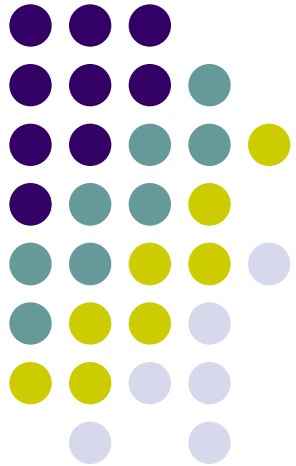
High Performance Computing with GPU

徐 云
中国科学技术大学计算机学院
国家高性能计算中心（合肥）
2019@ustc.hefei



主要内容

Ⅰ. Introduction to GPU

Ⅱ. GPU Architecture

Ⅲ. CUDA Programming

Ⅳ. Example: Matrix Multiplication

Ⅴ. Performance and Optimization

Part Ⅰ Introduction to GPU

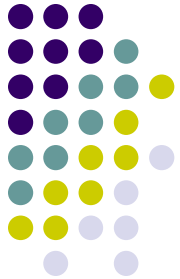
1. GPU的发展

2. CPU和GPU比较

3. GPU的应用和资源

注：部分资料来自于“高小鹏等, 通用计算中的GPU. 中国计算机学会通讯, 2009, 5(11)”

1.1 GPU与GPGPU

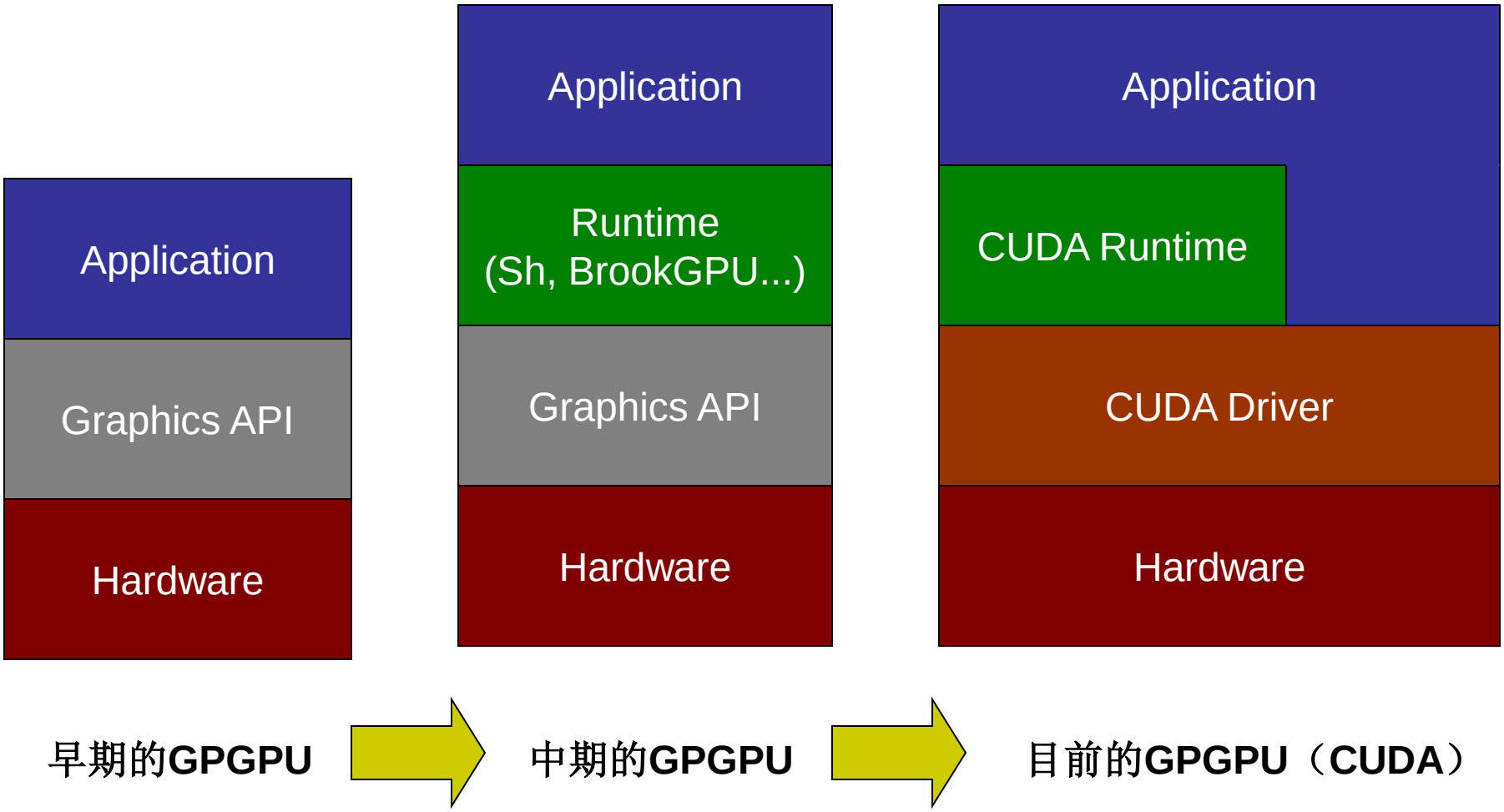


- 图形处理器(GPU, Graphics Process Unit)
  - 发展速度超过CPU
  - 今天的GPU不仅具备高质量和高性能图形处理能力，还可用于通用计算
- 用于通用计算的GPU(General-Purpose Computing on GPU，GPGPU)
  - 随着内部单元数量的快速增长及可编程性的持续改进，已经演化成为一个新型的并行计算平台
  - 一个必须引起重视的研究领域和技术

1.2 GPU的发展阶段

- 第一代GPU(1999年以前)：部分功能从CPU分离，实现硬件加速
  - GE(Geometry Engine)为代表，只能起到3D图像处理的加速作用，不具有软件编程特性
- 第二代GPU(1999年-2002年)：进一步硬件加速和有限的编程性
  - 1999年NVIDIA GeForce 256将T&L(Transform and Lighting)等功能从CPU分离出来，实现了快速变换
  - 2001年NVIDIA和ATI分别推出的GeForce3和Radeon 8500，图形硬件的流水线被定义为流处理器，出现了顶点级可编程性，同时像素级也具有有限的编程性，但GPU的编程性比较有限
- 第三代GPU(2002年以后)：方便的编程环境(如CUDA)
  - 2002年ATI发布的Radeon 9700和2003年NVIDIA GeForce FX的推出
  - 2006年NVIDIA与ATI分别为推出了CUDA(Computer Unified Device Architecture，统一计算架构)编程环境和CTM(Close To the Metal)编程环境

1.2 GPGPU的发展阶段 Cont.



1.3 GPGPU的时代已到来

- 随着GPU可编程性不断增强，特别是CUDA等编程环境的出现，使GPU通用计算编程的复杂性大幅度降低。

- 由于可编程性、功能、性能不断提升和完善，GPU已演化为一个新型可编程高性能并行计算资源。

- 全面开启GPU面向通用计算的新时代已到来。

Part Ⅰ Introduction to GPU

1. GPU的发展

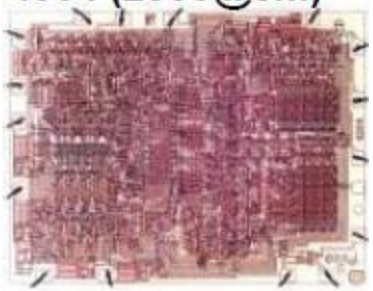
2. CPU和GPU比较

3. GPU的应用和资源

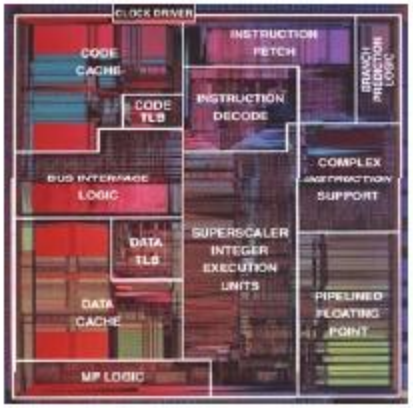
2.1 单核时代的摩尔定律

- CPU时钟频率每18个月翻一番
- CPU制造工艺逐渐接近物理极限
- 功耗和发热成为巨大的障碍

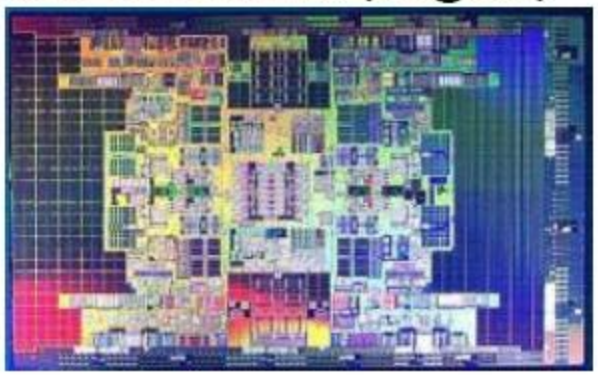
4004 (2300@8m)



Pentium 4 (42M@.18μ)

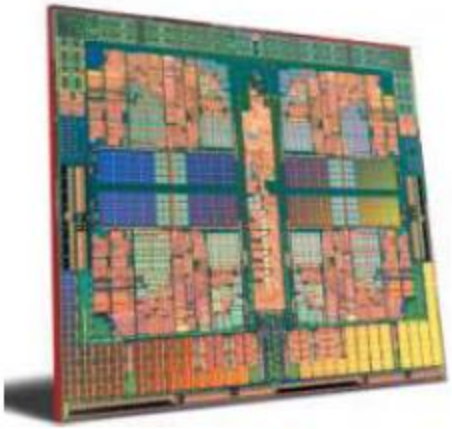


Quad-Core Itanium (2G@65n)

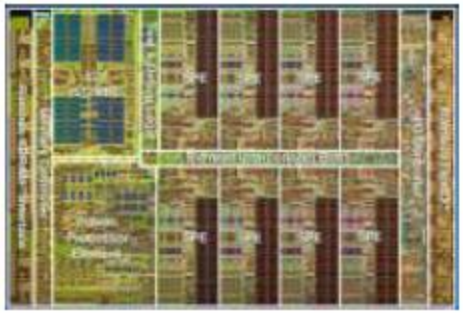


2.2 GPU是多核技术的代表之一

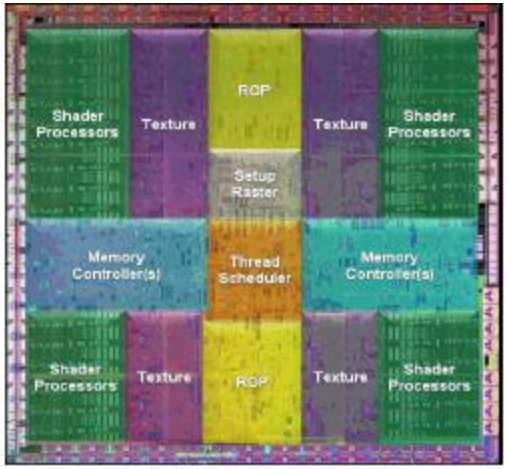
- 在一块芯片上集成多个较低功耗的核心
- 单个核心频率基本不变（一般在1-3GHz）
- 设计重心转向到多核的集成技术
- GPU是一种特殊的多核处理器



Quad-core Opteron

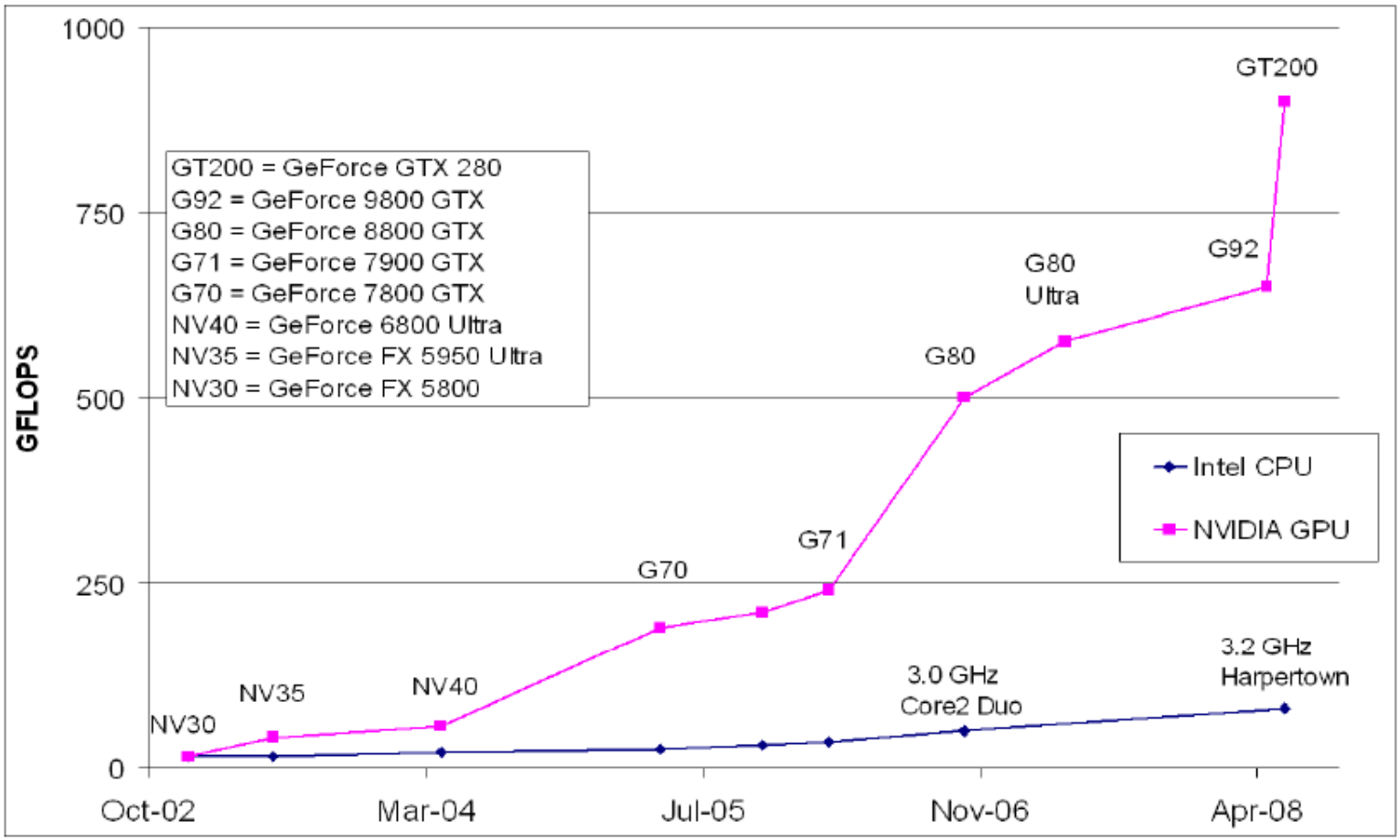


IBM Cell Broadband Engine

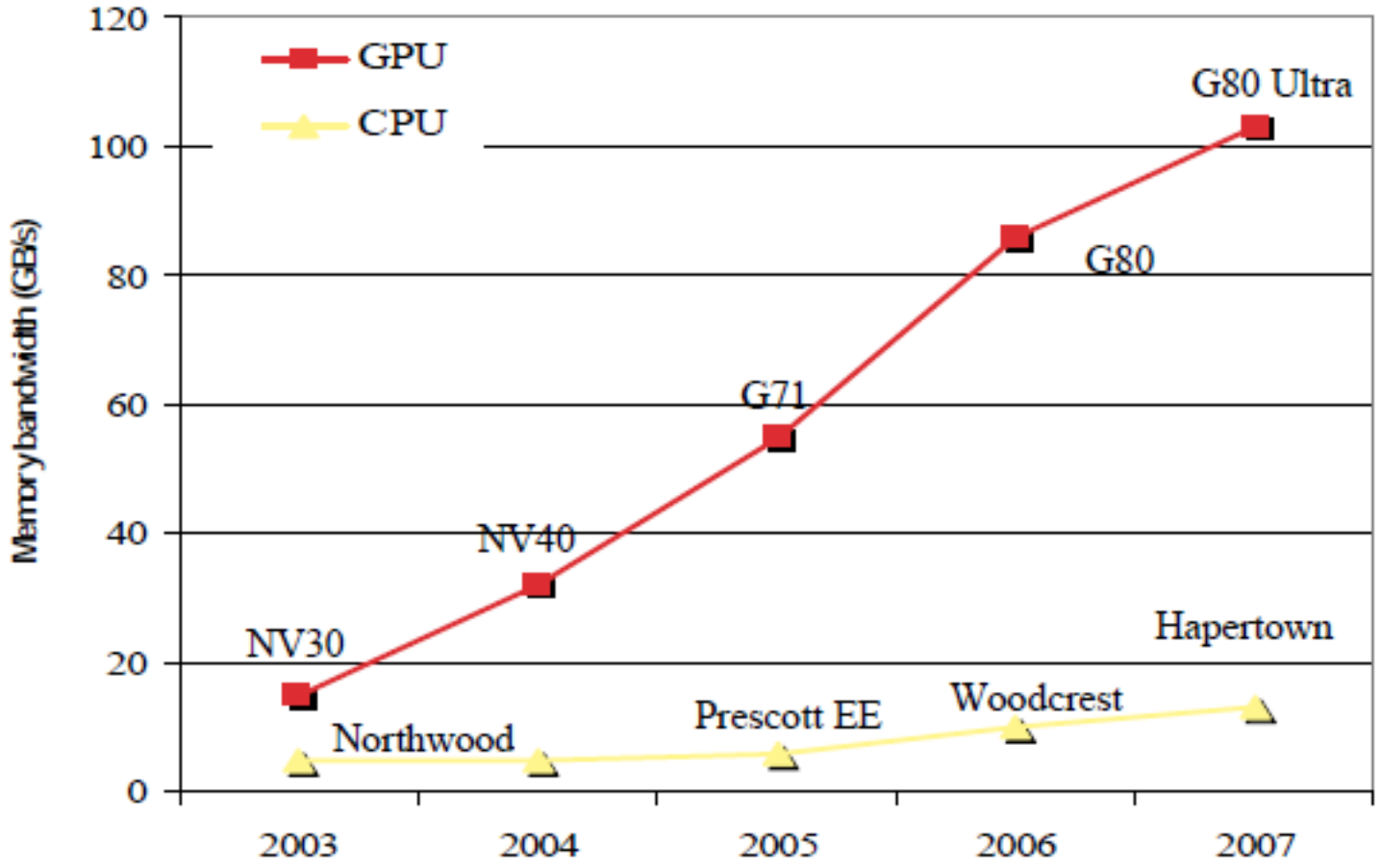


nVidia GT200

2.3 GPU和CPU浮点计算能力对比

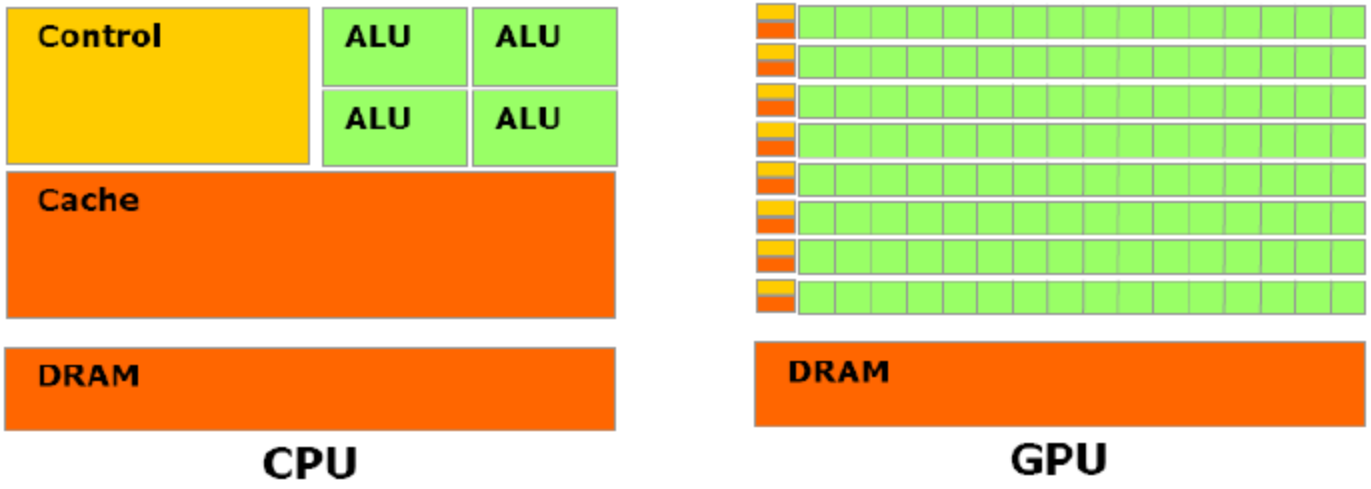


2.4 GPU和CPU存储器带宽对比



2.5 GPGPU的优势

- CPU：更多资源用于缓存和逻辑控制
- GPU：更多资源用于计算，适用于高并行性、大规模数据密集型、可预测的计算模式。



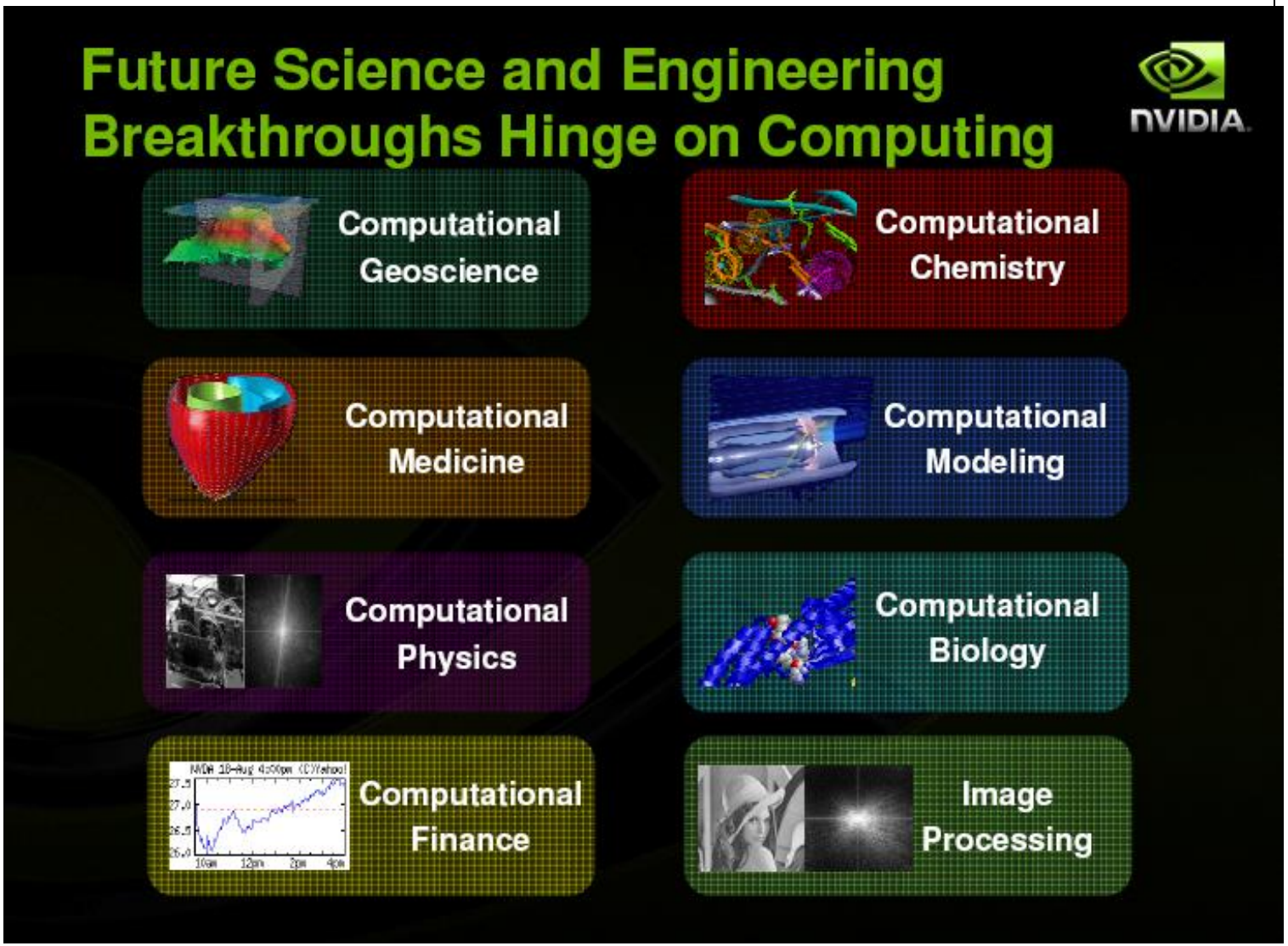
Part Ⅰ Introduction to GPU

1. GPU的发展

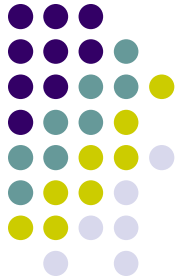
2. CPU和GPU比较

3. GPU的应用和资源

3.1 GPU的应用


Future Science and Engineering
Breakthroughs Hinge on Computing
NVIDIA
Computational
Geoscience
Computational
Chemistry
Computational
Medicine
Computational
Modeling
Computational
Physics
Computational
Biology
Computational
Finance
Image
Processing

3.2 GPU的资源



- NVIDIA CUDA Homepage
  - Contains downloads, documentation, examples and links
  - http://www.nvidia.com/cuda
- Programming Guide
- CUDA Forums
  - http://forums.nvidia.com
  - The CUDA designers actually read and respond on the forum
- Supercomputing 2007 CUDA Tutorials
  - http://www.gpgpu.org/sc2007/
- CUDA中文网站
  - http://www.cuda.net/zone_tech.html

主要内容

Ⅰ. Introduction to GPU

Ⅱ. GPU Architecture

Ⅲ. CUDA Programming

Ⅳ. Example: Matrix Multiplication

Ⅴ. Performance and Optimization

PartⅡ GPU Architecture

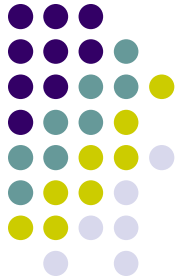
1. 已有的两类GPU结构

2. 存储器层次结构

3. 线程组织结构

4. 同步

1.1 支持通用计算的两类GPU结构



- 基于流处理器阵列的主流GPU结构
  - 以NVIDIA的GeForce8800GTX和ATI的HD 2900为代表
  - GeForce 8800GTX包含了128个流处理器，HD 2900包含了320个流处理器。这些流处理器可以支持浮点运算、分支处理、流水线、SIMD（Single Instruction Multiple Data，单指令流多数据流）等技术。
- 基于通用计算核心的GPU结构
  - Intel Larrabee核心是一组基于x86指令集的CPU核, CPU核拓展了x86指令集，并包含大量向量处理操作和若干专门的标量指令，同时还支持子例程以及缺页中断。
- 前者相对于后者具有更高的聚合计算性能，而后者则在可编程性上具有更大的优势。

1.2 基于流处理器阵列的GPU结构图

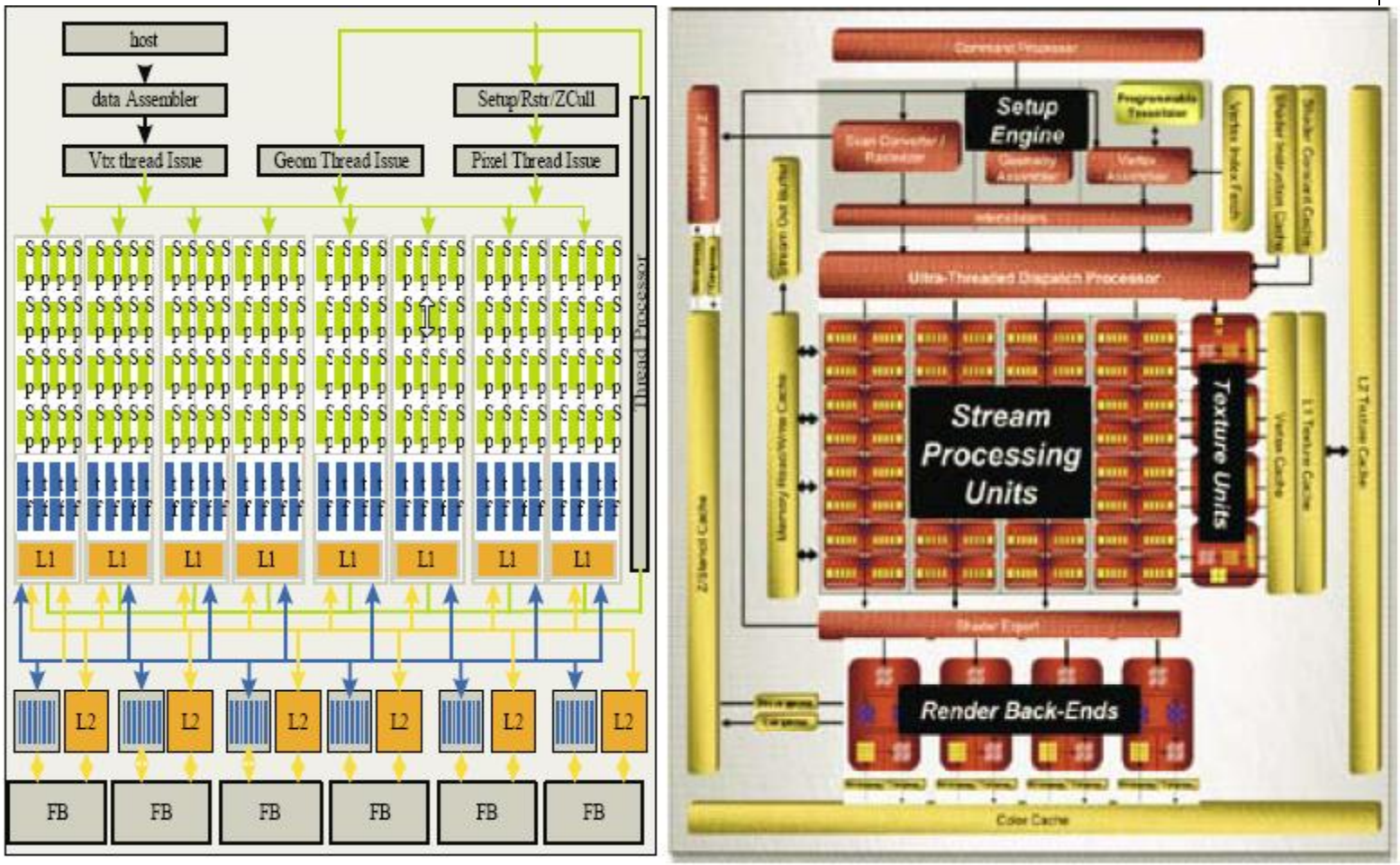


图1 GeForce 8800GTX（左）、HD 2900（右）

1.3 基于通用计算核心的GPU结构图

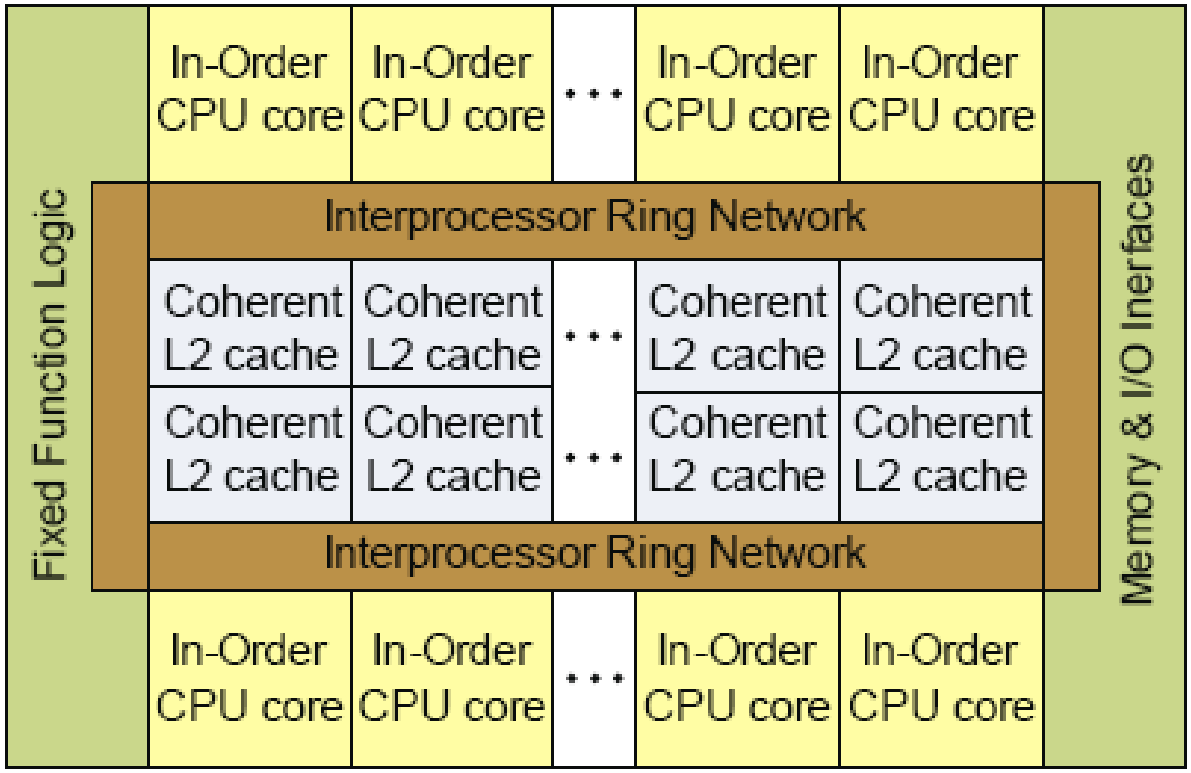
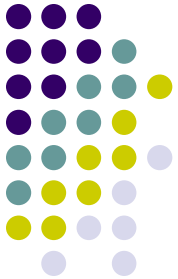
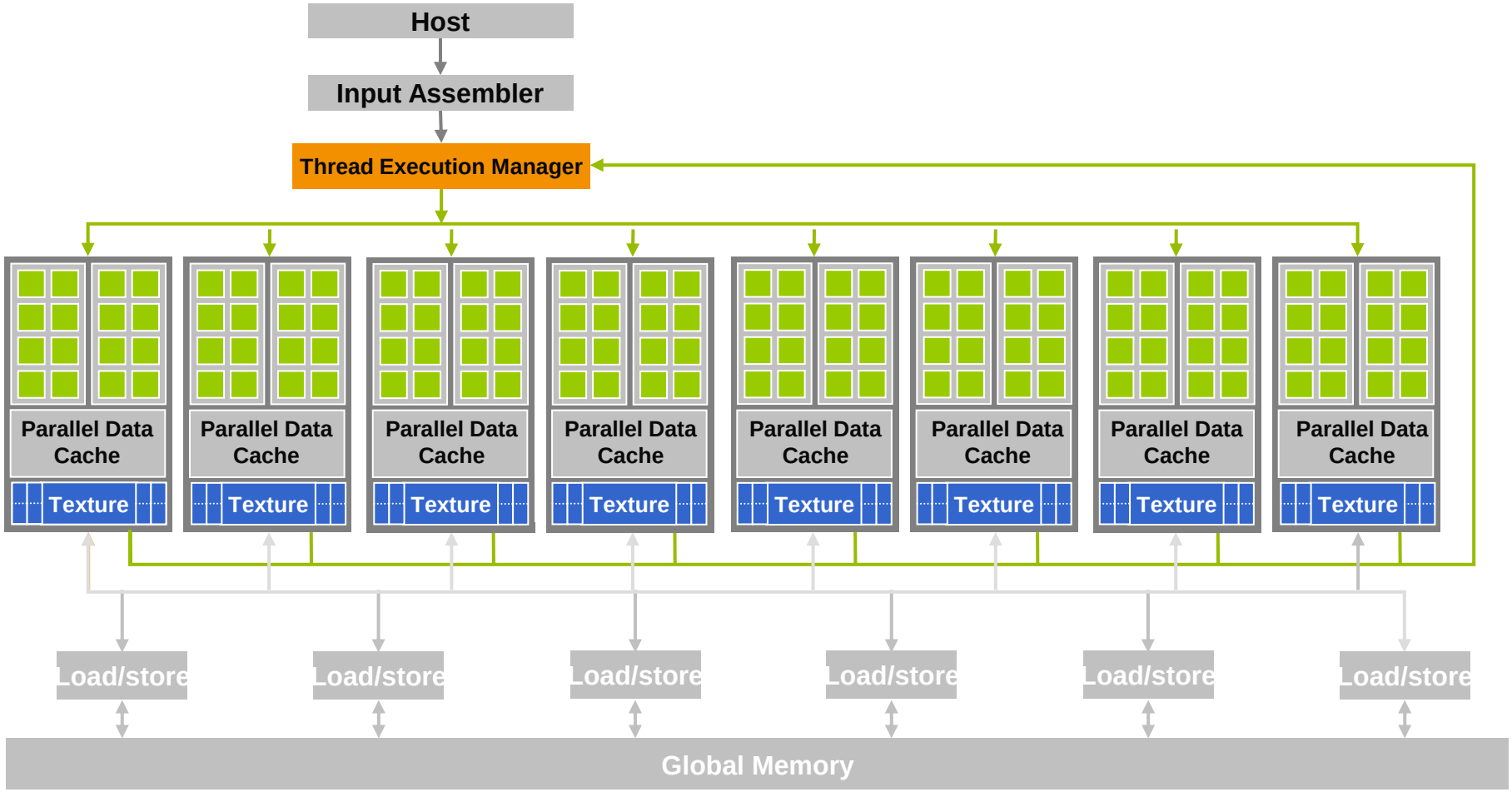


图2 Larrabee多核结构示意图

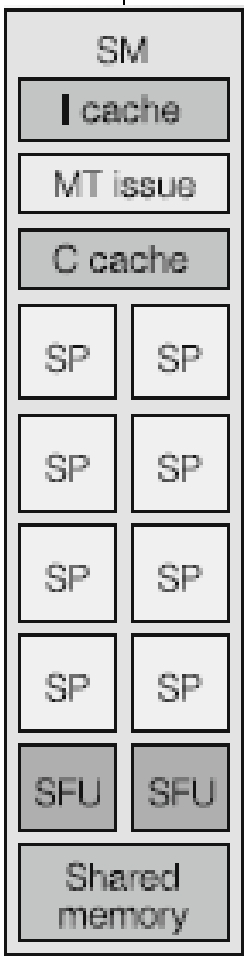
1.4 NVIDIA G80系列细解：Device Architecture





1.4 NVIDIA G80系列细解：SM(stream multiprocessor)

- GPU主要的组成单元，共16个SM
- 每个SM包含
  - 8个SP(scalar processor)，主频为1.35GHZ，所有SP受控同一个指令单元，同步执行
  - 两个SFU(special function unit)
  - 一个指令cache(I cache)
  - 一个常数cache(C cache) 8KB
  - 一个纹理cache(T cache) 6~8KB
  - 一个多线程发射单元(MT issue)
  - 一个16KB的shared memory，用于线程块内共享数据，访存速度很快
  - 8192个32位字大小的寄存器文件供共享
- 线程的创建、管理和执行由硬件调度，调度本身没有额外开销。



PartⅡ GPU Architecture

1. 已有的两类GPU结构

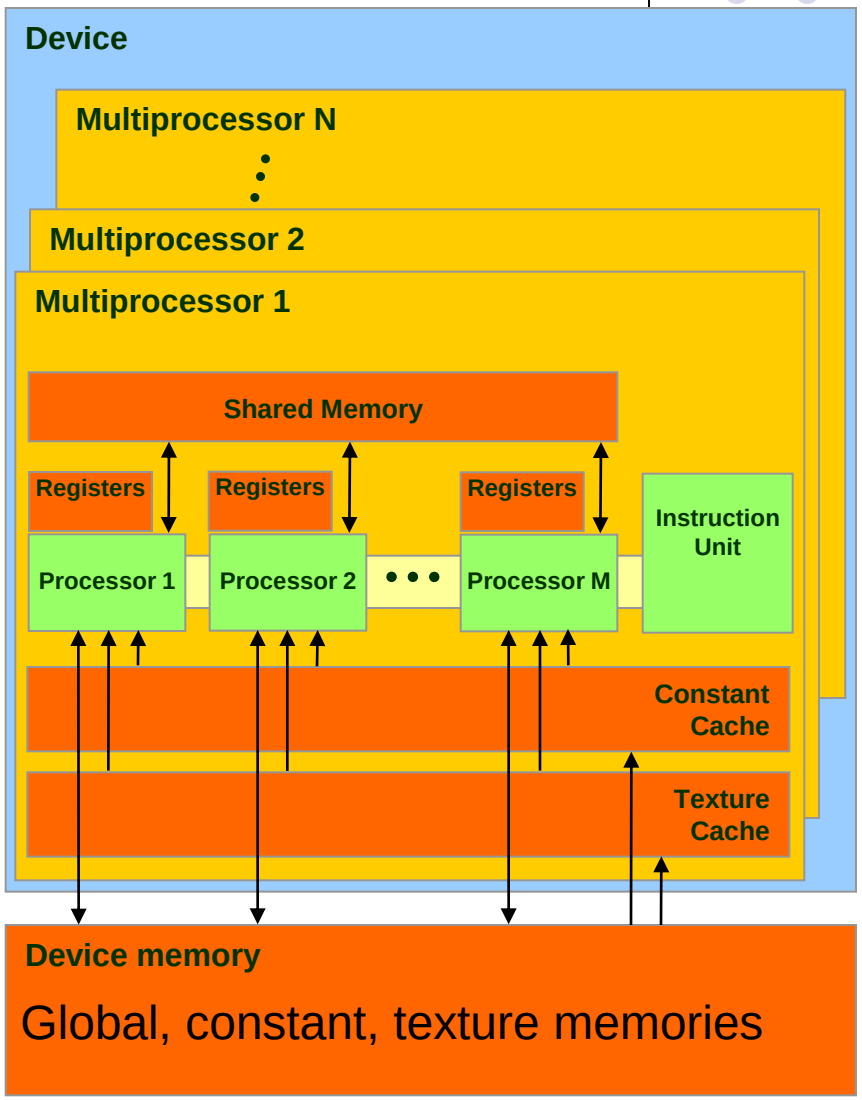
2. 存储器层次结构

3. 线程组织结构

4. 同步

2.1 存储器层次结构

- The local, global, constant, and texture spaces are regions of device memory
- Each multiprocessor has:
  - A set of 32-bit registers per processor
  - On-chip shared memory
    - ✓ Where the shared memory space resides
  - A read-only constant cache
    - ✓ To speed up access to the constant memory space
  - A read-only texture cache
    - ✓ To speed up access to the texture memory space

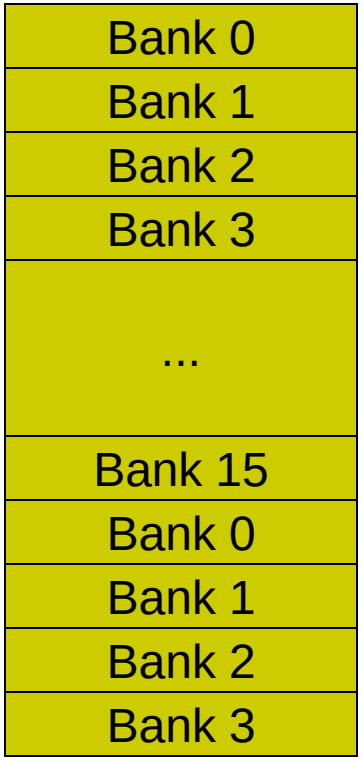


2.2 基本访存开销

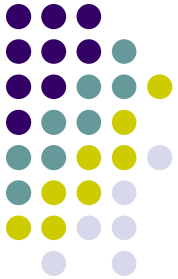
存储器类型	位置	是否被缓存	访问速度
寄存器（Registers）	芯片上	不被缓存	几乎没有额外延迟
共享存储器（Share Memory）	芯片上	不被缓存	同寄存器
全局存储器（Device Memory）	设备上	不被缓存	400-600时钟周期
本地存储器（Local Memory）	设备上	不被缓存	400-600时钟周期
固定存储器（Constant Memory）	设备上	被缓存	被缓存时：同寄存器 未被缓存： 400-600时钟周期
纹理存储器（Texture Memory）	设备上	被缓存	被缓存时：同寄存器 未被缓存： 400-600时钟周期

2.3 共享存储器与存储体冲突

- 访问共享存储器速度很快，只要不存在存储体冲突（Bank Conflict），其速度与寄存器一样
- 共享存储器划分：分为16个存储体(bank)，每个bank按连续4byte循环分配的。不同bank的数据可以并发访问。
- 存储体冲突：同一个bank的访问请求被序列化，造成多倍的访问延迟。
  - 例外：对同一个内存地址的访问使用广播方式，不会造成额外延迟



PartⅡ GPU Architecture



1. 已有的两类GPU结构

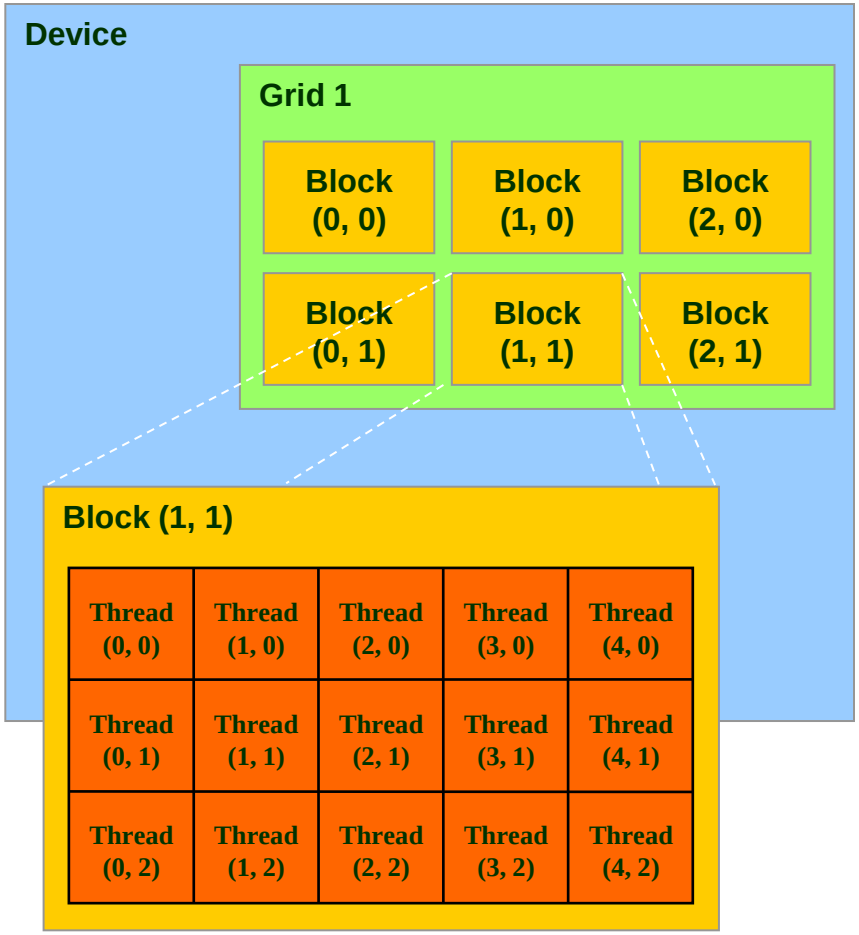
2. 存储器层次结构

3. 线程组织结构

4. 同步

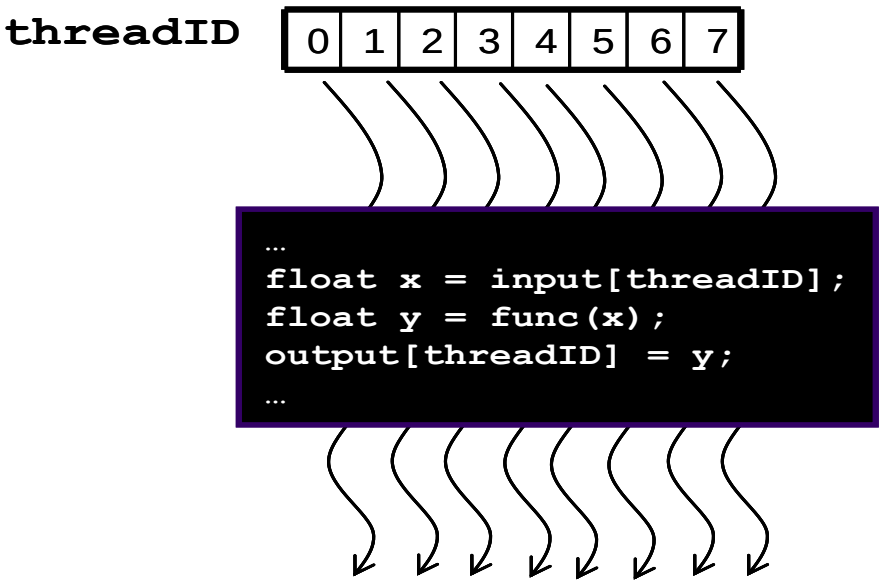
3.1 CUDA中的线程层次

- 线程：
  - CUDA中的基本执行单元；
  - 硬件支持，开销很小；
  - 所有线程执行相同的代码（STMD）
- 线程块：
  - 若干线程还可以组成块(Block，每个块至多512个线程)
  - 线程块可以呈一维、二维或者三维结构
  - 每个线程块分为若干个组(称为warp)，每个warp包含32个线程，物理上以SIMD方式并行
- 线程网格：
  - 若干个线程块可以组织成网格grid
  - Grid可以是一维或二维结构



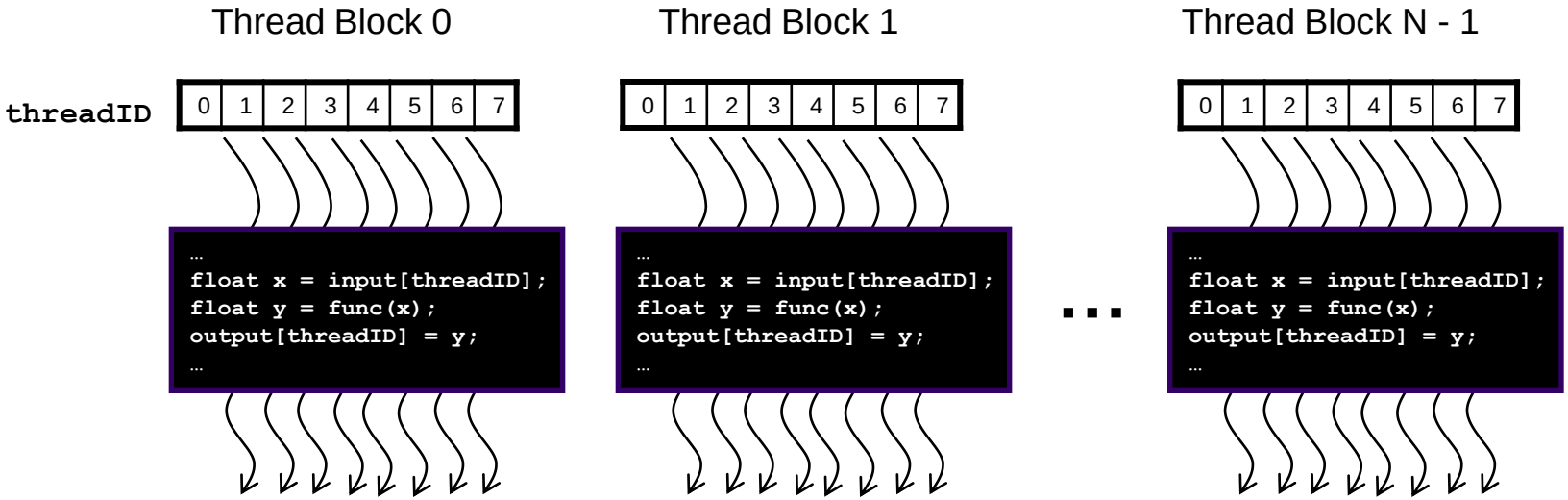
3.2 线程块ID和线程ID

- Thread id:
  a) local id: thread id in a block
  b) global id: thread id in a grid
- Compute thread global id : blockDim*blockId+threadId
- Each thread uses IDs to decide what data to work on



3.3 线程块中的线程合作

- Divide monolithic thread array into multiple blocks
  - Threads within a block cooperate via **shared memory, atomic operations** and **barrier synchronization**
  - Threads in different blocks cannot cooperate



PartⅡ GPU Architecture

1. 已有的两类GPU结构

2. 存储器层次结构

3. 线程组织结构

4. 同步

4.1 CPU与GPU之间的同步

- CPU启动内核kernel是异步的，即当CPU启动GPU执行kernel时，CPU并不等待GPU完成就立即返回，继续执行后面的代码。例如：

```
.....
kernel < < <gridDim, blockDim>>>(arg1, arg2);
c=a+b;
.....
```

- CPU在调用kernel后，就接着执行后面的c=a+b，而GPU在执行kernel函数，此时CPU和GPU是完全并行的工作。如果CPU在接下来的操作中需要用到GPU的计算结果，则CPU必须阻塞等待GPU执行完毕。可在kernel后添加一条同步语句实现。

```
.....
kernel < < <gridDim, blockDim>>>(arg1, arg2);
cudaThreadSynchronize（）；//实现CPU与GPU之间的同步
c=a+b;
.....
```

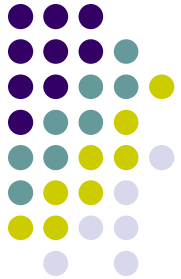
4.2 同一个block内的同步

- 前面提到，同一个block内的线程可以通过shared memory共享数据。
- 此外，同一个block内的线程还可以快速同步

```
__global__ void kernel(arg1, arg2)
{
    int tid=threadIdx.x;
    ……
     __syncthreads(); //用于实现同一个块内线程的同步
    ……
}
```

只有当同一个块内的所有线程都到达函数__syncthreads()时才会继续往下执行

4.3 不同block之间的同步



- 同一个grid中的不同线程块之间不能同步，即CUDA运行时库中没有提供此类函数

- 但可以通过终止一个kernel来实现同步

主要内容

Ⅰ. Introduction to GPU

Ⅱ. GPU Architecture

Ⅲ. CUDA Programming

Ⅳ. Example: Matrix Multiplication

Ⅴ. Performance and Optimization

Part Ⅲ CUDA Programming

1. CUDA软件架构
2. CUDA编程语言
3. 内核函数
4. 运行时API
5. CUDA程序结构
6. CUDA程序的编译、链接、调试

1 CUDA的软件架构

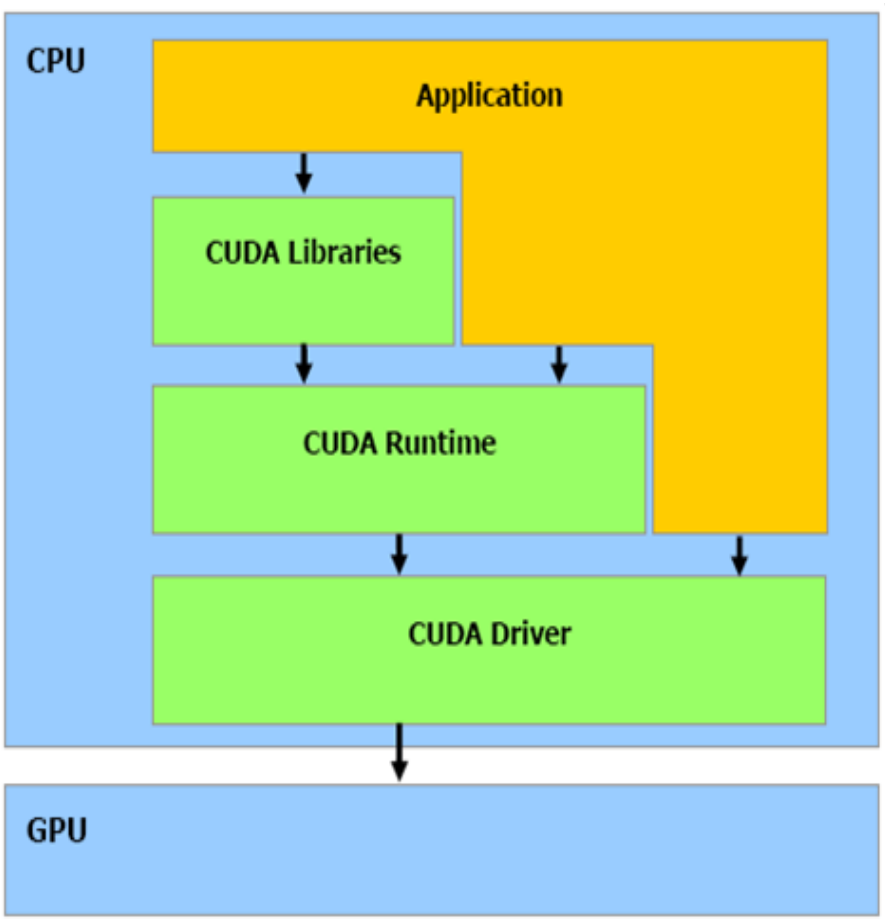
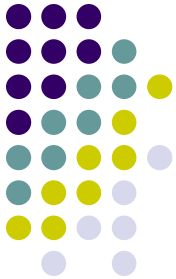


图 1-3. 统一计算设备架构软件堆栈

1 CUDA软件架构（续）



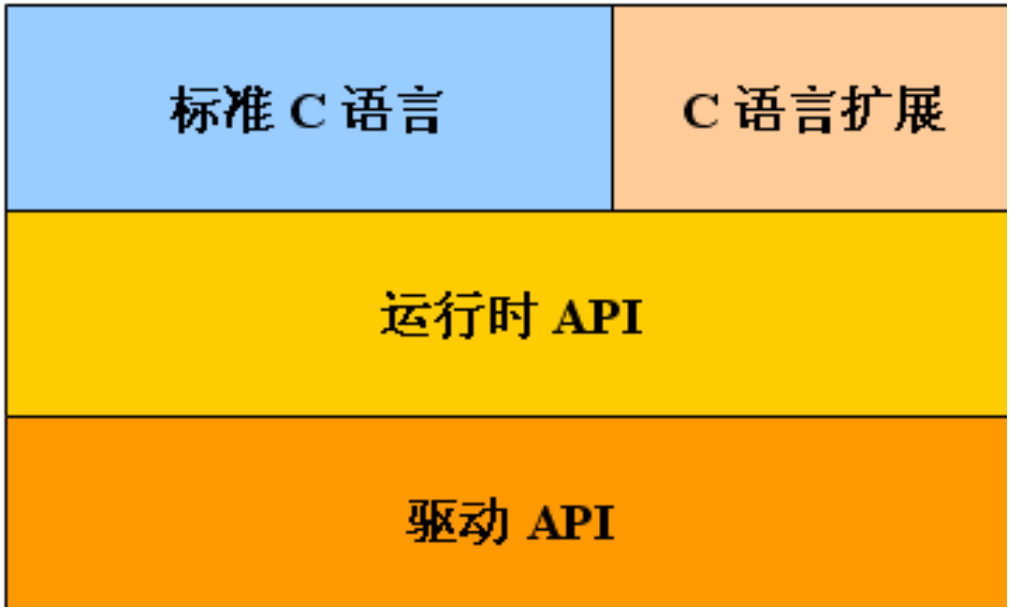
- 三个部分
  - 开发库（CUDA Library）， 目前包括两个标准的数学运算库CUFFT和CUBLAS
  - 运行时环境（CUDA Runtime），提供开发接口和运行时组件，包括基本数据类型的定义和各类计算、内存管理、设备访问和执行调度等函数
  - 驱动（CUDA Driver），提供了GPU的设备抽象级的访问接口，使得同一个CUDA应用可以正确的运行在所有支持CUDA的不同硬件上

Part Ⅲ CUDA Programming

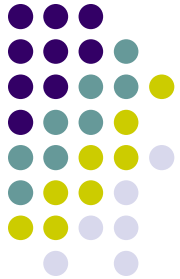
1. CUDA软件架构
2. CUDA编程语言
3. 内核函数
4. 运行时API
5. CUDA程序结构
6. CUDA程序的编译、链接、调试

2.1 CUDA编程语言

- CUDA编程语言主要以C语言为主，增加了若干定义和指令。



2.2 函数限定符



- 函数类型限定符需要指定函数的执行位置（主机或设备）和函数调用者（通过主机或通过设备）
- 在设备上执行的函数受到一些限制，如函数参数的数目固定，无法声明静态变量，不支持递归调用等等
- 用 _global_ 限定符定义的函数是从主机上调用设备函数的唯一方式，其调用是异步的，即立即返回

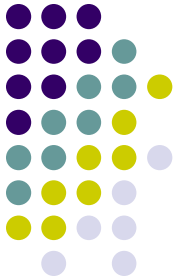
函数限定符	在何处执行	从何处调用	特性
device	设备	设备	函数的地址无法获取
global	设备	主机	返回类型必须为空
host	主机	主机	等同于不使用任何限定符

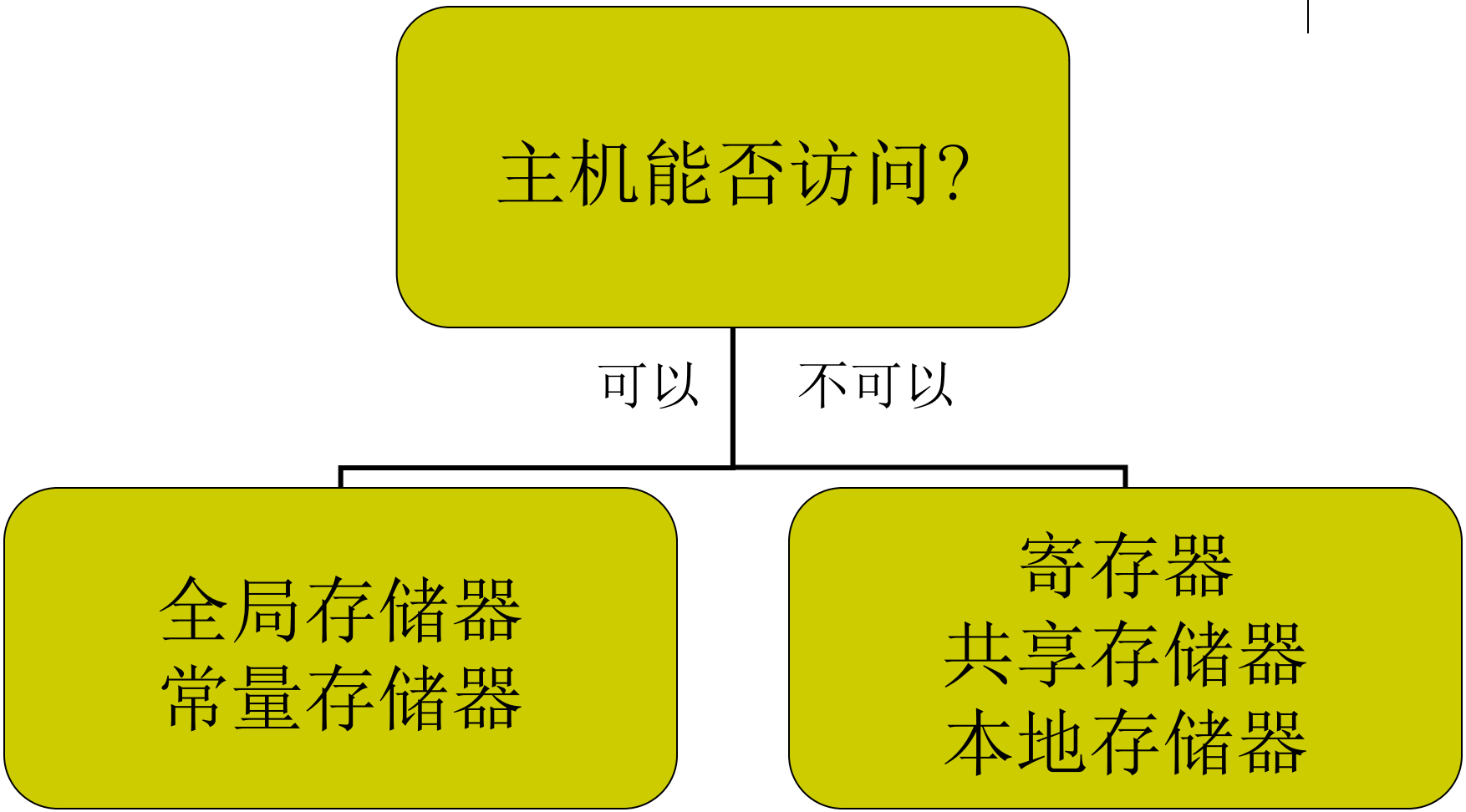
2.3 变量限定符

- _shared_ 限定符声明的变量只有在线程同步执行之后，才能保证共享变量对其他线程的正确性。
- 不带限定符的变量通常位于寄存器中。若寄存器不足，则置于本地存储器中

限定符	位于何处	可以访问的线程	主机访问
device	全局存储器	线程网格内的 所有线程	通过运行时库访问
constant	固定存储器	线程网格内的 所有线程	通过运行时库访问
shared	共享存储器	线程块内的 所有线程	不可从主机访问

主机能访问哪里的变量?





2.4 内置的向量类型

- 内置的向量类型都是结构体
- 用(u)+基本数据类型+数字1-4组成
  - 例如char2、uint3、ulong4等等。
- 特殊类型dim3，基本等同于uint3，区别只在于在定义dim3变量时，未指定的分量都自动初始化为1。
  - 一般用于定义线程块和线程网格的大小。

2.5 常用的内置变量

内置变量	类型	含义
gridDim	dim3	线程网格的维度
blockDim	dim3	线程块的维度
blockIdx	uint3	线程网格内块的索引
threadIdx	uint3	线程块内线程的索引
warpSize	int	一个warp块内包含的线程数

Part Ⅲ CUDA Programming

1. CUDA软件架构
2. CUDA编程语言
3. 内核函数
4. 运行时API
5. CUDA程序结构
6. CUDA程序的编译、链接、调试

3.1 内核函数（Kernel）

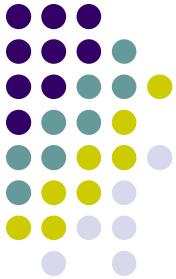
- 内核函数是特殊的一种函数，是从主机调用设备代码唯一的接口，相当于显卡环境中的主函数
- 内核函数的参数被通过共享存储器传递，从而造成可用的共享存储器空间减少（一般减少100字节以内）
- 内核函数使用__global__函数限定符声明，返回值为空

```
__global__ void KernelDemo(float* a, float* b, float* c)
{
    int i = threadIdx.x;
    c[i] = a[i] + b[i];
}
```

3.2 内核函数(kernel)的调用

- 调用内核函数需要使用KernelName<<<>>>()的方式
- <<<>>>内的参数用于指定执行内核函数的配置，包括线程网格，线程块的维度，以及需求的共享内存大小，例如 <<<DimGrid, DimBlock, MemSize>>>
  - DimGrid（dim3类型），用于指定网格的两个维度，第三维被忽略
  - DimBlock （dim3类型），指定线程块的三个维度
  - MemSize （ size_t类型），指定为此内核调用需要动态分配的共享存储器大小
- 若当前硬件无法满足用户指定的配置，则内核函数不会被执行，直接返回错误信息

3.3 内核调用的示例

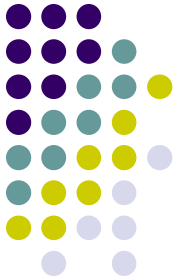


```
__global__ void KernelDemo(float* a, float* b, float* c) // 内核定义
{
    int i = threadIdx.x;
    c[i] = a[i] + b[i];
}

int main() //主函数
{
    dim3 dimGrid(1, 1, 1);
    dim3 dimBlock(100, 1, 1);
    KernelDemo <<< dimGrid, dimBlock,1024>>>(a,b,c); // 调用内核
}
```

Part Ⅲ CUDA Programming

1. CUDA软件架构
2. CUDA编程语言
3. 内核函数
4. 运行时API
5. CUDA程序结构
6. CUDA程序的编译、链接、调试

4.1 运行时API



- 设备管理
  - cudaGetDeviceCount()： 获得可用GPU设备的数目
  - cudaGetDeviceProperties()： 得到相关的硬件属性
  - 使用cudaSetDevice()： 选择本次计算使用的设备
  - 默认使用第一个可用的GPU设备，即device 0
- 内存管理
  - cudaMalloc()： 分配线性存储空间
  - cudaFree()： 释放分配的空间
  - cudaMemcpy()：内存拷贝
  - cudaMallocPitch()：分配二维数组空间并自动对齐
  - cudaMemcpyToSymbol()：将主机上的一块数据复制到GPU上的固定存储器

4.2 内存拷贝cudaMemcpy()

- 由于主机内存和设备内存是完全不同的两个内存空间，因此必须严格指定数据所在的位置。
- 四种不同的传输方式
  - 主机到主机（HostToHost）
  - 主机到设备（HostToDevice）
  - 设备到主机（DeviceToHost）
  - 设备到设备（DeviceToDevice）
- 其中主机到设备和设备到主机的传输需要经过主板上的PCI-E总线接口，一般带宽在1～2GB/s左右。而设备到设备的带宽可达40GB/s以上

4.3 计时函数

- CUDA自带一个精确的计时函数

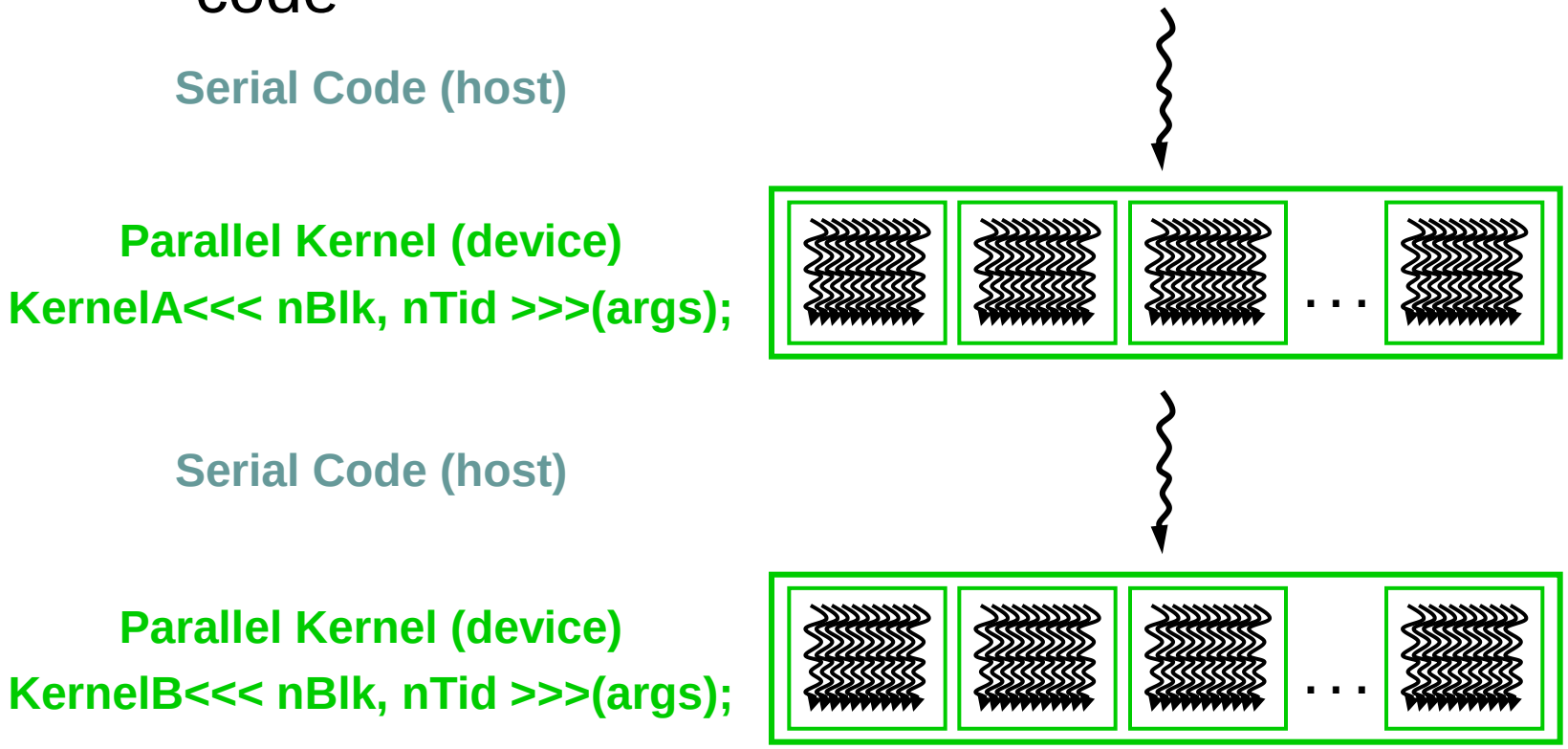
```
unsigned int timer = 0;
CUT_SAFE_CALL(cutCreateTimer(&timer)); //定义计时器
cudaThreadSynchronize();
CUT_SAFE_CALL(cutStartTimer(timer)); //计时器启动

CudaKernel<<<dimGrid, dimBlock, memsize>>>(); //GPU计算

cudaThreadSynchronize(); //等待计算完成
CUT_SAFE_CALL(cutStopTimer(timer) ); //计时器停止
 float timecost=cutGetAverageTimerValue(timer);  //获得计时结果
 printf("CUDA time %.3fms\n",timecost);
```

Part Ⅲ CUDA Programming

1. CUDA软件架构
2. CUDA编程语言
3. 内核函数
4. 运行时API
5. CUDA程序结构
6. CUDA程序的编译、链接、调试

5.1 CUDA程序结构

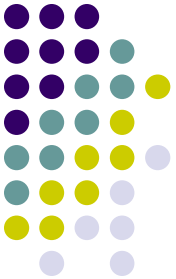
- Integrated host+device app C program
  - Serial or modestly parallel parts in **host** C code
  - Highly parallel parts in **device** SPMD kernel C code



5.2 CUDA程序的生命周期

- CUDA程序的生命周期：
  1. 主机代码执行
  2. 传输数据到GPU
  3. GPU执行
  4. 传输数据回CPU
  5. 继续主机代码执行
  6. 结束
- 如果有多个内核函数，需要重复2～4步

5.3 一个典型的CUDA程序

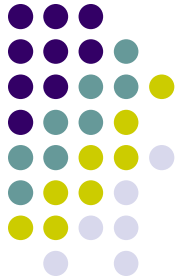


```
Main(){ //主函数
float *Md;
cudaMalloc((void**)&Md, size); //在GPU上分配空间
//从CPU复制数据到GPU
cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);

//调用内核函数
kernel<<<dimGrid, dimBlock>>> (arguments);

//从GPU将结果复制回CPU
CopyFromDeviceMatrix(M, Md);
FreeDeviceMatrix(Md);//释放GPU上分配的空间
}
```

Part Ⅲ CUDA Programming

1. CUDA软件架构
2. CUDA编程语言
3. 内核函数
4. 运行时API
5. CUDA程序结构
6. CUDA开发环境的安装及使用

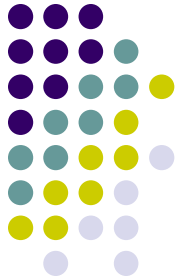
6.1 CUDA开发环境的安装（WinXP）



- 系统需求
  - Microsoft Visual Studio 2005或2008
  - 一块支持CUDA的显卡（若没有适用的显卡，也能安装和编程，但只能使用模拟器运行程序）

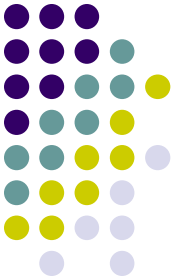
- 安装顺序（以2.0版本为例）
  - 1.显卡驱动 NVIDIADisplayWin2K(177_84)Int.exe
  - 2.工具包 NVIDIA_CUDA_toolkit_2.0_win32
  - 3.开发包 NVIDIA_CUDA_SDK_2.02.0811.0240_win32.exe
  - 以上资源可从Nvidia官网下载：http://www.nvidia.com/cuda

6.2 CUDA开发环境的配置



- 环境变量（自动配置）
  - CUDA_BIN_PATH = C:\CUDA\bin
  - CUDA_INC_PATH = C:\CUDA\include
  - CUDA_LIB_PATH = C:\CUDA\lib
- 设置Visual Studio 2005 （VS8）语法高亮
  - 打开目录C:\Program Files\NVIDIA Corporation\NVIDIA CUDA SDK\doc\syntax_highlighting\visual_studio_8
  - 复制目录下的usertype.dat到Microsoft Visual Studio 8\Common7\IDE目录，如果目标目录已有该文件，则手动将usertype.dat的内容添加到已有的usertype.dat文件后面
  - 在VS2005的“Tools->Options->Text Editor->File Extension”中添加对.cu后缀的支持，然后重启VS2005

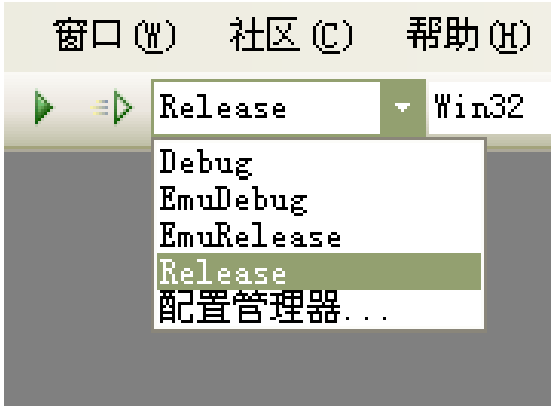
6.3 建立CUDA工程



- 方法一：使用已有项目
  - 打开示例文件夹 C:\Program Files\NVIDIA Corporation\NVIDIA CUDA SDK\projects
  - 复制任意一个现有项目，改名后即可直接使用
  - 该目录下的template专门用于创建新CUDA项目
- 方法二：建立全新项目
  - 需要添加系统变量，手动设置编译指令。
- 方法三： 使用CUDA向导（建议用此方法）CUDA_VS_Wizard_W32.2.0.exe

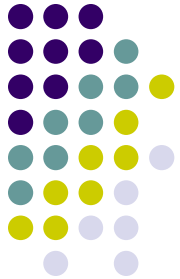
6.4 编译运行

- 普通的CUDA项目有四种配置
  - Debug
  - EmuDebug
  - EmuRelease
  - Release



- 模拟器运行使用EmuDebug或EmuRelease
- 真实显卡运行使用Debug或Release
- Debug配置包含更多调试信息，Release配置做了性能优化

6.5 调试



- CUDA代码分为主机代码和设备代码
  - 主机代码（Host）：运行于普通CPU上的代码
  - 设备代码（Device）：运行于GPU上的代码
- CUDA不支持在显卡设备上的代码调试
- 使用模拟器的EmuDebug配置，可以调试主机和设备代码，其中多线程是按顺序轮询执行，某些线程访问冲突的错误不可能重现
- 使用显卡Debug配置，仅可以调试主机代码

主要内容

Ⅰ. Introduction to GPU

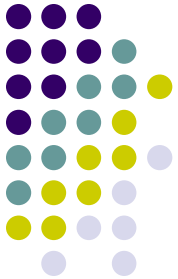
Ⅱ. GPU Architecture

Ⅲ. CUDA Programming

Ⅳ. Example: Matrix Multiplication

Ⅴ. Performance and Optimization

Part Ⅳ
Example: Matrix Multiplication



1. 串行的矩阵乘法在CPU上的实现

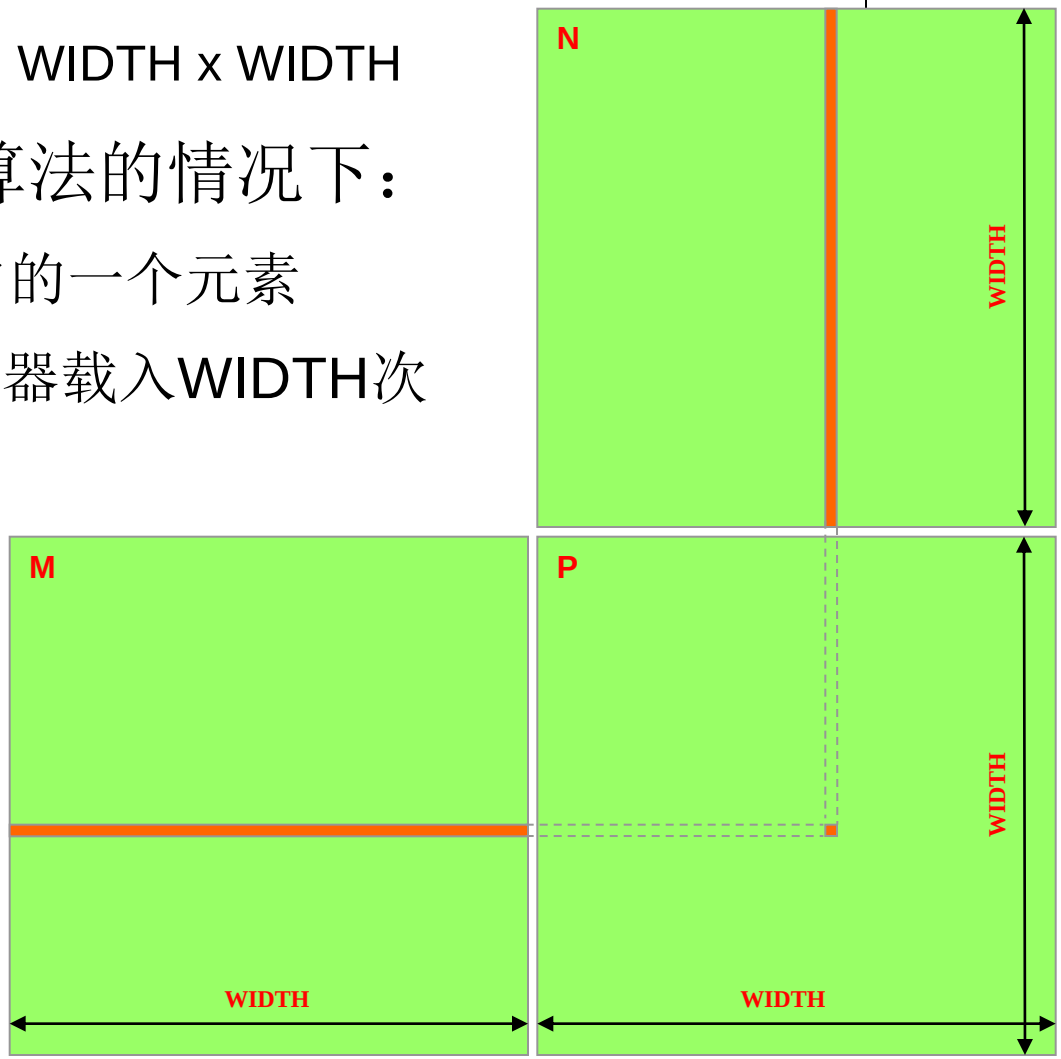
2. 并行的矩阵乘法在GPU上的实现

    2.1. 没有使用shared memory的实现

    2.2 使用了shared memory的实现

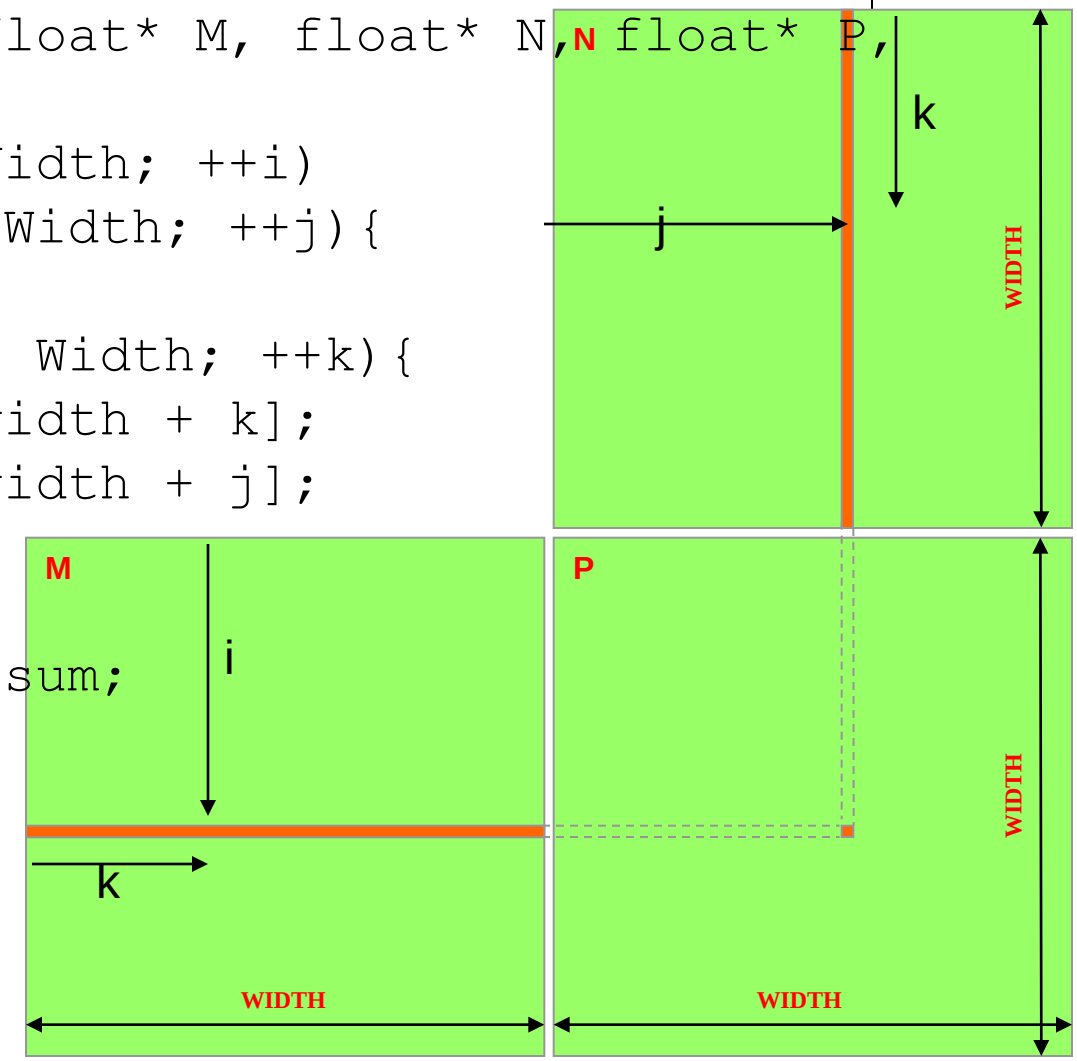
1.1 方形矩阵乘法及存在问题

- 矩阵P = M * N 大小为 WIDTH x WIDTH
- 在没有采用分片优化算法的情况下：
  - 一个线程计算P矩阵中的一个元素
  - M和N需要从全局存储器载入WIDTH次

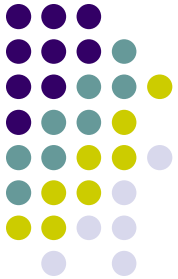


1.2 串行版本的矩阵乘法

```
// 宿主机的双精度矩阵乘法
void MatrixMulOnHost(float* M, float* N, float* P,
int Width){
 for (int i = 0; i < Width; ++i)
  for (int j = 0; j < Width; ++j){
   double sum = 0;
   for (int k = 0; k < Width; ++k){
    double a = M[i * width + k];
    double b = N[k * width + j];
    sum += a * b;
   }
   P[i * Width + j] = sum;
  }
}
```



Part Ⅳ
Example: Matrix Multiplication



1. 串行的矩阵乘法在CPU上的实现

2. 并行的矩阵乘法在GPU上的实现

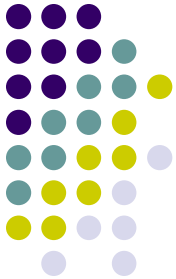
    2.1. 没有使用shared memory的实现

    2.2 使用了shared memory的实现

2 矩阵乘法在GPU上的实现

- 目的：用矩阵乘法说明CUDA编程中内存和线程管理的基本特性。
  - 共享存储器的用法；
  - 本地存储器、寄存器的用法；
  - 线程ID的用法；
  - 主机和设备之间数据传输的API；
  - 为方便描述，以方形矩阵来说明。

Part Ⅳ
Example: Matrix Multiplication



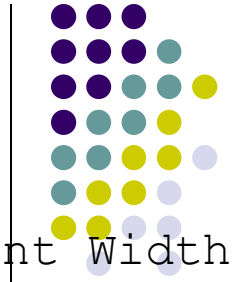
1. 串行的矩阵乘法在CPU上的实现

2. 并行的矩阵乘法在GPU上的实现

   2.1. 没有使用shared memory的实现

   2.2 使用了shared memory的实现

2.1.1 向GPU传输矩阵数据

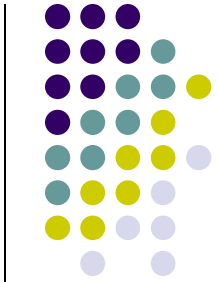


```
void MatrixMulOnDevice(float* M, float* N, float* P, int Width)
{
    int size = Width * Width * sizeof(float);
    float* Md, Nd, Pd;

    //设置调用内核函数时的线程数目
    dim3 dimBlock(Width, Width);
    dim3 dimGrid(1, 1);


    //在设备存储器上给M和N矩阵分配空间，并将数据复制到设备存储器中
    cudaMalloc(&Md, size);
    cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);
    cudaMalloc(&Nd, size);
    cudaMemcpy(Nd, N, size, cudaMemcpyHostToDevice);

    //在设备存储器上给P矩阵分配空间
    cudaMalloc(&Pd, size);
```

2.1.2 计算结果向主机传输

```
//内核函数调用，将在后续部分说明
//只使用了一个线程块(dimGrid(1,1))，此线程块中有Width*Width个线程
MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd, Width);

// 从设备中读取P矩阵的数据
cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost);

// 释放设备存储器中的空间
cudaFree(Md); cudaFree(Nd); cudaFree (Pd);
}
```

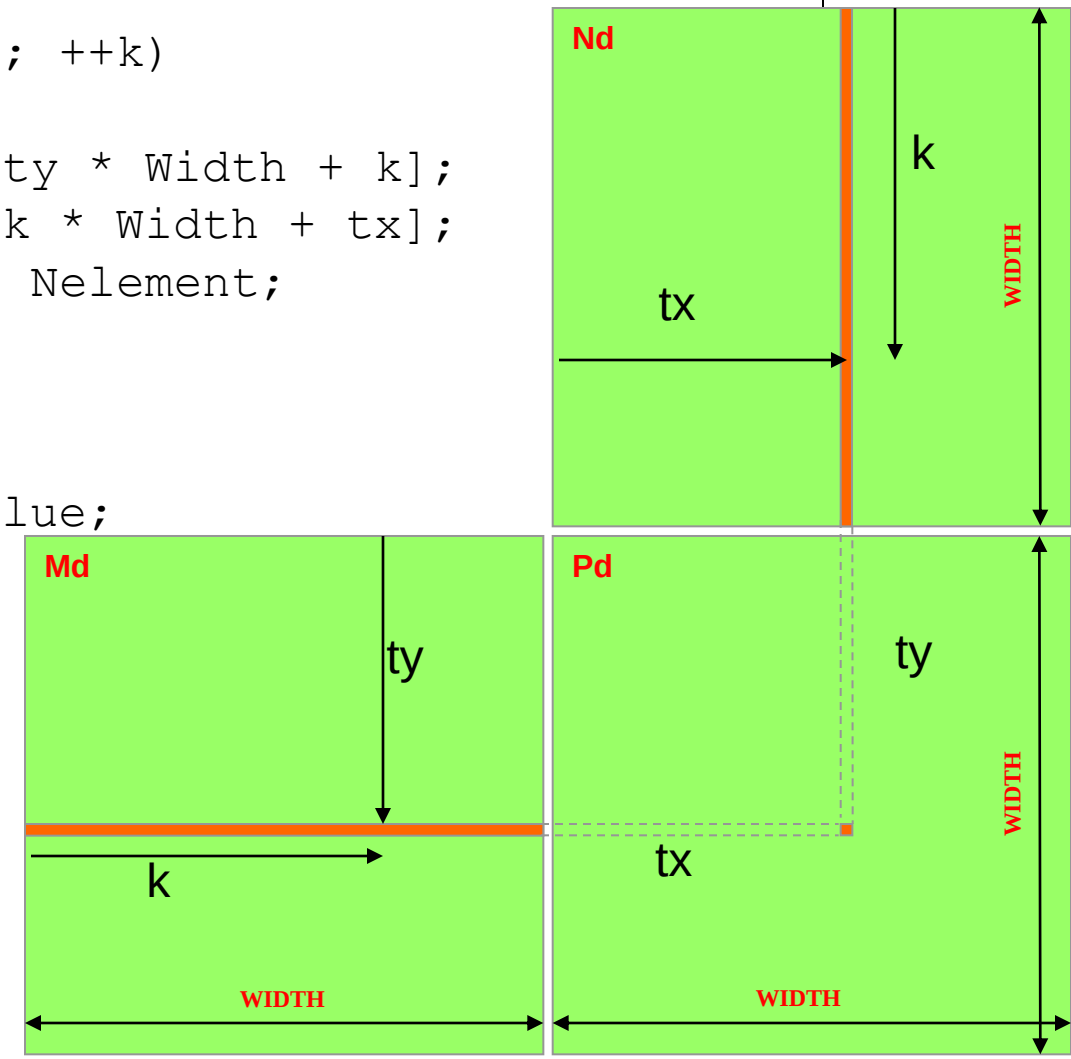
2.1.3 矩阵乘法的内核函数



```
// 矩阵乘法的内核函数——每个线程都要执行的代码
__global__ void MatrixMulKernel(float* Md, float* Nd, float*
Pd, int Width)
{
    // 2维的线程ID号
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // Pvalue用来保存被每个线程计算完成后的矩阵的元素
    float Pvalue = 0;
```

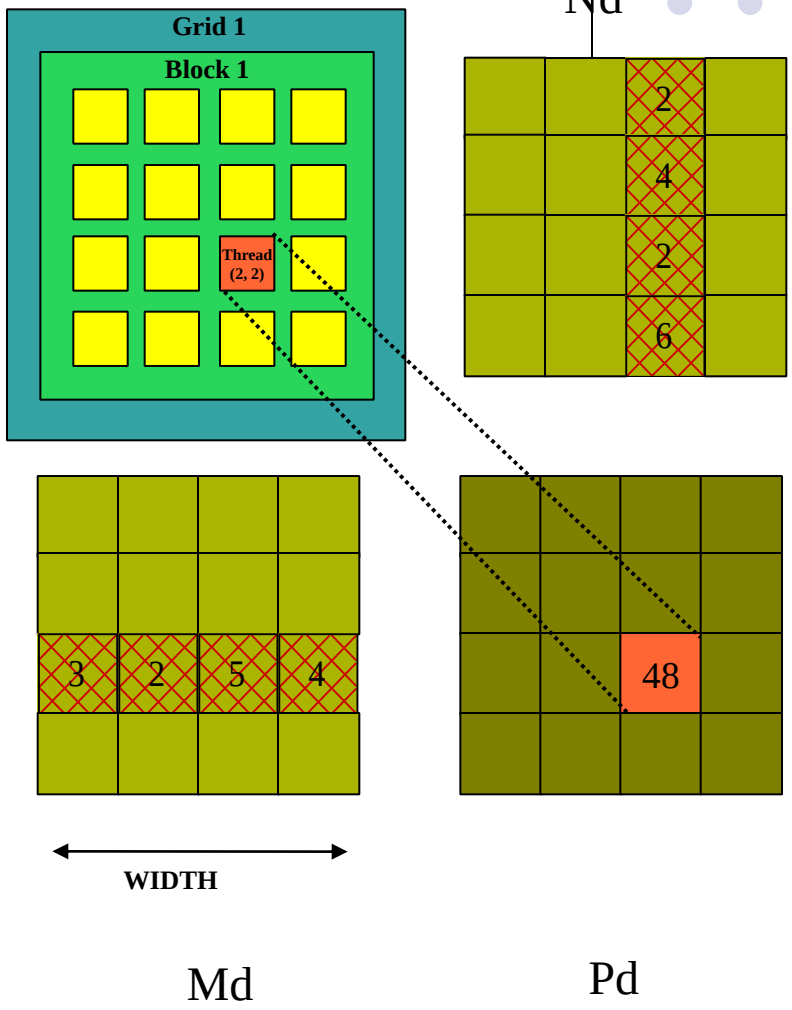
内核函数 Cont.

```
//每个线程计算一个元素
for (int k = 0; k < Width; ++k)
{
      float Melement = Md[ty * Width + k];
      float Nelement = Nd[k * Width + tx];
      Pvalue += Melement * Nelement;
}

// 将计算结果写入设备存储器中
Pd[ty * Width + tx] = Pvalue;
}
```



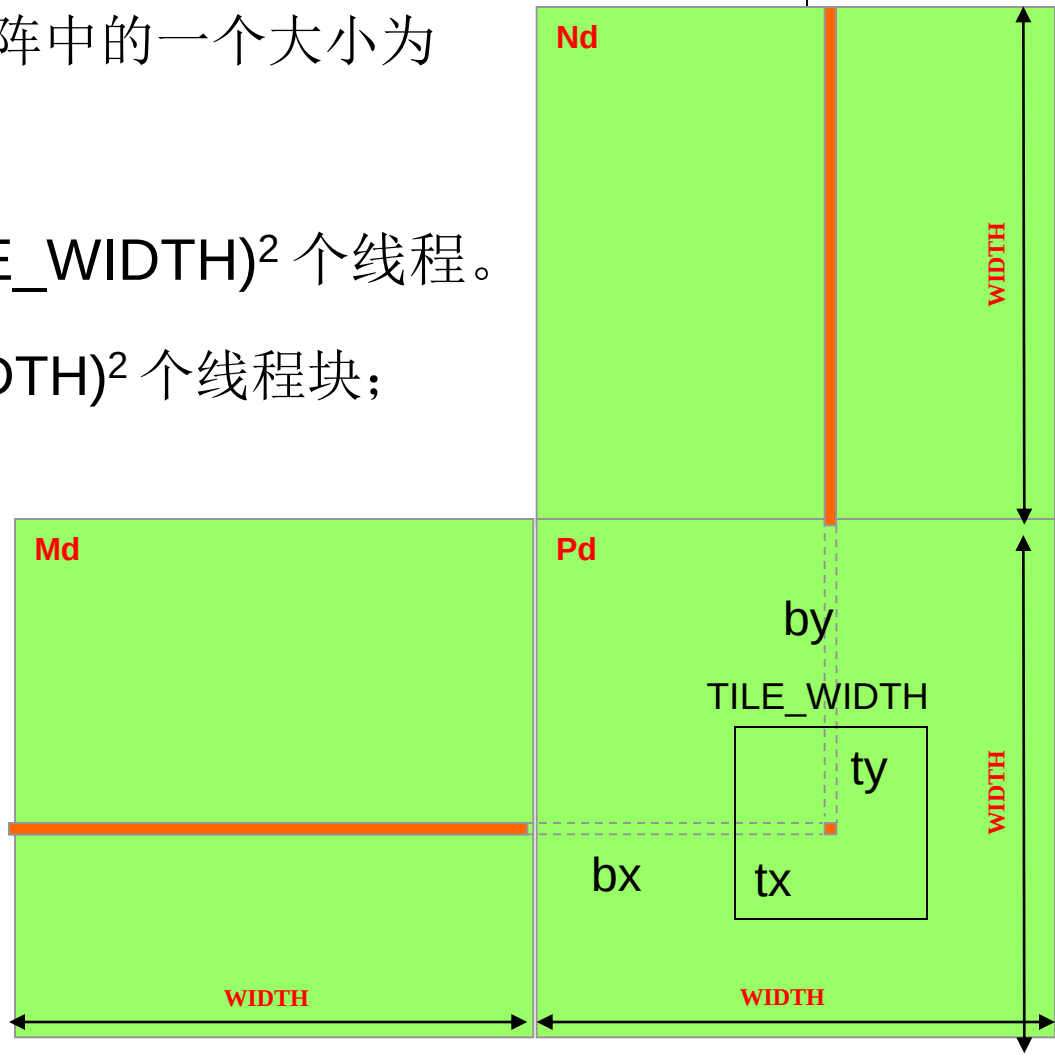
2.1.4 本方法讨论和存在问题

- 一个线程块中的每个线程计算Pd中的一个元素。
- 每个线程：
  - 载入矩阵Md中的一行；
  - 载入矩阵Nd中的一列；
  - 为每对Md和Nd元素执行了一次乘法和加法。
- 缺点：
  - 计算和片外存储器存访问比例接近1：1，受存储器延迟影响很大；
  - 矩阵的大小受到线程块所能容纳最大线程数（512个线程）的限制。



2.1.5 可处理任意大小矩阵的方法

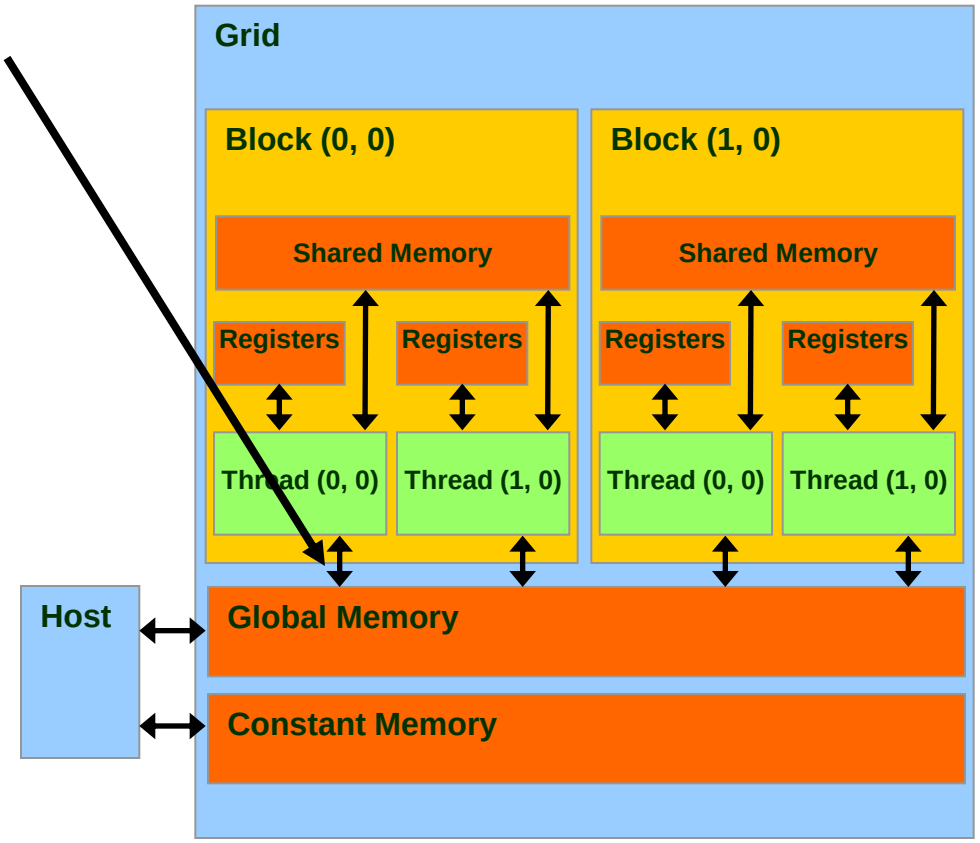
- 让每个线程块计算结果矩阵中的一个大小为(TILE_WIDTH)^{2}的子矩阵；
  - 每个线程块中有 (TILE_WIDTH)^2 个线程。
- 总共有(WIDTH/TILE_WIDTH)^2 个线程块；



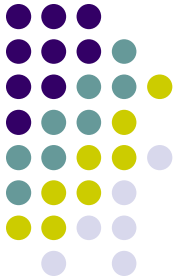
需要注意的是：当WIDTH/TILE_WIDTH大于最大的网格数量（64K）时，需要在内核函数附近设置一个循环！

2.1.6 G80显卡存储器带宽瓶颈

- 所有的线程都要访问全局存储器获取输入矩阵元素；
  - 每一次的单精度浮点乘法和加法需要两次的内存访问 (8 bytes)；
  - 全局存储器的访问带宽为86.4 GB/s；
  - 每秒钟可以读取21.6G个浮点数；
  - 每秒钟最多可以完成21.6GFlops。
- G80显卡的峰值速度为346.5GFlops；
- 效率仅为6%；
- 全局存储器成为计算瓶颈；
- 要充分使用高带宽的片上局部存储器。



Part Ⅳ
Example: Matrix Multiplication



1. 串行的矩阵乘法在CPU上的实现

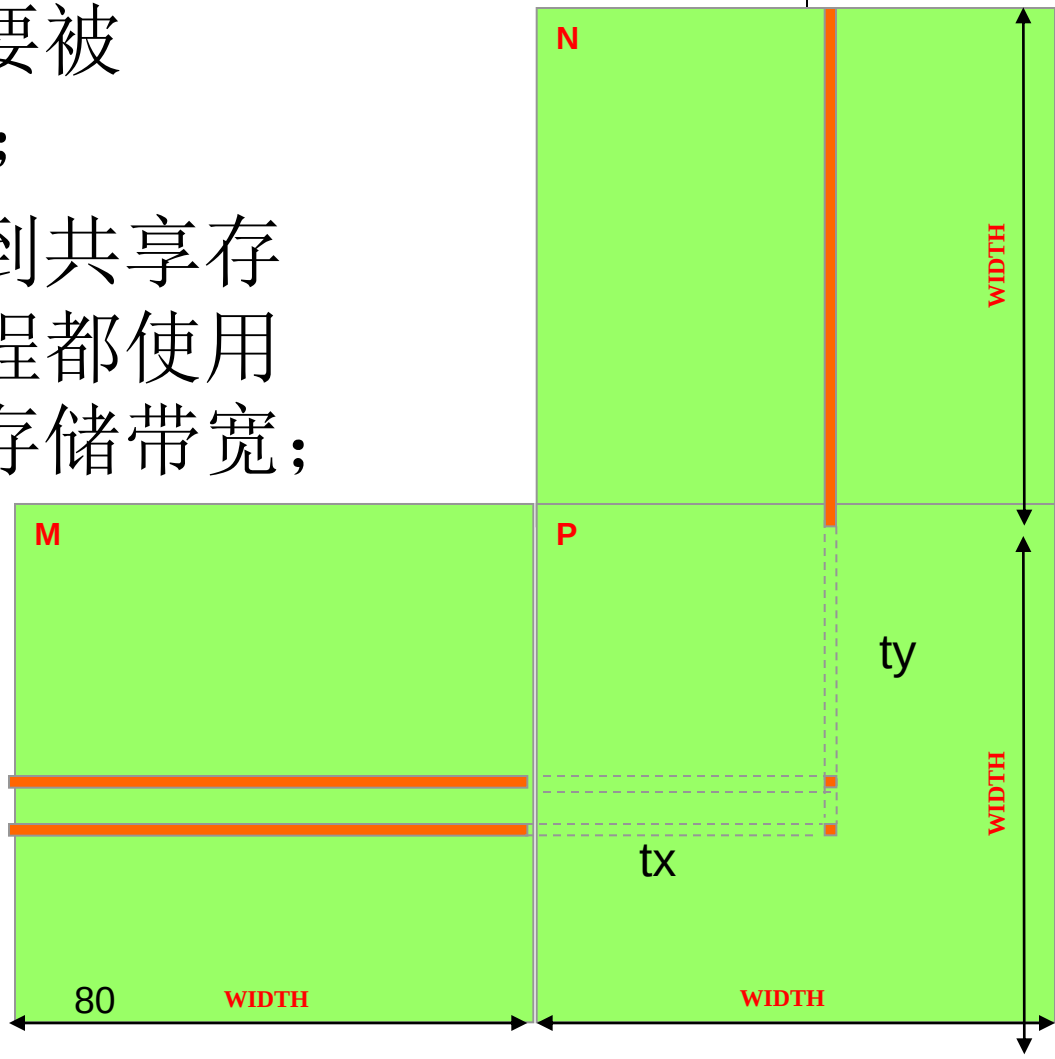
2. 并行的矩阵乘法在GPU上的实现

    2.1. 没有使用shared memory的实现

    2.2 使用了shared memory的实现

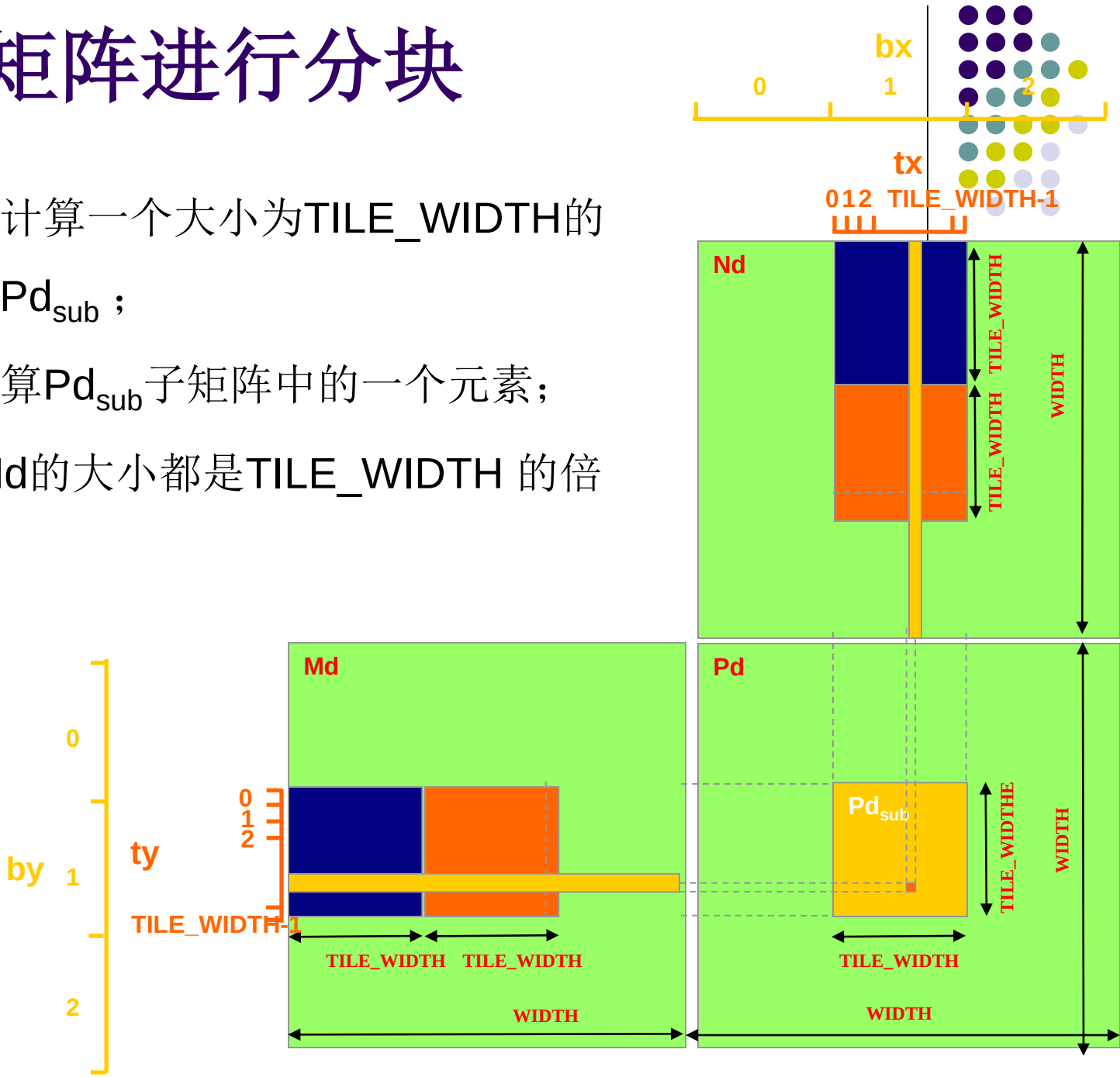
2.2.1 使用共享存储器以便重用全局存储器中的数据

- 每个输入元素都需要被WIDTH个线程读取；
- 将每个元素都装载到共享存储器中，让很多线程都使用本地数据以便减少存储带宽；
- 使用分片算法。

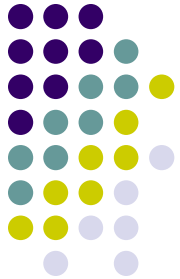


2.2.2 将矩阵进行分块

- 每个线程块计算一个大小为TILE_WIDTH的方形子矩阵Pd_{sub}；
- 每个线程计算Pd_{sub}子矩阵中的一个元素；
- 假设Md和Nd的大小都是TILE_WIDTH 的倍数。

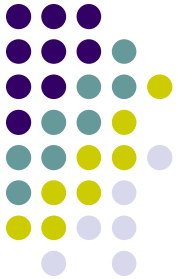


2.2.3 G80中首先需要考虑的事项



- 每个线程块内应该有较多的线程；
  - TILE_WIDTH=16时有 16*16 = 256 个线程。
- 分解为若干个线程块；
  - 一个1024*1024大小的Pd矩阵有64*64 = 4096 个线程块。
- 每个线程块从全局存储器将矩阵M和N的一小块读入到共享存储器中，然后完成计算；
  - 从全局存储器中读出2*256 = 512个单精度浮点数；
  - 完成 256 * (2*16) = 8,192 次浮点计算操作；
  - 浮点操作：全局存储器读出操作＝16: 1；
  - 全局存储器不再是性能瓶颈！

2.2.4 内核函数线程数配置



```
//每个线程块有TILE_WIDTH²个线程
dim3 dimBlock(TILE_WIDTH, TILE_WIDTH);

//有(Width/TILE_WIDTH)²个线程块
dim3 dimGrid(Width/TILE_WIDTH, Width/TILE_WIDTH);

//调用内核函数
MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd, Width);
```

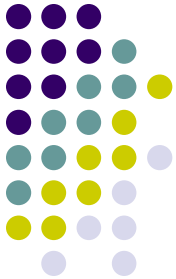
2.2.5 内核函数

```
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
//获得线程块号
int bx = blockIdx.x;
int by = blockIdx.y;

//获得块内的线程号
int tx = threadIdx.x;
int ty = threadIdx.y;

//Pvalue: 线程计算完成后的子矩阵元素——自动变量
float Pvalue = 0;

//循环，遍历M和N的所有子矩阵
for (int m = 0; m < Width/TILE_WIDTH; ++m) {
        //此处代码在下面
};
```

2.2.6 将数据装入共享存储器

```
// 获取指向当前矩阵M子矩阵的指针Msub
Float* Mdsub = GetSubMatrix(Md, m, by, Width);

//获取指向当前矩阵N的子矩阵的指针Nsub
Float* Ndsub = GetSubMatrix(Nd, bx, m, Width);

//共享存储器空间声明
__shared__float Mds[TILE_WIDTH][TILE_WIDTH];
__shared__float Nds[TILE_WIDTH][TILE_WIDTH];

// 每个线程载入M的子矩阵的一个元素
Mds[ty][tx] = GetMatrixElement(Mdsub, tx, ty);

//每个线程载入N的子矩阵的一个元素
Nds[ty][tx] = GetMatrixElement(Ndsub, tx, ty);
```

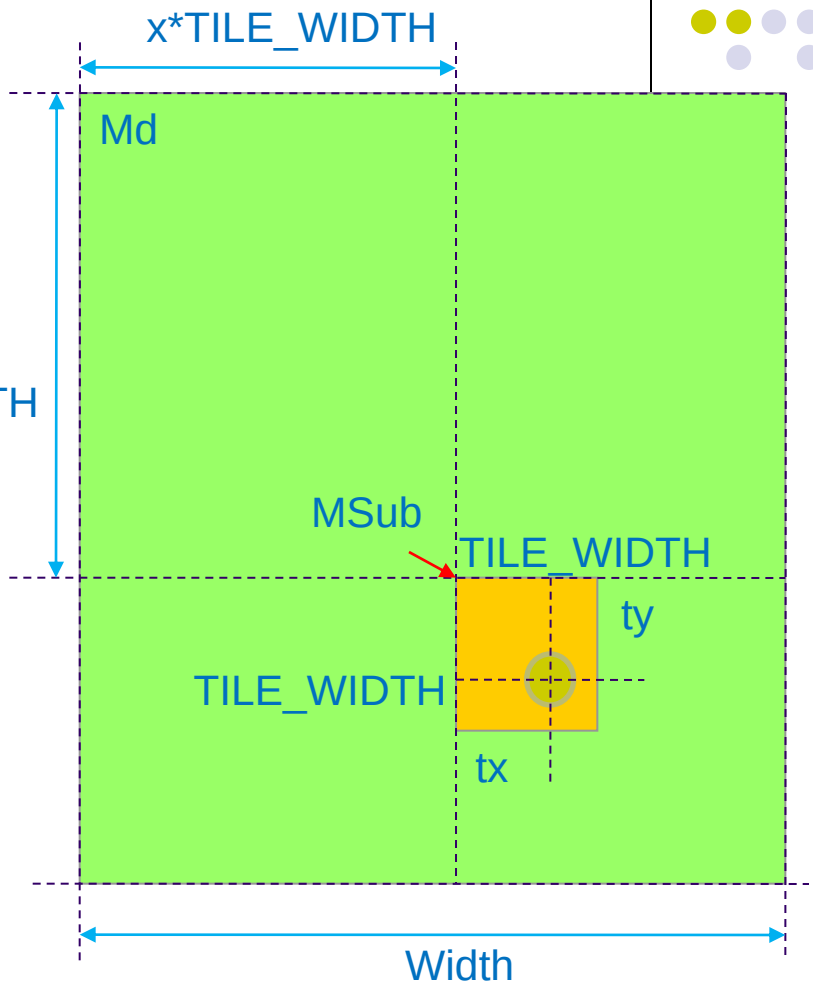
2.2.7 从shared memory中取数、计算



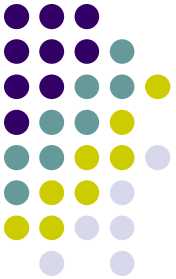
```
//同步，在计算之前，确保子矩阵所有的元素都已载入共享存储器中
    __syncthreads();

//每个线程计算线程块内子矩阵中的一个元素
for (int k = 0; k < TILE_WIDTH; ++k)
     Pvalue += Mds[ty][k] * Nds[k][tx];

//同步，确保重新载入新的M和N子矩阵数据前，上述计算操作已全部完成
__syncthreads();
}
```

2.2.8 一些其它代码

- GetSubMatrix(Pd, x, y, Width)
  - 获取第(x, y)号子矩阵的起始地址
  - Pd + y*TILE_WIDTH*Width + x*TILE_WIDTH;
- GetMatrixElement(Pdsub,tx,ty,Width)
  - 获取子矩阵中某个元素的地址
  - *(Pdsub+ty*Width+tx);



2.2.9 CUDA 代码 – 保存结果



```
// 获取指向矩阵P的子矩阵的指针
Matrix Psub = GetSubMatrix(P, bx, by);

//向全局存储器写入线程块计算后的结果子矩阵
//每个线程写入一个元素
SetMatrixElement(Psub, tx, ty, Pvalue);
```

Activity 9

- 思考题1：

  对于矩阵乘向量，如何有效进行线程设计和并行算法设计？

- 思考题2：

  对于稀疏矩阵乘向量，又存在什么问题及如何优化？

主要内容

Ⅰ. Introduction to GPU

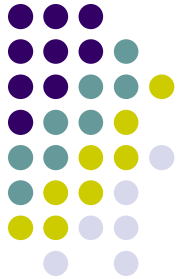
Ⅱ. GPU Architecture

Ⅲ. CUDA Programming

Ⅳ. Example: Matrix Multiplication

Ⅴ. Performance and Optimization

Part Ⅴ
Performance and Optimization



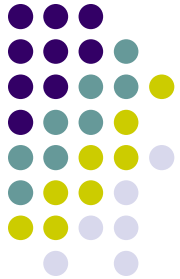
1. Global Memory Access

2. Shared Memory Access

3. Memory Latency Hiding

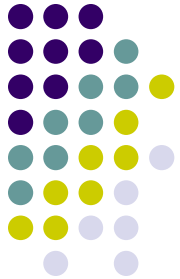
4. Algorithm Optimization for the GPU

1.1 Global Memory访存优化方法



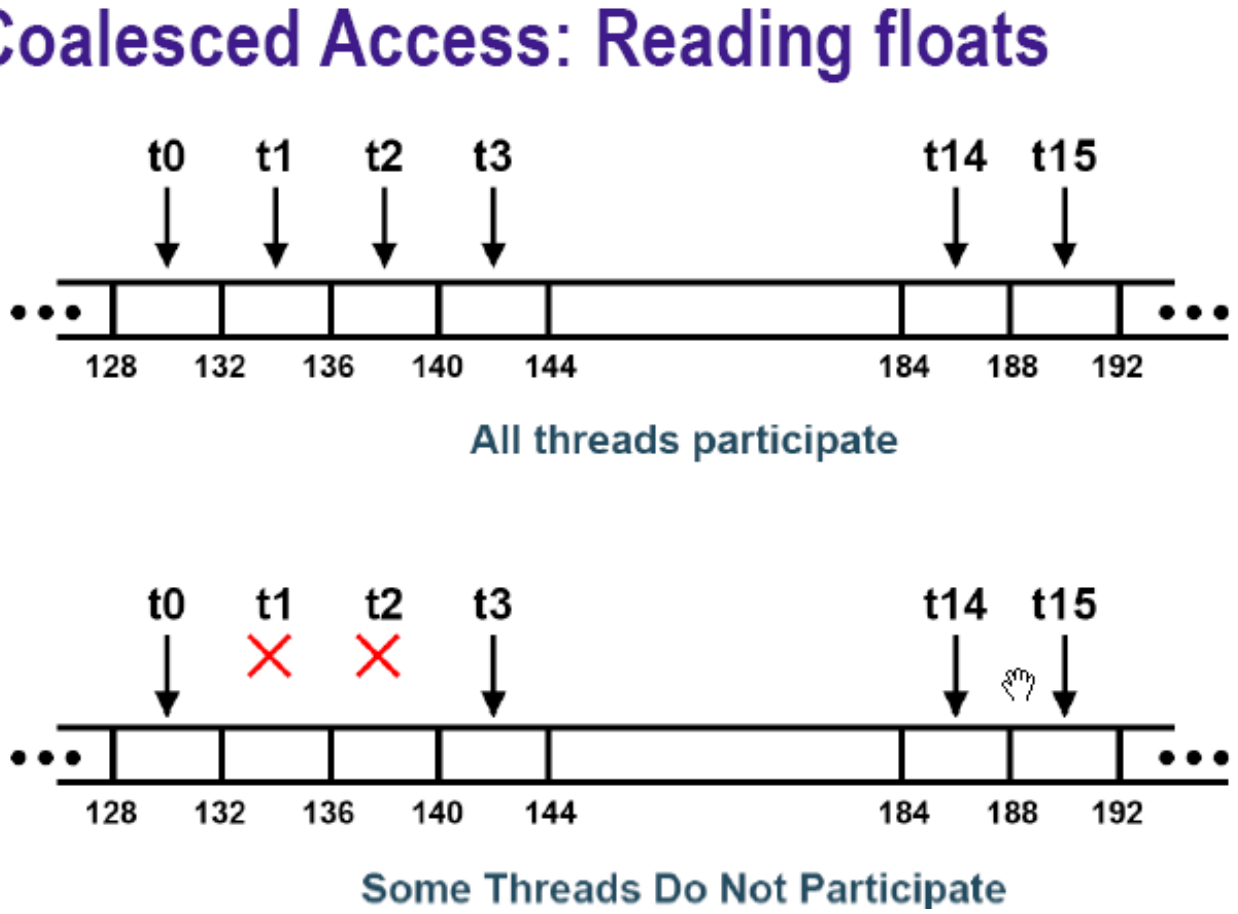
- 全局存储器延时：400~600 clock cycles；
  - 经常成为性能瓶颈。
- 优化措施：
  - 采用coalesced memory access；
  - 使用shared memory达到coalesced memory access和block内threads共享访问；
  - 增加访存线程，掩藏存储器延时；
    - 4次顺序访问至少需要4*400=1,600 cycle。
    - 4个并行线程，可以只需要：400+1+1+1=403 cycle。

1.2 Coalesced Access及其例子

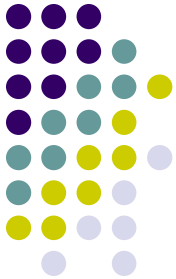


- half warp(16 threads)的coalesced access:
  - 顺序访问连续的global memory区域：
    - warp中第k个线程访问第k个地址；
    - 64bytes – each thread reads a word: int, float, …；
    - 128bytes – each thread reads a double-word: int2, float2, …；
    - 256bytes – each thread reads a quad-word: int4, float4, …；
  - 访问起始地址要求：
    - global memory区域的起始地址必须是该区域数据类型尺寸的整数倍；
- 例外：可以有某些中间线程不参加。

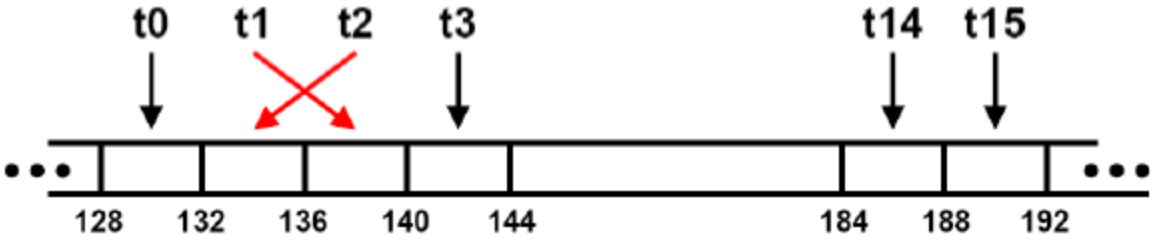
Coalesced Access示例



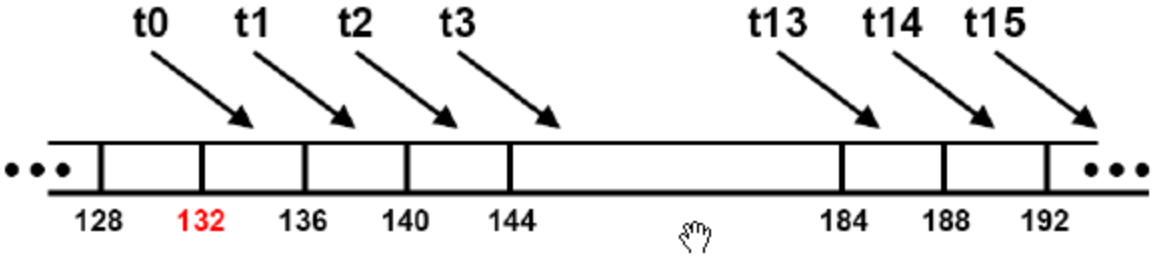
Coalesced Access示例 (Cont.)



Uncoalesced Access: Reading floats

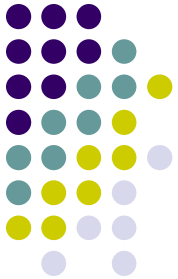


Permuted Access by Threads



Misaligned Starting Address (not a multiple of 64)

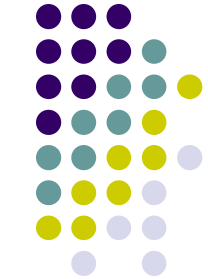
Coalesced Access示例 (Cont.)



Coalescing: Timing Results

- Experiment:
  - Kernel: read a float, increment, write back
  - 3M floats (12MB)
  - Times averaged over 10K runs
- 12K blocks x 256 threads:
  - 356µs – coalesced
  - 357µs – coalesced, some threads don't participate
  - 3,494µs – permuted/misaligned thread access

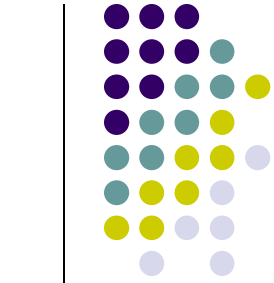
Coalesced Access示例(Cont.)



Uncoalesced float3 Code

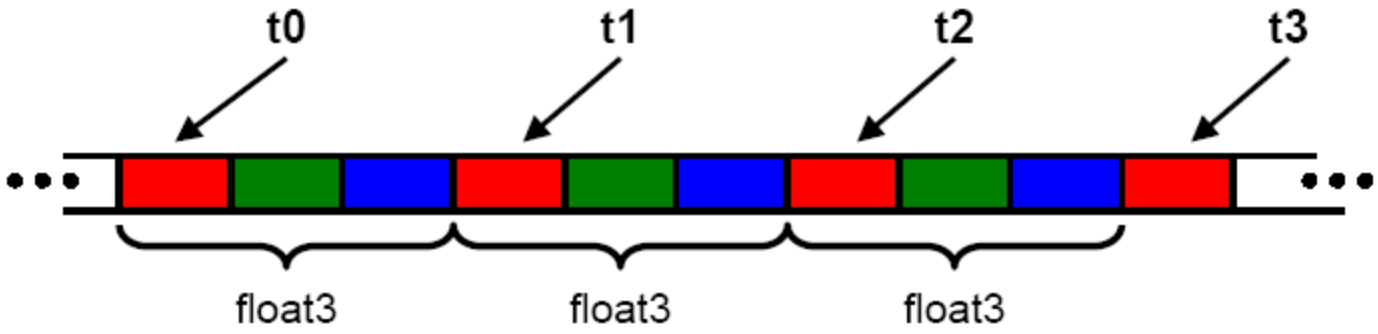
```
__global__ void accessFloat3(float3 *d_in, float3 d_out)
{
    int index = blockIdx.x * blockDim.x + threadIdx.x;
    float3 a = d_in[index];

    a.x += 2;
    a.y += 2;
    a.z += 2;

    d_out[index] = a;
}
```

Coalesced Access示例(Cont.)



Uncoalesced Access: float3 Case

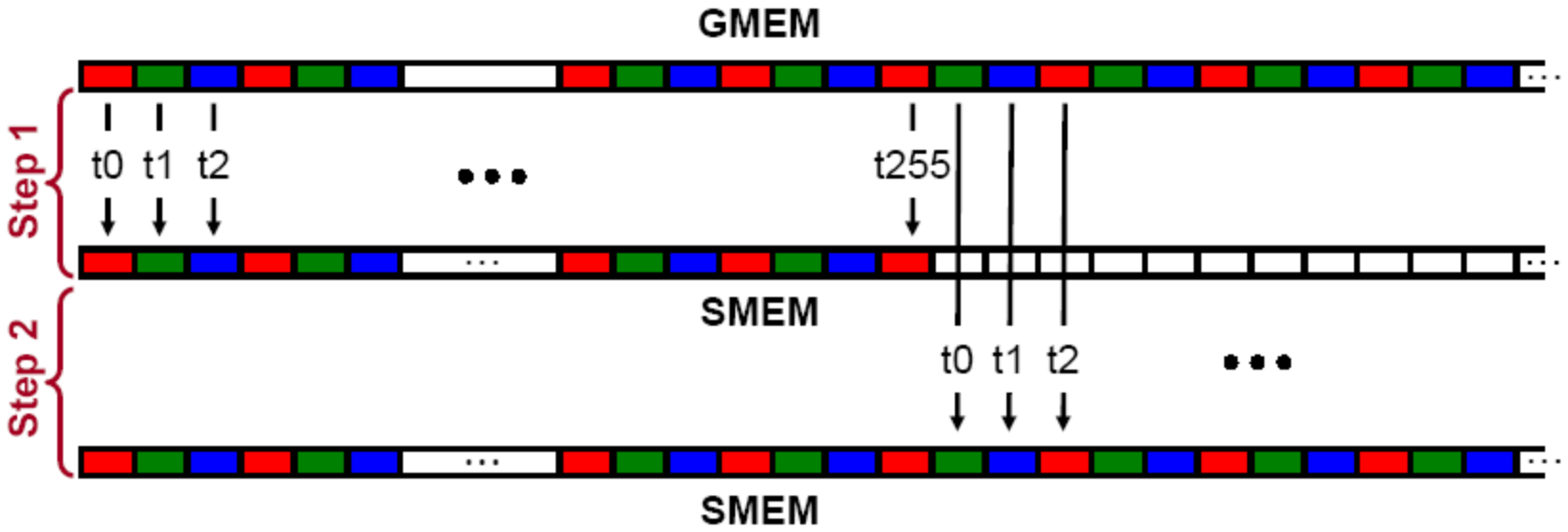
- float3 is 12 bytes
- Each thread ends up executing 3 reads
  - sizeof(float3) ≠ 4, 8, or 12
  - Half-warp reads three 64B non-contiguous regions



First read

Coalesced Access示例(Cont.)

Coalescing float3 Access



Similarly, Step3 starting at offset 512

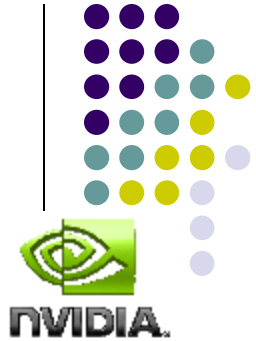
Coalesced Access示例(Cont.)

Coalesced Access: float3 Case

- Use shared memory to allow coalescing
  - Need sizeof(float3)*(threads/block) bytes of SMEM
  - Each thread reads 3 scalar floats:
    - Offsets: 0, (threads/block), 2*(threads/block)
    - These will likely be processed by other threads, so sync
- Processing
  - Each thread retrieves its float3 from SMEM array
    - Cast the SMEM pointer to (float3*)
    - Use thread ID as index
  - Rest of the compute code does not change!

Coalesced Access示例(Cont.)

Coalesced float3 Code



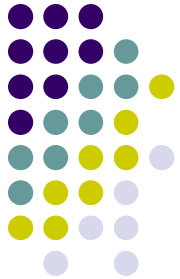
Read the input through SMEM

Compute code is not changed

Write the result through SMEM

```
__global__ void accessInt3Shared(float *g_in, float *g_out)
{
    int index = 3 * blockIdx.x * blockDim.x + threadIdx.x;
    __shared__ float s_data[256*3];
    s_data[threadIdx.x]       = g_in[index];
    s_data[threadIdx.x+256] = g_in[index+256];
    s_data[threadIdx.x+512] = g_in[index+512];
    __syncthreads();
    float3 a = ((float3*)s_data)[threadIdx.x];

    a.x += 2;
    a.y += 2;
    a.z += 2;

    ((float3*)s_data)[threadIdx.x] = a;
    __syncthreads();
    g_out[index]       = s_data[threadIdx.x];
    g_out[index+256] = s_data[threadIdx.x+256];
    g_out[index+512] = s_data[threadIdx.x+512];
}
```

Part Ⅴ
Performance and Optimization



1. Global Memory Access

2. Shared Memory Access

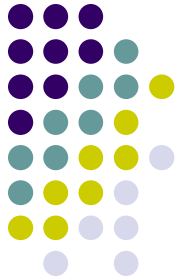
3. Memory Latency Hiding

4. Algorithm Optimization for the GPU

2.1 利用Shared Memory优化方法

- 几百倍快于global memory；
- 线程之间可以通过shared memory共享数据，进行合作计算；
- 使用一个或少量线程装载数据及在thread block内共享数据；
- 通过shared memory进行数据重组避免global memory的non-coalesceable；
- 使用shared memory时要避免bank conflicts。

Part Ⅴ
Performance and Optimization



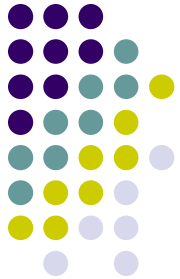
1. Global Memory Access

2. Shared Memory Access

3. Memory Latency Hiding

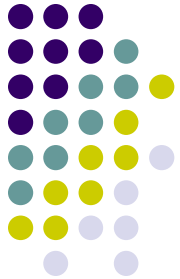
4. Algorithm Optimization for the GPU

3. Latency Hiding的方法



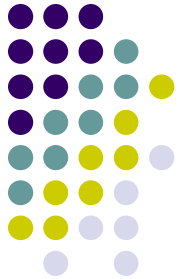
- 尽量增加SM上的线程数量，提高Occupancy（实际并发运行的warp个数/最大可能并发运行的warp个数）；
  - 限制条件：# of registers和# of shared memory, 一个SM可以并行处理768threads；
  - 100%Occupancy: 2 blocks X 384 threads；
    3 blocks X 256 threads；
    4 blocks X 192 threads；
    6 blocks X 128 threads；
    8 blocks X 96 threads；
  - 最小存储器延时：Occupancy≥50% and threads/blocks≥128。
- Thread block内的线程个数应该是warp size的整数倍，避免在一个warp内有分支语句。

3. Latency Hiding的方法(Cont.)



- Grid/Block Size Heuristics；
  - # of blocks / # of SMs > 1；
    - ✓ 每个SM至少有一个thread block可以执行。
  - 更好的选择：# of blocks / # of SMs > 2；
    - ✓ 每个SM有多个thread block可以执行。
  - 每个block占用SM一半以下的资源；
  - # of blocks > 100 使得适应将来的结构。

Part Ⅴ
Performance and Optimization



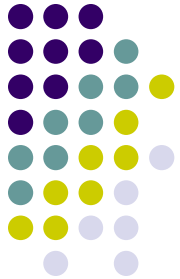
1. Global Memory Access

2. Shared Memory Access

3. Memory Latency Hiding

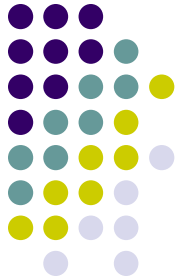
4. Algorithm Optimization for the GPU

4. Algorithm Optimization for the GPU



- 最大化独立并行性；
- 最大化算术计算强度(math/bandwidth)；
- 均匀划分使得GPU各个SM负载均衡；
- 降低资源使用，以便多个thread block在SM上运行；
- GPU上做更多的计算，避免与CPU数据传输；
  - device memory – host memory带宽远低于device memory – device带宽；
    - ✓ 4GB/s peak (PCI-ex16) vs. 80GB/s peak (Quadro FX5600)；
  - 计算中的数据结构在GPU上分配、操作、释放；
  - 组合整块数据传送要快于分小块多次传送。

了解一些运算的成本



- 4 clock cycle:
  - Floating point: add, multiply, fused multiply-add；
  - Integer add, bitwise operations, compare, min, max；
- 16 clock cycles:
  - reciprocal, reciprocal square root, log(x), 32-bit integer；
  - Multiplication；
- 32 clock cycles:
  - __sin(x), __cos(x) and __exp(x)；
- 36 clock cycles:
  - Floating point division (24-bit version in 20 cycles)；
- Particularly costly:
  - Integer division, modulo；
  - Remedy: Replace with shifting whenever possible；
- Double precision (when available) will perform at half the speed。