
CUDA C/C++ 编程介绍
chandlerz@nvidia.com 周国峰
Wuhan University 2017/10/13
NVIDIA

CUDA?

- CUDA (Compute Unified Device Architecture)
  - 使用 GPU 大量计算单元和高带宽实现通用计算的计算平台
  - 提出了对硬件和软件要求
    - 支持 CUDA 的GPU：https://developer.nvidia.com/cuda-gpus
    - CUDA: https://developer.nvidia.com/cuda-downloads
- CUDA C/C++
  - 基于标准的 C/C++
  - CUDA C/C++ 是对它的一个扩展，实现异构计算
- 主要介绍 CUDA C/C++ 编程

基本概念/术语

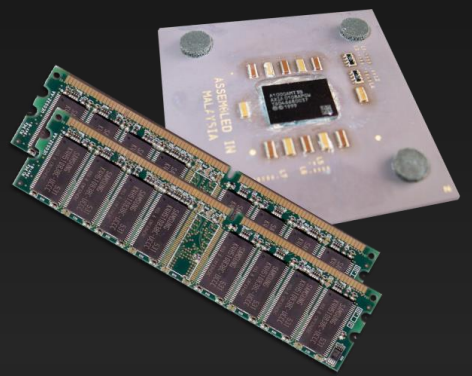
- 异构计算
- 线程块
- 线程
- 索引
- 内存组织
- 线程同步
- 异步执行

内容框架

- GPU 硬件架构
- 一个简单的例子：Hello, World!
- 一个实际的例子：矢量点乘
- 1D Stencil 计算
- 总结

Heterogeneous Computing (异构计算)

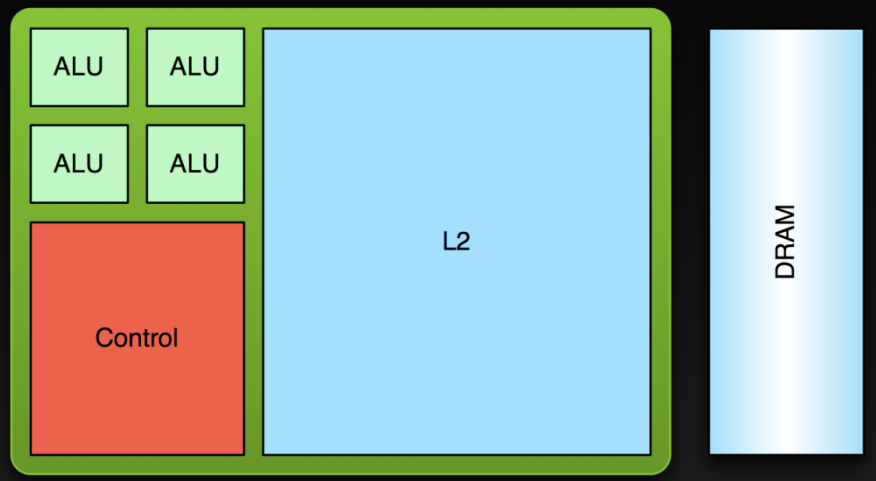
- 术语:
  - *Host* (主机) CPU 和它的内存 (host memory)
  - *Device* (设备) GPU 和它的内存 (device memory)



Host

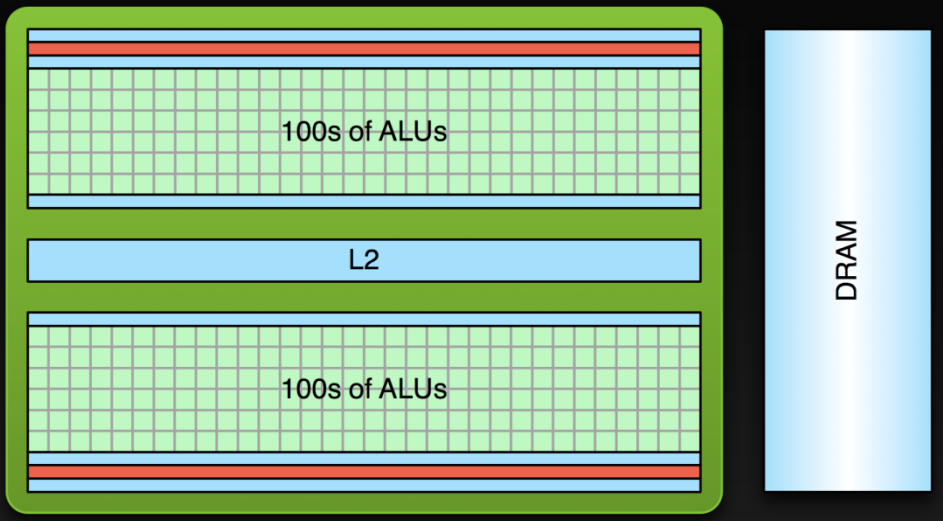
Device

低延迟 VS 高吞吐量



CPU

- 通过复杂的缓存体系结构，减小指令延迟
- 大量的晶体管处理逻辑任务



GPU

- 集成大量计算单元，获得高吞吐量
- 更多的晶体管用于数学计算

Heterogeneous Computing

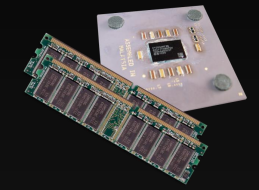
```
#include <iostream>
#include <algorithm>

using namespace std;

#define N          1024
#define RADIUS     3
#define BLOCK_SIZE 16

__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();

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

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

void fill_ints(int *x, int n) {
        fill_n(x, n, 1);
}

int main(void) {
        int *in, *out;          // host copies of a, b, c
        int *d_in, *d_out;      // device copies of a, b, c
        int size = (N + 2*RADIUS) * sizeof(int);

        // Alloc space for host copies and setup values
        in  = (int *)malloc(size); fill_ints(in,  N + 2*RADIUS);
        out = (int *)malloc(size); fill_ints(out, N + 2*RADIUS);

        // Alloc space for device copies
        cudaMalloc((void **)&d_in,  size);
        cudaMalloc((void **)&d_out, size);

        // Copy to device
        cudaMemcpy(d_in,  in,  size, cudaMemcpyHostToDevice);
        cudaMemcpy(d_out, out, size, cudaMemcpyHostToDevice);

        // Launch stencil_1d() kernel on GPU
        stencil_1d<<<N/BLOCK_SIZE,BLOCK_SIZE>>>(d_in + RADIUS, d_out + RADIUS);

        // Copy result back to host
        cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);

        // Cleanup
        free(in); free(out);
        cudaFree(d_in); cudaFree(d_out);
        return 0;
}
```

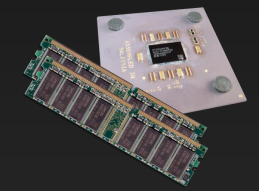
parallel fn

serial code

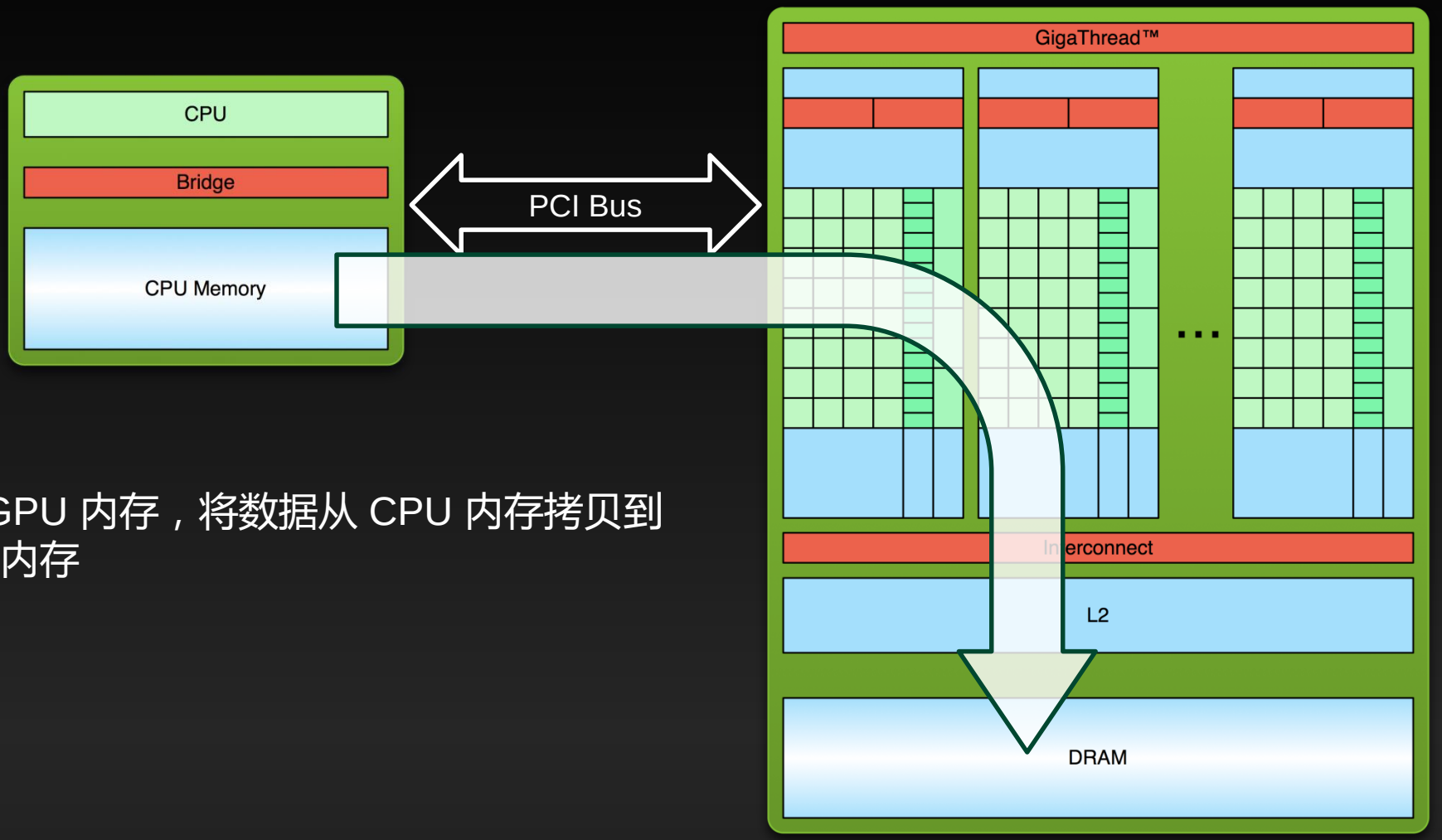
parallel code

serial code



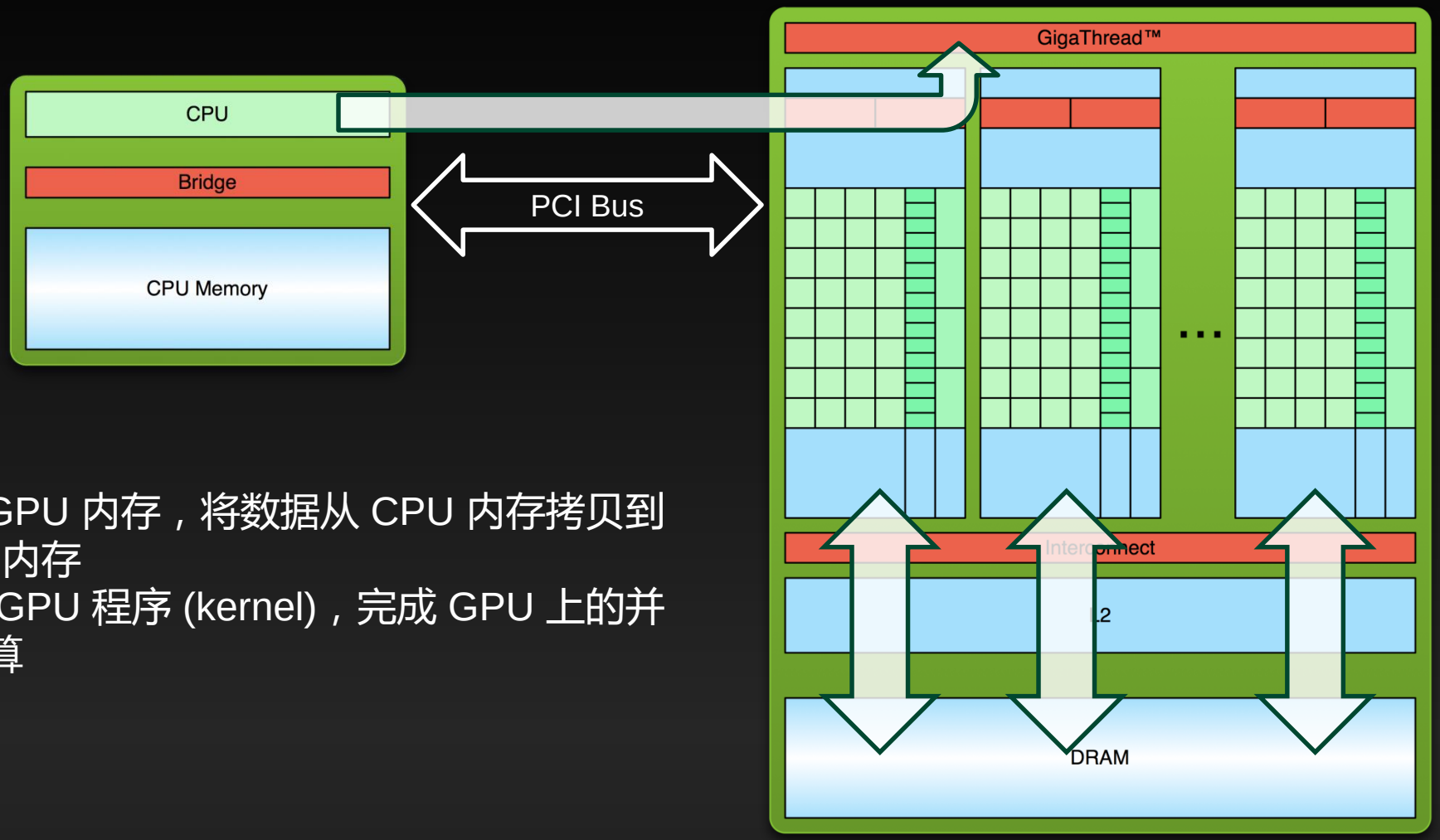


CUDA 编程的“三部曲”



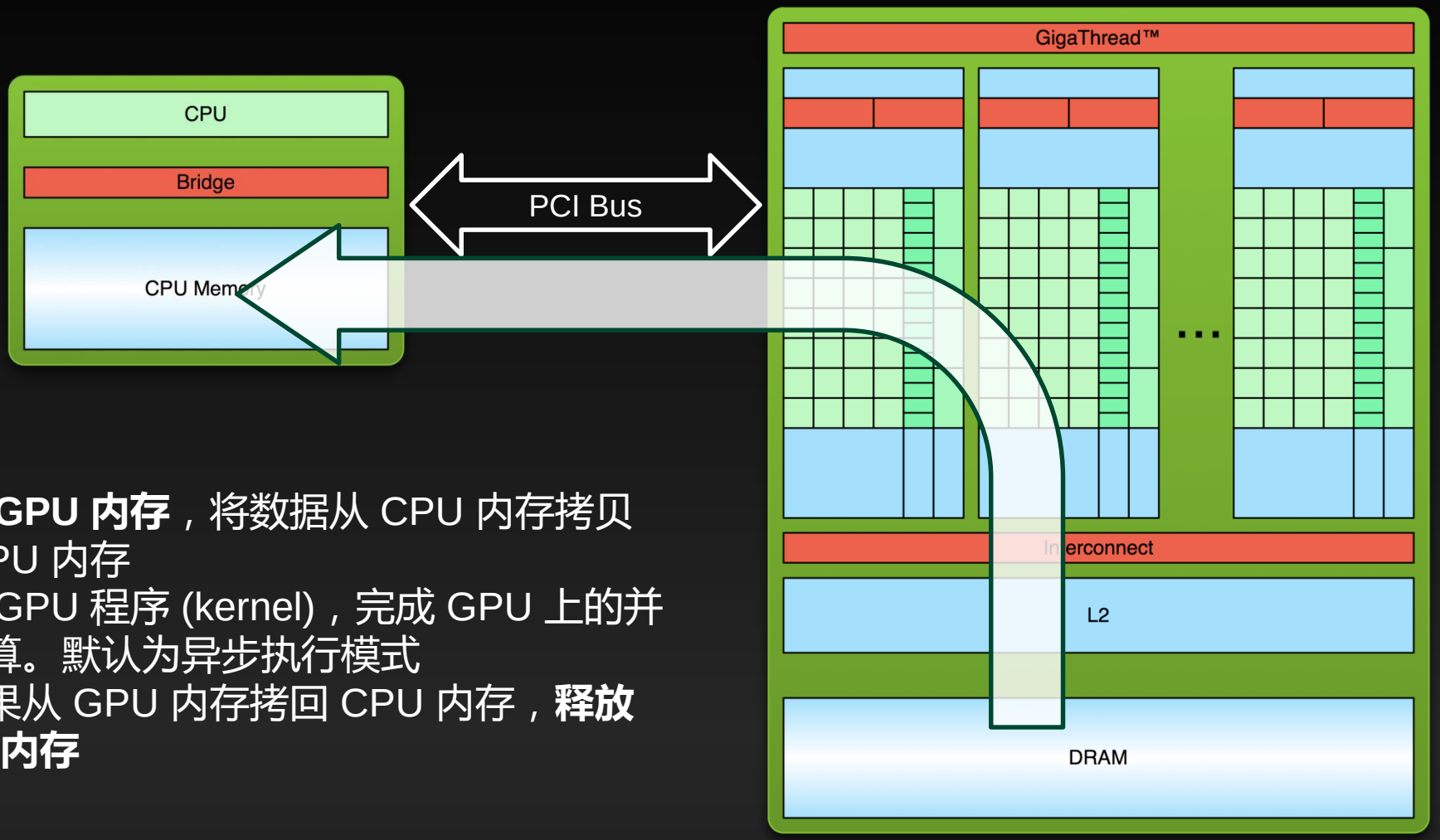
1. 申请GPU 内存，将数据从 CPU 内存拷贝到 GPU 内存

CUDA 编程的“三部曲”



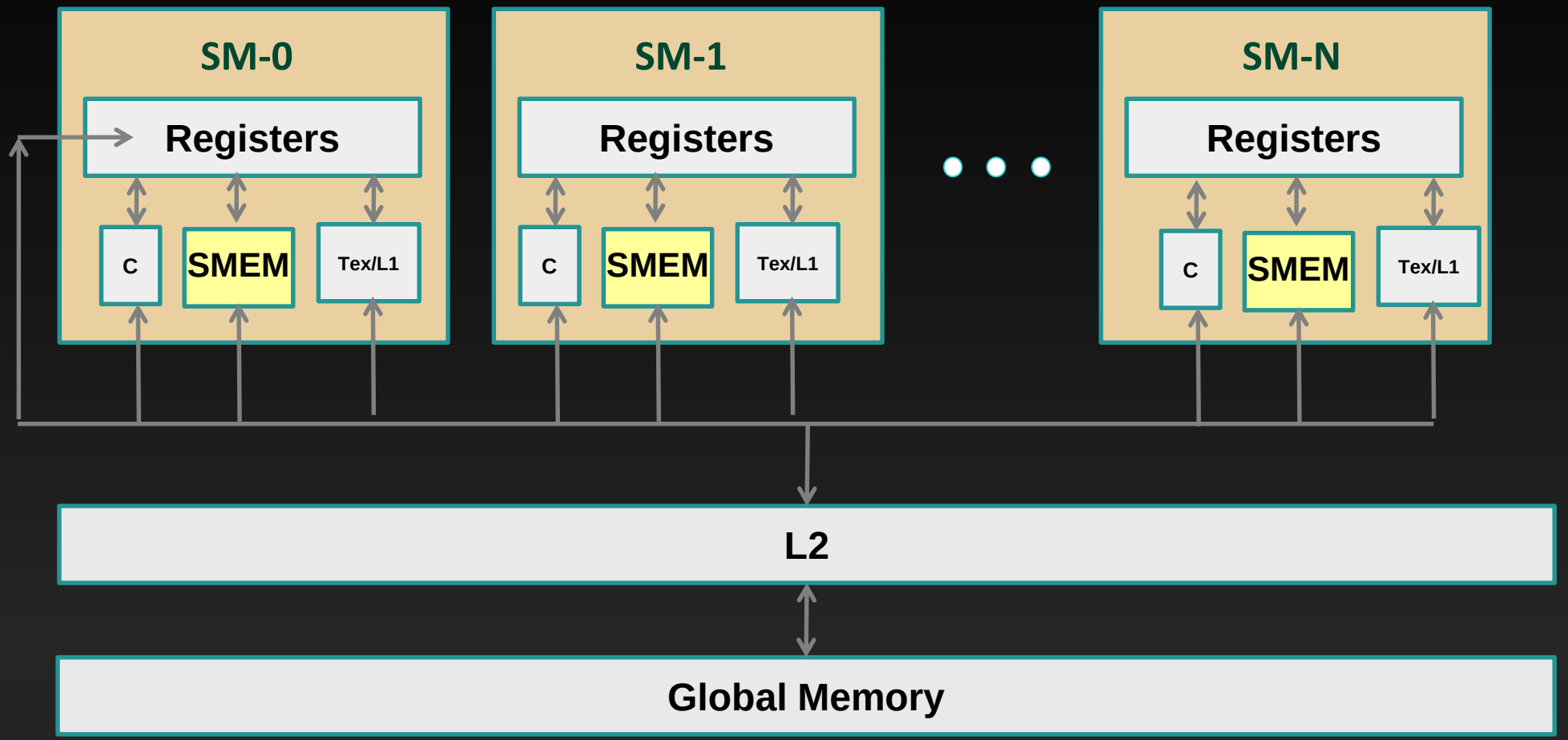
1. 申请GPU 内存，将数据从 CPU 内存拷贝到 GPU 内存
2. 加载 GPU 程序 (kernel)，完成 GPU 上的并行计算

CUDA 编程的“三部曲”



1. **申请 GPU 内存**，将数据从 CPU 内存拷贝到 GPU 内存
2. 加载 GPU 程序 (kernel)，完成 GPU 上的并行计算。默认为异步执行模式
3. 将结果从 GPU 内存拷回 CPU 内存，**释放 GPU 内存**

GPU 内存组织 (Pascal)



GPU 内存组织

- Global Memory
  - 位于片外；延迟~100 时钟周期；所有线程可见
- Shared Memory
  - 位于片上；延迟~10　时钟周期；以线程块为单位分配；线程块内所有线程共享
- Register
  - 位于片上；延迟~1　　时钟周期；以线程为单位分配；线程独享

- Global/Constant/Texture Memory
  - 都位于片外
  - 经过不同的 L1 cache L1/constant/Texture Cache
  - Conatant/Texture: Read-Only Memory

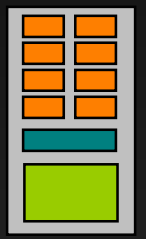
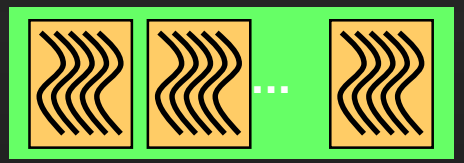
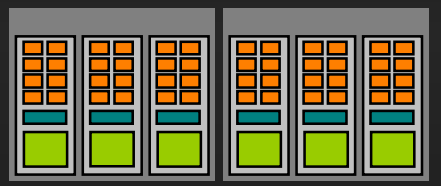
GPU P100 SM (Streaming Multiprocessor)

	GP100
SM/GPU	56
FP32 units	64
FP64 units	32
TensorCore	-
Register/SM	256 KB
Shared Memory/SM	64 KB
L1 Cache	24 KB

GPU V100 SM (Streaming Multiprocessor)

	GP100
SM/GPU	80
FP32 units	64
FP64 units	32
TensorCore	8
Register/SM	256 KB
Unified Shared Memory/ L1 Cache	128 KB

GPU CUDA 编程模型

Software	GPU	
 Thread	 CUDA Core	Threads are executed by cuda core
 Thread Block	 SM	Thread blocks are executed on SM
 Grid	 Device	A kernel is launched as a grid of thread blocks

第一个例子：HELLO WORLD!

Hello World! (C 代码)


```
int main(void) {
        printf("Hello World!\n");
        return 0;
}
```

- 标准的纯 C 代码，在 CPU 上执行

Hello World! (CUDA C 代码)


```
__global__ void mykernel(void) {
        printf("Hello World!\n");
}

int main(void) {
        mykernel<<<1,1>>>();
        return 0;
}
```

- 两个新的语法

__global__

```
__global__ void mykernel(void) {
        printf("Hello World!\n");
}
```

- CUDA C/C++ 关键字 __global__ 修饰的函数
  - 从主机端调用 (CC3.x, 也可以从设备端调用)
  - 在设备端执行
  - Pascal GPU (CC6.X, P100: CC6.0, P40: CC6.1)
- 其它的函数修饰符
  - __device__：设备端调用，设备端执行
  - __host__ ：主机端调用，主机端执行
  - __device__ 和 __host__ 可同时使用 (同时编译主机端和设备端两个版本)

<<<...>>>

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

- <<<X1, X2>>> 标记从主机端调用设备端函数以及相应的并行配置
  - 也叫做 “kernel launch” (kernel 启动)
  - X1 和 X2 分别为 kernel 的 grid (栅格) 和 block (线程块) 的设置
- 完成了在主机端调用、设备端执行一个 __global__ 函数过程

Hello World! (CUDA C 代码)

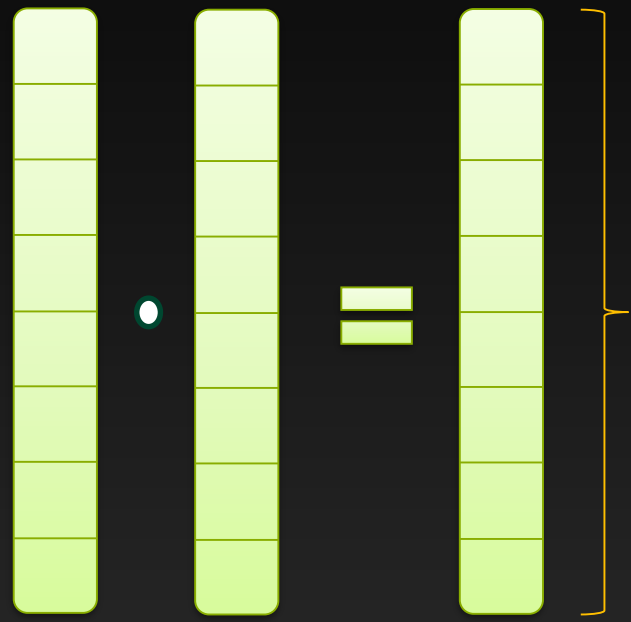

```
__global__ void mykernel(void) {
        printf("Hello World!\n");
}

int main(void) {
        mykernel<<<1,1>>>();
        return 0;
}
```

Output:

```
$ nvcc hello.cu
$ a.out
Hello World!
$
```

- mykernel 执行的任务很简单，“三部曲”只有“一部曲”

一个实际的例子：矢量点乘



归约求和 sum

一个实际的例子：矢量点乘 (只有一个分量)

- GPU 上实现两个数相乘

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

- `__global__` 修饰的 mul 函数
  - `mul()` 在主机端调用
  - `mul()` 在设备端执行

一个实际的例子：矢量点乘 (只有一个分量)

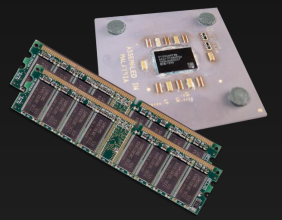
- a, b, c 都是指针

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

- mul() 在设备端执行，因此 a, b 和 c 必须指向设备内存
- 因此，事先需要在设备端分配内存

内存管理

- 主机内存和设备内存位于不同的物理空间
  - ***设备*** 指针只能指向 ***设备*** 内存
    - 可以在设备与主机之间传递
    - 不能在***主机***端引用 (不能取地址里的内容)
  - ***主机*** 指针只能指向 ***主机*** 内存
    - 可以在主机与设备之间传递
    - 不能在***设备***端引用 (不能取地址里的内容)



- 内存管理的 CUDA API
  - `cudaMalloc()`, `cudaFree()`, `cudaMemcpy()`
  - 与主机端的 `malloc()`, `free()`, `memcpy()` 类似

矢量点乘 (只有一个分量)：kernel

- 回到 `mul()` kernel

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

- 设备端内存的申请在 main() 完成...

矢量点乘 (只有一个分量): main


```
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);         // in bytes
```

Step 1: 申请 GPU 内存，CPU 到 GPU 数据拷贝

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

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

矢量点乘 (只有一个分量): main

Step 2: 启动 kernel，完成设备端代码的计算

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

Step 3: GPU 到 CPU 的数据拷贝，释放 GPU 内存

```
// Copy result back to host
cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);
// Cleanup
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
return 0;
}
```

并行执行

矢量点乘 (多分量, N=512)：kernel

- GPU 的计算适合大量并发的任务
  - 如何在 GPU上实现并发执行呢？

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

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

- 在 GPU 上执行 N 次
  - 如何确保每次执行的是不同的数据呢？

矢量点乘 (多分量, N=512) : block, grid

- <<<N, 1>>>: 并发执行 kernel mul N 次
- 术语: 在此，每次并发调用的 kernel 称为 一个 block
  - 所有 block 的集合称为 grid
  - <<<X1, X2>>> 中的 X1 设置 grid 的大小
  - 每个 block 都有一个唯一标记的 ID `blockIdx.x`
- 可以用 `blockIdx.x` 去数组中索引对应的元素，以确保对应的数据参与计算

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

矢量点乘 (多分量, N=512) : kernel


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

- 设备端，所有的 block 并行执行:

矢量点乘 (多分量，N=512)：不能完成归约

- 在设备端完成了相应分量的乘积。
- block 之间无法共享数据，不能在设备端完成归约求和操作

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

- 最终的归约求和在主机端完成。完整的 main 函数...

矢量点乘 (多分量, N=512)：main


```
#define N 512
int main(void) {
   int *a, *b, *c, sum=0;      // 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);
```

矢量点乘 (多分量, N=512)：main


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

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

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

for(int i=0; i<N; i++) sum += c[i]; // Reduce on Host

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

矢量点乘 (多分量, N=512)：设备端归约？

- 能不能在设备端完成归约呢？<<<X1, X2>>>
  - X1: 设置 kernel 的 block 数目；block 之间不能通信
  - X2: 设置 kernel 的 block 内 thread (线程) 数；
  - block 内线程间可以通过 shared memory (共享内存) 实现数据共享

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

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

  - block 内不同的线程用线程 ID 区分 `threadIdx.x`
  - `__shared__`：声明共享内存

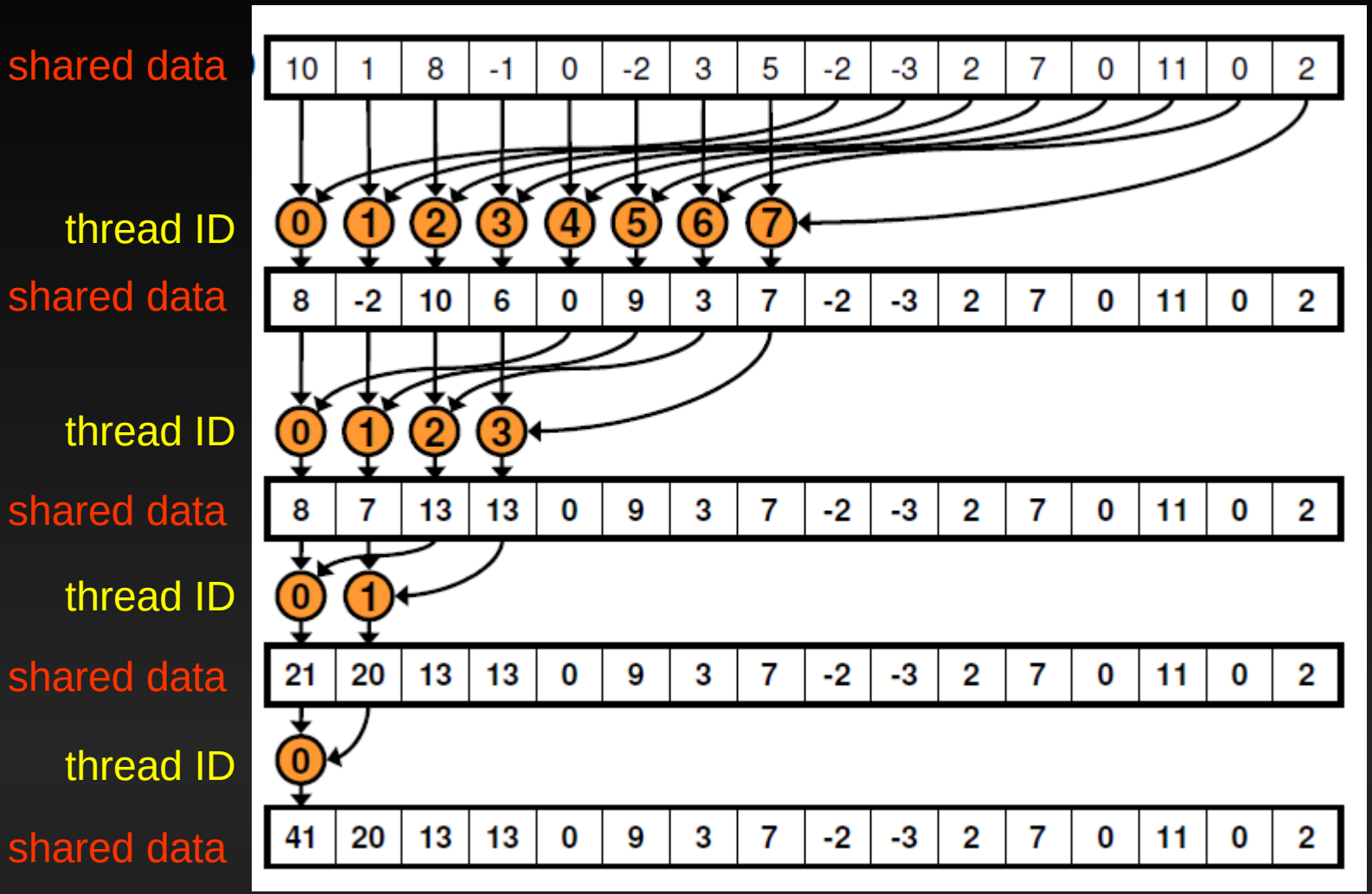
设备端归约：kernel

```
__global__ void mul(int *a, int *b, int *sum) {
        __shared__ int c[N];
        c[threadIdx.x] = a[threadIdx.x] * b[threadIdx.x];

        // Reduce code
}
```

- 设备端，所有的 thread 并行执行:

thread 0	thread 1	thread 2	thread 3
`c[0] = a[0] * b[0];`	`c[1] = a[1] * b[1];`	`c[2] = a[2] * b[2];`	`c[3] = a[3] * b[3];`

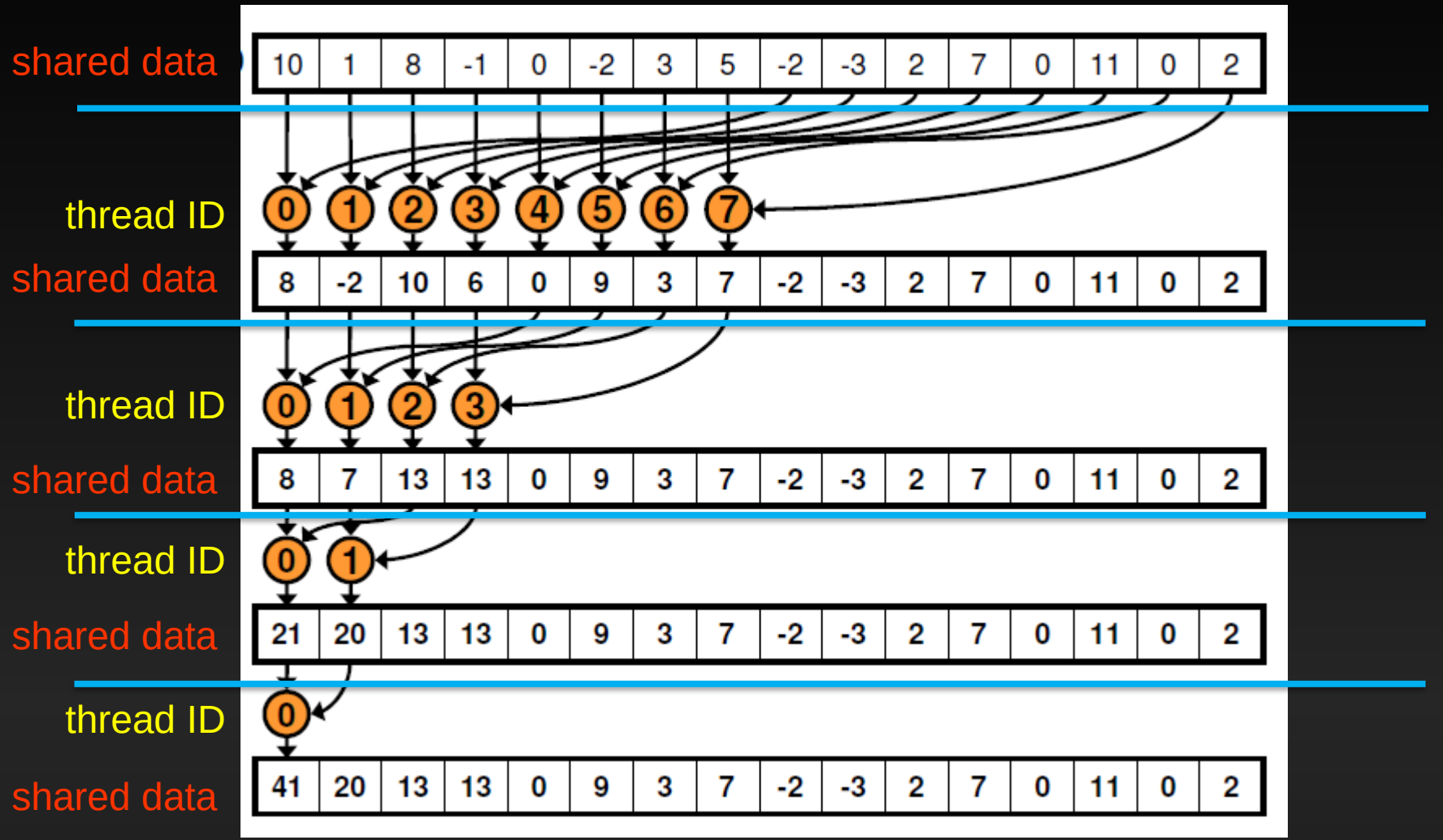
设备端归约实现


shared data
10 1 8 -1 0 -2 3 5 -2 -3 2 7 0 11 0 2
thread ID
0 1 2 3 4 5 6 7
shared data
8 -2 10 6 0 9 3 7 -2 -3 2 7 0 11 0 2
thread ID
0 1 2 3
shared data
8 7 13 13 0 9 3 7 -2 -3 2 7 0 11 0 2
thread ID
0 1
shared data
21 20 13 13 0 9 3 7 -2 -3 2 7 0 11 0 2
thread ID
0
shared data
41 20 13 13 0 9 3 7 -2 -3 2 7 0 11 0 2

设备端归约代码


```
// Reduce code
for (int i=N/2; i>0; i=i/2) {
        if (threadIdx.x < i) {
                c[threadIdx.x] +=
                                    c[threadIdx.x+i];
        }


}

if (threadIdx.x == 0)
        *sum = c[threadIdx.x]
```

设备端归约实现：问题？


shared data
10 1 8 -1 0 -2 3 5 -2 -3 2 7 0 11 0 2
thread ID
0 1 2 3 4 5 6 7
shared data
8 -2 10 6 0 9 3 7 -2 -3 2 7 0 11 0 2
thread ID
0 1 2 3
shared data
8 7 13 13 0 9 3 7 -2 -3 2 7 0 11 0 2
thread ID
0 1
shared data
21 20 13 13 0 9 3 7 -2 -3 2 7 0 11 0 2
thread ID
0
shared data
41 20 13 13 0 9 3 7 -2 -3 2 7 0 11 0 2

设备端归约：在哪同步呢？


```
__syncthreads();

// Reduce code
for (int i=N/2; i>0; i=i/2) {
        if (threadIdx.x < i) {
                c[threadIdx.x] +=
                                        c[threadIdx.x+i];
        }

        __syncthreads();
}

if (threadIdx.x == 0)
        *sum = c[threadIdx.x]
```

设备端归约: kernel 完整代码


```
__global__ void mul(int *a, int *b, int *sum) {
        __shared__ int c[N];
        c[threadIdx.x] = a[threadIdx.x] * b[threadIdx.x];

        __syncthreads();

        // Reduce code
        for (int i=N/2; i>0; i=i/2) {
                if (threadIdx.x < i)
                        c[threadIdx.x] +=
                                        c[threadIdx.x+i];
                __syncthreads();
        }
        if (threadIdx.x == 0) // shared memory to global memory
                *sum = c[threadIdx.x]
}
```

矢量点乘 (多分量, N=512)：main


```
#define N 512
int main(void) {
   int *a, *b, sum;           // host copies of a, b
   int *d_a, *d_b, *d_sum;    // device copies of a, b, sum
   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_sum, sizeof(int));

   // 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);
```

矢量点乘 (多分量, N=512)：main


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

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

    // Copy result back to host
    cudaMemcpy(&sum, d_sum, sizeof(int), cudaMemcpyDeviceToHost);

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

小结 (1 of 2)

- 主机和设备
  - *主机 Host*　　CPU & 内存
  - *设备 Device*　　GPU & 内存

- `__global__ __device__ __host__`
  - `__global__`: 主机端调用、设备端执行
  - `__device__`: 设备端调用、设备端执行
  - `__host__`: 主机端调用、主机端执行
  - `__host__ __device__`：同时在设备端和主机端生成两个版本

小结 (2 of 2)

- 编写 CUDA 程序的三步曲
  - 在设备端申请内存，并将数据从主机内存拷贝到设备内存
  - 调用 kernel，完成设备端的并发计算
  - 将计算结果从设备内存拷贝到主机内存，并释放内存
- 线程的两层组织结构
  - grid: block 的集合，由 blockIdx 标识。不同 block 之间不能通信
  - block: thread 的集合，由 threadIdx 标识
  - 同一个 block 内的线程可通过 shared memory 通信
  - 由 __sycnthreads() 同步

两种实现有什么不同呢？

- 两者的性能地有巨大的差异
  - 性能：多个线程块，每个只有一个线程 << 一个线程块，包含多个线程
- 为什么呢？
  - Warp： 同一个线程块内的连续 32 个线程
  - Warp 是 GPU 上执行和调度单元
  - 同一个 warp 内的所有线程执行相同的指令

Block 大小的设计

- E.g. blockDim = 160
  - 5 个 warp
- E.g. blockDim = 161
  - 6 个 warp

- 对于第一种情况
  - N 个 warp, 31 / 32 浪费

- 对于第二种情况
  - N/32 个 warp, 没有浪费

同时使用多个线程和线程块

矢量点乘：同时使用多个线程和线程块

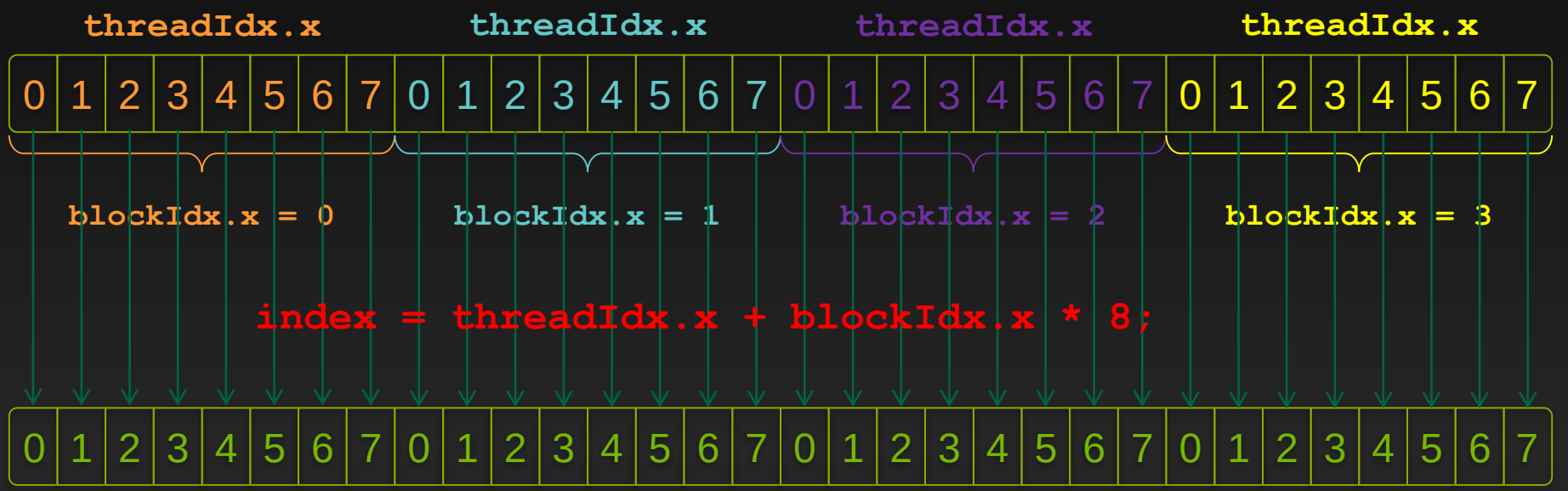
- 已经介绍了
  - 多个 blocks，每个 block 只有一个线程：无法归约
  - 一个 block，　每个block 包含多个线程　：block 中线程数目有限制，不能处理 N 非常大的情形

- **考虑采用多个 blocks 和 多个 threads 处理 N 很非常大的情形**

- 首先考虑此时数据的索引，也就是线程与待处理数据之间的对应关系

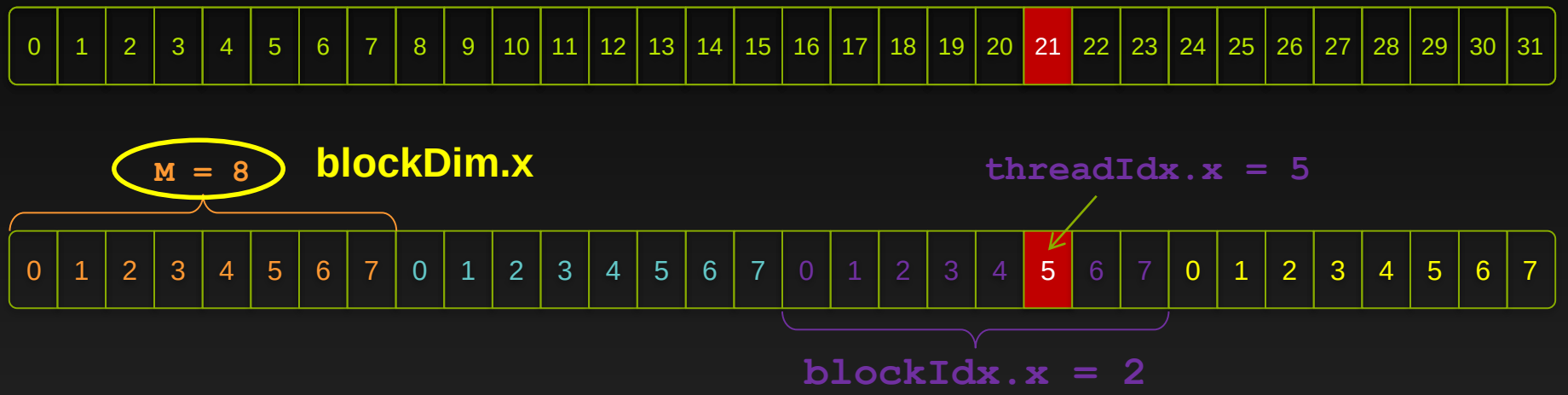
多个 blocks 和 threads 时数据的索引

- `blockIdx.x` 和 `threadIdx.x`
  - 考虑有4个 blocks、每个有 8 threads→线程与待处理数据的索引关系



数组索引: 实例

- 哪个线程块中的线程会计算这个红色的数据？




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

矢量点乘：同时使用多线程块和多线程

- 除 `blockIdx.x` 和 `threadIdx.x` 外，还要采用 `blockDim.x` 确定线程与待处理数据之间的对应关系

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

- 在线程块内，使用 shared memory 实现线程间通信，完成线程块内部分和的计算
- 线程块之间，无法通信。因此，部分和的归约（即总和），只能在主机端完成

同时使用多线程块和多线程：完整 kernel


```
__global__ void mul(int *a, int *b, int *sub_sum) {
        __shared__ int c[THREADS_PER_BLOCK];
        int index = threadIdx.x + blockIdx.x * blockDim.x;
        c[threadIdx.x] = a[index] * b[index];

        __syncthreads();
        // Reduce code
        for (int i=THREADS_PER_BLOCK/2; i>0; i=i/2) {
                if (threadIdx.x < i)
                        c[threadIdx.x] +=
                                        c[threadIdx.x+i];
                __syncthreads();
        }
        if (threadIdx.x == 0) // shared memory to global memory
                sub_sum[blockIdx.x] = c[threadIdx.x]
}
```

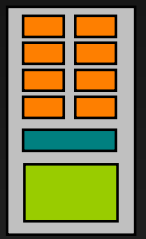
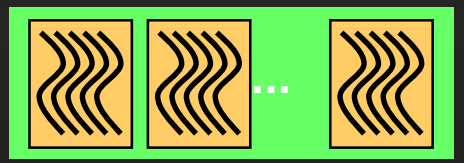
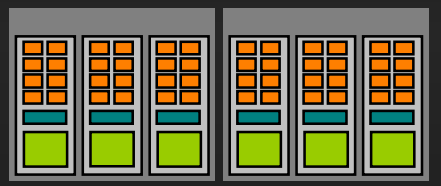
同时使用多线程块和多线程：完整 main


```
#define N (2048*2048)
#define THREADS_PER_BLOCK 512
#define BLOCKS_NUM (N / THREADS_PER_BLOCK )
int main(void) {
    int *a, *b, *sub_sum, sum=0;     // host copies of a, b, sub_sum
    int *d_a, *d_b, *d_sub_sum;      // device copies of a, b, sub_sum
    int size = N * sizeof(int);

    // Alloc space for device copies of a, b, sub_sum
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_sub_sum, sizeof(int)* BLOCKS_NUM );

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

同时使用多线程块和多线程：完整 main

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

// Launch mul() kernel on GPU
mul<<< BLOCKS_NUM,THREADS_PER_BLOCK>>>(d_a, d_b, d_sub_sum);

// Copy result back to host
cudaMemcpy(sub_sum, d_sub_sum, size, cudaMemcpyDeviceToHost);

// Reduce on Host
for(int i=0; i< BLOCKS_NUM; i++) sum += sub_sum[i];

// Cleanup
free(a); free(b); free(sub_sum);
cudaFree(d_a); cudaFree(d_b); cudaFree(d_sub_sum);return 0;}
```

GPU CUDA 编程模型

Software	GPU	
 Thread	 CUDA Core	Threads are executed by cuda core
 Thread Block	 SM	Thread blocks are executed on SM
 Grid	 Device	A kernel is launched as a grid of thread blocks

小结

- 多个线程和多个线程块协同
  - 利用内建变量 blockDim.x，实现线程与待处理数据的索引关系
  - 每个线程块完成部分和的归约
  - 部分和归约成总和在主机端完成

- 内建变量 (built-in)
  - threadIdx : 线程 ID
  - blockIdx : 线程块 ID
  - blockDim : block的维度
  - gridDim : grid 的维度
  - 这四个内建变量都是 dim3 型变量 (threadIdx.x; threadIdx.y; threadIdx.z)

STENCIL 计算 (模板计算)

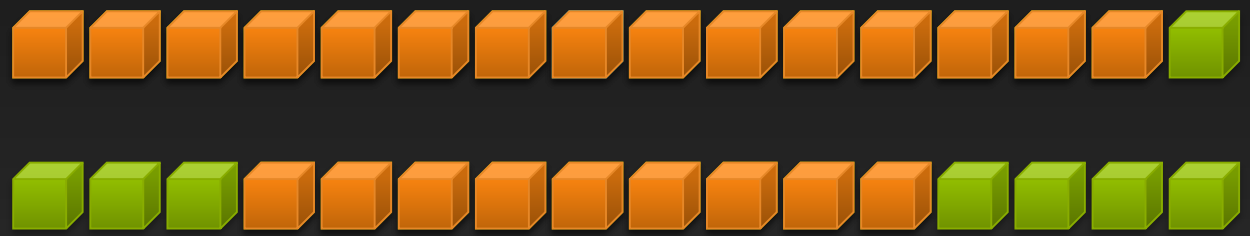
1D Stencil

- 对于每个点的计算
  - 需要用到周围固定的邻居
  - 每个方向的邻居数目记为 radius
- 考虑 1D Stencil 计算，radius =3。每个点的计算需要知道 7 个邻居的值

  - 假设对邻居 (包含当前点) 的操作为简单的求和操作

在 block 中实现 Stencil 计算

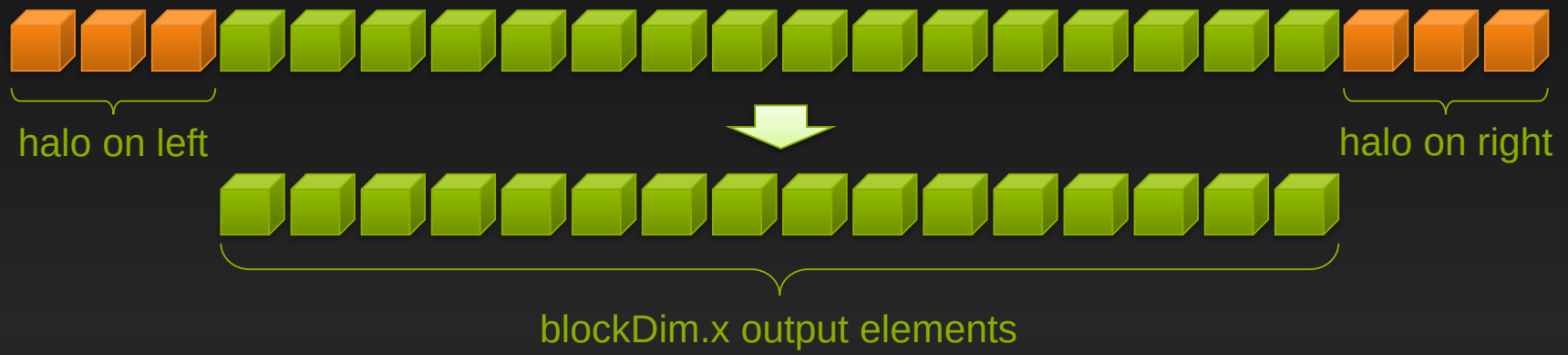
- 每个线程负责一个输出元素的计算
  - 线程块的大小 `blockDim.x` 等于输出元素的个数
- 先考虑不使用 shared memory，每个线程都从设备内存读取所需要的全部数据
  - 对 radius = 3，每个数据都需要读7次



  - 带宽浪费严重，考虑采用 shared memory 实现数据共享

使用共享内存实现数据共享

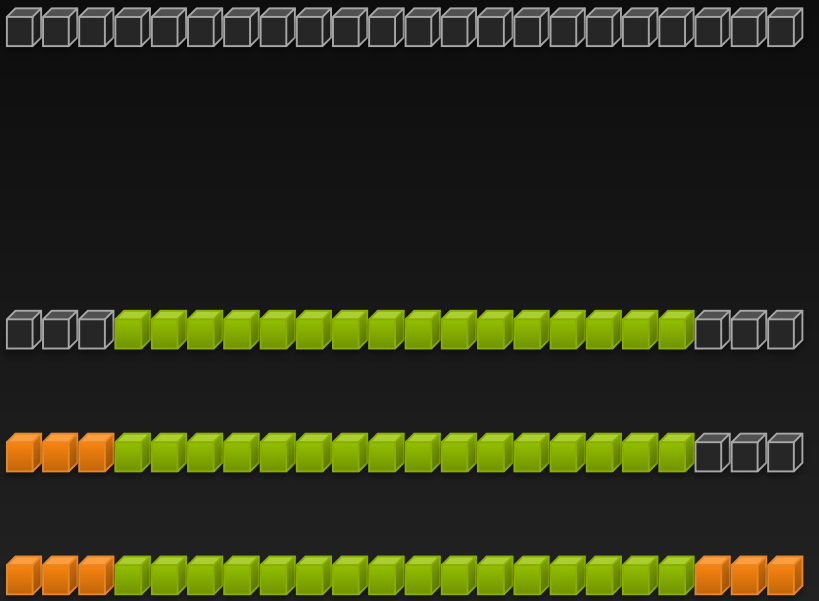
- shared memory 相当于数据缓存；执行顺序
  - 线程从 global memory 中读取 (`blockDim.x + 2 * radius`) 输入元素到 shared memory
  - 计算得到 `blockDim.x` 个输出结果
  - 将 `blockDim.x` 个输出结果写回到 global memory



- 每个数据从全局内存的读取次数从 7 次变为 1 次！！！

1D Stencil 计算的 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 计算的 Kernel


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

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

小结

- Stencil 计算
  - 相比全局内存，共享内存有更高的带宽、更小的延时
  - 考虑采用共享实现线程间数据共享，避免对全局内存的多次重复访问

CUDA C/C++ 编程介绍：今天介绍的内容

- 基本概念
  - 异构计算，主机和设备
  - __global__, __device__, __host__
  - 线程的两层组织结构，Grid 和 Block
- 内存组织架构
  - Shared memory/Global memory/Register
  - Global memory/Constant memory/Texture memory
  - Cache
- 内存管理
  - cudaMalloc(), cudaFree()
  - cudaMemcpyAsync(), cudaMemcpy()
- Stencil 计算
  - 利用shared memory减小对global memory 访问的例子

问题?