

第一讲 CUDA C介绍

CUDA C vs. Thrust vs. CUDA Libraries



目录

- 异构并行计算
- CUDA C、CUDA Libs、OpenACC
- CUDA编程模型
- 内存分配及数据转移接口函数
- 数据并行化与线程



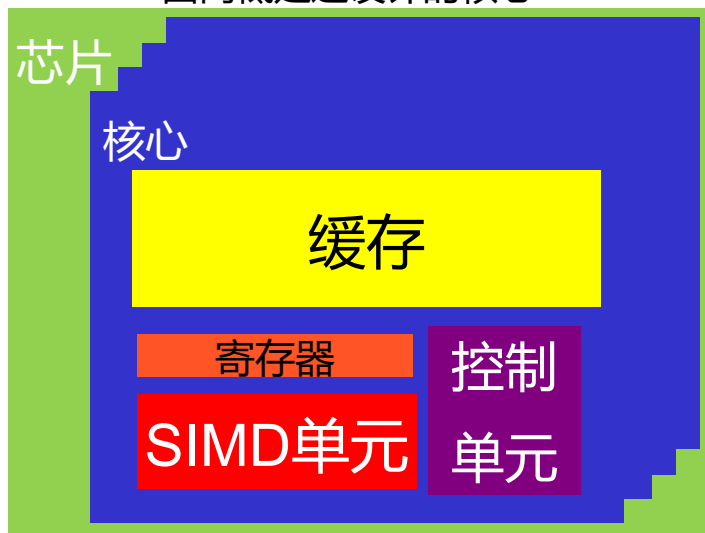
小节目标

- 学习低延迟设备（CPU）与大吞吐量设备（GPU）之间的主要差别
- 了解为什么越来越多应用使用这两种类型的设备

在设计上，CPU和GPU存在差异

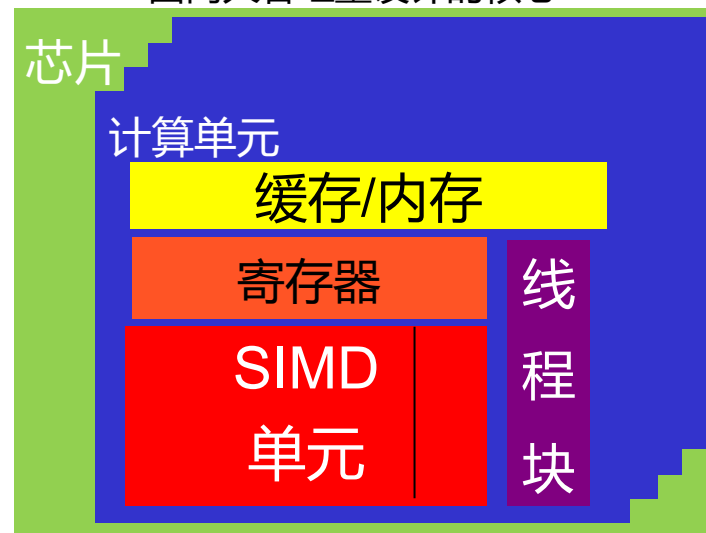
CPU

面向低延迟设计的核心

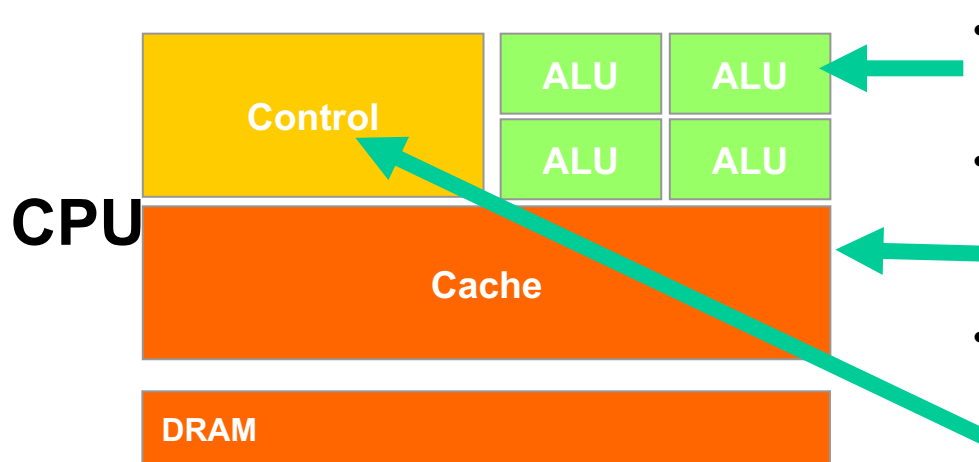


GPU

面向大吞吐量设计的核心



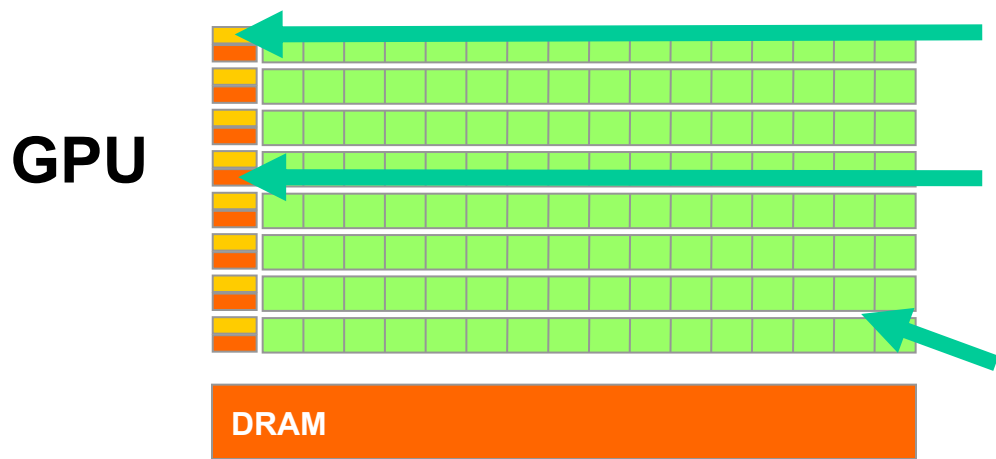
CPUs: 面向低延迟的设计



- 强大的运算器 (ALU)
 - 减少操作延迟
- 大容量缓存 (caches)
 - 将长延迟内存访问转换为短延迟缓存访问
- 精密控制单元 (control)
 - 用于减少分支延迟的分支预测方法
 - 用于减少数据延迟的数据转发方法



GPUs: 面向大吞吐量的设计



- 小容量缓存
 - 提高内存吞吐量
- 简单的控制单元
 - 没有分支预测
 - 没有数据转发
- 节能的运算器 (ALU)
 - 大量且高延迟, 但高度流水线化, 实现高吞吐量
- 大量线程块
 - 线程逻辑
 - 线程状态



大量应用同时使用CPU与GPU

- GPUs负责并行计算
 - 并行计算能力上，GPUs比CPUs强十倍以上
- CPUs负责串行计算
 - 串行计算能力上，CPUs比GPUs强十倍以上



目录

- 异构并行计算
- **CUDA C、CUDA Libs、OpenACC**
- CUDA编程模型
- 内存分配及数据转移接口函数
- 数据并行化与线程



小节目标

- 学习GPU计算的主要方法和开发者资源

加速应用的三种方式

应用

函数库

使用方便
性能出众

编译器指令

使用方便
可移植代码

编程语言

性能出众
灵活性高



GPU加速

线性代数

FFT, BLAS,
SPARSE, Matrix



CUDA|tools



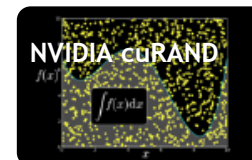
CUSP

数学

RAND, Statistics

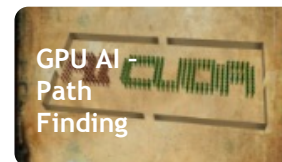
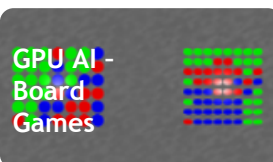


ArrayFire



数据结构与AI

Sort, Scan, Zero Sum



可视化处理

Image & Video



电子科技大学
University of Electronic Science and Technology of China

Thrust中的向量加法

```
thrust::device_vector<float> deviceInput1(inputLength);  
thrust::device_vector<float> deviceInput2(inputLength);  
thrust::device_vector<float> deviceOutput(inputLength);
```

```
thrust::copy(hostInput1, hostInput1 + inputLength,  
             deviceInput1.begin());
```

```
thrust::copy(hostInput2, hostInput2 + inputLength,  
             deviceInput2.begin());
```

```
thrust::transform(deviceInput1.begin(), deviceInput1.end(),  
                  deviceInput2.begin(), deviceOutput.begin(),  
                  thrust::plus<float>());
```



OpenACC

- 用于C, C++和FORTRAN的代码指令

```
#pragma acc parallel loop  
copyin(input1[0:inputLength],input2[0:inputLength]),  
copyout(output[0:inputLength])  
for(i = 0; i < inputLength; ++i) {  
    output[i] = input1[i] + input2[i];  
}
```



GPU 编程语言

Numerical analytics ►

MATLAB, Mathematica, LabVIEW

Fortran ►

CUDA Fortran

C ►

CUDA C

C++ ►

CUDA C++

Python ►

PyCUDA, Copperhead, Numba

F# ►

Alea.cuBase



目录

- 异构并行计算
- CUDA C、CUDA Libs、OpenACC
- **CUDA编程模型**
- 内存分配及数据转移接口函数
- 数据并行化与线程



编程模型

- 编程模型是底层计算机系统的抽象，用于表达算法和数据
- 程序语言和 API 用于上述抽象的实现
 - 具体的算法和数据结构
 - 独立于编程语言和 API 的选择

Programming mode, Patrick McCormick (LANL) et.al.,
<https://asc.llnl.gov/content/assets/docs/exascale-pmWG.pdf>



一些设计目标

- 扩展到 100 个内核，1000 个并行线程
- 专注于并行算法设计
- 支持异构系统(如CPU+GPU)
 - CPU和GPU有着各自独立的DRAM

关键的并行化特点

- 并发线程的层次化结构
- 轻量级同步指令
- 用于协作线程的共享内存模型



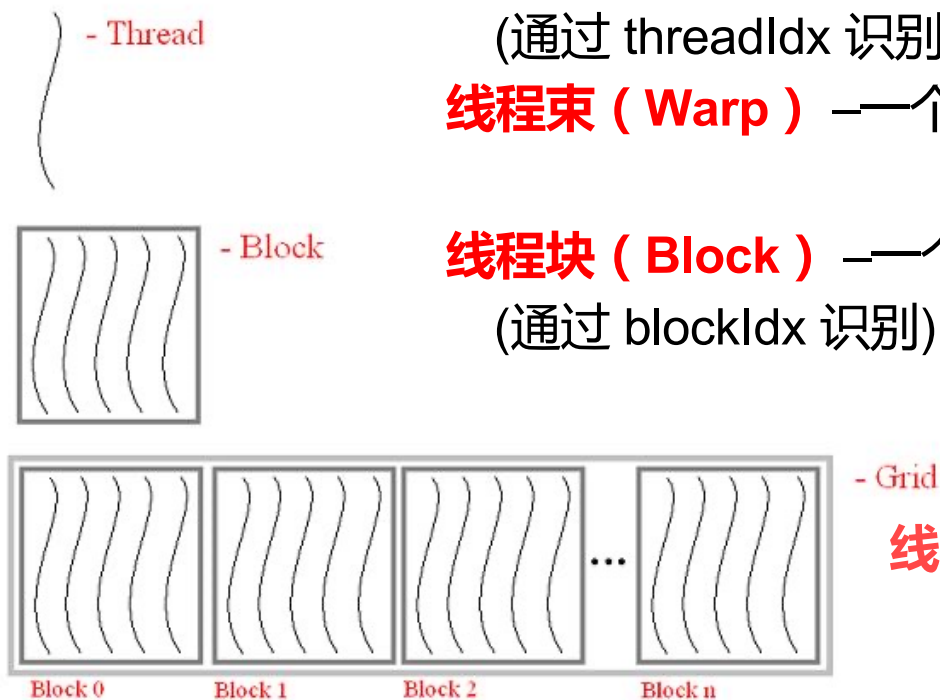
线程层次化结构

线程 (Thread) – 由 CUDA 运行时分发
(通过 threadIdx 识别)

线程束 (Warp) – 一个最多 32 个线程的调度单元

线程块 (Block) – 一个由用户定义的 1 到 512 个线程组。
(通过 blockIdx 识别)

线程网格 (Grid) – 包含一个或多个线程块。为每个 CUDA 核函数创建一个网格



CUDA 内存层次化结构

- CUDA 平台具有三种主要的内存类型

寄存器——用于自动变量和寄存器溢出的每个**线程**内存。

共享内存——每个**线程块**的低延迟内存，允许块内数据共享和同步。线程可以通过这块内存安全地共享数据，并且可以通过 `__syncthreads()` 进行屏障同步

全局内存——可以在线程块或线程网格之间共享的**设备级**内存



CUDA 硬件方面

- Tesla架构的主要组件是：
 - 流式多处理器 Streaming Multiprocessor
 - 标量处理器 Scalar Processor
 - 存储器层次结构 Memory hierarchy
 - 互联网络 Interconnection network
 - Host接口 Host interface



流式多处理器

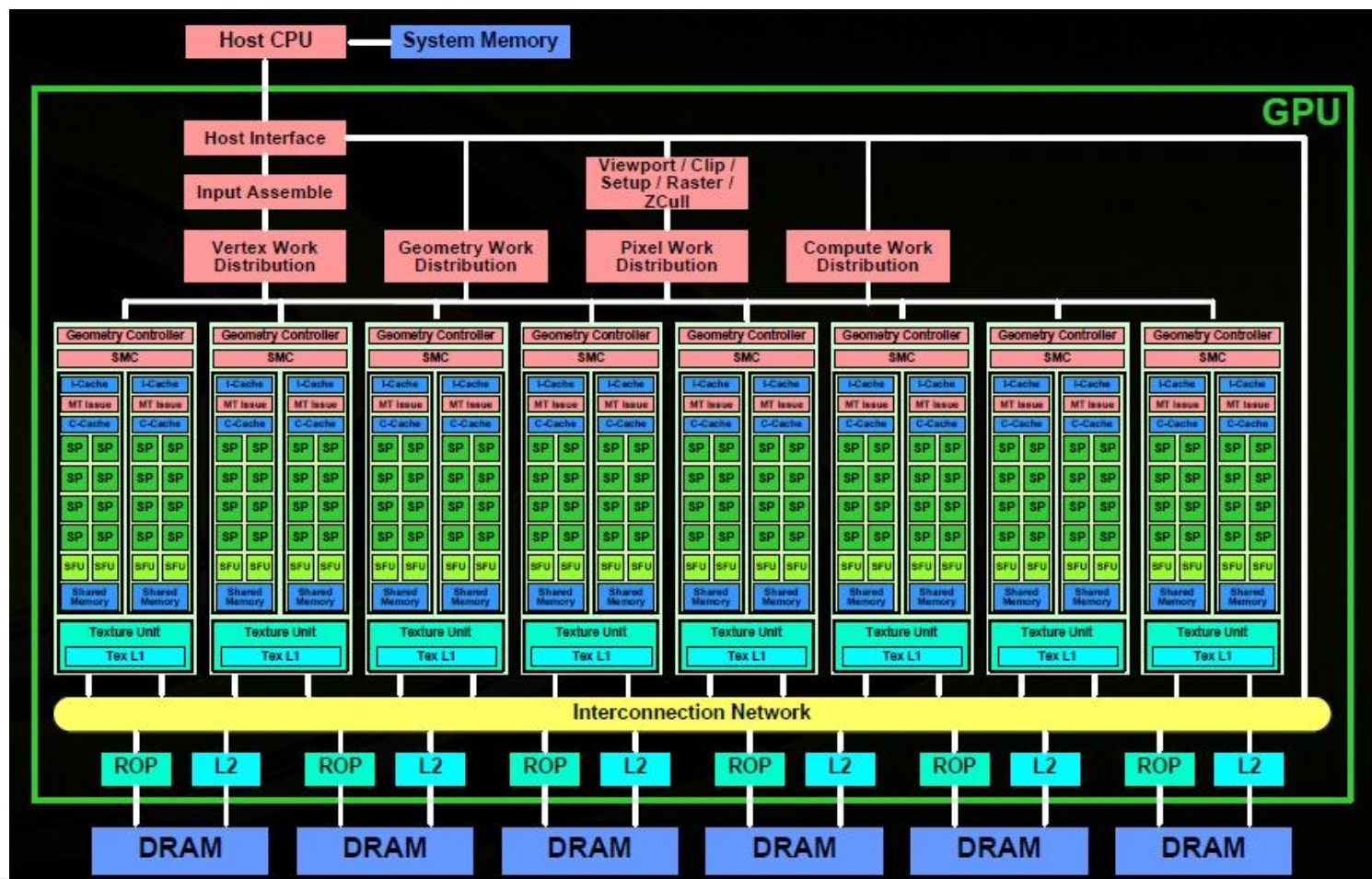
Streaming Multiprocessor (SM)



- 每个 SM 有 8 标量处理器(SP)
- 支持 IEEE 754 32位 浮点运算(不完全支持)
- 每个 SP 是一个 1.35 GHz 处理器 (峰值32 GFLOPS)
- 支持 32 和 64 位整型变量
- 8,192 动态分区的 32 位寄存器
- 硬件支持 768 线程(32 线程 , 24 SIMT线程束)
- 硬件上完成线程调度
- 16KB 的低延迟共享内存
- 2 个特殊功能单元(reciprocal square root, trig functions 等)



GPU



目录

- 异构并行计算
- CUDA C、CUDA Libs、OpenACC
- CUDA编程模型
- **内存分配及数据转移接口函数**
- 数据并行化与线程

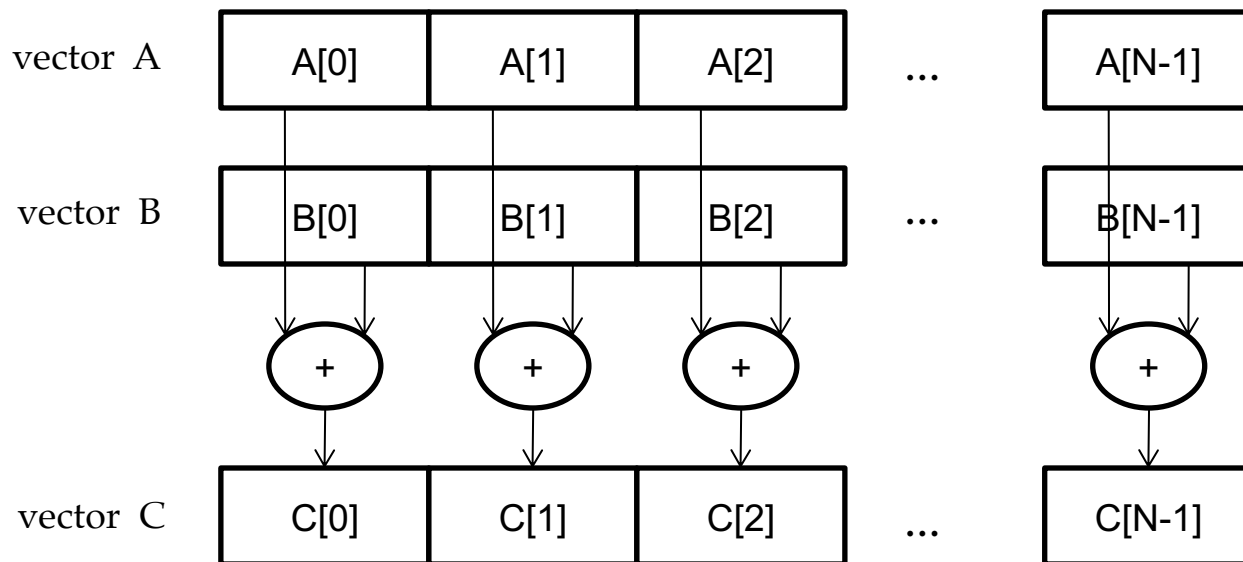


小节目标

- 学习 CUDA 主机代码中的基本 API 函数
 - 设备内存分配
 - 主机-设备间数据传输



数据并行- 向量加法



向量加法- 传统 C 代码

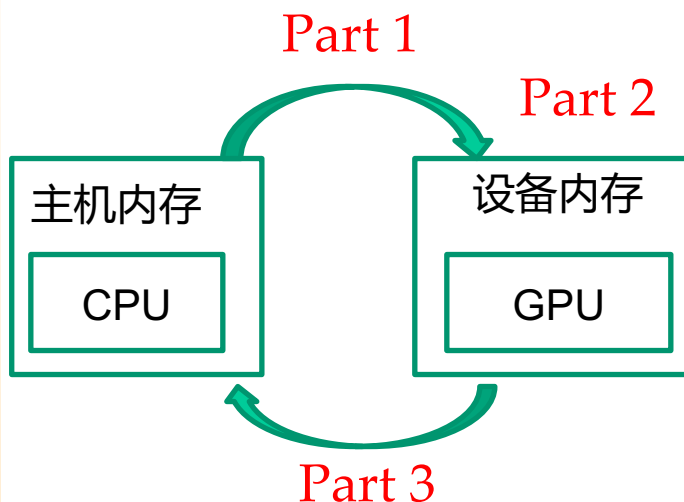
```
// Compute vector sum  $C = A + B$ 
void vecAdd(float *h_A, float *h_B, float *h_C, int n)
{
    int i;
    for (i = 0; i < n; i++) h_C[i] = h_A[i] + h_B[i];
}

int main()
{
    // Memory allocation for h_A, h_B, and h_C
    // I/O to read h_A and h_B, N elements
    ...

    (h_A, h_B, h_C, N);
}
```



异构计算 vecAdd CUDA 主机代码



```
#include <cuda.h>

void vecAdd(float *h_A, float *h_B, float *h_C, int n)
{
    int size = n* sizeof(float);
    float *d_A, *d_B, *d_C;

    // Part 1
    // Allocate device memory for A, B, and C
    // copy A and B to device memory

    // Part 2
    // Kernel launch code – the device performs the actual vector
    addition

    // Part 3
    // copy C from the device memory

}
```



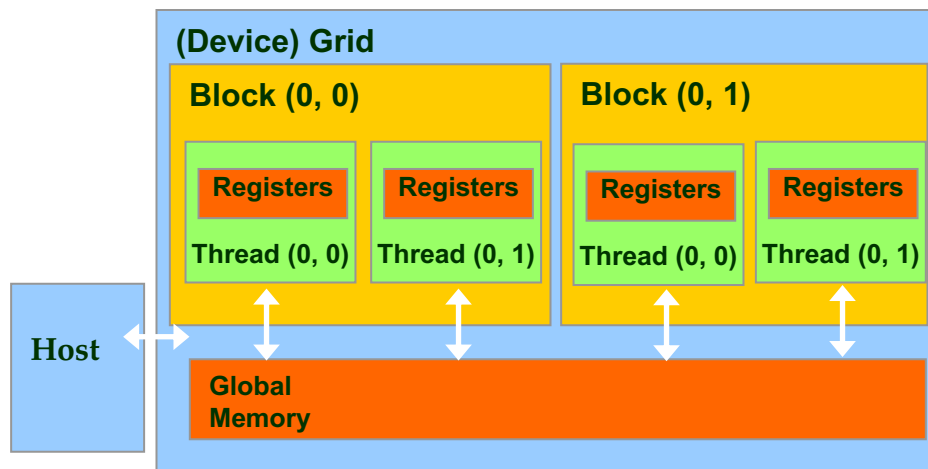
CUDA 内存的部分概述

- 设备代码能:

- 读/写 每个线程的寄存器
- 读写共享的全局内存

主机代码能

- 向/从线程网格的全局内存中转移数据



稍后我们将介绍更多的内存类型和更复杂的内存模型



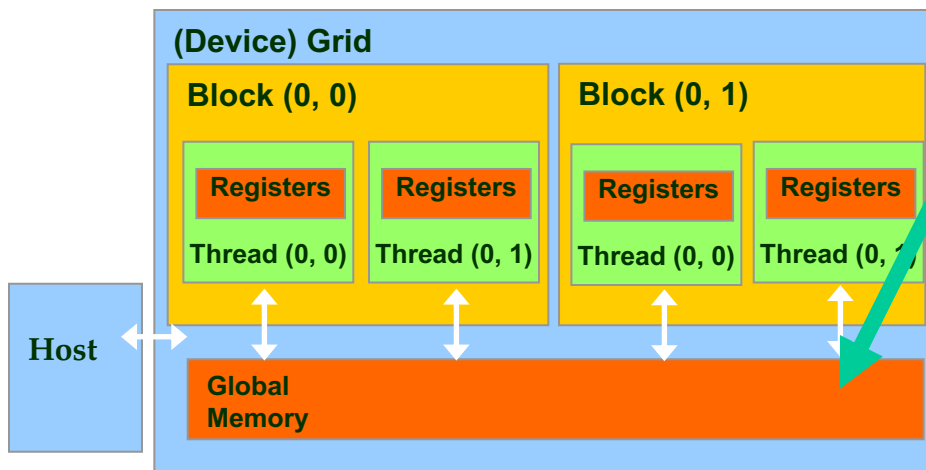
CUDA 设备内存管理 API 函数

- `cudaMalloc()`

- 在设备**全局内存**中分配一个对象
- 两个参数
 - 指向已分配对象的**指针的地址**
 - 已分配对象的**大小**，单位为 Bytes

- `cudaFree()`

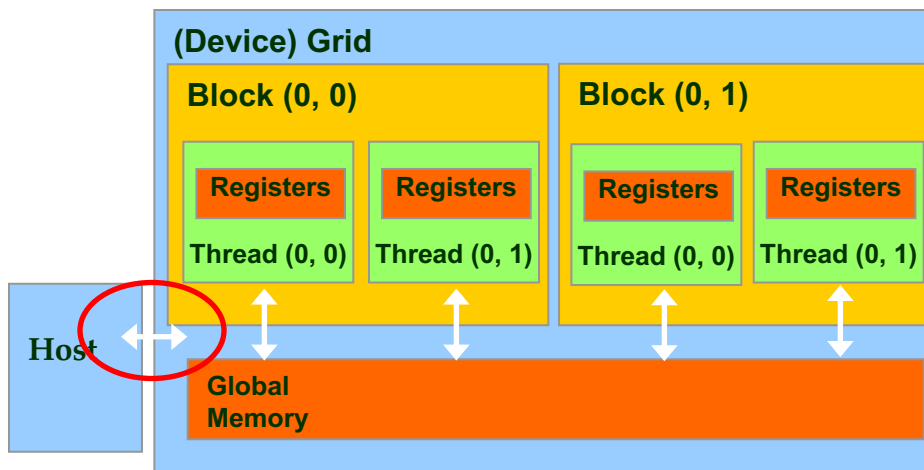
- 从设备全局内存中释放对象
- 一个参数
 - 指向被释放对象的**指针**



主机-设备数据转移 API 函数

- `cudaMemcpy()`

- 内存数据转移
- 四个参数
 - 指向目的地的指针
 - 指向源地址的指针
 - 转移数据的Bytes数
 - 转移的类型以及方向
- 传输到设备是异步的



向量加法 主机代码

```
void vecAdd(float *h_A, float *h_B, float *h_C, int n)
{
    int size = n * sizeof(float); float *d_A, *d_B, *d_C;

    cudaMalloc((void **) &d_A, size);
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMalloc((void **) &d_B, size);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
    cudaMalloc((void **) &d_C, size);

    // Kernel invocation code – to be shown later

    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
    cudaFree(d_A); cudaFree(d_B); cudaFree (d_C);
}
```



检查主机代码中的 API 错误

```
cudaError_t err = cudaMalloc((void **) &d_A, size);

if (err != cudaSuccess) {
    printf("%s in %s at line %d\n", cudaGetErrorString(err), __FILE__,
        __LINE__);
    exit(EXIT_FAILURE);
}
```


目录

- 异构并行计算
- CUDA C、CUDA Libs、OpenACC
- CUDA编程模型
- 内存分配及数据转移接口函数
- **数据并行化与线程**

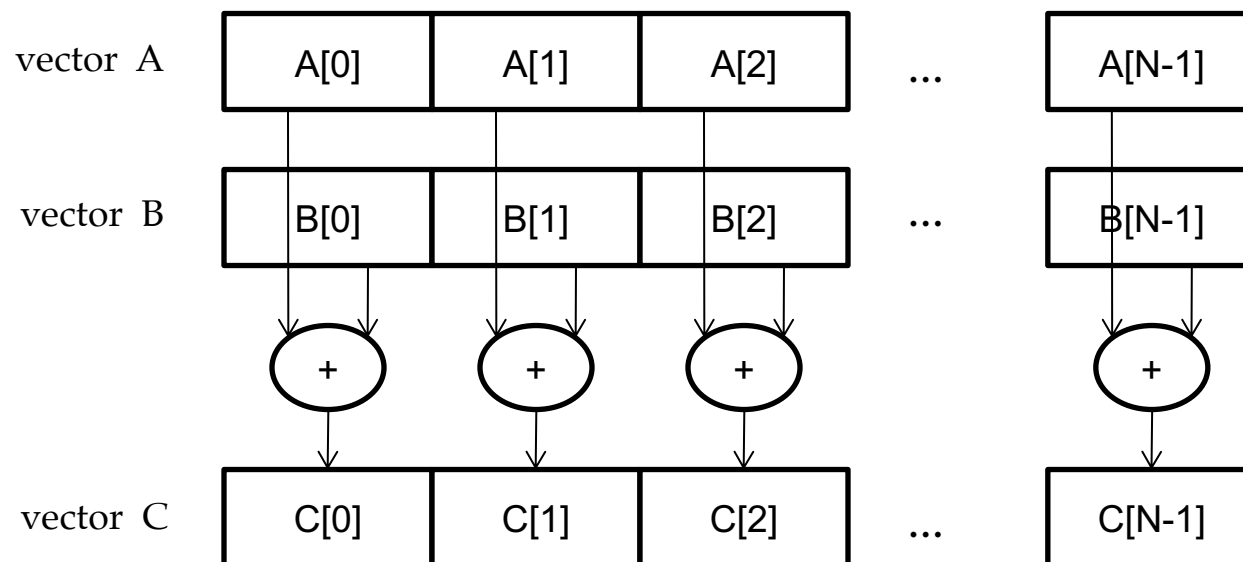


小节目标

- 了解 CUDA 线程，了解其利用数据并行性的主要机制
 - 分层线程组织
 - 启动并行执行
 - 线程索引到数据索引的映射

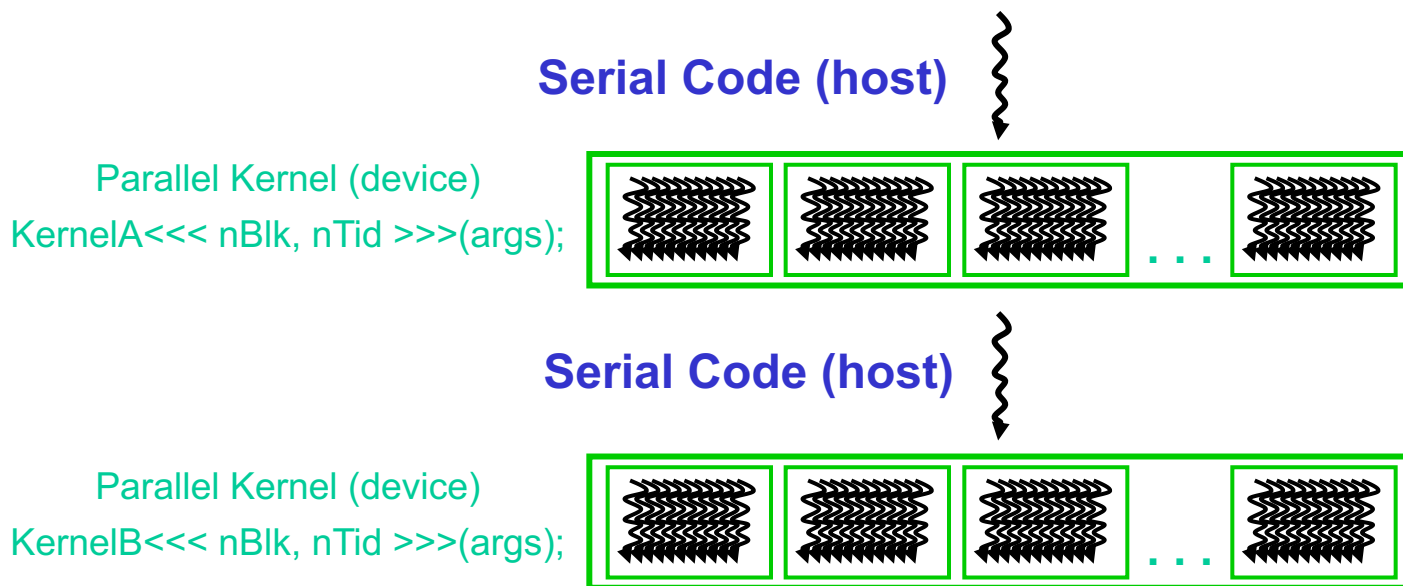


数据并行- 向量加法



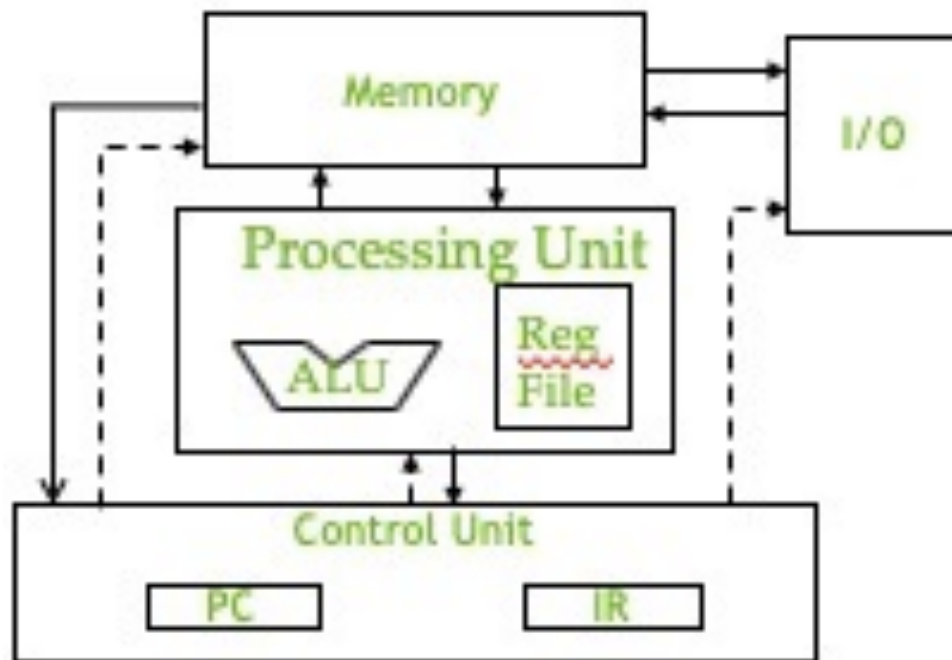
CUDA 执行模型

- 异构主机 (CPU) + 设备 (GPU) 应用 C 程序
 - 串行部分在**主机**的 C 程序中
 - 串行部分在**设备** SPMD 内核程序中



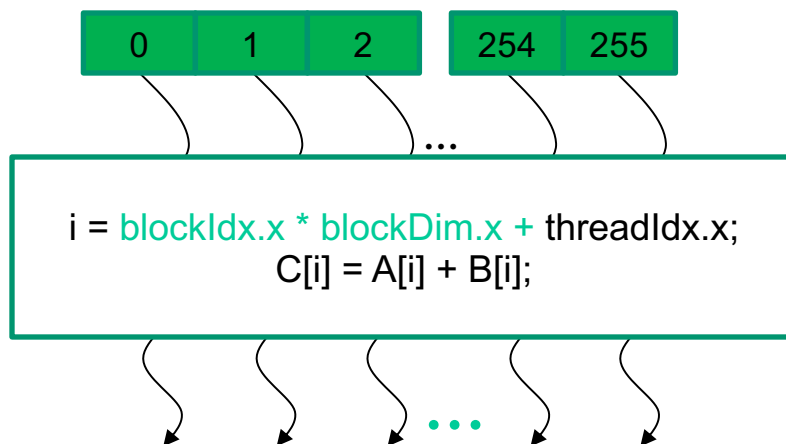
线程：Von-Neumann处理器

线程是“虚拟化”或“抽象”的
Von-Neumann处理器

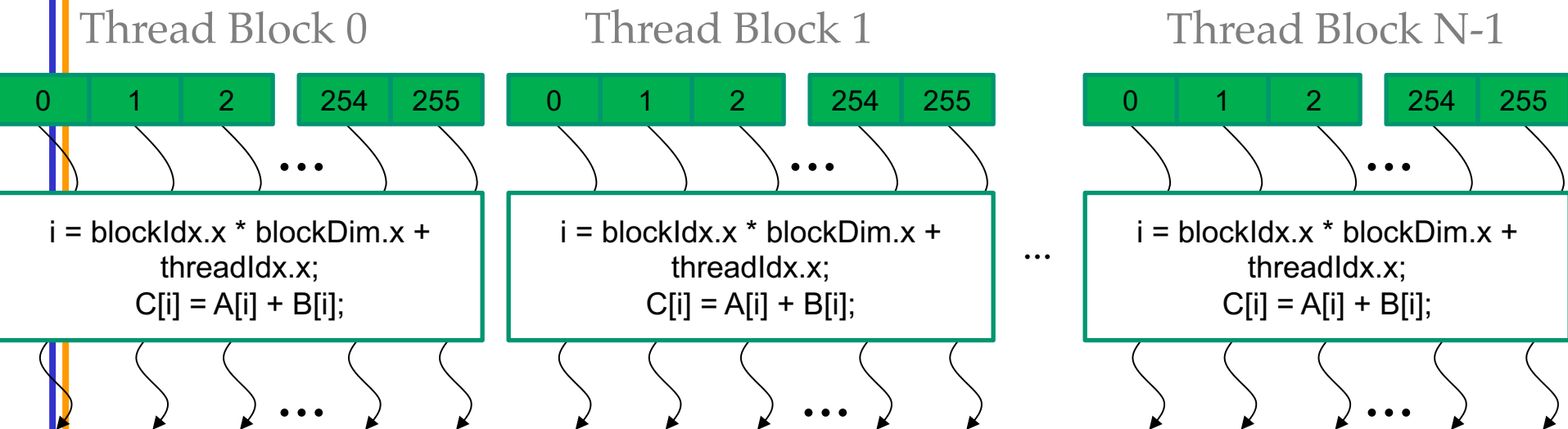


并行线程阵列

- CUDA 内核由线程网格（阵列）执行
 - 线程网格中的所有线程都运行相同的内核代码(SPMD , Single Program Multiple Data)
 - 每个线程都有用于计算内存地址和做出控制决策的索引



线程块: 可扩展的协作

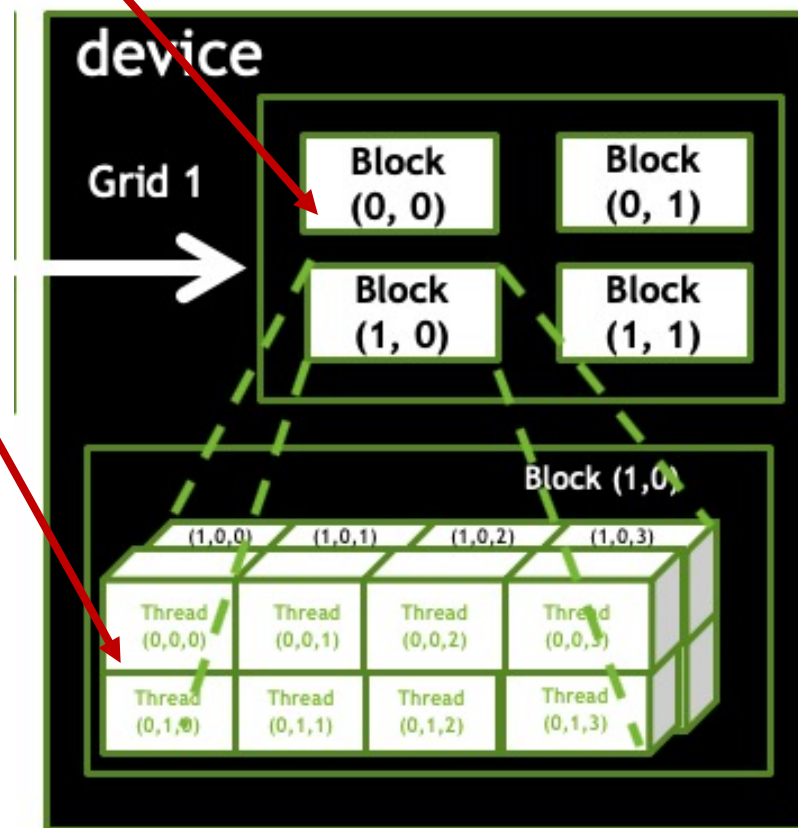


- 将线程阵列划分为多个线程块
 - 线程块内的线程通过**共享内存、原子操作和屏障同步**进行协作
 - **不同线程块中的线程不进行交互**



blockIdx 和 threadIdx

- 每个线程使用索引来决定处理哪些数据
 - **blockIdx**: 1D, 2D, or 3D (CUDA 4.0)
 - **threadIdx**: 1D, 2D, or 3D
- 处理多维数据时简化**内存寻址**
 - 图像处理
 - Solving PDEs on volumes
 - ...



NVCC 编译器

- NVIDIA 提供一个 CUDA-C 编译器
 - nvcc
- NVCC 编译设备代码，然后将代码转发到主机编译器（例如 g++）
- 可用于编译和链接仅限主机的应用程序

ANY QUESTIONS?

