

并行与分布式计算基础：第十六讲

杨超

chao_yang@pku.edu.cn

2019 秋



内容提纲

1 CUDA 编程-2

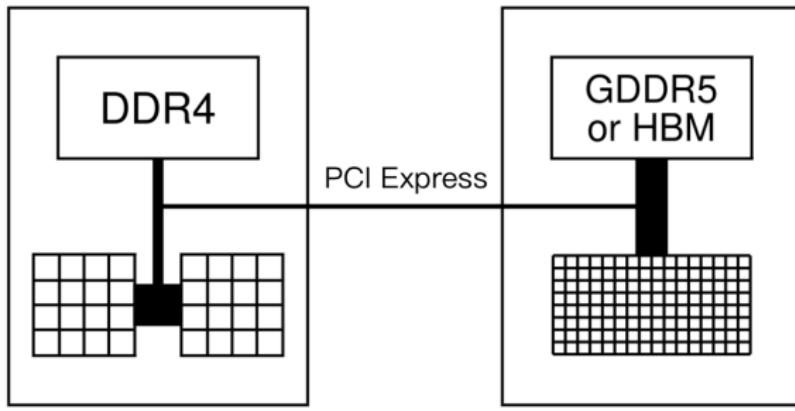
- 硬件视角
- 编程视角
- 重回 Hello World
- 程序举例：向量加法

硬件视角与编程视角

编程视角

主机 (host)

设备 (device)



硬件视角

motherboard

graphics card

可以是
多个

硬件视角

1 CUDA 编程-2

- 硬件视角
- 编程视角
- 重回 Hello World
- 程序举例：向量加法

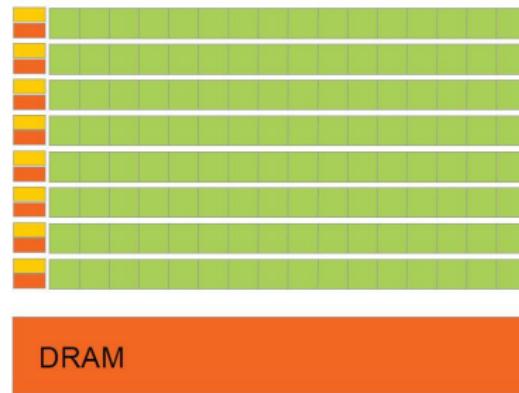
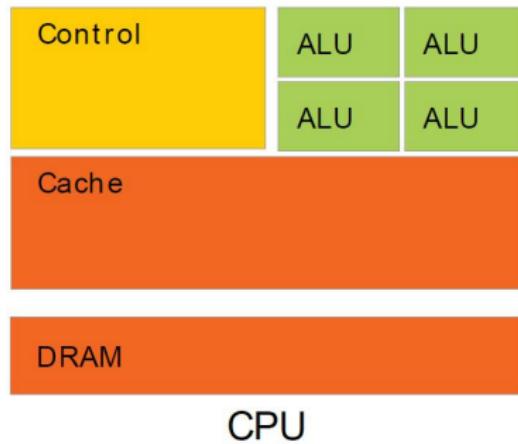
CPU 和 GPU 对比

CPU = Central Processing Unit:

- 面向延迟的设计；
- 非常大的 cache；
- 复杂的控制逻辑；
- 强大的 ALU；
- 不多的线程.

GPU = Graphic Processing Unit:

- 面向通量的设计；
- 小的 cache；
- 简单的控制逻辑；
- 高能效的 ALU；
- 大量的线程.



Nvidia GPU 总体架构简介

- NVIDIA 的 GPU 主要由多个 Streaming Multiprocessor (SM) 组成，SM 之间共享 L2 缓存；
- 每个 SM 由多个 Streaming Processor (SP) 组成，SP 之间共享控制逻辑和 L1 缓存；
- SP 主要包括单精度 (FP32) 核心、双精度 (FP64) 核心、特殊函数核心 (SFU) 等；
- 存储一般由 Graphics Double Data Rate (GDDR)，High Bandwidth Memory 2 (HBM2) 等组成；
- 与 CPU 传输数据一般使用 PCI-E (16-32 GB/s) 技术；
- 节点内 GPU 间数据传输可通过 NVLink (160-300 GB/s) 实现.

SIMT 与线程簇

每个 SM 中的核都是 SIMT (Single Instruction Multiple Threads):

- 每 32 个 (将来也许会增加) 线程一组，构成一个线程簇 (warp)；
- 同一个线程簇中的线程同时执行同样的程序；
- 每个线程处理的数据不同.

为什么性能高？

- 大量的线程数，从而大幅提升了吞吐能力；
- 无需上下文切换 (context switching)：每个线程将数据存储在私有存储 (寄存器) 中；
- 硬件内置的线程簇调度器 (warp scheduler) 高效调度线程簇，保障多个活跃的线程簇被同时执行，线程簇非活跃往往是因为等待数据.

P100 的 SM 架构



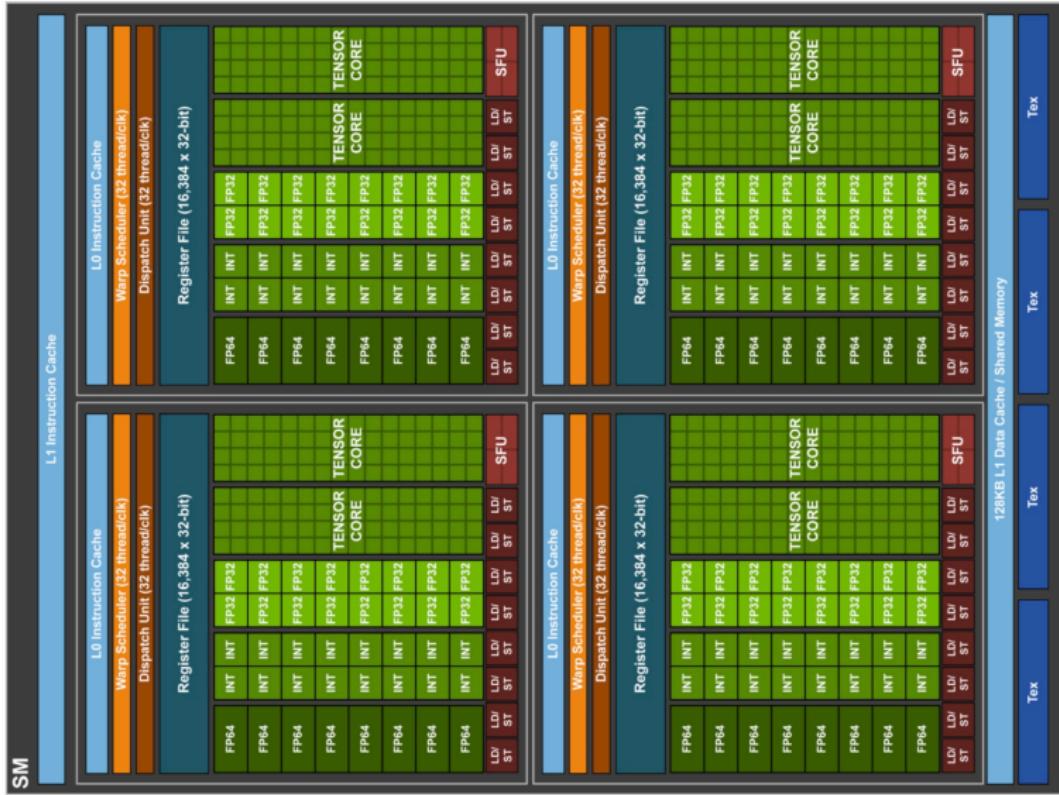
FP32 核数	FP64 核数	寄存器大小	共享存储大小	线程簇调度器
64	32	256 KB	64 KB	2

P100 总体架构



SM 个数	总 FP32 核数	总 FP64 核数	L2 Cache	带宽	内存大小
56	3584	1792	4096 KB	732 GB/s	16 GB HBM2

Tesla V100 的 SM 架构



Tesla V100 总体架构



SM 个数	总 FP32 核数	总 FP64 核数	总 Tensor 核数	L2 Cache	带宽	内存大小
80	5120	2560	640	6144 KB	900 GB/s	16 GB HBM2

几代 NVIDIA GPU 硬件参数总结

Tesla Product	Tesla K40	Tesla M40	Tesla P100	Tesla V100
GPU	GK180 (Kepler)	GM200 (Maxwell)	GP100 (Pascal)	GV100 (Volta)
SMs	15	24	56	80
TPCs	15	24	28	40
FP32 Cores / SM	192	128	64	64
FP32 Cores / GPU	2880	3072	3584	5120
FP64 Cores / SM	64	4	32	32
FP64 Cores / GPU	960	96	1792	2560
Tensor Cores / SM	NA	NA	NA	8
Tensor Cores / GPU	NA	NA	NA	640
GPU Boost Clock	810/875 MHz	1114 MHz	1480 MHz	1530 MHz
Peak FP32 TFLOPS ¹	5	6.8	10.6	15.7
Peak FP64 TFLOPS ¹	1.7	.21	5.3	7.8
Peak Tensor TFLOPS ¹	NA	NA	NA	125
Texture Units	240	192	224	320

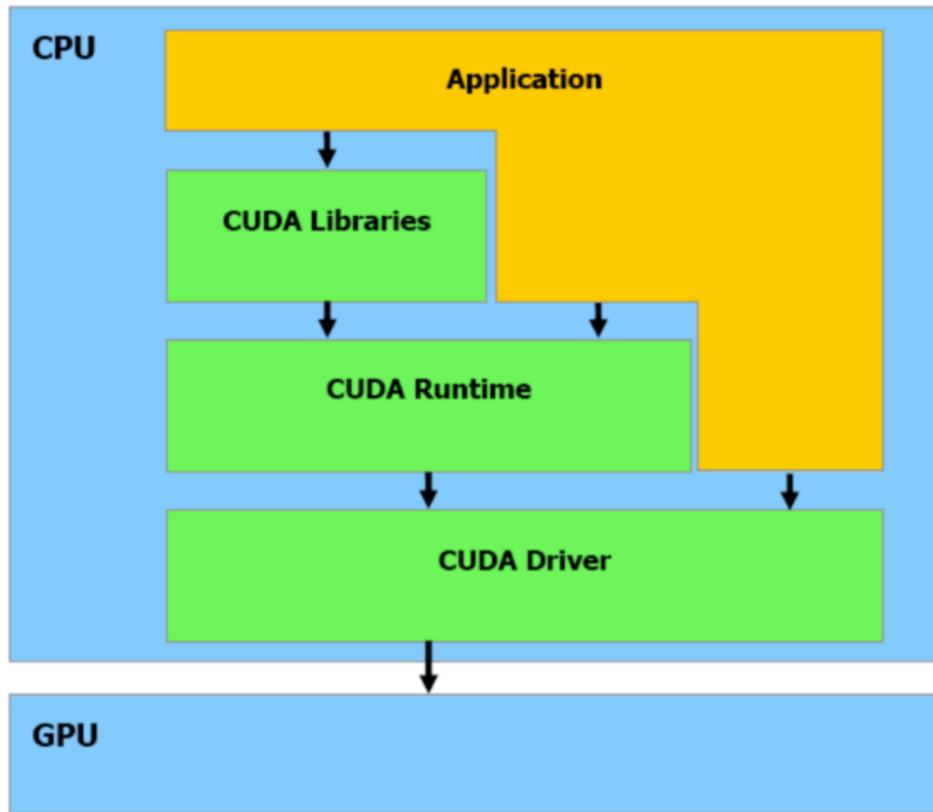
Memory Interface	384-bit GDDR5	384-bit GDDR5	4096-bit HBM2	4096-bit HBM2
Memory Size	Up to 12 GB	Up to 24 GB	16 GB	16 GB
L2 Cache Size	1536 KB	3072 KB	4096 KB	6144 KB
Shared Memory Size / SM	16 KB/32 KB/48 KB	96 KB	64 KB	Configurable up to 96 KB
Register File Size / SM	256 KB	256 KB	256 KB	256KB
Register File Size / GPU	3840 KB	6144 KB	14336 KB	20480 KB
TDP	235 Watts	250 Watts	300 Watts	300 Watts
Transistors	7.1 billion	8 billion	15.3 billion	21.1 billion
GPU Die Size	551 mm ²	601 mm ²	610 mm ²	815 mm ²
Manufacturing Process	28 nm	28 nm	16 nm FinFET+	12 nm FFN

编程视角

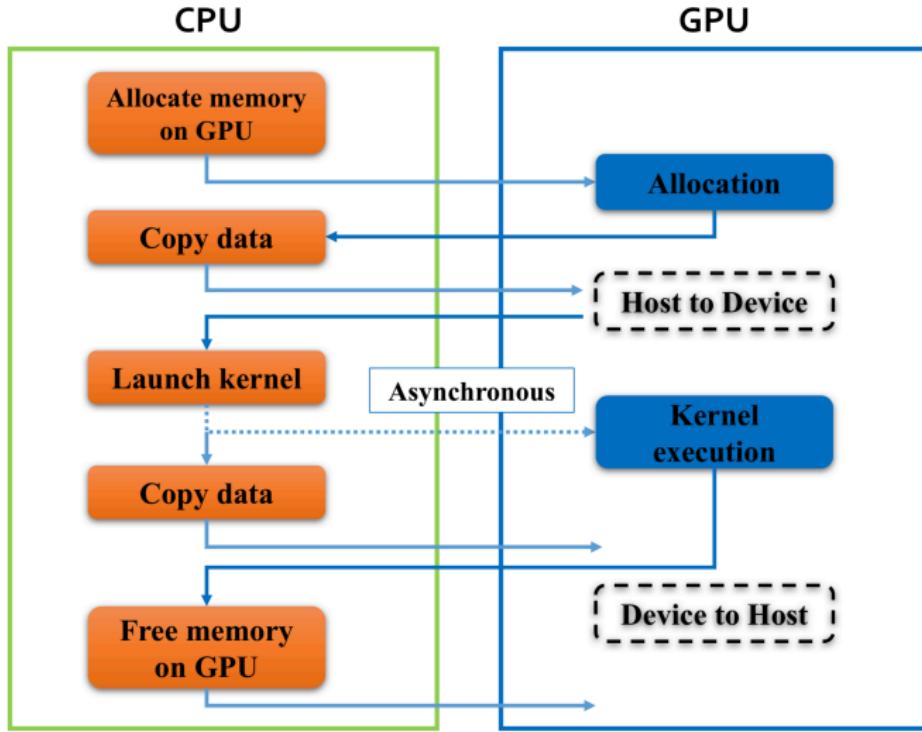
1 CUDA 编程-2

- 硬件视角
- 编程视角
- 重回 Hello World
- 程序举例：向量加法

CUDA 的软件架构



CUDA 异构并行计算流程一览



host 代码和 kernel 代码

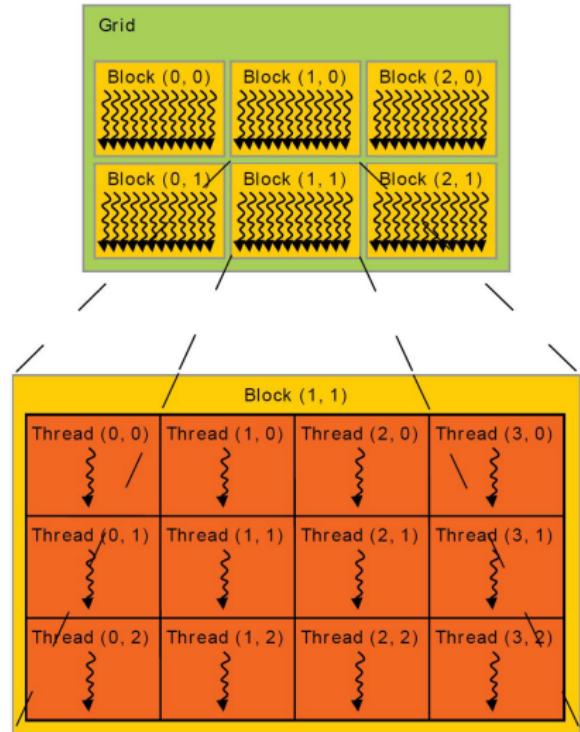
一个典型的 CUDA 程序由两部分组成：

- host 代码——CPU 端执行：
 - ▶ CPU 计算；
 - ▶ GPU 内存分配和释放；
 - ▶ CPU 数据与 GPU 数据的传输，包括常量和普通数据；
 - ▶ 检查错误信息、计时等.
- kernel 代码——GPU 端执行：
 - ▶ 每段 kernel 代码可以作为多个执行实例 (execution instances)；
 - ▶ 每个执行实例被一个 SM 执行，每个 SM 可以对应多个实例；
 - ▶ 每个执行实例可以由多个线程并发执行；
 - ▶ 每个线程拥有私有的数据信息；
 - ▶ 同一执行实例的线程间可以通过共享存储进行交互.

CUDA 线程与 kernel 程序的关系

一个 CUDA kernel 程序被一个线程网格 (thread grid) 中的所有线程块 (thread block) 的所有线程执行：

- 线程网格中线程块的个数决定了 kernel 程序有多少个执行实例；
- 线程块中线程的个数决定了每个执行实例由多少个线程执行；
- 每个线程通过各自的索引来访问所需数据和控制程序执行流程。



CUDA 并行网格组织层次

CUDA 并行线程网格分两层进行组织：

- 第一层：多个 thread 按照 1、2 或 3 维方式组成一个 block，同一 block 的线程可以通过共享内存、原子操作、栅栏同步来协作。
 - ▶ Block 的 x,y,z 轴大小通过内部结构体变量 blockDim 的 x,y,z 成员变量表示；
 - ▶ 每个线程的 x,y,z 轴坐标通过内部结构体变量 threadIdx 的 x,y,z 成员变量定位。
- 第二层：多个 block 按照 1、2 或 3 维方式组成一个 grid，不同 block 内线程相对独立。
 - ▶ Grid 的 x,y,z 轴大小通过内部结构体变量 gridDim 的 x,y,z 成员变量表示；
 - ▶ 每个 block 的 x,y,z 轴坐标通过内部结构体变量 blockIdx 的 x,y,z 成员变量定位。

重回 Hello World

1 CUDA 编程-2

- 硬件视角
- 编程视角
- 重回 Hello World
- 程序举例：向量加法

hello world!

cuda_hello.cu

```
1 __global__ void mykernel(void) {
2     printf("Hello World from GPU by thread %d!\n", threadIdx.x);
3 }
4 int main(void) {
5     // Use 1 thread
6     mykernel<<<1,1>>>();
7     // Flush the output
8     cudaDeviceSynchronize();
9     // Use 4 threads
10    mykernel<<<1,4>>>();
11    // Flush the output
12    cudaDeviceSynchronize();
13    return 0;
14 }
```

代码分析

```
1  __global__ void mykernel(void) {  
2      printf("Hello World from GPU by thread %d!\n", threadIdx.x);  
3 }
```

- CUDA C/C++ 关键字 `__global__` 表示两点：
 - ▶ 函数在 host 端被调用；
 - ▶ 函数在 device 端运行.
- `threadIdx.x` 是线程索引.

```
1 ...
2 mykernel<<<1, 1>>>();
3 ...
4 mykernel<<<1, 4>>>();
5 ...
```

- <<<...>>> 表示在 host 调用 device 代码，又称 kernel launch
 - ▶ 第一个参数代表线程网格大小，可为整数或 dim3 类型；
 - ▶ 第二个参数代表线程块大小，可为整数或 dim3 类型.

Kernel Launch

In its simplest form it looks like:

```
kernel_routine<<<gridDim, blockDim>>>(args);
```

- gridDim is the number of instances of the kernel (the “grid” size)
- blockDim is the number of threads within each instance (the “block” size)
- args is a limited number of arguments, usually mainly pointers to arrays in graphics memory, and some constants which get copied by value

The more general form allows gridDim and blockDim to be 2D or 3D to simplify application programs

具体到每个线程

At the lower level, when one instance of the kernel is started on a SM it is executed by a number of threads, each of which knows about:

- some variables passed as arguments
- pointers to arrays in device memory (also arguments)
- global constants in device memory
- shared memory and private registers/local variables
- some special variables:
 - `gridDim` size (or dimensions) of grid of blocks
 - `blockDim` size (or dimensions) of each block
 - `blockIdx` index (or 2D/3D indices) of block
 - `threadIdx` index (or 2D/3D indices) of thread
 - `warpSize` always 32 so far, but could change

线程网格组织层次举例：1D

1D grid with 4 blocks, each with 64 threads:

- gridDim = 4
- blockDim = 64
- blockIdx ranges from 0 to 3
- threadIdx ranges from 0 to 63



编译与运行

- 编译：

```
$ module load cuda
$ nvcc -o hello cuda_hello.cu
```

- nvcc -o hello hello_world.cu 在 host 和 device 端分别编译：
 - ▶ host 代码 (例如 main) 被 gcc 等 host 端编译器编译；
 - ▶ device 代码 (例如 mykernel) 被 NVIDIA 编译器编译.
- 运行结果：

```
$ ./hello
Hello World from GPU by thread 0!
Hello World from GPU by thread 0!
Hello World from GPU by thread 1!
Hello World from GPU by thread 2!
Hello World from GPU by thread 3!
```

程序举例：向量加法

1 CUDA 编程-2

- 硬件视角
- 编程视角
- 重回 Hello World
- 程序举例：向量加法

CPU 上的向量加法

```
1 // Compute vector sum h_C = h_A + h_B
2 void cpu_vec_add(float *h_A, float *h_B, float *h_C, int n) {
3     for (int i = 0; i < n; i++)
4         h_C[i] = h_A[i] + h_B[i];
5 }
6
7 int main() {
8     // Memory allocation for h_A, h_B, and h_C
9     ...
10    cpu_vec_add(h_A, h_B, h_C, N);
11 }
```

利用 GPU 实现向量加法的三个步骤

```
1 #include <cuda.h>
2 ...
3 void vec_add(float *h_A, float *h_B, float *h_C, int n) {
4     int size = n* sizeof(float);
5     float *d_A, *d_B, *d_C;
6     ...
7     // 1. Allocate device memory for A, B, and C
8     // Transfer A and B to device memory
9
10    // 2. Kernel launch code to have the device
11    // to perform the actual vector addition
12
13    // 3. Transfer C from device to host
14    // Free device memory for A, B, C
15 }
```

步骤一和步骤三的代码

```
1 void vec_add(float* h_A, float* h_B, float* h_C, int n) {  
2     ...  
3     // 1. Allocate device memory for A, B, and C  
4     cudaMalloc((void **) &d_A, size);  
5     cudaMalloc((void **) &d_B, size);  
6     cudaMalloc((void **) &d_C, size);  
7     // Transfer A and B to device memory  
8     cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);  
9     cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);  
10    // 2. Kernel invocation code to be shown later  
11    ...  
13    // 3. Transfer C from device to host  
14    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);  
15    // Free device memory for A, B, C  
16    cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);  
17 }  
18 }
```

内存分配

- `cudaError_t cudaMalloc(void** devPtr, size_t size)`

分配设备内存 (global memory):

- ▶ `devPtr`: 存放所分配设备内存地址;
- ▶ `size`: 分配内存的大小, 字节为单位;
- ▶ `cudaError_t`: 如果调用成功返回 `cudaSuccess`, 否则返回相应错误码.

- `cudaError_t cudaFree(void** devPtr)`

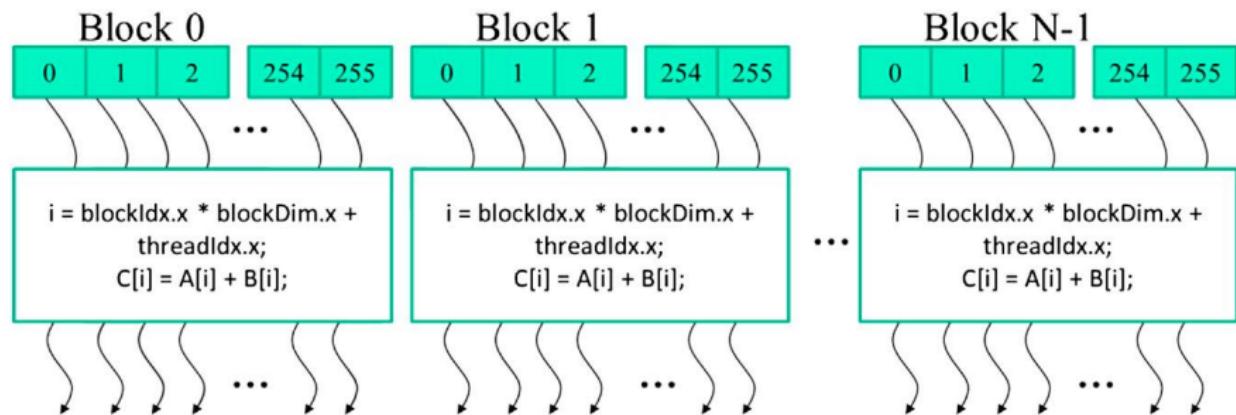
释放已经分配的设备内存:

- ▶ `devPtr`: 对应上述分配内存函数中的设备内存地址;
- ▶ `cudaError_t`: 如果调用成功返回 `cudaSuccess`, 否则返回相应错误码.

数据传输

- `cudaError_t cudaMemcpy(void* dst, const void* src, size_t count, cudaMemcpyKind kind)`
 - ▶ `dst`: 传输目的内存地址;
 - ▶ `src`: 数据源内存地址;
 - ▶ `count`: 传输的数据大小, 以字节为单位;
 - ▶ `cudaMemcpyKind`: 数据传输方式, 有:
 - ★ Host → Host: `cudaMemcpyHostToHost`;
 - ★ Host → Device: `cudaMemcpyHostToDevice`;
 - ★ Device → Host: `cudaMemcpyDeviceToHost`;
 - ★ Device → Device: `cudaMemcpyDeviceToDevice`;
 - ▶ `cudaError_t`: 如果调用成功返回 `cudaSuccess`, 否则返回相应错误码.

步骤二的线程组织



- Grid 中的 block 按照一维方式组织, 每个 block 中的 thread 也是按照一维方式组织;
- 可以看出向量加法中, 线程组织与数据的处理相对应.

步骤二：device 端的 kernel 代码

```
1 // Compute vector sum C = A + B
2 // Each thread performs one pair-wise addition
3 __global__
4 void vec_add_kernel(float* d_A, float* d_B, float* d_C, int n){
5     int i = blockDim.x * blockIdx.x + threadIdx.x;
6     if (i < n) d_C[i] = d_A[i] + d_B[i];
7 }
```

- 思考：为什么加上 `if (i < n)` 判断？

CUDA C 函数声明关键字

CUDA C 关键字	执行位置	调用位置
<code>--device__ type DeviceFunc()</code>	device	device
<code>--global__ void KernelFunc()</code>	device	host
<code>--host__ type HostFunc()</code>	host	host

- 默认函数是 host，所以一般 `--host__` 省略；
- `--global__` 定义了 kernel，只能返回 void；
- `--host__` 和 `--device__` 可以一起使用，告诉编译器生成 CPU 和 GPU 两个版本。

步骤二：host 上的 kernel launch

```
1 __host__
2 void vec_add(float* h_A, float* h_B, float* h_C, int n) {
3     ...
4     // d_A, d_B, d_C allocations and copies omitted
5     // Run ceil(n/256.0) blocks of 256 threads each
6     dim3 grid_dim(ceil(n/256.0),1,1);
7     dim3 block_dim(256,1,1);
8     vec_add_kernel<<<grid_dim, block_dim>>>(d_A, d_B, d_C, n);
9     ...
10 }
```

- 注意：此处 `__host__` 关键字可以省略；
- 思考：为什么写成 `ceil(n/256.0)`？

kernel launch 的一般形式

- 假设 kernel 函数定义：`_global_ void Func(float* input)`
- 那么在 host 上发起时的执行配置：

```
Func<<<grid_dim, blk_dim, shared_size, stream_id>>>(input)
```

- ▶ `grid_dim`: 整型或 `dim3` 类型, 定义 grid 的大小, 总线程块数量等于 `grid_dim.x × grid_dim.y × grid_dim.z`;
- ▶ `blk_dim`: 整型或 `dim3` 类型, 定义每个 block 的大小, 总线程数量等于 `blk_dim.x × blk_dim.y × blk_dim.z`;
- ▶ `shared_size`: 可选参数, 默认为 0, `size_t` 类型, 确定为每个线程块动态分配的 shared memory 大小 (暂时忽略);
- ▶ `stream_id`: 可选参数, 默认为 0, `cudaStream_t` 类型, 定义执行关联的 stream (暂时忽略).

统计 kernel 运行时间

- 创建和销毁 CUDA event:

```
1  cudaEvent_t start, stop;  
2  cudaEventCreate(&start);  
3  cudaEventCreate(&stop);  
4  ...  
5  cudaEventDestroy(start);  
6  cudaEventDestroy(stop);
```

- 利用 CUDA event 统计 kernel 耗时:

```
1  cudaEventRecord(start, 0);  
2  ...  
3  cudaEventRecord(stop, 0);  
4  cudaEventSynchronize(stop);  
5  float elapsedTime;  
6  cudaEventElapsedTime(&elapsedTime, start, stop);
```

完整 host 端和 devide 端代码

host 端代码

```
1 void vec_add(float *h_A, float *h_B, float *h_C, int n) {  
2     int size = n * sizeof(float);  
3     float *d_A, *d_B, *d_C;  
4     cudaEvent_t start, stop;  
5     float elapsed_time = 0.0;  
6     cudaEventCreate(&start);  
7     cudaEventCreate(&stop);  
8  
9     //1. Allocate device memory and transfer data to device  
10    cudaMalloc((void **) &d_A, size);  
11    cudaMalloc((void **) &d_B, size);  
12    cudaMalloc((void **) &d_C, size);  
13    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);  
14    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);  
15  
16    // 2. Configure execution and launch kernel  
17    dim3 grid_dim(ceil(n/256.0), 1, 1);  
18    dim3 block_dim(256, 1, 1);
```

```
19     cudaEventRecord(start, 0);
20     vec_add_kernel<<<grid_dim, block_dim>>>(d_A, d_B, d_C, n);
21     cudaEventRecord(stop, 0);
22     cudaEventSynchronize(stop);
23     cudaEventElapsedTime(&elapsed_time, start, stop);
24
25     // 3. Transfer result back to host and free device memory
26     cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
27     cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
28 }
```

device 端代码

```
1 // Compute vector sum C = A + B
2 // Each thread performs one pair-wise addition
3 __global__
4 void vec_add_kernel(float* d_A, float* d_B, float* d_C, int n){
5     int i = blockDim.x * blockIdx.x + threadIdx.x;
6     if (i < n) d_C[i] = d_A[i] + d_B[i];
7 }
```

main 函数中的调用代码

```
1 int main(void) {
2
3     int N = 1024000;
4     ...
5     // Memory allocation for vectors A, B and C
6     float *h_A = rand_vec(N);
7     float *h_B = rand_vec(N);
8     float *h_C = raw_vec(N);
9     ...
10    vec_add(h_A, h_B, h_C, N);
11    ...
12 }
```

- 部分运行结果：

```
Vector length: 102400.  
CPU: 0.00183 sec  
grid dim: 400, 1, 1.  
block dim: 256, 1, 1.  
kernel time: 0.00009 sec.  
GPU: 0.37654 sec  
All values correct.  
...  
Vector length: 1024000.  
CPU: 0.01484 sec  
grid dim: 4000, 1, 1.  
block dim: 256, 1, 1.  
kernel time: 0.00011 sec.  
GPU: 0.35207 sec  
All values correct.
```

- 思考 1：为什么 GPU 总时间比 CPU 还慢？
- 思考 2：为什么规模增加 10 倍，但是 kernel 计算时间增加不多？