

并行计算

Parallel Computing

主讲 孙经纬

2024年 春季学期

概要

- 第三篇 并行编程
 - 第十三章 并行程序设计基础
 - 第十四章 共享存储系统并行编程
 - 第十五章 分布存储系统并行编程
 - **补充章节1 GPU并行编程**
 - 补充章节2 关于并行编程的更多话题

GPU并行编程

I. GPU简介

II. GPU结构

III. CUDA编程

IV. 性能与优化

V. 案例：矩阵乘法

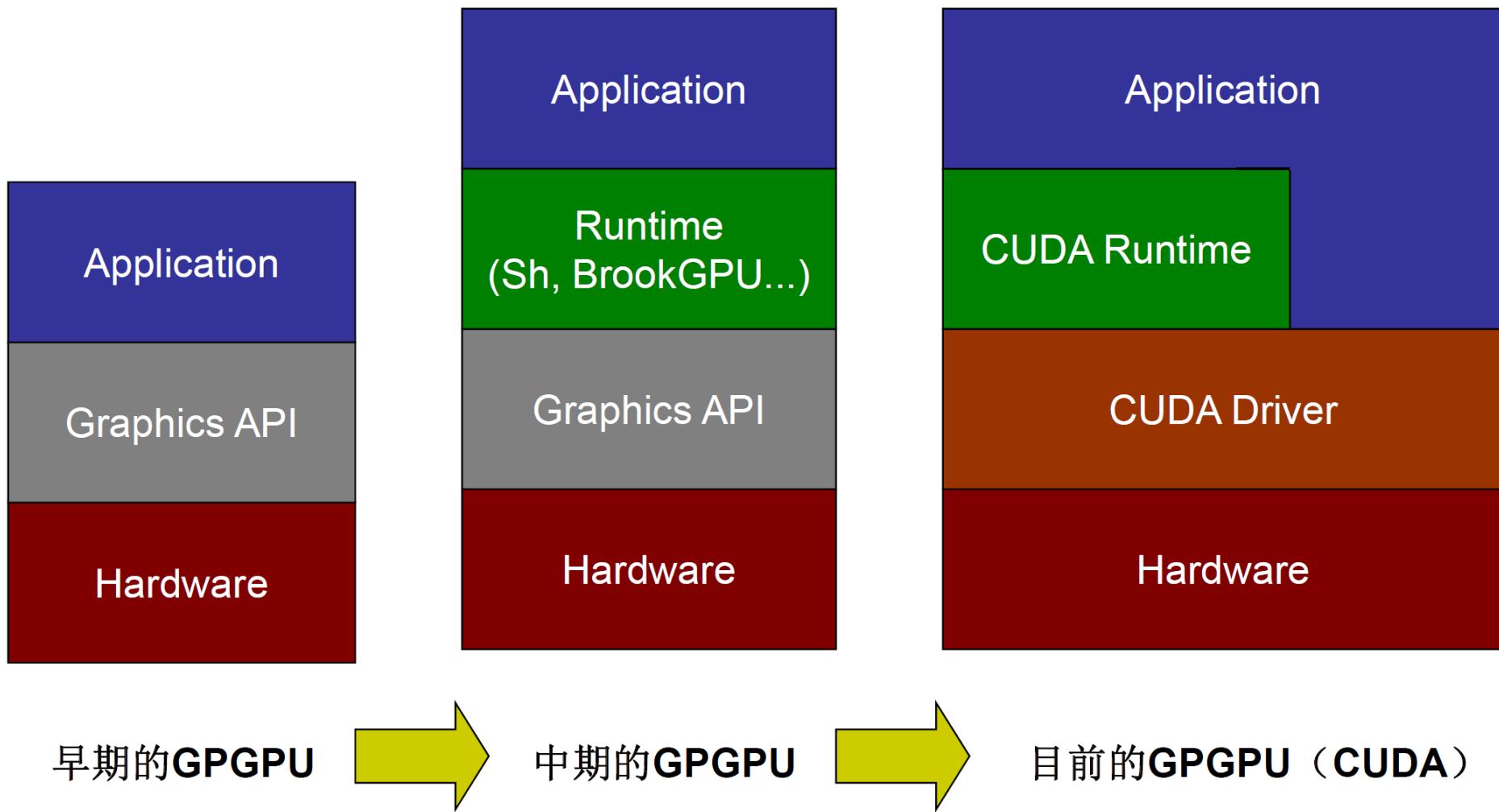
GPU简介

- 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 (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，已经演化成为一个重要的并行计算平台
- 一个必须重视的研究领域和技术

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Radeon 9700, 2003年NVIDIA推出GeForce FX
 - 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

GPU的发展阶段

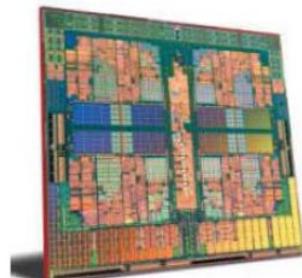


GPU的发展阶段

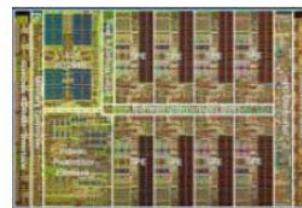
- GPU可编程性不断增强，特别是CUDA等编程环境的出现，使GPU通用计算编程的复杂性大幅度降低。
- 由于可编程性、功能、性能不断提升和完善，GPU已演化为一个新型可编程高性能并行计算资源。

CPU和GPU

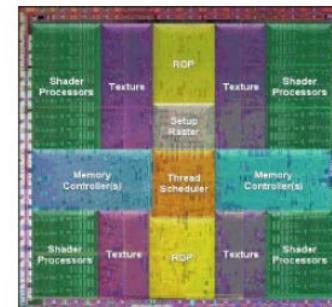
- 多核(multicore)CPU试图在基本保持单核性能的前提下通过扩展核数来提高总体计算性能。
- GPU则侧重在一块芯片上集成多个较低功耗的核心
 - 单个核心频率基本不变 (一般在1-3GHz)
 - 设计重心转向到多核的集成技术，核心数量不断增长
- GPU是一种特殊的多核处理器



Quad-core Opteron



IBM Cell Broadband Engine



nVidia GT200

CPU和GPU

对CPU和GPU做了非常清晰的比较和总结

Fermi white paper by Peter N. Glaskowsky:

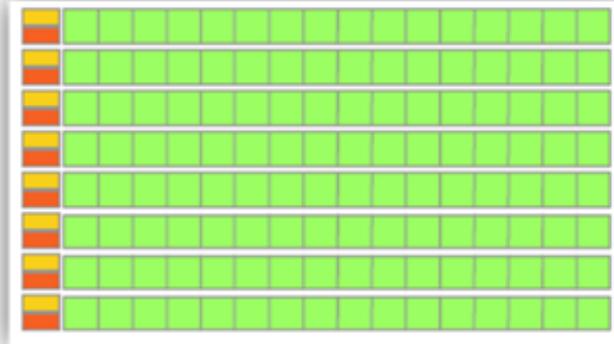
- GPU computing isn't meant to replace CPU computing. Each approach has advantages for certain kinds of software. As explained earlier, CPUs are optimized for applications where most of the work is being done by a limited number of threads, especially where the threads exhibit high data locality, a mix of different operations, and a high percentage of conditional branches.
- GPU design aims at the other end of the spectrum: applications with multiple threads that are dominated by longer sequences of computational instructions. Over the last few years, GPUs have become much better at thread handling, data caching, virtual memory management, flow control, and other CPU-like features, but the distinction between computationally intensive software and control-flow intensive software is fundamental.

CPU



- * Low compute density
- * Complex control logic
- * Large caches (L1\$/L2\$, etc.)
- * Optimized for serial operations
 - Fewer execution units (ALUs)
 - Higher clock speeds
- * Shallow pipelines (<30 stages)
- * Low Latency Tolerance
- * Newer CPUs have more parallelism

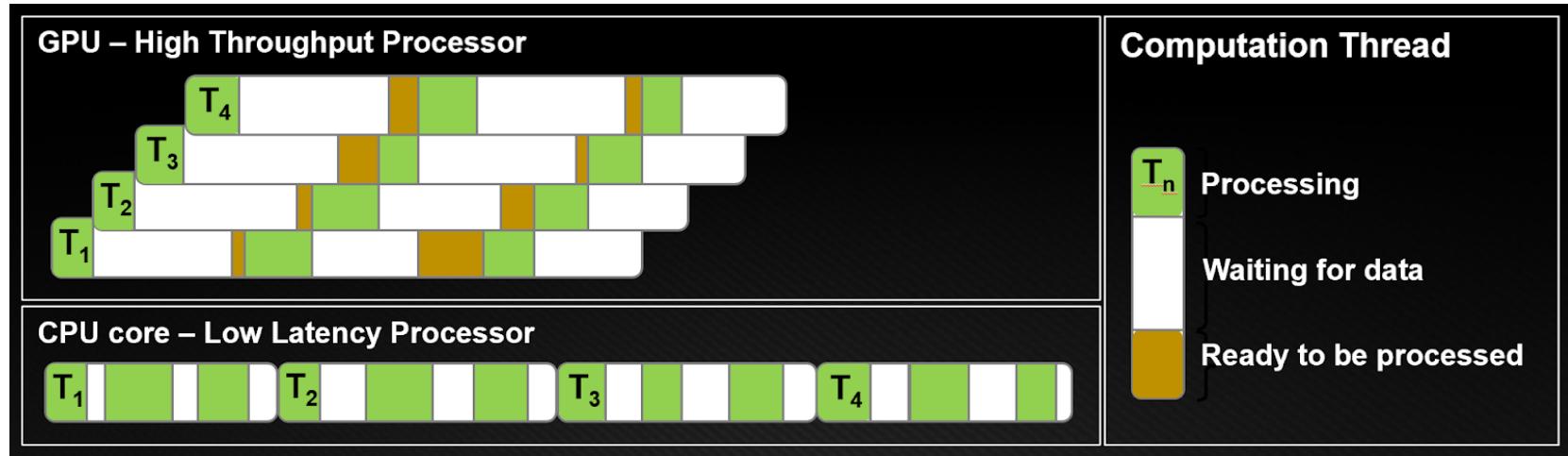
GPU



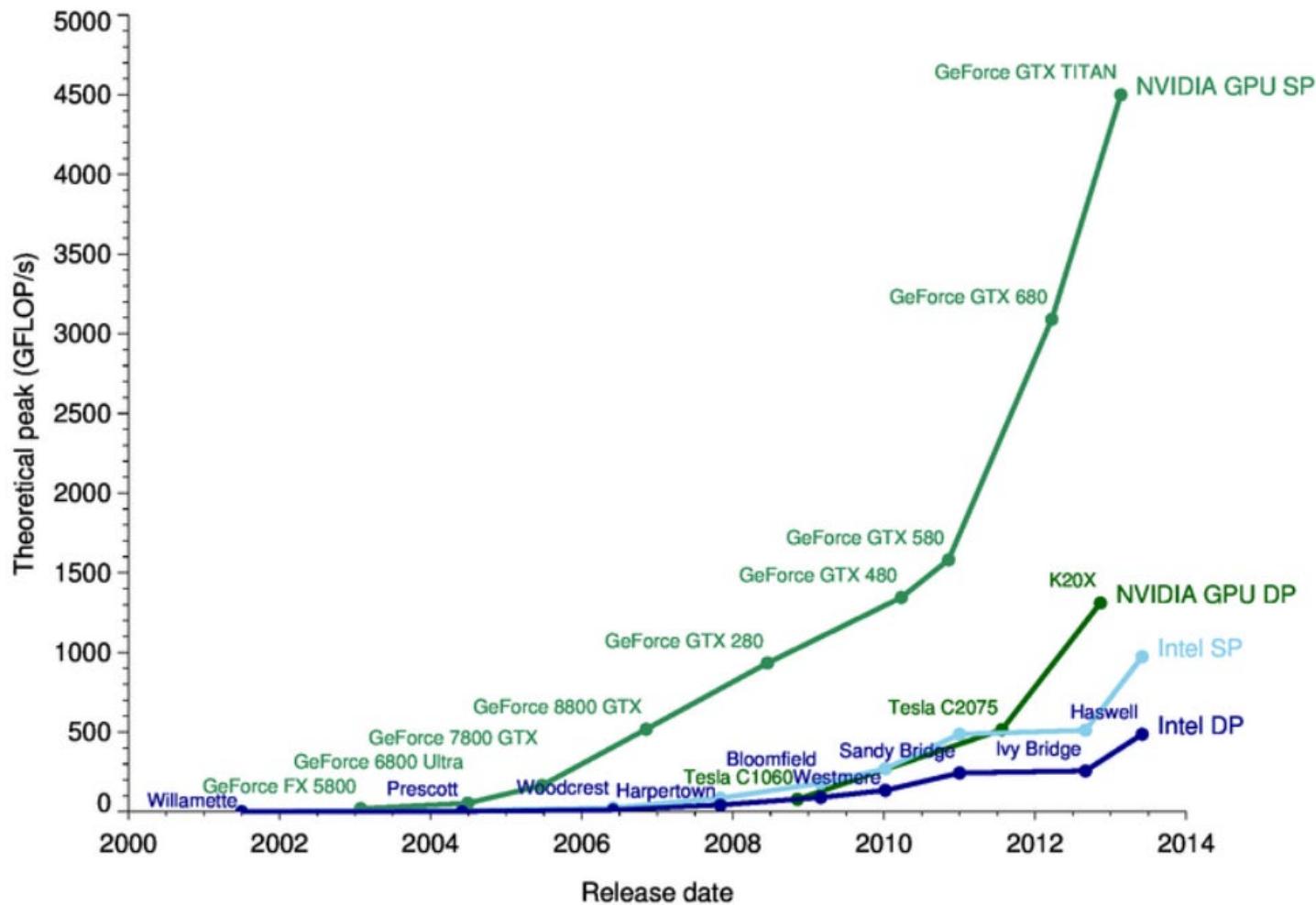
- * High compute density
- * High Computations per Memory Access
- * Built for parallel operations
 - Many parallel execution units (ALUs)
 - Graphics is the best known case of parallelism
- * Deep pipelines (hundreds of stages)
- * High Throughput
- * High Latency Tolerance
- * Newer GPUs:
 - Better flow control logic (becoming more CPU-like)
 - Scatter/Gather Memory Access
 - Don't have one-way pipelines anymore

CPU和GPU

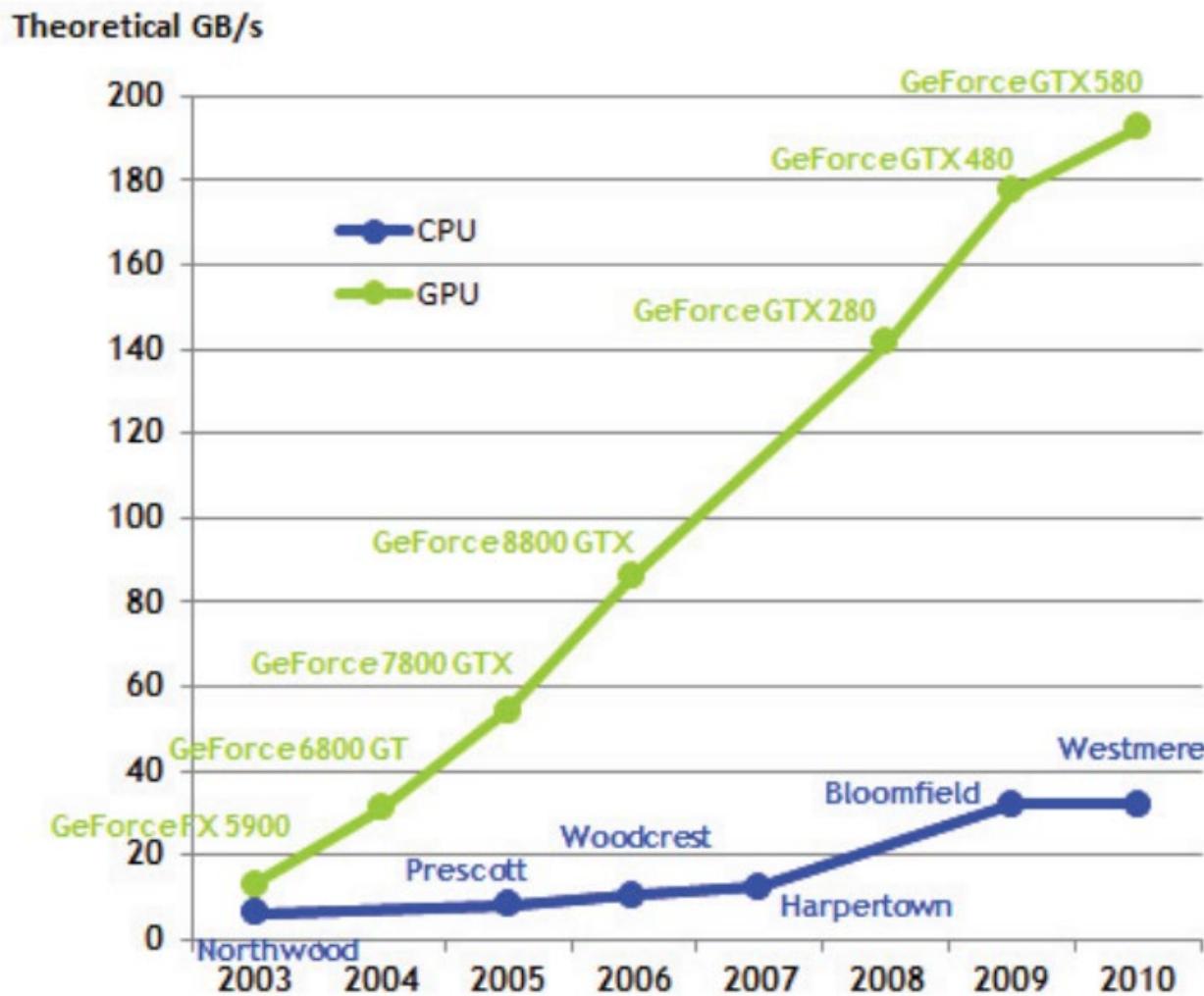
- CPU: 更多资源用于缓存和逻辑控制
- GPU: 更多资源用于计算, 适用于高并行性、大规模数据密集型、可预测的计算模式。



CPU和GPU 浮点计算能力对比



CPU和GPU 存储器带宽对比

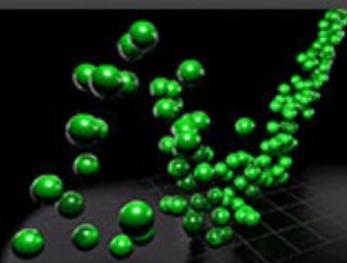


GPU应用

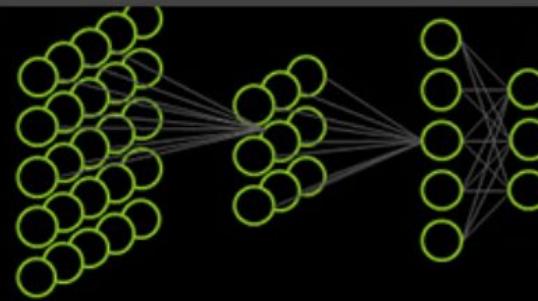
Domains with CUDA-Accelerated Applications

CUDA accelerates applications across a wide range of domains from image processing, to deep learning, numerical analytics and computational science.

COMPUTATIONAL CHEMISTRY



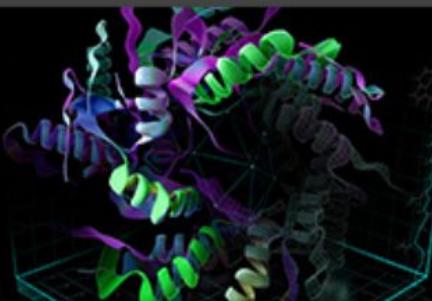
MACHINE LEARNING



DATA SCIENCE



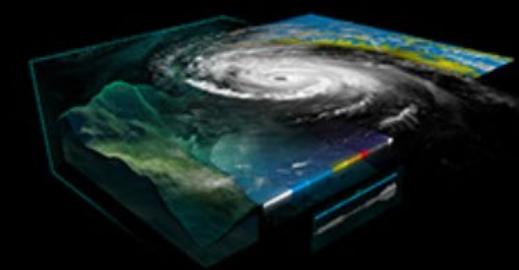
BIOINFORMATICS



COMPUTATIONAL FLUID DYNAMICS



WEATHER AND CLIMATE



GPU并行编程

I. GPU简介

II. GPU结构

III. CUDA编程

IV. 性能与优化

V. 案例：矩阵乘法

两个发展路线

支持通用计算的异构加速器曾经有两个主要的分支

1. 基于流处理器阵列的主流GPU结构

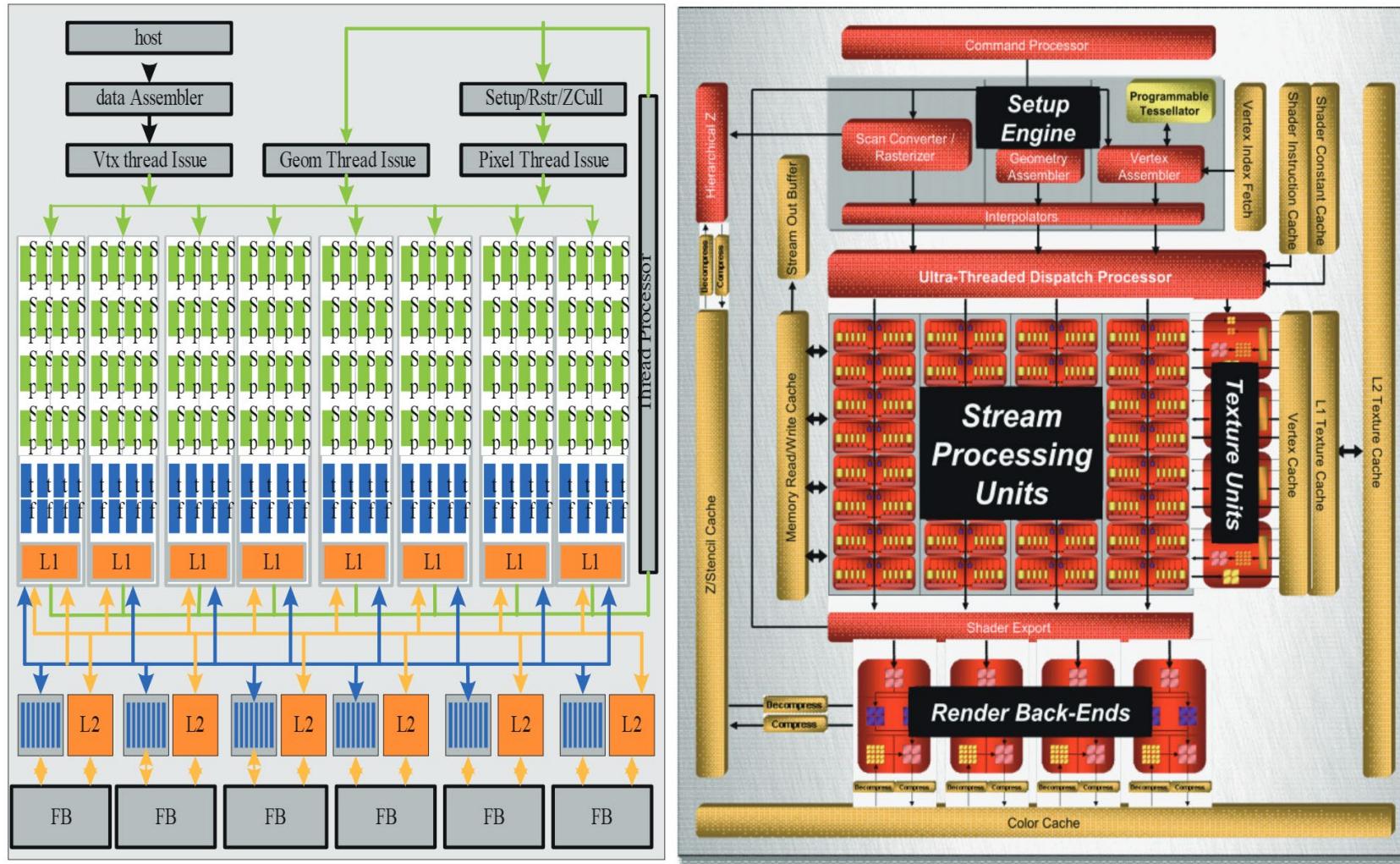
以NVIDIA和ATI（AMD）的GPU为代表

2. 基于通用计算核心的结构

以Intel Larrabee、Xeon Phi为代表

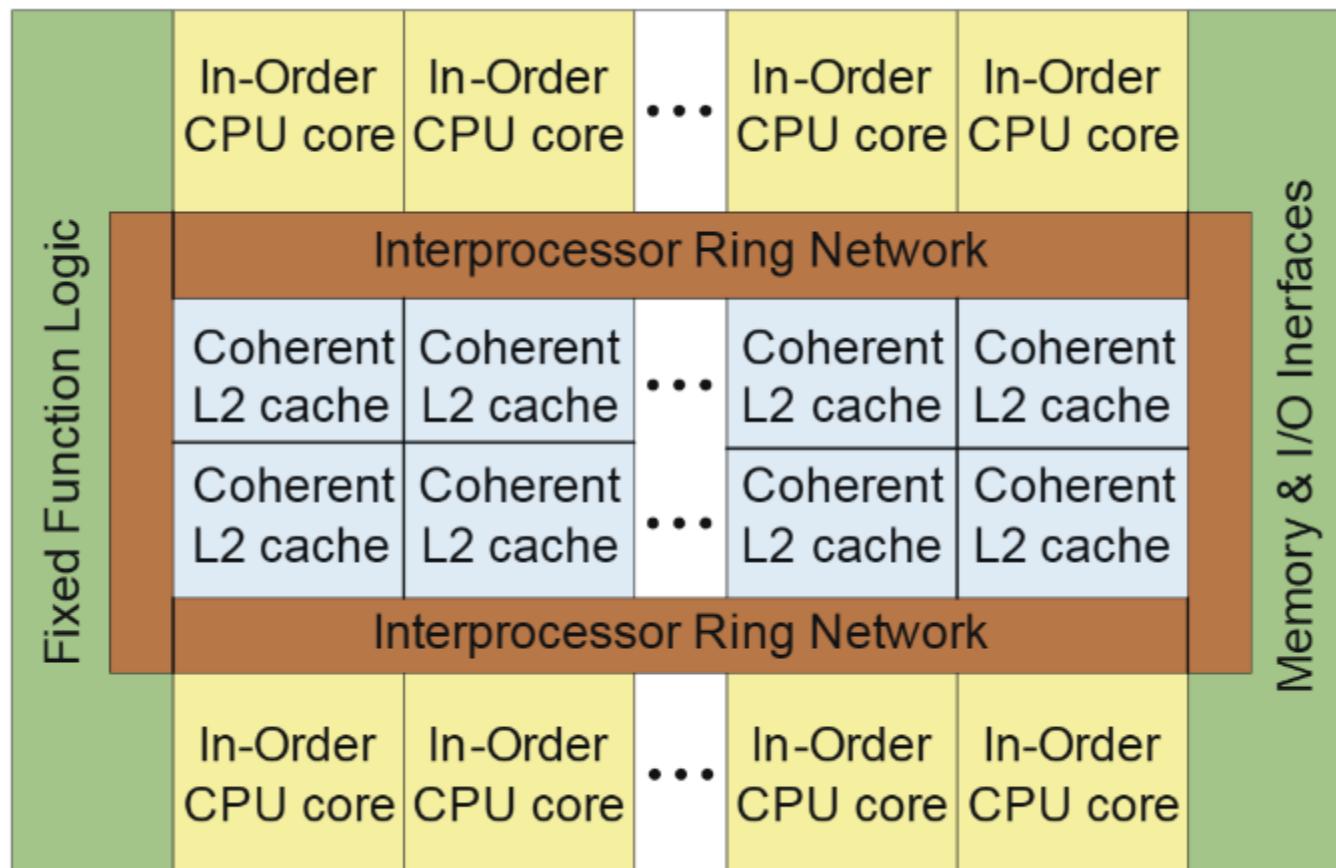
- 前者具有更高的并行计算性能，而后者则在可编程性上具有更大的优势

基于流处理器阵列的GPU结构图



GeForce 8800GTX (左)、HD 2900 (右)

基于通用计算核心的结构图



Larrabee多核结构示意图

Intel的发展路线

- Larrabee

Larrabee x86 GPU并未发布就于2009年宣告项目终止了



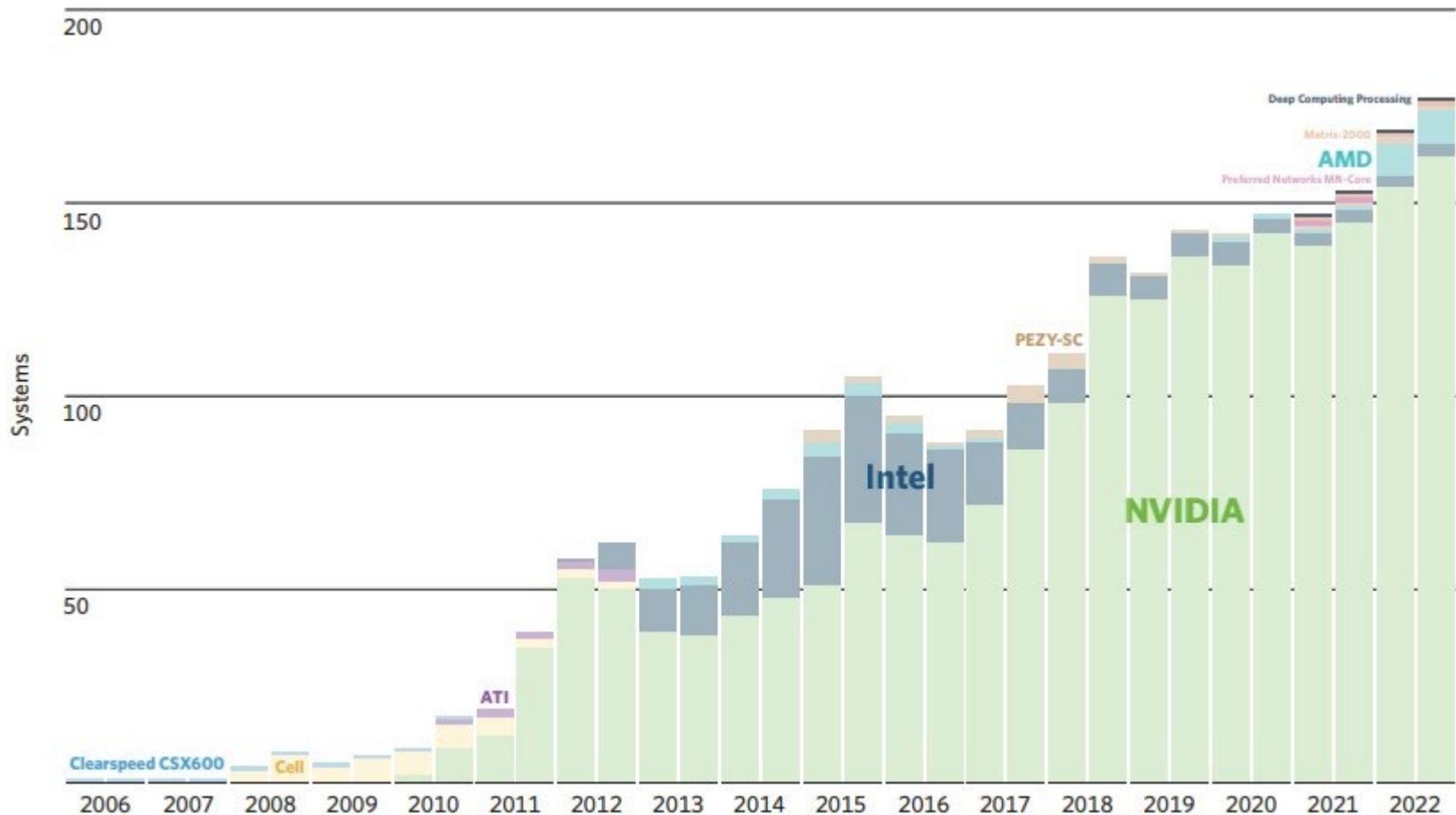
Intel的发展路线

- Xeon Phi (2010-2020)

Larrabee GPU最后转变成了Xeon Phi协处理器。 Xeon Phi没有图形功能，但保留了Larrabee的一些特点，例如：

- x86架构，可以使用OpenMP编程
- 512-bit SIMD向量计算单元
- Xeon Phi 31S1P是天河2号的主要算力组成部分
- 由于需求不足、Intel 10nm工艺节点困难等原因，Xeon Phi系列协处理器于2020年终止。

Top500中来自不同厂商的异构加速器数量



NVIDIA GPU

- GPU围绕**流式多处理器**(Stream Multiprocessor, SM)的**可扩展阵列**搭建。
- 通过**复制**这种结构的构建块来实现GPU的硬件并行。
- GPU中的每一个SM都能支持数百个线程并发执行，每个GPU通常有多个SM，所以在一个GPU上并发执行数千个线程是有可能的。

可以将一个SM类比为
多核CPU中的一个核

NVIDIA Fermi GPU

- 2010年NVIDIA发布Fermi系列GPU
- 被称之为“The First Complete GPU Computing Architecture”
- 后续的结构基本类似，因此本课程以Fermi为例来介绍GPU的硬件结构，主要从以下三个方面：
 - 执行单元结构 —— 执行硬件的物理结构
 - 存储器层次结构 —— 存储硬件的物理结构
 - 线程组织结构 —— 组织执行和存储硬件的逻辑结构

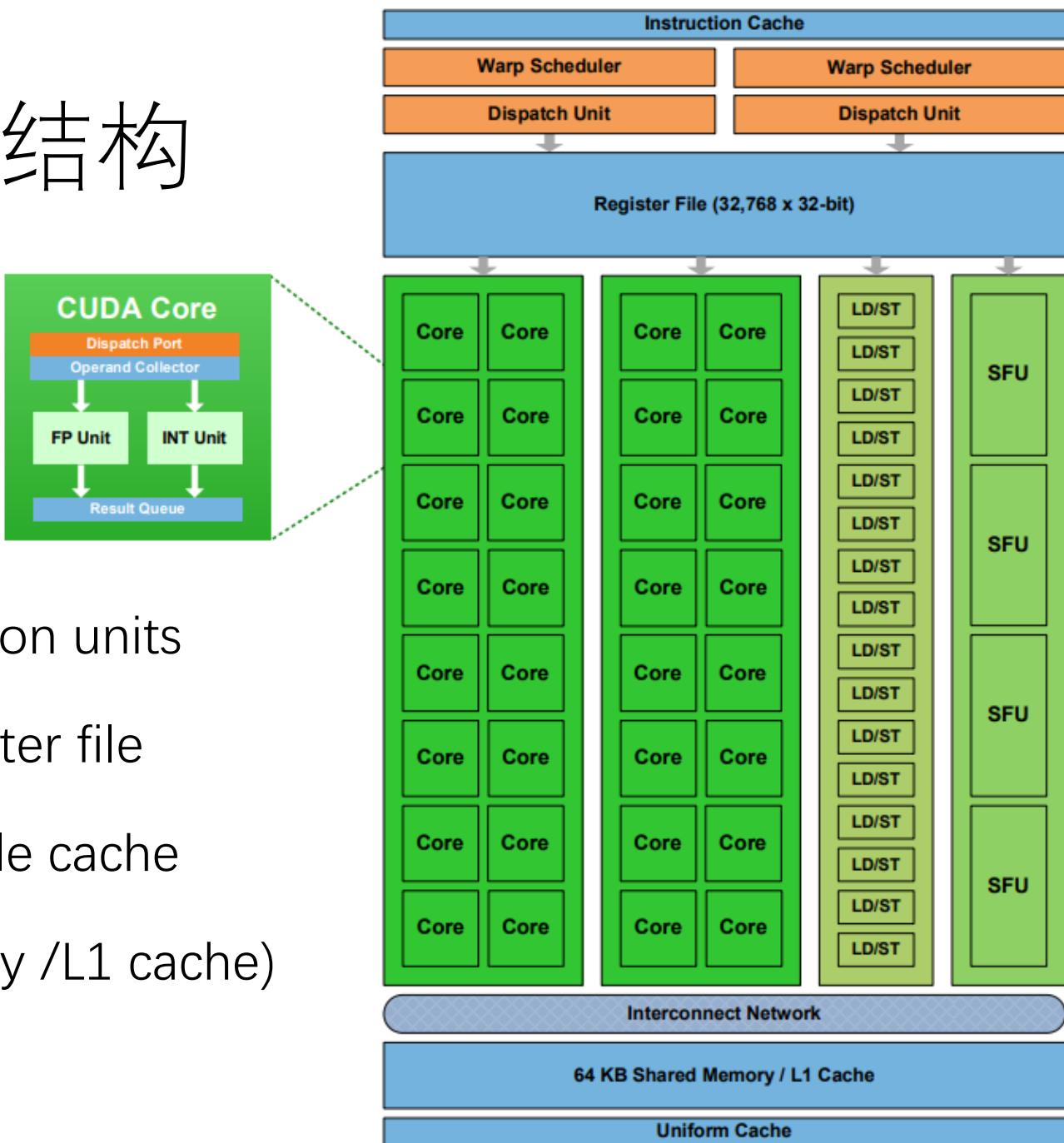
(也叫编程模型)

执行单元结构

每个SM包括：

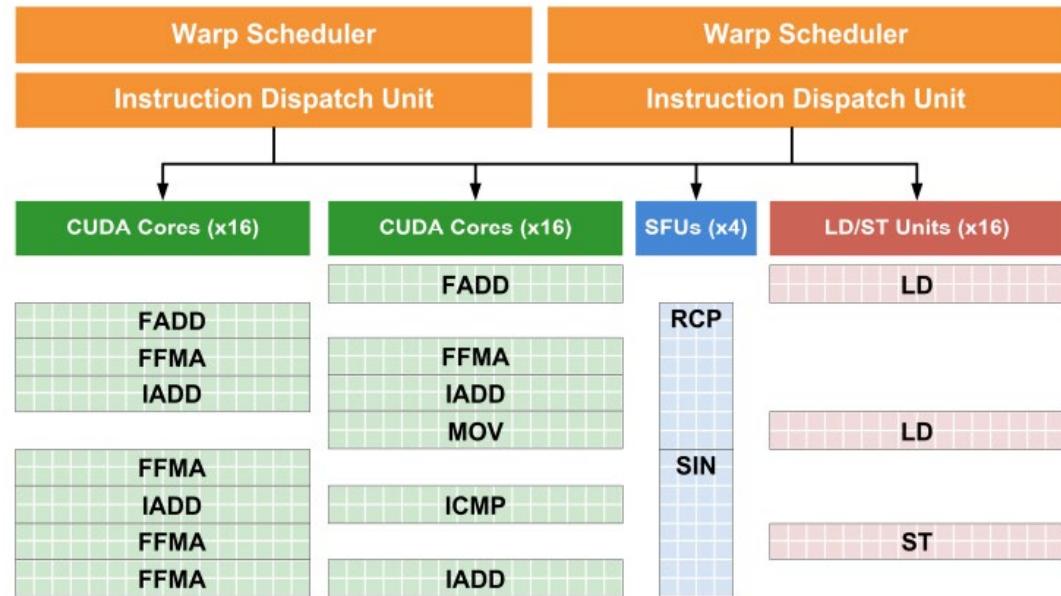
- 32 Cores
- 16 LD/ST units
- 4 special-function units
- 32K-word register file
- 64K configurable cache

(shared memory /L1 cache)



执行单元结构

- 2个调度器
- 4组执行单元区块

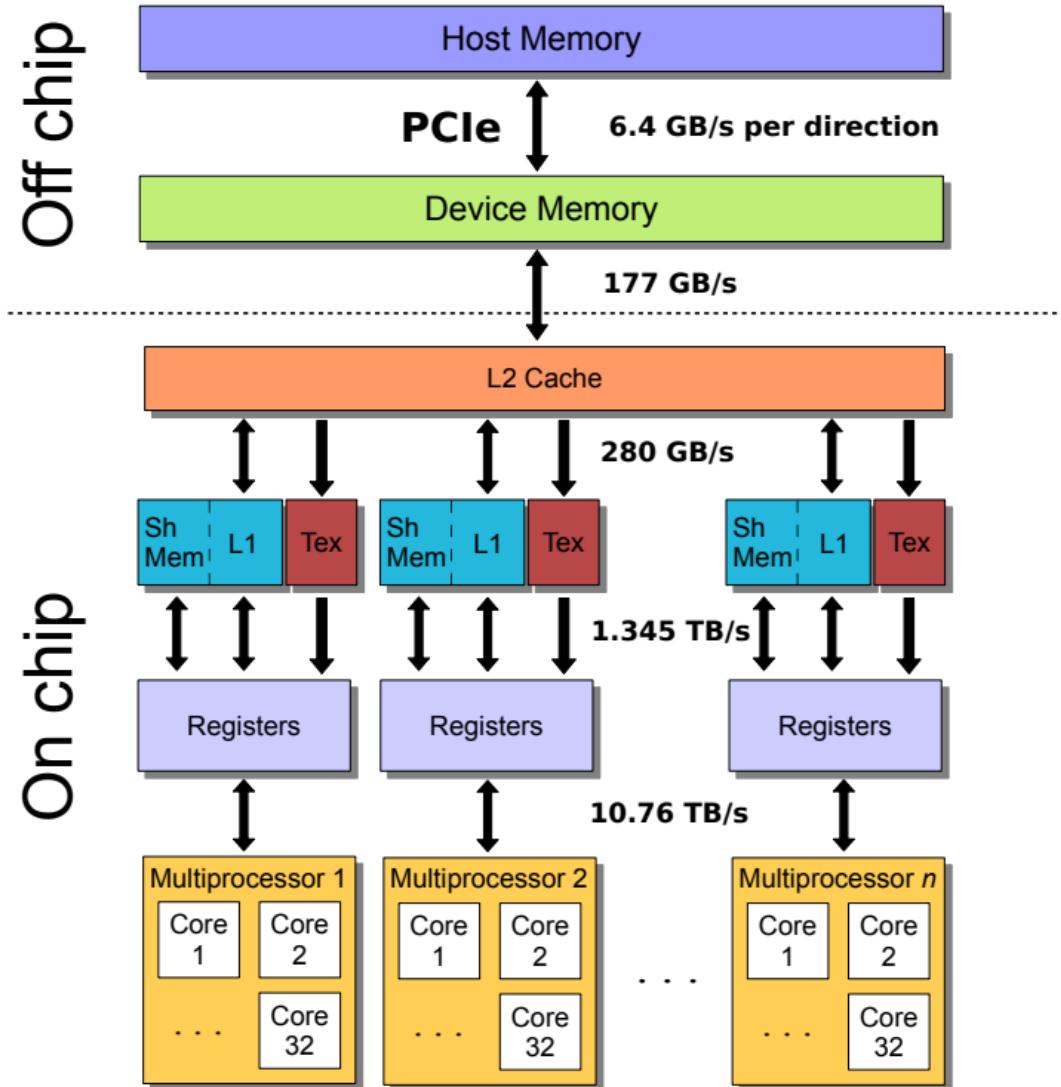


- FMA表示Fused Multiply-Add指令
- SFU用于复杂数学函数计算，例如三角函数
- 1个周期可以在1或2组执行单元区块上发送最多32条指令

存储层次结构

- Register
- L1 Cache
- Shared Memory
- L2 Cache
- Device Memory
 - Global Memory
 - Constant Memory
 - Texture Memory
- Host Memory

带宽数据基于GTX480
Accelerating Radio Astronomy Cross-Correlation with Graphics Processing Units



存储层次结构

按照访存性能从快到慢可以分成以下几个层级：

1. Register 相比与CPU来说数量非常多，线程私有
2. L1 + Shared + Cached Constant & Texture 线程块私有
3. L2 所有线程共享
4. Device Memory = Global + Constant +Texture
5. Host Memory

存储层次结构

“只读”的意思是在kernel执行中不可修改，kernel执行前可以修改。

存储器	位置	是否缓存	访问延迟	说明
寄存器 Register	On-chip	否	极低	过多分配会导致溢出至Local Memory (性能等于Global Memory)
共享存储器 Shared Memory	On-chip	否	同寄存器	可编程的L1 Cache
全局存储器 Global Memory	Off-chip	是	400-600时钟周期	
常量存储器 Constant Memory	Off-chip	是	缓内：同寄存器 缓外：400-600时钟周期	只读。可以在数据广播给多个线程时提升性能
纹理存储器 Texture Memory	Off-chip	是	缓内：同寄存器 缓外：400-600时钟周期	只读。可以在二维访存中提升性能

存储体冲突 (Bank Conflict)

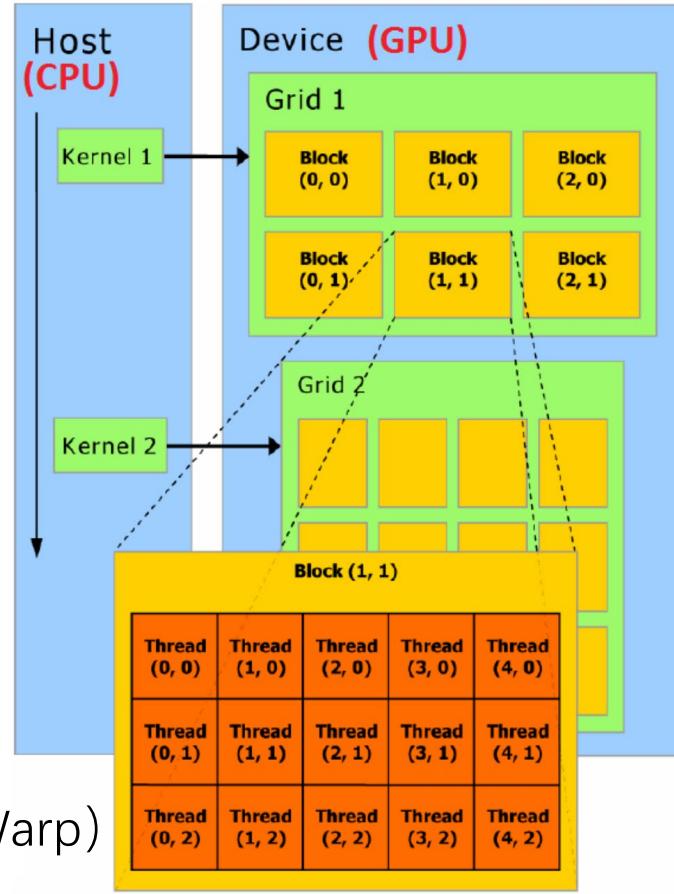
- 访问共享存储器的速度很快
如果不存在**存储体冲突**, 速度与寄存器一样
- Fermi的共享存储器分为**循环分布**的32个存储体 (Bank)
$$\text{Bank Number} = (\text{Address}/4) \bmod 32$$
- 不同存储体可以并发访问
- 存储体冲突**:

4 bytes为一个单位

如果多个线程访问同一个存储体, 则访问序列化 (串行执行)

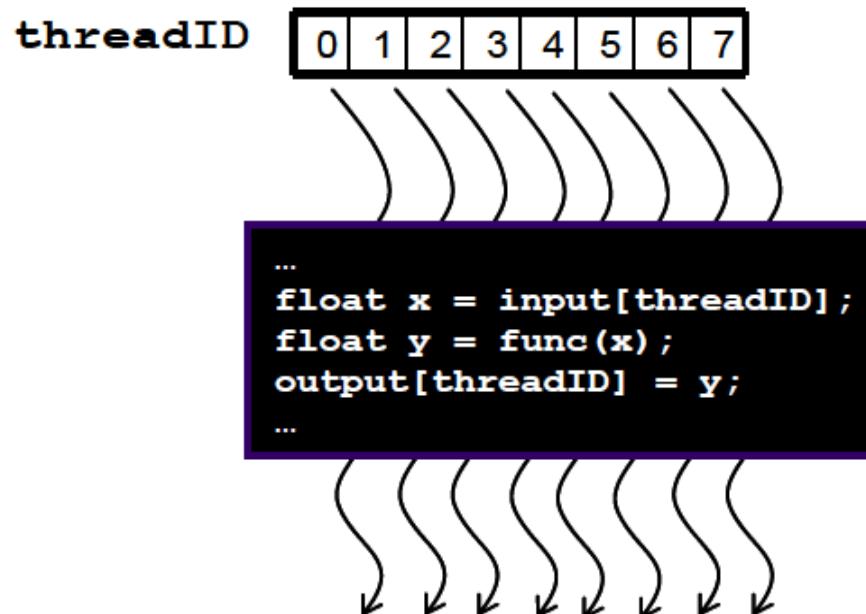
线程组织结构

- 线程 (Thread)
 - CUDA程序中的基本执行单元
 - 硬件支持，创建、结束、调度开销很低
- 线程块 (Thread Block)
 - 多个线程可以组成一个线程块
 - 块内每32个线程为一个线程束 (Thread Warp)
 - 一个线程束内的所有线程执行相同的代码
(Single Instruction Multiple Thread, SIMD)
- 线程网格 (Thread Grid)
 - 多个线程块可以组成一个线程网格



线程组织结构

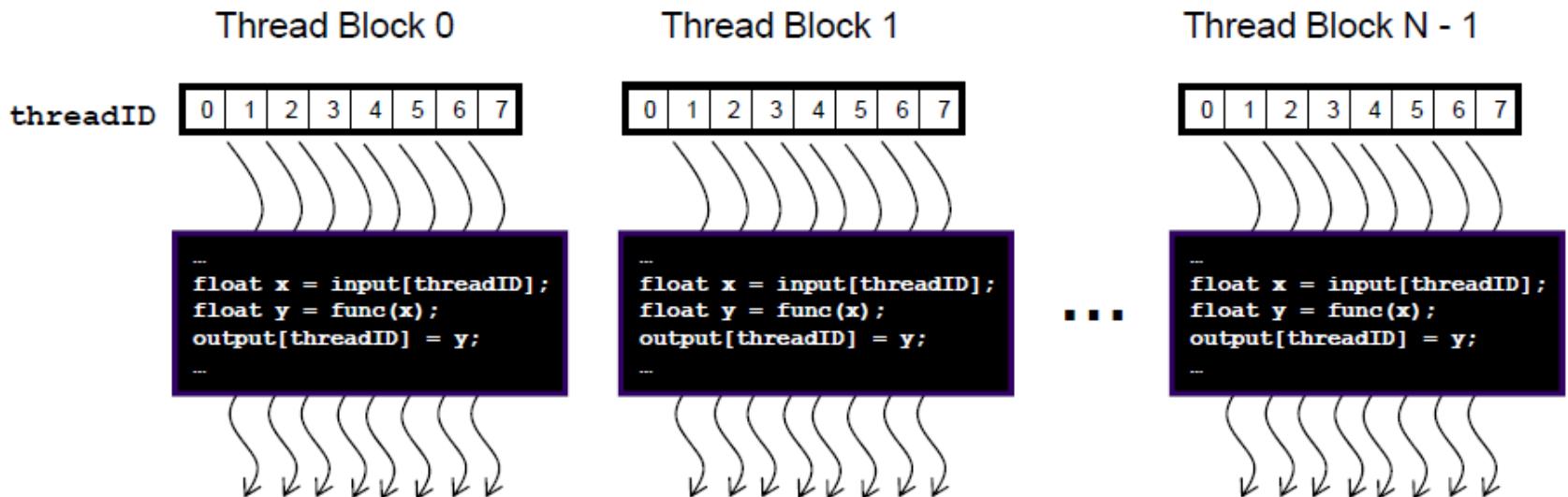
- Thread id:
 - a) local id: thread id in a block
 - b) global id: thread id in a grid
- Compute thread global id : **blockDim*blockIdx+threadIdx**
- Each thread uses id to decide which data to work on



线程组织结构

能否通过global
memory协作?

- Threads within a block can cooperate via **shared memory**, **atomic operations** and **barrier synchronization**
- Threads in different blocks **cannot** cooperate



GPU并行编程

I. GPU简介

II. GPU结构

III. CUDA编程

IV. 性能与优化

V. 案例：矩阵乘法

CUDA软件栈

USE-CASES



Speech



Translate



Recommender

CONSUMER INTERNET



Healthcare



Manufacturing



Finance

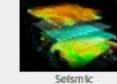
INDUSTRIAL APPLICATIONS



Molecular Simulations



Weather Forecasting



Seismic Mapping

SUPERCOMPUTING

APPS & FRAMEWORKS



Amber

NAMD

+600 Applications

CATIA



Ps



CUDA-X LIBRARIES

MACHINE LEARNING



cuML

cuGRAPH

DL / HPC

cuDNN

CUTLASS

TENSORRT

CUDA Math Libraries

LANGUAGES



OpenACC



LLVM Compiler For CUDA

CUDA

CUDA TOOLKIT

CUDA COMPILER

DEVELOPER TOOLS

DEBUGGERS

PROFILERS

CUDA C++ CORE

CUDA DRIVER

MEMORY MANAGEMENT

WINDOWS & GRAPHICS

COMMS LIBRARIES

OS PLATFORMS



CentOS



Windows Server

CUDA软件栈

- CUDA Library: 提供了大量高性能库函数，例如cuBLAS、cuSPARSE、cuFFT、cuDNN等等
- CUDA Toolkit:
 - 提供开发环境，包括编译器、调试器、性能分析工具等
 - 提供运行时环境，包括内存管理、设备访问和执行调度等
- CUDA Driver: 提供了抽象的设备访问接口，使得同一个CUDA程序可以正确的运行在所有支持CUDA的GPU上

CUDA编程

- CUDA C在C/C++的基础上扩展而来
- 最好的学习资料是官方文档

<https://docs.nvidia.com/cuda/cuda-c-programming-guide/#c-language-extensions>

(其他网络资料零碎不全，而书籍通常有滞后性)

CUDA编程——限定符

- 函数执行空间限定符 (Function Execution Space Specifiers)
 - 函数执行空间限定符指定函数执行位置（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，会立即返回

函数限定符	在何处执行	从何处调用	特性
<code>_device_</code>	设备	设备	函数的地址无法获取
<code>_global_</code>	设备	主机	返回类型必须为空
<code>_host_</code>	主机	主机	等同于不使用任何限定符

CUDA编程——限定符

- 变量存储空间限定符 (Variable Memory Space Specifiers)
 - 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 (Local Memory, 物理上等同于Global Memory)
 - `_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。

限定符	位于何处	可以访问的线程	主机访问
<code>_device_</code>	全局存储器	线程网格内的所有线程	通过运行时库访问
<code>_constant_</code>	常量存储器	线程网格内的所有线程	通过运行时库访问
<code>_shared_</code>	共享存储器	线程块内的所有线程	不可从主机访问

CUDA编程——内置向量类型

- 内置的向量类型都是结构体
 - 结构体成员x, y, z, w分别表示第1、2、3、4个分量
- 用基本数据类型+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

CUDA编程——内置变量

- 内置变量用于获得线程网格和线程块的大小以及线程块和线程的编号。**只能在设备上执行的函数中使用。**

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

CUDA编程——核函数

- 核函数（Kernel Function）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GPU中的main函数
- 核函数的参数通过**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）
- 核函数使用`_global_`限定符声明，返回值为空

CUDA编程——核函数

- 调用核函数需要使用KernelName<<<>>>()的方式
- <<<>>>内的参数用于指定执行核函数的配置，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)， 网格的两个维度，第三维被忽略
 - DimBlock (dim3类型)， 线程块的三个维度
 - MemSize (size_t类型)， 此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，则核函数不会被执行，直接返回错误信息

CUDA编程——核函数

- 核函数定义和调用示例

```
// 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);
    ...
}
```

CUDA编程——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

cudaSetDevice(): 选择本次计算使用的设备， 默认使用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上的常量存储器

CUDA编程——运行时API

内存拷贝cudaMemcpy()

- 主机内存和设备内存是两个存储空间，必须指定数据位置。
- 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)

- 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，带宽较低，需要尽可能避免

CUDA编程——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，就执行后面的c=a+b，而GPU在执行kernel函数，**此时CPU和GPU完全并行工作**。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;  
.....
```

CUDA编程——同步

- 同一个block内的线程可以快速同步

```
__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
```

CUDA编程——同步

- 多个线程块之间同步以及多个设备之间的同步比较困难，CUDA运行时库没有直接提供此类函数
- CUDA 9引入的Cooperative Groups可以实现多个线程块之间同步以及多个设备之间的同步
- 具体实现可参考<https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#cooperative-groups>

CUDA编程——计时函数

- CUDA提供高精度计时函数
- 使用示例：

```
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);
```

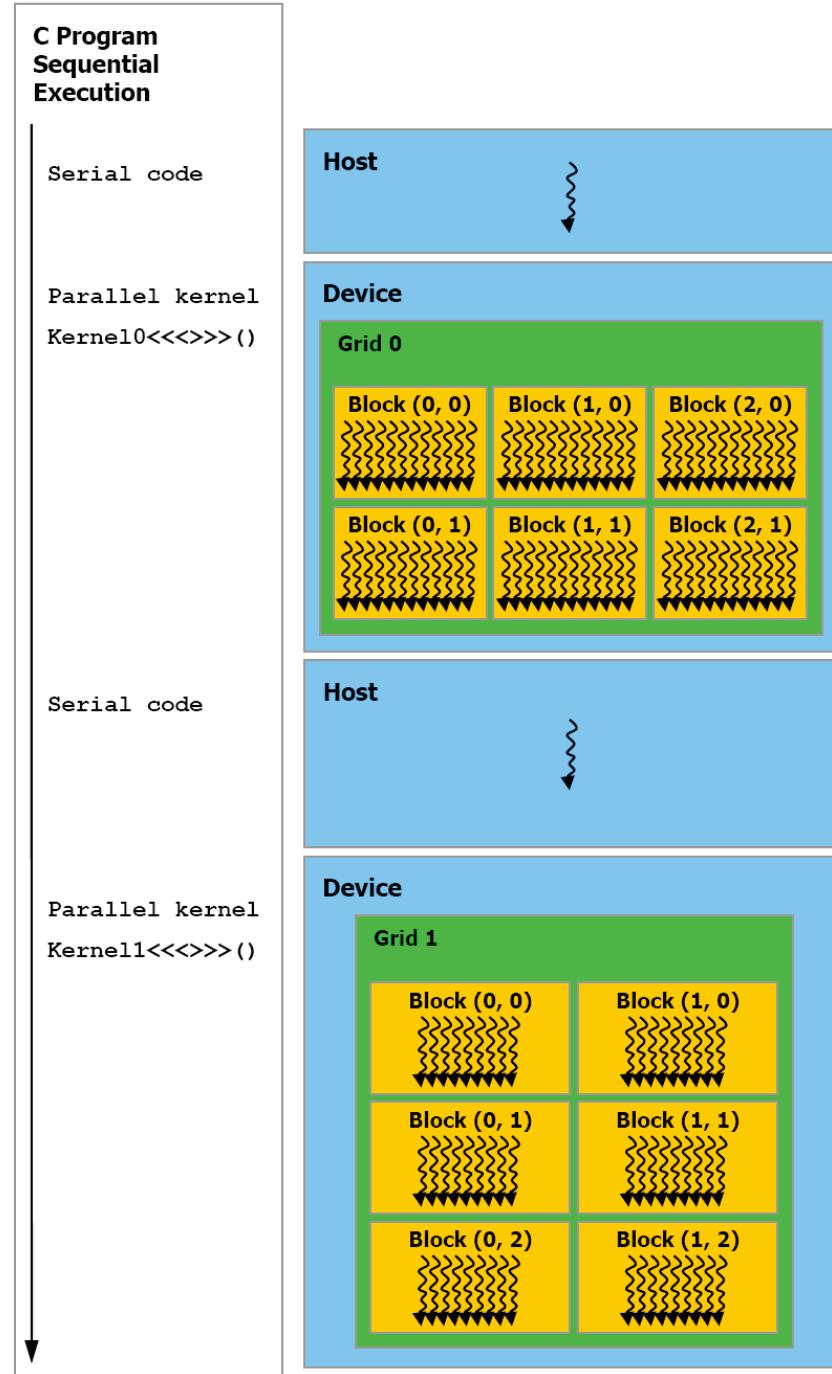
CUDA编程——计时函数

- 另一种使用cudaEvent计时的方法：

```
cudaEvent_t start, stop; // Create events for timing  
cudaEventCreate(&start);  
cudaEventCreate(&stop);  
cudaEventRecord(start, 0); // Start record  
CudaKernel<<<dimGrid, dimBlock, memsize>>>(); //GPU计算  
cudaEventRecord(stop, 0); // Stop record  
cudaEventSynchronize(stop);  
float milliseconds = 0; // Calculate elapsed time  
cudaEventElapsedTime(&milliseconds, start, stop);  
printf("CudaKernel took %f ms.\n", milliseconds);
```

CUDA程序结构

- 主机端代码类似Fork-Join风格
- 串行代码由CPU执行
- 并行代码由GPU执行
- 核函数相当于开启一个并行域
 - 类似OpenMP
 - 注意核函数是异步调用的



CUDA程序结构

- CUDA程序的一般执行步骤
 1. CPU代码执行
 2. 传输数据到GPU
 3. GPU代码执行
 4. 传输数据回CPU
 5. CPU代码执行
 6. 结束
- 如果有多个核函数，需要重复2 ~ 4步

GPU并行编程

I. GPU简介

II. GPU结构

III. CUDA编程

IV. 性能与优化

V. 案例：矩阵乘法

性能测量与分析

- Nsight Systems <https://developer.nvidia.com/nsight-systems>
GPU and CPU sampling and tracing
- Nsight Compute <https://developer.nvidia.com/nsight-compute>
GPU kernel profiling
- CUPTI <https://developer.nvidia.com/cupti>
CUDA Profiling Tools Interface

CUDA性能优化

- CUDA性能优化的一般原则

1. 最大化硬件利用率；
2. 最大化访存吞吐率；
3. 最大化指令吞吐率；

参考文档

<https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#performance-guidelines>
<https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/#optimizing-cuda-applications>

优化硬件利用率——任务并发

- 在CUDA中，以下任意两个任务可以并发执行：
 1. Computation on the host;
 2. Computation on the device;
 3. Memory transfers from the host to the device;
 4. Memory transfers from the device to the host;
 5. Memory transfers within the memory of a given device;
 6. Memory transfers among devices.
- 核函数是异步调用的，可以实现并发执行

```
cudaMemcpyAsync(a_d, a_h, size, cudaMemcpyHostToDevice, 0);  
kernel<<<grid, block>>>(a_d);  
cpuFunction();
```

优化硬件利用率——延迟隐藏

- At every instruction issue time, a warp scheduler selects a warp that has threads ready to execute its next instruction and issues the instruction to those threads.
- The number of clock cycles it takes for a warp to be ready to execute its next instruction is called the **latency**, and **full utilization is achieved when all warp schedulers always have some instruction to issue for some warp at every clock cycle during that latency period**, or in other words, when latency is completely “hidden”.

优化硬件利用率——延迟隐藏

- 具体的机制以及性能建模过程比较复杂
- 简单点理解就是：
 - 问题：一个线程束为了准备下一条指令的执行，需要一定准备时间
 - 解决方向：
 - 设置足够多的线程数量，使得一些线程束在准备的时候，另一些线程束可以执行指令（被称为活跃线程束），始终保持GPU高利用率。
 - 让准备时间尽可能短，这需要让指令的操作数尽可能在寄存器/L1 cache中，减少取数耗时；尽可能减少memory fence和同步的使用。

这两个优化方向是非正交的。将两个方向分别优化到极致，并不一定得到最好性能。

memory fence类似OpenMP的flush，保证所有线程的存储器一致性。

优化硬件利用率——占用率计算器

- 如何评估是否设置了足够多的线程数量？
- Occupancy Calculator
- 有三种形式：
 - API, 由cuda_occupancy.h提供 <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#occupancy-calculator>
 - Nsight Compute性能分析工具提供 <https://docs.nvidia.com/nsight-compute/NsightCompute/index.html#occupancy-calculator>
 - Excel表格 <https://docs.nvidia.com/cuda/archive/12.2.1/cuda-occupancy-calculator/index.html> [Deprecated]

优化硬件利用率——占用率计算器

AutoSave (Off)

CUDA_Occupancy_Calculator.xls - Compatibility Mode

File Home Insert Draw Page Layout Formulas Data Review View Help Team Search

MyThread... 320

CUDA Occupancy Calculator

Just follow steps 1, 2, and 3 below! (or click here for help)

1.) Select Compute Capability (click): 7.0 [\(Help\)](#)

2.) Select Shared Memory Size Config (bytes): 32768 [\(Help\)](#)

2.) Enter your resource usage:

Threads Per Block	320
Registers Per Thread	37
Shared Memory Per Block (bytes)	0

(Don't edit anything below this line)

3.) GPU Occupancy Data is displayed here and in the graphs:

Active Threads per Multiprocessor	1280
Active Warps per Multiprocessor	40
Active Thread Blocks per Multiprocessor	4
Occupancy of each Multiprocessor	63%

Physical Limits for GPU Compute Capability: 7.0

Threads per Warp	32
Max Warps per Multiprocessor	64
Max Thread Blocks per Multiprocessor	32
Max Threads per Multiprocessor	2048
Maximum Thread Block Size	1024
Registers per Multiprocessor	65536
Max Registers per Thread Block	65536
Max Registers per Thread	255
Shared Memory per Multiprocessor (bytes)	32768
Max Shared Memory per Block	32768
Register allocation unit size	256
Register allocation granularity	warp
Shared Memory allocation unit size	256
Warp allocation granularity	4

Allocated Resources

	Per Block	Limit Per SM	= Allocatable Blocks Per SM
Warp (Threads Per Block / Threads Per Warp)	10	64	6
Registers (Warp limit per SM due to per-warp reg count)	10	48	4
Shared Memory (Bytes)	0	32768	32

Note: SM is an abbreviation for (Streaming) Multiprocessor

[Click Here for detailed instructions on how to use this occupancy calculator.](#)
[For more information on NVIDIA CUDA, visit http://developer.nvidia.com/cuda](http://developer.nvidia.com/cuda)

Your chosen resource usage is indicated by the red triangle on the graphs. The other data points represent the range of possible block sizes, register counts, and shared memory allocation.

Impact of Varying Block Size

Impact of Varying Shared Memory Usage Per Block

Impact of Varying Register Count Per Thread

Calculator Help GPU Data Copyright & License Display Settings 85%

优化访存吞吐率——全局存储器

- 全局存储器延迟：400~600 clock cycles
- 容易成为性能瓶颈

优化措施：

- 合并访存
- 向量数据类型
- 延迟隐藏

优化访存吞吐率——全局存储器

内存事务 (Memory Transaction)

- 在 CUDA 中，内存事务是指数据从一个地址移动到另一个地址的过程。一次内存事务移动的数据量是固定的，可以是32, 64, 128 bytes (不同硬件上不一样)。
- 如果要优化访存吞吐率，需要同时访问足够多的连续数据。
- CUDA提供了合并访存机制，无需在代码中显式地构造向量数据的读写，就能自动实现访存优化。

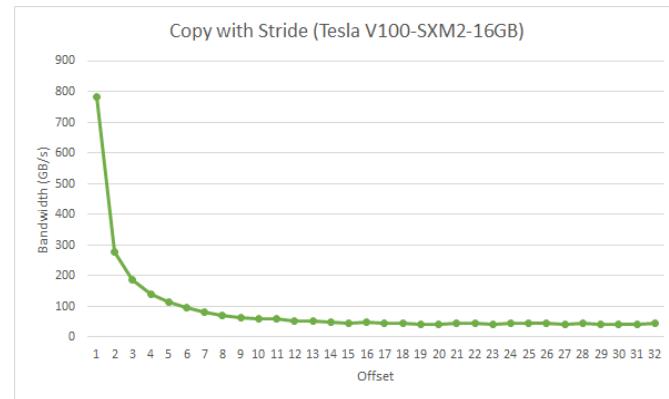
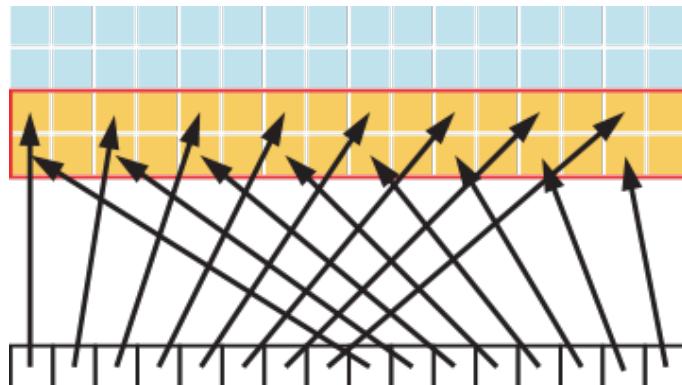
优化访存吞吐率——全局存储器

合并访存 (Coalesced Memory Access)

- 当一个线程块内的多个线程同时访问内存时，如果访问的是**相邻**的内存地址，会被**合并**成一个或多个内存事务。

```
__global__ void strideCopy(float *odata, float* idata, int stride)
{
    int xid = (blockIdx.x*blockDim.x + threadIdx.x)*stride;
    odata[xid] = idata[xid];
}
```

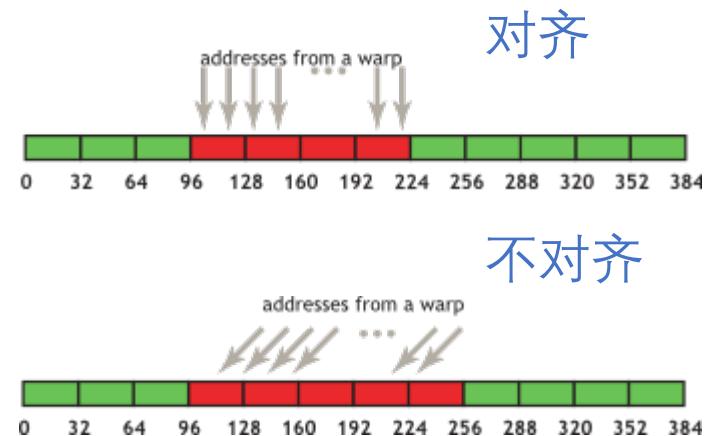
当 $stride > 1$ 时，
不连续的访存
导致带宽浪费



优化访存吞吐率——全局存储器

合并访存 —— 存储对齐 (Memory Alignment)

- 访存的首地址需要是内存事务大小的整倍数
- 假设需要访存一个 32×4 bytes数组
 - 首地址96：对齐，4次访存
 - 首地址112：不对齐，5次访存
- 可以使用`_align_(n)`控制对齐
- 通过CUDA运行时API分配的内存，例如`cudaMalloc()`，保证对齐到至少256字节

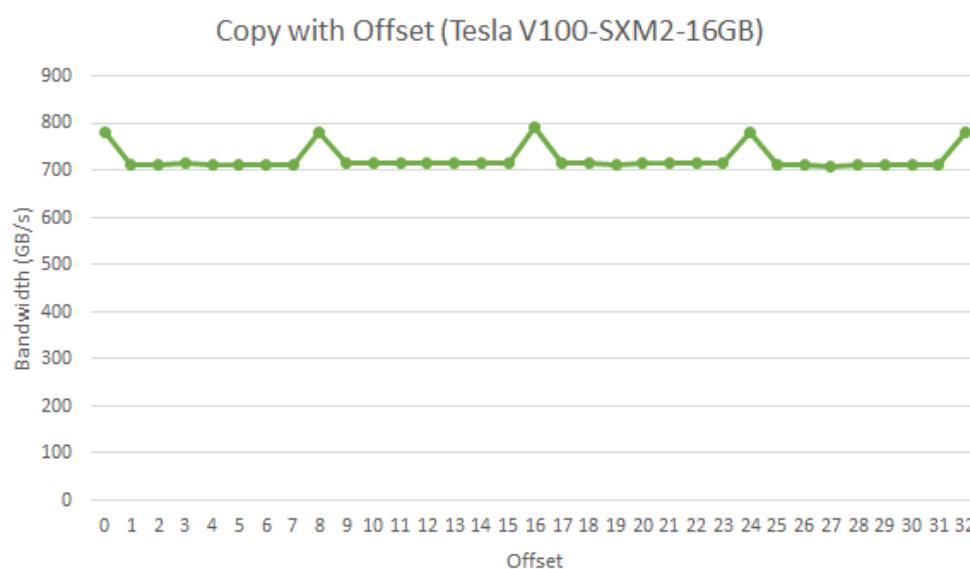


优化访存吞吐率——全局存储器

合并访存 —— 存储对齐

- 观察不同偏移量下数据传输的带宽

```
__global__ void offsetCopy(float *odata, float* idata, int offset)
{
    int xid = blockIdx.x * blockDim.x + threadIdx.x + offset;
    odata[xid] = idata[xid];
}
```

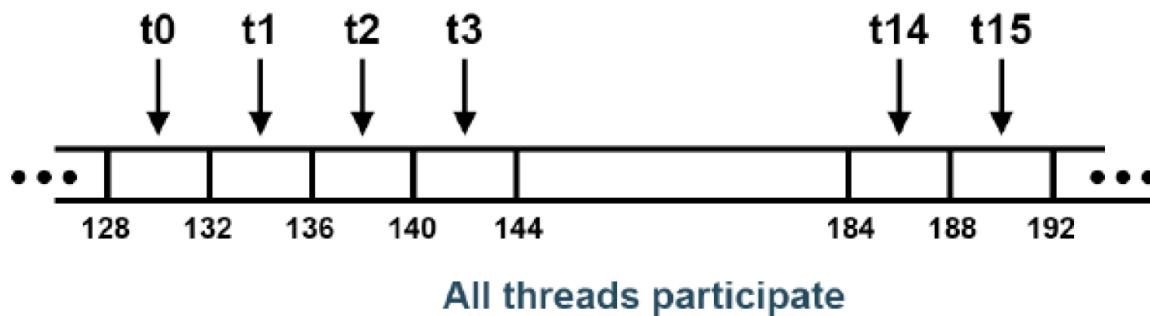


当offset为8的倍数(32 bytes),
带宽更高。

优化访存吞吐率——全局存储器

合并访存的正面案例

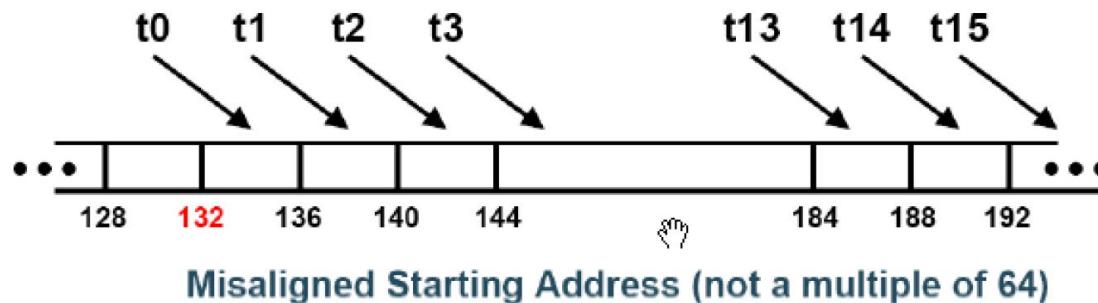
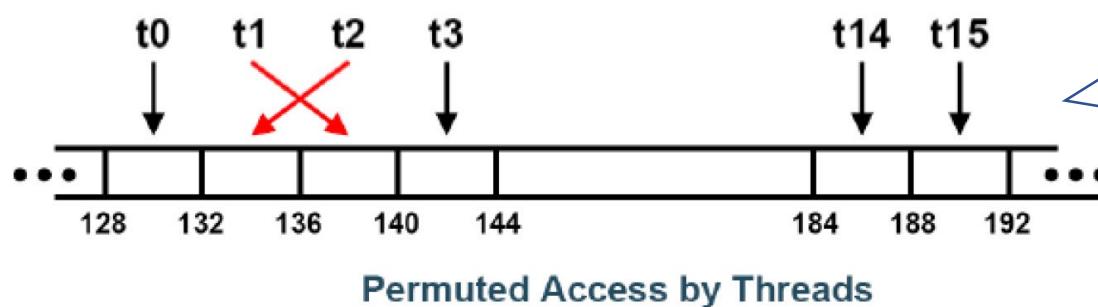
Coalesced Access: Reading floats



优化访存吞吐率——全局存储器

合并访存的负面案例

Uncoalesced Access: Reading floats



优化访存吞吐率——全局存储器

- 显式地使用向量数据类型，可以进一步优化访存
- 向量数据类型
 - 一个线程可以一次性加载或存储多个连续的数据元素
 - 可以减少存储指令数和索引计算

常用向量类型：

char2, char4, short2, short4, int2, int4, longlong2uchar2, uchar4, ushort2,
ushort4, uint2, uint4, ulonglong2float, float2, float4, double2, half2

优化访存吞吐率——全局存储器

延迟隐藏

- CUDA 设备通过执行大量的线程来隐藏单个线程的延迟
- 对全局存储器的访存延迟也可以通过此方法来隐藏
- 使用足够多的线程/线程块

优化访存吞吐率——本地存储器

- 本地存储器（Local Memory）延迟：和全局存储器一样
- 本地存储器无法编程控制，由编译器自动分配使用
- 哪些情况会导致本地存储器的使用：
 - 无法确定其是否以常量索引的数组
 - 消耗太多寄存器空间的大型结构或数组
 - 寄存器分配溢出

“本地存储器”
这个名字容易
引起误解。

优化措施：

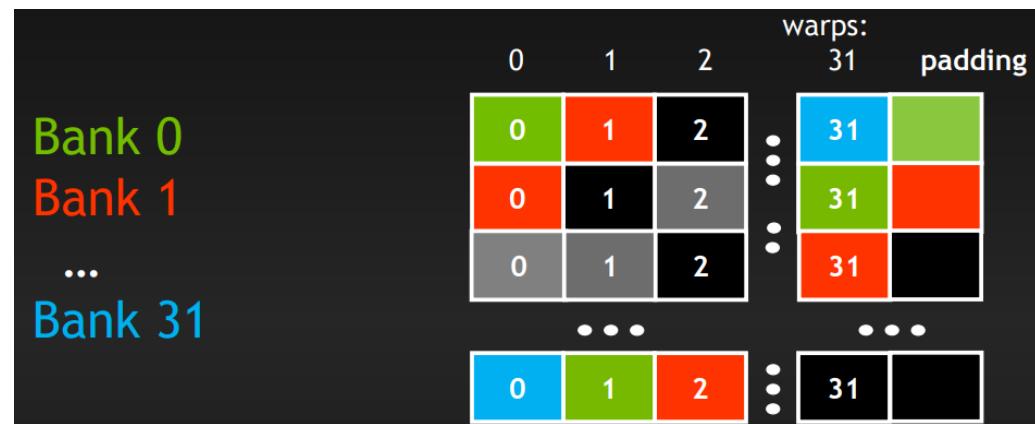
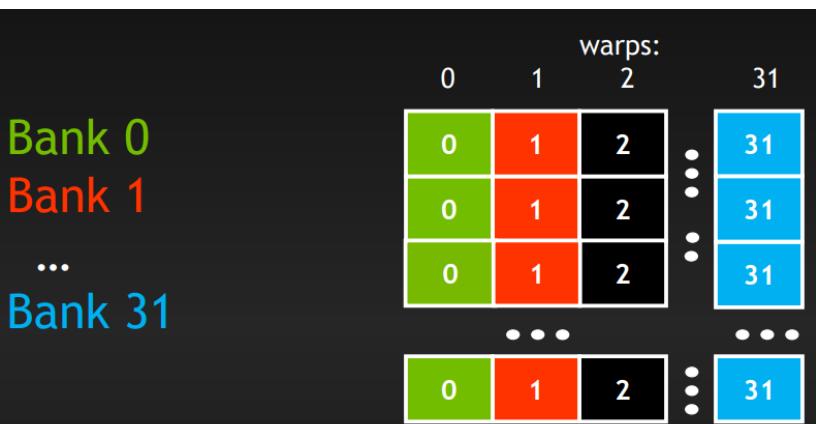
- 减少上述三种情况的发生

优化访存吞吐率——共享存储器

- 如果数据可以复用，应当尽量放在共享存储器
- 需要避免Bank Conflict。Padding是一种常用解决方法

举例：32个warp访问共享存储器中的 32×32 数组

- 每个warp需要读取一列数据，则产生32路冲突（左图）
- 增加一列空数据，将数组变为 32×33 ，此时无冲突（右图）



优化指令吞吐率——算术指令

- 尽可能使用高吞吐率指令
 - 使用单精度、半精度代替双精度浮点数
 - 使用intrinsic function代替常规数学函数
<https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#intrinsic-functions>
 - 使用位运算代替一些整数乘法/除法

优化指令吞吐率——算术指令

典型的运算周期数

吞吐率数据参考<https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#arithmetic-instructions>

- 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
- Double precision will perform at half the speed

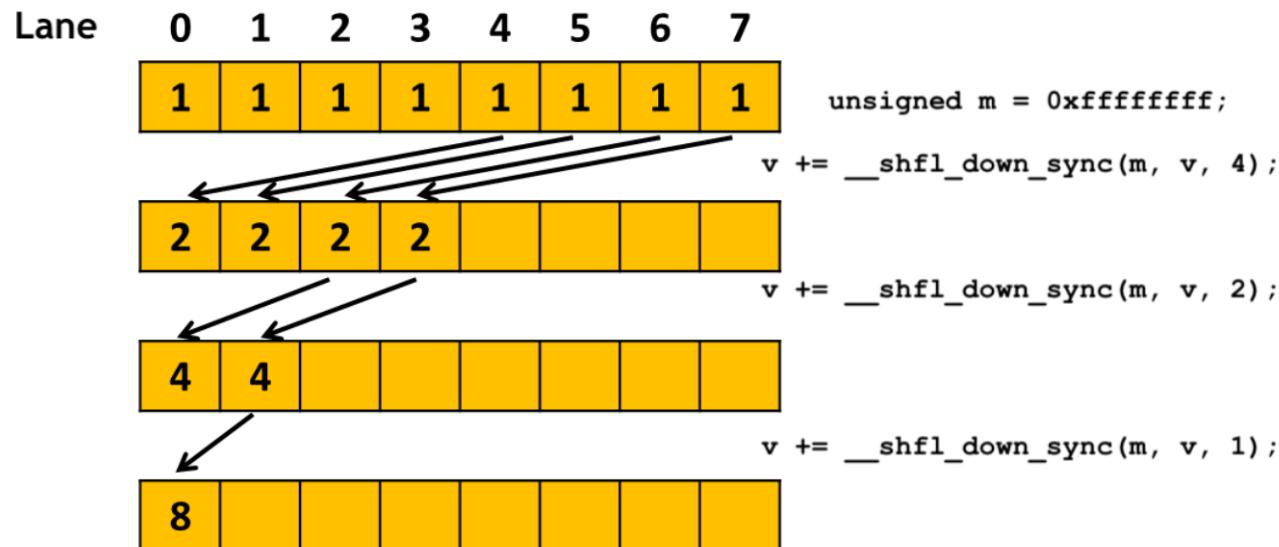
优化指令吞吐率——控制指令

- 尽可能避免在一个线程束内产生分支
 - SIMT会导致分支序列化
 - 循环展开可以减少循环导致的分支
 - Volta架构之前，线程束内分支代码是严格序列化的。
 - Volta架构开始允许线程束内的线程也有一定程度的独立调度，可以隐藏一部分延迟。
 - 总体来说，分支对性能的影响仍然非常大。
-
- The diagram illustrates the execution of a thread block over time. It shows two main sections: 'Pre-Volta' and 'Post-Volta'. In the 'Pre-Volta' section, a code snippet is shown:
- ```
if (tid<4){
 A;
 B;
} else {
 X;
 Y;
}
Z;
```
- The execution timeline starts with a 'diverge' point. From this point, four threads branch into 'A;' and 'B;'. Both 'A;' and 'B;' paths lead to a 'Stall' period. After the stall, the threads diverge again into 'X;' and 'Y;'. Both 'X;' and 'Y;' paths lead to another 'Stall' period. Finally, all threads converge into a 'Z;' path. In the 'Post-Volta' section, the execution is shown as a single, continuous flow. The 'diverge' point is followed by a 'Stall' period. After the stall, the threads diverge into 'X;', 'Y;', and 'Z;'. All three paths then converge into a 'sync warp' (highlighted in yellow) and finally into a 'Z;' path. This visualizes how Volta's warp-level parallelism allows for more efficient execution of diverging branches.

# 优化指令吞吐率——同步指令

- 尽可能减少使用同步
- 借助一些Warp-Level Primitives, 可能可以减少同步开销

<https://developer.nvidia.com/blog/using-cuda-warp-level-primitives/>



Part of a warp-level parallel reduction using shfl\_down\_sync().

# GPU并行编程

I. GPU简介

II. GPU结构

III. CUDA编程

IV. 性能与优化

V. 案例：矩阵乘法

# 矩阵乘法

- 矩阵乘法是很好的GPU性能优化学习案例
- 串行矩阵乘法：

```
1 // Function to multiply two matrices A and B, storing the result in C
2 void matrixMultiply(int N, double** A, double** B, double** C) {
3 // Perform matrix multiplication C = A * B
4 for (int i = 0; i < N; i++) {
5 for (int j = 0; j < N; j++) {
6 for (int k = 0; k < N; k++) {
7 C[i][j] += A[i][k] * B[k][j];
8 }
9 }
10 }
11 }
```

# 矩阵乘法

- OpenMP并行矩阵乘法：

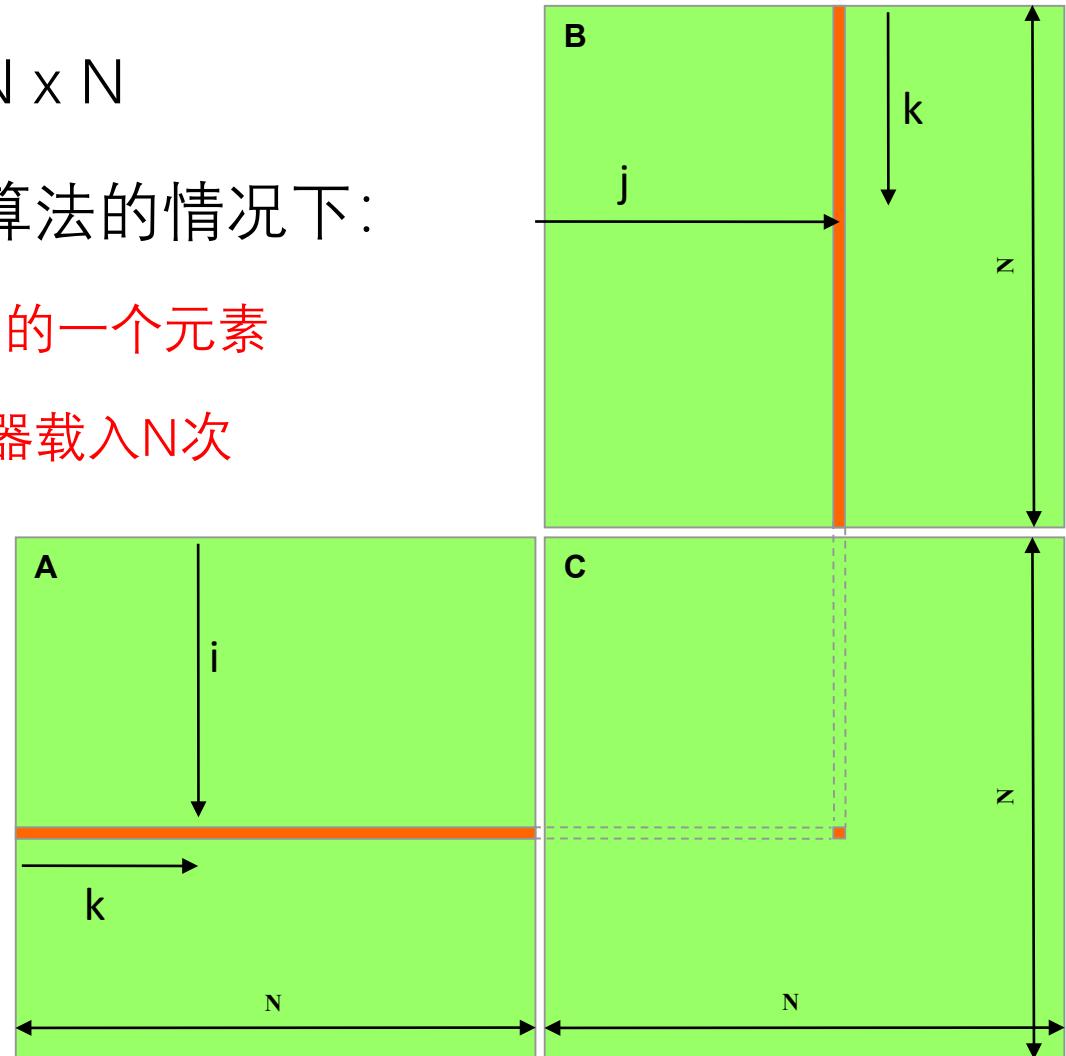
```
// Function to perform matrix multiplication
void matrixMultiply(float* A, float* B, float* C, int N) {
 #pragma omp parallel for collapse(2)
 for (int i = 0; i < N; i++) {
 for (int j = 0; j < N; j++) {
 float sum = 0.0f;
 for (int k = 0; k < N; k++) {
 sum += A[i * N + k] * B[k * N + j];
 }
 C[i * N + j] = sum;
 }
 }
}
```

# CUDA矩阵乘法

- 简单并行
- 分块优化
- 更细致的性能测量、分析与优化

# 简单并行

- 矩阵  $C = A * B$  大小为  $N \times N$
- 在没有采用分块优化算法的情况下：
  - 一个线程计算  $C$  矩阵中的一个元素
  - $A$  和  $B$  需要从全局存储器载入  $N$  次



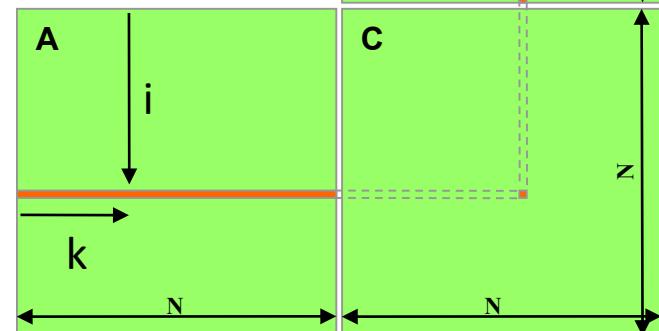
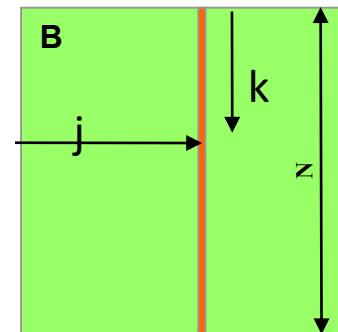
# 简单并行

```
// Kernel definition
__global__ void matrixMul(const float *A, const float *B, float *C, int N) {
 int row = blockIdx.y * blockDim.y + threadIdx.y;
 int col = blockIdx.x * blockDim.x + threadIdx.x;

 if(row < N && col < N) {
 float sum = 0.0f;
 for (int k = 0; k < N; k++) {
 sum += A[row * N + k] * B[k * N + col];
 }
 C[row * N + col] = sum;
 }
}

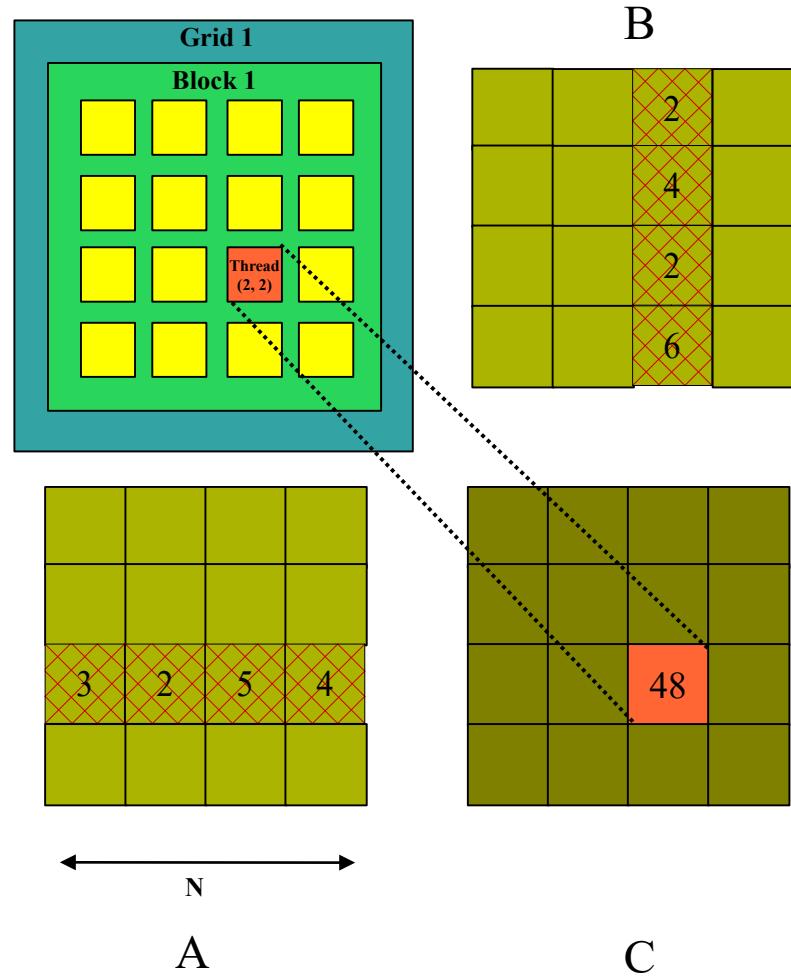
// Setup the execution configuration
dim3 threadsPerBlock(N, N);
dim3 numBlocks(1, 1);

// Execute the matrix multiplication kernel
matrixMul<<<numBlocks, threadsPerBlock>>>(d_A, d_B, d_C, N);
```



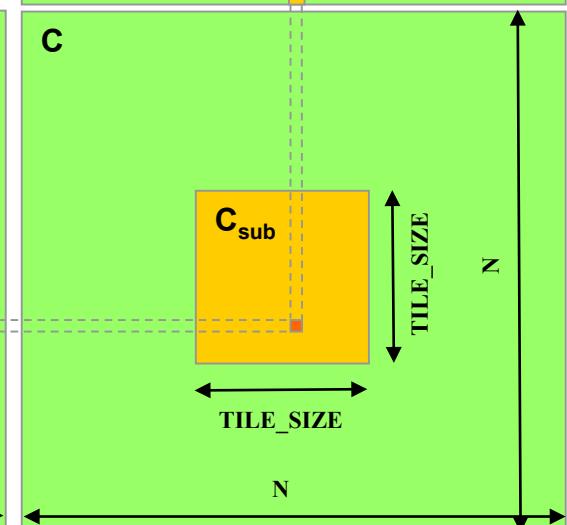
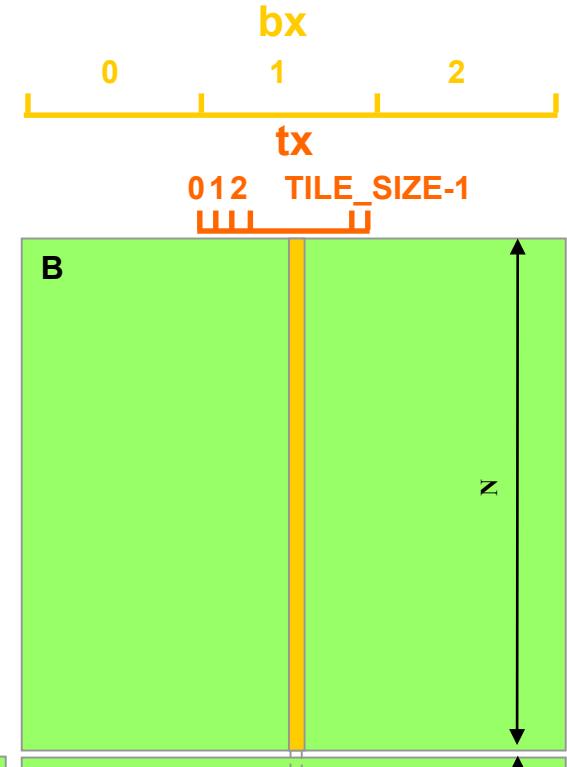
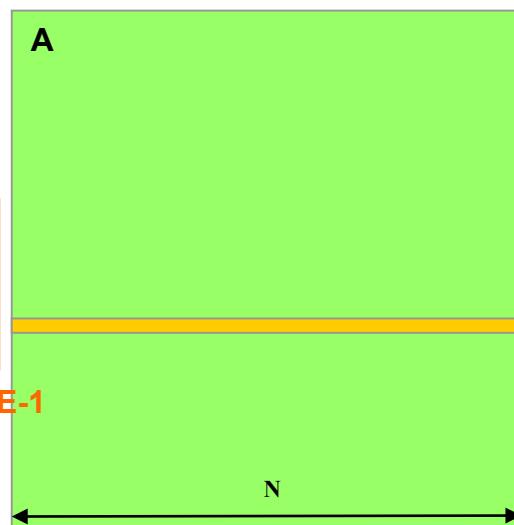
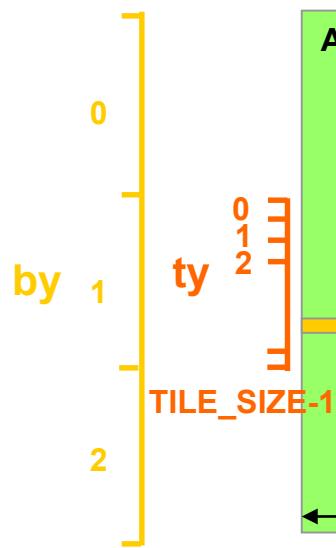
# 简单并行

- 一个线程块中的每个线程计算C中的一个元素
- 每个线程
  - 载入矩阵A中的一行
  - 载入矩阵B中的一列
  - 为每对A和B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, 受存储器带宽影响很大;
  - 矩阵的大小受到线程块所能容纳最大线程数（1024）的限制



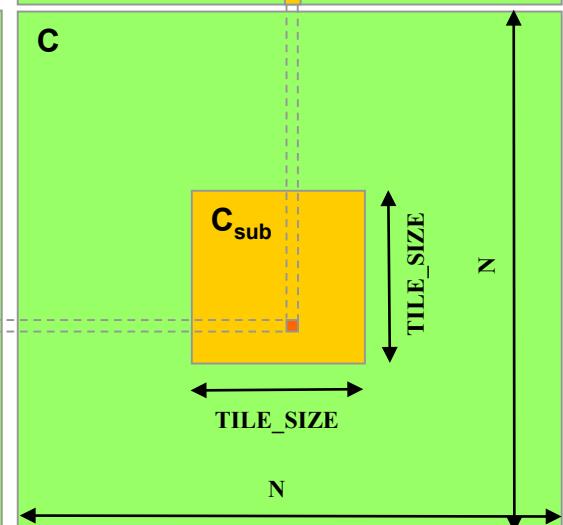
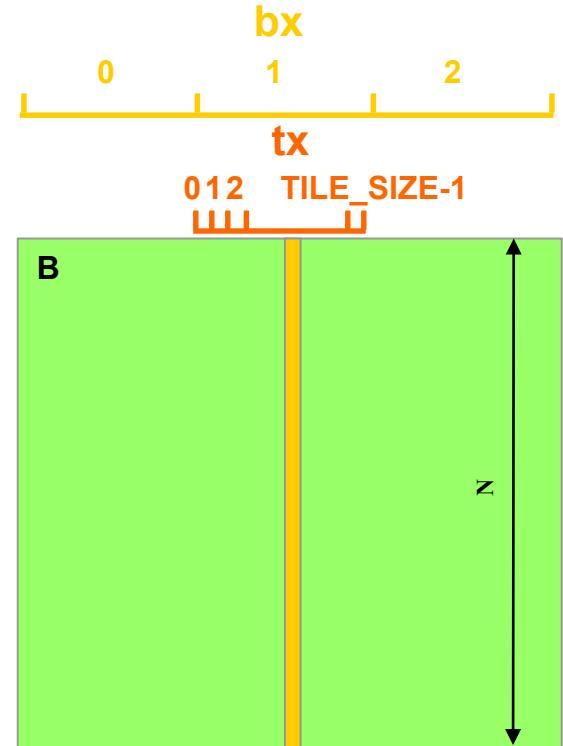
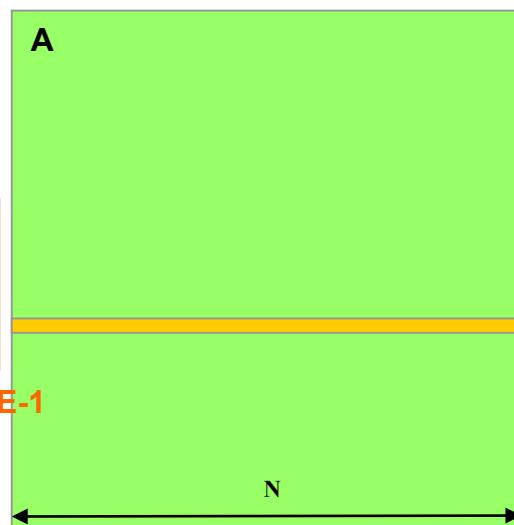
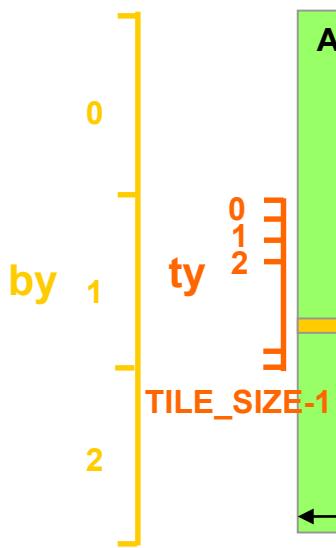
# 分块优化

- 将矩阵分块 (Tile)
- 每个线程块负责一个Tile
- 每个线程负责Tile中的一个元素
- 总共有 $(N/TILE\_SIZE)^2$ 个线程块



# 分块优化

- 每个输入元素都需要被N个线程读取
- 将分块装载到共享存储器中，让多个线程使用



```
#define BLOCK_SIZE 16 // Define the size of the block

__global__ void matrixMulShared(float *A, float *B, float *C, int N) {
 int bx = blockIdx.x;
 int by = blockIdx.y;
 int tx = threadIdx.x;
 int ty = threadIdx.y;

 // Identify the row and column of the C element to work on
 int Row = by * BLOCK_SIZE + ty;
 int Col = bx * BLOCK_SIZE + tx;

 float Cvalue = 0;

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

 for (int m = 0; m < (N / BLOCK_SIZE); ++m) {
 As[ty][tx] = A[Row * N + (m * BLOCK_SIZE + tx)];
 Bs[ty][tx] = B[(m * BLOCK_SIZE + ty) * N + Col];
 __syncthreads();

 for (int k = 0; k < BLOCK_SIZE; ++k) {
 Cvalue += As[ty][k] * Bs[k][tx];
 }
 __syncthreads();
 }

 C[Row * N + Col] = Cvalue;
}

// Setup the execution configuration
dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
dim3 dimGrid(N / BLOCK_SIZE, N / BLOCK_SIZE);
```