.2.4.1 赋值运算符 .....	183
D.2.4.2 地址运算符 .....	183
D.2.5 函数 .....	183
D.2.5.1 编译器生成的函数 .....	183
D.2.5.2 函数参数 .....	184
D.2.5.3 函数内静态变量 .....	184
D.2.5.4 函数指针 .....	184
D.2.5.5 函数递归 .....	185
D.2.6 类 .....	185
D.2.6.1 数据成员 .....	185
D.2.6.2 函数成员 .....	185
D.2.6.3 虚函数 .....	185
D.2.6.4 虚基类 .....	185
D.2.6.5 Windows相关 .....	185
D.2.7 模板 .....	186
<b>附录 E 纹理获取 .....</b>	<b>187</b>
E.1 最近点取样 .....	187
E.2 线性滤波 .....	187
E.3 查找表 .....	189
<b>附录 F 计算能力 .....</b>	<b>191</b>
F.1 特性和技术规范 .....	191
F.2 浮点标准 .....	195
F.3 计算能力1.x .....	198

F.3.1	架构	198
F.3.2	全局存储器	199
F.3.2.1	计算能力1.0和1.1的设备	199
F.3.2.2	计算能力1.2和1.3的设备	199
F.3.3	共享存储器	201
F.3.3.1	32位步长访问	201
F.3.3.2	32位广播访问	202
F.3.3.3	8位和16位访问	205
F.3.3.4	大于32位访问	205
F.4	计算能力2.x	206
F.4.1	架构	206
F.4.2	全局存储器	208
F.4.3	共享存储器	209
F.4.3.1	32位步长访问	209
F.4.3.2	大于32位访问	210
F.4.4	常量存储器	211
F.5	计算能力3.x	211
F.5.1	架构	211
F.5.2	全局存储器访问	212
F.5.3	共享存储器	213
F.5.3.1	64位模式	213
F.5.3.2	32位模式	213
附录 G	驱动API	215
G.1	上下文	218
G.2	模块	219
G.3	内核执行	220
G.4	运行时API和驱动API的互操作性	222
G.5	注意	223

## 表 格

5.1	原生算术指令吞吐量/每时钟每流多处理器操作数目 .....	102
B.1	内置类型的对齐要求 .....	116
C.1	数学标准库函数及其最大ULP 错误。最大错误表示为正确舍入 的单精度结果和CUDA 库函数返回的结果之差（以ulp 计算）的 绝对值 .....	165
C.2	数学标准库函数及其最大ULP 错误最大错误表示为正确舍入的 双精度结果和CUDA 库函数返回的结果之差(以ulp 计算)的绝对值 .....	169
C.3	受-use-fast-math影响的函数 .....	172
C.4	CUDA 运行时库支持的单精度浮点内部函数及其误差范围 .....	173
C.5	CUDA 运行时库支持的双精度浮点内置函数及其误差范围 .....	174
F.1	不同计算能力支持的特性 .....	191
F.2	依据计算能力的技术规范 .....	192
G.1	CUDA驱动API中可用的对象 .....	215



## 插 图

1.1 GPU和CPU每秒浮点操作数 .....	1
1.2 GPU和CPU存储器带宽 .....	2
1.3 GPU将更多的晶体管用于数据处理 .....	2
1.4 CUDA设计为支持多种语言和应用编程接口 .....	3
1.5 自动可扩展性 .....	5
2.1 线程块网格 .....	9
2.2 存储器层次 .....	12
2.3 异构编程 .....	13
3.1 没有共享存储器的矩阵相乘 .....	28
3.2 使用共享存储器的矩阵相乘 .....	33
3.3 驱动API是向后兼容，而非向前兼容 .....	83
E.1 4个像素的一维纹理最近点取样 .....	188
E.2 4个像素、钳位寻址模式下的一维纹理线性滤波 .....	189
E.3 使用线性滤波的一维查找表 .....	190
F.1 一个线程束访问全局存储器的例子，每个线程4字节和相关的基 于计算能力的存储器事务 .....	200
F.2 按步长访问共享存储器 .....	203
F.3 不规则的共享存储器访问 .....	204
G.1 库上下文管理 .....	219





# 第一章 导论

## 1.1 从图形处理到通用并行计算

市场对实时、高清晰度的三维图形具有无法满足的需求，由于这种需求的推动，可编程图形处理器（GPU）已经演化成高并行度，多线程，拥有强大计算能力和极高存储器带宽的多核处理器，如图1.1和图1.2所示：

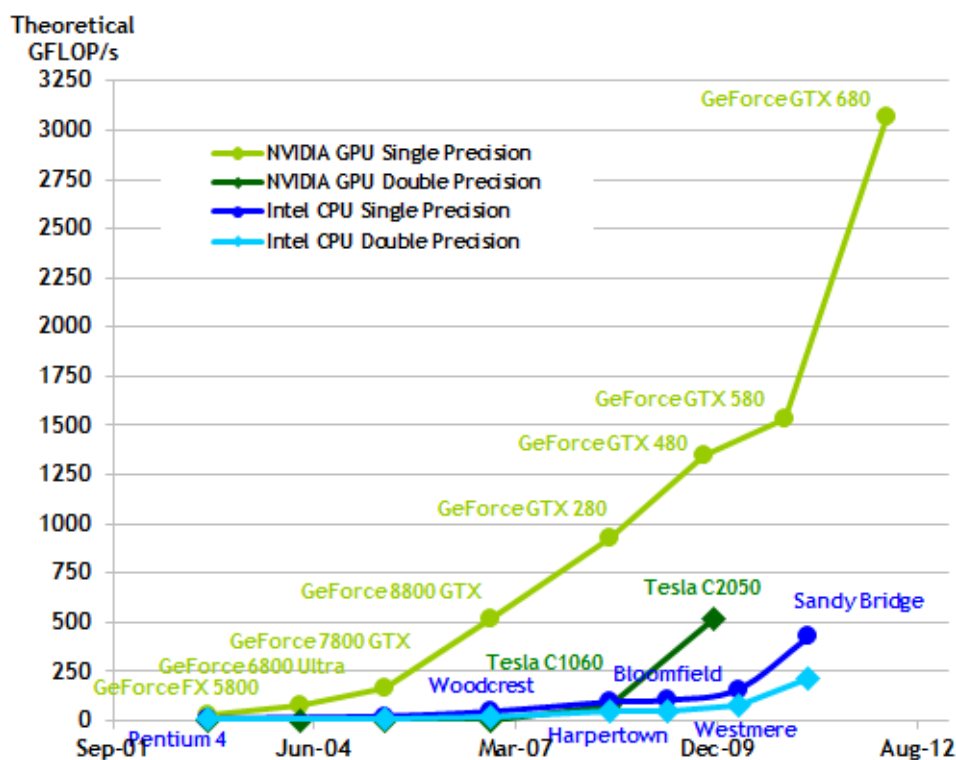


图 1.1: GPU和CPU每秒浮点操作数

GPU和CPU的浮点计算能力差异的原因是：GPU是特别为计算密集，高并行度计算（如同图像渲染）设计的，因此将更多的晶体管用于数据处理而不是数据缓存和流控，如图1.3所示。

特别地，GPU非常适合处理那些能够表示为数据并行计算（同一程序在多个数据上并行执行）的问题，数据并行计算的算术计算密度（算术操作和存储

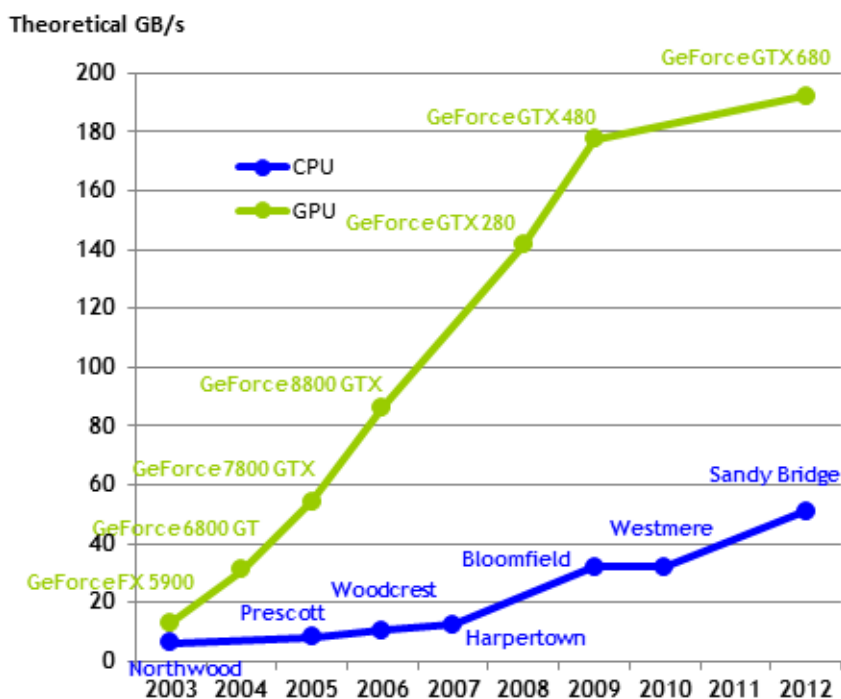


图 1.2: GPU和CPU存储器带宽

器操作的比例) 非常高。由于同一程序在每个元素上执行, 因此对复杂流控的要求非常少, 更因为在多个元素上执行和高计算密度, 访存延迟可以被计算隐藏, 因此无须大的数据缓存。

数据并行处理将数据元素映射到并行处理的线程上。很多处理大的数据集的应用可以使用数据并行处理模型加速。三维图像渲染处理中, 大量的像素和顶点被映射到并行线程。类似地, 图像和多媒体处理应用、图像渲染的后处理、视频编解码、图像缩放、立体视觉和模式识别能够将图像块和像素映射到



图 1.3: GPU将更多的晶体管用于数据处理

并行处理的线程。实际上，除了图像渲染和处理领域，还有很多算法已被数据并行处理加速，从普通信号处理或物理模拟到计算金融或计算生物学。

## 1.2 CUDA<sup>TM</sup>：一种通用并行计算架构

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

CUDA包含一个让开发者能够使用C作为高级编程语言的软件环境。如图1.4所示，其它的语言、应用编程接口（API）和基于编译制导的方式也被支持，如FORTRAN、Direct Compute和OpenACC。

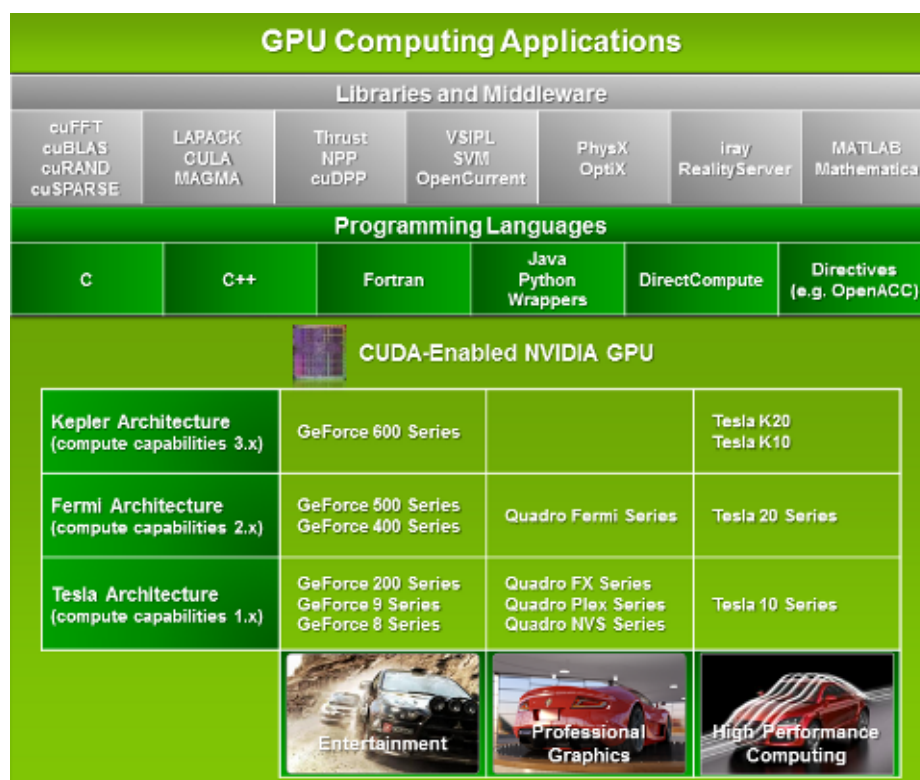


图 1.4: CUDA设计为支持多种语言和应用编程接口

## 1.3 一种可扩展的编程模型

多核CPU和众核GPU的出现，意味着主流处理器芯片现在已经是并行系

统了。更进一步的说，他们的并行度将继续以摩尔定律扩展。面临的挑战是开发透明的扩展并行度以利用不断增加的处理器核心数的应用软件，更像三维图形应用透明的扩展他们的并行度到不同数目核心的GPU上一样。

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

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

这些抽象提供了细粒度数据并行度和线程并行度，嵌套在粗粒度数据并行和任务并行中。他们引导程序员将问题划分为可以被多个块内线程独立并行处理的粗粒度子问题，而每个子问题又被分为可以被一个块内线程并行协作处理的更小的片段。这种分解通过在处理子问题的时候允许线程协作保持了语言的表达性，同时保证了自动可扩展性。事实上，每个块可被调度到可用处理器核心的任意一个上，以任何顺序，并行或者串行执行，这使得已编译好的CUDA程序能够在任意核心的GPU上执行，如图1.5所示，只有运行时系统需要知道物理处理器的数量。

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

注意：GPU围绕流多处理器阵列组建（查看四以了解更多细节）。多线程程序被划分为线程块，线程块的执行相互独立，所以程序在一个流多处理器多的GPU上执行时间自动（译者注：指不需要用户做任何工作）的比在一个流多处理器少的GPU上少。

## 1.4 文档结构

本文档包括以下各章

- 介绍：CUDA基本介绍
- 编程模型：CUDA编程模型要点
- 编程接口：编程接口描述

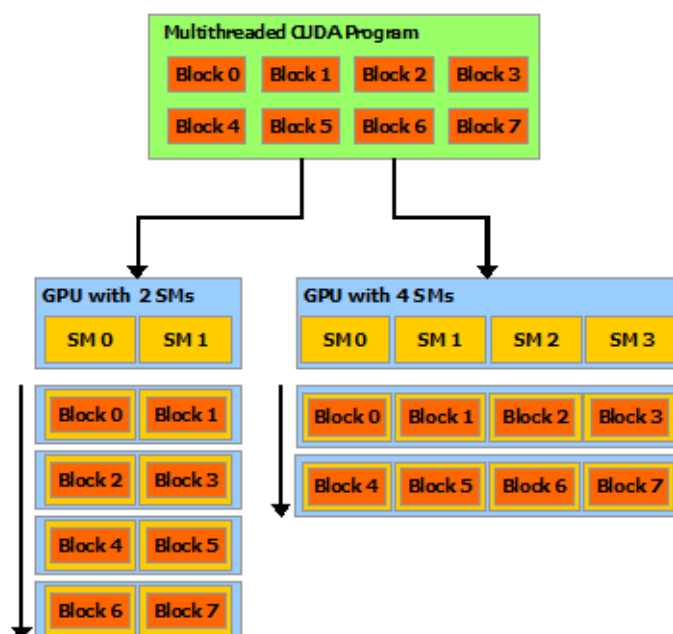


图 1.5: 自动可扩展性

- 硬件实现：硬件实现描述
- 性能指南：给出一些获得最高性能的优化指南
- 支持CUDA的GPU：给出所有支持CUDA的GPU列表
- C语言扩展：CUDA C扩展的详细说明
- 数学函数：CUDA支持的数学函数列表
- C/C++语言支持：设备代码支持的C++特性列表
- 纹理获取：更详细的纹理获取说明
- 计算能力：给出各种设备的技术规范，更多架构详细说明
- 驱动API：低层驱动API的介绍。



## 第二章 编程模型

本章引入了CUDA编程模型背后的主要概念，方式是概述它们是怎样使用C语言表示的。更多的关于CUDA C的描述在[三章](#)。

本章使用的向量相加例子的完整代码可在SDK中的vectorAdd代码样本中找到。

### 2.1 内核

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

内核使用\_\_global\_\_声明符定义，使用一种新<<< ... >>>执行配置语法指定执行某一指定内核的线程数（参看[B](#)）。每个执行内核的线程拥有一个独一无二的线程ID，可以通过内置的threadIdx变量在内核中访问（译者注：这只说明在块内是唯一的，并不一定是全局唯一的）。

下面的样本代码将两个长度为N的向量A和B相加，并将结果存入向量C中。

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

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

```

    ...
}

```

这里，N个线程中的每一个执行VecAdd()的一次成对加法（译者注：由于只使用了一个块，因此线程ID是唯一的）。

## 2.2 线程层次

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

线程索引和线程ID直接相关：对于一维的块，它们相同；对于二维长度为(Dx,Dy)的块，线程索引为(x,y)的线程ID是(x+yDx)；对于三维长度为(Dx,Dy,Dz)的块，索引为(x,y,z)的线程ID为(x+yDx+zDxDy)（译者注：这和我们使用C数组的方式不一样，大家注意理解）。

下面的例子代码将两个长度为N\*N的矩阵A和B相加，然后将结果写入矩阵C。

```

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

int main()
{
    ...
    // Kernel invocation with one block of N * N * 1 threads
    int numBlocks = 1;
    dim3 threadsPerBlock(N, N);
}

```



```

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

```

由于块内的所有线程必须存在于同一个处理器核心中且共享该核心有限的存储器资源，因此，一个块内的线程数目是有限的。在目前的GPU上，一个线程块可以包含多达1024个线程。

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

线程块被组织成一维、二维或三维的线程网格，如图2.1所示。一个网格内的线程块数往往由被处理的数据量而不是系统的处理器数决定，前者往往远超后者。

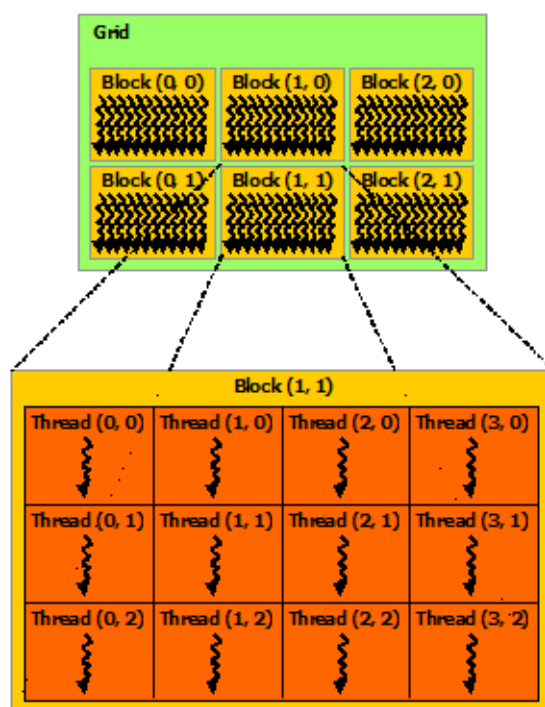


图 2.1: 线程块网格

线程块内线程数和网格内线程块数由<<< ... >>>语法确定，参数可以是整形或者dim3类型。二维的块或网格的尺寸可以以和上一个例子相同的方式指定。

网格内的每个块可以通过一维、二维或三维索引唯一确定，在内核中此索引可通过内置的`blockIdx`变量访问。块的尺寸(dimension)可以在内核中通过内置变量`blockDim`访问。

为了处理多个块，扩展前面的`MatAdd()`例子后，代码成了下面的样子。

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

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

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

线程块必须独立执行：而且能够以任意顺序，串行或者并行执行。这种独立性要求使得线程块可以以任何顺序在任意数目核心上调度，如1.5所示，保证了程序员能够写出能够随核心数目扩展的代码（enabling programmers to write code that scales with the number of cores）。

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

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

## 2.3 存储器层次

在执行期间，CUDA线程可能访问来自多个存储器空间的数据，如图2.2所示。每个线程有私有的本地存储器。每个块有对块内所有线程可见的共享存储器，共享存储器的生命期和块相同。所有的线程可访问同一全局存储器。

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

在同一应用中发射的内核之间，全局，常量和纹理存储器空间是持久的。

## 2.4 异构编程

如图2.3所示，CUDA编程模型假设CUDA线程在物理上独立的设备上执行，设备作为主机的协处理器，主机运行C程序。例如，内核在GPU上执行，而C程序的其它部分在CPU上执行就是这种模式。

CUDA编程模型同时假设主机和设备各自都维护着自己独立的DRAM存储器空间，各自被称为主机存储器空间和设备存储器空间。因此，程序通过调用CUDA运行时，来管理对内核可见的全局、常量和纹理存储器空间（参看三）。这包括设备存储器分配和释放，也包括在主机和设备间的数据传输。

注意：串行代码在主机上执行而并行代码在设备上执行

## 2.5 计算能力

设备的计算能力由主修订号和次修订号定义。

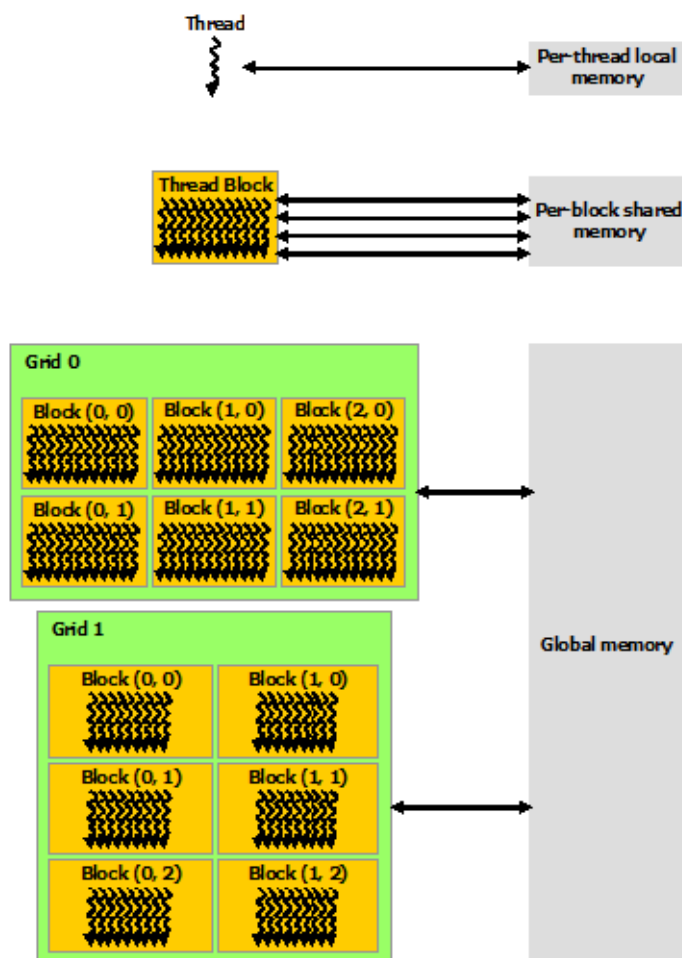


图 2.2: 存储器层次

主修订号相同的设备基于相同的核心架构。基于Kepler架构的设备的主修订号为3，基于Fermi架构的设备的主修订号为2，基于Tesla架构的设备的主修订号为1。

次修订号对应着对核心架构的增量提升，也可能包含了新特性。

支持CUDA的GPU列出了所有支持CUDA的设备和它们的计算能力。F给出了各计算能力设备的技术规范。

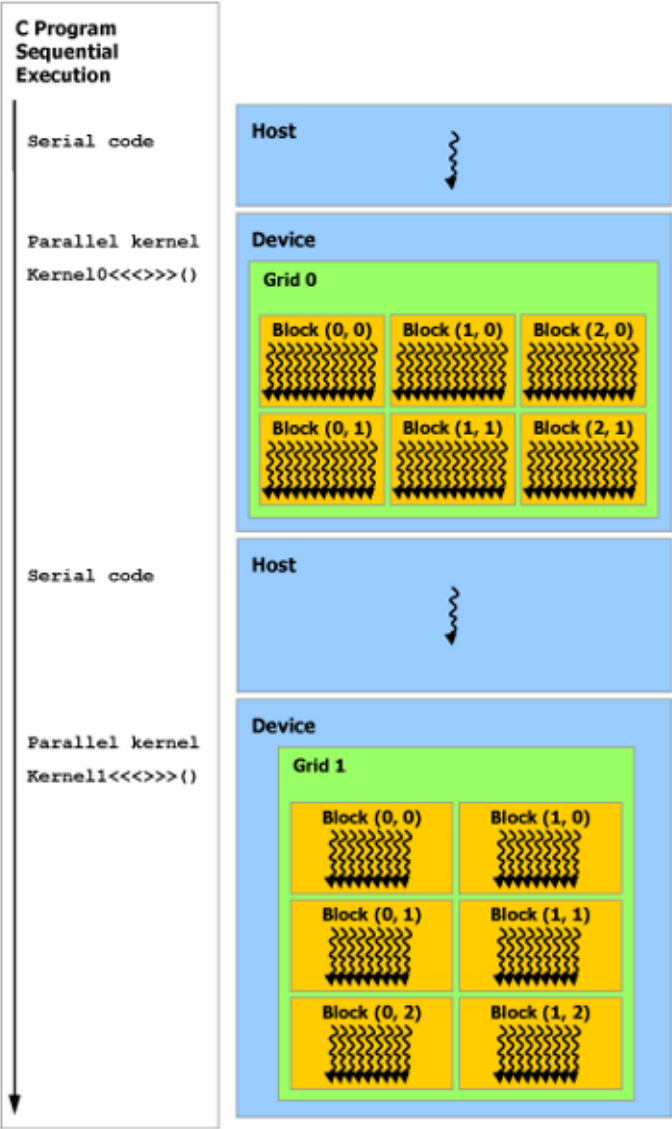


图 2.3: 异构编程



## 第三章 编程接口

CUDA C向熟悉C语言的用户提供了一种编写设备上执行的代码的简单路径。

CUDA C包括C的最小扩展集和一个运行时库。

[二](#)已经介绍了语言的核心扩展，这些扩展允许程序员像定义C函数一样定义内核和在每次内核调用时使用新的语法指定网格和块的大小。[B](#)提供了所有扩展的详尽描述。任何包含某些扩展的源文件必须使用nvcc编译，如使用nvcc编译节指出那样。

[3.2](#)介绍运行时API，运行时API在主机上执行，它提供了分配和释放设备存储器、在主机和显存间传输数据、管理多设备的系统的函数等等。详尽的描述请查看CUDA参考手册。

运行时API是基于驱动API构建的，应用也可以访问驱动API。驱动API通过展示低层的概念提供了额外的控制，如CUDA上下文一类似设备上的主机进程、CUDA模块一类似设备上的动态链接库。大多数应用不使用驱动API，因为在使用运行时(runtime)API时他们不需要这额外的控制，上下文和模块管理都是隐式的，因此代码更简明。[G](#)介绍了驱动API。详尽的描述请查看CUDA参考手册。

### 3.1 用nvcc编译

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

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

### 3.1.1 编译流程

#### 3.1.1.1 离线编译

nvcc可编译同时包含主机代码（在主机上执行的代码）和设备代码（在设备上执行的代码）的源文件。nvcc的基本流程包括分离主机和设备代码然后：

- 将设备代码编译成汇编形式（PTX代码）或者二进制形式（cubin对象），
- 将执行配置节引入的<<<, >>>语法转化为必要的CUDA C运行时函数调用以加载和启动每个已编译的内核（来自PTX代码或者cubin对象）。

修改后的主机代码要么被输出为C代码供其它工具编译，要么在编译的最后阶段被nvcc调用主机编译器输出为目标代码。

应用然后能够：

- 要么链接到生成的主机代码（这是最常见的情况），
- 要么忽略生成的主机代码（如果有）在设备上使用CUDA驱动API（参见[G](#)）装载和执行PTX源码或cubin对象。

#### 3.1.1.2 即时编译

任何在运行时被应用加载的PTX代码会被设备驱动进一步编译成二进制代码，这称为即时编译。即时编译增加了应用加载时间，但允许应用从最新编译器改进中获益，也是应用能够在未来硬件上运行的唯一方法，这些硬件在应用编译时还不存在。细节在[3.1.3](#)。

当设备驱动为某些应用即时编译某些PTX代码，它自动缓存生成的二进制代码的一个副本以避免在以后调用应用时重复编译。当设备驱动升级后该缓存（称为计算缓存）自动失效，所以应用能够从设备驱动内置的新的即时编译器获益。

环境变量可用于控制即时编译：

- 设置CUDA\_CACHE\_DISABLE为1使缓存失效（也就是没有二进制代码增加到缓存或从缓存中检索）。



- `CUDA_CACHE_MAXSIZE`以字节为单位指定了计算缓存的大小；默认尺寸是32MB，最大尺寸是4 GB；大小超过缓存尺寸的二进制代码不会被缓存；需要时会清理旧的二进制代码以为新二进制代码腾出空间。
- `CUDA_CACHE_PATH`指定了计算缓存文件存储的目录；默认值是：
  - Windows系统上，`%APPDATA\NVIDIA\ComputeCache`,
  - MacOS系统上，`$HOME/Library/Application\Support/NVIDIA/ComputeCache`,
  - Linux系统上，`~/.nv/ComputeCache`.
- 设置`CUDA_FORCE_PTX_JIT`为1强制设备驱动忽略任何嵌入在应用中的二进制代码（参见3.1.3）而即时编译嵌入的PTX代码；如果内核没有嵌入的PTX代码，加载失败；这个环境变量可以用于验证应用中是否嵌入了PTX代码和即时编译是否如预期工作以保证应用能够和将来的设备向前兼容。

### 3.1.2 二进制兼容性

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

### 3.1.3 PTX兼容性

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

为某些特殊计算能力生成的PTX代码始终能够被编译成相等或更高计算能力设备上的二进制代码。（译者注：PTX保证完全的向后兼容，而二进制只保证主修订号相同的向后兼容）

### 3.1.4 应用兼容性

为了在特定计算能力的设备上执行代码，应用加载的二进制或PTX代码必须满足如二进制兼容性节和PTX兼容性节说明的计算能力兼容性。特别地，为了能在将来更高计算能力（不能产生二进制代码）的架构上执行，应用必须装载PTX代码并为那些设备即时编译（参见3.1.1.2）。

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

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

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

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

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

例如，x.cu可有一个使用原子指令的优化代码途径，只能支持计算能力1.1或更高的设备。\_\_CUDA\_ARCH\_\_宏可以基于计算能力采用不同的代码路径。它只在设备代码中定义。例如，当使用“arch=compute\_11”编译时，\_\_CUDA\_ARCH\_\_等于110。

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

nvcc用户手册为-arch,-code和-gencode编译器选项列出了多种简写。如“arch=sm\_13”是“arch=compute\_13 tcode=compute\_13,sm\_13”的简写（等价于“-gencode arch=compute\_13,code='compute\_13,sm\_13'”）。

### 3.1.5 C/C++兼容性

编译器前端依据C++语法规则处理CUDA源文件。主机代码完整支持C++。设备代码只完整支持C++的一个子集，详见D。

### 3.1.6 64位兼容性

64位版本的nvcc以64位模式编译设备代码（也就是指针是64位的）。只有在主机代码是以64位模式编译的时候，设备代码才支持64位模式。

类似地，32位的nvcc以32位模式编译设备代码，使用32位模式编译的设备代码只支持以32位模式编译的主机代码。

32位的nvcc使用-m64编译选项以64位模式编译设备代码。

64位的nvcc使用-m32编译选项以32位模式编译设备代码。

## 3.2 CUDA C运行时

cuda动态库是运行时的实现，它包含在应用的安装包中，所有的函数前缀都是cuda。

如2.4所述，CUDA编程模型假设系统包含主机和设备，它们都有自己独立的存储器。3.2.2给出了一个操纵设备存储器的函数的简介。

3.2.3描述了如何使用线程层次引入的共享存储器以最大化性能。

3.2.4引入了分页锁定主机存储器，需要它以重叠内核执行和主机和设备间的数据传输。

3.2.5描述了支持系统中不同层次的异步并发执行的概念和API。

3.2.6节描述了展示了编程模型如何扩展到拥有连接多个设备的主机系统。

3.2.6.7描述了如何合适的检查主机产生的错误。

3.2.7提到操纵CUDA C调用栈的运行时函数。

3.2.8展现了纹理和表面存储器空间，它们提供了另一种访问设备存储器的方式；它们是GPU纹理硬件的一个子集。

3.2.9引入了多种运行时提供的函数，以和两大主要的图形API OpenGL和Direct3D互操作。

### 3.2.1 初始化

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

在初始化时，运行时为系统中的每个设备建立一个上下文（[G.1](#)提供了上下文的更多细节）。这个上下文作为设备的主要上下文，被应用中的主机线程共享。这些都是隐式发生的，运行时并没有将主要上下文展示给应用。

当主机线程调用`cudaDeviceReset()`时，这销毁了主机线程操作的设备的主要上下文。任何以这个设备为当前设备的主机线程调用的运行时函数将为设备重新建立一个主要上下文。

### 3.2.2 设备存储器

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

设备存储器可被分配为线性存储器或CUDA数组。

CUDA数组是不透明的存储器层次，为纹理获取做了优化。它们的细节在[3.2.8](#)描述。

计算能力1.x的设备，其线性存储器存在于32位地址空间内，计算能力2.0的设备，其线性存储器存在于40位地址空间内，所以独立分配的存储器实体能够通过指针引用，如二叉树。

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

```
// Device code
__global__ void VecAdd(float* A, float* B, float* C, int N)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < N)
        C[i] = A[i] + B[i];
}
```

```
}

// Host code
int main()
{
    int N = ...;
    size_t size = N * sizeof( float );

    // Allocate input vectors h_A and h_B in host memory
    float * h_A = (float*)malloc(size);
    float * h_B = (float*)malloc(size);

    // Initialize input vectors
    ...

    // Allocate vectors in device memory
    float * d_A;
    cudaMalloc(&d_A, size);
    float * d_B;
    cudaMalloc(&d_B, size);
    float * d_C;
    cudaMalloc(&d_C, size);

    // Copy vectors from host memory to device memory
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    // Invoke kernel
    int threadsPerBlock = 256;
    int blocksPerGrid =
        (N + threadsPerBlock - 1) / threadsPerBlock;
    VecAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, N);
```



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

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

```
// Host code
int width = 64, height = 64, depth = 64;
cudaExtent extent = make_cudaExtent(width * sizeof(float),
                                     height, depth);

cudaPitchedPtr devPitchedPtr;
cudaMalloc3D(&devPitchedPtr, extent);
MyKernel<<<100, 512>>>(devPitchedPtr, width, height, depth);

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

```
    }  
  }  
}
```

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

下面的例子代码复制了一些主机存储器数组到常量存储器中：

```
__constant__ float constData[256];  
float data[256];  
cudaMemcpyToSymbol(constData, data, sizeof(data));  
cudaMemcpyFromSymbol(data, constData, sizeof(data));  
  
__device__ float devData;  
float value = 3.14f;  
cudaMemcpyToSymbol(devData, &value, sizeof(float));  
  
__device__ float * devPointer;  
float * ptr;  
cudaMalloc(&ptr, 256 * sizeof(float));  
cudaMemcpyToSymbol(devPointer, &ptr, sizeof(ptr));
```

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

### 3.2.3 共享存储器

共享存储器使用`__shared__`限定词分配，详见B.2。

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



下面的代码是矩阵相乘的一个直接的实现，没有利用到共享存储器。每个线程读入A的一行和B的一列，然后计算C中对应的元素，如图3.1所示。这样，A读了B.width次，B读了A.height次。

```
// Matrices are stored in row-major order:
//  $M(row, col) = *(M.elements + row * M.width + col)$ 
typedef struct {
    int width;
    int height;
    float * elements;
} Matrix;

// Thread block size
#define BLOCK_SIZE 16

// Forward declaration of the matrix multiplication kernel
__global__ void MatMulKernel(const Matrix, const Matrix, Matrix);

// Matrix multiplication - Host code
// Matrix dimensions are assumed to be multiples of BLOCK_SIZE
void MatMul(const Matrix A, const Matrix B, Matrix C)
{
    // Load A and B to device memory
    Matrix d_A;
    d_A.width = A.width; d_A.height = A.height;
    size_t size = A.width * A.height * sizeof(float);
    cudaMalloc(&d_A.elements, size);
    cudaMemcpy(d_A.elements, A.elements, size,
               cudaMemcpyHostToDevice);
    Matrix d_B;
    d_B.width = B.width; d_B.height = B.height;
    size = B.width * B.height * sizeof(float);
```

```
    cudaMalloc(&d_B.elements, size);
    cudaMemcpy(d_B.elements, B.elements, size,
               cudaMemcpyHostToDevice);

    // Allocate C in device memory
    Matrix d_C;
    d_C.width = C.width; d_C.height = C.height;
    size = C.width * C.height * sizeof( float );
    cudaMalloc(&d_C.elements, size);

    // Invoke kernel
    dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
    dim3 dimGrid(B.width / dimBlock.x, A.height / dimBlock.y);
    MatMulKernel<<<dimGrid, dimBlock>>>(d_A, d_B, d_C);

    // Read C from device memory
    cudaMemcpy(C.elements, Cd.elements, size,
               cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(d_A.elements);
    cudaFree(d_B.elements);
    cudaFree(d_C.elements);
}

// Matrix multiplication kernel called by MatMul()
__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C)
{
    // Each thread computes one element of C
    // by accumulating results into Cvalue
    float Cvalue = 0;
    int row = blockIdx.y * blockDim.y + threadIdx.y;
```

```

    int col = blockIdx.x * blockDim.x + threadIdx.x;
    for (int e = 0; e < A.width; ++e)
        Cvalue += A.elements[row * A.width + e]
                  * B.elements[e * B.width + col];
    C.elements[row * C.width + col] = Cvalue;
}

```

下面的例子代码利用了共享存储器实现矩阵相乘。本实现中，每个线程块负责计算一个小方阵Csub，Csub是C的一部分，而块内的每个线程计算Csub的一个元素。如3.2所示。Csub等于两个长方形矩阵的乘积：A的子矩阵尺寸是（A.width,block\_size），行索引与Csub相同，B的子矩阵的尺寸是（block\_size,A.width），列索引与Csub相同。为了满足设备的资源，两个长方形的子矩阵分割为尺寸为block\_size的方阵，Csub是这些方阵积的和。每次乘法的计算是这样的，首先从全局存储器中将二个对应的方阵载入共享存储器中，载入的方式是一个线程载入一个矩阵元素，然后一个线程计算乘积的一个元素。每个线程积累每次乘法的结果并写入寄存器中，结束后，再写入全局存储器。

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

前面代码中的Matrix 类型增加了一个stride域，这样子矩阵能够用同样的类型有效表示。\_\_device\_\_函数（见B.1）用于读写元素和从矩阵中建立子矩阵。

```

// Matrices are stored in row-major order:
// M(row, col) = *(M.elements + row * M.stride + col)
typedef struct {
    int width;
    int height;
    int stride;
    float * elements;
} Matrix;

// Get a matrix element

```

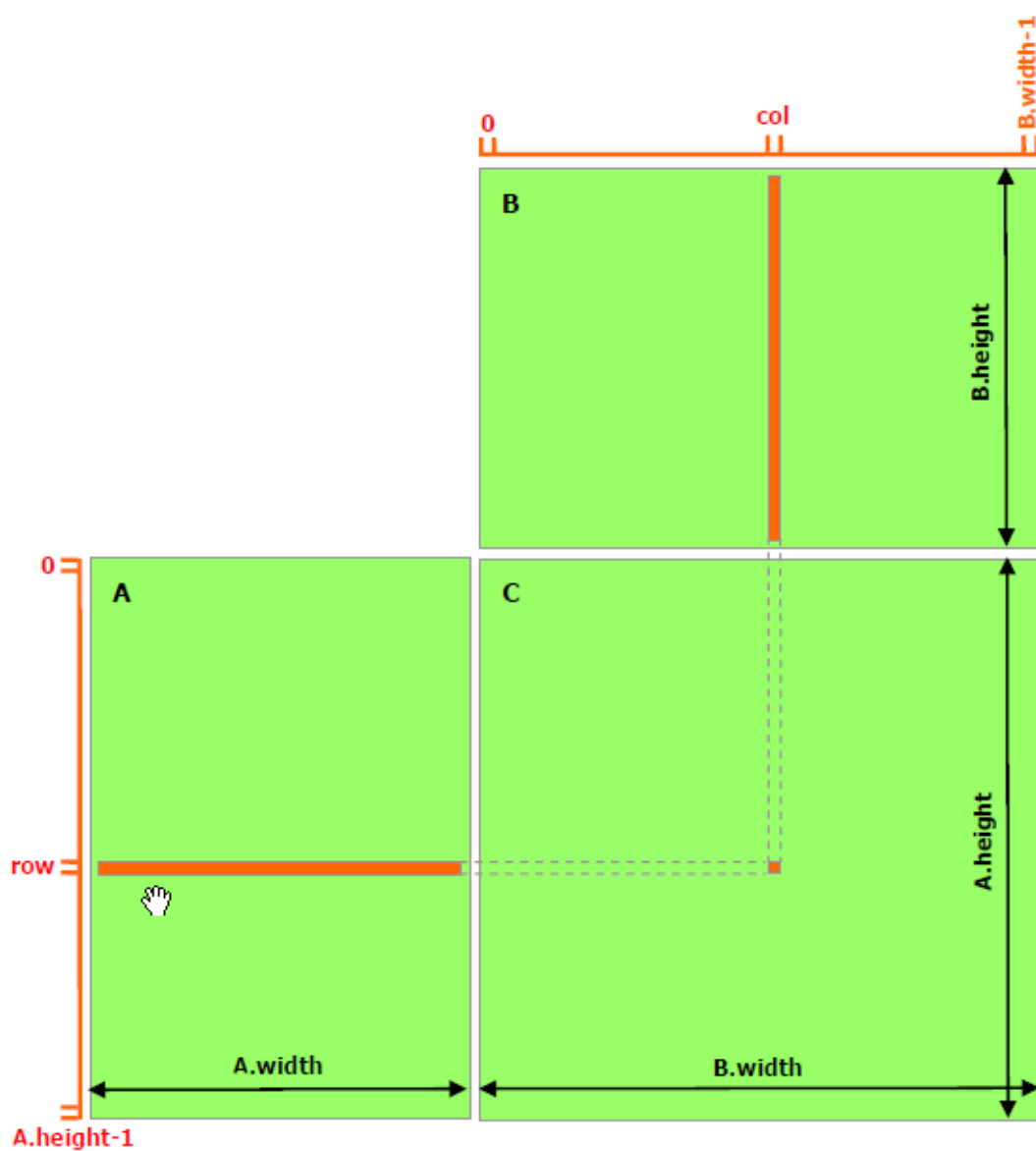


图 3.1: 没有共享存储器的矩阵相乘

```

__device__ float GetElement(const Matrix A, int row, int col)
{
    return A.elements[row * A.stride + col];
}

```

```
// Set a matrix element
__device__ void SetElement(Matrix A, int row, int col,
                           float value)
{
    A.elements[row * A.stride + col] = value;
}

// Get the BLOCK_SIZExBLOCK_SIZE sub-matrix Asub of A that is
// located col sub-matrices to the right and row sub-matrices down
// from the upper-left corner of A
__device__ Matrix GetSubMatrix(Matrix A, int row, int col)
{
    Matrix Asub;
    Asub.width  = BLOCK_SIZE;
    Asub.height = BLOCK_SIZE;
    Asub.stride = A.stride;
    Asub.elements = &A.elements[A.stride * BLOCK_SIZE * row
                                + BLOCK_SIZE * col];

    return Asub;
}

// Thread block size
#define BLOCK_SIZE 16

// Forward declaration of the matrix multiplication kernel
__global__ void MatMulKernel(const Matrix, const Matrix, Matrix);

// Matrix multiplication - Host code
// Matrix dimensions are assumed to be multiples of BLOCK_SIZE
void MatMul(const Matrix A, const Matrix B, Matrix C)
{
    // Load A and B to device memory
```

```
Matrix d_A;
d_A.width = d_A.stride = A.width; d_A.height = A.height;
size_t size = A.width * A.height * sizeof( float );
cudaMalloc(&d_A.elements, size);
cudaMemcpy(d_A.elements, A.elements, size,
           cudaMemcpyHostToDevice);

Matrix d_B;
d_B.width = d_B.stride = B.width; d_B.height = B.height;
size = B.width * B.height * sizeof( float );
cudaMalloc(&d_B.elements, size);
cudaMemcpy(d_B.elements, B.elements, size,
           cudaMemcpyHostToDevice);

// Allocate C in device memory
Matrix d_C;
d_C.width = d_C.stride = C.width; d_C.height = C.height;
size = C.width * C.height * sizeof( float );
cudaMalloc(&d_C.elements, size);

// Invoke kernel
dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
dim3 dimGrid(B.width / dimBlock.x, A.height / dimBlock.y);
MatMulKernel<<<dimGrid, dimBlock>>>(d_A, d_B, d_C);

// Read C from device memory
cudaMemcpy(C.elements, d_C.elements, size,
           cudaMemcpyDeviceToHost);

// Free device memory
cudaFree(d_A.elements);
cudaFree(d_B.elements);
cudaFree(d_C.elements);
```

```
}

// Matrix multiplication kernel called by MatMul()
__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C)
{
    // Block row and column
    int blockRow = blockIdx.y;
    int blockCol = blockIdx.x;

    // Each thread block computes one sub-matrix Csub of C
    Matrix Csub = GetSubMatrix(C, blockRow, blockCol);

    // Each thread computes one element of Csub
    // by accumulating results into Cvalue
    float Cvalue = 0;

    // Thread row and column within Csub
    int row = threadIdx.y;
    int col = threadIdx.x;

    // Loop over all the sub-matrices of A and B that are
    // required to compute Csub
    // Multiply each pair of sub-matrices together
    // and accumulate the results
    for (int m = 0; m < (A.width / BLOCK_SIZE); ++m) {

        // Get sub-matrix Asub of A
        Matrix Asub = GetSubMatrix(A, blockRow, m);

        // Get sub-matrix Bsub of B
        Matrix Bsub = GetSubMatrix(B, m, blockCol);
```

```
// Shared memory used to store Asub and Bsub respectively
__shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
__shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

// Load Asub and Bsub from device memory to shared memory
// Each thread loads one element of each sub-matrix
As[row][col] = GetElement(Asub, row, col);
Bs[row][col] = GetElement(Bsub, row, col);

// Synchronize to make sure the sub-matrices are loaded
// before starting the computation
__syncthreads();

// Multiply Asub and Bsub together
for (int e = 0; e < BLOCK_SIZE; ++e)
    Cvalue += As[row][e] * Bs[e][col];

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

// Write Csub to device memory
// Each thread writes one element
SetElement(Csub, row, col, Cvalue);
}
```

### 3.2.4 分页锁定主机存储器

运行时提供了使用分页锁定主机存储器（也称为pinned）的函数（与常规的使用malloc()分配的可分页的主机存储器不同）：



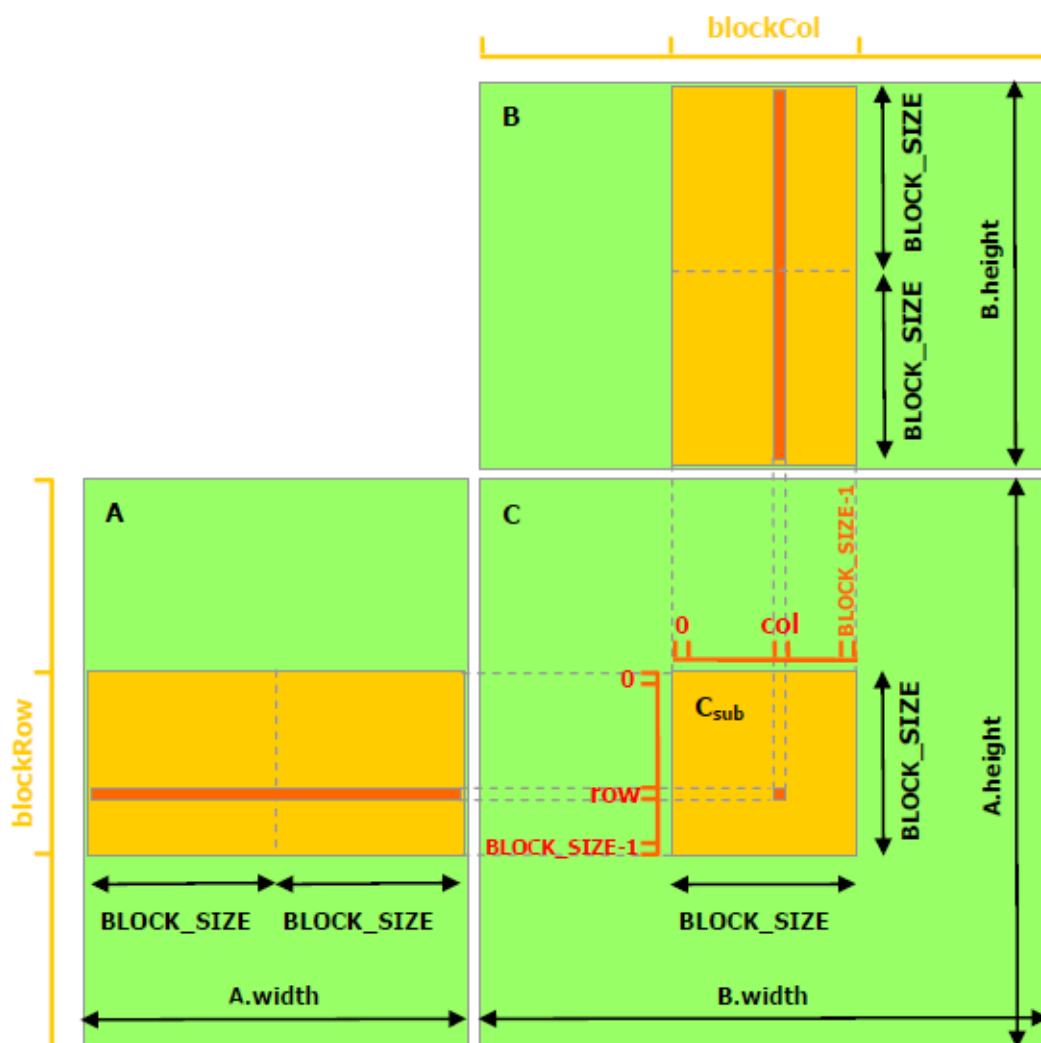


图 3.2: 使用共享存储器的矩阵相乘

- `cudaHostAlloc()`和`cudaFreeHost()`分配和释放分页锁定主机存储器；
- `cudaHostRegister()`分页锁定一段使用`malloc()`分配的存储器。

使用分页锁定主机存储器有许多优点：

- 如3.2.5提到的，在某些设备上，设备存储器和分页锁定主机存储器间数据拷贝可与内核执行并发进行；

- 在一些设备上，分页锁定主机内存可映射到设备地址空间，减少了和设备间的数据拷贝，详见[3.2.4.3](#)；
- 在有前端总线的系统上，如果主机存储器是分页锁定的，主机存储器和设备存储器间的带宽会高些，如果再加上[3.2.4.2](#)所描述的写结合（write-combining）的话，带宽会更高。

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

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

#### 3.2.4.1 可分享存储器(portable memory)

一块分页锁定存储器可被系统中的所有设备使用（参看[3.2.6](#)以了解更多的多设备系统细节），但是默认的情况下，上面说的使用分布锁定存储器的好处只有分配它时，正在使用的设备可以享有（如果可能的话，所有的设备共享同一个地址空间，参见[3.2.6.6](#)）。为了让所有线程可以使用分布锁定共享存储器的好处，可以在使用cudaHostAlloc()分配时传入cudaHostAllocPortable标签，或者在使用cudaHostRegister()分布锁定存储器时，传入cudaHostRegisterPortable标签。

#### 3.2.4.2 写结合存储器

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

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

#### 3.2.4.3 被映射存储器

在一些设备上，在使用cudaHostAlloc()分配时传入cudaHostAllocMapped标签或者在使用cudaHostRegister()分布锁定一块主机存储器时使用cudaHostRegi-

sterMapped标签，可分配一块被映射到设备地址空间的分页锁定主机存储器。这块存储器有两个地址：一个在主机存储器上，一个在设备存储器上。主机指针是从`cudaHostAlloc()`或`malloc()`返回的，设备指针可通过`cudaHostGetDevicePointer()`函数检索到，可以使用这个设备指针在内核中访问这块存储器。唯一的例外是主机和设备使用统一地址空间时，参见3.2.6.6。

从内核中直接访问主机存储器有许多优点：

- 无须在设备上分配存储器，也不用在这块存储器和主机存储器间显式传输数据；数据传输是在内核需要的时候隐式进行的。
- 无须使用流（参见3.2.5.5）重叠数据传输和内核执行；数据传输和内核执行自动重叠。

由于被映射分页锁定存储器在主机和设备间共享，应用必须使用流或事件（参见3.2.5）来同步存储器访问以避免任何潜在的读后写，写后读，或写后写危害。

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

如果设备不支持被映射分页锁定存储器，`cudaHostGetDevicePointer()`将会返回错误。应用可以检查`canMapHostMemory`属性应用以查询这种能力，如果支持映射分页锁定主机存储器，将会返回1。

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

### 3.2.5 异步并发执行

#### 3.2.5.1 主机和设备间异步执行

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

- 内核发射；

- 设备内两个不同地址间的存储器拷贝函数；
- 主机和设备内拷贝小于64KB的存储器块；
- 存储器拷贝函数中带有Async后缀的；
- 设置设备存储器的函数调用。

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

在下面的情形中，内核启动是同步的：

- 应用通过CUDA调试器或CUDA profiler（cuda-gdb, CUDA Visual Profiler, Parallel Nsight）运行时，所有的内核发射都是同步的。
- 通过剖分器（Nsight, Visual Profiler）收集硬件计数器。

### 3.2.5.2 数据传输和内核执行重叠

一些计算能力1.1或更高的设备可在内核执行时，在分页锁定存储器和设备存储器之间拷贝数据。应用可以通过检查`asyncEngineCount` 设备属性查询这种能力（参见3.2.6），如果其大于0，说明设备支持数据传输和内核执行重叠。对于计算能力1.x的设备，这种能力只支持不涉及CUDA数组和使用`cudaMallocPitch()`分配的二维数组的存储器拷贝（参见3.2.2）。

### 3.2.5.3 并发内核执行

一些计算能力2.x的设备可并发执行多个内核。应用可以检查`concurrentKernels`属性以查询这种能力）（参见3.2.6），如果等于1，说明支持。

计算能力3.5的设备最大可并发执行的内核数目是32，其余的是16。

来自不同CUDA上下文的内核不能并发执行。

使用了许多纹理或大量本地存储器的内核和其它内核并发执行的可能性比较小。

### 3.2.5.4 并发数据传输

在计算能力2.x的设备上，从主机分页锁定存储器复制数据到设备存储器和从设备存储器复制数据到主机分页锁定存储器，这两个操作可并发执行。

应用可以通过检查`asyncEngineCount` 属性查询这种能力，如果等于2，说明支持。

### 3.2.5.5 流

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

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

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

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

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

每个流将它的hostPtr输入数组的部分拷贝到设备存储器数组inputDevPtr，调用MyKernel()内核处理inputDevPtr，然后将结果outputDevPtr传输回hostPtr同样的部分。3.2.5.5描述了例子中的流如何依赖设备的计算能力重叠。必须注意为了使用重叠hostPtr必须指向分页锁定主机存储器。

调用cudaStreamDestroy()来释放流。

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

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

**默认流** 没有使用流参数的内核启动和主机设备间数据拷贝，或者等价地将流参数设为0，此时发射到默认流。因此它们顺序执行。

### 显式同步

有很多方法显式的在流之间同步。

cudaDeviceSynchronize()直到前面所有流中的命令都执行完。

cudaStreamSynchronize()以某个流为参数，强制运行时等待该流中的任务都完成。可用于同步主机和特定流，同时允许其它流继续执行。

cudaStreamWaitEvent()以一个流和一个事件为参数（参见事件节），使得在调用cudaStreamWaitEvent()后加入到指定流的所有命令暂缓执行直到事件完成。流可以是0，此时在调用cudaStreamWaitEvent()后加入到所有流的所有命令等待事件完成。

cudaStreamQuery()用于查询流中的所有之前的命令是否已经完成。

为了避免不必要的性能损失，这些函数最好用于计时或隔离失败的发射或存储器拷贝。

### 隐式同步

如果是下面中的任何一种操作在来自不同流的两个命令之间，这两个命令也不能并发：

- 分页锁定主机存储器分配，
- 设备存储器分配，

- 设备存储器设置，
- 设备内两个不同地址间的存储器拷贝函数；
- 默认流中调用的任何CUDA命令
- [F.4](#)描述的一级缓存/共享存储器之间配置切换。

对于计算能力3.0及以下且支持并发内核执行的设备，任何需要依赖检测以确定内核发射是否完成的操作：

- 只有来自CUDA上下文中任何流中，所有在被检测内核前面的内核启动的线程块开始执行，被检测的内核才能够开始执行；
- 会阻塞CUDA上下文中后面任何流中所有的内核发射直至被检测的内核发射完成。

需要依赖检测的操作包括同一个流中的一些其它类似被检测的启动命令和流中的任何cudaStreamQuery()调用。因此，应用应当遵守这些指导以提升潜在的内核并发执行：

- 所有独立操作应当在依赖操作之前发出，
- 任何类型同步尽量延后。

### 重叠行为

两个流的重叠执行数量依赖于发射到每个流的命令的顺序和设备是否支持数据传输和内核执行重叠（[3.2.5.2](#)）、并发内核执行（[3.2.5.3](#)）、并发数据传输（参见[3.2.5.4](#)）。

例如，在不支持并发数据传输的设备上，[3.2.5.5](#)例程的两个流并没有重叠，因为发射到流1的从主机到设备的存储器拷贝在发射到流0的从设备到主机的存储器拷贝之后，因此只有发射到流0的设备到主机的存储器拷贝完成它才开始。如果代码重写成如下方式（同时假设设备支持数据传输和内核执行重叠）。

```
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]);

```

此时发射到流1的从主机到设备的存储器拷贝和发射到流0的内核执行重叠。

在支持并发数据传输的设备上，3.2.5.5例程的两个流重叠：发射到流1的从主机到设备的存储器拷贝和发射到流0的设备到主机的存储器拷贝，甚至和发射到流0的内核执行（假设设备支持数据传输和内核执行重叠）。但是内核执行不可能重叠，因为发射到流1的第二个内核执行在发射到流0的设备到主机的存储器拷贝之后，因此会被阻塞直到发射到流0的内核执行完成。如果代码被重写成上面的样子，内核执行就重叠了（假设设备支持并发内核执行），因为发射到流1的第二个内核执行在发射到流0的设备到主机的存储器拷贝之前。然而在这种情况下，发射到流0的设备到主机的存储器拷贝只和发射到流1的内核执行的最后一个线程块重叠，这只占总内核执行时间的一小部分。

### 回调

运行时通过cudaStreamAddCallback()提供了一种在任何执行点向流插入回调的方式。回调是一个函数，一旦在插入点之前发射到流的所有命令执行完成，回调就会在主机上执行。在流0中的回调，只能在插入点之前其它流的所有命令都完成后才能执行。

下面的代码例子将回调函数MyCallbak插入到两个流中发射的主机到设备存储器的拷贝、内核执行和设备到主机的存储器拷贝操作之后。在每个设备到主机的存储器拷贝完成后该回调将会在主机上执行。

```

void CUDART_CB MyCallback(void *data){
    printf("Inside_callback_%.d\n", (int)data);
}
...
for (int i = 0; i < 2; ++i) {

```



```
    cudaMemcpyAsync(devPtrIn[i], hostPtr[i], size,  
                    cudaMemcpyHostToDevice, stream[i]);  
    MyKernel<<<100, 512, 0, stream[i]>>>(devPtrOut[i], devPtrIn[i],  
        size);  
    cudaMemcpyAsync(hostPtr[i], devPtrOut[i], size,  
                    cudaMemcpyDeviceToHost, stream[i]);  
    cudaStreamAddCallback(stream[i], MyCallback, (void*)i, 0);  
}
```

回调可通过在将其插入流时，使用`cudaStreamCallbackBlocking`标志指定为阻塞。在阻塞的回调之后发射到流中（如果回调插入在流0中，那么所有的发射到任何流中）的命令只有回调完成后才开始执行。

阻塞回调必须不能直接或间接的调用CUDA API，因为此时回调会等待自己，这导致死锁。

### 3.2.5.6 事件

通过在应用的任意点上异步地记载事件和查询事件是否完成，运行时提供了精密地监测设备运行进度和精确计时。当事件记载点前面，事件指定的流中的所有任务或者指定流中的命令全部完成时，事件被记载。只有记载点之前所有的流中的任务/命令都已完成，0号流的事件才会记载。

#### 创建和销毁

下面的代码创建了两个事件：

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

以下面的方式销毁它们：

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

#### 过去的时间

创建和销毁节建立的事件可以用下面的方式给创建和销毁节的代码计时：

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

### 3.2.5.7 同步调用

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

## 3.2.6 多设备系统

### 3.2.6.1 枚举设备

主机系统上可以有多个设备。下面的代码展示了怎样枚举这些设备、查询它们的属性、确定有多少个支持CUDA的设备。

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

```
printf("Device_%d_has_compute_capability_%d.%d.\n",
      device, deviceProp.major, deviceProp.minor);
}
```

### 3.2.6.2 设备指定

在任何时候，主机线程都可以使用`cudaSetDevice()`来设置它操作的设备。设备存储器分配和内核执行都作用在当前的设备上；流和事件关联当前设备。如果没有`cudaSetDevice()`调用，当前设备为0号设备。

下面的例程描述了设置当前设备如何影响存储器分配和内核执行。

```
size_t size = 1024 * sizeof( float );
cudaSetDevice(0);           // Set device 0 as current
float * p0;
cudaMalloc(&p0, size);      // Allocate memory on device 0
MyKernel<<<1000, 128>>>(p0); // Launch kernel on device 0
cudaSetDevice(1);          // Set device 1 as current
float * p1;
cudaMalloc(&p1, size);      // Allocate memory on device 1
MyKernel<<<1000, 128>>>(p1); // Launch kernel on device 1
```

### 3.2.6.3 流和事件行为

如下面的例程所示，如果内核执行和存储器拷贝发射到非关联到当前设备的流，它们将会失败。

```
cudaSetDevice(0);           // Set device 0 as current
cudaStream_t s0;
cudaStreamCreate(&s0);      // Create stream s0 on device 0
MyKernel<<<100, 64, 0, s0>>>(); // Launch kernel on device 0 in s0
cudaSetDevice(1);          // Set device 1 as current
cudaStream_t s1;
cudaStreamCreate(&s1);      // Create stream s1 on device 1
```

```
MyKernel<<<100, 64, 0, s1>>>(); // Launch kernel on device 1 in s1

// This kernel launch will fail:
MyKernel<<<100, 64, 0, s0>>>(); // Launch kernel on device 1 in s0
```

如果输入事件和输入流关联到不同的设备，`cudaEventRecord()`将失败。

如果两个输入事件关联到不同的设备，`cudaEventElapsedTime()`将会失败。

即使输入事件关联的设备并非当前设备，`cudaEventSynchronize()`和`cudaEventQuery()`也会成功。

即使输入流和输入事件关联到不同的设备，`cudaStreamWaitEvent()`也会成功。因此`cudaStreamWaitEvent()`可用于在不同的设备同步彼此。

每个设备有自己的默认流（参见3.2.5.5），因此在一个设备上发射到默认流的一个命令会和发射到另一个设备上默认流中的命令并发执行。

#### 3.2.6.4 p2p存储器访问

当应用以64位进程运行时，以TCC模式在win7/Vista、在win XP或者在Linux上，计算能力2.0或以上，Tesla系列设备能够访问彼此的存储器（即运行在一个设备上的内核可以解引用指向另一个设备存储器的指针）。只要两个设备上的`cudaDeviceCanAccessPeer()`返回true，这种p2p的存储器访问特性在它们间得到支持。

如下例所示，必须通过调用`cudaDeviceEnablePeerAccess()`启用两个设备间的p2p存储器访问支持。

两个设备使用统一存储器地址（参见3.2.6.6），因为同一指针可用于访问两个设备的存储器，如下面的代码所示。

```
cudaSetDevice(0); // Set device 0 as current
float * p0;
size_t size = 1024 * sizeof( float );
cudaMalloc(&p0, size); // Allocate memory on device 0
MyKernel<<<1000, 128>>>(p0); // Launch kernel on device 0
cudaSetDevice(1); // Set device 1 as current
```

```

cudaDeviceEnablePeerAccess(0, 0); // Enable peer-to-peer access
                                   // with device 0

// Launch kernel on device 1
// This kernel launch can access memory on device 0 at address p0
MyKernel<<<1000, 128>>>(p0);

```

### 3.2.6.5 p2p存储器复制

可以在两个不同设备间的存储器上复制存储器内容。

当两个设备使用统一存储器地址空间（参见3.2.6.6）时，使用设备存储器节提到的普通的存储器拷贝函数即可。否则使用cudaMemcpyPeer()、cudaMemcpyPeerAsync()、cudaMemcpy3Dpeer()或者cudaMemcpy3DpeerAsync()，如下面的代码所示。

```

cudaSetDevice(0); // Set device 0 as current
float * p0;
size_t size = 1024 * sizeof( float );
cudaMalloc(&p0, size); // Allocate memory on device 0
cudaSetDevice(1); // Set device 1 as current
float * p1;
cudaMalloc(&p1, size); // Allocate memory on device 1
cudaSetDevice(0); // Set device 0 as current
MyKernel<<<1000, 128>>>(p0); // Launch kernel on device 0
cudaSetDevice(1); // Set device 1 as current
cudaMemcpyPeer(p1, 1, p0, 0, size); // Copy p0 to p1
MyKernel<<<1000, 128>>>(p1); // Launch kernel on device 1

```

两个不同设备之间的存储器复制：

- 直到前面发射到任何一个设备的命令执行完，才开始执行
- 只有在它们执行完之后，后面发射到两者中任一设备的异步命令（参见3.2.5）可开始。

注意如果通过如p2p存储器访问节描述的`cudaDeviceEnablePeerAccess()`启用两个设备间的p2p访问，两个设备间的p2p存储器拷贝就没有必要通过主机进行，因此更快。

### 3.2.6.6 统一虚拟地址空间

对于计算能力2.0或以上的设备，当应用以64位进程运行时，以TCC模式在win7/Vista（只支持Tesla系列设备）、在win XP或者在Linux上，主机和设备使用单一的地址空间。主机通过`cudaHostAlloc()`分配的存储器和使用`cudaMalloc*()`在任意设备上分配的存储器使用这个虚拟地址空间；指针指向那个存储器空间（主机存储器或任意一个设备存储器）可以通过`cudaPointerGetAttributes()`确定。因此：

- 当在使用统一地址空间的设备间复制存储器时，`cudaMemcpy*()`中的`cudaMemcpyKind`参数没有作用，可用设置成`cudaMemcpyDefault`；
- 通过`cudaHostAlloc()`分配的存储器默认在使用统一地址空间的设备间是可分享的（参见3.2.4.1），`cudaHostAlloc()`返回的指针可由这些设备上的内核直接使用（即，无需使用`cudaHostGetDevicePointer()`获得设备指针）。

应用可以使用`unifiedAddressing`设备属性（参见3.2.6）查询某个设备是否使用统一地址空间，如果返回1，即支持。

### 3.2.6.7 错误检查

所有的运行时函数都返回错误码，但对于异步函数（参见3.2.5），由于会在任务结束前返回，因此错误码不能报告异步调用的错误；错误码只报告在任务执行之前的错误，典型的错误有关参数有效性；如果异步调用出错，错误将在后面某个无关的函数调用中出现。

唯一能够检查异步调用出错的方式是通过在异步调用函数后面使用`cudaDeviceSynchronize()`同步（或使用3.2.5介绍的其它同步机制），然后检查`cudaDeviceSynchronize()`的返回值。

运行时为每个主机线程维护着一个初始化为`cudaSuccess`的错误变量，每次错误发生（可以是参数不正确或异步错误）时，该变量会被错误码重写。

`cudaPeekAtLastError()`返回这个变量，`cudaGetLastError()`会返回这个变量，并将它重新设置为`cudaSuccess`。

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

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

### 3.2.7 调用栈

在计算能力2.x的设备上，调用栈的长度可以使用`cudaDeviceGetLimit()`查询，使用`cudaDeviceSetLimit()`设置。

当调用栈上溢时，如果通过CUDA调试器（`cuda-gdb`，`Parallel Nsight`）运行，内核会因为栈上溢失败，否则会出现无法确定的启动（`unspecified launch`）错误。

### 3.2.8 纹理和表面存储器

CUDA支持纹理硬件的一个子集，GPU使用这个子集访问纹理存储器和表面存储器以处理图形。如3.2.2所示，从纹理存储器或表面存储器而不是全局存储器中读数据有许多性能好处。

有两种不同的访问纹理和表面存储器的API:

- 所有设备都支持的纹理引用API,
- 只在计算能力3.x的设备上得到支持的纹理对象API.

纹理引用API具有纹理对象API没有限制。3.2.8.1提到了它们。

### 3.2.8.1 纹理存储器

如B.8所示，在内核中，调用纹理获取设备函数读纹理存储器。调用某个纹理获取函数读取纹理的过程称为纹理获取。每个纹理获取需要指定一个参数，如果使用纹理对象API，此参数称为纹理对象；如果是使用纹理引用API，此参数为纹理引用。

纹理对象或纹理引用指定：

- 纹理，纹理定义了被获取的纹理存储器部分。如3.2.8.1描述，纹理对象在运行创建，在建立纹理对象时需要指定纹理。纹理参考在编译时创建，必须使用运行时函数将纹理引用绑定到纹理。多种不同的纹理参考可能绑定到同一纹理或者绑定到存储器重叠的纹理。纹理可以是线性存储器的任何区域或一个CUDA数组。
- 维数，维数指定纹理是作为一维的数组使用一个纹理坐标、二维数组使用两个纹理坐标、还是三维数组使用三维坐标来寻址。数组的元素称为texels，是纹理元素的简称。纹理的宽度、高度和深度指数组每维的长度。F.2依据计算能力列出了最大纹理宽度、高度和深度。
- 纹理元素类型，它被限制在基本的整型、单精度浮点型和由char,short,int long, long long ,float, double定义的1, 2, 4个分量组成的向量类型。
- 读取模式，指cudaReadModeNormalizedFloat或cudaReadModeElementType。如果它是cudaReadModeNormalizedFloat且纹理元素是16位或者8位整形，实际返回值是浮点类型，对于无符号整型，整形全范围被映射到[0.0, 1.0]，对于有符号整型，映射成[-1.0, 1.0]；例如，无符号八位值为0xff的纹理元素映射为1；如果ReadMode是cudaReadModeElementType，不会进行转换；
- 纹理坐标是否归一化。默认情况下，纹理使用[0, N) 范围内的浮点坐标引用，其中N是坐标对应维度的尺寸。例如，尺寸为64\*32的纹理可引用的坐标范围是x维[0, 63]和y维[0, 31]。归一化的纹理坐标范围指定为[0.0, 1.0-1/N]而不是[0, N-1]，所以同样的64\*32纹理的归一化坐标x维和y维可寻址范围都是[0, 1.0-1/N]。归一化的纹理坐标天然的符合某些应用的要求，如果为了让纹理坐标独立于纹理尺寸，就更可取了



- 寻址模式，调用B.8设备函数时使用越界坐标是有效的。寻址模式定义了这种情况下的行为。默认寻址模式是将坐标钳位到有效范围：当使用非归一化纹理坐标时，为[0, N-1]；对于归一化坐标，为[0, 1.0)。如果指定了边界模式，越界访问会返回0。对于归一化坐标，循环模式和镜面模式也可用。当使用循环寻址模式时，每个坐标x会被转化成 $\text{frac}(x) = x - \text{floor}(x)$ ，其中 $\text{floor}(x)$ 指不大于x的最大整数。在使用镜面坐标时，每个坐标x会被转化成：如果 $\text{floor}(x)$ 是奇数， $1 - \text{frac}(x)$ ；否则 $\text{frac}(x)$ 。寻址模式以一个长度为三的数组指定，其第一、第二和第三个元素分别指定了纹理坐标三个方向的寻址模式。寻址模式是：`cudaAddressModeBorder`、`cudaAddressModeClamp`、`cudaAddressModeWrap`和`cudaAddressModeMirror`。其中`cudaAddressModeWrap`和`cudaAddressModeMirror`只支持归一化坐标。
- 滤波模式指定了纹理获取时返回值如何依据输入纹理坐标计算。线程纹理滤波只能对返回值配置为浮点型的纹理起作用。它在周围的纹理元素点上执行低精度插值。如果启用滤波，纹理获取点周围的点被读取，纹理获取点的返回值基于某些元素进行插值，纹理获取点坐标落入那些元素的坐标中间。对于一维的纹理进行简单的线性插值，二维纹理使用双线性插值，而三维纹理使用三线性插值。E给出了许多细节。滤波模式是：`cudaFilterModePoint`和`cudaFilterModeLinear`。如果是`cudaFilterModePoint`，返回值是最靠近纹理获取坐标的元素的值。如果是`cudaFilterModeLinear`，返回值是2（一维纹理）、4（二维纹理）、或8（三维纹理）个最接近纹理获取坐标进行线程插值。`cudaFilterModeLinear`只对返回值是浮点类型的纹理有效。

**纹理对象API** 使用`cudaCreateTextureObject()`从类型为`cudaResourceDesc`的资源描述符创建纹理对象，`cudaResourceDesc`定义如下：

```
struct cudaTextureDesc
{
    enum cudaTextureAddressMode addressMode[3];
    enum cudaTextureFilterMode filterMode;
    enum cudaTextureReadMode readMode;
    int sRGB;
```

```
    int normalizedCoords;
    unsigned int maxAnisotropy;
    enum cudaTextureFilterMode mipmapFilterMode;
    float mipmapLevelBias;
    float minMipmapLevelClamp;
    float maxMipmapLevelClamp;
};
```

其中：

- addressMode指定寻址模式；
- filterMode指定滤波模式；
- readMode指定寻址模式；
- normalizedCorrds指定纹理坐标是否归一化；

参考手册以了解sRGB, maxAnisotropy, mipmapFilterMode, mipmapLevelBias, minMipmapLevelClamp和maxMipmapLevelClamp。

下面的代码例子在纹理上应用了一些简单的转换。

```
// Simple transformation kernel
__global__ void transformKernel(float* output,
                                cudaTextureObject_t texObj,
                                int width, int height,
                                float theta)
{
    // Calculate normalized texture coordinates
    unsigned int x = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned int y = blockIdx.y * blockDim.y + threadIdx.y;

    float u = x / (float)width;
    float v = y / (float)height;
```

```
// Transform coordinates
u -= 0.5f;
v -= 0.5f;
float tu = u * cosf(theta) - v * sinf(theta) + 0.5f;
float tv = v * cosf(theta) + u * sinf(theta) + 0.5f;

// Read from texture and write to global memory
output[y * width + x] = tex2D<float>(texObj, tu, tv);
}

// Host code
int main()
{
    // Allocate CUDA array in device memory
    cudaChannelFormatDesc channelDesc =
        cudaCreateChannelDesc(32, 0, 0, 0,
                               cudaChannelFormatKindFloat);

    cudaArray* cuArray;
    cudaMallocArray(&cuArray, &channelDesc, width, height);

    // Copy to device memory some data located at address h_data
    // in host memory
    cudaMemcpyToArray(cuArray, 0, 0, h_data, size,
                     cudaMemcpyHostToDevice);

    // Specify texture
    struct cudaResourceDesc resDesc;
    memset(&resDesc, 0, sizeof(resDesc));
    resDesc.resType = cudaResourceTypeArray;
    resDesc.res.array.array = cuArray;

    // Specify texture object parameters
```

```
struct cudaTextureDesc texDesc;
memset(&texDesc, 0, sizeof(texDesc));
texDesc.addressMode[0] = cudaAddressModeWrap;
texDesc.addressMode[1] = cudaAddressModeWrap;
texDesc.filterMode      = cudaFilterModeLinear;
texDesc.readMode        = cudaReadModeElementType;
texDesc.normalizedCoords = 1;

// Create texture object
cudaTextureObject_t texObj = 0;
cudaCreateTextureObject(&texObj, &resDesc, &texDesc, NULL);

// Allocate result of transformation in device memory
float * output;
cudaMalloc(&output, width * height * sizeof(float));

// Invoke kernel
dim3 dimBlock(16, 16);
dim3 dimGrid((width + dimBlock.x - 1) / dimBlock.x,
             (height + dimBlock.y - 1) / dimBlock.y);
transformKernel<<<dimGrid, dimBlock>>>(output,
                                         texObj, width, height,
                                         angle);

// Destroy texture object
cudaDestroyTextureObject(texObj);

// Free device memory
cudaFreeArray(cuArray);
cudaFree(output);

return 0;
```

```
}

```

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

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

其中：

- **DataType**指定纹理元素数据类型；
- **Type**指定纹理参考的类型，且等于cudaTextureType1D（一维纹理），cudaTextureType2D（二维纹理）或cudaTextureType3D（三维纹理），或者cudaTextureType1Dlayered（一维层次纹理）或cudaTextureType2Dlayered（二维层次纹理），Type是可选的，默认为cudaTextureType1D；
- **ReadMode**指定读取模式。ReadMode是个可选参数，默认为cudaReadModeElementType。

纹理参考只能被声明为全局静态变量，且不能作为函数的参数传递。

其它的纹理引用属性是可变的，且能够在运行时通过主机运行时更改。如参考手册中所解释的，运行时API有一个低级的C风格的接口和一个高级的C++风格的接口。texture类型是在高级API中定义的一个结构体，公有继承自在低级API中定义的textureReference类型。textureReference定义如下：

```
struct textureReference {
    int normalized;
    enum cudaTextureFilterMode filterMode;
    enum cudaTextureAddressMode addressMode[3];
    struct cudaChannelFormatDesc channelDesc;
    int sRGB;
    unsigned int maxAnisotropy;
    enum cudaTextureFilterMode mipmapFilterMode;
    float mipmapLevelBias;
    float minMipmapLevelClamp;
```

```

float                                     maxMipmapLevelClamp;
}

```

- `normalized` 指定纹理坐标是否归一化；
- `filterMode` 指定滤波模式；
- `addressMode` 指定寻址模式；
- `channelDesc` 描述获取纹理时返回值的格式；它必须和 `DataType` 匹配；`channelDesc` 类型定义如下：

```

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

```

其中 `x`、`y`、`z` 和 `w` 是返回值各组件的位数，而 `f` 为：

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

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

在内核中使用纹理参考从纹理存储器中读取数据之前，对于线性存储器必须使用 `cudaBindTexture()` 或 `cudaBindTexture2D()`，对于 CUDA 数组，必须使用 `cudaBindTextureToArray()`，将纹理参考绑定到纹理。`cudaUnbindTexture()` 用于解绑定纹理参考。建议使用 `cudaMallocPitch()` 分配在线性存储器中的二维纹理，然后使用其返回的列长作为 `cudaBindTexture2D()` 的参数。

下面的代码将纹理参考绑定到 `devPtr` 指针指向的线性存储器：

- 使用低级 API：

```
texture<float, cudaTextureType2D,  
        cudaReadModeElementType> texRef;  
textureReference* texRefPtr;  
cudaGetTextureReference(&texRefPtr, texRef);  
cudaChannelFormatDesc channelDesc =  
        cudaCreateChannelDesc<float>();  
size_t offset ;  
cudaBindTexture2D(&offset, texRefPtr, devPtr, &channelDesc,  
        width, height, pitch);
```

- 使用高级API

```
texture<float, cudaTextureType2D,  
        cudaReadModeElementType> texRef;  
cudaChannelFormatDesc channelDesc =  
        cudaCreateChannelDesc<float>();  
size_t offset ;  
cudaBindTexture2D(&offset, texRef, devPtr, channelDesc,  
        width, height, pitch);
```

下面的代码将纹理绑定到CUDA数组cuArray:

- 使用低级API

```
texture<float, cudaTextureType2D, cudaReadModeElementType>  
texRef;  
textureReference* texRefPtr;  
cudaGetTextureReference(&texRefPtr, texRef);  
cudaChannelFormatDesc channelDesc;  
cudaGetChannelDesc(&channelDesc, cuArray);  
cudaBindTextureToArray(texRef, cuArray, &channelDesc);
```

- 使用高级API

```
texture<float, cudaTextureType2D,  
        cudaReadModeElementType> texRef;  
cudaBindTextureToArray(texRef, cuArray);
```

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

下面的代码在内核中应用了一些简单的转换。(译者注：该变换是沿中心旋转theta)

```
// 2D float texture  
texture<float, cudaTextureType2D, cudaReadModeElementType>  
texRef;  
  
// Simple transformation kernel  
__global__ void transformKernel(float* output,  
                                int width, int height,  
                                float theta)  
{  
    // Calculate normalized texture coordinates  
    unsigned int x = blockIdx.x * blockDim.x + threadIdx.x;  
    unsigned int y = blockIdx.y * blockDim.y + threadIdx.y;  
  
    float u = x / (float)width;  
    float v = y / (float)height;  
  
    // Transform coordinates  
    u -= 0.5f;  
    v -= 0.5f;  
    float tu = u * cosf(theta) - v * sinf(theta) + 0.5f;  
    float tv = v * cosf(theta) + u * sinf(theta) + 0.5f;
```



```
// Read from texture and write to global memory
output[y * width + x] = tex2D(texRef, tu, tv);
}

// Host code
int main()
{
    // Allocate CUDA array in device memory
    cudaChannelFormatDesc channelDesc =
        cudaCreateChannelDesc(32, 0, 0, 0,
                               cudaChannelFormatKindFloat);

    cudaArray* cuArray;
    cudaMallocArray(&cuArray, &channelDesc, width, height);

    // Copy to device memory some data located at address h_data
    // in host memory
    cudaMemcpyToArray(cuArray, 0, 0, h_data, size,
                     cudaMemcpyHostToDevice);

    // Set texture reference parameters
    texRef.addressMode[0] = cudaAddressModeWrap;
    texRef.addressMode[1] = cudaAddressModeWrap;
    texRef.filterMode     = cudaFilterModeLinear;
    texRef.normalized     = true;

    // Bind the array to the texture reference
    cudaBindTextureToArray(texRef, cuArray, channelDesc);

    // Allocate result of transformation in device memory
    float * output;
    cudaMalloc(&output, width * height * sizeof(float));
```

```
// Invoke kernel
dim3 dimBlock(16, 16);
dim3 dimGrid((width + dimBlock.x - 1) / dimBlock.x,
              (height + dimBlock.y - 1) / dimBlock.y);
transformKernel<<<dimGrid, dimBlock>>>(output, width, height,
                                         angle);

// Free device memory
cudaFreeArray(cuArray);
cudaFree(output);

return 0;
}
```

### 16位浮点纹理

CUDA数组支持的16位浮点或者半精度格式与IEEE-754-2008的binary2格式一样。

CUDA C不支持对应的数据类型，但提供了内置函数以通过unsigned short和32位浮点之间转换：`_float2half(float)`和`_half2float(unsigned short)`。这些函数只在设备代码中得到支持。对应的主机端代码可以在OpenEXR库中找到。

在纹理获取之中，在任何滤波进行之前，16位浮点组件提升到32位浮点。

可以使用`cudaCreateChannelDescHalf*`()函数建造一个16位浮点格式的通道描述。

### 层次纹理

一维或者二维的层次纹理（如Direct3D中的纹理数组，OpenGL中的数组纹理）是由一系列层次组成的纹理，这些层次通常具有相同维度、尺寸和数据类型的纹理。

一个一维的层次纹理使用整数索引和一个浮点纹理坐标寻址，以访问层次中的一个元素，这个索引标识一系列中的某一层。二维层次使用一个整形索引

标识纹理和两个浮点纹理坐标以访问层次中的一个像素。

层次纹理只能被绑定到以cudaArrayLayered标签（对于一维层次纹理高度为0）使用cudaMalloc3DArray()建造的CUDA数组。

层次纹理使用B.8.1.5和B.8.1.6描述的设备函数获取，纹理滤波只局限于一层而非多层。

层次纹理只在计算能力2.0及以上的设备上得到支持。

### 立方位图纹理

立方位图纹理是二维层次纹理的一种特殊类型，它具有六个层以表示立方体的六个面：

- 一层的高度等于宽度。
- 立方位图使用三个纹理坐标x,y,z寻址，这三个坐标解释为以立方体中心为原点指向立方体某个面的方向向量，并返回对应该面的纹理层的纹理元素。更确切的说：面通过最大的坐标m选取，且对应的层通过坐标(s/m+1)/2和(t/m+1)/2寻址，m、s和t的定义如下：

表1. 立方位图获取					
		face	m	s	t
$ x  >  y $ and $ x  >  z $	$x \geq 0$	0	x	-z	-y
	$x < 0$	1	-x	z	-y
$ y  >  x $ and $ y  >  z $	$y \geq 0$	2	y	x	z
	$y < 0$	3	-y	x	-z
$ z  >  x $ and $ z  >  y $	$z \geq 0$	4	z	x	-y
	$z < 0$	5	-z	-x	-y

层次纹理只能是CUDA数组，这个数组通过调用cudaMalloc3DArray()时使用cudaArrayCubemap标签创建。

立方位图纹理使用B.8.1.7描述的设备函数获取。

立方位图纹理只有计算能力2.0及以上的设备中得到支持。

### 层次立方位图纹理

层次立方位图纹理是一个层次纹理，其层由同样维度的立方位图组成。

层次立方体图纹理使用一个整数索引和三个浮点纹理坐标寻址；整数索引标记某个立方体图纹理，而三位坐标寻址该立方体图纹理的某个元素。

层次纹理只能是CUDA数组，这个数组通过调用cudaMalloc3DArray()时使用cudaArrayCubemap标签创建

层次立方体图纹理使用B.8.1.8描述的设备函数获取。纹理滤波只会在某层内进行而不会跨层进行。

层次立方体图纹理只有计算能力2.0及以上的设备中得到支持。

### 纹理收集

纹理收集是一种特殊的纹理获取，其只支持二维纹理。纹理收集通过tex2Dgather()函数执行，其参数和tex2D()相似，外加一个comp参数，comp只可能取0,1,2,3。tex2Dgather()返回4个32位数，这4个数是用于正常纹理获取的双线性插值的4个向量的分量。例如，如果这正常纹理获取得到的4个向量为(253,20,31,255)、(250,25,29,254)、(249,16,37,253)和(251,22,30,250)，且comp为2，则tex2Dgather()返回值为(31,29,37,30)。

纹理收集只支持使用cudaArrayTextureGather标签建立的CUDA数组，且长度要小于F.2中规定，这些规定要比正式的纹理获取小。

纹理收集只支持计算能力2.0以上的设备。

### 3.2.8.2 表面存储器(surface)

在计算能力2.0或以上的设备上，使用cudaArraySurfaceLoadStore标签建立的CUDA数组，可以通过表面对象或表面参考使用B.9描述的函数读写。

F.2根据计算能力列出了最大的表面宽度、高度和深度。

### 表面对象API

使用cudaCreateSurfaceObject()从类型为cudaResourceDesc的资源描述符创建表面对象。

下面的代码应用了一些简单的变换到纹理上。

```
// Simple copy kernel
__global__ void copyKernel(cudaSurfaceObject inputSurfObj,
                           cudaSurfaceObject outputSurfObj,
                           int width, int height)
```

```
{
    // Calculate surface coordinates
    unsigned int x = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned int y = blockIdx.y * blockDim.y + threadIdx.y;
    if (x < width && y < height) {
        uchar4 data;
        // Read from input surface
        surf2Dread(&data, inputSurfObj, x * 4, y);
        // Write to output surface
        surf2Dwrite(data, outputSurfObj, x * 4, y);
    }
}

// Host code
int main()
{
    // Allocate CUDA arrays in device memory
    cudaChannelFormatDesc channelDesc =
        cudaCreateChannelDesc(8, 8, 8, 8,
                               cudaChannelFormatKindUnsigned);

    cudaArray* cuInputArray;
    cudaMallocArray(&cuInputArray, &channelDesc, width, height,
                    cudaArraySurfaceLoadStore);

    cudaArray* cuOutputArray;
    cudaMallocArray(&cuOutputArray, &channelDesc, width, height,
                    cudaArraySurfaceLoadStore);

    // Copy to device memory some data located at address h_data
    // in host memory
    cudaMemcpyToArray(cuInputArray, 0, 0, h_data, size,
                      cudaMemcpyHostToDevice);
}
```

```
// Specify surface
struct cudaResourceDesc resDesc;
memset(&resDesc, 0, sizeof(resDesc));
resDesc.resType = cudaResourceTypeArray;

// Create the surface objects
resDesc.res.array.array = cuInputArray;
cudaSurfaceObject inputSurfObj = 0;
cudaCreateSurfaceObject(&inputSurfObj, &resDesc);
resDesc.res.array.array = cuOutputArray;
cudaSurfaceObject outputSurfObj = 0;
cudaCreateSurfaceObject(&outputSurfObj, &resDesc);

// Invoke kernel
dim3 dimBlock(16, 16);
dim3 dimGrid((width + dimBlock.x - 1) / dimBlock.x,
              (height + dimBlock.y - 1) / dimBlock.y);
copyKernel<<<dimGrid, dimBlock>>>(inputSurfObj,
                                   outputSurfObj,
                                   width, height);

// Destroy surface objects
cudaDestroySurfaceObject(inputSurfObj);
cudaDestroySurfaceObject(outputSurfObj);

// Free device memory
cudaFreeArray(cuInputArray);
cudaFreeArray(cuOutputArray);

return 0;
}
```

### 表面参考API

表面参考定义在文件域内，声明为surface类型

```
surface<void, Type> surfRef;
```

其中Type指定表面参考的类型，其值为cudaSurfaceType1D, cudaSurfaceType2D, cudaSurfaceType3D, cudaSurfaceTypeCubemap, cudaSurfaceType1D-Layered, cudaSurfaceType2DLayered, cudaSurfaceTypeCubemapLayered；Type是可选的，其默认值为cudaSurfaceType1D。表面参考只能声明为全局静态变量，且不能作为参数传递给函数。

在使用表面参考读写CUDA 数组前，必须使用cudaBindSurfaceToArray绑定到CUDA数组。

下面的例程绑定表面参考到CUDA数组cuArray。

- 使用低级API:

```
surface<void, cudaSurfaceType2D> surfRef;
surfaceReference* surfRefPtr;
cudaGetSurfaceReference(&surfRefPtr, "surfRef");
cudaChannelFormatDesc channelDesc;
cudaGetChannelDesc(&channelDesc, cuArray);
cudaBindSurfaceToArray(surfRef, cuArray, &channelDesc);
```

- 使用高级API:

```
surface<void, cudaSurfaceType2D> surfRef;
cudaBindSurfaceToArray(surfRef, cuArray);
```

使用表面函数读写的CUDA时的类型必须匹配并且通过类型必须匹配的表面参考；否则读写CUDA数组的结果未定义。

不像纹理存储器，表面存储器使用字节寻址。这意味着通过纹理函数访问纹理元素的x坐标需要乘以元素的字节数以通过表面函数访问同

一个元素。例如，绑定到纹理参考texRef的一个一维浮点CUDA数组，其纹理坐标x处的元素，和一个表面参考surfRef，通过texRef使用tex1d(texRef, x)访问，对应的通过surfRef以surf1Dread(surfRef, 4\*x)访问。同样地，绑定到纹理参考texRef的一个二维浮点CUDA数组，其纹理坐标x, y处的元素，和一个表面参考surfRef，通过texRef使用tex2d(texRef, x, y)访问，对应的通过surfRef以surf2Dread(surfRef, 4\*x, y)访问（y坐标的字节偏移在底层通过CUDA数组的行距计算）。

下面的例程做了一些简单转换：

```
// 2D surfaces
surface<void, 2> inputSurfRef;
surface<void, 2> outputSurfRef;

// Simple copy kernel
__global__ void copyKernel(int width, int height)
{
    // Calculate surface coordinates
    unsigned int x = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned int y = blockIdx.y * blockDim.y + threadIdx.y;
    if (x < width && y < height) {
        uchar4 data;
        // Read from input surface
        surf2Dread(&data, inputSurfRef, x * 4, y);
        // Write to output surface
        surf2Dwrite(data, outputSurfRef, x * 4, y);
    }
}

// Host code
int main()
{
    // Allocate CUDA arrays in device memory
```



```
cudaChannelFormatDesc channelDesc =
    cudaCreateChannelDesc(8, 8, 8, 8,
                          cudaChannelFormatKindUnsigned);

cudaArray* cuInputArray;
cudaMallocArray(&cuInputArray, &channelDesc, width, height,
               cudaArraySurfaceLoadStore);

cudaArray* cuOutputArray;
cudaMallocArray(&cuOutputArray, &channelDesc, width, height,
               cudaArraySurfaceLoadStore);

// Copy to device memory some data located at address h_data
// in host memory
cudaMemcpyToArray(cuInputArray, 0, 0, h_data, size,
                  cudaMemcpyHostToDevice);

// Bind the arrays to the surface references
cudaBindSurfaceToArray(inputSurfRef, cuInputArray);
cudaBindSurfaceToArray(outputSurfRef, cuOutputArray);

// Invoke kernel
dim3 dimBlock(16, 16);
dim3 dimGrid((width + dimBlock.x - 1) / dimBlock.x,
             (height + dimBlock.y - 1) / dimBlock.y);
copyKernel<<<dimGrid, dimBlock>>>(width, height);

// Free device memory
cudaFreeArray(cuInputArray);
cudaFreeArray(cuOutputArray);

return 0;
}
```

---

### 立方位图表面

立方位图表面像一维层次表面，通过surfCubemapread()和surfCubemapwrite()访问。即，使用一个整型索引确定一个面，以两个浮点纹理坐标在对应该面的纹理层内寻址一个元素。面的顺序如3.2.8.1所示。

### 层次立方位图表面

层次立方位图表面像二维层次表面，使用surfCubemapLayeredread()和surfCubemapLayeredwrite()访问。即，使用一个整型索引确定一个面，以两个浮点纹理坐标在对应该面的纹理层内寻址一个元素。面的顺序如3.2.8.1所示，所以索引 $(2*6+3)$ 访问第三个立方位图的第四个面。

### 3.2.8.3 CUDA 数组

CUDA数组是为纹理获取优化的不透明的存储器层次。它们可以是一维的，二维的或三维的，也可由多个元素组成，每个元素可有1，2或4个组件，这些组件可能是有符号或无符号8，16或32位整形，16位浮点，或32位浮点。CUDA数组只能在内核中通过3.2.8.1描述的纹理获取读取或3.2.8.2描述的表面读写访问。

### 3.2.8.4 读写一致性

纹理和表面存储器是有缓存的（参见3.2.2），且在同一个内核调用，缓存并不和全局存储器写和表面存储器写保持一致，因此任何纹理获取或表面读一个在同一内核中被全局存储器写或者表面写过的地址，其结果是不确定的。换言之，一个线程能安全的读一些纹理或表面存储器位置当且仅当这个位置已经被前一个内核调用或存储器拷贝更新过，但是并非被同一内核的当前线程或其它线程更新过。

### 3.2.9 图形学互操作性

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

资源必须先要在CUDA中注册，才能被3.2.9.1和3.2.9.2提到的函数映射。这些函数返回一个指向cudaGraphicsResource类型结构体的CUDA图形资源。资源注册是潜在高消耗的，因此通常每个资源只注册一次。可以使用cudaGraphicsUnregisterResource()来注销CUDA图形资源。

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

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

通过OpenGL或Direct3D访问被映射到CUDA的OpenGL或Direct3D的资源，其结果未定义。3.2.9.1和3.2.9.2给出了每种图形API的特性和一些代码例子。3.2.9.3给出了系统在SLI模式下的特性。

### 3.2.9.1 OpenGL互操作性

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

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

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

纹理或渲染缓存对象使用cudaGraphicsGLRegisterImage()注册，在CUDA中，它们表现为CUDA数组，内核通过将其绑定到纹理参考或表面参考以读取。如果该资源使用使用cudaGraphicsRegisterFlagsSurfaceLoadStore标签注册，也可通过表面参考写，也可通过cudaMemcpy2D()调用读写。cudaGraphicsGLRegisterImage()支持有1、2、4个组件和使用内置的float类型（例如，GL\_RGBA\_FLOAT32）、归一化整数（例如GL\_RGBA8，GL\_INTENSITY16）和非归一化整数（例如GL\_RGBA8UI）（请注意，由于GL\_RGBA8UI是OpenGL3.0纹理格式，只能被着色器写，不能被固定功能的流水线写）。

资源共享的OpenGL上下文必须是调用OpenGL互操作API的主机线程的当前上下文。

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

```
GLuint positionsVBO;
struct cudaGraphicsResource* positionsVBO_CUDA;

int main()
{
    // Initialize OpenGL and GLUT for device 0
    // and make the OpenGL context current
    ...
    glutDisplayFunc(display);

    // Explicitly set device 0
    cudaGLSetGLDevice(0);

    // Create buffer object and register it with CUDA
    glGenBuffers(1, positionsVBO);
    glBindBuffer(GL_ARRAY_BUFFER, &positionsVBO);
    unsigned int size = width * height * 4 * sizeof(float);
    glBufferData(GL_ARRAY_BUFFER, size, 0, GL_DYNAMIC_DRAW);
    glBindBuffer(GL_ARRAY_BUFFER, 0);
    cudaGraphicsGLRegisterBuffer(&positionsVBO_CUDA,
                                positionsVBO,
                                cudaGraphicsMapFlagsWriteDiscard);

    // Launch rendering loop
    glutMainLoop();

    ...
}
```

```
}

void display()
{
    // Map buffer object for writing from CUDA
    float4* positions;
    cudaGraphicsMapResources(1, &positionsVBO_CUDA, 0);
    size_t num_bytes;
    cudaGraphicsResourceGetMappedPointer((void*)&positions,
                                         &num_bytes,
                                         positionsVBO_CUDA));

    // Execute kernel
    dim3 dimBlock(16, 16, 1);
    dim3 dimGrid(width / dimBlock.x, height / dimBlock.y, 1);
    createVertices<<<dimGrid, dimBlock>>>(positions, time,
                                         width, height);

    // Unmap buffer object
    cudaGraphicsUnmapResources(1, &positionsVBO_CUDA, 0);

    // Render from buffer object
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glBindBuffer(GL_ARRAY_BUFFER, positionsVBO);
    glVertexPointer(4, GL_FLOAT, 0, 0);
    glEnableClientState(GL_VERTEX_ARRAY);
    glDrawArrays(GL_POINTS, 0, width * height);
    glDisableClientState(GL_VERTEX_ARRAY);

    // Swap buffers
    glutSwapBuffers();
    glutPostRedisplay();
}
```

```
}

void deleteVBO()
{
    cudaGraphicsUnregisterResource(positionsVBO_CUDA);
    glDeleteBuffers(1, &positionsVBO);
}

__global__ void createVertices(float4* positions, float time,
                               unsigned int width, unsigned int
                               height)
{
    unsigned int x = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned int y = blockIdx.y * blockDim.y + threadIdx.y;

    // Calculate uv coordinates
    float u = x / (float)width;
    float v = y / (float)height;
    u = u * 2.0f - 1.0f;
    v = v * 2.0f - 1.0f;

    // calculate simple sine wave pattern
    float freq = 4.0f;
    float w = sinf(u * freq + time)
        * cosf(v * freq + time) * 0.5f;

    // Write positions
    positions[y * width + x] = make_float4(u, w, v, 1.0f);
}
```

在Windows系统上和对于Quadro显卡，可以用cudaWGLGetDevice()检索关联到wglEnumGpusNV()返回的句柄的CUDA设备。Quadro显卡与OpenGL的

互操作性能比GeForce和Tesla要好。在一个多GPU的系统中，在Quadro GPU上运行OpenGL渲染，在其它的GPU进行CUDA计算。

### 3.2.9.2 Direct3D互操作性

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

一个CUDA上下文每次只能和一个Direct3D设备互操作，且CUDA上下文和Direct3D设备必须在同一个GPU上创建，另外当创建设备时要注意下列情况：Direct3D9设备必须在创建时将DeviceType设置成D3DDEVTYPE\_HAL且将BehaviorFlags设置成D3DCREATE\_HARDWARE\_VERTEXPROCESSING，

Direct3D 10和Direct3D 11 设备创建时必须将DriverType设置成D3D\_DRIVER\_TYPE\_HARDWARE。

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

存在一组调用以创建和Direct3D设备互操作的CUDA上下文，这些Direct3D设备使用工作在AFR（可替代帧渲染）模式下的NVIDIA速力（SLI）：cudaD3D[9|10|11]GetDevices()。cudaD3D[9|10|11]GetDevices()调用可以获得一个CUDA设备句柄列表，这些句柄可以作为最后一个参数（可选的）传给cudaD3D[9|10|11]SetDirect3DDevice()。

应用有下面两种方法可以选择：创建多个CPU线程，每个使用一个不同的CUDA上下文；或者单个CPU线程使用多个CUDA上下文。如果为每个GPU使用一个独立的CPU线程，每个CUDA上下文由每个CPU线程调用CUDA运行时创建。cudaD3D[9|10|11]SetDirect3DDevice()使用

cudaD3D[9|10|11]GetDevices()返回的CUDA设备句柄之一。

如果使用单个CPU线程，为了和使用NVIDIA速力的Direct3D设备互操作，不得不使用CUDA 驱动API函数创建CUDA上下文。应用依靠CUDA驱动API和运行时API的互操作性，这种互操作允许调用cuCtxPushCurrent()和cuCtxPopCurrent()以在既定时间改变活跃CUDA上下文。

可以被映射到CUDA地址空间的Direct3D资源有Direct3D缓冲区，纹理和

表面。可以使用cudaGraphicsD3D9RegisterResource(), cudaGraphicsD3D10RegisterResource()和cudaGraphicsD3D11RegisterResource()注册这些资源。

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

Direct3D 9版本

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

int main()
{
    // Initialize Direct3D
    D3D = Direct3DCreate9(D3D_SDK_VERSION);

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

    // Create device
```



```
...
D3D->CreateDevice(adapter, D3DDEVTYPE_HAL, hWnd,
                  D3DCREATE_HARDWARE_VERTEXPROCESSING
                  ,
                  &params, &device);

// Register device with CUDA
cudaD3D9SetDirect3DDevice(device);

// Create vertex buffer and register it with CUDA
unsigned int size = width * height * sizeof(CUSTOMVERTEX);
device->CreateVertexBuffer(size, 0, D3DFVF_CUSTOMVERTEX,
                          D3DPOOL_DEFAULT, &positionsVB, 0);
cudaGraphicsD3D9RegisterResource(&positionsVB_CUDA,
                                positionsVB,
                                cudaGraphicsRegisterFlagsNone);
cudaGraphicsResourceSetMapFlags(positionsVB_CUDA,
                                cudaGraphicsMapFlagsWriteDiscard);

// Launch rendering loop
while (...) {
    ...
    Render();
    ...
}
...
}

void Render()
{
    // Map vertex buffer for writing from CUDA
    float4* positions;
```

```

    cudaGraphicsMapResources(1, &positionsVB_CUDA, 0);
    size_t num_bytes;
    cudaGraphicsResourceGetMappedPointer((void**)&positions,
                                         &num_bytes,
                                         positionsVB_CUDA));

    // Execute kernel
    dim3 dimBlock(16, 16, 1);
    dim3 dimGrid(width / dimBlock.x, height / dimBlock.y, 1);
    createVertices<<<dimGrid, dimBlock>>>(positions, time,
                                         width, height);

    // Unmap vertex buffer
    cudaGraphicsUnmapResources(1, &positionsVB_CUDA, 0);

    // Draw and present
    ...
}

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

__global__ void createVertices(float4* positions, float time,
                              unsigned int width, unsigned int
                              height)
{
    unsigned int x = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned int y = blockIdx.y * blockDim.y + threadIdx.y;

```

```

    // Calculate uv coordinates
    float u = x / (float)width;
    float v = y / (float)height;
    u = u * 2.0f - 1.0f;
    v = v * 2.0f - 1.0f;

    // Calculate simple sine wave pattern
    float freq = 4.0f;
    float w = sinf(u * freq + time)
               * cosf(v * freq + time) * 0.5f;

    // Write positions
    positions[y * width + x] =
        make_float4(u, w, v, _int_as_float (0xff00ff00));
}

```

Direct3D 10版本:

```

ID3D10Device* device;
struct CUSTOMVERTEX {
    FLOAT x, y, z;
    DWORD color;
};
ID3D10Buffer* positionsVB;
struct cudaGraphicsResource* positionsVB_CUDA;

int main()
{
    // Get a CUDA-enabled adapter
    IDXGIFactory* factory;
    CreateDXGIFactory(_uuidof(IDXGIFactory), (void**)&factory);
    IDXGIAdapter* adapter = 0;
    for (unsigned int i = 0; !adapter; ++i) {

```

```
    if (FAILED(factory->EnumAdapters(i, &adapter))
        break;
    int dev;
    if (cudaD3D10GetDevice(&dev, adapter) == cudaSuccess)
        break;
    adapter->Release();
}
factory->Release();

// Create swap chain and device
...
D3D10CreateDeviceAndSwapChain(adapter,
                              D3D10_DRIVER_TYPE_HARDWARE,
                              0,
                              D3D10_CREATE_DEVICE_DEBUG,
                              D3D10_SDK_VERSION,
                              &swapChainDesc, &swapChain,
                              &device);

adapter->Release();

// Register device with CUDA
cudaD3D10SetDirect3DDevice(device);

// Create vertex buffer and register it with CUDA
unsigned int size = width * height * sizeof(CUSTOMVERTEX);
D3D10_BUFFER_DESC bufferDesc;
bufferDesc.Usage          = D3D10_USAGE_DEFAULT;
bufferDesc.ByteWidth      = size;
bufferDesc.BindFlags      = D3D10_BIND_VERTEX_BUFFER;
bufferDesc.CPUAccessFlags = 0;
bufferDesc.MiscFlags      = 0;
device->CreateBuffer(&bufferDesc, 0, &positionsVB);
```



```
// Unmap vertex buffer
cudaGraphicsUnmapResources(1, &positionsVB_CUDA, 0);

// Draw and present
...
}

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

__global__ void createVertices(float4* positions, float time,
                               unsigned int width, unsigned int
                               height)
{
    unsigned int x = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned int y = blockIdx.y * blockDim.y + threadIdx.y;

    // Calculate uv coordinates
    float u = x / (float)width;
    float v = y / (float)height;
    u = u * 2.0f - 1.0f;
    v = v * 2.0f - 1.0f;

    // Calculate simple sine wave pattern
    float freq = 4.0f;
    float w = sinf(u * freq + time)
        * cosf(v * freq + time) * 0.5f;
```

```
// Write positions
positions[y * width + x] =
    make_float4(u, w, v,  _int_as_float (0 xff00ff00 ));
}
```

Direct3D 11 版本:

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

int main()
{
    // Get a CUDA-enabled adapter
    IDXGIFactory* factory;
    CreateDXGIFactory(_uuidof(IDXGIFactory), (void**)&factory);
    IDXGIAdapter* adapter = 0;
    for (unsigned int i = 0; !adapter; ++i) {
        if (FAILED(factory->EnumAdapters(i, &adapter))
            break;
        int dev;
        if (cudaD3D11GetDevice(&dev, adapter) == cudaSuccess)
            break;
        adapter->Release();
    }
    factory->Release();

    // Create swap chain and device
    ...
```

[illegible]



```
// Launch rendering loop
while (...) {
    ...
    Render();
    ...
}
...
}

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

    // Execute kernel
    dim3 dimBlock(16, 16, 1);
    dim3 dimGrid(width / dimBlock.x, height / dimBlock.y, 1);
    createVertices<<<dimGrid, dimBlock>>>(positions, time,
                                         width, height);

    // Unmap vertex buffer
    cudaGraphicsUnmapResources(1, &positionsVB_CUDA, 0);

    // Draw and present
    ...
}
```

```

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

__global__ void createVertices(float4* positions, float time,
                               unsigned int width, unsigned int height)
{
    unsigned int x = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned int y = blockIdx.y * blockDim.y + threadIdx.y;

    // Calculate uv coordinates
    float u = x / (float)width;
    float v = y / (float)height;
    u = u * 2.0f - 1.0f;
    v = v * 2.0f - 1.0f;

    // Calculate simple sine wave pattern
    float freq = 4.0f;
    float w = sinf(u * freq + time)
        * cosf(v * freq + time) * 0.5f;

    // Write positions
    positions[y * width + x] =
        make_float4(u, w, v, __int_as_float(0xff00ff00));
}

```

### 3.2.9.3 SLI（速力）互操作性

在一个多GPU的系统中，所有的支持CUDA的GPU都可以被驱动和运行时作为独立的设备访问。当系统在SLI模式时，有许多特殊的考虑，如下所描

述。

首先，所有在一个CUDA设备上的存储器分配将消耗其它GPU的存储器，这是Direct3D设备的SLI配置的一部分。因为这点，存储器分配将会比我们希望的要早失败。

其次，应用不得不建立多个CUDA上下文，每个在SLI配置下的GPU一个，且有一个不同的GPU被Direct3D或OpenGL设备用于在每一帧时渲染。应用能够使用`cudaD3D[9|10|11]GetDevices()`或`cudaGLGetDevices()`系列调用以确定当前和下一帧进行渲染的GPU。有了这信息，应用将映射Direct3D或OpenGL资源到关联到`cudaD3D[9|10|11]GetDevices()`或`cudaGLGetDevices()`返回的设备的CUDA上下文，此时`deviceList`参数设置为

`CU_D3D10_DEVICE_LIST_CURRENT_FRAME`。

参见3.2.9.1和3.2.9.2以了解CUDA是分别如何与Direct3D或OpenGL互操作。

### 3.3 版本和兼容性

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

驱动API的版本在驱动头文件中定义为`CUDA_VERSION`。开发人员可用其检查其应用是否要比当前版本更新的驱动。这非常重要，因为驱动API是向后兼容的，这意味着特定版本编译的应用、插件和库（包括C运行时）能够在以后发布的驱动上工作，如图3.3所示。但是驱动API不是向前兼容的，这意味着特定版本编译的应用、插件和库（包括C运行时）不能够在以前发布的驱动上工作。

特别要注意混合版本是不支持的；尤其：

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

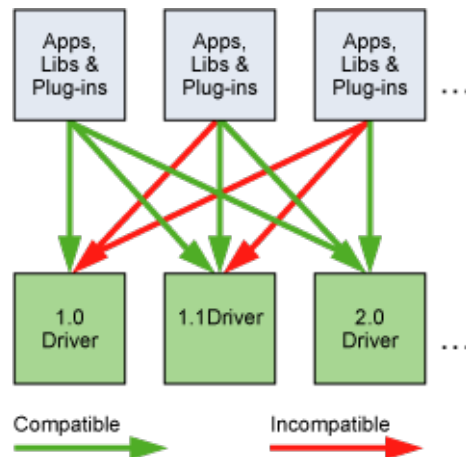


图 3.3: 驱动API是向后兼容，而非向前兼容

### 3.4 计算模式

在Linux和Windows Server 2008及更高版本上运行的Tesla解决方案，可以使用NVIDIA的系统管理接口（nvidia-smi）设置系统上任何设备的计算模式的为下面的三种之一，nvidia-smi是一个作为Linux驱动一部分发布的工具。

- 默认模式：多个主机线程可同时使用设备（使用运行时调用`cudaSetDevice()`，或使用驱动API时将关联到设备的上下文作为当前上下文）。
- 互斥进程计算模式：在系统的所有进程之间，一个设备上只能建立一个CUDA上下文，该上下文可以成为建立该上下文的进程中的许多线程的当前上下文。
- 互斥进程和线程计算模式：在系统的所有进程之间，一个设备只能建立一个CUDA上下文，而且一次只能成为一个线程的上下文。
- 禁止模式：不允许任何主机线程使用设备。

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

应用可检查`computeMode`属性（参见3.2.6）以查询设备的计算模式。

### 3.5 模式切换

GPU 将部分DRAM 存储器用于所谓的主表面 (primary surface)，它用于刷新显示器，用户查看显示器的输出。当用户通过更改显示器的分辨率或位深度（使用NVIDIA 的控制面板或Windows的显示控制面板）初始化模式切换时，主表面所需的存储器数量随之改变。例如，如果用户将显示器的分辨率从1280\*1024\*32 位更改为1600\*1200\*32 位，系统必须为主表面分配7.68 MB 的存储器，而不是5.24 MB（使用防锯齿设置运行的全屏图形应用可能需要为主表面分配更多存储器用于显示）。在Windows 上，其他事件也可能会启动显示模式切换，包括启动全屏DirectX 应用程序、按Alt+Tab 键从全屏DirectX 应用中切换出来或者按Ctrl+Alt+Del 键锁定计算机。

如果模式切换增加了主表面所需的存储器数量，系统可能就必须挪用分配给CUDA 应用的存储器，因此模式切换可能导致任何调用CUDA运行时的应用崩溃并返回无效上下文错误。

### 3.6 Windows上的Tesla计算集群模式

使用NVIDIA系统管理接口 (nvidia-smi)，对于计算能力2.0及以上的Tesla和Quadro系列设备，Windows设备驱动能够进入TCC（Tesla计算集群）模式。

这种模式有下列主要优点：

- 它使得在非NVIDIA集成显卡的集群节点上使用GPU成为可能；
- 可以通过远程桌面使用GPU，直接或者通过集群管理系统使用GPU都依赖远程桌面；
- 它使得作为Windows服务运行的应用能够使用GPU（即在会话0）。

然而TCC模式不支持任何图形功能。



## 第四章 硬件实现

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

流多处理器设计为能同时并发执行上百线程。为了管理如此多的线程，多处理器采用了一种称为SIMT（单指令，多线程）的独一无二的架构，这在4.1节描述。指令流水线化以利用单线程内的指令级并行，流多处理器也使用如4.2节所描述的硬件同时多线程利用线程级并行。与CPU核心不同地，指令顺序发射，而且没有分支预测和猜测执行。

4.1节和4.2节描述了所有设备都相同的流多处理器架构特色。F.3、F.4和F.5分别提供了计算能力1.x、2.x和3.x的特性。

### 4.1 SIMT 架构

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

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

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

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

如果一个束执行非原子指令为多个线程写入全局存储器或共享存储器的同一位置，串行写入该位置变量的数目依赖于设备的计算能力（参见F.3、F.4和F.5）且那个线程最后写入无法确定。

如果束执行的原子指令（参见B.11）为束内多个线程读、修改和写入全局存储器的同一位置，每次读、修改和写入都会串行执行，但是他们执行的顺序没有定义。

## 4.2 硬件多线程

流多处理器处理的每个束的执行上下文（程序计数器，寄存器等）在束的生存期内被维护在片上。从一个执行上下文切换到另一个执行上下文没有消耗，而且在每个指令发射间，束调度器选择所有线程已准备好执行的束（活动束）并且向这些线程发射下个指令。

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

多处理器能够为一个内核常驻和同时处理的块和束的数量依赖于内核使用的寄存器和共享存储器数量和多处理器拥有的寄存器和共享存储器总量。同时每个多处理器有一个最大的常驻块数量和常驻束数量。这些限制包括多处理器上可用寄存器和共享存储器的数量是设备计算能力的函数，且其值在F中给出。在计算能力1.x的设备上，如果多处理器没有足够的可用寄存器或共享存储器以处理至少一个块，内核将会发射失败。

块内总束数量如下：

$$\text{ceil}(\frac{T}{W_{size}}, 1)$$



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

为线程块分配的总寄存器数量、总共享存储器数量可通过CUDA占用率计算器 (CUDA Occupancy Calculator) 得到, CUDA占用率计算器可在CUDA软件开发包中找到。



## 第五章 性能指南

### 5.1 总体性能优化策略

性能优化始终围绕三个基本策略：

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

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

### 5.2 最大化利用率

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

#### 5.2.1 应用层次

在高层次上，应用应当通过使用异步函数调用和[3.2.5.5](#)描述的流来最大化主机，设备和连接主机和设备的总线的并行执行。应当把每个处理器最擅长的任务分配给它：串行工作分配给主机；并行工作分配给设备。

对于并行工作，在算法中，由于某些线程为了与其它线程共享数据而同步导致并行性中断的点，有两种情况：或那些线程属于同一个块，这种情况下，它们只要使用`_syncthreads()`和在同一个内核调用中使用共享存储器共享数据，或属于不同的块，这种情况下，它们必须使用两个不同的内核调用以通过全局存储器共享数据，一个内核将数据写入全局存储器，另一个从全局存储器中

读。第二种情况优化得比较差，因为它增加了额外的内核调用消耗和全局存储器通信量。为了减弱这种影响，应当以一种将线程间通信局限在一个块的方式将算法映射到CUDA编程模型。

### 5.2.2 设备层次

在低层次，应用应当最大化设备内多处理器间的并行执行。

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

对于计算能力2.x的设备，多个内核可在设备上并发执行，因此可以使用流来发射多个内核并发执行（参见3.2.5.3）来获得最大利用率。（译者注：此时要注意这些内核之间要没有依赖）

### 5.2.3 多处理器层次

在一个更低的层次，应用应当最大化多处理器内部的各种功能单元的并行执行。

如4.2节所描述的，GPU多处理器依赖线程级并行来最大化其功能单元的利用。利用率和常驻束数量直接相关。在每次指令发射时，束调度器选择一个已准备好执行的束并将下个指令发射给束内的活动线程。束准备执行下一条指令花费的时钟周期数称为延迟，如果在延迟期间，每个时钟周期内，束调度器有一些指令可为某些束发射就可获得完全的利用，或换句话说，每个束的延迟可被其它束完全隐藏。要多少条指令才能隐藏L个时钟的延迟依赖各个指令的吞吐量（参见5.4.1了解不同指令的吞吐量）。对于所有指令的最大吞吐量。

- 对于计算能力1.x要 $L/4$ （最近取整）条指令，因为多处理器每四个时钟周期为每束发射一条该指令，如F.3所述。
- 对于计算能力2.0要L条指令，因为多处理器每两个时钟周期为每两个束每次每束发射一条指令，如F.4所述。
- 对于计算能力2.1要2L条指令，因为多处理器每两个时钟周期为两个束每束每次发射两条指令，如F.4所述。
- 对于计算能力3.x需要8L条指令，因为多处理器每时钟周期为四个束每个束发射两条指令，如F.5所述。

对于计算能力2.0的设备，每个周期为两个不同的束发射两条指令，每束一条。对于计算能力2.1的设备，每个周期为两个不同的束发射两对指令，每束发射一对。

对于计算能力3.x的设备，每个周期为四个束发射四对指令，每束一对。

束没有准备好执行它下条指令的最常见原因是指令的输入操作数没有准备好。

如果所有输入操作数是寄存器，延迟是因为寄存器依赖，也就是说，一些输入操作数是由一些还没有完成的在前面的指令写的。在紧接着的寄存器依赖的情况下（一些输入操作数是由前面的指令写的），延迟等于前面指令的执行时间且在此期间束调度器必须为不同的束调度指令。指令不同，执行时间也会变化，但是典型地大约22个时钟周期，这需要计算能力1.x设备上的6个束，计算能力2.x设备上的22个束，计算能力3.x设备上的44个束来隐藏，或者需要更多的束（因为我们假设指令是按照最大吞吐量运算的）。对于计算能力2.1及以上的设备这也要求具有足够的指令级并行，使得调度器能够为每个束发射一对指令。

如果某些输入操作数在片下存储器中，延迟更高：对于计算能力1.x和2.x大约400到800时钟周期；对于计算能力3.x大约200到400个周期。在如此高的延迟期保持束调度器繁忙需要的束数依赖于内核代码和指令级并行度；一般，如果没有片下存储器的操作数的指令（即大多数时候指算术指令）数与片下存储器的操作的指令比率小的话（这个比率经常称为程序的运算密度），要求更多束。比如，假设比率是30，在1.x和2.x的设备上片下存储器延迟为600个周期，在3.x的设备上，片下存储器的延迟为300个周期。为了隐藏延迟，对于计算能力1.x的设备大约要5个束，对于计算能力2.x的设备大约20个束，对于计算能力3.x的设备大约40个束。

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

对于给定的内核调用，一个多处理器上的块和束数目依赖调用时的执行配置（参见B.18），多处理器的存储器资源和如4.2描述的内核资源要求。为了帮助程序员基于寄存器和共享存储器要求选择合适的线程块尺寸，CUDA软件开

发工具包提供了一个称为CUDA占有率计算器的电子表格，在这里，占有率定义为常驻块数目和最大常驻块数目之比（F为各种计算能力给出了数据）。

寄存器、本地存储器、共享存储器和常量存储器的使用可在编译时使用—ptxas-options=-v选项报告。

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

内核使用的寄存器数目对常驻束数目有显著影响。例如，计算能力1.2的设备，如果内核使用了16个寄存器且每个块有512个线程且要求非常少的共享存储器，这样多处理器可常驻两个块(即32束)，因为它们要求 $2 \times 512 \times 16$ 个寄存器，这匹配多处理器的可用寄存器数目。但是如果只要内核多使用一个寄存器，就只有一个块（16束）能够常驻多处理器上了，因为两个块要求 $2 \times 512 \times 17$ 个寄存器，这大于多处理器的可用寄存器数目。因此编译器试图在保证寄存器溢出的前提下最小化寄存器使用和指令数目。可以使用-maxrregcount编译选项或如B.19描述的发射绑定控制寄存器使用。

每个双精度变量（在支持本地双精度的设备上，即计算能力1.3或更高的设备）和每个long long变量使用两个寄存器。但是计算能力1.2或以上的寄存器总量是低计算能力的至少两倍。

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

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

### 5.3 最大化存储器吞吐量

最大化应用的总体存储器吞吐量的第一步是最小化低带宽的数据传输。

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

也意味着通过最大化片上存储器使用以最小化全局存储器和设备间的数据传输：如共享存储器和缓存（如计算能力2.x的设备上的L1/L2缓存，所有设备

上的纹理缓存和常量缓存)。

共享存储器等价于用户管理的缓存：应用显式的分配和访问它。如3.2节所描述的，一个常用的编程模式是将来自设备存储器的数据存储在共享存储器，换句话说，让块内的每个线程：

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

对于一些应用（如对于那些访问全局存储器是数据依赖的），传统的硬件管理的缓存更适用于发掘数据局部性。如F.4所提到的，对于计算能力2.x的设备，同一片上存储器用于L1和共享存储器，至于在每个内核调用时为L1和共享存储器各分配多少则可以配置。

内核的存储器访问吞吐量依赖于每种类型存储器的访问模式，其性能可相差一个数量级。最大化存储器吞吐量的下一步是依据5.3.2描述的最优化存储器访问模式重新组织存储器访问。对于全局存储器来说，这种优化非常重要，因为全局存储器的带宽比较低，所以非优化的全局存储器访问对性能有更大的影响。

### 5.3.1 主机和设备的数据传输

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

另外由于每次传输的消耗，将多次小的传输组合成一次大的传输会比单独传输更好。

对于有前端总线的系统，使用3.2.4描述的分页锁定主机存储器进行主机和设备间的数据传输会获得更好的性能。



另外，在使用被映射主机存储器时（参见3.2.4.3），没有必要分配任何设备存储器和显式在主机和设备间传输数据。数据传输会在每次内核访问映射存储器时隐式进行。为了最大化性能，这些存储器访问必须像全局存储器一样满足合并访问要求（参见5.3.2）。假定被映射主机存储器只被读写一次，使用被映射主机存储器性能会提高。

在集成系统上，设备存储器和主机存储器在物理上是相同的，任何设备和主机间的数据传输都是多余的，此时应当使用被映射主机存储器。应用可以检查integrated设备属性(参见3.2.6)来查询设备是不是集成的，如果是，返回1。

### 5.3.2 设备存储器访问

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

#### 5.3.2.1 全局存储器

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

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

要多少次通信和对吞吐量的最终影响都随计算能力变化。对于计算能力1.0和1.1的设备，线程间地址的分布满足合并访问的要求是非常严格的。更高计算能力的设备上就宽松了许多。对于计算能力2.x的设备，存储器通信是缓存的，挖掘了数据局部性以减少合并访问对性能的影响。F.3、F.4和F.5给出了全局存储器访问如何被各种计算能力的设备处理的细节。

为了最大化全局存储器吞吐量，最大化合并访问非常重要：

- 遵从F.3、F.4和F.5的最优访问模式，



- 使用满足尺寸和对齐要求的数据类型，详见[5.3.2.1](#)节
- 在某些情况下，填充数据，如[5.3.2.1](#)描述的访问二维数组。

### 尺寸和对齐要求

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

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

像float2或float4这样的内置数据类型自动满足对齐要求。

对于结构体，尺寸和对齐要求可用对齐修饰符\_\_align\_\_(8)或\_\_align\_\_(16)让编译器保证。如

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

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

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

## 二维数组

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

$$\text{BaseAddress} + \text{width} * \text{ty} + \text{tx}$$

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

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

### 5.3.2.2 本地存储器

本地存储器访问仅对某些自动变量发生，如B.2所述。编译器可能放入本地存储器的自动变量是：

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

检查PTX汇编代码（编译时使用`-ptx`或`-keep`选项获得）可以分辨，如果一个变量在编译的第一阶段被放入本地存储器，会使用`.local`助记符声明，使

用ld.local和st.local访问。即使此时没有放入本地存储器中，在随后的编译阶段，如果觉得对于目标架构它消耗了太多的寄存器，也会被放入本地存储器中：使用cuobjdump检查cubin对象会分辨是不是这种情况。另外编译时使用—ptxas-options=-v选项，编译器会报告内核总的本地存储器用量（lmem）。注意一些数学函数有的实现路径也可能访问了本地存储器。

本地存储器存在于设备存储器空间，所有本地存储器访问延迟像全局存储器一样高，带宽和全局存储器一样低，且服从于5.3.2描述的存储器合并访问要求。本地存储器的组织使得连续的线程ID访问连续的32位字。只要束内所有线程访问同一相对地址（如数组变量的同一索引，结构体变量的同一成员）访问就可完全合并。

在计算能力2.x的设备上，本地存储器访问总是以和全局存储器访问（参见F.4）同样的方式被缓存在L1和L2。

### 5.3.2.3 共享存储器

由于共享存储器位于芯片上，因而共享存储器空间比本地和全局存储器空间的速度都要快得多。

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

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

为了获得最大化的性能，理解存储器地址如何映射到存储体以调度存储器请求，以最小化存储体冲突就很重要。这些在F.3、F.4和F.5节为计算能力1.x、计算能力2.x和计算能力3.x分别描述。

### 5.3.2.4 常量存储器

常量存储器空间存在于设备存储器并被缓存到常理缓存中，参见F.3和F.4。

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

一个请求然后分成多个请求，使得它们访问的地址不同（译者注：也就是说各个请求只访问一个地址），吞吐量减少到一个等于独立请求的数量的因子。

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

### 5.3.2.5 纹理和表面存储器

纹理和表面存储器空间存在于设备存储器中，并被缓存到纹理缓存，因此纹理获取或表面读仅需在缓存丢失时读取一次设备存储器，否则只需花费一次纹理缓存读取。纹理缓存已为二维空间局部性而优化，因此同一个束的线程读取二维相邻纹理或表面地址的将实现最高性能。此外，它设计用于以固定的延迟执行流获取，一次缓存命中将减少DRAM 带宽要求，而非延迟。

与从全局存储器或常量存储器读取设备存储器的方法相比，通过纹理或表面获取读取设备存储器可能是一种更有优势的替代方法。

- 如果存储器读取不符合全局或常量存储器读取模式，纹理存储器会得到好性能，如果在纹理获取的时候存在局部性，能得到更高的带宽；
- 地址计算在内核外由特定单元处理；
- 在一个操作中包装的数据可能在一次操作中广播到独立的变量；
- 8位和16位整形输入数据可能转化为32位浮点值，范围为[0.0, 1.0]或[-1.0, 1.0]（参见3.2.8）。

## 5.4 最大化指令吞吐量

为了最大化指令吞吐量，应用应当：

- 最小化吞吐量较低的指令的使用；包括为速度牺牲精度，如果不影响最终结果的话，如使用内置函数取代常规函数（内置函数列在C.2），单精度取代双精度，或将非规格化数刷为0。

- 最小化由流控指令引起的束内分支（参见5.4.2）；
- 减少指令数目，例如，如B.6所描述的只要可能就优化同步点或如B.2.4所描述的使用受限的指针。

本节，吞吐量以每时钟周期每个多处理器操作数目给定。对于尺寸为32的束，一条指令产生32个操作。因此，如果T是每个时钟周期操作数目，指令吞吐量就是每32/T时钟周期一个指令。

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

### 5.4.1 算术指令

5.1给出了各种计算能力的设备其硬件本地支持的算术指令吞吐量。

其它指令和函数基于本地指令实现。对于计算能力1.x和2.x的设备其实现可能不同，而不同的编译器版本编译后的本地指令数量也可能波动。对于复杂函数，可能有依赖于输入的多条代码路径。可以使用cuobjdump检查cubin对象中的一个特定实现。

一些函数的实现在CUDA头文件中(math.functions.h, device.functions.h等)可以找到。

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

#### 单精度浮点加和内置乘

`_fadd_r[d,u]`, `_fmul_r[d,u]`, and `_fmaf_r[n,z,d,u]`（参见5.4.1），对于计算能力1.x的设备，编译成几十个指令，但对于计算能力2.x及以上的设备映射成单条本地指令

#### 单精度浮点除

`_fdividef(x, y)`（参见5.4.1）提供了比除法操作符快的单精度浮点除法

#### 单精度浮点倒数平方根

表 5.1: 原生算术指令吞吐量/每时钟每流多处理器操作数目

	计算能力					
	1.0/1.1/1.2	1.3	2.0	2.1	3.0	3.5
32-bit floating-point add, multiply, multiplyadd	8	8	32	48	192	192
64-bit floating-point add, multiply, multiplyadd	1	1	16(*)	4	8	64
32-bit integer add	10	10	32	48	160	160
32-bit integer compare	10	10	32	48	160	160
32-bit integer shift	8	8	16	16	32	64
Logical operations	8	8	32	48	160	160
32-bit integer multiply, multiplyadd, sum of absolute difference	多条指令	多条指令	16	16	32	32
24-bit integer multiply(__mul24)	8	8	多条指令	多条指令	多条指令	多条指令
32-bit floating-point reciprocal, reciprocal square root, base-2 logarithm(__log2f), base 2 exponential(exp2f), sine(__sinf), cosine(__cosf)	2	2	4	8	32	32
Type conversions from 8-bit and 16-bit integer to 32-bit types	8	8	16	16	128	128
Type conversions from and to 64-bit types	多条指令	1	16(*)	4	8	32
All other type conversions	8	8	16	16	32	32

为了保存IEEE-754的语义，只有当倒数和平方根都是近似的时候（即-prec-div=false和-prec-sqrt=false），编译器才将1.0/sqrtf()优化为rsqrtf()。所以应当在需要的时候直接调用rsqrtf()。

### 单精度浮点平方根

单精度浮点平方根实现为倒数平方根的倒数而不是倒数平方根的乘法，因此对和无穷能得到正确的结果。

### 正弦和余弦

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

更精确地，参数归约代码包含两个称为快速路径和慢路径的代码。

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

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

因为慢路径相比快路径要更多的寄存器，故将中间结果放到本地存储器中试图减少寄存器的用量，由于本地存储器的高延迟和低带宽（参见5.3.2），可能影响性能。目前，为单精度使用了28字节的本地存储器，而双精度使用了44字节。但具体的数量可能会改变。

由于慢路径计算量长和使用了本地存储器，任何三角函数的吞吐量慢路径比快路径相差一个数量级。

### 整数算术

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

在计算能力2.x及以上的设备上，32位整数乘法是本地支持的，而24位不是。\_\_u]mul24使用多条指令使用，因此不应当使用。

整数除法和模余非常昂贵：计算能力1.x的设备上要几十条指令，而计算能力2.x及以上的设备上少于二十条指令。只要可能就应当避免使用或用位操



作符取代：如果 $n$ 是2的乘方， $(i/n)$  等于  $(i \gg \log_2(n))$ ，而  $(i \% n)$  等于  $(i \& (n-1))$ ；如果 $n$ 是字面量，编译器会自动实现转换。

`__brev`、`__brev11`、`__popc`和`__popcll`，对于计算能力1.x编译成几十个指令，但是对于计算能力2.x `__brev`和`__popc`映射成一条指令，而`__brevll`和`__popcll`就几条指令。

`__clz`、`__clzll`、`__ffs`和`__ffsll`编译成的指令数在计算能力2.x及以上的设备上比计算能力1.x的要少。

### 类型转换

有时，编译器会插入类型转换指令，这增加了执行周期。下面的就是这种情况：

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

后面一种情况可以使用单精度浮点常量避免，使用后缀`f`定义，如`3.14159-2653589793f`、`1.0f`、`0.5f`。

### 5.4.2 控制流指令

任何流控制指令（`if`、`switch`、`do`、`for`、`while`）都会导致同一束的线程分支（即走向不同的执行路径），从而显著影响有效指令吞吐量。如果出现这种情况，不同的执行路径必须序列化，因而增加了为该束执行的指令总数。当完成所有不同的执行路径时，线程将重新汇聚到同一执行路径。

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

有些时候，编译器可能会展开循环或使用分支谓词来优化`if` 或`switch`语句，下面将详细说明。在这些情况下，不会有任何`warp`分支。程序员还可使用`#pragma unroll` 伪指令控制循环的展开（请参见B.20）。



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

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

### 5.4.3 同步指令

对于计算能力1.x的设备，\_\_syncthreads()是每个时钟周期8个操作，对于计算能力2.x的设备每时钟周期16个操作，对于计算能力3.x的设备，每周期128个操作。

注意：\_\_syncthreads()能通过强制某些多处理器闲着从而影响性能，细节见[5.3.2](#)。



## 附录 A 支持CUDA的GPU

[这里](#)列出了所有支持CUDA的GPU及其计算能力。

计算能力、流多处理器数、时钟频率、总显存大小和其它的一些属性可能通过运行时查询。



## 附录 B C语言扩展

### B.1 函数类型限定符

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

#### B.1.1 `__device__`

使用`__device__` 限定符声明的函数：

- 在设备上执行；
- 仅可通过设备调用。

#### B.1.2 `__global__`

使用`__global__` 限定符可将函数声明为内核。此类函数：

- 在设备上执行；
- 只能通过主机调用。

`__global__` 函数的返回类型必须为空。

对`__global__` 函数的任何调用都必须按[B.18](#)的方法指定其执行配置。

`__global__` 函数的调用是异步的，也就是说它会在设备执行完成之前返回。

#### B.1.3 `__host__`

使用`__host__` 限定符声明的函数：

- 在主机上执行；
- 仅可通过主机调用。

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

`__global__` 和`__host__` 限定符不能一起使用。

但`__host__` 限定符也可与`__device__` 限定符一起使用，此时函数将为主机和设备同时进行编译。

3.1.3引入的`__CUDA_ARCH__`宏可用于区别主机和设备间不同的代码路径。

```
__host__ __device__ func()
{
    #if __CUDA_ARCH__ == 100
        // Device code path for compute capability 1.0
    #elif __CUDA_ARCH__ == 200
        // Device code path for compute capability 2.0
    #elif __CUDA_ARCH__ == 300
        // Device code path for compute capability 3.0
    #elif !defined(__CUDA_ARCH__)
        // Host code path
    #endif
}
```

#### B.1.4 `__noinline__` 和 `__forceinline__`

在计算能力1.x的设备上，默认情况下，`__device__` 函数总是内联的；在计算能力2.x及以上的设备上，只有在编译器认为必要时才内联。

`__noinline__` 函数限定符暗示编译器尽可能不要内联该函数。函数体必须位于所调用的同一个文件内。在计算能力1.x的设备上，如果函数有指针参数或者参数列表比较长，编译器会忽视`__noinline__`限定符。在计算能力2.x的设备上，`__noinline__`永远有效。

`__forceinline__`限制符强制编译器内联。

## B.2 变量类型限定符

变量类型限定符指定变量在设备上的存储位置。

在设备代码中声明的自动变量，如果不带`__device__`、`__shared__`和`__constant__`限定符中的任何一个时通常位于寄存器中。但在某些情况下，编译器可能选择将其置于本地存储器中，这将带来性能损耗，详见[5.3.2](#)。

### B.2.1 `__device__`

`__device__` 限定符声明位于设备上的变量。

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

- 位于全局存储器空间中；
- 与应用程序具有相同的生命周期；
- 网格内的所有线程都可访问，主机也可通过运行时库访问(运行时API中的`cudaGetSymbolAddress()` / `cudaGetSymbolSize()` / `cudaMemcpyToSymbol()` / `cudaMemcpyFromSymbol()`函数和驱动API中的`cuModuleGetGlobal()`函数)。

### B.2.2 `__constant__`

`__constant__` 限定符可与`__device__` 限定符一起使用，此时`__device__`是可选的，所声明的变量具有以下特征：

- 位于常量存储器空间中；
- 与应用程序具有相同的生命周期；
- 网格内的所有线程都可访问，主机也可通过运行时库访问(运行时API中的`cudaGetSymbolAddress()` / `cudaGetSymbolSize()` / `cudaMemcpyToSymbol()` / `cudaMemcpyFromSymbol()`函数和驱动API中的`cuModuleGetGlobal()`函数)。

### B.2.3 \_\_shared\_\_

`__shared__` 限定符可与 `__device__` 限定符一起使用，此时 `__device__` 是可选的，所声明的变量具有以下特征：

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

将共享存储器中的变量声明为动态数组时，例如：

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

数组的大小将在启动时确定（参见B.18）。所有变量均以这种形式声明，在存储器中的同一地址开始，因此数组中的变量布局必须通过偏移显式管理。例如，如果一名用户希望在动态分配的共享存储器内获得与以下代码对应的内容：

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

在动态分配的共享存储器，可以通过下面的方式声明和初始化：

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

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



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

### B.2.4 \_\_restrict\_\_

nvcc通过\_\_restrict\_\_关键字支持受限的（restricted）指针。

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

下面是一个别名问题的例子，使用受限指针能够帮助编译器减少指令数：

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

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

式。类似地，编译器不能只是重排c[4]为c[0]和c[1]的计算，因为前面C[3]的写入可能改变C[4]的读入。

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

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

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

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

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

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

## B.3 内置变量类型

**B.3.1** char1、uchar1、char2、uchar2、char3、uchar3、char4、uchar4、short1、ushort1、short2、ushort2、short3、ushort3、short4、ushort4、int1、uint1、int2、uint2、int3、uint3、int4、uint4、long1、ulong1、long2、ulong2、long3、ulong3、long4、ulong4、float1、float2、float3、float4、double2

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

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

这将创建一个类型为int2的向量，值为(x, y)。

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

### B.3.2 dim3类型

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

## B.4 内置变量

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

### B.4.1 gridDim

此变量的类型为dim3（参见[B.3.2](#)），包含网格的维度。

### B.4.2 blockIdx

此变量的类型为uint3（参见[B.3.1](#)），包含网格内的块索引。

表 B.1: 内置类型的对齐要求

Type	Alignment
char1, uchar1	1
char2, uchar2	2
char3, uchar3	1
char4, uchar4	4
short1, ushort1	2
short2, ushort2	4
short3, ushort3	2
short4, ushort4	8
int1, uint1	4
int2, uint2	8
int3, uint3	4
int4, uint4	16
long1, ulong1	如果sizeof(long) == sizeof(int),4;否则8
long2, ulong2	如果sizeof(long) == sizeof(int),8;否则16
long3, ulong3	如果sizeof(long) == sizeof(int),4;否则8
long4, ulong4	16
longlong1	8
longlong2	16
float1	4
float2	8
float3	4
float4	16
double1	8

### B.4.3 blockDim

此变量的类型为dim3（参见B.3.2），包含块的维度。

### B.4.4 threadIdx

此变量的类型为uint3（参见第B.3.1），包含块内的线程索引。

### B.4.5 warpSize

此变量的类型为int，包含以线程为单位的warp 块大小。

## B.5 存储器栅栏函数

```
void __threadfence_block();
```

等待直到在此函数调用前的所有全局存储器和共享存储器访问对块内所有线程可见。

```
void __threadfence();
```

等待直到在此函数调用前的所有全局存储器和共享存储器访问对下列线程可见：

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

```
void __threadfence_system();
```

等待直到在此函数调用前的所有全局存储器和共享存储器访问对下列线程可见：

- 对于共享存储器，对块内所有线程可见，
- 对于全局存储器，对设备上的所有线程可见，
- 分页锁定主机存储器，对主机线程可见（参见3.2.4）。

`__threadfence_system()`只支持计算能力2.x的设备。

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

一个用处是当线程消费其它线程生产的数据时，就如下面的代码描述的一个内核在一次调用中计算一个N个数的数组的和。每个块先计算数组的一部分并将结果存储到全局存储器。当所有的块都完成后，最后一个块从全局存储器中读取部分和并将它们加和得到最终结果。为了决定那个块最后完成，每个块原子地递增计数器以通知计算完成并存储部分和（参见B.11了解原子函数）。最后一个块是得到计数器的值为`gridDim.x-1`的块。如果在存储部分和和递增计数器值时没有放置栅栏，计数器值可能在部分和存储之前增加了，这样最后一个块可能在部分和没有更新就开始读取部分和了。

```
__device__ unsigned int count = 0;
__shared__ bool isLastBlockDone;
__global__ void sum(const float* array, unsigned int N,
                    float * result)
{
    // Each block sums a subset of the input array
    float partialSum = calculatePartialSum(array, N);

    if (threadIdx.x == 0) {

        // Thread 0 of each block stores the partial sum
        // to global memory
        result[blockIdx.x] = partialSum;

        // Thread 0 makes sure its result is visible to
        // all other threads
        __threadfence();

        // Thread 0 of each block signals that it is done
```

```
    unsigned int value = atomicInc(&count, gridDim.x);

    // Thread 0 of each block determines if its block is
    // the last block to be done
    isLastBlockDone = (value == (gridDim.x - 1));
}

// Synchronize to make sure that each thread reads
// the correct value of isLastBlockDone
__syncthreads();

if (isLastBlockDone) {

    // The last block sums the partial sums
    // stored in result[0 .. gridDim.x-1]
    float totalSum = calculateTotalSum(result);

    if (threadIdx.x == 0) {

        // Thread 0 of last block stores total sum
        // to global memory and resets count so that
        // next kernel call works properly
        result[0] = totalSum;
        count = 0;
    }
}
}
```

## B.6 同步函数

```
void __syncthreads();
```

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

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

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

计算能力2.x及以上的设备还支持下面三种 `__syncthreads()` 的变体。

```
int __syncthreads_count(int predicate);
```

等价于包含额外特性的 `__syncthreads()`，即为块内所有线程计算 `predicate` 并返回 `predicate` 值不是0的线程数目。

```
int __syncthreads_and(int predicate);
```

等价于包含额外特性的 `__syncthreads()`，即为块内所有线程计算 `predicate` 并且仅当所有线程的 `predicate` 值不是0才返回非零值。

```
int __syncthreads_or(int predicate);
```

等价于包含额外特性的 `__syncthreads()`，即为块内所有线程计算 `predicate` 并且只要有一个线程的 `predicate` 值不是零，返回的就是非零值。

## B.7 数学函数

参考手册中列出了在设备代码中支持的所有标准C/C++库数学函数及内置函数。C提供了一些相关函数的精度信息。

## B.8 纹理函数

### B.8.1 纹理对象函数

#### B.8.1.1 `tex1Dfetch()`



```
template<class T>
T tex1Dfetch(cudaTextureObject_t texObj, int x);
```

使用整形坐标`x`获取一维纹理对象`texObj`指定的线性存储器。`tex1Dfetch()`只支持非归一化纹理坐标，因此只支持边界和钳位寻址模式。也不进行纹理滤波。对于整形，这些函数可选的将整形提升到单精度浮点型。

#### B.8.1.2 tex1D()

```
template<class T>
T tex1D(cudaTextureObject texObj, float x);
```

使用浮点纹理坐标`x`获取一维纹理对象`texObj`指定的CUDA数组。

#### B.8.1.3 tex2D()

```
template<class DataType, enum cudaTextureReadMode readMode>
Type tex2D(texture<DataType, cudaTextureType2D, readMode> texRef,
           float x, float y);
```

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

#### B.8.1.4 tex3D()

```
template<class DataType, enum cudaTextureReadMode readMode>
Type tex3D(texture<DataType, cudaTextureType3D, readMode> texRef,
           float x, float y, float z);
```

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

#### B.8.1.5 tex1DLayered()

```
template<class DataType, enum cudaTextureReadMode readMode>
Type tex1DLayered(
    texture<DataType, cudaTextureType1DLayered, readMode> texRef,
    float x, int layer);
```

使用纹理坐标x和索引layer获取绑定到一维层次纹理引用texRef的CUDA数组。

#### B.8.1.6 tex2DLayered()

```
template<class DataType, enum cudaTextureReadMode readMode>
Type tex2DLayered(
    texture<DataType, cudaTextureType2DLayered, readMode> texRef,
    float x, float y, int layer);
```

使用纹理坐标x、y和索引layer获取绑定到二维层次纹理引用texRef的CUDA数组

#### B.8.1.7 texCubemap()

```
template<class DataType, enum cudaTextureReadMode readMode>
Type texCubemap(
    texture<DataType, cudaTextureTypeCubemap, readMode> texRef,
    float x, float y, float z);
```

使用纹理坐标x,y和z获取绑定到立方体图纹理引用的CUDA数组。

#### B.8.1.8 texCubemapLayered()

```
template<class DataType, enum cudaTextureReadMode readMode>
Type texCubemapLayered(
    texture<DataType, cudaTextureTypeCubemapLayered, readMode>
    texRef,
    float x, float y, float z, int layer);
```

使用三维纹理坐标x,y和z及索引layer获取绑定到层次立方图纹理引用的CUDA数组。

#### B.8.1.9 tex2Dgather()

```
template<class DataType, enum cudaTextureReadMode readMode>
Type tex2Dgather(
    texture<DataType, cudaTextureType2D, readMode> texRef,
    float x, float y, int comp = 0);
```

使用纹理坐标x,y获取绑定到纹理引用的CUDA数组，并返回其第comp个分量组成的向量。

### B.8.2 纹理参考函数

#### B.8.2.1 tex1Dfetch()

```
template<class DataType>
Type tex1Dfetch(
    texture<DataType, cudaTextureType1D,
            cudaReadModeElementType> texRef,
    int x);

float tex1Dfetch(
    texture<unsigned char, cudaTextureType1D,
            cudaReadModeNormalizedFloat> texRef,
    int x);

float tex1Dfetch(
    texture<signed char, cudaTextureType1D,
            cudaReadModeNormalizedFloat> texRef,
    int x);

float tex1Dfetch(
```

```
texture<unsigned short, cudaTextureType1D,  
        cudaReadModeNormalizedFloat> texRef,  
int x);  
  
float tex1Dfetch(  
    texture<signed short, cudaTextureType1D,  
            cudaReadModeNormalizedFloat> texRef,  
    int x);
```

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

除了上面的函数，也支持二元组和四元组；例如：

```
float4 tex1Dfetch(  
    texture<uchar4, cudaTextureType1D,  
            cudaReadModeNormalizedFloat> texRef,  
    int x);
```

使用纹理坐标x获取绑定到texRef的线性存储器区域。

#### B.8.2.2 tex1D()

```
template<class DataType, enum cudaTextureReadMode readMode>  
Type tex1D(texture<DataType, cudaTextureType1D, readMode> texRef,  
           float x);
```

使用浮点纹理坐标x获取绑定到纹理引用texRef的CUDA数组。

#### B.8.2.3 tex2D()

```
template<class DataType, enum cudaTextureReadMode readMode>  
Type tex2D(texture<DataType, cudaTextureType2D, readMode> texRef,  
           float x, float y);
```

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

#### B.8.2.4 tex3D()

```
template<class DataType, enum cudaTextureReadMode readMode>
Type tex3D(texture<DataType, cudaTextureType3D, readMode> texRef,
           float x, float y, float z);
```

使用纹理坐标x,y和z获取绑定到texRef的CUDA数组。

#### B.8.2.5 tex1DLayered()

```
template<class DataType, enum cudaTextureReadMode readMode>
Type tex1DLayered(
    texture<DataType, cudaTextureType1DLayered, readMode> texRef,
    float x, int layer);
```

使用纹理坐标x和索引layer获取绑定到一维层次纹理引用texRef的CUDA数组。

#### B.8.2.6 tex2DLayered()

```
template<class DataType, enum cudaTextureReadMode readMode>
Type tex2DLayered(
    texture<DataType, cudaTextureType2DLayered, readMode> texRef,
    float x, float y, int layer);
```

使用纹理坐标x、 y和索引layer获取绑定到二维层次纹理引用texRef的CUDA数组

#### B.8.2.7 texCubemap()

```
template<class DataType, enum cudaTextureReadMode readMode>
Type texCubemap(
    texture<DataType, cudaTextureTypeCubemap, readMode> texRef,
```

```
float x, float y, float z);
```

使用纹理坐标x,y和z获取绑定到立方图纹理引用的CUDA数组。

#### B.8.2.8 texCubemapLayered()

```
template<class DataType, enum cudaTextureReadMode readMode>
Type texCubemapLayered(
    texture<DataType, cudaTextureTypeCubemapLayered, readMode>
        texRef,
    float x, float y, float z, int layer);
```

使用三维纹理坐标x,y和z及索引layer获取绑定到层次立方图纹理引用的CUDA数组。

#### B.8.2.9 tex2Dgather()

```
template<class DataType, enum cudaTextureReadMode readMode>
Type tex2Dgather(
    texture<DataType, cudaTextureType2D, readMode> texRef,
    float x, float y, int comp = 0);
```

使用纹理坐标x,y获取绑定到纹理引用的CUDA数组，并返回其第comp个分量组成的向量。

### B.9 表面函数(surface)

只能计算能力2.0及以上的设备才支持表面函数。

[B.9.1](#)描述了表面对象，[B.9.2](#)描述了表面引用。

本节，boundaryMode 指明边界模式，边界模式指定了越界的表面坐标如何处理；其可选值有cudaBondaryModeClamp，此时越界坐标被钳位到有效坐标内；cudaBondaryModeZero，此时越界读返回，越界写被忽略；cudaBondaryModeTrap，此时越界读写导致内核崩溃。

## B.9.1 表面对象函数

### B.9.1.1 surf1Dread()

```
template<class T>
T surf1Dread(cudaSurfaceObject surfObj, int x,
             boundaryMode = cudaBoundaryModeTrap);
```

使用坐标x读取表面对象surfObj指定的CUDA数组。

### B.9.1.2 surf1Dwrite()

```
template<class T>
void surf1Dwrite(T data,
                 cudaSurfaceObject surfObj,
                 int x,
                 boundaryMode = cudaBoundaryModeTrap);
```

将值data写入一维表面对象surfObj指定的存储器位置x。

### B.9.1.3 surf2Dread()

```
template<class T>
T surf2Dread(cudaSurfaceObject surfObj,
             int x, int y,
             boundaryMode = cudaBoundaryModeTrap);
template<class T>
void surf2Dread(T* data,
               cudaSurfaceObject surfObj,
               int x, int y,
               boundaryMode = cudaBoundaryModeTrap);
```

使用坐标x,y读取二维表面对象指定的CUDA数组的值。

#### B.9.1.4 surf2Dwrite()

```
template<class T>
void surf2Dwrite(T data,
                 cudaSurfaceObject surfObj,
                 int x, int y,
                 boundaryMode = cudaBoundaryModeTrap);
```

将data写入二维表面对象surfObj指定的表面存储器位置x,y。

#### B.9.1.5 surf3Dread()

```
template<class T>
T surf3Dread(cudaSurfaceObject surfObj,
             int x, int y, int z,
             boundaryMode = cudaBoundaryModeTrap);
template<class T>
void surf3Dread(T* data,
               cudaSurfaceObject surfObj,
               int x, int y, int z,
               boundaryMode = cudaBoundaryModeTrap);
```

使用坐标x、y和z读取三维表面对象指定的CUDA数组。

#### B.9.1.6 surf3Dwrite()

```
template<class T>
void surf3Dwrite(T data,
                 cudaSurfaceObject surfObj,
                 int x, int y, int z,
                 boundaryMode = cudaBoundaryModeTrap);
```

将值data写入三维表面对象指定的CUDA数组位置x,y,z。



**B.9.1.7 surf1DLayeredread()**

```
template<class T>
T surf1DLayeredread(
    cudaSurfaceObject surfObj,
    int x, int layer,
    boundaryMode = cudaBoundaryModeTrap);

template<class T>
void surf1DLayeredread(T data,
    cudaSurfaceObject surfObj,
    int x, int layer,
    boundaryMode = cudaBoundaryModeTrap);
```

使用索引layer和坐标x读取一维层次表面对象surfObj指定的CUDA数组。

**B.9.1.8 surf1DLayeredwrite()**

```
template<class Type>
void surf1DLayeredwrite(T data,
    cudaSurfaceObject surfObj,
    int x, int layer,
    boundaryMode = cudaBoundaryModeTrap);
```

将值data写入一维层次表面对象surfObj指定的CUDA数组的layer层，坐标x位置。

**B.9.1.9 surf2DLayeredread()**

```
template<class T>
T surf2DLayeredread(
    cudaSurfaceObject surfObj,
    int x, int y, int layer,
    boundaryMode = cudaBoundaryModeTrap);

template<class T>
```

```
void surf2DLayeredread(T data,  
                        cudaSurfaceObject surfObj,  
                        int x, int y, int layer ,  
                        boundaryMode = cudaBoundaryModeTrap);
```

使用索引layer和坐标x读取二维层次表面对象surfObj指定的CUDA数组。

#### B.9.1.10 surf2DLayeredwrite()

```
template<class T>  
void surf2DLayeredwrite(T data,  
                        cudaSurfaceObject surfObj,  
                        int x, int y, int layer ,  
                        boundaryMode = cudaBoundaryModeTrap);
```

使用索引layer和坐标x,y将值data写入二维层次表面对象surfObj指定的CUDA数组。

#### B.9.1.11 surfCubemapread()

```
template<class T>  
T surfCubemapread(  
    cudaSurfaceObject surfObj,  
    int x, int y, int face ,  
    boundaryMode = cudaBoundaryModeTrap);  
template<class T>  
void surfCubemapread(T data,  
    cudaSurfaceObject surfObj,  
    int x, int y, int face ,  
    boundaryMode = cudaBoundaryModeTrap);
```

使用坐标x,y和面索引face读取立方体图表面对象surfObj指定的CUDA数组。

**B.9.1.12 surfCubemapwrite()**

```
template<class T>
void surfCubemapwrite(T data,
                      cudaSurfaceObject surfObj,
                      int x, int y, int face,
                      boundaryMode = cudaBoundaryModeTrap);
```

使用面索引face和坐标x,y将值data写入立方位图表面对象指定的CUDA数组。

**B.9.1.13 surfCubemapLayeredread()**

```
template<class T>
T surfCubemapLayeredread(
    cudaSurfaceObject surfObj,
    int x, int y, int layerFace,
    boundaryMode = cudaBoundaryModeTrap);
template<class T>
void surfCubemapLayeredread(T data,
    cudaSurfaceObject surfObj,
    int x, int y, int layerFace,
    boundaryMode = cudaBoundaryModeTrap);
```

使用索引layerFace和坐标x,y读取层次立方位图表面对象surfObj指定的CUDA数组。

**B.9.1.14 surfCubemapLayeredwrite()**

```
template<class T>
void surfCubemapLayeredwrite(T data,
    cudaSurfaceObject surfObj,
    int x, int y, int layerFace,
    boundaryMode = cudaBoundaryModeTrap);
```

将值data写入层次立方图表面对象surfObj指定的CUDA数组的layer层，坐标x,y位置。

## B.9.2 表面引用API

### B.9.2.1 surf1Dread()

```
template<class Type>
Type surf1Dread(surface<void, cudaSurfaceType1D> surfRef,
                int x,
                boundaryMode = cudaBoundaryModeTrap);
template<class Type>
void surf1Dread(Type data,
                surface<void, cudaSurfaceType1D> surfRef,
                int x,
                boundaryMode = cudaBoundaryModeTrap);
```

使用坐标x读取绑定到表面参考surfRef的CUDA数组。

### B.9.2.2 surf1Dwrite()

```
template<class Type>
void surf1Dwrite(Type data,
                 surface<void, cudaSurfaceType1D> surfRef,
                 int x,
                 boundaryMode = cudaBoundaryModeTrap);
```

将值data写入绑定到surfRef的表面存储器位置x。

### B.9.2.3 surf2Dread()

```
template<class Type>
Type surf2Dread(surface<void, cudaSurfaceType2D> surfRef,
                int x, int y,
                boundaryMode = cudaBoundaryModeTrap);
```

```
template<class Type>
void surf2Dread(Type* data,
                surface<void, cudaSurfaceType2D> surfRef,
                int x, int y,
                boundaryMode = cudaBoundaryModeTrap);
```

读取绑定的surfRef的表面存储器位置x, y的值。

#### B.9.2.4 surf2Dwrite()

```
template<class Type>
void surf3Dwrite(Type data,
                 surface<void, cudaSurfaceType3D> surfRef,
                 int x, int y, int z,
                 boundaryMode = cudaBoundaryModeTrap);
```

将data写入绑定到surfRef的表面存储器位置x,y。

#### B.9.2.5 surf3Dread()

```
template<class Type>
Type surf3Dread(surface<void, cudaSurfaceType3D> surfRef,
               int x, int y, int z,
               boundaryMode = cudaBoundaryModeTrap);
template<class Type>
void surf3Dread(Type* data,
                surface<void, cudaSurfaceType3D> surfRef,
                int x, int y, int z,
                boundaryMode = cudaBoundaryModeTrap);
```

使用纹理坐标x,y,z读取绑定到三维表面引用surfRef的CUDA数组。

#### B.9.2.6 surf3Dwrite()

```
template<class Type>
void surf3Dwrite(Type data,
                 surface<void, cudaSurfaceType3D> surfRef,
                 int x, int y, int z,
                 boundaryMode = cudaBoundaryModeTrap);
```

将值data写入三维表面引用surfRef绑定的CUDA数组位置x,y,z。

#### B.9.2.7 surf1DLayeredread()

```
template<class Type>
Type surf1DLayeredread(
    surface<void, cudaSurfaceType1DLayered> surfRef,
    int x, int layer,
    boundaryMode = cudaBoundaryModeTrap);

template<class Type>
void surf1DLayeredread(Type* data,
                      surface<void, cudaSurfaceType1DLayered> surfRef,
                      int x, int layer,
                      boundaryMode = cudaBoundaryModeTrap);
```

使用层索引layer和坐标x读取绑定到表面引用surfRef的CUDA数组。

#### B.9.2.8 surf1DLayeredwrite()

```
template<class Type>
void surf1DLayeredwrite(Type data,
                       surface<void, cudaSurfaceType1DLayered> surfRef,
                       int x, int layer,
                       boundaryMode = cudaBoundaryModeTrap);
```

使用索引layer和坐标x，将值data写入绑定到一维层次表面引用surfRef的CUDA数组。

**B.9.2.9 surf2DLayeredread()**

```
template<class Type>
Type surf2DLayeredread(
    surface<void, cudaSurfaceType2DLayered> surfRef,
    int x, int y, int layer,
    boundaryMode = cudaBoundaryModeTrap);
template<class Type>
void surf2DLayeredread(Type* data,
    surface<void, cudaSurfaceType2DLayered> surfRef,
    int x, int y, int layer,
    boundaryMode = cudaBoundaryModeTrap);
```

使用索引layer和坐标x,y读取绑定到表面引用surfRef的CUDA数组。

**B.9.2.10 surf2DLayeredwrite()**

```
template<class Type>
void surf2DLayeredwrite(Type data,
    surface<void, cudaSurfaceType2DLayered> surfRef,
    int x, int y, int layer,
    boundaryMode = cudaBoundaryModeTrap);
```

将值data写入二维层次表面引用绑定的CUDA数组的layer层，坐标x,y。

**B.9.2.11 surfCubemapread()**

```
template<class Type>
Type surfCubemapread(
    surface<void, cudaSurfaceTypeCubemap> surfRef,
    int x, int y, int face,
    boundaryMode = cudaBoundaryModeTrap);
template<class Type>
void surfCubemapread(Type* data,
```

```
surface<void, cudaSurfaceTypeCubemap> surfRef,  
int x, int y, int face,  
boundaryMode = cudaBoundaryModeTrap);
```

使用面索引face和坐标x,y读取绑定到立方位图表面引用的CUDA数组。

#### B.9.2.12 surfCubemapwrite()

```
template<class Type>  
void surfCubemapwrite(Type data,  
surface<void, cudaSurfaceTypeCubemap> surfRef,  
int x, int y, int face,  
boundaryMode = cudaBoundaryModeTrap);
```

将值data写入到绑定到立方位图表面引用的CUDA数组，索引是layerFace，坐标x,y。

#### B.9.2.13 surfCubemapLayeredread()

```
template<class Type>  
Type surfCubemapLayeredread(  
surface<void, cudaSurfaceTypeCubemapLayered> surfRef,  
int x, int y, int layerFace,  
boundaryMode = cudaBoundaryModeTrap);  
template<class Type>  
void surfCubemapLayeredread(Type data,  
surface<void, cudaSurfaceTypeCubemapLayered> surfRef,  
int x, int y, int layerFace,  
boundaryMode = cudaBoundaryModeTrap);
```

使用索引layerFace和坐标x,y读取绑定到层次立方位图表面引用surfRef的CUDA数组。



### B.9.2.14 surfCubemapLayeredwrite()

```
template<class Type>
void surfCubemapLayeredwrite(Type data,
                             surface<void, cudaSurfaceTypeCubemapLayered> surfRef,
                             int x, int y, int layerFace,
                             boundaryMode = cudaBoundaryModeTrap);
```

使用索引layerFace和坐标x,y将值data写入绑定到层次立方图表面引用surfRef的CUDA数组。

## B.10 时间函数

```
clock_t clock();
long long int clock64();
```

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

## B.11 原子函数

原子函数对位于全局存储器或共享存储器内的一个32 位或64 位字执行读取修改写入原子操作。例如，atomicAdd() 将全局或共享存储器内的某个地址读取字，将其与一个整型相加，并将结果写回同一地址。说操作是原子的，是因为它的执行不受其他线程的干扰。换句话说，在操作完成前，其他任何线程都无法访问此地址。原子函数只能在设备代码中使用，而且从主机或其它设备的观点来看，作用在被映射分页锁定存储器（参见3.2.4.3）上的原子函数不是原子的。

如F.1所述，不同计算能力对原子函数的支持不同：

- 原子函数只在计算能力1.1或更高的设备上可用。

- 操作共享存储器中32位整型和全局存储器中64 位整型的原子函数只在计算能力为1.2或更高的设备上可用。
- 操作共享存储器里的64位字的原子函数只在计算能力2.x的设备上可用。
- 唯有atomicExch()和atomicAdd()能够操作32位浮点数：
  - 计算能力1.1及以上的设备，atomicExch()可以作用在全局存储器上。
  - 计算能力1.2及以上的设备，atomicExch()可以作用在共享存储器上。
  - 计算能力2.x及以上的设备，atomicAdd()可以作用在共享存储器和全局存储器上。

注意：但是任何一个原子函数都可以基于atomicCAS()（比较并交换）实现。例如双精度的atomicAdd()实现如下：

```
__device__ double atomicAdd(double* address, double val)
{
    unsigned long long int* address_as_ull =
        (unsigned long long int*)address;
    unsigned long long int old = *address_as_ull, assumed;
    do {
        assumed = old;
        old = atomicCAS(address_as_ull, assumed,
            __double_as_longlong(val +
                __longlong_as_double(assumed)));
    } while (assumed != old);
    return __longlong_as_double(old);
}
```

### B.11.1 数学函数

#### B.11.1.1 atomicAdd()

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

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

浮点版本的atomAdd()只支持计算能力2.x及以上的设备。

#### B.11.1.2 atomicSub()

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

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

#### B.11.1.3 atomicExch()

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

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

#### B.11.1.4 atomicMin()

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

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

64位版本的atomicMin() 只在计算能力3.5及以上的设备中得到支持。

#### B.11.1.5 atomicMax()

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

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

64位版本的atomicMin() 只在计算能力3.5及以上的设备中得到支持。

#### B.11.1.6 atomicInc()

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

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

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

### B.11.1.8 atomicCAS()

[illegible]

### B.11.2 位逻辑函数

[illegible]

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

64位版本只在计算能力3.5及以上设备上得到支持。

#### B.11.2.2 `atomicOr()`

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

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

64位版本只在计算能力3.5及以上设备上得到支持

#### B.11.2.3 `atomicXor()`

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

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

64位版本只在计算能力3.5及以上设备上得到支持

## B.12 束表决（warp vote）函数

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

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

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

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

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

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

为束内所有线程计算predicate值，并返回一个整数，如果束内第N个线程的predicate值为非零，则该整数的第N位为1。计算能力为2.x及以上的设备支持此函数。

## B.13 束洗牌函数

\_\_shfl、\_\_shfl\_up、\_\_shfl\_down和\_\_shfl\_xor在束内线程间交换变量值。

这些函数只在计算能力3.x的设备中得到支持。

### B.13.1 概览

\_\_shfl内置函数允许束内线程不使用共享存储器交换变量值。束内活跃线程同时发生每线程4字节的交换。8字节数的交换必须拆成两个\_\_shfl()调用。

线程只能从另外一个参与\_\_shfl()的目标线程那里读取数据，如果目标线程并不参与交换，返回的数据未定义。

\_\_shfl()内置函数有一个可选宽度参数，这个参数允许将束划分成片段-如以SIMD方式将束分成4组，每组8个线程交换数据。如果宽度小于warpSize，束的每个子组就像一个独立的从0索引开始的实体。线程只能和同一个子组内的线程交换数据。子组的宽度必须是2的幂，以保证束能够被均分。如果子组的宽度不是2的幂或大于warpSize，结果没有定义。

```
int __shfl (int var, int srcLane, int width=warpSize);  
float __shfl (float var, int srcLane, int width=warpSize);
```

`__shfl()`返回线程`srcLane`持有的`var`。如果`srcLane`不在`[0:width-1]`范围内，返回调用线程自己的`var`值。

```
int __shfl_up(int var, unsigned int delta, int width=warpSize);
float __shfl_up(float var, unsigned int delta, int width=warpSize);
```

`__shfl_up()`返回线程索引为`id`的线程持有的`var`值，其中`id`等于调用线程的束索引减去`delta`。从效果上来说，就像是将`var`在束内上移了`delta`个线程。调用线程束索引小于`delta`的，其`var`不发生改变。

```
int __shfl_down(int var, unsigned int delta, int width=warpSize);
float __shfl_down(float var, unsigned int delta, int width=warpSize);
```

`__shfl_down()`返回线程索引为`id`的线程持有的`var`值，其中`id`等于调用线程的束索引加上`delta`。从效果上来说，就像是将`var`在束内下移了`delta`个线程。调用线程束索引大于`warpSize-delta`的，其`var`不发生改变。

```
int __shfl_xor(int var, int laneMask, int width=warpSize);
float __shfl_xor(float var, int laneMask, int width=warpSize);
```

`__shfl_xor()`返回线程索引为`id`的线程持有的`var`值，其中`id`等于调用线程的束索引异或`laneMask`。如果`id`不在`[0:width-1]`范围内，返回调用线程自己的`var`。

### B.13.2 在束内广播一个值

```
--global__ void bcast(int arg) {
    int laneId = threadIdx.x & 0x1f;
    int value;
    if (laneId == 0)           // Note unused variable for
        value = arg;         // all threads except lane 0
    value = __shfl(value, 0);  // Get “value” from lane 0
    if (value != arg)
        printf “(Thread %d failed.\n” n, threadIdx.x);
}
```



```

void main() {
    bcast<<< 1, 32 >>>(1234);
    cudaDeviceSynchronize();
}

```

### B.13.3 计算8个线程的前缀和

```

__global__ void scan4() {
    // Seed sample starting value (inverse of lane ID)
    int value = 31 - laneId;

    // Loop to accumulate scan within my partition.
    // Scan requires log2(n) == 3 steps for 8 threads
    // It works by an accumulated sum up the warp
    // by 1, 2, 4, 8 etc. steps.
    for (int i=1; i<=4; i*=2) {
        // Note: shfl requires all threads being
        // accessed to be active. Therefore we do
        // the __shfl unconditionally so that we
        // can read even from threads which won' t do a
        // sum, and then conditionally assign the result.
        int n = __shfl_up(value, i, 8);
        if (laneId >= i)
            value += n;
    }

    printf "(Thread %d final value = %d\" n, threadIdx.x, value);
}

void main() {
    scan4<<< 1, 32 >>>();
    cudaDeviceSynchronize();
}

```

```
}

```

#### B.13.4 束内求和

```
__global__ void warpReduce() {
    // Seed starting value as inverse lane ID
    int value = 31 - laneId;

    // Use XOR mode to perform butterfly reduction
    for (int i=16; i>=1; i/=2)
        value += __shfl_xor(value, i, 32);

    // “value” now contains the sum across all threads
    printf “(Thread %d final value = %d\\” n, threadIdx.x, value);
}

void main() {
    warpReduce<<< 1, 32 >>>();
    cudaDeviceSynchronize();
}
```

### B.14 取样计数器函数

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

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

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

第一个多处理器的，1，..，7号计数器值可通过CUDA profiler取得，方式是在profiler.conf文件中列出prof\_trigger\_00，prof\_trigger\_01，..，prof\_trigger\_07，

等等（详见profiler手册）。在每次内核调用前，所有的计数器重置（注意当应用通过CUDA调试器或CUDA profiler运行时(cuda-gdb, CUDA Visual Profiler, Parallel Nsight)，所有的发射都是同步的）。

## B.15 断言

断言只在计算能力2.x及以上的设备中得到支持。

```
void assert(int expression);
```

如果expression等于0，停止内核执行。如果程序在调试器内执行，这将先激发一个断点，调试器就可以探查设备的当前状态。否则在调用cudaDeviceSynchronize()、cudaStreamSynchronize()或cudaEventsSynchronize()之后，所有expression为0的线程将会向标准错误流stderr打印一条消息。消息的格式如下：

```
<filename>:<line number>:<function>:  
block: [blockId.x,blockId.x,blockIdx.z],  
thread: [threadIdx.x,threadIdx.y,threadIdx.z]  
Assertion '<expression>' failed .
```

任何随后作用在同一设备上的主机端同步调用将返回cudaErrorAssert。除非使用cudaDeviceReset()重新初始化设备，否则不能向此设备发送命令。

如果expression不是0，内核执行不受影响。

如下面的代码：

```
#include <assert.h>  
  
// assert() is only supported  
// for devices of compute capability 2.0 and higher  
#if defined(__CUDA_ARCH__) && (__CUDA_ARCH__ < 200)  
#undef assert  
#define assert(arg)  
#endif
```

```
--global__ void testAssert(void)
{
    int is_one = 1;
    int should_be_one = 0;

    // This will have no effect
    assert(is_one);

    // This will halt kernel execution
    assert(should_be_one);
}

int main(int argc, char* argv[])
{
    testAssert<<<1,1>>>();
    cudaDeviceSynchronize();
    return 0;
}
```

将输出：

```
test.cu:19: void testAssert(): block: [0,0,0], thread: [0,0,0] Assertion
'should_be_one' failed .
```

断言的目的是调试。它会降低性能，因此建议在产品代码中关闭它们。可通过在包含assert.h之前定义NDEBUG宏关闭。要注意expression不能包含副作用(如++i > 0)，否则关闭断言会影响代码的功能。

## B.16 格式化输出

格式化输出只在计算能力2.x及以上的设备上可用。

```
int printf(const char *format[, arg, ...]) ;
```

将内核中的数据格式化输出到主机端流。

内核内printf函数和标准C库的printf函数非常相似，用户可参考主机手册对printf的详细说明。本质上，format传递的字符串中格式化参数被对应的参数替代后输出到主机流。下面列出了支持的格式化字符串。

printf像其它任何的设备端函数一样在调用线程的上下文中被每个线程执行。在有多个线程的内核中，意味着直接调用，将会被每一个线程执行。主机流会出现输出字符串的多个版本，每个调用printf的线程一个版本。

如果只有一个线程要输出数据，依赖程序员将输出限制在某个线程上。

和标准C中printf返回打印字符的个数不同，CUDA的printf返回解析的参数个数。如果格式化字符串后没有参数，返回0；如果格式化字符串是NULL,返回-1；如果出现内部错误，返回-2。

### B.16.1 格式化符号

和标准C的printf一样，格式化符的形式如下：`%[flags][width][.precision][size]type`支持下面的域：

标签：'#' ' ' '0' '+' '-'

宽度：'\*' '0-9'

精确度：'0-9'

尺寸：'h' 'l' 'll'

类型：'%cdiouxXpeEfgGaAs'

注意CUDA的printf接受任何标签、宽度、精度、尺寸和类型的联用，而不管它们是否有效。也就是说"%hd"会被接收而且printf会试图在参数列表的对应位置打印双精度变量。

### B.16.2 限制

printf()最终在主机系统上格式化输出。这意味着格式化字符串必须能够被主机编译器和C库理解。尽量保证CUDA支持的格式化字符是最常用的主机的格式化字符的一个子集，但是它真实的表现依旧是依赖主机的。

如B.16.1描述的，printf()接受任何有效标签和类型的联用。因为无法保证最终的格式化输出结果是有效的还是无效的。这导致了如果格式化字符串中包含了无效联用，其输出结果是没有定义的。

除格式化字符串外，`printf()`最多可接受32参数。超过此值的将会被忽略。

由于在64位windows上，`long`类型的长度和其它平台不同（windows平台四个字节，其它64位平台八个字节）。在非windows 64位平台上编译的程序在windows 64位平台上执行，所用使用“%ld”格式化字符的输出都崩溃。因此建议编译平台和执行平台一致以保证安全性。

`printf()`的输出缓冲区在内核启动前设置成固定长度（参见B.16.3）。它是循环的，如果在内核执行时产生了过多的输出，旧的输出会被覆盖。只有在下列操作之一执行时会刷新：

- 通过<<<, >>>或`cuLaunch()`发射内核（启动开始和如果`CUDA_LAUNCH_BLOCKING`环境变量设置成1时启动结束），
- 通过`cudaDeviceSynchronize()`, `cuCtxSynchronze()`, `cudaStreamSynchronize()` `cuStreamSynchronize()` `cudaEventSynchronize()`或`cudaEventSynchronize()`的同步，
- 使用`cudaMemcpy*()`或`cuMemcpy*()`的阻塞版本的存储器复制，
- 通过`cuModuleLoad()`或`cuModuleUnload()`的模块加载和卸载，
- 通过`cudaDeviceReset()`或`cuCtxDestroy()`销毁上下文。

注意程序退出时，缓冲区不会自动刷新。用户须显式的调用`cudaDeviceReset()`或`cuCtxDestroy`。

### B.16.3 相关的主机端API

下列API用于获得或设置用于传输`printf()`参数和内部元数据到主机的缓冲区大小（默认1M）：

```
cudaDeviceGetLimit(size_t *size, cudaLimitPrintfFifoSize)
cudaDeviceSetLimit(cudaLimitPrintfFifoSize, size_t size)
```

### B.16.4 例程

下面代码：

```
#include "stdio.h"

// printf() is only supported
// for devices of compute capability 2.0 and higher
#if defined(__CUDA_ARCH__) && (__CUDA_ARCH__ < 200)
    #define printf(f, ...) ((void)(f, __VA_ARGS__),0)
#endif

__global__ void helloCUDA(float f)
{
    printf("Hello thread %d, f=%f\n", threadIdx.x, f);
}

int main()
{
    helloCUDA<<<1, 5>>>(1.2345f);
    cudaDeviceSynchronize();
    return 0;
}
```

输出：

```
Hello thread 2, f=1.2345
Hello thread 1, f=1.2345
Hello thread 4, f=1.2345
Hello thread 0, f=1.2345
Hello thread 3, f=1.2345
```

注意输出的行数和网格内启动的线程数有关。如同我们所希望的，全局值（如float f）所有线程共享，本地值（如threadIdx.x）每个线程都不同。

下面的代码：

```
#include "stdio.h"

// printf() is only supported
// for devices of compute capability 2.0 and higher
#if defined(__CUDA_ARCH__) && (__CUDA_ARCH__ < 200)
    #define printf(f, ...) ((void)(f, __VA_ARGS__),0)
#endif

__global__ void helloCUDA(float f)
{
    if (threadIdx.x == 0)
        printf("Hello_thread_%d,f=%f\n", threadIdx.x, f) ;
}

int main()
{
    helloCUDA<<<1, 5>>>(1.2345f);
    cudaDeviceSynchronize();
    return 0;
}
```

输出：

```
Hello thread 0, f=1.2345
```

if()语句用于限制在调用时那个线程调用printf(),因此只有一行输出。

## B.17 动态全局存储器分配

只有计算能力2.x及以上的设备支持动态全局存储器分配。

```
void* malloc(size_t size);
void free(void* ptr);
```



动态分配和释放来自全局存储器中的固定尺寸的堆。

```
void* memcpy(void* dest, const void* src, size_t size);
```

从src指向的地址拷贝size个字节到dest指向的地址。

```
void* memset(void* ptr, int value, size_t size);
```

从ptr指向的地址开始赋值size个字节，每个字节设置成value。

CUDA内核函数内的malloc()函数从设备堆中分配至少size字节，且返回指向已分配存储器的指针，如果没有足够的存储器以满足这次分配就返回NULL。返回的指针保证对齐到16字节边界。

CUDA内核函数内的free()函数释放ptr指向的存储器，ptr必须由前面的malloc()调用返回。如果ptr是NULL，free()调用被忽略。在同一个ptr上重复调用free()，行为未定义。

既定的CUDA线程通过malloc()分配的存储器在CUDA上下文生命期内都存在，或直到其被显式调用free()函数为止。它可以被其它的CUDA线程使用，即使是后面的内核启动中的线程。任何CUDA线程可以释放其它线程分配的存储器，但是要保证同一指针不会被释放多次。

### B.17.1 堆存储器分配

设备存储器堆拥有固定的尺寸，在任何使用malloc()或free()的程序装载进上下文前，堆的尺寸必须指定。如果任何使用malloc()的程序没有显式的指定堆的尺寸，默认尺寸是8 M。

下面的API获得或设置堆尺寸。

```
cudaDeviceGetLimit(size_t* size, cudaLimitMallocHeapSize)  
cudaDeviceSetLimit(cudaLimitMallocHeapSize, size_t size)
```

堆尺寸保证至少size字节。cuCtxGetLimit()和cudaDeviceGetLimit()返回当前要求的堆尺寸。

实际的堆存储器分配发生在模块被加载进上下文时，加载可以是显式的通过CUDA驱动API，也可以是隐式的通过CUDA运行时API。如果存储器分配失败，模块加载会产生CUDA\_ERROR\_SHARED\_OBJECT\_INIT\_FAILED错误。

一旦模块加载已经发生堆尺寸就不能改变，也不能依据需要动态更改尺寸。

为设备堆保留的存储器是通过主机端CUDA调用（如cudaMalloc()）分配的存储器的补充。

### B.17.2 与设备存储器API的互操作

使用malloc()分配的存储器不能使用运行时（即调用任何来自3.2的函数释放设备存储器函数）释放。

类似地，运行时（即调用任何来自3.2的分配设备存储器函数）分配的存储器不能使用free()释放。

### B.17.3 例程

#### B.17.3.1 每个线程的分配

下面的代码：

```
#include <stdlib.h>
#include <stdio.h>

__global__ void mallocTest()
{
    size_t size = 123;
    char* ptr = (char*)malloc(size);
    memset(ptr, 0, size);
    printf("Thread_%d_got_pointer:_%p\n", threadIdx.x, ptr);
    free(ptr);
}

int main()
{
    // Set a heap size of 128 megabytes. Note that this must
    // be done before any kernel is launched.
```

```
    cudaDeviceSetLimit(cudaLimitMallocHeapSize, 128*1024*1024);  
    mallocTest<<<1, 5>>>();  
    cudaDeviceSynchronize();  
    return 0;  
}
```

将输出：

```
Thread 0 got pointer: 00057020  
Thread 1 got pointer: 0005708c  
Thread 2 got pointer: 000570f8  
Thread 3 got pointer: 00057164  
Thread 4 got pointer: 000571d0
```

注意每个线程怎样遇上malloc()命令和获得它自己的分配。(具体的指针值可能不同)

### B.17.3.2 每个线程块的分配

```
#include <stdlib.h>  
  
__global__ void mallocTest()  
{  
    __shared__ int* data;  
  
    // The first thread in the block does the allocation and initialization  
    // and then shares the pointer with all other threads through shared  
    memory,  
    // so that access can easily be coalesced.  
    // 64 bytes per thread are allocated.  
    if (threadIdx.x == 0) {  
        size_t size = blockDim.x * 64;  
        data = (int*)malloc(size);  
        memset(data, 0, size);  
    }
```

```
    }
    __syncthreads();

    // Check for failure
    if (data == NULL)
        return;

    // Threads index into the memory, ensuring coalescence
    int* ptr = data;
    for (int i = 0; i < 64; ++i)
        ptr[i * blockDim.x + threadIdx.x] = threadIdx.x;

    // Ensure all threads complete before freeing
    __syncthreads();

    // Only one thread may free the memory!
    if (threadIdx.x == 0)
        free(data);
}

int main()
{
    cudaDeviceSetLimit(cudaLimitMallocHeapSize, 128*1024*1024);
    mallocTest<<<10, 128>>>();
    cudaDeviceSynchronize();
    return 0;
}
```

### B.17.3.3 在内核启动之间持久的分配

```
#include <stdlib.h>
#include <stdio.h>
```

```
#define NUM_BLOCKS 20

__device__ int* dataptr[NUM_BLOCKS]; // Per-block pointer

__global__ void allocmem()
{
    // Only the first thread in the block does the allocation
    // since we want only one allocation per block.
    if (threadIdx.x == 0)
        dataptr[blockIdx.x] = (int*)malloc(blockDim.x * 4);
    __syncthreads();

    // Check for failure
    if (dataptr[blockIdx.x] == NULL)
        return;

    // Zero the data with all threads in parallel
    dataptr[blockIdx.x][threadIdx.x] = 0;
}

// Simple example: store thread ID into each element
__global__ void usemem()
{
    int* ptr = dataptr[blockIdx.x];
    if (ptr != NULL)
        ptr[threadIdx.x] += threadIdx.x;
}

// Print the content of the buffer before freeing it
__global__ void freemem()
{
```

```
int* ptr = dataptr[blockIdx.x];
if (ptr != NULL)
    printf("Block_%d, Thread_%d: final_value = %d\n",
           blockIdx.x, threadIdx.x, ptr[threadIdx.x]);

// Only free from one thread!
if (threadIdx.x == 0)
    free(ptr);
}

int main()
{
    cudaDeviceSetLimit(cudaLimitMallocHeapSize, 128*1024*1024);

    // Allocate memory
    allocmem<<< NUM_BLOCKS, 10 >>>();

    // Use memory
    usemem<<< NUM_BLOCKS, 10 >>>();
    usemem<<< NUM_BLOCKS, 10 >>>();
    usemem<<< NUM_BLOCKS, 10 >>>();

    // Free memory
    freemem<<< NUM_BLOCKS, 10 >>>();

    cudaDeviceSynchronize();

    return 0;
}
```

## B.18 执行配置

任何对 `_global_` 函数的调用都必须指定该调用的执行配置。执行配置定义将用于在该设备上执行函数的网格和块的维度，以及相关的流（参见3.2了解流的详细内容）。

使用运行时API时，可通过在函数名称和括号参数列表之间插入 `<<< Dg, Db, Ns, S >>>` 形式的表达式来指定，其中：

- `Dg` 的类型为 `dim3`（参见B.3.2），指定网格的维度和大小，`Dg.x * Dg.y` 等于所发射的块数量；对于计算能力1.x的设备，`Dg.z` 必须等于1；
- `Db` 的类型为 `dim3`（参见B.3.2），指定各块的维度和大小，`Db.x * Db.y * Db.z` 等于各块的线程数量；
- `Ns` 的类型为 `size_t`，指定各块为此调用动态分配的共享存储器（除静态分配的存储器之外），这些动态分配的存储器可供声明为动态数组的其他任何变量使用（参见3.2.3），`Ns` 是一个可选参数，默认值为0；
- `S` 的类型为 `cudaStream_t`，指定相关流；`S` 是一个可选参数，默认值为0。

举例来说，一个函数的声明如下：

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

必须通过如下方法来调用此函数：

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

执行配置的参数将在实际函数参数之前被求值，对于计算能力1.x的设备，通过共享存储器同时传递给设备。

如果 `Dg` 或 `Db` 大于设备允许的最大值，对指定设备的最大值参看F，或 `Ns` 大于设备上可用的共享存储器最大值减去静态分配、函数参数（为计算能力1.x）和执行配置所需的共享存储器数量，则函数将失败。

## B.19 启动绑定

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

因此，编译器会试探性的在保持寄存器溢出（参见5.3.2）的前提下最小化寄存器使用和最小化指令数量。应用可以以启动绑定的形式提供额外的信息给编译器以辅助这种试探，启动绑定在定义内核函数时使用\_\_launch\_bounds\_\_()限定符指定。

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

- maxThreadsPerBlock指定应用启动MyKernel内核时每个块内允许的最大线程数；它被编译成.maxntid PTX指令；
- minBlocksPerMultiprocessor是可选的，其指定每个多处理器最小常驻块数量；它被编译成.minnctapersm PTX指令。

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

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



- 如果minBlocksPerMultiprocessor和maxThreadsPerBlock都指定，编译器可能增加寄存器使用以减少指令数量和更好的隐藏单线程指令延迟。

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

为既定内核优化发射绑定依据主架构修订号变化。例子代码展示了怎样使用3.1.3的\_\_CUDA\_ARCH宏解决这个问题。

```
#define THREADS_PER_BLOCK 256
#if __CUDA_ARCH__ >= 200
    #define MY_KERNEL_MAX_THREADS (2 *
        THREADS_PER_BLOCK)
    #define MY_KERNEL_MIN_BLOCKS 3
#else
    #define MY_KERNEL_MAX_THREADS THREADS_PER_BLOCK
    #define MY_KERNEL_MIN_BLOCKS 2
#endif

// Device code
__global__ void
__launch_bounds__(MY_KERNEL_MAX_THREADS,
    MY_KERNEL_MIN_BLOCKS)
MyKernel(...)
{
    ...
}
```

通常，使用最大块内线程数量（\_\_launch\_bounds\_\_()的第一个参数指定）调用MyKernel，在执行配置时，倾向使用MY\_KERNEL\_MAX\_THREADS作为块内线程数：

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

但是这不会工作，因为如3.1.3提到的`__CUDA_ARCH`并没有在主机代码中指定，所以即使`__CUDA_ARCH`大于200，`MyKernel`也会以每块256个线程发射。块内线程数应当以下面的方式确定：

- 或者在编译时使用不依赖`__CUDA_ARCH__`的宏，如：

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

- 或者在运行时基于计算能力

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

使用`ptxas-options=-v`编译器选项可以报告寄存器用量。常驻块数量可从CUDA profiler中给出的占用率（参见5.3.2了解占用率的定义）得到。

对于`__global__`函数的寄存器用量也可以使用`-maxrregcount`编译器选项控制。对于启动绑定的函数，`-maxrregcount`值会被忽略。

## B.20 `#pragma unroll`

默认情况下，编译器将展开具有已知循环计数的小循环。`#pragma unroll`指令可用于控制任何给定循环的展开操作。它必须紧接于循环之前，而且仅应用于该循环。可选择性的在其后接一个数字（译者注：即使是宏也是不允许的，必须是字面数值），指定必须展开多少次循环。

例如，在下面的代码示例中：

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

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

#pragma unroll 1 将阻止编译器展开一个循环。

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

## B.21 SIMD 视频指令

PTX指令集3.0引入了SIMD(单指令，多数据)视频指令，它能够操作一对16位值或4个8位值。这些指令在计算能力3.0的设备上可用。

SIMD视频指令是：

- vadd2, vadd4
- vsub2, vsub4
- vavg2, vavg4
- vabsdiff2, vabsdiff4
- vmin2, vmin4
- vmax2, vmax4
- vset2, vset4

CUDA程序能够通过asm()语句包含PTX指令，比如SIMD视频指令。asm()语句的基本语法是：

```
asm("template-string" : "constraint"(output) : "constraint"(input));
```

一个使用vabsdiff4指令的例子是：

```
asm("vabsdiff4.u32.u32.u32.add" " %0,%1,%2,%3;" : "=r" (result): "r" (A), "r" (B), "r" (C));
```

这使用`vabsdiff4`指令计算一个整型4字节SIMD绝对差的和。绝对差值通过使用SIMD方式计算无符号整型A和B的每个字节得到。可选的求和运算(`.add`)指定求这些差的和。

参考文档“在CUDA中使用内联PTX汇编”了解在代码中使用汇编的细节。参考PTX指令集文档“并行线程执行指令集版本3.0”了解你所使用PTX版本的详细指令信息。

## 附录 C 数学函数

参考手册列举了所有设备代码支持的C/C++标准库的数学函数和其描述，及内置函数（它们只能在设备代码中使用）。

本附录提供了这些函数中部分函数的精度信息。

### C.1 标准函数

本节的函数既可用于主机代码也可用于设备代码。

此节指出各函数在设备上执行时的误差范围。当函数在主机上执行，而主机并未提供此函数时，误差范围同样适用。

这里列举的误差范围经过了广泛的测试，但并没有穷尽测试，所以不能保证没有偏差。

#### C.1.1 单精度浮点函数

加法和乘法是符合IEEE 的，因此误差最多为0.5 ulp。但编译器往往将其整合为一个单独的乘加指令（FMAD），在计算能力1.x的设备上，FMAD截取乘法的中间结果(参见C.1.1)。可通过使用 `_fadd_rn()` 和 `_fmul_rn()` 函数来避免这样的整合（请参见C.2）。

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

表 C.1: 数学标准库函数及其最大ULP 错误。最大错误表示为正确舍入的单精度结果和CUDA 库函数返回的结果之差（以ulp 计算）的绝对值

函数	最大ulp 错误
<code>x+y</code>	（IEEE-754 就近舍入到最近偶数） （在计算能力1.x设备上，加法合并成FMAD 时除外）

$x*y$	(IEEE-754 就近舍入到最近偶数) (在计算能力1.x设备上, 加法合并成FMAD 时除外)
$x/y$	计算能力 $\geq 2$ 的设备, 使用-prec-div=true编译时,; 其它, 2 (整个范围)
$1/x$	计算能力 $\geq 2$ 的设备, 使用-prec-div=true编译时,; 其它, 1 (整个范围)
$1/\text{sqrtf}(x)$ $\text{rsqrtf}(x)$	2 (整个范围) 只有当它被编译器转化为 $\text{rsqrtf}(x)$ 才应用到 $1/\text{sqrtf}(x)$
$\text{sqrtf}(x)$	计算能力 $\geq 2$ 的设备, 使用-prec-div=true编译时,; 其它, 3(整个范围)
$\text{cbrtf}(x)$	1(整个范围)
$\text{rcbrtf}(x)$	2(整个范围)
$\text{hypotf}(x,y)$	3(整个范围)
$\text{expf}(x)$	2(整个范围)
$\text{exp2f}(x)$	2(整个范围)
$\text{exp10f}(x)$	2(整个范围)
$\text{expm1f}(x)$	1(整个范围)
$\text{logf}(x)$	1(整个范围)
$\text{log2f}(x)$	3(整个范围)
$\text{log10f}(x)$	3(整个范围)
$\text{log1pf}(x)$	2(整个范围)
$\text{sinf}(x)$	2(整个范围)
$\text{cosf}(x)$	2(整个范围)
$\text{tanf}(x)$	4(整个范围)
$\text{sincosf}(x,\text{sptr},\text{cptr})$	2(整个范围)
$\text{sinpif}(x)$	2(整个范围)
$\text{cospif}(x)$	2(整个范围)
$\text{sincospif}(x, \text{ sptr}, \text{ cptr})$	2(整个范围)
$\text{asinf}(x)$	4(整个范围)

$\operatorname{acosf}(x)$	3(整个范围)
$\operatorname{atanf}(x)$	2(整个范围)
$\operatorname{atan2f}(y,x)$	3(整个范围)
$\operatorname{sinhf}(x)$	3(整个范围)
$\operatorname{coshf}(x)$	2(整个范围)
$\operatorname{tanhf}(x)$	2(整个范围)
$\operatorname{asinhf}(x)$	3(整个范围)
$\operatorname{acoshf}(x)$	4(整个范围)
$\operatorname{atanhf}(x)$	3(整个范围)
$\operatorname{powf}(x,y)$	8(整个范围)
$\operatorname{erff}(x)$	3(整个范围)
$\operatorname{erfcf}(x)$	6(整个范围)
$\operatorname{erfinvf}(x)$	3(整个范围)
$\operatorname{erfcinvf}(x)$	4(整个范围)
$\operatorname{lgammaf}(x)$	6(outside interval-10.001...-2.264;larger inside)
$\operatorname{tgammaf}(x)$	11(整个范围)
$\operatorname{fmaf}(x,y,z)$	0(整个范围)
$\operatorname{frexpf}(x,\exp)$	0(整个范围)
$\operatorname{ldexpf}(x,\exp)$	0(整个范围)
$\operatorname{scalbnf}(x,n)$	0(整个范围)
$\operatorname{scalblnf}(x,l)$	0(整个范围)
$\operatorname{logbf}(x)$	0(整个范围)
$\operatorname{ilogbf}(x)$	0(整个范围)
$\operatorname{j0f}(x)$	$ x  < 8$ 时, 9 否则, 最大绝对误差为 $2.2 \times 10^{-6}$
$\operatorname{j1f}(x)$	$ x  < 8$ 时, 9 否则, 最大绝对误差为 $2.2 \times 10^{-6}$
$\operatorname{jnf}(x)$	如果 $n=128$ , 最大绝对误差为 $2.2 \times 10^{-6}$
$\operatorname{y0f}(x)$	$ x  < 8$ 时, 9 否则, 最大绝对误差为 $2.2 \times 10^{-6}$

y1f(x)	$ x  < 8$ 时, 9 否则, 最大绝对误差为 $2.2x10^{-6}$
ynf(x)	$ x  < 8$ 时, 9 否则, 最大绝对误差为 $2.2x10^{-6}$
fmodf(x,y)	0(整个范围)
remainderf(x,y)	0(整个范围)
remquof(x,y,iptr)	0(整个范围)
modff(x,iptr)	0(整个范围)
fdimf(x,y)	0(整个范围)
truncf(x)	0(整个范围)
roundf(x)	0(整个范围)
rintf(x)	0(整个范围)
nearbyintf(x)	0(整个范围)
ceilf(x)	0(整个范围)
floorf(x)	0(整个范围)
lrintf(x)	0(整个范围)
lroundf(x)	0(整个范围)
llrintf(x)	0(整个范围)
llroundf(x)	0(整个范围)

### C.1.2 双精度浮点函数

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

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



表 C.2: 数学标准库函数及其最大ULP 错误最大错误表示为正确舍入的双精度结果和CUDA 库函数返回的结果之差(以ulp 计算)的绝对值

函数	最大ulp错误
$x+y$	0(IEEE-754舍入到最近的偶数)
$x*y$	0(IEEE-754舍入到最近的偶数)
$x/y$	0(IEEE-754舍入到最近的偶数)
$1/x$	0(IEEE-754舍入到最近的偶数)
$\text{sqrt}(x)$	0 (IEEE-754舍入到最近的偶数)
$\text{rsqrt}(x)$	1(整个范围)
$\text{cbrt}(x)$	1(整个范围)
$\text{rcbrt}(x)$	1(整个范围)
$\text{hypot}(x,y)$	2(整个范围)
$\text{exp}(x)$	1(整个范围)
$\text{exp2}(x)$	1(整个范围)
$\text{exp10}(x)$	1(整个范围)
$\text{expm1}(x)$	1(整个范围)
$\log(x)$	1(整个范围)
$\log2(x)$	1(整个范围)
$\log10(x)$	1(整个范围)
$\log1px(x)$	1(整个范围)
$\sin(x)$	2(整个范围)
$\cos(x)$	2(整个范围)
$\tan(x)$	2(整个范围)
$\text{sincos}(x, \text{sptr}, \text{cptr})$	2(整个范围)
$\text{sinpi}(x)$	2(整个范围)
$\text{cospi}(x)$	2(整个范围)
$\text{sincospi}(x, \text{sptr}, \text{cptr})$	1(整个范围)
$\text{asin}(x)$	2(整个范围)

$\text{acos}(x)$	2(整个范围)
$\text{atan}(x)$	2(整个范围)
$\text{atan2}(y,x)$	2(整个范围)
$\text{sinh}(x)$	1(整个范围)
$\text{cosh}(x)$	1(整个范围)
$\text{tanh}(x)$	1(整个范围)
$\text{asinh}(x)$	2(整个范围)
$\text{acosh}(x)$	2(整个范围)
$\text{atanh}(x)$	2(整个范围)
$\text{pow}(x,y)$	2(整个范围)
$\text{erf}(x)$	2(整个范围)
$\text{erfc}(x)$	4(整个范围)
$\text{erfinv}(x)$	5(整个范围)
$\text{erfcinv}(x)$	8(整个范围)
$\text{lgamma}(x)$	4(outside interval-11.0001...-2.2637;larger inside)
$\text{tgamma}(x)$	8(整个范围)
$\text{fma}(x,y,z)$	0(IEEE-754 传入到最近的偶数)
$\text{frexp}(x,\text{exp})$	0(整个范围)
$\text{ldexp}(x,\text{exp})$	0(整个范围)
$\text{scalbn}(x,n)$	0(整个范围)
$\text{scalbln}(x,l)$	0(整个范围)
$\text{logb}(x)$	0(整个范围)
$\text{ilogb}(x)$	0(整个范围)
$\text{j0}(x)$	$ x  < 8$ 时, 7 否则, 最大绝对误差为 $5x10^{-12}$
$\text{j1}(x)$	$ x  < 8$ 时, 7 否则, 最大绝对误差为 $5x10^{-12}$
$\text{jn}(x)$	如果 $n=128$ , 最大绝对误差为 $5x10^{-12}$
$\text{y0}(x)$	$ x  < 8$ 时, 7 否则, 最大绝对误差为 $5x10^{-12}$

y1(x)	$ x  < 8$ 时, 7 否则, 最大绝对误差为 $5x10^{-12}$
yn(x)	$ x  < 8$ 时, 7 否则, 最大绝对误差为 $5x10^{-12}$
fmod(x,y)	0(整个范围)
remainder(x,y)	0(整个范围)
remquo(x,y,iptr)	0(整个范围)
mod(x,iptr)	0(整个范围)
fdim(x,y)	0(整个范围)
trunc(x)	0(整个范围)
round(x)	0(整个范围)
rint(x)	0(整个范围)
nearbyint(x)	0(整个范围)
ceil(x)	0(整个范围)
floor(x)	0(整个范围)
lrint(x)	0(整个范围)
lround(x)	0(整个范围)
llrint(x)	0(整个范围)
llround(x)	0(整个范围)

## C.2 内置函数

这一节列举了仅在设备代码中支持的内置函数。

这些函数中包括C.1所列函数的精确度较低但速度更快的版本；它们具有相同的名称，另外加上`_`前缀（如`_sinf(x)`）。编译器有个选项（`-use-fast-math`）强制C.3的每个函数编译成其对应的内置版本。内置函数除了降低了受影响函数的精度，也可能在某些特殊情况的处理上不同。一种更健壮的方法是在性能提升和性质改变（如精度降低和特殊发问处理）可以容忍的情况下，选择性的将部分函数改成内置版本。

使用`_rn`前缀的函数将使用就近舍入到偶数模式操作；

使用`_rz`前缀的函数将使用向零舍入模式操作；

表 C.3: 受-use-fast-math影响的函数

运算符/函数	设备函数
$x/y$	<code>__fdivdef(x,y)</code>
<code>sinf(x)</code>	<code>__sinf(x)</code>
<code>cosf(x)</code>	<code>__cosf(x)</code>
<code>tanf(x)</code>	<code>__tanf(x)</code>
<code>sincosf(x,sptr, cptr)</code>	<code>__sincosf(x,sptr,cptr)</code>
<code>logf(x)</code>	<code>__logf(x)</code>
<code>log2f(x)</code>	<code>__log2f(x)</code>
<code>log10f(x)</code>	<code>__log10f(x)</code>
<code>expf(x)</code>	<code>__expf(x)</code>
<code>exp10f(x)</code>	<code>__exp10f(x)</code>
<code>powf(x,y)</code>	<code>__powf(x,y)</code>

使用`_ru` 前缀的函数将使用上舍入（向正无穷大）模式；

使用`_rd` 前缀的函数将使用下舍入（向负无穷大）模式。

### C.2.1 单精度浮点函数

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

浮点除法的精度依赖于设备的计算能力和代码是以`-prec-div=false`还是`-prec-div=true`编译。当代码以`-prec-div=false`编译时，普通浮点除法和`_fdivdef(x, y)` 具有相同的精确度，但在 $2^{126} < y < 2^{128}$  时，`_fdivdef(x, y)` 的结果为0，而普通除法将在C.4列举的精确度内提供正确的结果。同样，在 $2^{126} < y < 2^{128}$  时，如果 $x$  是无穷大，`_fdivdef(x, y)` 将得到结果NaN（无穷大乘以0 的结果），而普通除法将返回无穷大。另一方面，以`-prec-div=true`编译或者不加此选项，在计算能力2.x及以上的设备上，普通除法是符合IEEE标准的。

### C.2.2 双精度浮点函数

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

表 C.4: CUDA 运行时库支持的单精度浮点内部函数及其误差范围

函数	误差范围
<code>__fadd_[rn,rz,ru,rd](x,y)</code>	符合IEEE
<code>__fmul_[rn,rz,ru,rd](x,y)</code>	符合IEEE
<code>__fmaf_[rn,rz,ru,rd](x,y,z)</code>	符合IEEE
<code>__frcp_[rn,rz,ru,rd](x)</code>	符合IEEE
<code>__fsqrt_[rn,rz,ru,rd](x)</code>	符合IEEE
<code>__fdiv_[rn,rz,ru,rd](x,y)</code>	符合IEEE
<code>__fdividef(x,y)</code>	如果y 在 $[2^{-126}, 2^{126}]$ 区间内, 则最大ulp 误差为2.
<code>__expf(x)</code>	最大ulp 误差为 $2 + \text{floor}(\text{abs}(1.16 * x))$ 。
<code>__exp10f(x)</code>	最大ulp 误差为 $2 + \text{floor}(\text{abs}(2.95 * x))$ 。
<code>__logf(x)</code>	如果x 在 $[0.5, 2]$ 区间内, 则最大绝对误差为 $2^{-21.41}$ , 否则最大ulp 误差为3。
<code>__log2f(x)</code>	如果x 在 $[0.5, 2]$ 区间内, 则最大绝对误差为 $2^{-22}$ , 否则最大ulp 误差为2。
<code>__log10f(x)</code>	如果x 在 $[0.5, 2]$ 区间内, 则最大绝对误差为 $2^{-24}$ , 否则最大ulp 误差为3。
<code>__sinf(x)</code>	如果x 在 $[-\pi, \pi]$ 区间内, 则最大绝对误差为 $2^{-21.41}$ , 否则更大。
<code>__cosf(x)</code>	如果x 在 $[-\pi, \pi]$ 区间内, 则最大绝对误差为 $2^{-21.19}$ , 否则更大。
<code>__sincosf(x,sptr,cptr)</code>	与sinf(x) 和cosf(x) 相同。
<code>__tanf(x)</code>	继承自以下实现: $__sinf(x)x(1/__cosf(x))$ 。
<code>__powf(x, y)</code>	继承自以下实现: $exp2f(yx\_log2f(x))$ 。

表 C.5: CUDA 运行时库支持的双精度浮点内置函数及其误差范围

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

## 附录 D C++语言支持

正如3.1所描述的，使用nvcc编译的源码文件能够混合主机代码和设备代码。

对于主机代码，nvcc支持主机C++编译器支持的C++ ISO/IEC 14882:2003规范的任意部分。

对于设备代码，在D.2描述的限制下，nvcc支持D.1描述的特性；不支持运行时类型信息、异常处理和C++标准库。

### D.1 代码例子

#### D.1.1 数据类

```
class PixelRGBA {
public:
    __device__ PixelRGBA(): r_(0), g_(0), b_(0), a_(0) { }

    __device__ PixelRGBA(unsigned char r, unsigned char g,
                          unsigned char b, unsigned char a = 255):
        r_(r), g_(g), b_(b), a_(a) { }

private:
    unsigned char r_, g_, b_, a_;

    friend PixelRGBA operator+(const PixelRGBA const PixelRGBA
                               &);
};

__device__
PixelRGBA operator+(const PixelRGBA& p1, const PixelRGBA& p2)
{
```

```

    return PixelRGBA(p1.r_ + p2.r_, p1.g_ + p2.g_,
                    p1.b_ + p2.b_, p1.a_ + p2.a_);
}

__device__ void func(void)
{
    PixelRGBA p1, p2;
    // ... // Initialization of p1 and p2 here
    PixelRGBA p3 = p1 + p2;
}

```

### D.1.2 派生类

```

__device__ void* operator new(size_t bytes, MemoryPool& p);
__device__ void operator delete(void*, MemoryPool& p);
class Shape {
public:
    __device__ Shape(void) { }
    __device__ void putThis(PrintBuffer *p) const;
    __device__ virtual void Draw(PrintBuffer *p) const {
        p->put("Shapeless");
    }
    __device__ virtual ~Shape() {}
};
class Point : public Shape {
public:
    __device__ Point() : x(0), y(0) {}
    __device__ Point(int ix, int iy) : x(ix), y(iy) { }
    __device__ void PutCoord(PrintBuffer *p) const;
    __device__ void Draw(PrintBuffer *p) const;
    __device__ ~Point() {}
private:

```



```
    int x, y;
};
__device__ Shape* GetPointObj(MemoryPool& pool)
{
    Shape* shape = new(pool) Point(rand(-20,10), rand(-100,-20));
    return shape;
}
```

### D.1.3 类模板

```
template <class T>
class myValues {
    T values[MAX_VALUES];
public:
    __device__ myValues(T clear) { ... }
    __device__ void setValue(int Idx, T value) { ... }
    __device__ void putToMemory(T* valueLocation) { ... }
};

template <class T>
void __global__ useValues(T* memoryBuffer) {
    myValues<T> myLocation(0);
    ...
}

__device__ void* buffer;

int main()
{
    ...
    useValues<int><<<blocks, threads>>>(buffer);
    ...
}
```

```
}
```

#### D.1.4 函数模板

```
template <typename T>
__device__ bool func(T x)
{
    ...
    return (...);
}

template <>
__device__ bool func<int>(T x) // Specialization
{
    return true;
}

// Explicit argument specification
bool result = func<double>(0.5);

// Implicit argument deduction
int x = 1;
bool result = func(x);
```

#### D.1.5 函子类

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

```
};

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

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

// Host code
int main()
{
    ...
    VectorOperation<<<blocks, threads>>>(v1, v2, v3, N, Add());
    ...
}
```

## D.2 限制

### D.2.1 预处理符号

如果`__global__`函数模板从主机实例化和启动，无论`__CUDA_ARCH__`宏是否存在或者取何值，函数模板必须使用相同的类型实例化。

下面的例子中，`kern < int >`只在`__CUDA_ARCH__`没有定义的情况下实例化，这不被支持。

```
__device__ int result;
template <typename T>
__global__ void kern(T in)
{
    result = in;
}

__host__ __device__ void foo(void)
{
    #if !defined(__CUDA_ARCH__)
        kern<<<1,1>>>(1); // instantiation "kern<int>"
    #endif
}

int main(void)
{
    foo();
    cudaDeviceSynchronize();
    return 0;
}
```

### D.2.2 限定符

#### D.2.2.1 设备存储器限定符

`__device__`、`__shared__`和`__constant__`限定符不允许：

- 类、结构体和联合成员
- 形式参数
- 主机上执行函数的局部变量

`__shared__` 和 `__constant__` 变量具有隐含的静态存储。

`__device__` 和 `__constant__` 变量仅允许在命名空间作用域内定义（包括全局命名空间）。

命名空间作用域内定义的类型为 `__device__`、`__shared__` 和 `__constant__` 的变量不能有非空的构造器和析构器。类的构造器是一个无关紧要的构造器，或者满足下面的所有条件时，在编译单元的某个时间点被认为是空的：

- 构造器函数已定义。
- 构造器函数没有参数，初始化列表为空，函数体为空。
- 类没有虚函数或虚基类。
- 所有基类的默认构造器可认为为空。
- 类的所有非静态类类型的数据成员或数组的默认构造器可被认为是空的。

类的析构器是一个无关紧要的构造器，或者满足下面的所有条件时，在编译单元的某个时间点被认为是空的：

- 析构器已定义。
- 析构器函数体没有语句。
- 类没有虚函数和虚基类。
- 所有基类的析构器可被认为是空。
- 类的所有非静态类类型数据成员或数组的默认析构器可被认为是空的。

如果以全程序编译模式编译时（参见nvcc用户手册），`__device__`、`__shared__`和`__constant__` 变量不能使用`extern` 关键字定义为外部变量。唯一的例外是如3.2.3描述的动态分配的共享存储器。

如果以分离编译模式编译（参见nvcc用户手册），`__device__`、`__shared__`和`__constant__` 变量能够使用`extern` 关键字定义为外部变量。当nvlink找不到外部变量的定义时，会产生错误（除非是动态分配的共享存储器）。

### D.2.2.2 Volatile限定符

只有在执行`__threadfence_block()`、`__threadfence()`或`__syncthreads()`之后（参见B.5和B.6节），在此之前对全局存储器或共享存储器写入的才能保证对其它线程可见。只要满足这个要求，编译器可任意优化对全局存储器或共享存储器的读写。

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

### D.2.3 指针

在计算能力1.x的设备上，只要编译器能够确定在设备上执行的代码中的指针指向的是共享存储器空间、全局存储器空间或本地存储器，此类指针即受支持，否则将仅限于指向在全局存储器空间中分配或声明的存储器。在计算能力2.0及以下的设备上，指针的支持没有限制。

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

通过取`__device__`、`__shared__` 或`__constant__` 变量的地址而获得的地址仅可在设备代码中使用。通过`cudaGetSymbolAddress()`（参见3.2.2）获取的`__device__` 或`__constant__` 变量的地址仅可在主机代码中使用。

由于使用了C++的语法规则，`void`指针（如`malloc()`函数返回）必须要转型才能赋值给非`void`指针。

## D.2.4 运算符

### D.2.4.1 赋值运算符

不能在设备端给 `__constant__` 变量赋值，仅可通过主机运行时函数从主机赋值（参见3.2.2）。

`__shared__` 变量的声明中不可包含初始化。

不允许为B.3定义的任何内置变量赋值。

### D.2.4.2 地址运算符

不允许取B.3定义的任何内置变量的地址。

## D.2.5 函数

### D.2.5.1 编译器生成的函数

编译器生成函数的执行空间限定符（`__host__`，`__device__`）是所有调用此函数的函数的限定符的并集（注意，在此分析中，`__global__`调用函数都被认为是`__device__`调用者）。例如：

```
class Base {
    int x;
public:
    __host__ __device__ Base(void) : x(10) {}
};

class Derived : public Base {
    int y;
};

class Other: public Base {
    int z;
};

__device__ void foo(void)
```

```
{
    Derived D1;
    Other D2;
}

__host__ void bar(void)
{
    Other D3;
}
```

这里，编译器生成的构造器函数”Derived::Derived”将被视为为\_\_device\_\_函数，因为它只被\_\_device\_\_函数”foo”调用。编译器生成的构造器函数”Other::Other”将被视为”\_\_device\_\_ \_\_host\_\_函数，因为它同时被\_\_device\_\_函数”foo”和\_\_host\_\_函数”bar”调用。

#### D.2.5.2 函数参数

\_\_global\_\_ 函数参数将传递给设备：

- 计算能力1.x的使用共享存储器传递，且大小限制为256 字节，
- 计算能力2.x及以上的设备上的通过常量存储器传递，且其大小限制为4K字节。

\_\_device\_\_ 和\_\_global\_\_ 函数的函数体内无法声明静态变量。

#### D.2.5.3 函数内静态变量

不能在\_\_device\_\_和\_\_global\_\_函数体内声明静态变量。

#### D.2.5.4 函数指针

主机代码支持指向\_\_global\_\_函数的函数指针，但是设备代码不支持。

指向\_\_device\_\_函数的函数指针只在计算能力2.x及以上的设备上得到支持。

不允许在主机代码中取\_\_device\_\_ 函数地址。



### D.2.5.5 函数递归

`__global__` 函数不支持递归。

`__device__` 函数只在为计算能力2.x及以上的设备编译的设备代码上支持递归。

## D.2.6 类

### D.2.6.1 数据成员

不支持静态数据成员。

设备代码中位域的层次目前和Windows上的主机代码的层次不一致。

### D.2.6.2 函数成员

`__global__` 函数不能做静态成员函数

### D.2.6.3 虚函数

当子类覆盖父类的虚函数时，执行空间限定符必须匹配。

不允许将有虚函数的类的对象作为参数传给 `__global__` 函数。

虚函数表由编译器存放在全局存储器或者常量存储器中。

### D.2.6.4 虚基类

不允许将虚基类派生类的对象作为参数传给 `__global__` 函数。

### D.2.6.5 Windows相关

在Windows上，对于类类型T的一个C++对象，满足下列条件的任意一种，CUDA编译器可能产生和微软编译器不同的存储器层次。

- T 有虚函数或直接或间接的派生自有虚函数的类；
- T有直接或间接的虚基类；
- T直接或间接的多继承自多个空基类。

在主机和设备代码中，这些对象的尺寸可能也不相同。只要在主机代码和设备代码中互斥的使用T，程序就可正确工作。不要在主机和设备代码间传递类型T的对象（如作为 `__global__` 函数的参数或通过 `cudaMemcpy*()` 调用）。

### D.2.7 模板

`__global__` 函数模板不能够实例化为在函数内定义或是类或结构体的私有成员的类型或类型重定义（`typedef`），如下面的样本代码所示。

```
template <typename T>
__global__ void myKernel1(void) { }

template <typename T>
__global__ void myKernel2(T par) { }

class myClass {
private:
    struct inner_t { };
public:
    static void launch(void)
    {
        // Both kernel launches below are disallowed
        // as myKernel1 and myKernel2 are instantiated
        // with private type inner_t

        myKernel1<inner_t><<<1,1>>>();

        inner_t var;
        myKernel2<<<1,1>>>(var);
    }
};
```

## 附录 E 纹理获取

本附录提供了用于根据纹理引用(参见3.2.8.1)的不同属性计算纹理函数(参见B.8)的返回值的公式。

绑定到纹理参考的纹理表示为数组T,

- 对于一维纹理, 它有 $N$  个纹理元素
- 对于二维纹理它有 $N \times M$ 个元素,
- 对于三维纹理它有 $N \times M \times L$ 个元素。

它将使用非归一化的纹理坐标 $x$ 、 $y$  和 $z$  或归一化的纹理坐标 $x/N$ ,  $y/M$  和 $z/L$ 获取, 这在3.2.8.1中描述。本附录假设纹理坐标必须位于T 的有效寻址范围内。3.2.8.1解释了越界坐标如何依据寻址模式映射到有效寻址范围内。

### E.1 最近点取样

在这种过滤模式中, 纹理获取返回的值如下:

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

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

E.1展示了一维纹理的最近点取样( $N = 4$ )。

对于整型纹理来说, 纹理获取的返回值可重新映射到 $[0.0, 1.0]$  (参见3.2.8.1)。

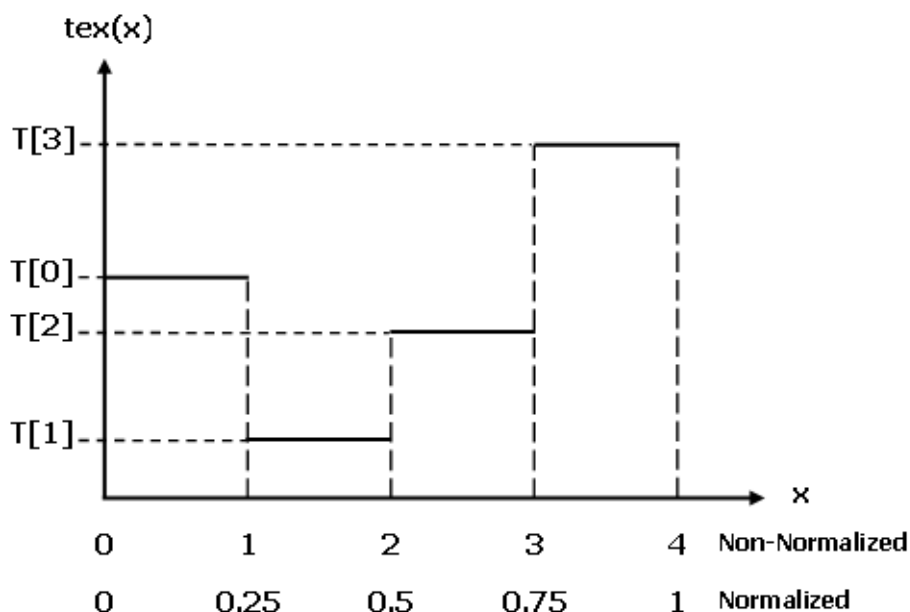


图 E.1: 4个像素的一维纹理最近点取样

## E.2 线性滤波

在这种仅对浮点纹理可用的过滤模式中，纹理获取的返回值如下：

- 对于一维纹理是  $tex(x) = (1 - \alpha)T[i] + \alpha T[i + 1]$
- 对于二维纹理是  $tex(x, y) = (1 - \alpha)(1 - \beta)T[i, j] + \alpha(1 - \beta)T[i + 1, j] + (1 - \alpha)\beta T[i, j + 1] + \alpha\beta T[i + 1, j + 1]$
- 对于三维纹理是  $tex(x, y, z) = (1 - \alpha)(1 - \beta)(1 - \gamma)T[i, j, k] + \alpha(1 - \beta)(1 - \gamma)T[i + 1, j, k] + (1 - \alpha)(1 - \beta)\gamma T[i, j, k + 1] + \alpha(1 - \beta)\gamma T[i + 1, j, k + 1] + (1 - \alpha)\beta\gamma T[i, j + 1, k + 1] + \alpha\beta\gamma T[i + 1, j + 1, k + 1]$

其中：

- $i = \text{floor}(x_B), \alpha = \text{frac}(x_B), x_B = x - 0.5;$
- $j = \text{floor}(y_B), \beta = \text{frac}(y_B), y_B = y - 0.5;$
- $k = \text{floor}(z_B), \gamma = \text{frac}(z_B), z_B = z - 0.5。$

$\alpha$ 、 $\beta$  和  $\gamma$  存储在9位的定点格式中，其中8位表示分数值（所以1.0是准确表示的）。

E.2展示了一维纹理的线性取样， $N = 4$ 。

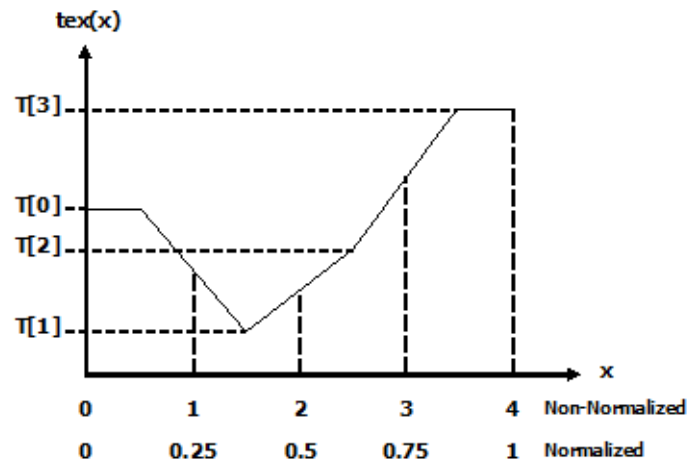


图 E.2: 4个像素、钳位寻址模式下的一维纹理线性滤波

### E.3 查找表

查找表函数 $TL(x)$ 可实现为 $TL(x) = tex((N - 1)/R)x + 0.5)$ ，这样即可确保 $TL(0) = T[0]$  且  $TL(R) = T[N - 1]$ 。其中 $x$ 处于 $[0, R]$  的范围内。

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

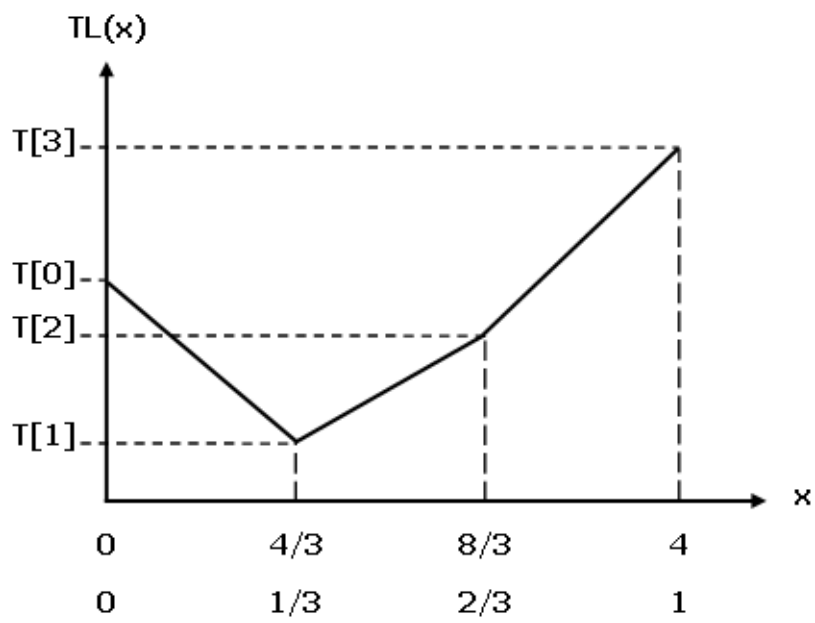


图 E.3: 使用线性滤波的一维查找表

## 附录 F 计算能力

设备的计算能力决定了它大部分规范和特性。

F.1给出了各计算能力设备的特性和技术规范。

C.1.1回顾了与IEEE浮点标准的符合程度。

F.3、F.4和F.5分别给出了计算能力1.x、2.x和3.x的设备更多架构细节。

## F.1 特性和技术规范

表 F.1: 不同计算能力支持的特性

特性支持	计算能力					
特性支持（所有计算能力都支持的特性没有列出）	1.0	1.1	1.2	1.3	2.x, 3.0	3.5
作用在全局存储器上32位整形原子函数（参见 <a href="#">B.11</a> ）	No	Yes				
作用在全局存储器上32位浮点原子函数atomicExch()（参见 <a href="#">B.11</a> ）						
作用在共享存储器上32位字的整形原子函数（参见 <a href="#">B.11</a> ）	No		Yes			
作用在共享存储器上32位浮点原子函数atomicExch()（参见 <a href="#">B.11</a> ）						

作用在全局存储器上64位字的整形原子函数（参见B.11）			
束 表 决 函 数 （参见B.12）			
双精度浮点数	No	Yes	
作用在共享存储器上的64位整型原子函数（参见B.11）	No		Yes
作用在全局和共享存储器上32位字的浮点原子加函数（参见B.11）			
__ballot()（参见B.12）			
__theadfence_system()（参见B.5）			
__syncthreads_count(), __syncthreads_and(), syncthreads_or()（）			
表面函数（）			
三维线程网格			
束洗牌函数	No		Yes

表 F.2: 依据计算能力的技术规范

	计算能力						
技术规范	1.0	1.1	1.2	1.3	2.x	3.0	3.5
线程网格的最大维数	2				3		
网格的最大x维最大尺寸	65535					$2^{31} - 1$	



网格的最大y和z维尺寸	65535			
线程块的最大维数	3			
块最大x或y维尺寸	512		1024	
块最大z维尺寸	64			
块内最大线程数	512		1024	
线程束（warp）尺寸	32			
多处理器最大常驻块数量	8			16
多处理器最大常驻线程束数量	24	32	48	64
多处理器最大常驻线程数量	768	1024	1536	2048
多处理器的32位寄存器数量	8K	16k	32k	64k
每线程的最大32位寄存器数量	128		63	255
多处理器最大共享存储器数量	16 KB		48 KB	
共享存储器存储体数量	16		32	
线程的本地存储器数量	16 KB		512 KB	
常量存储器尺寸	64 KB			
每个多处理器常量缓存数量	8 KB			
多处理器纹理缓存数量	和设备相关，在6 KB到8 KB之间			

绑定到CUDA数组的一维纹理参考最大宽度	8192	65535
绑定到线性存储器的一维纹理参考最大宽度	2 <sup>27</sup>	
一维层次纹理参考的最大宽度和层次数	8192*512	16384*2048
绑定到CUDA数组的二维纹理参考的最大宽和高	65535*32768	65535*65535
绑定到线性存储器的二维纹理参考的最大宽和高	65000*65000	65000*65000
绑定到支持纹理收集的CUDA数组的二维纹理参考的最大宽和高	N/A	16384 x 16384
二维层次纹理参考的最大宽度、高度和层次数	8192 x 8192 x 512	16384 x 16384 x 2048
绑定到CUDA数组的三维纹理参考的最大宽度，调度和深度	2048 x 2048 x 2048	4096 x 4096 x 4096
立方体图纹理参考的最大宽度（和高度）	N/A	16384

层次立方图参考的最大宽度（和高度）和层数	N/A	16384 x 2046	
一个内核可以绑定的最大纹理数目	128	256	
绑定到CUDA数组的一维表面参考的最大宽度	N/A	65535	
一维层次表面参考的最大宽度和层数		65536 x 2048	
绑定到CUDA数组的二维表面参考的最大宽度和高度		65536 x 32768	
二维层次表面参考的最大宽度，高度和层数		65536 x 32768 x 2048	
绑定到CUDA数组的三维表面参考的最大宽度，高度和深度		65536 x 32768 x 2048	
绑定到CUDA数组的立方图表面参考的最大宽度（和高度）		32768	
内核绑定的最大表面数目		8	16
内核的最大指令数量	2百万	5亿1千2百万	

## F.2 浮点标准

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

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

为了缓和这些限制，通过下列内置方式提供符合IEEE的软（因此更慢）实现（参见C.2）：

- \* `__fmaf_r[n,z,u,d](float,float,float)`：IEEE舍入模式的单精度积和乘加，
- \* `__frcp_r[n,z,u,d](float)`：IEEE舍入模式的单精度倒数，
- \* `__fdiv_r[n,z,u,d](float,float)`：IEEE舍入模式的单精度除法，
- \* `__fsqrt_r[n,z,u,d](float)`：IEEE舍入模式的单精度平方根，

- \* `_fadd_r[n,z,u,d](float,float)`: IEEE直接舍入模式的单精度加法,
- \* `_fmul_r[u,d](float,float)`: IEEE直接舍入模式的单精度乘法;
- 对于计算能力1.x的设备上的双精度浮点数:
  - \* 舍入到最近偶数是唯一支持倒数, 除法和平方根的IEEE舍入模式。

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

对于计算能力2.x及以上的设备, 代码必须使用`-ftz=false`, `-prec-div=true`和`-prec-sqrt=true`以保证符合IEEE(这是默认设置, 详细的编译选项参见nvcc用户手册); 使用`-ftz=true`, `-prec-div=false`和`-prec-sqrt=false`编译的代码更接近为计算能力1.x设备生成的代码。

加法和乘法经常被组合成一个单独的乘加指令:

- 为计算能力1.x的设备生成单精度的FMAD
- 为计算能力2.x的设备生成单精度的FFMA

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

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

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

IEEE-754没有定义整数除以0和整数上溢时的行为。对于计算设备来说, 没有机制检测这些异常的发生。整数除以0会产生一个不确定的, 设备特定的值。

[这里](#)包含了更多NVIDIA GPU的浮点精确度的信息。

## F.3 计算能力1.x

### F.3.1 架构

对于计算能力1.x，一个多处理器包含：

- 8个CUDA核心用于算术操作，
- 1个双精度浮点操作单元用于双精度浮点算术操作（计算能力1.3），
- 2个特殊函数单元用于单精度浮点超越函数（这些单元同时也处理单精度浮点乘法），
- 1个束调度器。

为了为束内所有线程执行一条指令，束调度器发射指令必须花费：

- 为整数和单精度浮点算术指令4个时钟周期，
- 为双精度浮点算术指令32个时钟周期(这只对计算能力1.3的设备)，
- 为单精度超越指令16个时钟周期。

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

多处理器组成纹理处理集群（Texture Processor Cluster，TPC）。每个TPC中多处理器数目是：

- 对于计算能力1.0和1.1的设备是2，
- 对于计算能力1.2和1.3的设备是3。

每个TPC有一个被所有多处理器共享的只读纹理缓存，加速来自纹理存储器空间的读操作，纹理存储器在设备存储器中。每个多处理器通过纹理单元来访问纹理缓存，纹理单元实现了3.2.8提到的多种寻址模式和数据滤波。

全局存储器和本地存储器在设备存储器中且没有缓存。

### F.3.2 全局存储器

来自一个束的一个全局存储器请求被分成两个存储器请求，每个对应半束，独立发射。[F.3.2.1](#)和[F.3.2.2](#)描述了半束线程的存储器访问如何被合并成一次或多次存储器事务，这依赖于设备计算能力。[F.1](#)显示了一些基于计算能力的全局存储器访问和对应存储器事务的例子。

最终的存储器事务贡献了存储器吞吐量。

#### F.3.2.1 计算能力1.0和1.1的设备

为了合并访问，半束的存储器请求必须满足下列条件：

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

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

如果半束不满足这些条件，16个单独的32位存储器事务被发射。

#### F.3.2.2 计算能力1.2和1.3的设备

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

更准确地，使用下面的协议决定半束内线程必要的存储器事务：

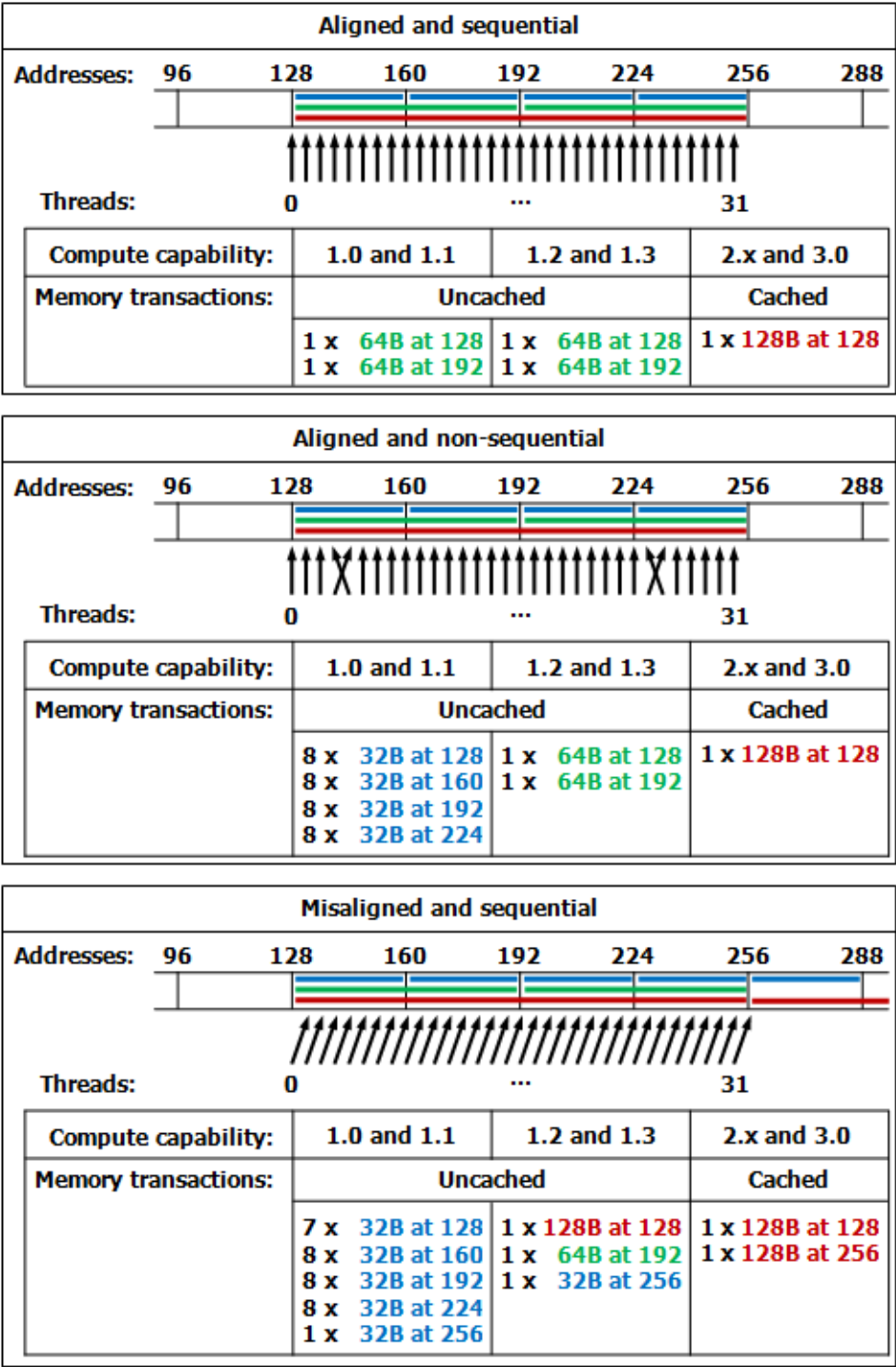


图 F.1: 一个线程束访问全局存储器的例子，每个线程4字节和相关的基于计算能力的存储器事务



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

### F.3.3 共享存储器

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

束的一次共享存储器访问被分成两次存储器访问，每次为半个线程束（half warp）发射指令，两个半束之间是相互独立的串行发射，因此前半束的线程和后半束的线程不可能出现存储体冲突）。

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

#### F.3.3.1 32位步长访问

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

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

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

[F.2](#)为计算能力3.0的设备展示的一些按步长访问的例子。这些例子对于计算能力1.x的设备同样有效，但是存储体数目是16而不是32。另外对于计算能力1.x的设备中间例子的访问模式会产生2路冲突。

### F.3.3.2 32位广播访问

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

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

在每个周期内，那个字被选为广播字和为每个存储体选择的地址没有定义。

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

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

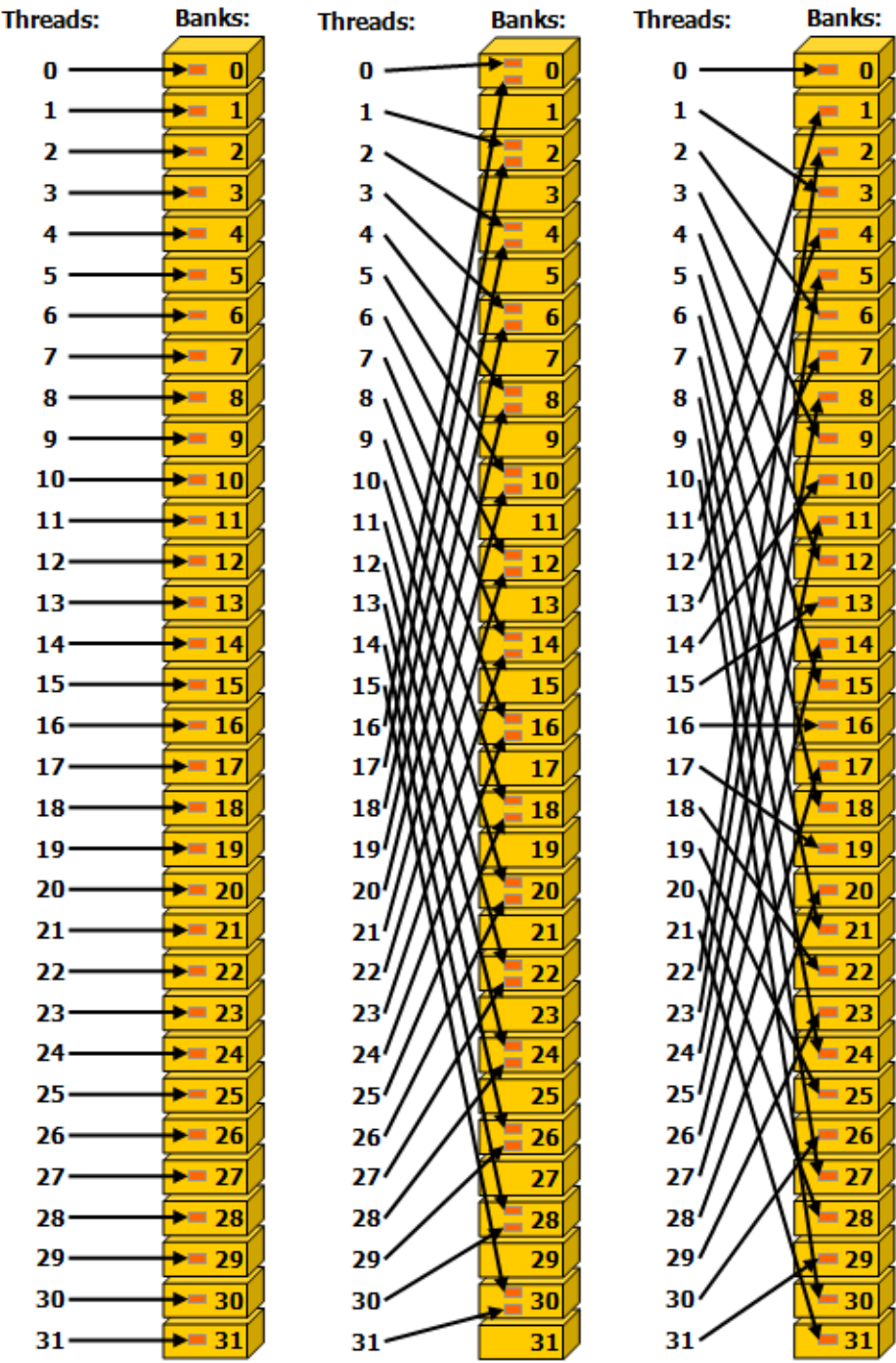


图 F.2: 按步长访问共享存储器

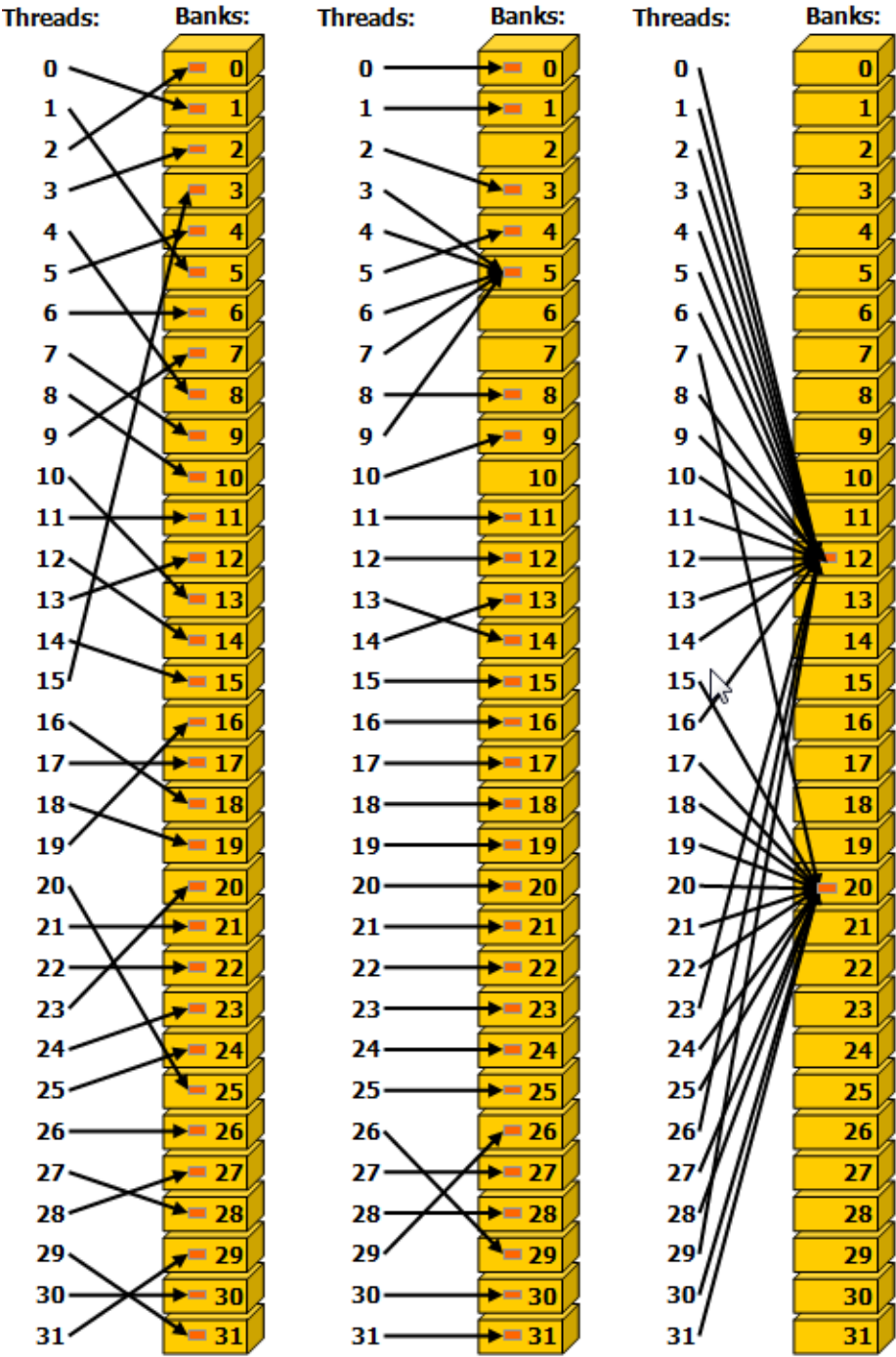


图 F.3: 不规则的共享存储器访问

### F.3.3.3 8位和16位访问

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

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

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

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

### F.3.3.4 大于32位访问

每个线程大于32位访问会被拆成32位的访问，这典型的会产生存储体冲突。

例如，如下的对double数组的访问会产生2路存储体冲突：

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

由于存储器请求被编译成2个独立的步长为2的32位访问。一种避免存储体冲突的方式是将double操作数拆成两个，就像下面的代码一样：

```
__shared__ int shared_lo [32];
__shared__ int shared_hi [32];

double dataIn;
shared_lo[BaseIndex + tid] = __double2loint(dataIn);
shared_hi[BaseIndex + tid] = __double2hiint(dataIn);

double dataOut =
    _hiloint2double(shared_hi[BaseIndex + tid],
                    shared_lo[BaseIndex + tid]);
```

这可能并不能提高性能，在计算能力2.x及以上的设备上这确实会降低性能。

对于结构体同样有效。如下面的代码：

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

导致：

- 三次独立的没有存储体冲突的访问，如果type定义为：

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

因为每个成员以奇数步长被访问，步长是3个32位字：

- 两个独立的有存储体冲突的访问，如果type定义为

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

因为每个成员以偶数步长被访问，步长是2个32位字。

## F.4 计算能力2.x

### F.4.1 架构

对于计算能力2.x的设备，一个多处理器包含：

- 对于计算能力2.0的设备：
  - 32个CUDA核心用于整形和浮点算术操作，
  - 4个特殊函数单元用于单精度浮点超越函数，
- 对于计算能力2.1的设备：

- 48个CUDA核心用于整形和浮点算术操作，
- 8个特殊函数单元用于单精度浮点超越函数，
- 2个线程束调度器。在每个指令发射时间，每一个束调度器发射：
  - 为计算能力2.0的设备发射一条指令，
  - 为计算能力2.1的设备发射两条指令。

如果存在已准备好执行的线程束，第一个束调度器负责奇数ID的束且第二个束调度器负责偶数ID的束。唯一的例外是一个束调度器发射一个双精度指令，此时，另一个调度器不能发射任何指令。

一个线程束调度器只能为多处理器（multiprocessor）内的一半CUDA核心（CUDA Core）服务。为了为束内所有线程执行一条指令，束调度器必须为整数或浮点算术指令在2个时钟周期发射指令。

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

每个多处理器有一级缓存，所有多处理器共享二级缓存，二者都用于缓存全局或本地的访问，包括临时寄存器溢出。缓存行为（如读是缓存在L1和L2还是只缓存在L2）能够基于访问使用读写指令修饰符部分配置。

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

```
// Device code
__global__ void MyKernel(int* foo, int* bar, int a)
{
    ...
}

// Host code

// Runtime API
```

```
// cudaFuncCachePreferShared: shared memory is 48 KB
// cudaFuncCachePreferL1: shared memory is 16 KB
// cudaFuncCachePreferNone: no preference
cudaFuncSetCacheConfig(MyKernel, cudaFuncCachePreferShared)

// Or via a function pointer:
void (*funcPtr)(int *, int *, int);
funcPtr = MyKernel;
cudaFuncSetCacheConfig(*funcPtr, cudaFuncCachePreferShared);
```

默认缓存配置没有偏好，如果内核配置为没有偏好，由当前线程/上下文的配置决定。当前线程/上下文的配置可用`cudaDeviceSetCacheConfig()/cuCtxSetCacheConfig()`设置（详见参考手册）。如果它们也没有偏好（采用默认设置），此时选择最近使用最多的配置，除非需要改变缓存配置以启动内核（如由于共享存储器的要求）。初始配置是48KB共享存储器和16KB 一级缓存。

应用可能查询二级缓存的大小，这可检查设备的`l2CacheSize`属性（参见3.2.6）。二级缓存最大为768KB。

多处理器组成图形处理器群（GPC）。每个GPC包含4个多处理器。

每个多处理器有一个只读纹理缓存以加速读取纹理存储器空间，纹理存储器在设备存储器上。通过纹理单元访问纹理缓存，纹理单元实现了多种寻址模式和数据滤波，这些在3.2.8说明。

#### F.4.2 全局存储器

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

缓存线是128字节且映射到设备存储器中一个128字节对齐的段。缓存到一级和二级缓存的存储器访问使用128字节的存储器事务而只缓存到二级缓存的存储器访问使用32字节的存储器事务。万一存储器访问分散，只缓存到二级缓存能够减少过度读取。

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



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

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

注意线程能够以任何顺序访问任何字，包括同一个字。

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

[F.1](#)展示了一些基于计算能力的全局存储器访问的例子。

### F.4.3 共享存储器

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

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

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

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

同样，和计算能力1.x的设备不同的有，束的前一半和后一半的线程访问共享存储器可能存在存储体冲突。

#### F.4.3.1 32位步长访问

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

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

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

[F.2](#)展示计算能力3.x的设备上，以某些步长访问共享存储器的例子。它们同样适用于计算能力2.x。但是对于计算能力2.x的设备，中间例子的访问模式会产生2路存储器冲突。

#### F.4.3.2 大于32位访问

64位和128位访问被特殊处理以最小化存储体冲突，细节如下。

其它大于32位的访问被拆成32位，64位或128位访问。下面的代码：

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

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

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

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

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

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

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

#### F.4.4 常量存储器

除了常量存储器空间得到所有计算能力设备的支持外（`__constant__`声明的变量存储位置），计算能力2.x的设备支持LDU指令，编译器使用LDU指令装载变量：

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

### F.5 计算能力3.x

#### F.5.1 架构

一个多处理器包含：

- 192个用于算术操作的CUDA核，
- 32个用于单精度浮点超越函数的特殊函数单元，
- 4个束调度器。

当多处理器得到束执行时，它首先将束分发到4个调度器上。然后，在每个指令发射期，每个调度器发射两条独立指令到分配给它且准备好执行的束。

流多处理器有一个由各功能单元共享的只读常量缓存，该缓存加速来自常量存储器空间的读，常量存储器空间存在于设备存储器中。

每个多处理器有一级缓存，所有多处理器共享二级缓存，二者都用于缓存全局或本地存储器的访问，包括临时寄存器溢出。缓存行为（如读是缓存在L1和L2还是只缓存在L2）能够基于访问使用读写指令修饰符部分配置。

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

```
// Device code
__global__ void MyKernel()
{
    ...
}

// Host code

// Runtime API
// cudaFuncCachePreferShared: shared memory is 48 KB
// cudaFuncCachePreferEqual: shared memory is 32 KB
// cudaFuncCachePreferL1: shared memory is 16 KB
// cudaFuncCachePreferNone: no preference
cudaFuncSetCacheConfig(MyKernel, cudaFuncCachePreferShared)
```

默认缓存配置没有偏好，如果内核配置为没有偏好，由当前线程/上下文的配置决定。当前线程/上下文的配置可用`cudaDeviceSetCacheConfig()/cuCtxSetCacheConfig()`设置（详见参考手册）。如果它们也没有偏好（采用默认设置），此时选择最近使用最多的配置，除非需要改变缓存配置以启动内核（如由于共享存储器的要求）。初始配置是48KB共享存储器和16KB 一级缓存。

应用可能查询二级缓存的大小，这可检查设备的`l2CacheSize`属性（参见3.2.6）。二级缓存最大为1.5MB。

多处理器组成图形处理器群（GPC）。每个GPC包含3个多处理器。

每个多处理器有一个48KB的只读数据缓存以加速读取纹理存储器空间。多处理器或直接访问（只适用于计算能力3.5的设备），或通过纹理单元。纹理单元实现了多种寻址模式和数据滤波，这些在3.2.8说明。当通过纹理单元访问时，这只读数据缓存也被称为纹理缓存。

### F.5.2 全局存储器访问

对于计算能力3.x的设备来说，全局存储器访问只被缓存到二级缓存，也有可能被缓存到只读数据缓存；但不会缓存到一级缓存。

缓存到二级缓存的行为和计算能力2.x的设备一致（参见F.4.2）。

编译器确定某个给定的全局存储器读是否缓存到只读数据缓存。要使全局存储器读被缓存到只读数据缓存，要求数据必须是只读的。为了允许编译器确定条件满足，用于加载数据的指针应该使用`const __restrict__`修饰。

F.1基于计算能力展示了一些全局存储器访问的例子。

### F.5.3 共享存储器

共享存储器有32个存储体，两种寻址模式，如下所述。

寻址模式可能使用`cudaDeviceGetSharedMemConfig()`查询，还可通过`cudaDeviceSetSharedMemConfig()`设置。每个存储器的带宽是每个时钟64位。

F.2展示了一些按步长访问的例子。

F.3展示了一些有关广播机制的共享存储器读访问。

#### F.5.3.1 64位模式

相邻的64位字映射到相邻的存储体。

一个束的两个线程访问共享存储器64位字的任何子字都不会产生存储器冲突。在这种情况下，对于读，64位字会被广播到发起访存请求的线程；对于写，每一个子字都会只会被一个线程写（具体由那个线程写，不确定）。

这种模式，相同的64位访问模式相比计算能力2.x的设备会产生更少的存储体冲突，对于32位访问，相同和更少。

#### F.5.3.2 32位模式

相邻的32位字映射到相邻的存储体。

一个束的两个线程访问共享存储器32位字的任何子字都不会产生存储器冲突，或者访问地址在同一个对齐的64个字的段（即段的首索引是64的倍数）中的两个32位字。在这种情况下，对于读，32位字会被广播到发起访存请求的线程；对于写，每一个子字都会只会被一个线程写（具体由那个线程写，不确定）。

这种模式，相同的64位访问模式相比计算能力2.x的设备会产生相同或更少的存储体冲突。



## 附录 G 驱动API

本附录假设你了解[3.2](#)的概念。

驱动API的实现在nvcuda动态库中，其所有的入口点前缀为cu，在设备驱动安装时，nvcuda会被拷贝到系统中。

驱动API是基于句柄的，命令式的：大多数对象通过不透明的句柄引用，函数通过句柄操作对象。

驱动API可用的对象总结如[G.1](#)

表 G.1: CUDA驱动API中可用的对象

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

在调用任何其它驱动API前必须用cuInit()初始化驱动API。然后必须创建一个CUDA上下文，该上下文关联到特定设备并成为主机线程的当前上下文，详见[G.1](#)。

在CUDA上下文中，内核必须显式的作为PTX或二进制对象被主机代码加载，参见[G.2](#)。用C写的内核必须独立的编译成PTX或二进制对象。发射内核使用的API入口点可参见[G.3](#)。

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

下面是2.1节例子的驱动API实现:

```
int main()
{
    int N = ...;
    size_t size = N * sizeof( float );

    // Allocate input vectors h_A and h_B in host memory
    float * h_A = (float*)malloc(size);
    float * h_B = (float*)malloc(size);

    // Initialize input vectors
    ...

    // Initialize
    cuInit(0);

    // Get number of devices supporting CUDA
    int deviceCount = 0;
    cuDeviceGetCount(&deviceCount);
    if (deviceCount == 0) {
        printf("There is no device supporting CUDA.\n");
        exit (0);
    }

    // Get handle for device 0
    CUdevice cuDevice;
    cuDeviceGet(&cuDevice, 0);

    // Create context
    CUcontext cuContext;
    cuCtxCreate(&cuContext, 0, cuDevice);
```



```
// Create module from binary file
CUmodule cuModule;
cuModuleLoad(&cuModule, "VecAdd.ptx");

// Allocate vectors in device memory
CUdeviceptr d_A;
cuMemAlloc(&d_A, size);
CUdeviceptr d_B;
cuMemAlloc(&d_B, size);
CUdeviceptr d_C;
cuMemAlloc(&d_C, size);

// Copy vectors from host memory to device memory
cuMemcpyHtoD(d_A, h_A, size);
cuMemcpyHtoD(d_B, h_B, size);

// Get function handle from module
CUfunction vecAdd;
cuModuleGetFunction(&vecAdd, cuModule, "VecAdd");

// Invoke kernel
int threadsPerBlock = 256;
int blocksPerGrid =
    (N + threadsPerBlock - 1) / threadsPerBlock;
void* args[] = { &d_A, &d_B, &d_C, &N };
cuLaunchKernel(vecAdd,
               blocksPerGrid, 1, 1, threadsPerBlock, 1, 1,
               0, 0, args, 0);

...
}
```

---

全部代码可在SDK中的vectorAddDrv例子中找到。

## G.1 上下文

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

一个主机线程在某时只能有一个当前设备上下文。当使用cuCtxCreate() 创建上下文时，它将成为主机调用线程的当前上下文。如果有效上下文不是线程的当前上下文，在该线程中操作的CUDA 函数（不涉及设备模拟或上下文管理的大多数函数）将返回CUDA\_ERROR\_INVALID\_CONTEXT。

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

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

引用计数有利于同一上下文中第三方授权代码间的互操作。比如，如果载入了三个使用相同上下文的库，则每个库都将调用cuCtxAttach() 来递增计数，并在库不再使用该上下文时调用cuCtxDetach() 递减计数。对大多数库来说，应用应当在载入或初始化库之前创建一个上下文，通过这种方式，应用可使用自己的启发式（heuristics）方法来创建上下文，库只需在传递给它的上下文中简单操作。希望创建自己的上下文的库（其客户端并不了解这种情况，并且可能已经创建或未创建自己的上下文）可使用cuCtxPushCurrent() 和cuCtxPopCurrent()，如G.1所示。

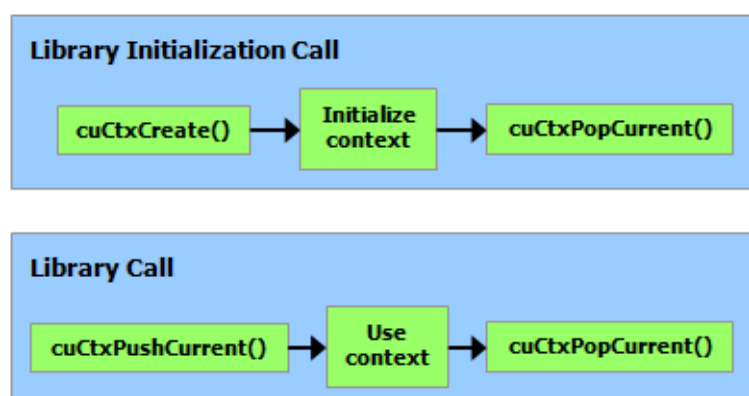


图 G.1: 库上下文管理

## G.2 模块

模块是可动态加载的设备代码和数据包，和Windows 中的DLL类似，是由nvcc输出的（参见3.1）。所有符号的名称，包括函数、全局变量、纹理参考或表面参考均在模块范围内维护，所以独立的第三方编写的模块可在同一CUDA 上下文中互操作。

下面的代码示例载入了一个模块并检索内核的句柄：

```
CUmodule cuModule;
cuModuleLoad(&cuModule, "myModule.ptx");
CUfunction myKernel;
cuModuleGetFunction(&myKernel, cuModule, "MyKernel");
```

下面的代码编译并加载来自于PTX代码的新模块且解析编译错误：

```
#define ERROR_BUFFER_SIZE 100
CUmodule cuModule;
CUjit_option options [3];
void* values [3];
char* PTXCode = "some_PTX_code";
options [0] = CU_ASM_ERROR_LOG_BUFFER;
values [0] = (void*)malloc(ERROR_BUFFER_SIZE);
options [1] = CU_ASM_ERROR_LOG_BUFFER_SIZE_BYTES;
```

```

values[1] = (void*)ERROR_BUFFER_SIZE;
options[2] = CU_ASM_TARGET_FROM_CUCONTEXT;
values[2] = 0;
cuModuleLoadDataEx(&cuModule, PTXCode, 3, options, values);
for (int i = 0; i < values[1]; ++i) {
    // Parse error string here
}

```

### G.3 内核执行

cuLaunchKernel()启动一个指定执行配置的内核。

参数要么作为指针数组（靠近cuLaunchKernel()的最后一个参数）传递，此时第n个指针对应第n个参数且指向要拷贝参数的存储器区域，要么作为额外选项之一（cuLaunchKernel()的最后参数）。

当参数作为额外选项传递（CU\_LAUNCH\_PARAM\_BUFFER\_POINTER选项），它们作为单个缓冲区的指针传递，此时通过为每个设备代码中的参数类型匹配对齐要求，它们之间保证合适的偏移。

设备代码中内置向量类型的对齐要求列在B.3.1中。对于所有其它基本类型，设备代码和主机代码的对齐要求一致，且可通过使用\_alignof()获得。唯一的例外是，当主机编译器将双精度和long long（或者64位机的long）对齐在单字边界而非双字边界（例如使用gcc的-mno-align-double编译选项）时，因为在设备代码中，这些类型永远以双字对齐。

CUdeviceptr是整形，但代表指针，所以它的对齐要求是\_alignof(void\*)。

下面的代码使用宏（ALIGN\_UP）调整每个参数的偏移以满足对齐要求，另一个宏（ADD\_TO\_PARAM\_BUFFER()）将每个参数加到参数缓冲区，该参数缓冲区被传递到CU\_LAUNCH\_PARAM\_BUFFER\_POINTER选项。

```

#define ALIGN_UP(offset, alignment) \
    (offset) = ((offset) + (alignment) - 1) & ~((alignment) - 1)

char paramBuffer[1024];
size_t paramBufferSize = 0;

```

```

#define ADD_TO_PARAM_BUFFER(value, alignment) \
do { \
    paramBufferSize = ALIGN_UP(paramBufferSize, alignment); \
    memcpy(paramBuffer + paramBufferSize, \
        &(value), sizeof(value)); \
    paramBufferSize += sizeof(value); \
} while (0)

int i;
ADD_TO_PARAM_BUFFER(i, __alignof(i));
float4 f4;
ADD_TO_PARAM_BUFFER(f4, 16); // float4's alignment is 16
char c;
ADD_TO_PARAM_BUFFER(c, __alignof(c));
float f;
ADD_TO_PARAM_BUFFER(f, __alignof(f));
CUdeviceptr devPtr;
ADD_TO_PARAM_BUFFER(devPtr, __alignof(devPtr));
float2 f2;
ADD_TO_PARAM_BUFFER(f2, 8); // float2's alignment is 8

void* extra[] = {
    CU_LAUNCH_PARAM_BUFFER_POINTER, paramBuffer,
    CU_LAUNCH_PARAM_BUFFER_SIZE, &paramBufferSize,
    CU_LAUNCH_PARAM_END
};
cuLaunchKernel(cuFunction,
    blockDim, blockDim, blockDim,
    gridWidth, gridHeight, gridDepth,
    0, 0, 0, extra);

```

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

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

## G.4 运行时API和驱动API的互操作性

应用可以混合使用运行时API和驱动API。

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

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

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

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

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

手册中设备和版本管理节的所有函数可互换使用。

## G.5 注意

本手册直接翻译自官方手册，相关法律或版权问题请以原版为准。

