

COMP4007: 并行处理和体系结构

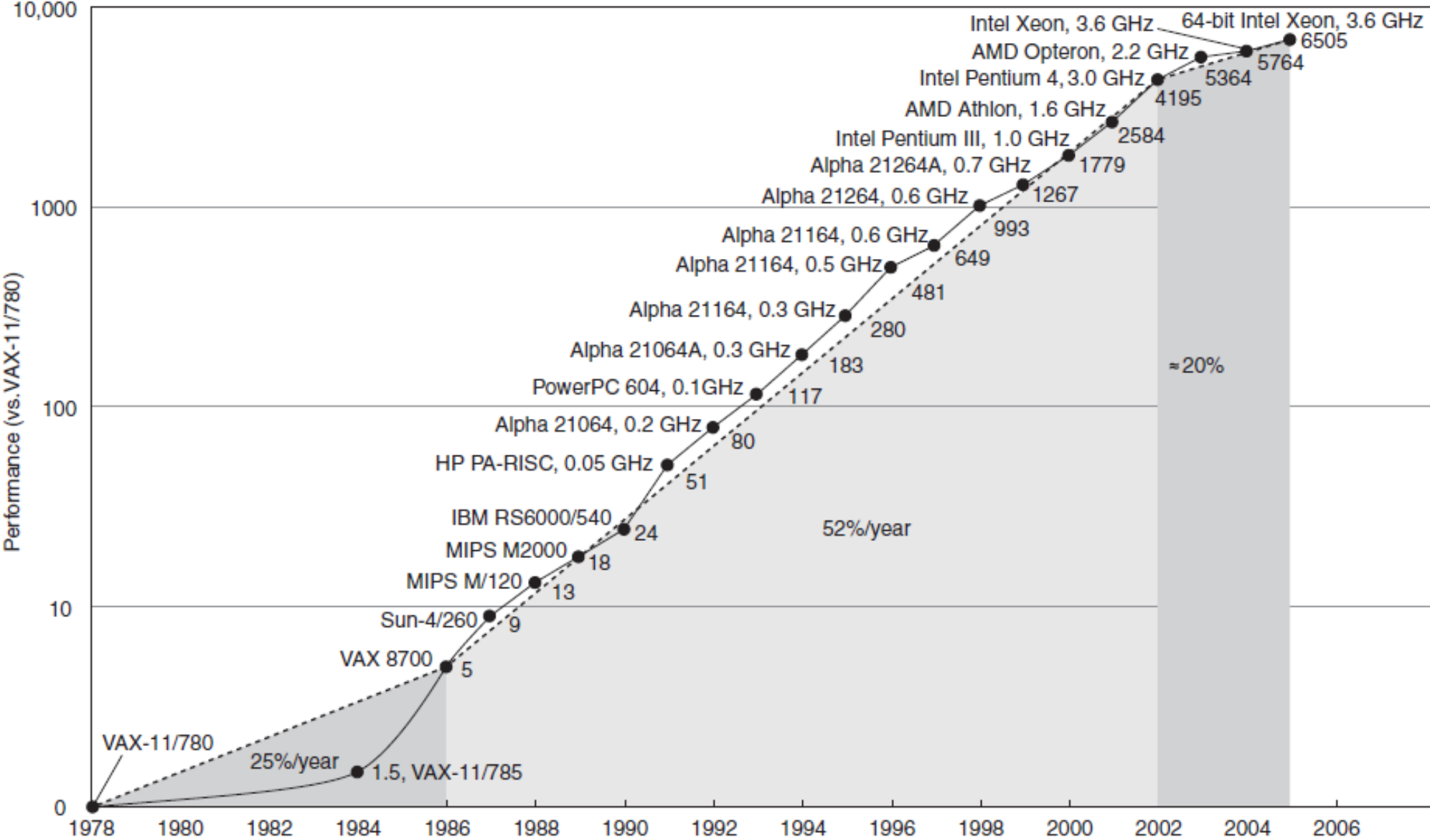
第五章： GPU计算和CUDA并行编程I

授课老师：王强、施少怀
助 教：刘虎成、林稳翔

哈尔滨工业大学（深圳）

- ▶ 基于图形处理器(Graphics Processing Unit, GPU)的计算简介
 - ✎ 处理器发展历史
 - ✎ 架构
- ▶ 统一计算架构编程模型(Compute Unified Device Architecture, CUDA)基础
 - ✎ workflow
 - ✎ 基本接口
 - ✎ 示例

处理器性能发展历史



- ▶ 2002 年之前，通过提高处理器的工作频率实现处理器性能的提升
- ▶ 2002年之后，处理器频率保持在 2GHz - 4GHz 之间
 - ✎ 处理器面临散热问题
 - ✎ 主要通过引入并行性提高性能
 - ☞ 多核心
 - ☞ SIMD(如MMX, SSE, AVX等指令集)

▶ 多核 CPU 将两个或多个独立内核整合到一个封装中

✎ IBM 于 2001 年推出的 POWER4 在单个芯片上集成了两个完整的 CPU 内核。

☞ POWER5, POWER6, POWER7 (4-8核)

✎ Sun 于 2005 年发布了配备 8 个内核的 Niagara UltraSPARC 处理器，并于 2010 年发布了配备 16 个内核的 SPARC T3 处理器

✎ 英特尔和 AMD 于 2005 年推出了双核处理器

☞ 2014 英特尔于 2006 年推出 4 核处理器，2010 年推出 8 核处理器，2011 年推出 10 核处理器，2014 年推出 18 核处理器

☞ AMD 于 2007 年发布 4 核处理器，2010 年发布 12 核处理器，2014 年发布 16 核处理器

▶ 众核处理器比多核处理器拥有更多的特殊的处理单元

✎ 它们辅助处理器进行特殊计算，被称为协处理器(Coprocessors)

▶ 英特尔 MIC: 多集成核心架构

✎ 处理器名: 英特尔至强(Intel Xeon Phi)

✎ 包含约 60 个拥有SIMD单元(向量处理单元, vector processing unit)的核心

✎ 兼容x86指令集

✎ 被许多排名前 500 的超级计算机使用(如天河II)

▶ 图形处理器[Graphics Processing Unit, GPU]

✎ 英伟达(Nvidia): GTX980, Tesla K40, RTX3090, Tesla A100

✎ AMD: FirePro S10000, MI100

✎ 在排名前 500 的超级计算机中也很受欢迎

多核 vs. 众核

年份	处理器	核数	单精度浮点 峰值计算性能	双精度浮点 峰值计算性能	半精度浮点 峰值计算性能	Bandwidth内存 带宽	功率
2010	Intel Xeon X5650	6	0.128T	0.064T	N/A	32GB/s	95W
2013	Intel Xeon Phi 7120p	61	2.4T	1.2T	N/A	352GB/s	300W
2013	Nvidia Tesla K40	2880	4.29T	1.43T	N/A	288GB/s	235W
2012	AMD FirePro S10000 ¹	2x1792	5.91T	1.48T	N/A	480GB/s	375W
2020	Nvidia Tesla A100	6912	19.5T	9.7T	312T	1555GB/s	400W
2020	AMD Instinct™ MI100	7680	23.1T	11.5T	184.6T	1228.8GB/s	300W

¹AMD FirePro S10000拥有2块GPU芯片

图形处理器[Graphics Processing Unit, GPU]



哈爾濱工業大學(深圳)
HARBIN INSTITUTE OF TECHNOLOGY, SHENZHEN

▶ GPU用于加速创建输出到显示器的图像

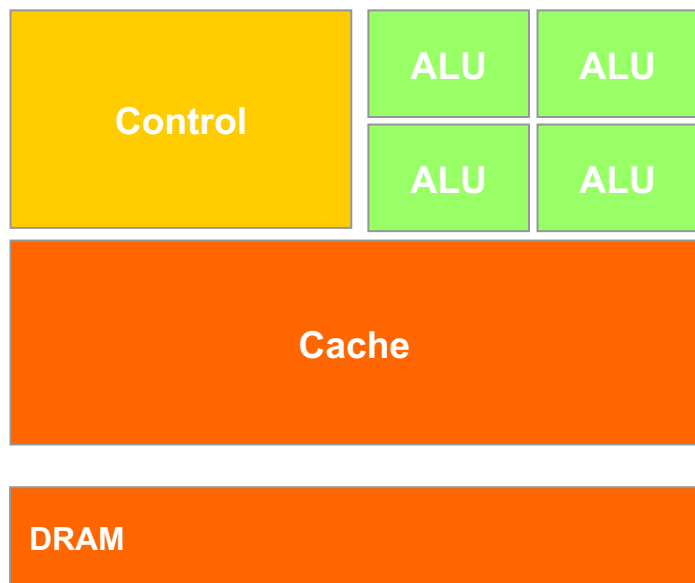
✎分辨率为 1920x1080 的显示器每秒需处理数十帧图像，每帧图像需要处理 200 万像素点的信息！

▶ 如今的GPU使用数百到数千个核心进行计算

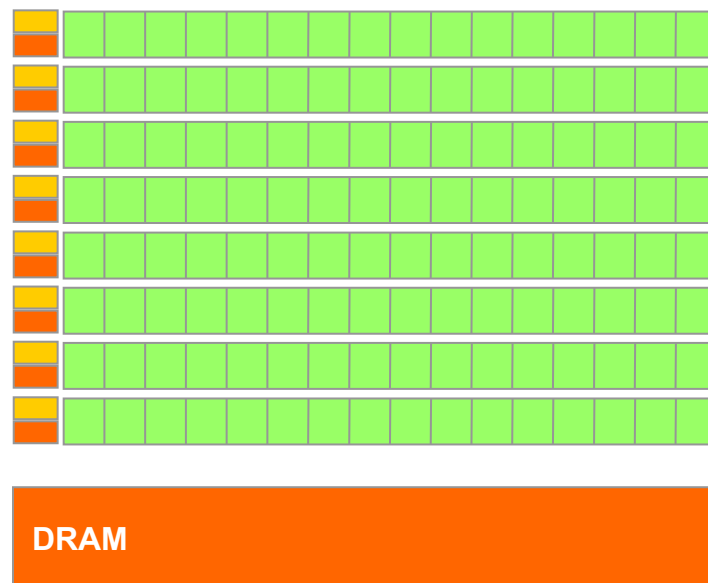
✎每个核心的性能和功能都比 CPU 核心弱

✎成百上千的 GPU 核心拥有强大的性能

不同的设计理念



- ▶ 针对低延迟访问缓存数据集进行了优化
- ▶ 针对乱序执行(out-of-order)和预测执行(speculative execution)进行了优化

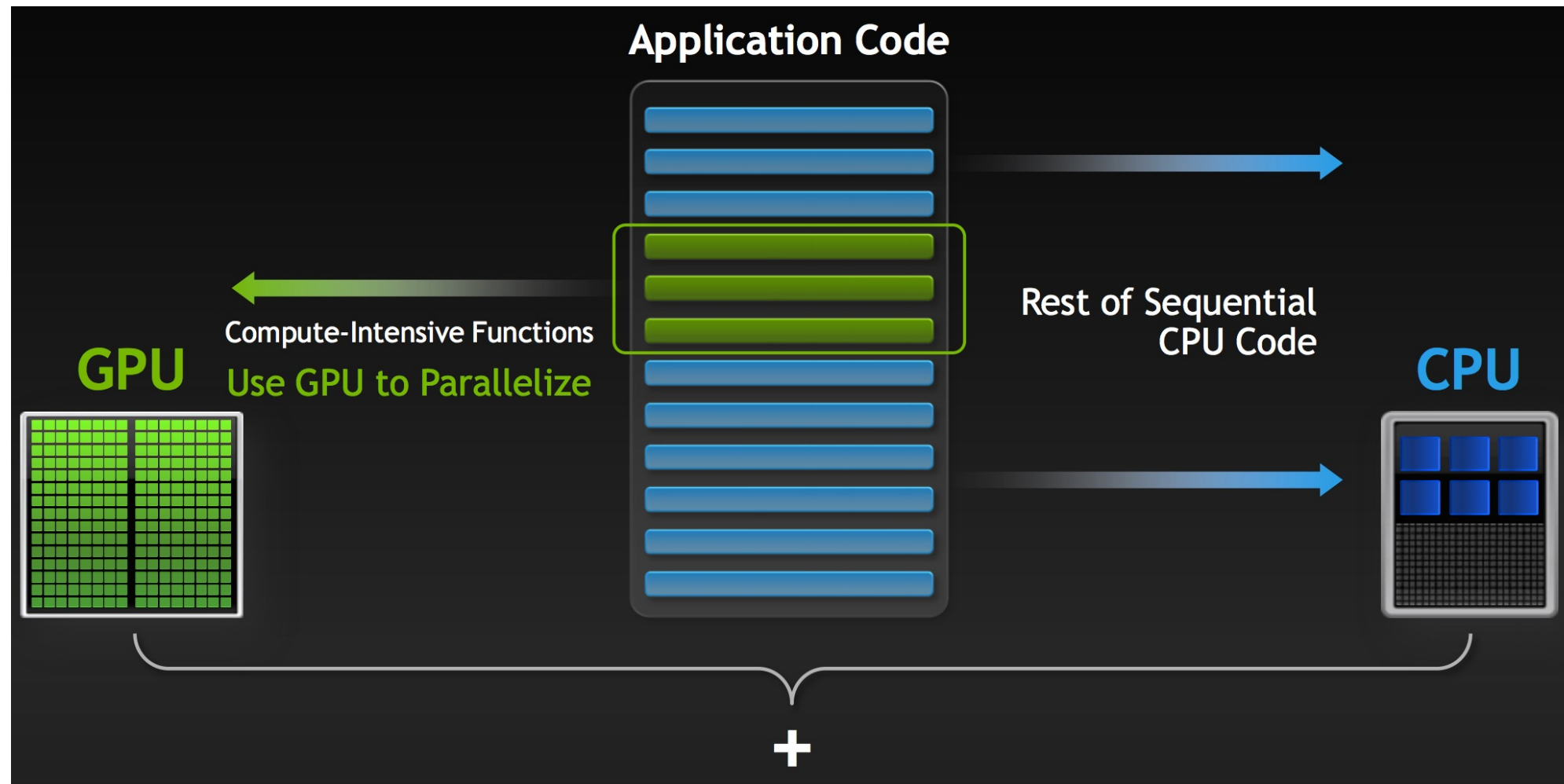


- ▶ 针对数据并行、高通量计算进行了优化
- ▶ 可容忍内存延迟的架构
- ▶ 更多的计算单元

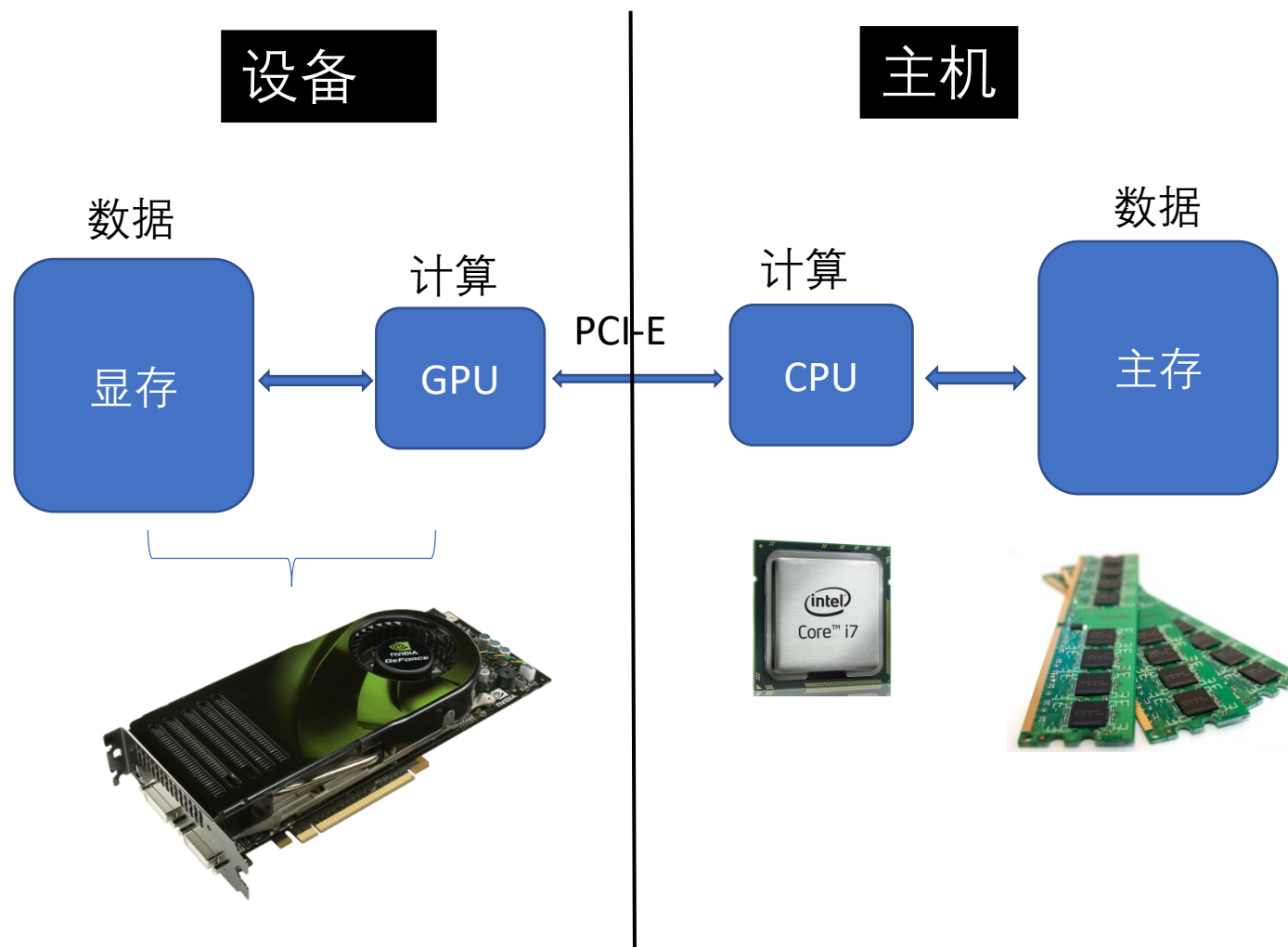
GPGPU: 基于GPU的通用计算

- ▶ 专为 2D/3D 图形设计
- ▶ GPGPU 是指使用 GPU 加速图形处理以外的其他应用程序
 - ✎ 计算金融、数据挖掘、机器学习、数据分析、成像与视觉、生物信息学、计算机辅助设计、分子动力学、量子化学等。
 - ✎ 也称为“GPU计算”

小变化，大提速



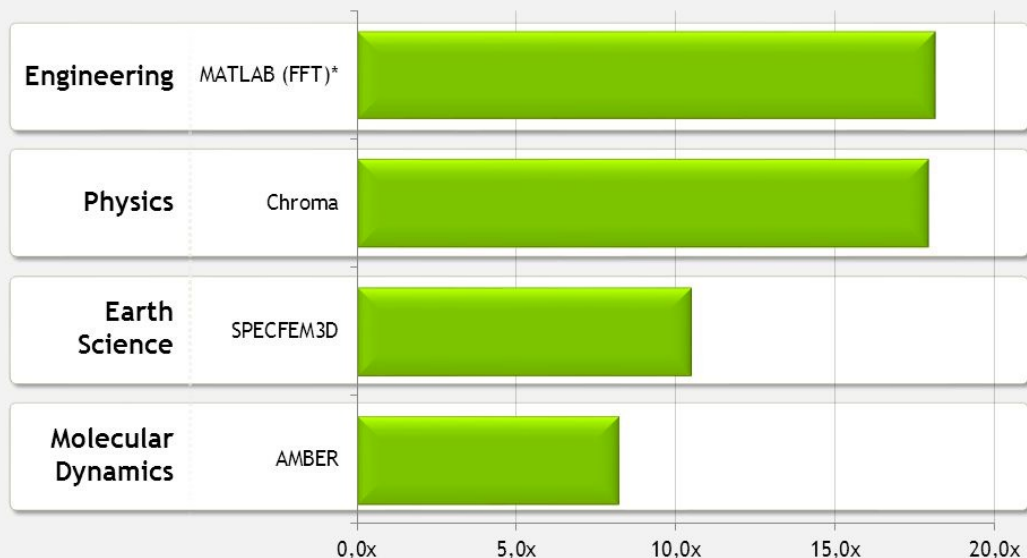
异构计算



GPU 加速科学与工程发展

Fastest Performance on Scientific Applications

Tesla K20X Speed-Up over Sandy Bridge CPUs



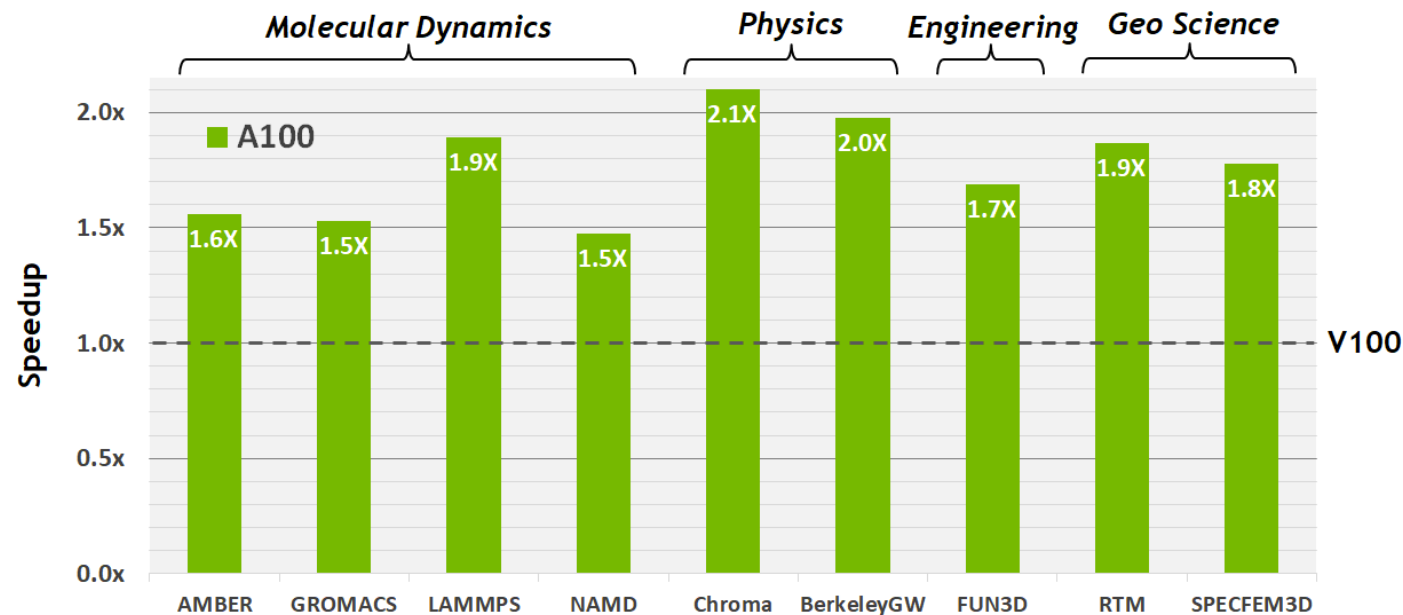
CPU results: Dual socket E5-2687w, 3.10 GHz, GPU results: Dual socket E5-2687w + 2 Tesla K20X GPUs

*MATLAB results comparing one i7-2600K CPU vs with Tesla K20 GPU

Disclaimer: Non-NVIDIA implementations may not have been fully optimized

© NVIDIA 2013

ACCELERATING HPC



All results are measured

Except BerkeleyGW, V100 used is single V100 SXM2, A100 used is single A100 SXM4

More apps detail: AMBER based on PME-Cellulose, GROMACS with STMV (h-bond), LAMMPS with Atomic Fluid LJ-2.5, NAMD with v3.0a1 STMV_NVE

Chroma with szsc121_24_128, FUN3D with dpw, RTM with Isotropic Radius 4 1024^3, SPECFEM3D with Cartesian four material model

BerkeleyGW based on Chi Sum and uses 8xV100 in DGX-1, vs 8xA100 in DGX A100

GPU几乎是深度学习的必备硬件！

GPU架构的两个主要组成部分

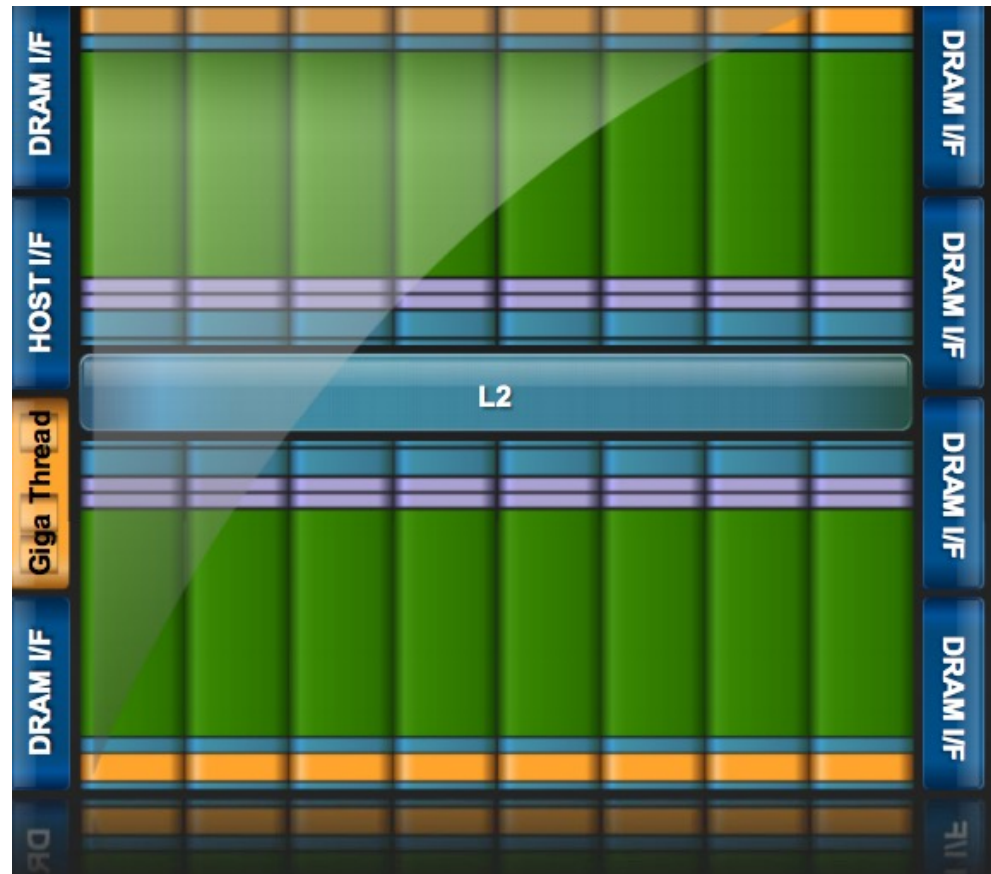


▶ 全局存储(Global memory)

- ✎ 类似于CPU服务器中的主存
- ✎ GPU 和 CPU 均可访问
- ✎ 目前最大为 80 GB
- ✎ Tesla系列的带宽目前高达 2,039 GB/s
- ✎ Quadro 和 Tesla 系列产品有 ECC 开/关选项

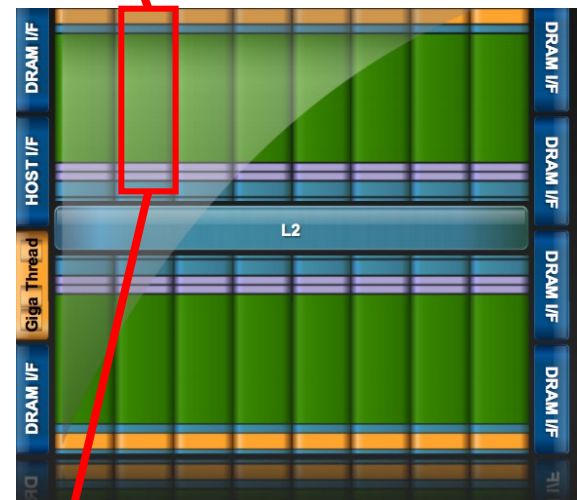
▶ 流式多处理器[Streaming Multiprocessors, SMs]

- ✎ 执行计算
- ✎ SM拥有:
 - ✎ 控制单元
 - ✎ 寄存器
 - ✎ 执行流水线
 - ✎ 缓存

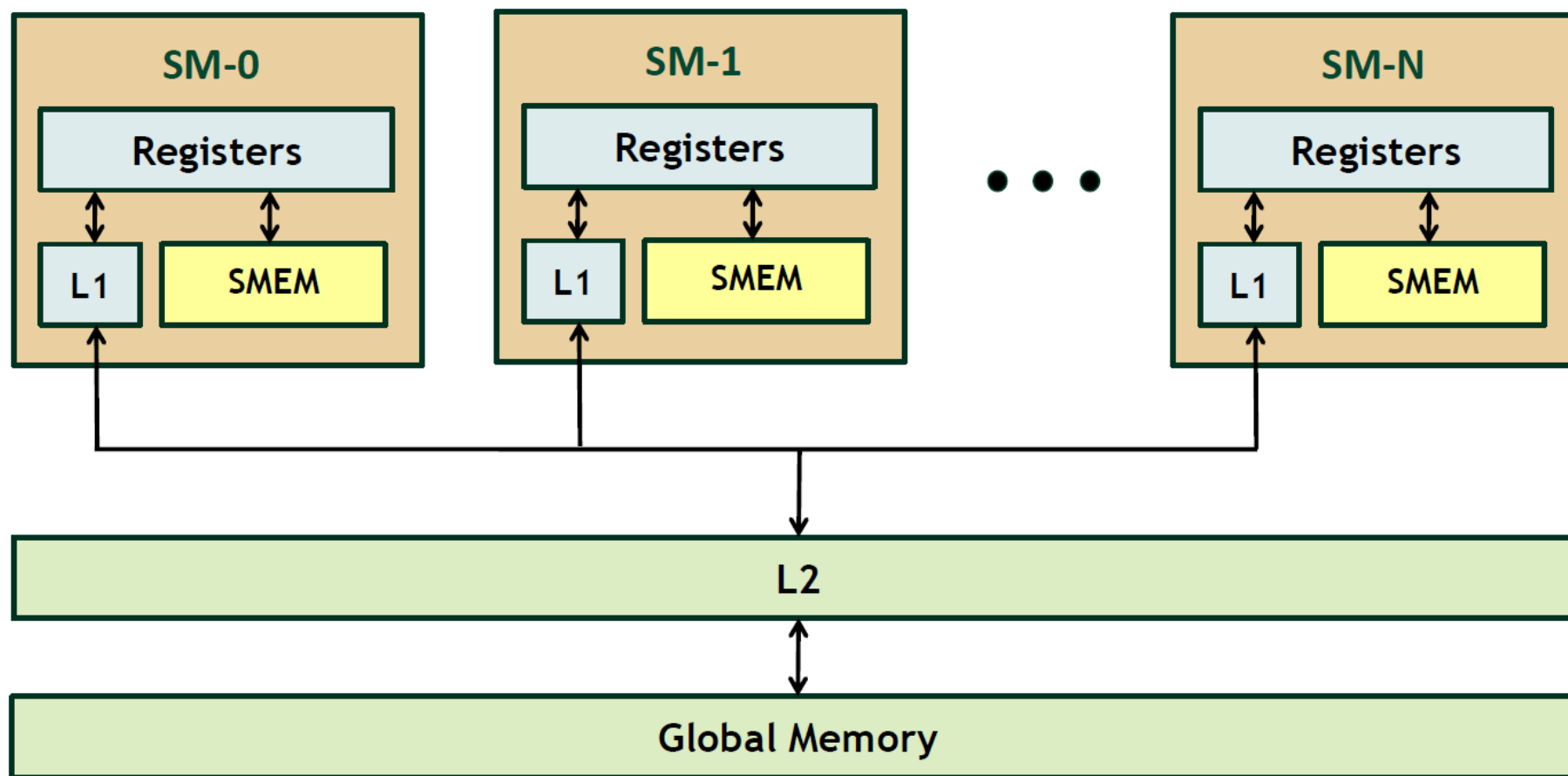


费米(Fermi): 流式多处理器

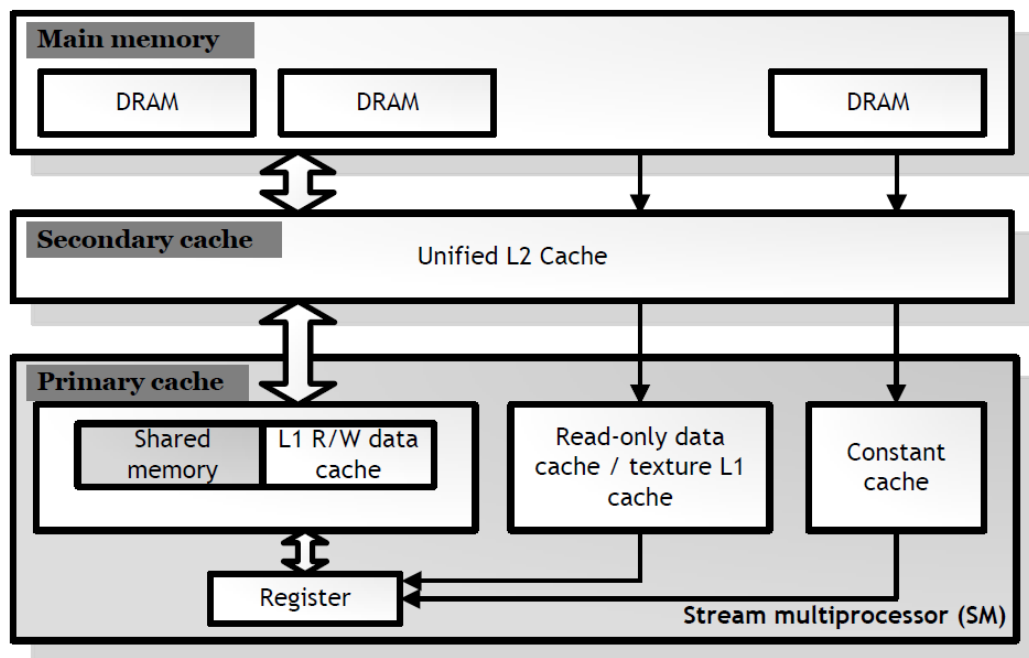
- ▶ 每个SM拥有32个CUDA核
 - ✎ 32次 fp32 操作/周期
 - ✎ 16次 fp64 操作/周期
 - ✎ 32次 int32 操作/周期
- ▶ 2个warp调度器
 - ✎ 最多可并发1536个线程
- ▶ 4 个特殊功能单元
- ▶ 64KB 共享内存 + L1 高速缓存
- ▶ 32K 32 位寄存器



GPU 内存层次结构

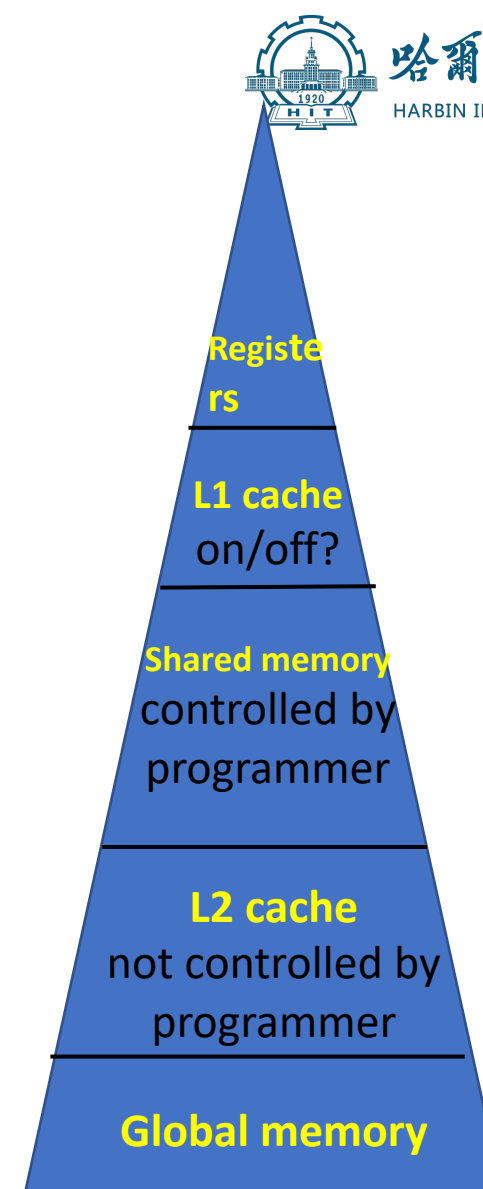


GPU 内存层次结构



▶ 示例: P100(PCIe)

- ✎ 寄存器: 14336 KB
- ✎ L1缓存/共享内存: 64 KB per SM
- ✎ L2缓存: 4096 KB
- ✎ 全局内存: 16 GB
- ✎ 内存(显存)带宽: 732 GB/s

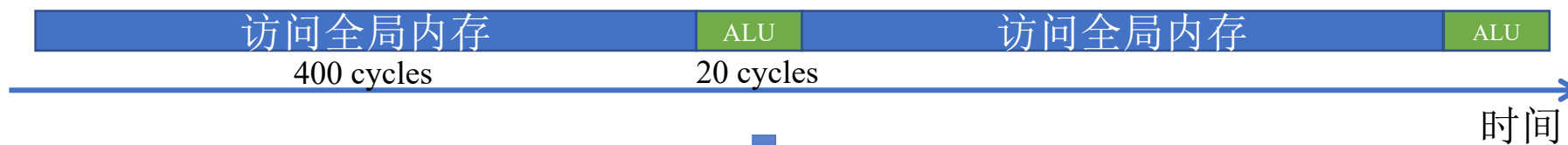


从多线程到高内存延迟

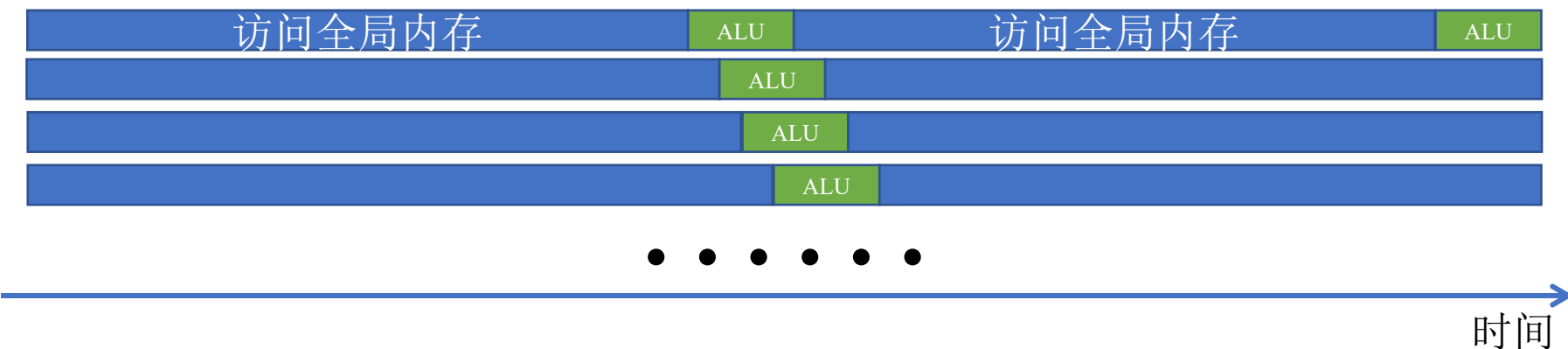
► 计算 vs 内存

✎ 示例, 9340 GFLOPS vs 732 GB/s

单线程视角

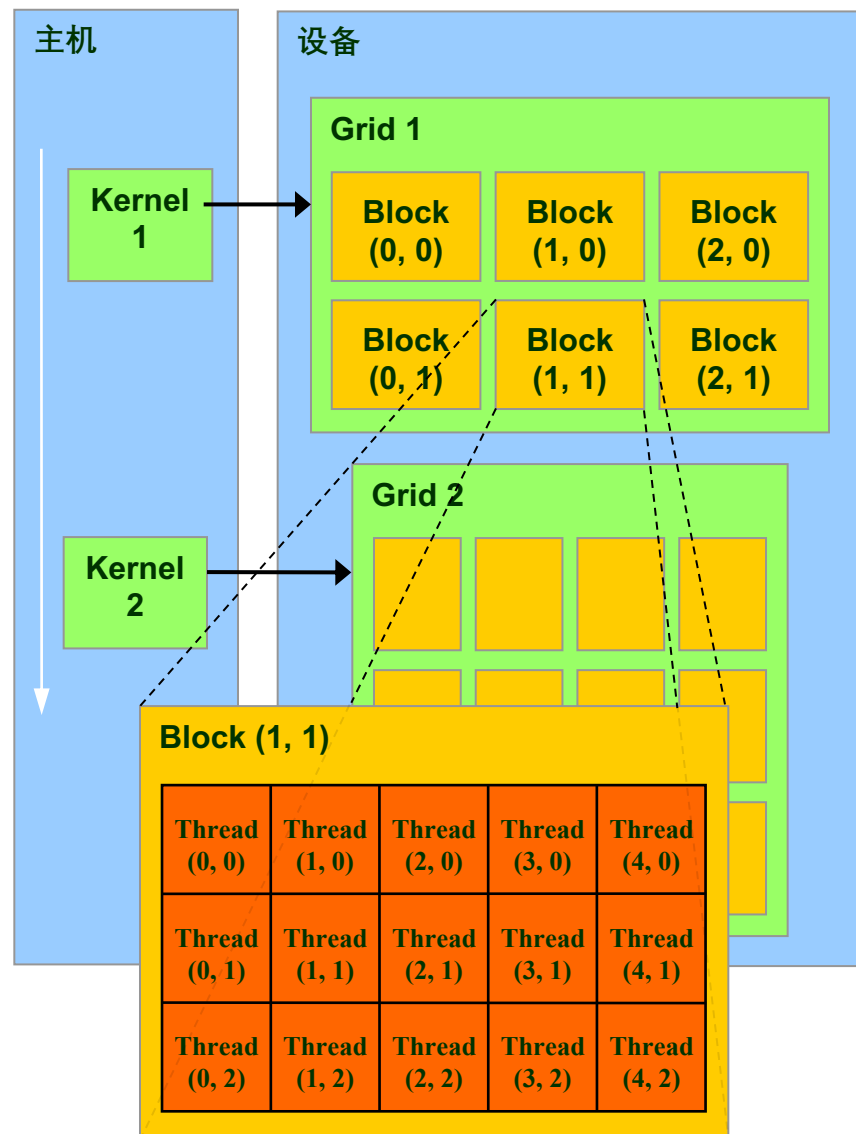


多个线程视角 (针对每个多级 ALU 流水线)

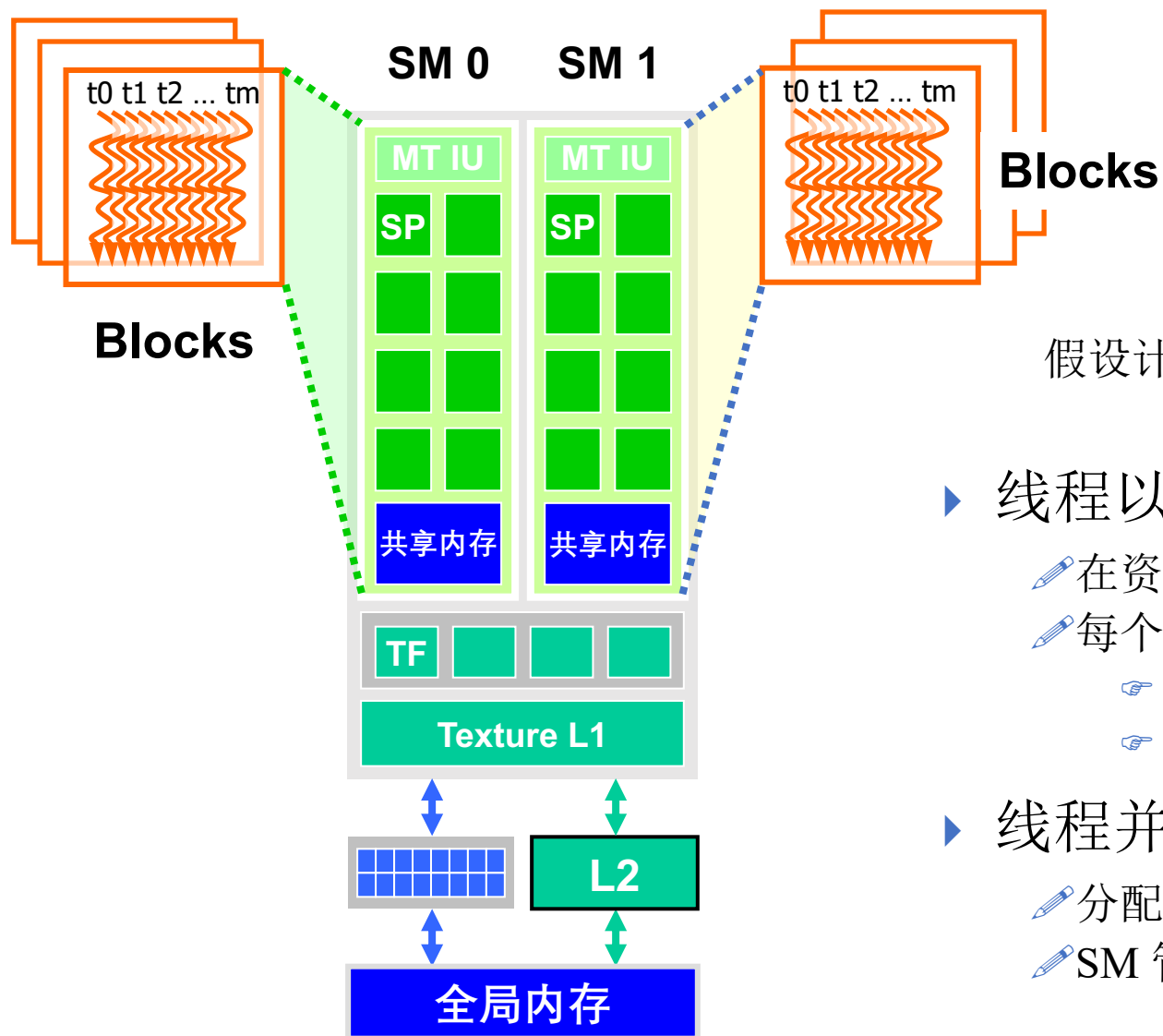


硬件线程生命周期

- ▶ 网格(Grid)由GPU启动
- ▶ 线程块(Thread Blocks)按顺序分配给所有SM
 - ✎ 每个SM可能被分配超过1个线程块
- ▶ SM以Warp为单位启动一组线程
 - ✎ 2级并行
- ▶ 就绪的线程以Warp为单位被SM调度和执行
- ▶ 资源会在所有的Warp和线程块运行完成后释放
 - ✎ GPU可被分配更多线程块



SM 执行线程块



假设计算能力为5.0:

- ▶ 线程以线程块(Block)大小为粒度被分配给SM
 - ✎ 在资源允许的情况下, 每个 SM 最多可被分配 32 线程块
 - ✎ 每个SM最多可被分配2048个线程
 - ✎ 可能是 1024 (线程/块) * 2 块
 - ✎ 或 64 (线程/块) * 32 块等。
- ▶ 线程并行运行
 - ✎ 分配/维护线程 ID
 - ✎ SM 管理/调度线程执行

线程调度/执行

- 每个线程块被划分为多个包含32个线程的Warp

✎ 这是实现时的决策，并非 CUDA 编程模型的一部分

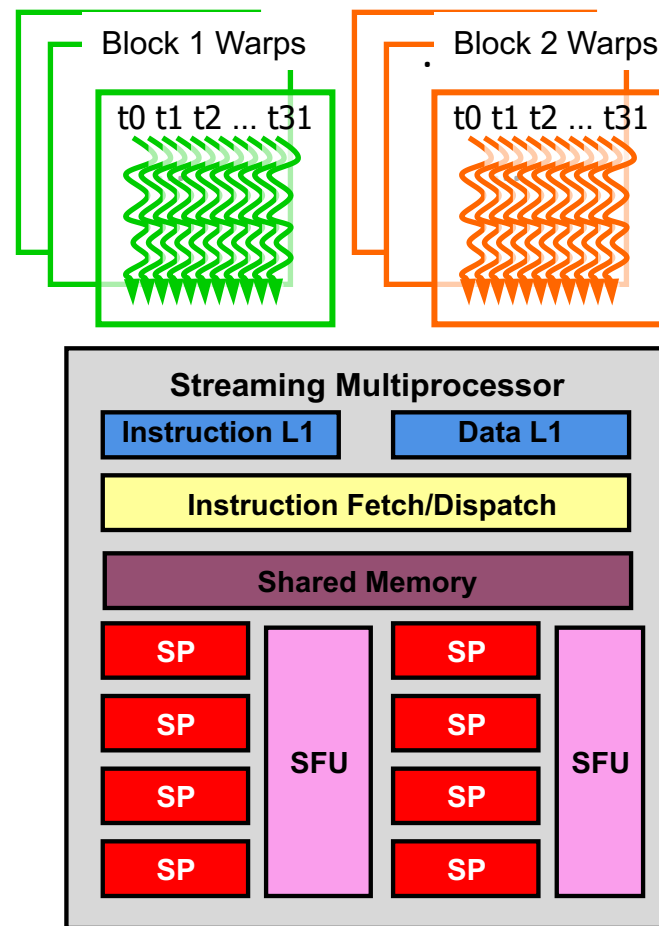
- Warp是 SM 中的调度单位

- 如果某个SM被分配了3个线程块，每个线程块有512个线程，那么调度时共有多少个Warp呢？

✎ 每个线程块会被划分为 $512/32 = 16$ Warps

✎ 共有 $16 * 3 = 48$ Warps

✎ 在任何时间点，48 个 Warps 中的某些会被选择进行取指和执行

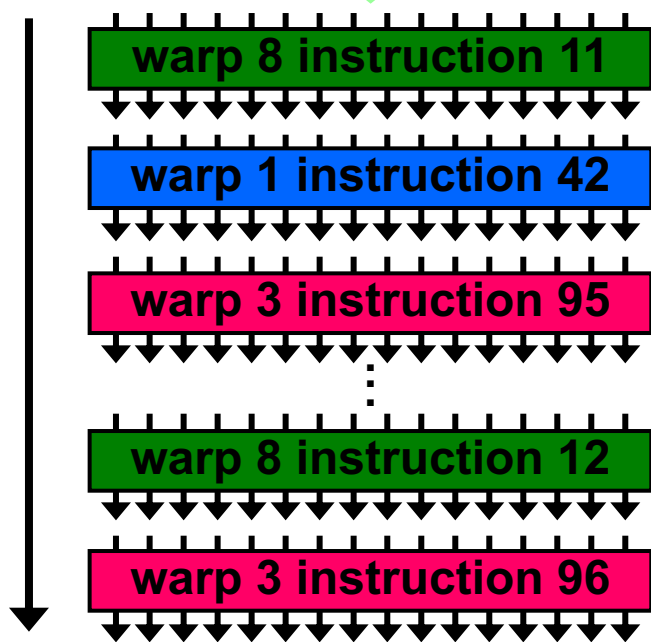


SM Warp调度



SM multithreaded
Warp scheduler

time

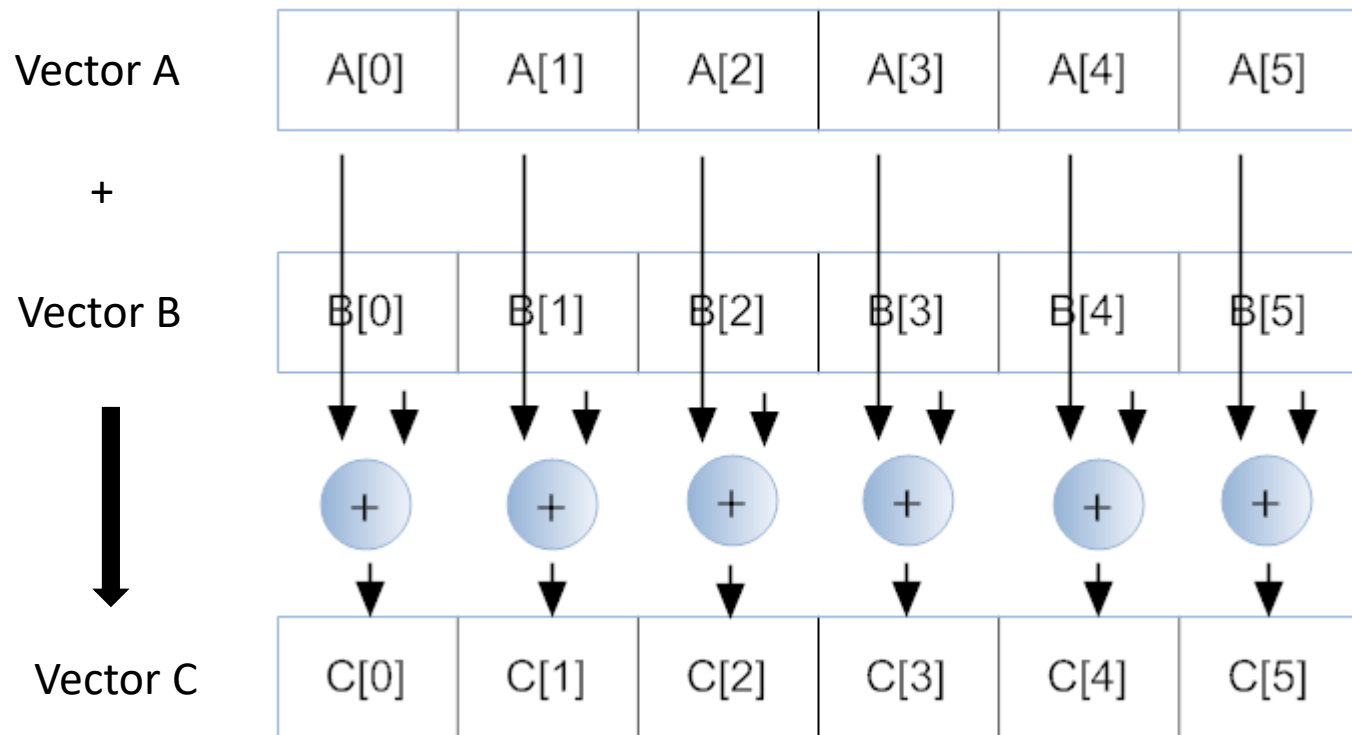


- ▶ SM 在硬件层面实现对 Warp 的零开销调度
 - ✎ 下一条指令所需要数据已就绪的 Warp 可被执行
 - ✎ 符合条件的 Warp 会依据优先调度策略被选择和执行
 - ✎ 被选择执行的 Warp 中的所有线程都执行相同的指令
- ▶ 示例: 在 G80 中, 为 Warp 中的所有线程发送相同指令需要 4 个时钟周期
 - ✎ 如果每执行 4 条指令需要访问一次全局内存
 - ✎ 那么最少需要 13 个 Warps 才能完全承受 200 个周期的内存延迟

回顾: 数据并行



- 在数据并行中，每个处理器对不同的数据执行相同的任务




▶ CUDA

 面向 Nvidia GPU

▶ OpenCL

 主流操作系统上提供跨设备支持

 CPU, GPUs (Nvidia, AMD, Intel), FPGA,

▶ OpenACC

 基于指令的性能可移植并行编程模型

CUDA: 计算统一设备架构

▶ CUDA是一种面向异构计算的通用编程模型

✎ 出于简化GPU计算的目的由Nvidia于2007年提出

✎ 用户可通过它在 GPU 上生成大量线程

✎ GPU 成为一种利用超级线程并行处理海量数据的协处理器

✎ CUDA 包括库、编译器和编程语言扩展

▶ CUDA 计算设备

- ✎ CPU 或主机的协处理器
- ✎ 拥有自己的 DRAM(设备内存)
- ✎ 能够同时运行众多线程
- ✎ 通常被认为是 GPU，但也可以是其他类型的并行处理设备

▶ 应用程序的数据并行处理部分表现为在多个线程上运行的设备内核

▶ GPU和 CPU 线程的区别

- ✎ GPU 线程极为轻量
 - ☞ 创建开销极小
- ✎ GPU 需要成千上万个线程才能充分发挥效率
 - ☞ 多核 CPU 只需要几个线程即可充分发挥效率

- ▶ GPU 设备发展迅速，每一代硬件都会引入许多新功能
- ▶ “计算能力”用于识别 GPU 硬件支持的功能
- ▶ “计算能力”包括主版本号和次版本号 (x.y)，例如 1.3、2.0、3.5、5.0、7.0
 - ✎ 主要版本号相同的设备采用相同的核心架构
 - ✎ 次版本号与核心架构的增量改进相对应

▶ SPMD

- ✎ 单程序/进程多数据
- ✎ 任务被分配到多个处理器上同时运行并处理不同的输入
- ✎ MPI 是一个使用分布式内存的 SPMD 实例

▶ 运行于GPU上的SPMD:

- ✎ CUDA 是一个使用共享内存的 SPMD 例子
- ✎ GPU 使用同一程序并行处理多个子任务
- ✎ 子任务可以从全局共享内存中读取数据(“gather”), 也可以将数据写回内存中的任意位置(“scatter”)。

- ▶ CUDA 支持多种编程语言，如 Fortran，C/C++，Java和Python

- ✎ 本课程选择 CUDA C/C++

- ▶ CUDA C/C++ 是对C/C++ 语言的扩展

- ✎ 一组新的关键词

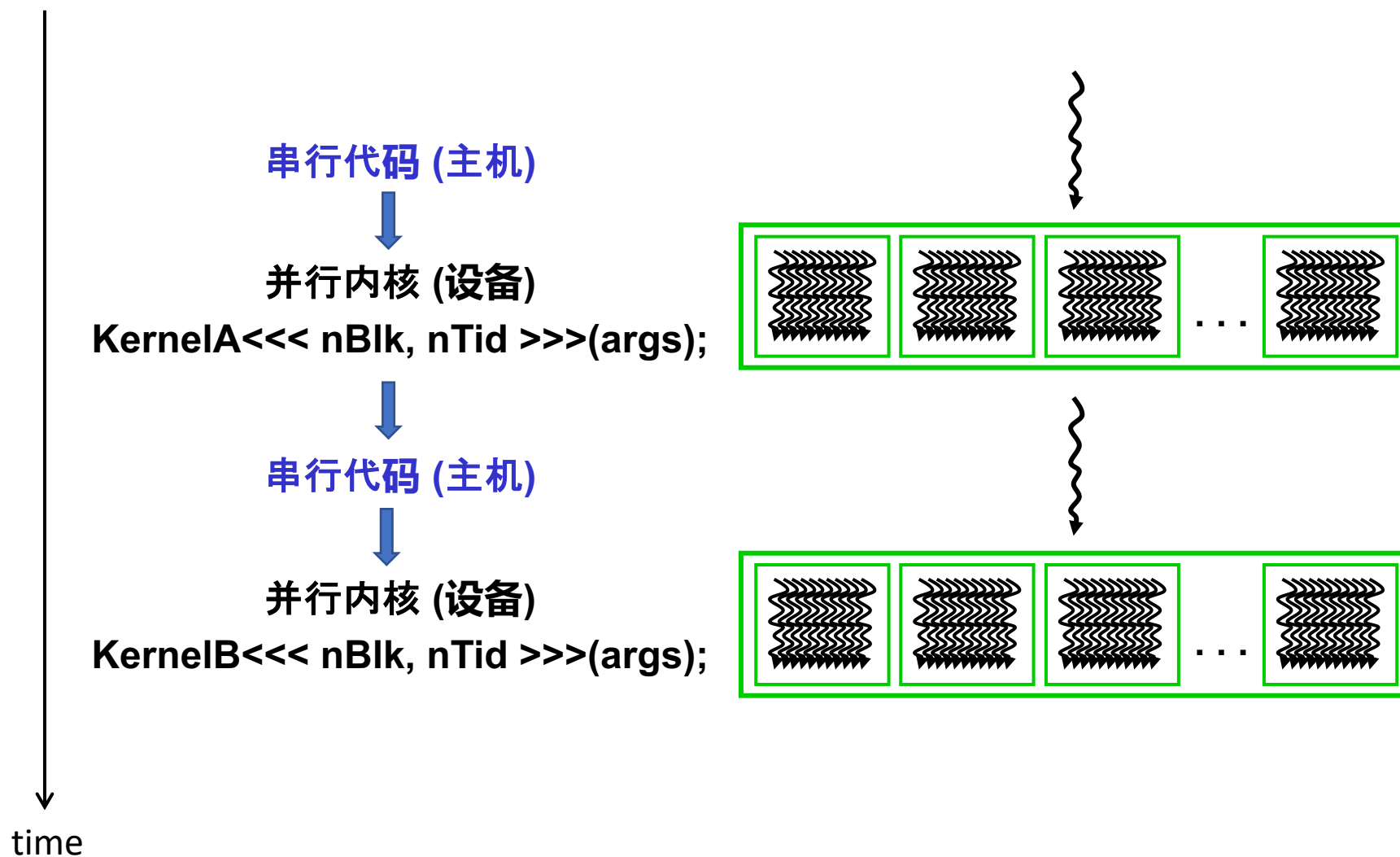
- ✎ 一套应用程序接口

- ▶ CUDA C/C++：集成主机+设备的 C/C++ 程序

- ✎ 主机 C/C++ 代码中的串行或适度并行部分，包括 `main()` 函数

- ✎ 设备 C/C++ 代码中高度并行的部分，称为内核

执行 CUDA C 程序



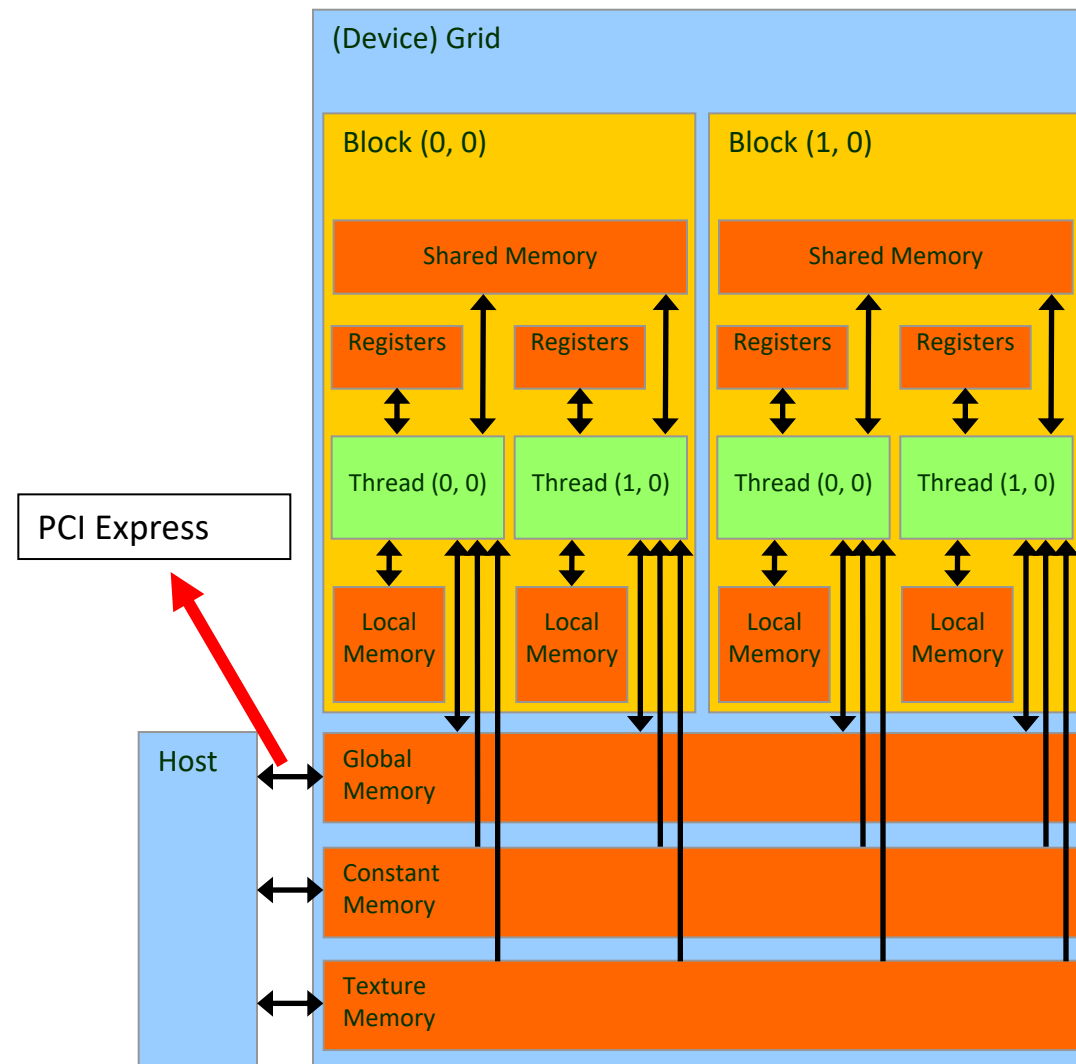
- ▶ 在 CUDA 中，主机（即 CPU）和设备（即 GPU）各自拥有独立的内存空间
- ▶ 在 GPU 上执行内核，需要
 1. 在设备上分配内存
 2. 将数据从主机内存传输到已分配的设备内存
- ▶ 设备执行完毕后，需要将结果数据从设备内存传回主机

CUDA 内存模型

▶ 每个线程均可以:

- ✎ 读/写线程寄存器
- ✎ 读/写线程本地内存
- ✎ 读/写所属线程块共享内存
- ✎ 读/写所属网格全局内存
- ✎ 读所属网格常量内存
- ✎ 读所属网格纹理内存

▶ 主机可以读/写全局、常量和纹理内存



内存操作函数

标准C函数	CUDA C函数	cudaMemcpyKind
malloc	cudaMalloc	cudaMemcpyHostToHost
memcpy	cudaMemcpy	cudaMemcpyHostToDevice
memset	cudaMemset	cudaMemcpyDeviceToHost
free	cudaFree	cudaMemcpyDeviceToDevice

cudaError_t cudaMalloc(void** devPtr, size_t size)

cudaError_t cudaMemcpy(void* dst, const void* src, size_t count,
enum **cudaMemcpyKind** kind, cudaStream_t stream = 0)

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

cudaError_t cudaFree(void* devPtr)

cudaError_t: {cudaSuccess, cudaErrorMemoryAllocation, ...}

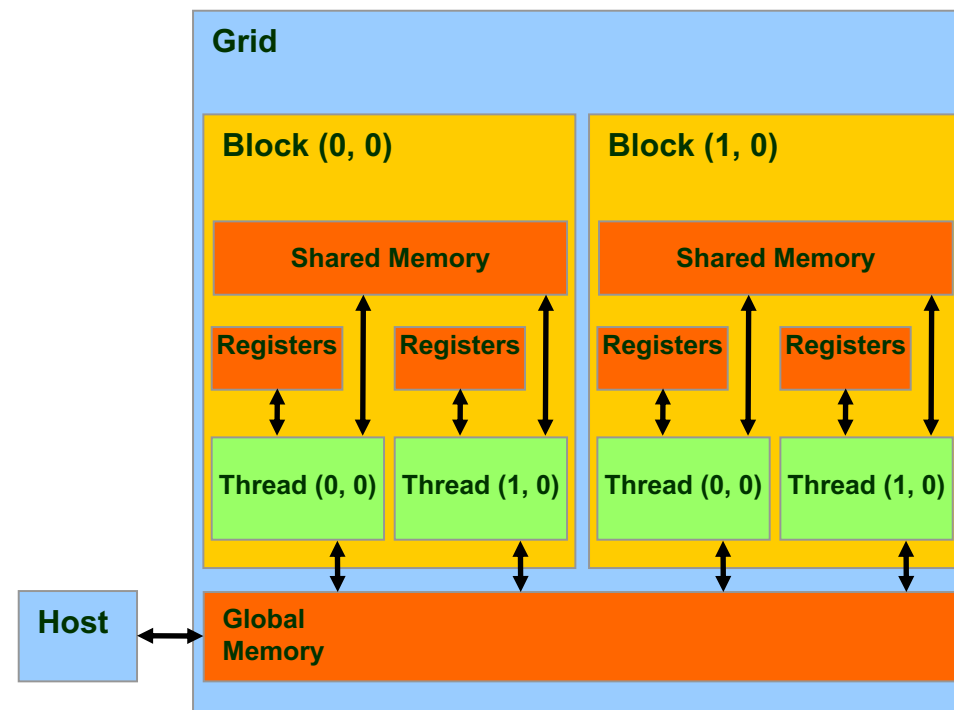
CUDA 设备内存分配

▶ cudaMalloc()

- ✎ 在设备全局内存中分配对象
- ✎ 需要两个参数
- ✎ 分配对象指针的地址
- ✎ 分配对象的大小

▶ cudaFree()

- ✎ 从设备全局内存中释放对象
- ✎ 被释放对象的指针



▶ 代码示例:

- ✎ 分配一个 $64 * 64$ 大小的单精度浮点数组
- ✎ 指针 `M_d` 指向分配的存储空间的首地址
- ✎ “d”通常用来表示设备数据结构

```
TILE_WIDTH = 64;  
float* M_d;  
int size = TILE_WIDTH * TILE_WIDTH * sizeof(float);
```

```
cudaMalloc((void**)&M_d, size);  
cudaFree(M_d);
```

CUDA 主机-设备数据传输

▶ cudaMemcpy()

✎ 主机和设备之间的内存数据传输

✎ 需要四个参数

✎ 指向目的地址的指针

✎ 指向源地址的指针

✎ 传输的字节数

✎ 传输类型

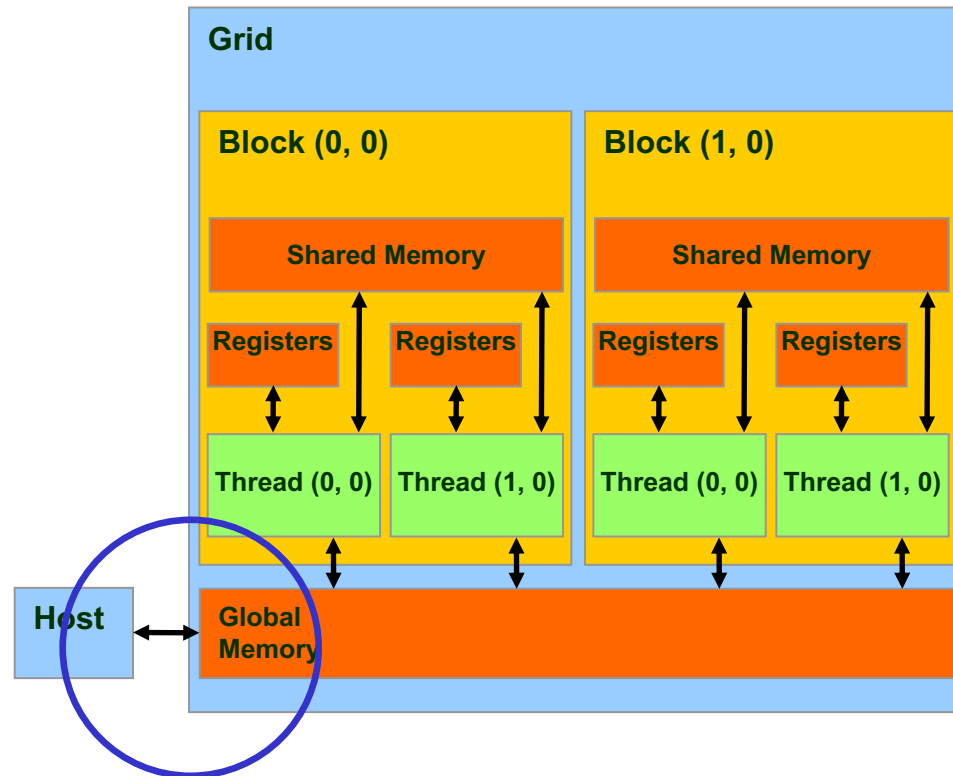
▢ 主机到主机

▢ 主机到设备

▢ 设备到主机

▢ 设备到设备

▶ 不一定能用于GPU之间直接的数据传输



▶ 代码示例:

- ✎ 传输一个 $64 * 64$ 大小的单精度浮点数组
- ✎ `M`指向数据在主机内存中地址, `M_d`指向数据在设备内存中的地址
- ✎ `cudaMemcpyHostToDevice` 和 `cudaMemcpyDeviceToHost` 是符号常数

```
cudaMemcpy(M_d, M, size, cudaMemcpyHostToDevice);
```

```
cudaMemcpy(M, M_d, size, cudaMemcpyDeviceToHost);
```

同步内存复制

▶ cudaMemcpy() 是同步(或阻塞)复制函数

✎ 之前发出的所有 CUDA 调用都完成后才会开始

✎ 在同步 cudaMemcpy() 完成之前，后续的 CUDA 调用不能开始

GPU 上的矢量加法



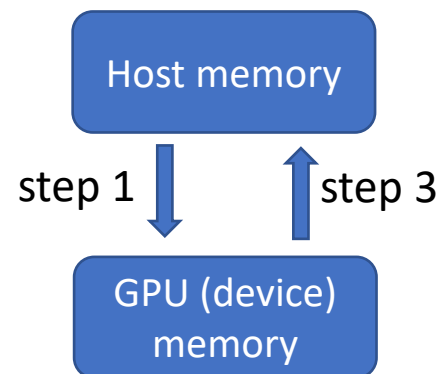
```
#include <cuda.h>
```

```
void vecAdd(float *A, float *B, float *C, int n)
{
    int size = n * sizeof(float);
    float *A_d, *B_d, *C_d;
```

```
//step 1: allocate device memory for A, B, and C; then copy A and B to device memory
```

```
//step 2: launch the kernel code to perform the actual vector addition on GPU
```

```
//step 3: copy C from device memory and free device memory
}
```



- ▶ 内核函数是并行阶段所有线程要执行的代码

- ✎ 所有线程执行相同的代码：SPMD

- ▶ 当主机代码启动内核时，会生成一个线程网格。这些线程按两级层次结构组织。

- ✎ 每个网格由许多线程块组成

- ☞ 每个线程块在对应的网格中都有一个唯一 ID：blockIdx

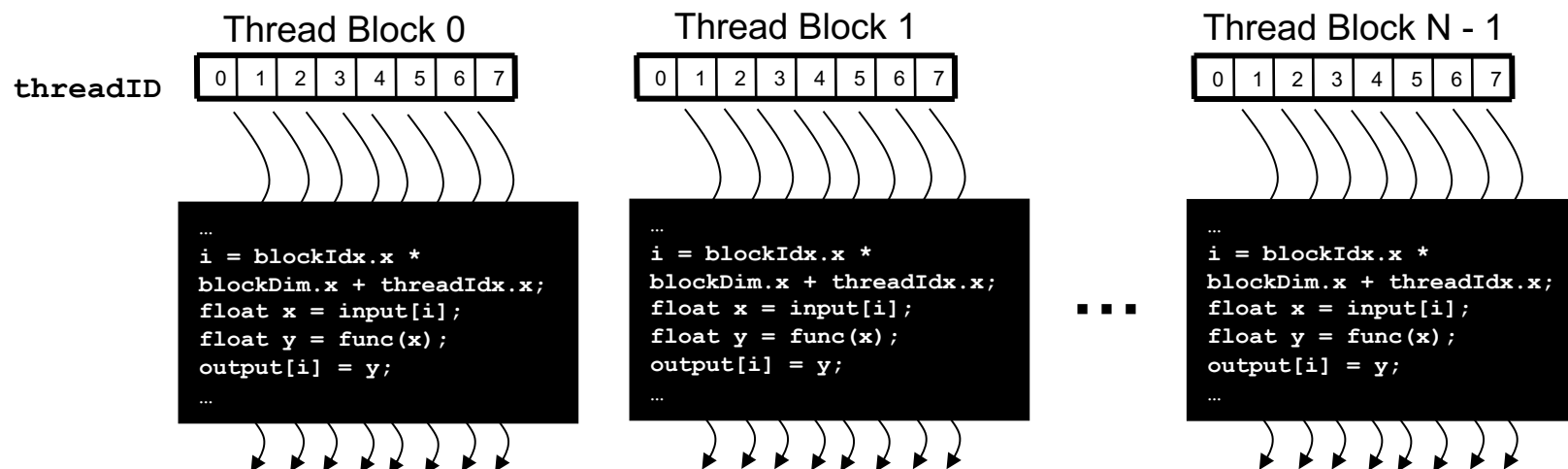
- ✎ 每个线程块包含多个线程

- ☞ 每个线程在对应的线程块中都有一个唯一ID：threadIdx

▸ 网格中的线程被组织成多个线程块

✍ 线程块内的线程通过共享内存、原子操作和同步屏障进行合作

✍ 不同线程块中的线程无法合作



线程块ID和线程ID



▶ 每个线程使用 ID 来决定处理哪些数据

✎ 线程块ID: 1D、2D 或 3D

✎ 线程ID: 1D、2D 或 3D

✎ CUDA 采用 dim3 数据结构表示3D 向量

✎ dim3 实际上是一种C 语言中的结构体, 它包含三个无符号整数字段 x、y 和 z

▶ 预定义内置变量

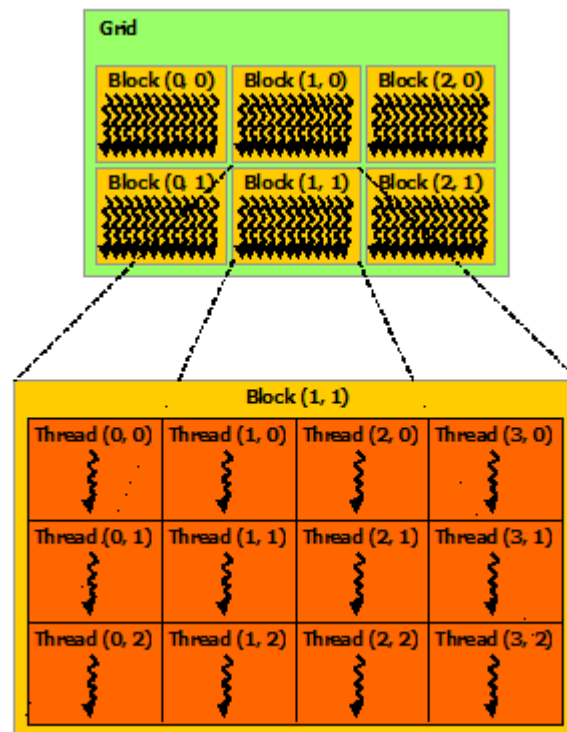
✎ dim3 blockDim: 网格大小

✎ dim3 blockDim: 线程块大小

✎ dim3 blockIdx: 线程块的三维索引

✎ dim3 threadIdx: 线程的三维索引

Example of
2-D grid and 2-D block



	执行位置	调用位置
<code>__device__ float DeviceFunc()</code>	设备	设备
<code>__global__ void KernelFunc()</code>	设备	主机
<code>__host__ float HostFunc()</code>	主机	主机

- ▶ `__global__` 用于定义内核函数
 - ✎ 返回值必须为void
- ▶ `__device__` 和 `__host__` 可同时使用
 - ✎ 编译器会生成两个版本的函数

- ▶ 调用内核函数时必须使用执行配置：

```
__global__ void KernelFunc(...);  
dim3    DimGrid(100, 50);    // 5000线程块  
dim3    DimBlock(4, 8, 8);    // 256个线程/线程块  
size_t  SharedMemBytes = 64; // 64字节共享内存  
KernelFunc<<< DimGrid, DimBlock, SharedMemBytes >>> (...);
```


- ▶ 从 CUDA 1.0 开始，默认对内核函数的任何调用都是异步的，阻塞调用需要显式指定同步方式

- ▶ 在 CUDA 中，调用内核默认是异步操作
- ▶ CPU 将继续执行后面的语句，而不会等待内核执行完毕
 - ✎ 当 CPU 遇到同步 CUDA 应用程序接口（如 `cudaMemcpy()`）时，它会等待前一个内核函数执行完毕
 - ✎ 同步 CUDA 应用程序接口 `cudaDeviceSynchronize()` 可用于阻塞 CPU，直到 GPU 完成之前请求的所有任务。

`cudaError_t cudaDeviceSynchronize(void);`


CUDA 内核限制


▶ 目前，CUDA 内核函数有以下限制

 只能访问设备内存

 返回值必须为void

 不支持可变数量的参数

 不支持静态变量

 不支持函数指针

GPU 示例 1

- ▶ 为 n 个整数分配 CPU 内存
- ▶ 为 n 个整数分配 GPU 内存
- ▶ 将 GPU 内存初始化为 0s
- ▶ 将数据从 GPU 复制到 CPU
- ▶ 输出数值



```
#include <cuda.h>
#include <stdio.h>
int main() {
    int dimx= 16;
    int num_bytes= dimx* sizeof(int);
    int*d_a= 0, *h_a= 0; // device and host pointers
    h_a= (int*)malloc(num_bytes);
    cudaMalloc((void**)&d_a, num_bytes);
    if (0 == h_a|| 0 == d_a) {
        printf("couldn't allocate memory\n");
        return 1;
    }
    cudaMemset(d_a, 0, num_bytes);
    cudaMemcpy(h_a, d_a, num_bytes, cudaMemcpyDeviceToHost);
    for (int i= 0; i< dimx; i++)
        printf("%d\n", h_a[i]);
    free(h_a);
    cudaFree(d_a);
    return 0;
}
```


GPU 示例 2

- ▶ 在GPU中按线程 ID 初始化数组
- ▶ 将数组从GPU复制到CPU中
- ▶ 输出数值

A kernel function:

```
__global__ void mykernel(int* a)  
{  
    int idx= blockIdx.x* blockDim.x+ threadIdx.x;  
    a[idx] = 7;  
}
```



```
#include <cuda.h>
#include <stdio.h>

__global__ void mykernel(int* a) {
    int idx= blockIdx.x* blockDim.x+ threadIdx.x; // locate the data item handled by this thread
    a[idx] = threadIdx.x;
}

int main() {
    int dimx= 16, num_bytes= dimx*sizeof(int);
    int*d_a= 0, *h_a= 0; // device and host pointers
    h_a= (int*)malloc(num_bytes);
    cudaMalloc((void**)&d_a, num_bytes);
    cudaMemcpy(d_a, 0, num_bytes);
    dim3 grid, block;
    block.x= 4; // each block has 4 threads
    grid.x= dimx / block.x; // # of blocks is calculated
    mykernel<<<grid, block>>>(d_a);
    cudaMemcpy(h_a, d_a, num_bytes, cudaMemcpyDeviceToHost);
    for(inti= 0; i< dimx; i++)
        printf("%d\n", h_a[i]);
    free(h_a);
    cudaFree(d_a);
    return 0;
}
```

输出结果:

0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3

GPU 示例 3 : 矢量加法



将在GPU上执行的内核函数

```
// compute vector sum C = A + B
// each thread performs one pair-wise addition
__global__ void vecAdd( float *A, float *B, float *C, int n)
{
    // locate the memory
    int i = threadIdx.x + blockDim.x * blockIdx.x;

    // perform the addition
    if(i < n) C[i] = A[i] + B[i];
}
```

示例: 矢量加法(续)



将在CPU执行的代码

```
int main ()
{
    int n = 10000;
    // allocate and initialize host (CPU) memory
    float *H_A = ..., *H_B = ..., *H_C = ...;
    // allocate device (GPU) memory
    float *A_d, *B_d, *C_d;
    cudaMalloc(...); ...
    // copy host memory to device
    cudaMemcpy(...);...
    // run 16 blocks of 256 threads each
    vecAdd<<< ceil(n/256.0), 256 >>>(d_A, d_B, d_C, n);
    // copy result to host
    cudaMemcpy(...);
    cudaFree(A_d);...
}
```

▶ 使用错误处理宏来封装所有 CUDA API 调用的做法非常普遍

✎ 简化错误检查编码

```
#define CHECK(call)
{
    const cudaError_t error = call;
    if (error != cudaSuccess)
    {
        printf("Error: %s:%d, ", __FILE__, __LINE__);
        printf("code:%d, reason: %s\n", error, cudaGetErrorString(error));
        exit(1);
    }
}
```

E.g.:

CHECK(cudaMemcpy(...));

__FILE__ 和 __LINE__ 是标准的预定义宏，分别表示当前行对应的文件的文件名和在文档中的行位置。

- ▶ 第2章, David B. Kirk and Wen-mei W. Hwu, Programming Massively Parallel Processors, 2nd Edition, Morgan Kaufmann, 2013. [PDF: https://safari.ethz.ch/architecture/fall2019/lib/exe/fetch.php?media=2013_programming_massively_parallel_processors_a_hands-on_approach_2nd.pdf]

- ▶ David B. Kirk and Wen-mei W. Hwu, Programming Massively Parallel Processors, 2nd Edition, Morgan Kaufmann, 2013.
- ▶ CUDA C/C++ Programming Guide: <https://docs.nvidia.com/cuda/cuda-c-programming-guide/>