北京理工大学
BEIJING INSTITUTE OF TECHNOLOGY

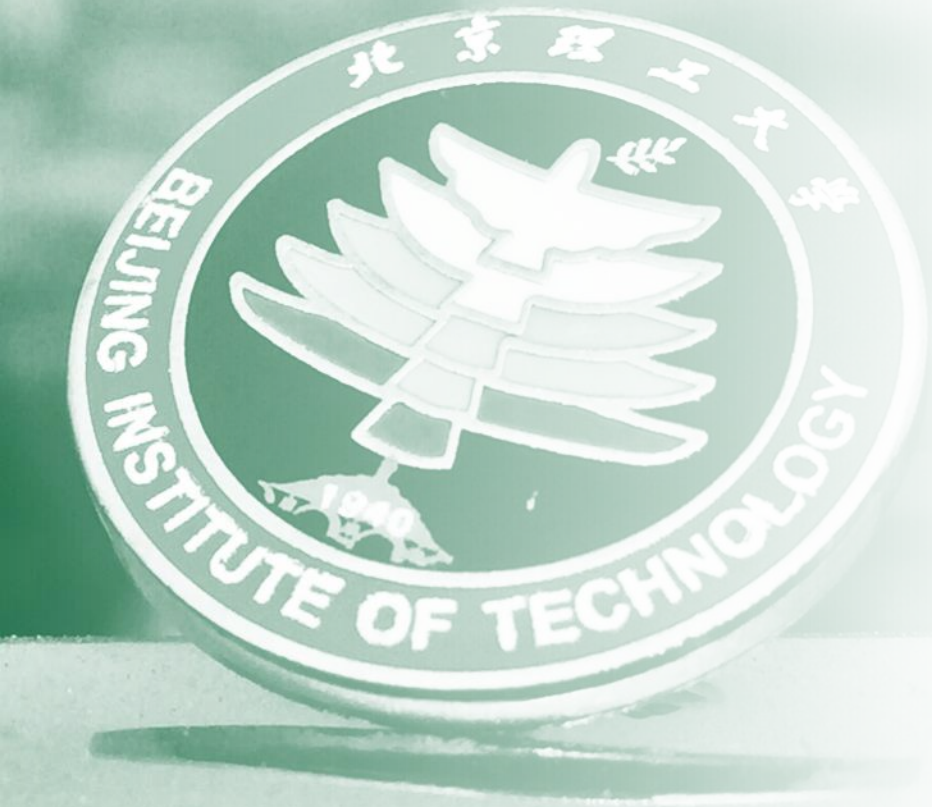
并行编程原理与实践

10. CUDA编程

王一拙、计卫星

北京理工大学计算机学院

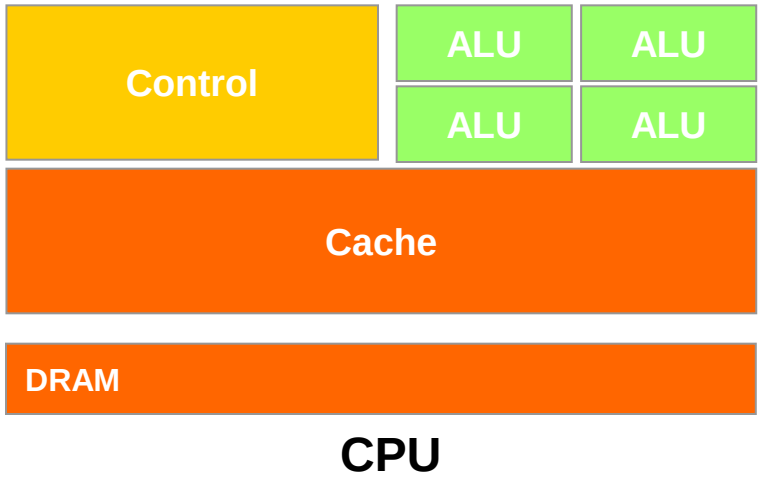
目录 | CONTENTS



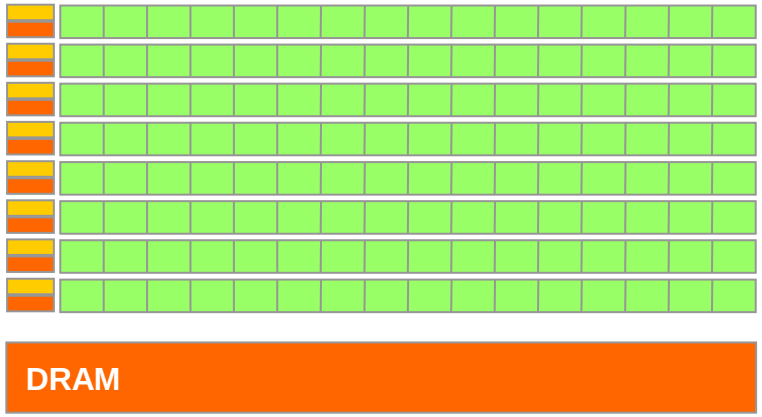
1 GPU编程概述

GPU与CPU硬件架构的对比

- CPU：更多资源用于缓存及流控制
- GPU：更多资源用于数据计算
  - 适合具备可预测、针对数组的计算模式



CPU



GPU

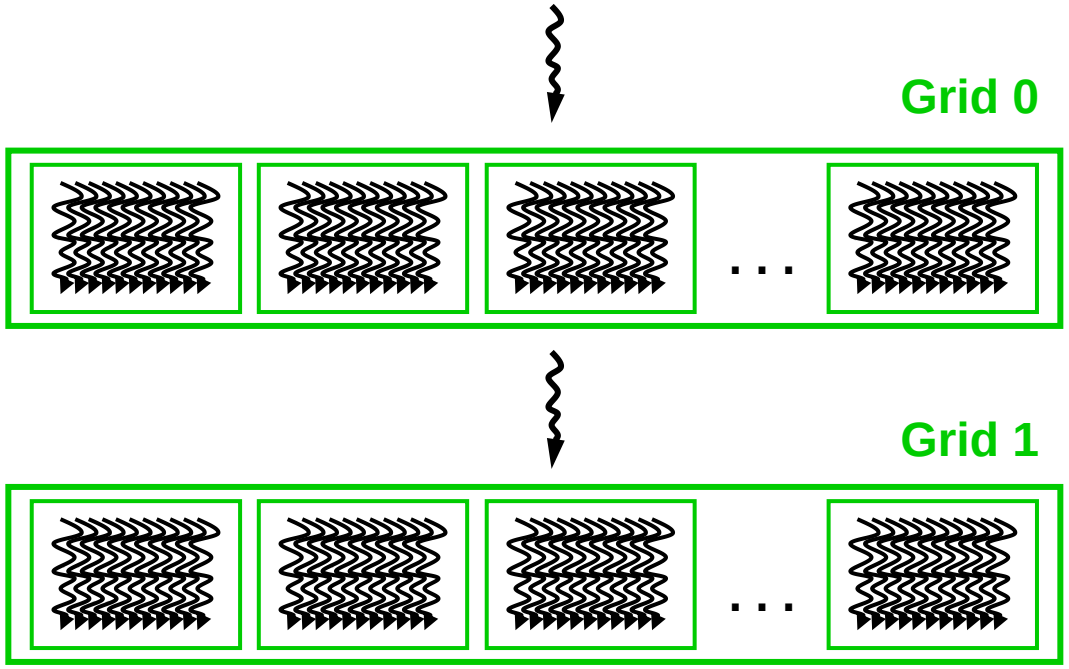
- CUDA (Compute Unified Device Architecture)有效结合CPU+GPU编程
  - 串行部分在CPU上运行
  - 并行部分在GPU上运行

CPU Serial Code

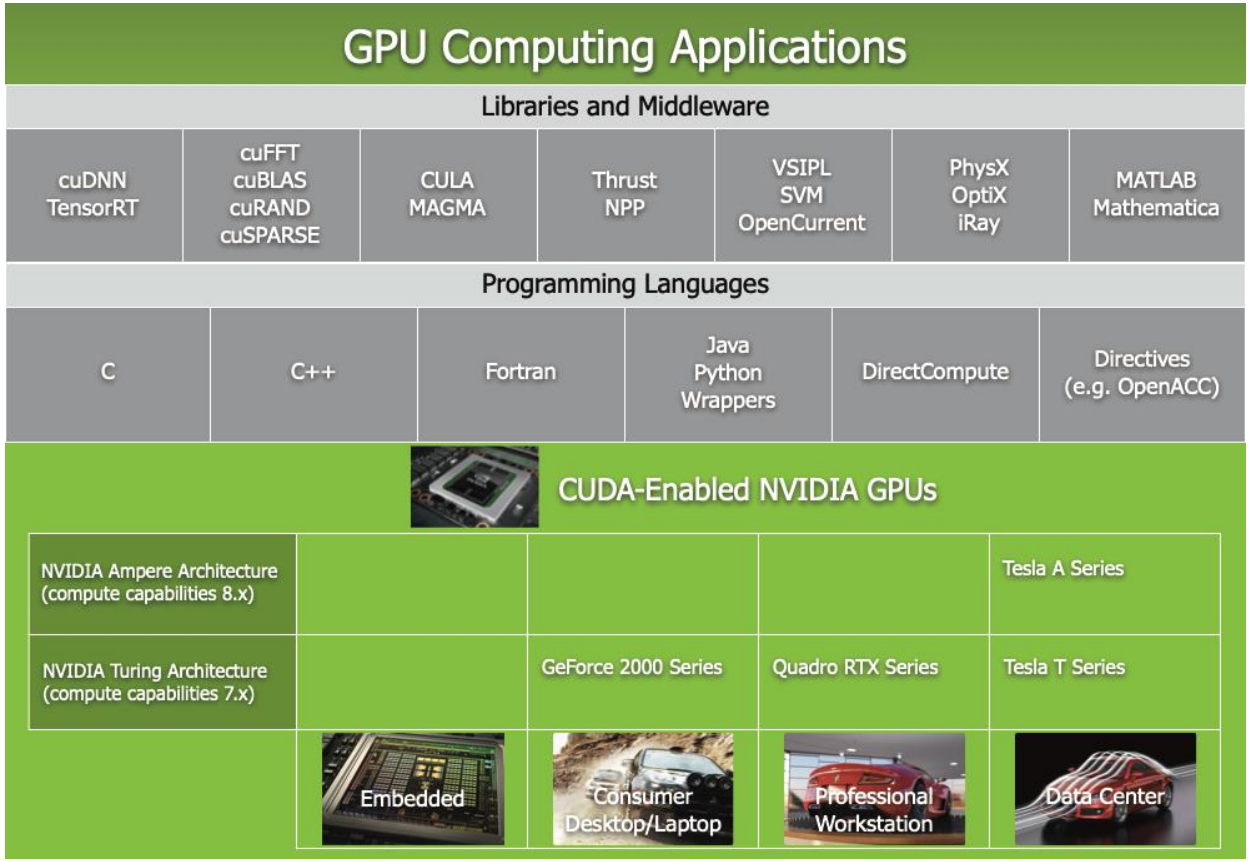
GPU Parallel Kernel
KernelA<<< nBlk, nTid >>>(args);

CPU Serial Code

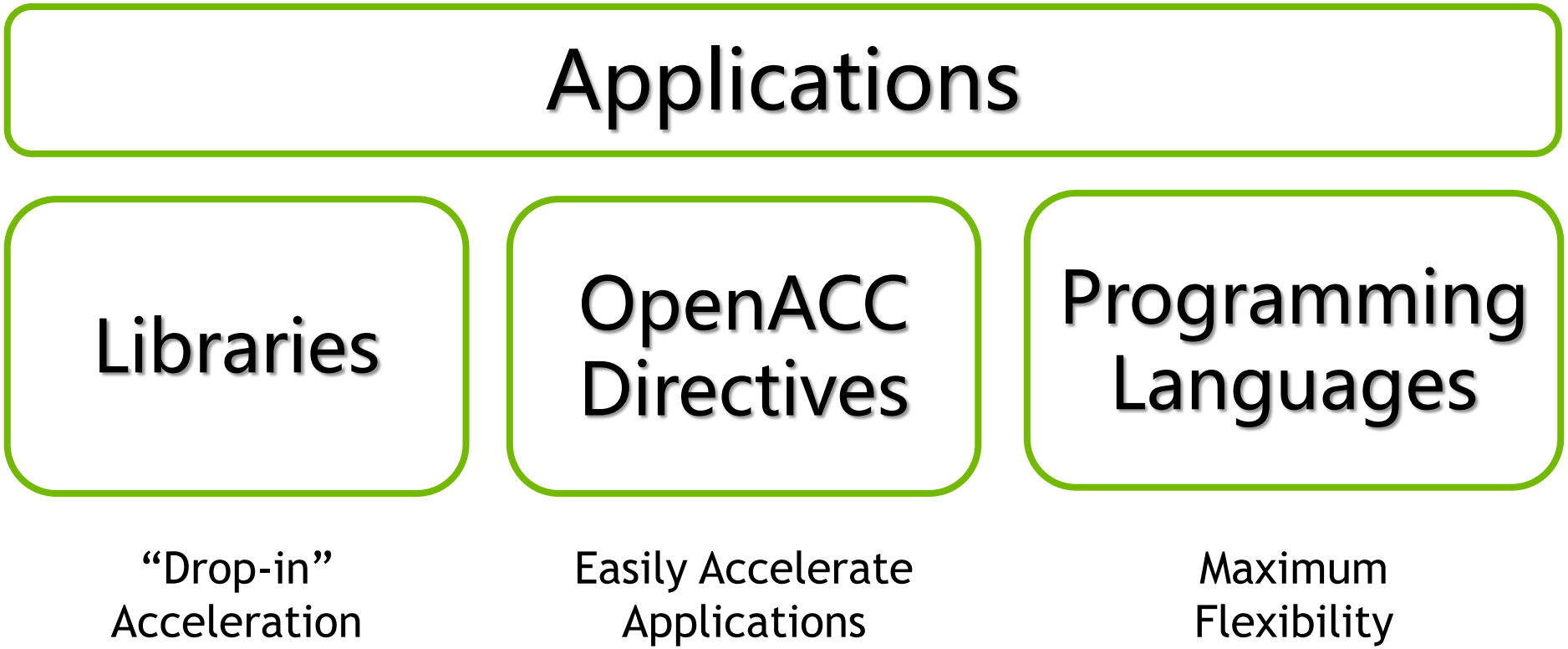
GPU Parallel Kernel
KernelB<<< nBlk, nTid >>>(args);



GUDA支持不同语言和编程接口



三种用GPU加速应用的方式

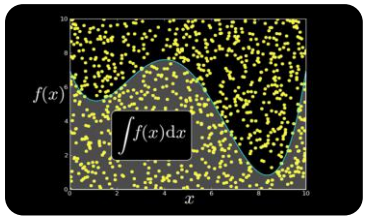


Libraries: Easy, High-Quality Acceleration

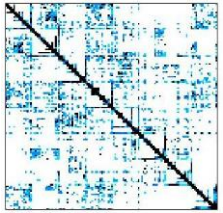
- Ease of use：使用库可实现GPU加速，而无需深入了解GPU编程技术
- "Drop-in"：许多GPU加速库都遵循标准API，因此只需最少的代码更改即可实现加速
- Quality：GPU加速库提供了应用中广泛用到的一些函数的高质量实现
- Performance：NVDIA库都是经过专家调优的

一些GPU加速库


NVIDIA cuBLAS



NVIDIA cuRAND


NVIDIA cuSPARSE


NVIDIA NPP

GPU VSIPL
Vector Signal Image Processing

CULA tools
GPU Accelerated Linear Algebra

MAGMA
ICL UT
Matrix Algebra on GPU and Multicore
open source initiative

NVIDIA cuFFT

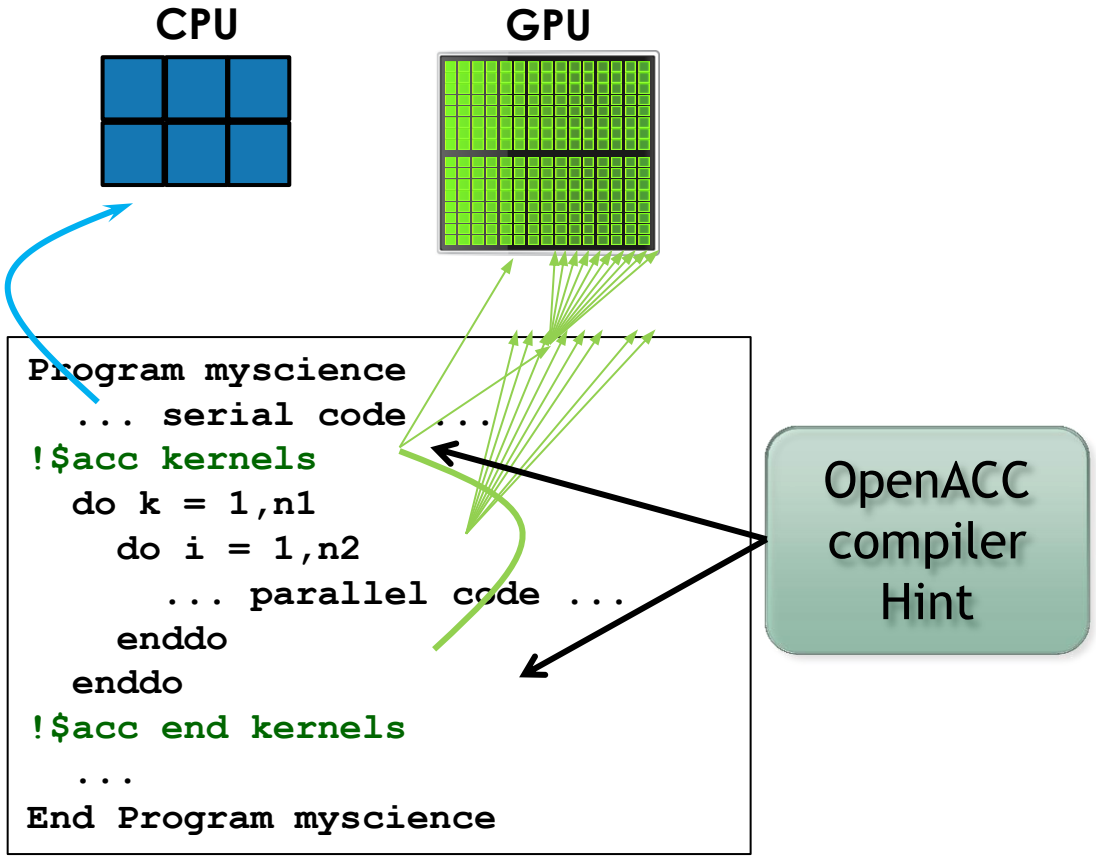
ROGUE WAVE SOFTWARE
IMSL Library


ArrayFire Matrix Computations

CUSP
Sparse Linear Algebra
open source initiative

Thrust
C++ STL Features for CUDA
open source initiative

OpenACC Directives



原始 Fortran 或 C 代码

Simple Compiler hints

Compiler Parallelizes code

Works on many-core GPUs & multicore CPUs

OpenACC：已成为GPU Directives的工业标准

- Easy：编译指导指令是加速计算密集型应用程序的简便途径
- Open：OpenACC是一个开放的标准，可移植性、扩展性好，开源社区活跃
- Powerful： OpenACC提供了丰富的功能，能实现对GPU的全面访问

OpenACC®
DIRECTIVES FOR ACCELERATORS

GPU编程语言

Numerical analytics	MATLAB, Mathematica, LabVIEW
Fortran	OpenACC, CUDA Fortran
C	OpenACC, CUDA C
C++	Thrust, CUDA C++
Python	PyCUDA, Copperhead
F#	Alea.cuBase

相关网络资源

- Download CUDA Toolkit & SDK:

  www.nvidia.com/getcuda

- Programming Guide/Best Practices:

  docs.nvidia.com

- Questions:
  - NVIDIA Developer forums: forums.developer.nvidia.com
  - Search or ask on: www.stackoverflow.com/tags/cuda
- General: developer.nvidia.com/cuda-toolkit

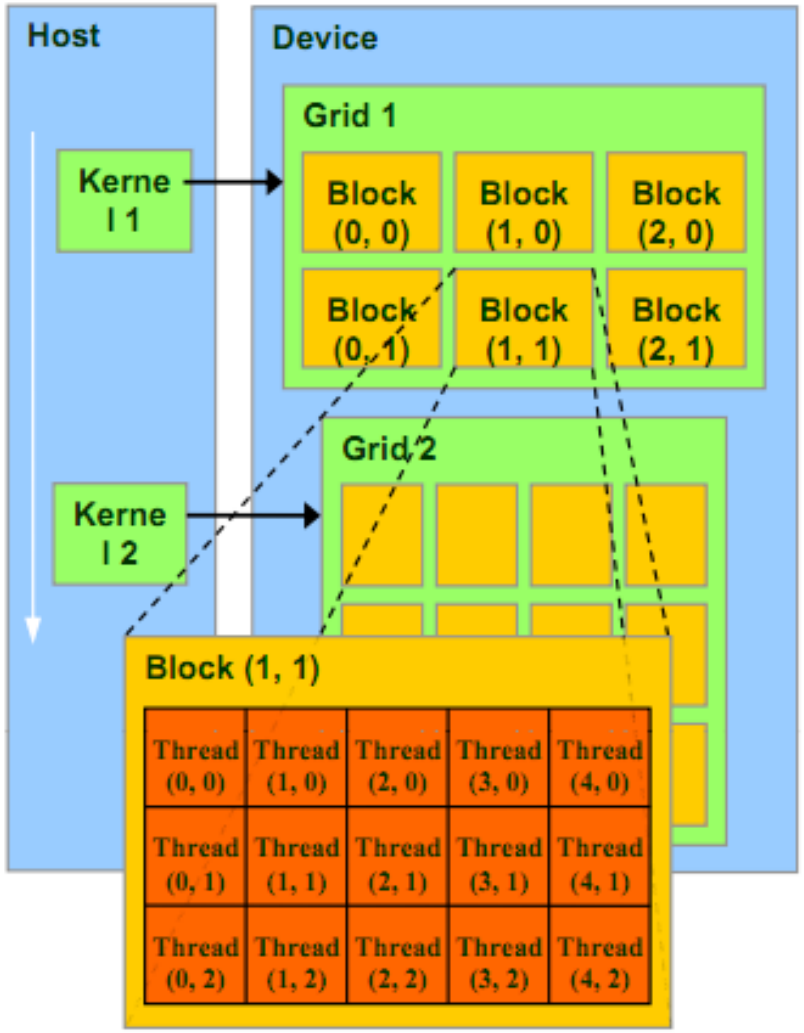
2 CUDA编程模型

CUDA设备与线程

- 计算设备（**device**）
  - 作为CPU（**host**）的协处理器
  - 有独立的存储器（device memory）
  - 同时启动大量线程
- 计算密集部分/数据并行部分使用**kernel**函数实现
  - 通过调用kernel函数在设备端创建大量并行的轻量级线程
- GPU与CPU线程的区别
  - GPU的线程非常轻量，线程切换~1 cycle，而CPU需要~1000 cycle
  - GPU上的线程数足够多时才能有效利用GPU的计算能力

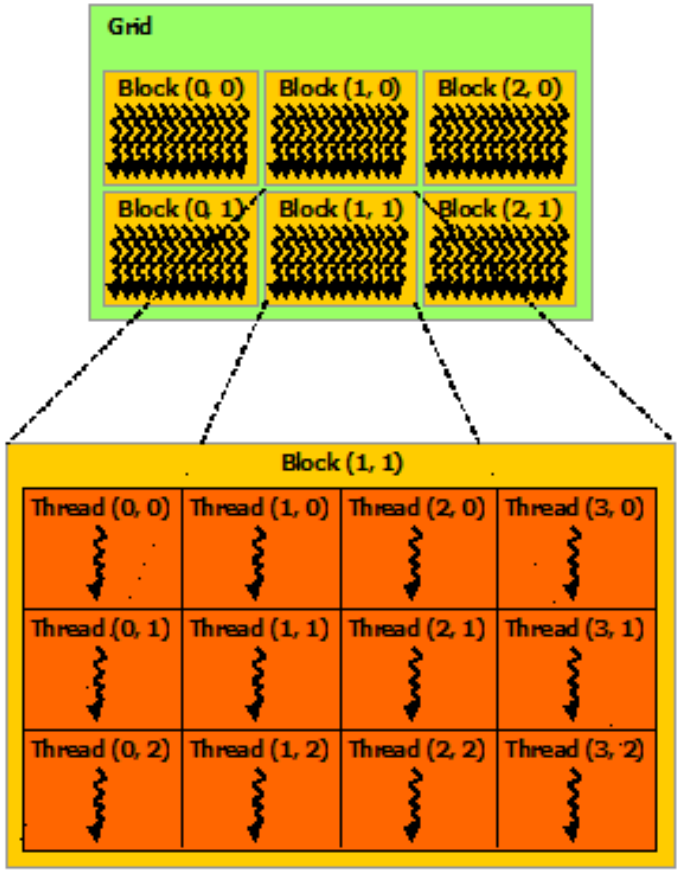
CUDA线程的组织结构

- Thread: 并行的基本单位
- Block: 互相合作的一组线程
  - 以1维、2维或3维组织
  - 允许彼此同步
  - 通过快速共享内存交换数据
  - 一个Grid里各Block线程数相同
  - 最多包含512个线程
- Grid:一维或多维线程块(block)
  - 以1维、2维或3维组织
  - 共享全局内存

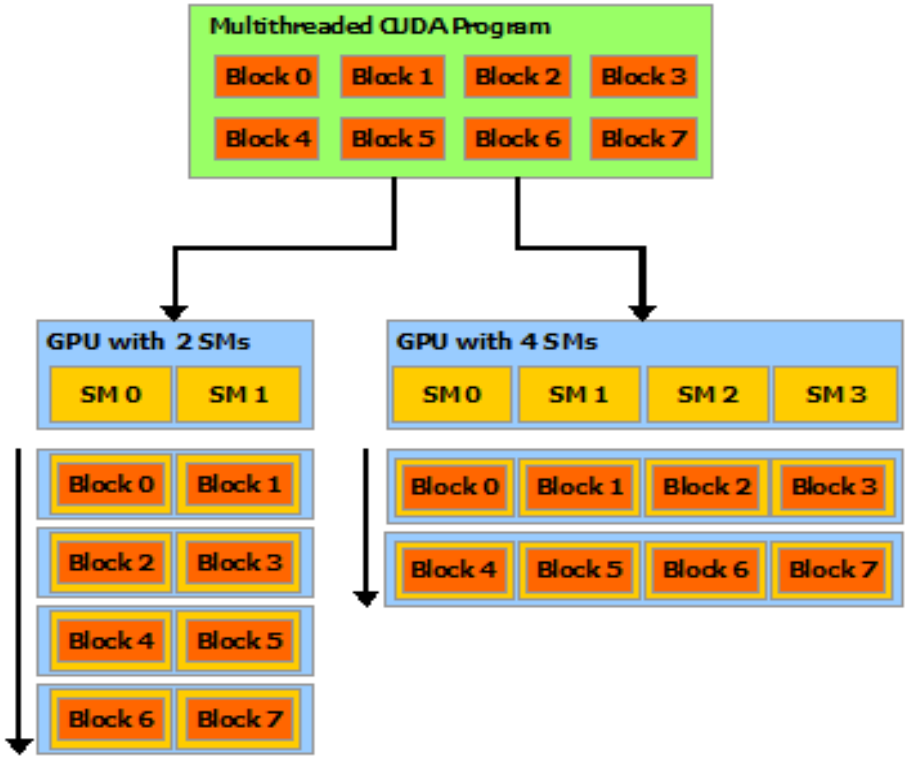


CUDA线程的组织结构

- Blocks 和 Threads 具有**3**维索引
  - **blockIdx, threadIdx**
  - **gridDim, blockDim**
- 一个Block里的线程ID计算：
  - 一维Block: threadIdx.x
  - 二维Block：threadIdx.x + threadIdx.y*blockDim.x
  - 三维Block：threadIdx.x + threadIdx.y*blockDim.x + threadIdx.z*blockDim.x*blockDim.y



- **Kernel: 在GPU上执行的核心程序**
  - **One kernel <-> one grid**



存储器模型与内存分配

R/W per-thread registers

1-cycle latency

R/W per-thread local memory

Slow – register spilling to global memory

R/W per-block shared memory

1-cycle latency

But bank conflicts may drag down

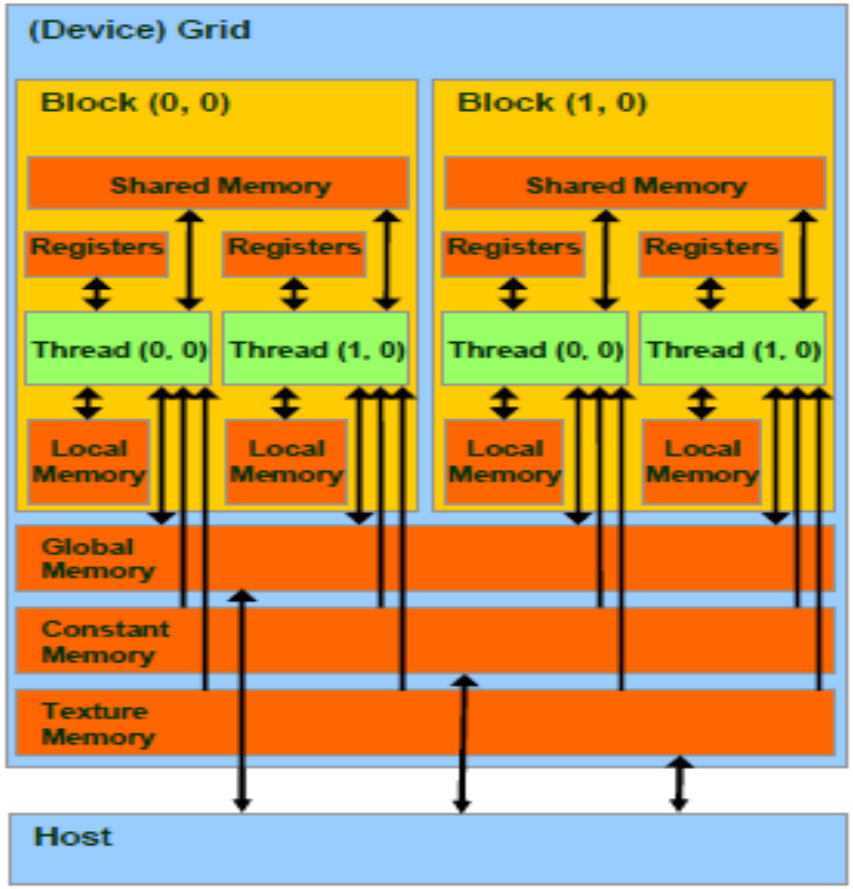
R/W per-grid global memory

~500-cycle latency

But coalescing accessing could hide latency

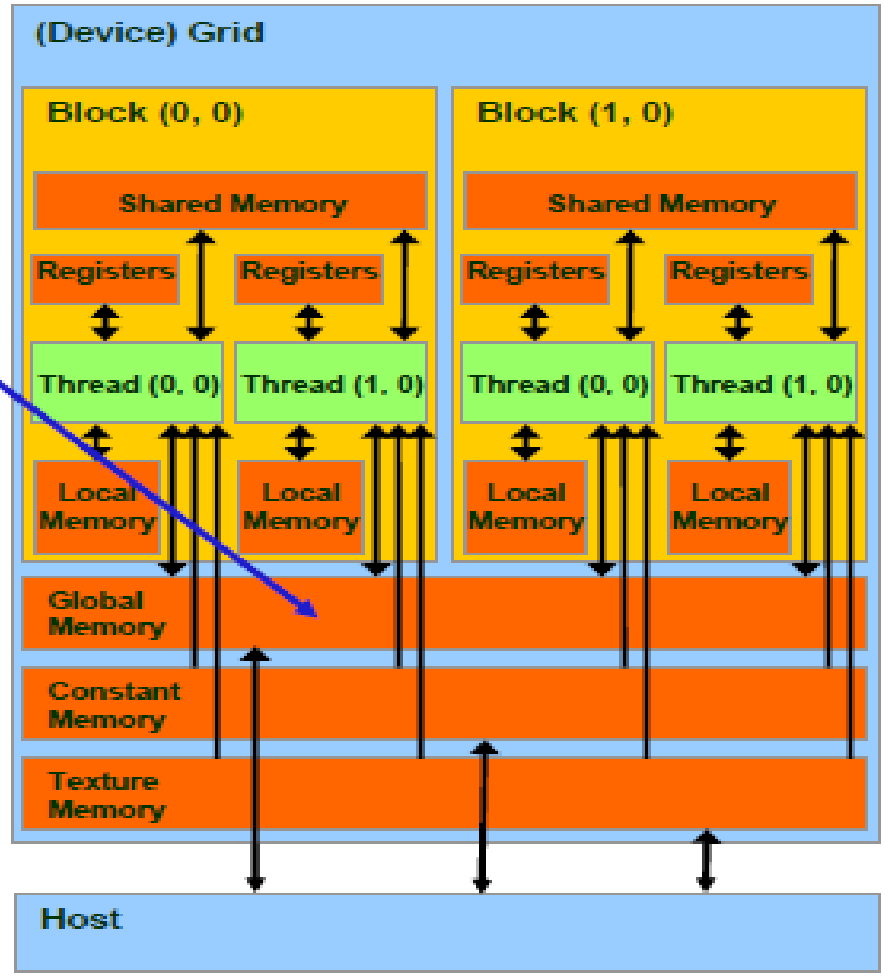
Read only per-grid constant and texture memories

~500-cycle latency But cached

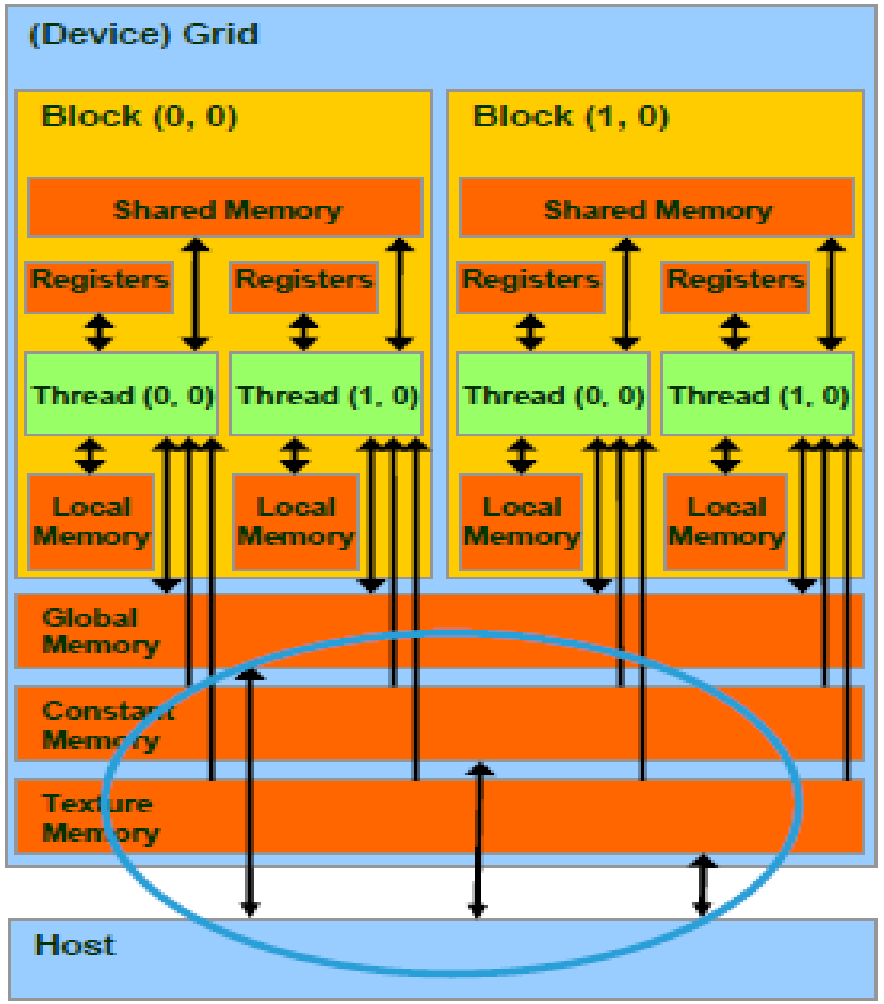


GPU Global Memory分配

- cudaMalloc()
  - 分配显存中的global memory
  - 两个参数
    - 对象数组指针
    - 数组尺寸
- cudaFree()
  - 释放显存中的global memory
  - 一个参数
    - 对象数组指针



- **Host - Device数据交换**
  - cudaMemcpy()
    - 在存储器直接传输数据
    - 四个参数
      - 目的对象数组指针
      - 源对象数组指针
      - 数组尺寸
      - 传输方向
        - Host到Host
        - Host到Device
        - Device到Host
        - Device到Device



CUDA 函数声明

	函数 执行处	函数 调用处
`__global__ void KernelFunc()`	**device**	**host**
`__host__ float HostFunc()`	**host**	**host**
`__device__ float DeviceFunc()`	**device**	**device**

- __global__ 定义一个kernel函数
  - 必须返回 void类型
- 如果没有标明前缀，那么函数默认为__host__
- __device__和__host__能够同时用

CUDA引入的变量修饰词

- __device__
  - 储存于GPU上的global memory空间
  - 和应用程序具有相同的生命期(lifetime)
  - 可被grid中所有线程存取，CPU代码通过runtime函数存取
- __constant__
  - 储存于GPU上的constant memory空间
  - 和应用程序具有相同的生命期
  - 可被grid中所有线程存取，CPU代码通过runtime函数存取
- __shared__
  - 储存于GPU上block内的共享存储器
  - 和block具有相同的生命期
  - 只能被block内的线程存取
- 无修饰（Local变量）
  - 储存于SM内的寄存器或local memory
  - 和thread具有相同的生命期
  - Thread私有

CUDA程序的编译

- 使用nvcc编译工具

  nvcc <filename>.cu [-o excutable]

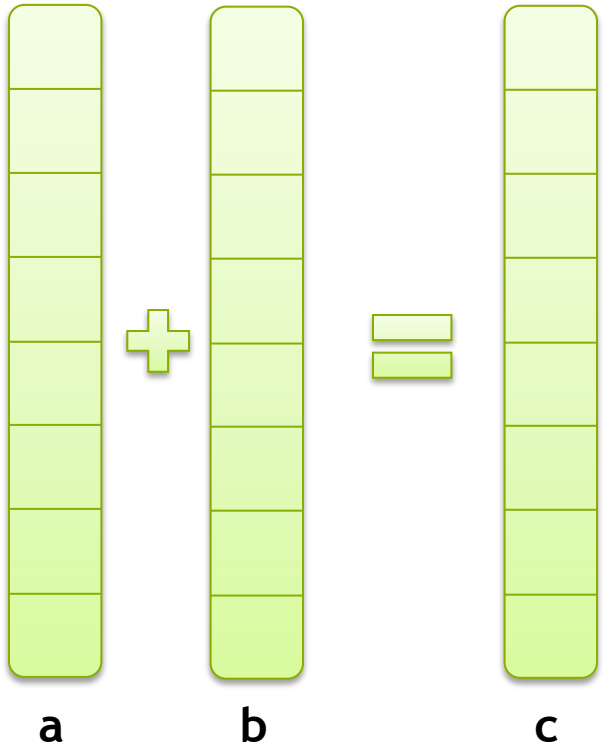
- 调试选项：-g（debug）、-deviceemu（CPU模拟GPU）
- nvcc 区分host 和 device 端的源代码
  - Device 函数 (e.g. mykernel()) 由NVIDIA 编译器处理
  - Host 函数 (e.g. main()) 由host端的标准默认编译器处理

    gcc, cl.exe

3 CUDA C语言编程

用GPU实现向量求和

- 先实现在GPU上计算两个整型数求和
- 再扩展到整型数组的并行求和



用GPU实现向量求和

- 先实现在GPU上计算两个整型数求和

```
__global__ void add(int *a, int *b, int *c) {
        *c = *a + *b;
}
```

- `add()` will execute on the device
- `add()` will be called from the host

用GPU实现向量求和

- 先实现在GPU上计算两个整型数求和

```
__global__ void add(int *a, int *b, int *c) {
        *c = *a + *b;
}
```

•add() 运行在device上，因此a, b 和c 应指向设备内存
•需要在GPU的内存中申请空间

用GPU实现向量求和

- 先实现在GPU上计算两个整型数求和

```
int main(void) {
        int a, b, c;                // host copies of a, b, c
        int *d_a, *d_b, *d_c;       // device copies of a, b, c
        int size = sizeof(int);

        // Allocate space for device copies of a, b, c
        cudaMalloc((void **)&d_a, size);
        cudaMalloc((void **)&d_b, size);
        cudaMalloc((void **)&d_c, size);

        // Setup input values
        a = 2;
        b = 7;
```

用GPU实现向量求和

- 先实现在GPU上计算两个整型数求和

```
    // Copy inputs to device
    cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);

    // Launch add() kernel on GPU
    add<<<1,1>>>(d_a, d_b, d_c);

    // Copy result back to host
    cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);

    // Cleanup
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

- 3.1 初识CUDA程序
- 3.2 Blocks
- 3.3 Threads
- 3.4 结合Blocks和Threads
- 3.5 线程协作 — 共享内存和同步
- 3.6 管理GPU设备

用GPU实现向量求和

- 如何在GPU上用大量线程并行计算？

```
add<<< 1, 1 >>>();
```

```
add<<< N, 1 >>>();
```

- GPU线程并行执行 `add()` 函数N次

用GPU实现向量求和

- 多个线程并行执行add()函数来实现向量求和

- 每个执行add() 的线程作为一个 block
  - 一组block作为一个grid
  - 每次add()的执行通过 `blockIdx.x` 得到当前线程的索引

```
__global__ void add(int *a, int *b, int *c) {
        c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```

- 通过 `blockIdx.x` 索引访问数组, 从而使得各个block线程计算不同的元素

3.2 Blocks

用GPU实现向量求和

```
__global__ void add(int *a, int *b, int *c) {
        c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```

- 在GPU上，这些Block的线程并行执行:

Block 0

```
c[0]  = a[0] + b[0];
```

Block 1

```
c[1]  = a[1] + b[1];
```

Block 2

```
c[2]  = a[2] + b[2];
```

Block 3

```
c[3]  = a[3] + b[3];
```

用GPU实现向量求和

```
#define N 512
int main(void) {
    int *a  *b  *c          // host copies of a, b, c
    int *d_a, *d_b, *d_c;  // device copies of a, b, c
    int size = N * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    // Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);
```

用GPU实现向量求和

```
    // Copy inputs to device
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

    // Launch add() kernel on GPU with N blocks
    add<<<N,1>>>(d_a, d_b, d_c);

    // Copy result back to host
    cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

    // Cleanup
    free(a); free(b); free(c);
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

回顾

- CUDA编程中的host与device
  - ***Host*** CPU
  - ***Device*** GPU
- 使用`__global__`声明一个函数为CUDA核函数
  - 核函数运行在Device端
  - 在Host端调用核函数
- 进行设备内存的管理：**`cudaMalloc()`、`cudaMemcpy()`、`cudaFree()`**
- 调用CUDA核函数
  - **`add<<<N,1>>>(...)`**
  - 使用`blockIdx.x`作为block的索引

用GPU实现向量求和

- 一个block可以包含多个thread
- 修改add( )函数，使用多个thread，而不是多个block进行并行计算

用GPU实现向量求和

```
#define N 512
int main(void) {
    int *a, *b, *c;                 // host copies of a, b, c
    int *d_a, *d_b, *d_c;           // device copies of a, b, c
    int size = N * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    // Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);
```

用GPU实现向量求和

```
    // Copy inputs to device
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

    // Launch add() kernel on GPU with N threads
    add<<<1,N>>>(d_a, d_b, d_c);

    // Copy result back to host
    cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

    // Cleanup
    free(a); free(b); free(c);
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

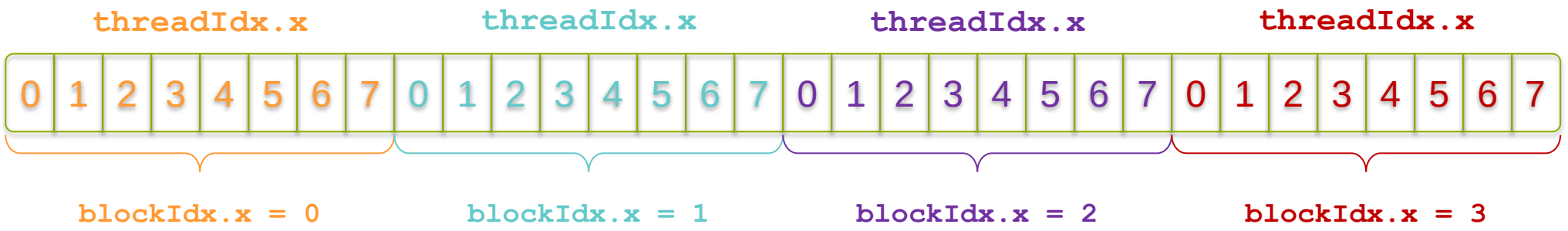
- 3.1 初识CUDA程序
- 3.2 Blocks
- 3.3 Threads
- 3.4 结合Blocks和Threads
- 3.5 线程协作 — 共享内存和同步
- 3.6 管理GPU设备

用GPU实现向量求和

- 前面的两种实现：
  - N个block，每个block一个thread
  - 一个block，其中N个thread
- 接下来使用多个block和多个thread

Blocks和Threads的索引

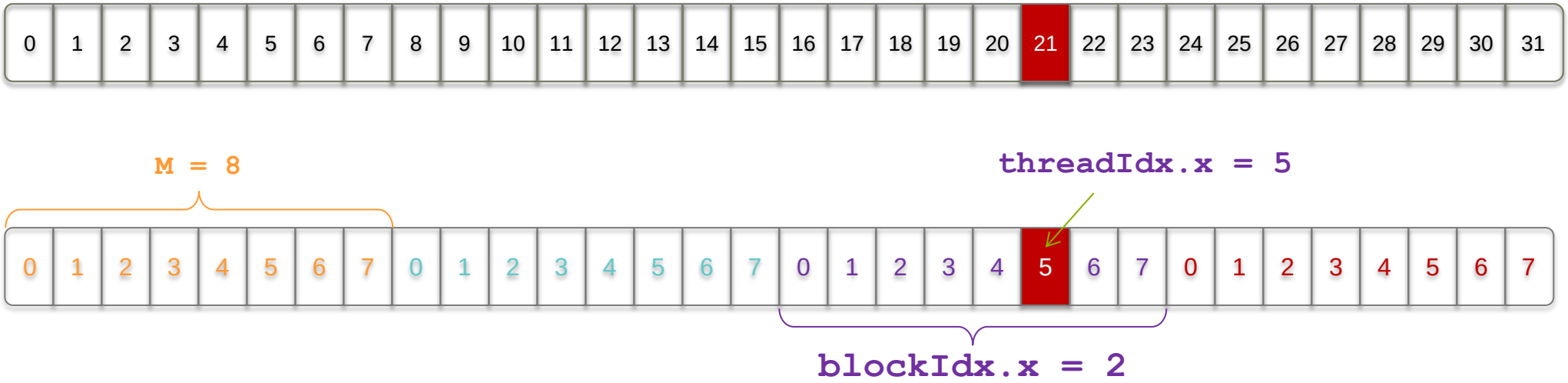
- 一维数组上block和thead的索引(8 threads/block)



- 线程索引的计算（M threads/block ）：

```
int index = threadIdx.x + blockIdx.x * M;
```

Blocks和Threads的索引




```
int index = threadIdx.x + blockIdx.x * M;
          =       5     +      2       * 8;
          = 21;
```

用GPU实现向量求和

- 内置变量**blockDim.x**表示每个block的线程数量
- 修改kernel函数add()

```
__global__ void add(int *a, int *b, int *c) {
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    c[index] = a[index] + b[index];
}
```

- 修改main函数

用GPU实现向量求和

```
#define N (2048*2048)
#define THREADS_PER_BLOCK 512
int main(void) {
    int *a, *b, *c;                 // host copies of a, b, c
    int *d_a, *d_b, *d_c;           // device copies of a, b, c
    int size = N * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    // Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);
```

用GPU实现向量求和

```
    // Copy inputs to device
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

    // Launch add() kernel on GPU
    add<<<N/THREADS_PER_BLOCK,THREADS_PER_BLOCK>>>(d_a, d_b, d_c);

    // Copy result back to host
    cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

    // Cleanup
    free(a); free(b); free(c);
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

用GPU实现向量求和

- 如果向量中元素个数不是blockDim.x的整数倍怎么办？
- 避免数组访问越界

```
__global__ void add(int *a, int *b, int *c, int n) {
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    if (index < n)
        c[index] = a[index] + b[index];
}
```

- 更改核函数调用

```
add<<<(N + M-1) / M,M>>>(d_a, d_b, d_c, N);
```

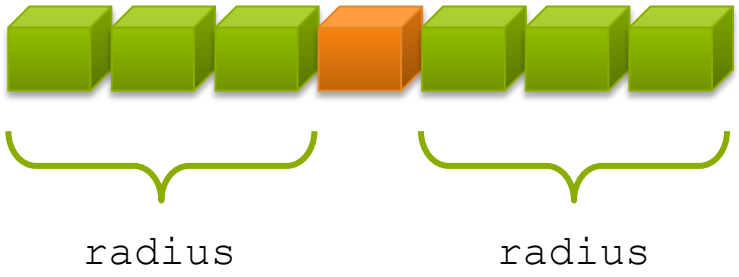
3.4 结合Blocks和Threads

- **为什么要使用Threads？**
  - grid ⇨ blocks ⇨ threads
  - grid ⇔ kernel （应用）
  - blocks + threads 提供了更灵活的线程组织和管理方式
  - threads具备blocks没有的同步、通信机制

- 3.1 初识CUDA程序
- 3.2 Blocks
- 3.3 Threads
- 3.4 结合Blocks和Threads
- 3.5 线程协作 — 共享内存和同步
- 3.6 管理GPU设备

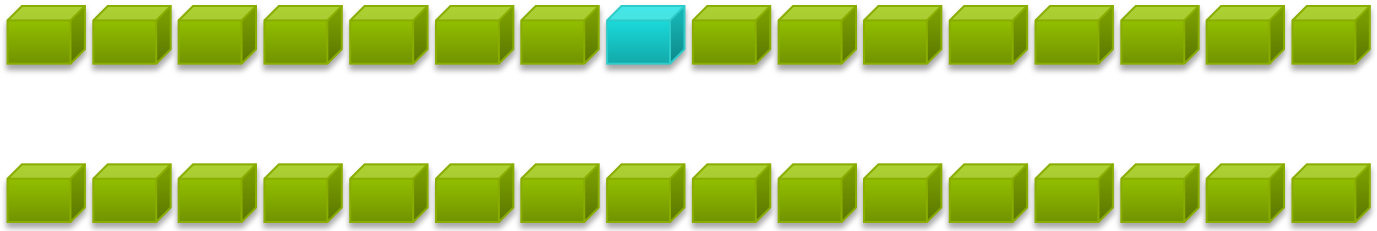
1D Stencil

- 在一维数组上应用一维蒙板运算
  - 每个输出元素值是当前位置输入元素值与相邻半径范围内元素的和



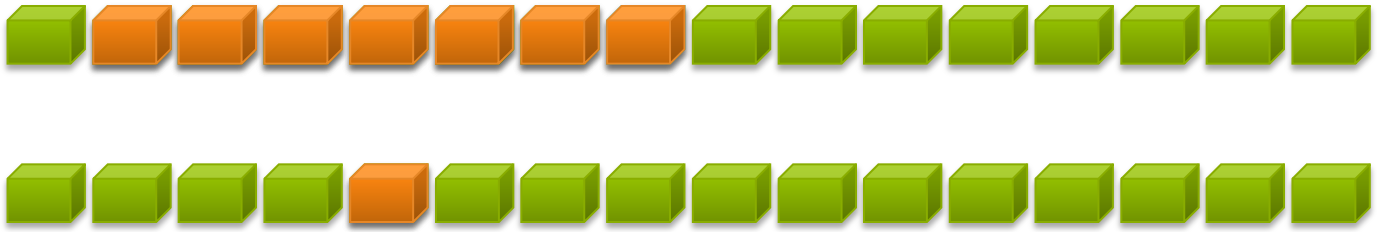
1D Stencil

- 每个block包含blockDim.x线程，每个线程对应一个输出元素
- 输入元素会被多个线程读取多次



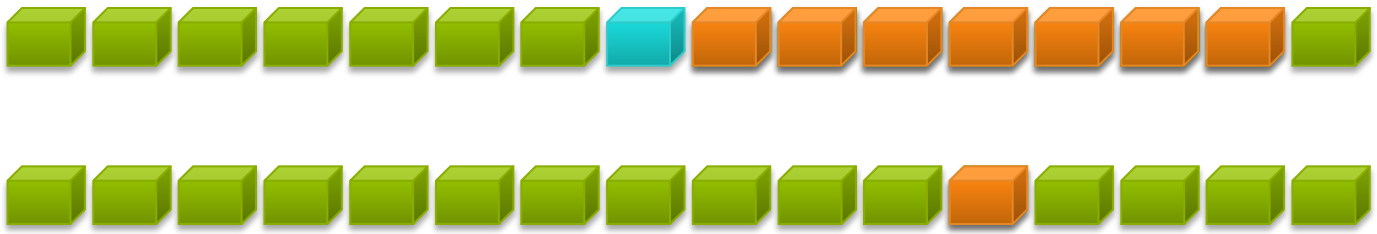
1D Stencil

- 每个block包含blockDim.x线程，每个线程对应一个输出元素
- 输入元素会被多个线程读取多次



1D Stencil

- 每个block包含blockDim.x线程，每个线程对应一个输出元素
- 输入元素会被多个线程读取多次

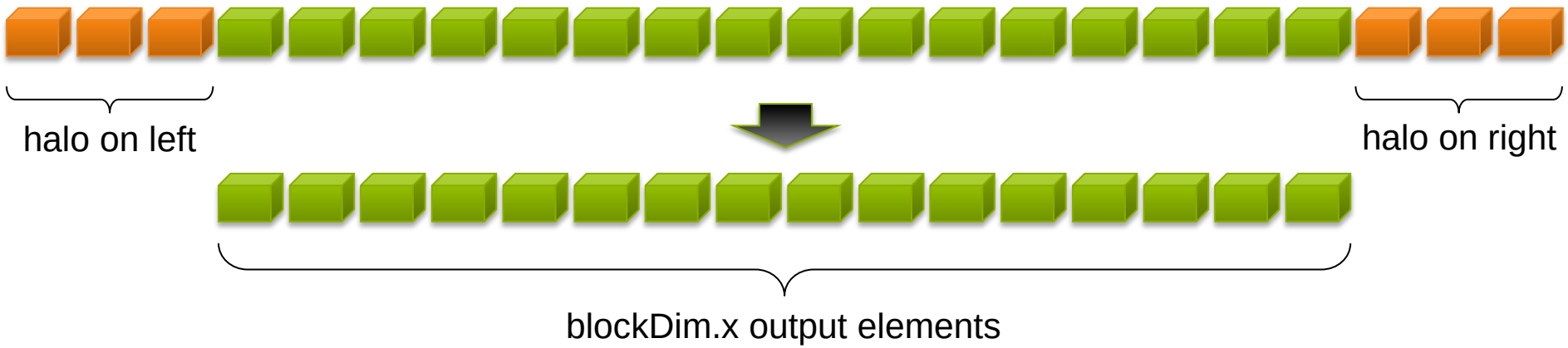


1D Stencil

- 在线程间共享数据
  - 同一个block中的线程可通过shared memory共享数据
  - 使用__shared__声明，每个block分配一份共享内存
  - 一个block的共享数据不能被另一个block的线程访问

1D Stencil

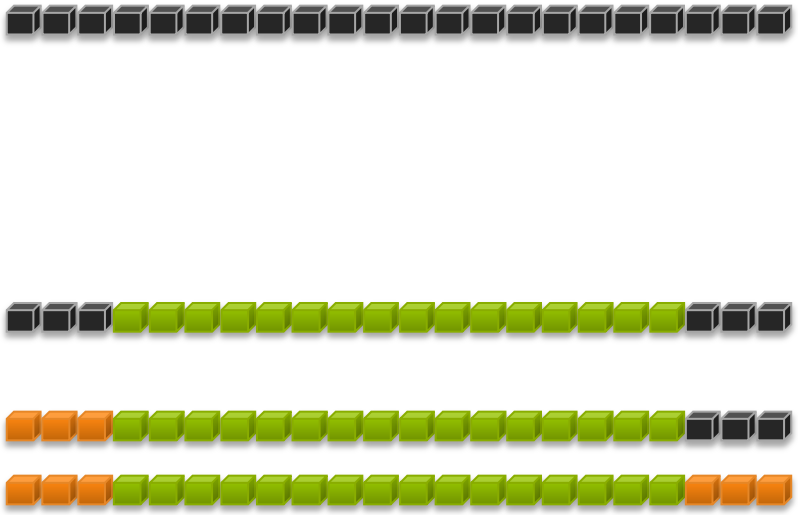
- 在线程间共享数据
  - 从global memory读取(blockDim.x + 2 * radius)个数据到shared memory
  - 计算 blockDim.x 个输出数据
  - 将 blockDim.x 个输出数据写入 global memory
  - 每个 block 需要读入一些边界元素（a halo of radius elements ）



1D Stencil Kernel

```
__global__ void stencil_1d(int *in, int *out) {
  __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
  int gindex = threadIdx.x + blockIdx.x * blockDim.x;
  int lindex = threadIdx.x + RADIUS;

  // Read input elements into shared memory
  temp[lindex] = in[gindex];
  if (threadIdx.x < RADIUS) {
    temp[lindex - RADIUS] = in[gindex - RADIUS];
    temp[lindex + BLOCK_SIZE] =
      in[gindex + BLOCK_SIZE];
  }
```



1D Stencil Kernel

```
  // Apply the stencil
  int result = 0;
  for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
    result += temp[lindex + offset];

  // Store the result
  out[gindex] = result;
}
```

1D Stencil Kernel

- 假设线程18在线程0将边界元素读入共享内存前开始计算

数据竞争

```
temp[lindex] = in[gindex];          Store at temp[18]
if (threadIdx.x < RADIUS) {
  temp[lindex - RADIUS = in[gindex - RADIUS];          Skipped, threadIdx > RADIUS
  temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
}

int result = 0;
result += temp[lindex + 1];          Load from temp[19]
```

1D Stencil Kernel

- 调用`void __syncthreads();`同步一个block里的线程
- 用于避免RAW / WAR / WAW 数据竞争
- __syncthreads()只会同步同一个块中的线程
- 同一个块中的线程必须都能到达同步点
  - 如果在条件分支中调用__syncthreads() ，应保证同一个块的线程通过相同的分支

1D Stencil Kernel

```
__global__ void stencil_1d(int *in, int *out) {
    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
    int gindex = threadIdx.x + blockIdx.x * blockDim.x;
    int lindex = threadIdx.x + radius;

    // Read input elements into shared memory
    temp[lindex] = in[gindex];
    if (threadIdx.x < RADIUS) {
        temp[lindex - RADIUS] = in[gindex - RADIUS];
        temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
    }

    // Synchronize (ensure all the data is available)
    __syncthreads();
```

1D Stencil Kernel

```
    // Apply the stencil
    int result = 0;
    for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
        result += temp[lindex + offset];

    // Store the result
    out[gindex] = result;
}
```

- 3.1 初识CUDA程序
- 3.2 Blocks
- 3.3 Threads
- 3.4 结合Blocks和Threads
- 3.5 线程协作 — 共享内存和同步
- 3.6 管理GPU设备

- **Kernel的调用是异步的**
  - Host端调用kernel函数后会立刻返回
- **CPU在使用kernel函数的计算结果前需要同步**

`cudaMemcpy()`	同步拷贝，阻塞CPU的执行，直到拷贝完成 只有当之前的CUDA核函数调用都完成后拷贝才开始
`cudaMemcpyAsync()`	异步拷贝，不阻塞CPU的执行
`cudaDeviceSynchronize()`	阻塞CPU的执行，直到之前所有的CUDA核函数调用都完成

错误报告

- 所有CUDA API 调用都会返回一个error code (cudaError_t)
  - API调用本身的错误
  - 之前异步调用（如调用kernel函数）中的错误
- 获取最近的错误代码：

```
cudaError_t cudaGetLastError(void)
```

- 得到错误描述：

```
char *cudaGetErrorString(cudaError_t)

printf("%s\n", cudaGetErrorString(cudaGetLastError()));
```

3.6 管理GPU设备

- 查询和选择GPU设备

```
cudaGetDeviceCount(int *count)
cudaSetDevice(int device)
cudaGetDevice(int *device)
cudaGetDeviceProperties(cudaDeviceProp *prop, int device)
```

- 多个CPU线程可以共享一个设备
- 一个CPU线程能够管理多个设备

`cudaSetDevice(i)` 选择当前设备

`cudaMemcpy(...)` 设备间的拷贝


北京理工大學
BEIJING INSTITUTE OF TECHNOLOGY
谢谢!
德以明理 学以精工