深蓝学院
shenlanxueyuan.com

CUDA C编程
及GPU基本知识

课程目标

理论部分

- 学习认识GPU以及如何使用CUDA
- 如何编程和维护

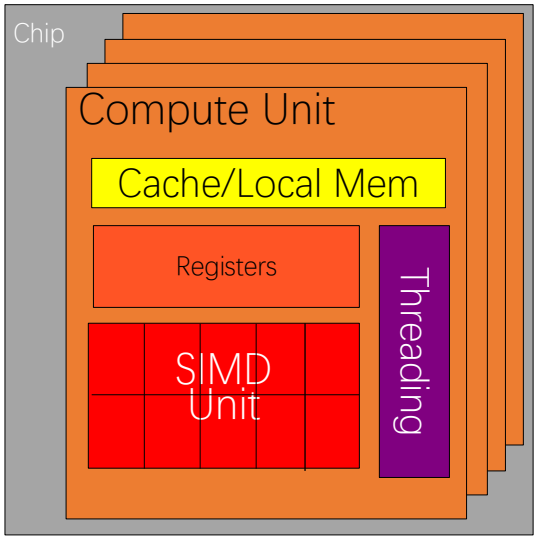
技能部分

- 并行计算的基本准则和样式
- 并行处理器特征和限制
- 使用方法

CPU和GPU架构

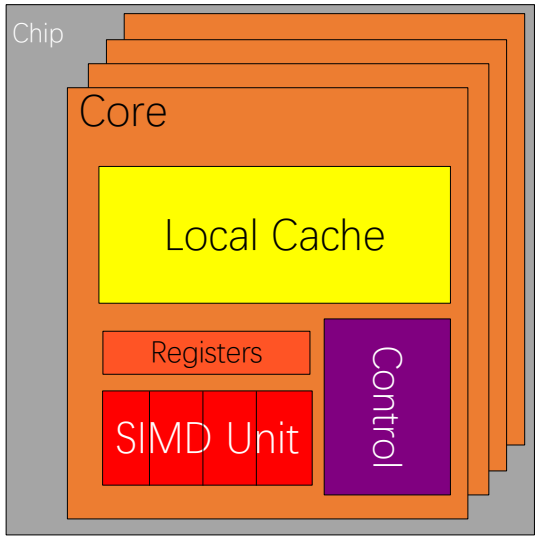
GPU

吞吐导向内核



CPU

延迟导向内核



补充：显卡、GPU和CUDA的联系

参考资料 https://blog.csdn.net/wu_nan_nan/article/details/45603299

CPUs: 延迟导向设计

内存大

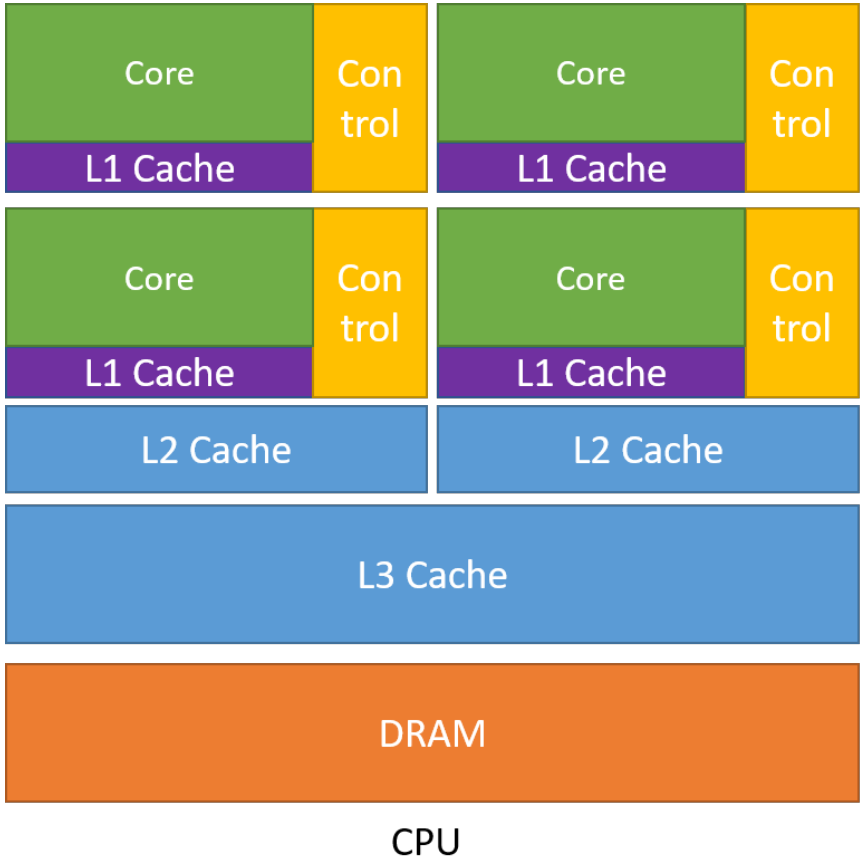
- 多级缓存结构提高访存速度

控制复杂

- 分支预测机制
- 流水线数据前送

运算单元强大

- 整型浮点型复杂运算速度快



GPUs: 吞吐导向设计

缓存小

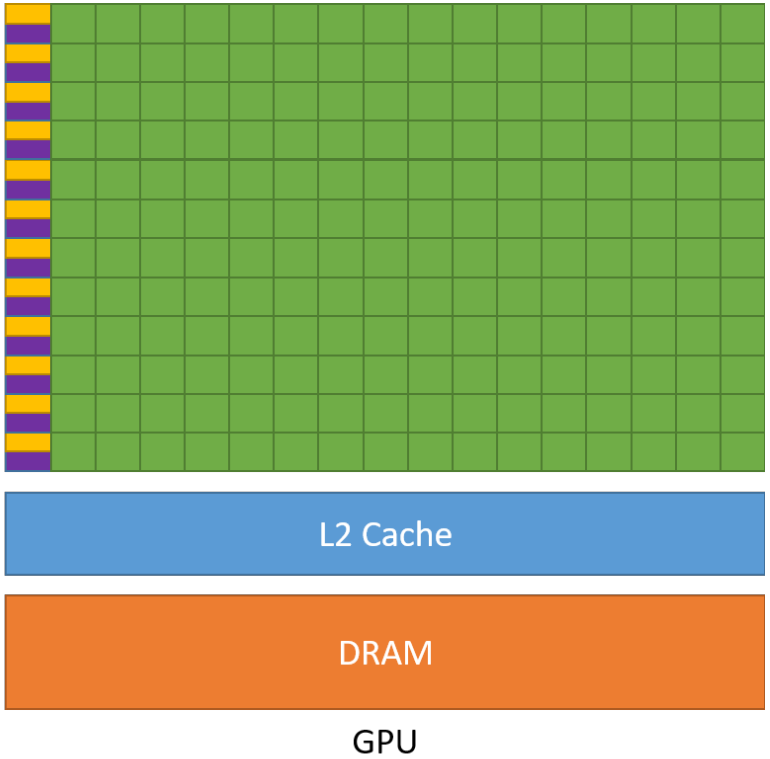
- 提高内存吞吐

控制简单

- 没有分支预测
- 没有数据转发

精简运算单元

- 多长延时流水线以实现高吞吐量
- 需要大量线程来容忍延迟



补充：显存与内存的联系

显存其实和内存一样，也是用来暂存资料的存储空间，不过显存是帮GPU存储的，而内存是帮CPU存储的。

GPU&CPU特点

CPUs: 连续计算部分，延迟优先

- CPU比 GPU ，单条复杂指令延迟快10倍以上

GPUs: 并行计算部分，吞吐优先

- GPU比 CPU ，单位时间内执行指令数量10倍以上

GPU编程：什么样的问题适合GPU

计算密集：数值计算的比例要远大于内存操作，因此内存访问的延时可以被计算掩盖。

数据并行： 大任务可以拆解为执行相同指令的小任务，因此对复杂流程控制的需求较低。

作业 1.1

CPU

（1）流水线前传机制

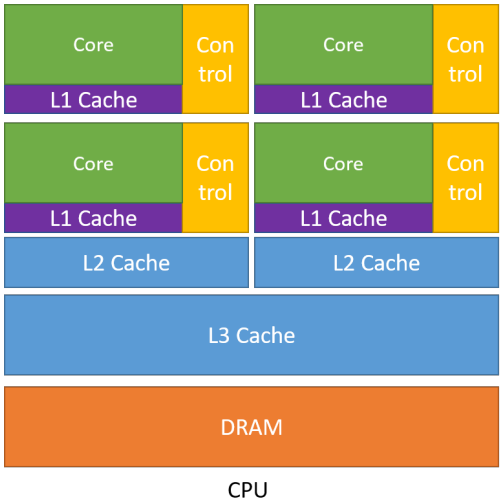
通过调研计算机体系结构的书籍，深入理解流水线前传机制，以及该机制如何使CPU的效率显著增加的。

（2）CPU的三级缓存

CPU中有L1 Cache(一级缓存)、L2 Cache(二级缓存)、L3 Cache(三级缓存)共3级缓存，这3级缓存的特点是什么，哪些内容适合放在哪一级别的缓存上。

（3）什么样的问题适合GPU

思考自己日常编程解决的任务中，哪些适合交给GPU处理。



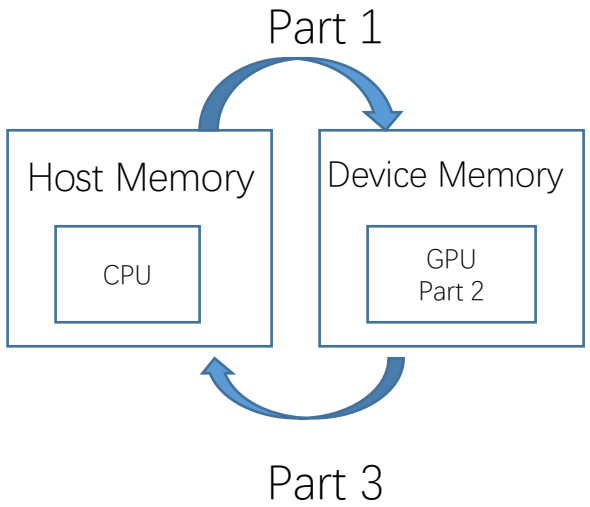
GPU编程与CUDA

CUDA（Compute Unified Device Architecture），由英伟达公司2007年开始推出，初衷是为GPU增加一个易用的编程接口，让开发者无需学习复杂的着色语言或者图形处理原语。

OpenCL（Open Computing Languge）是2008年发布的异构平台并行编程的开放标准，也是一个编程框架。OpenCL相比CUDA，支持的平台更多，除了GPU还支持CPU、DSP、FPGA等设备。

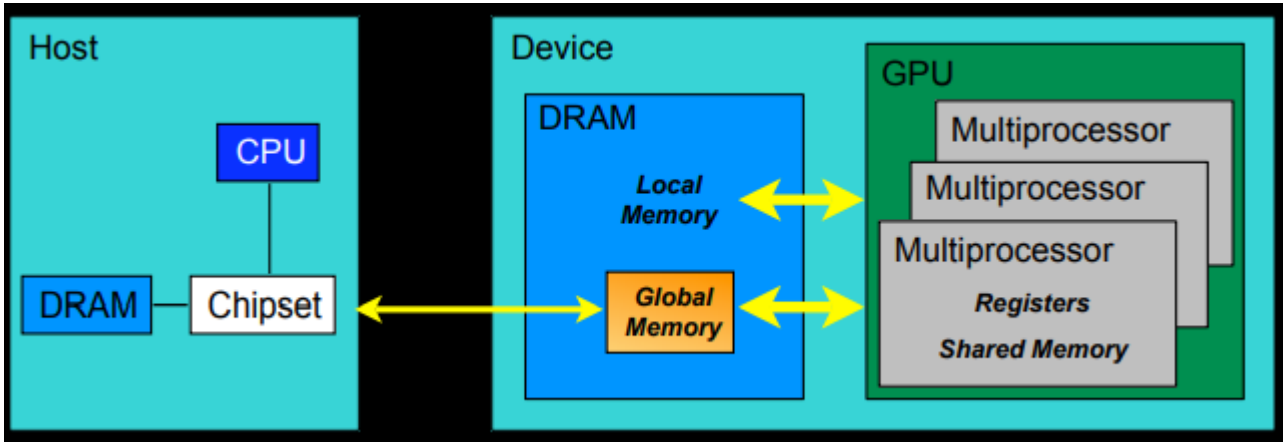
CUDA编程并行计算整体流程

```
void GPUkernel(float* A, float* B, float* C, int n)
{
1. // Allocate device memory for A, B, and C
   // copy A and B to device memory

2. // Kernel launch code – to have the device
   // to perform the actual vector addition

3. // copy C from the device memory
   // Free device vectors
}
```



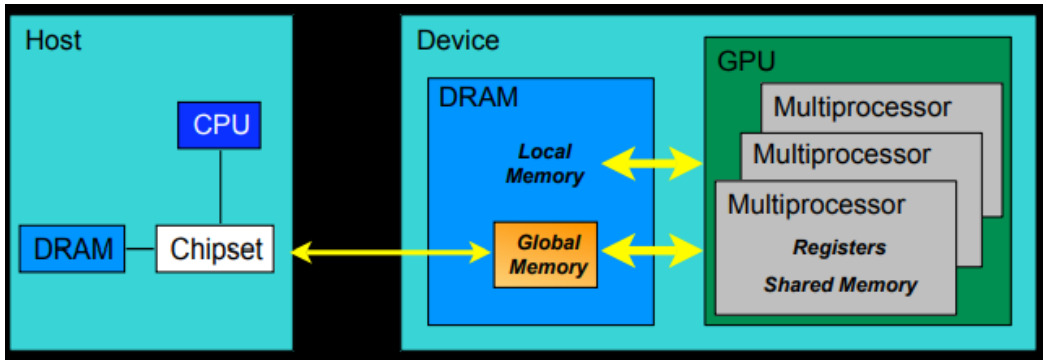
CUDA编程术语：硬件

- Device=GPU
- Host=CPU
- Kernel=GPU上运行的函数



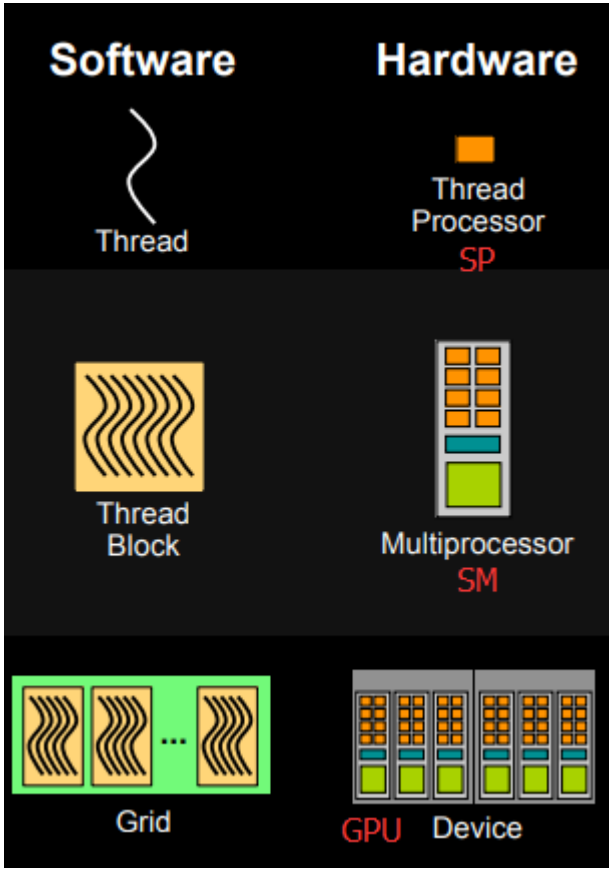
CUDA编程术语：内存模型

- CUDA中的内存模型分为以下几个层次：
  - 每个线程处理器（SP）都用自己的registers（寄存器）
  - 每个SP都有自己的local memory（局部内存），register和local memory只能被线程自己访问
  - 每个多核处理器（SM）内都有自己的shared memory（共享内存），shared memory 可以被线程块内所有线程访问
  - 一个GPU的所有SM共有一块global memory（全局内存），不同线程块的线程都可使用



CUDA编程术语：软件

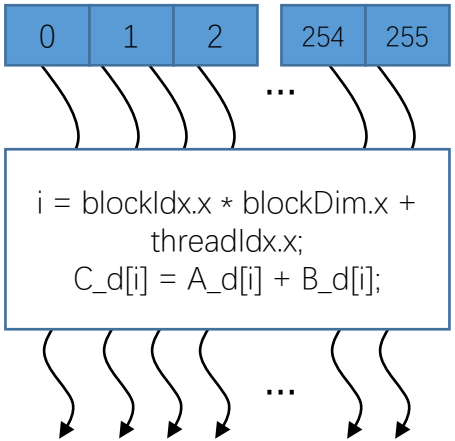
- CUDA中的内存模型分为以下几个层次：
  - 线程处理器（SP）对应线程（thread）
  - 多核处理器（SM）对应线程块（thread block）
  - 设备端（device）对应线程块组合体（grid）
- 一个kernel其实由一个grid来执行
- 一个kernel一次只能在一个GPU上执行



线程块：可扩展的集合体

将线程数组分成多个块

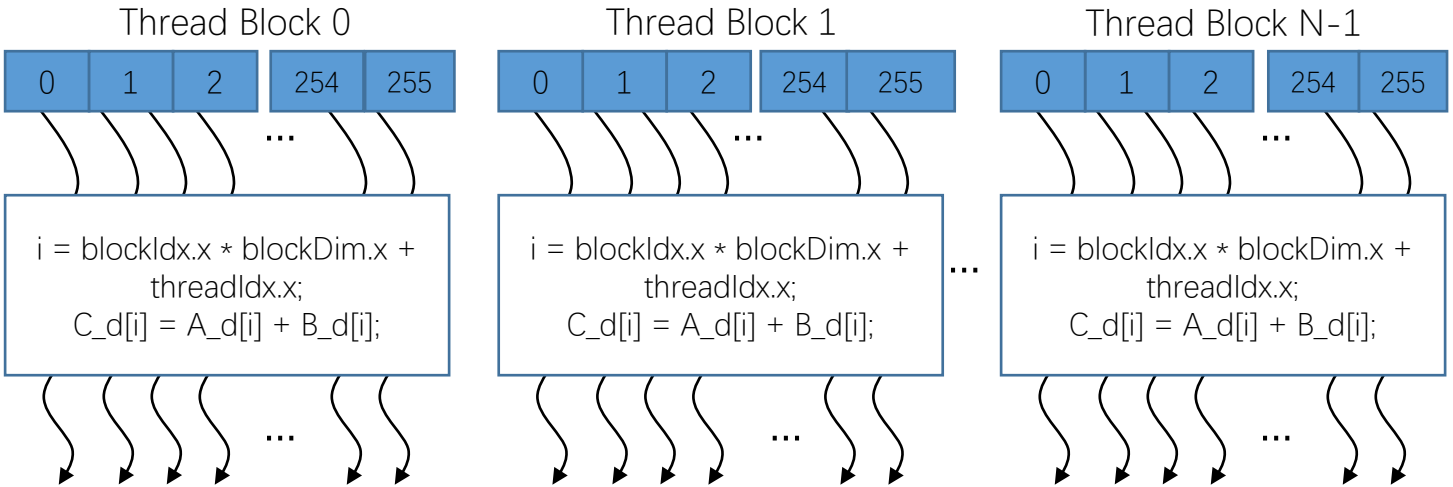
- 块内的线程通过共享内存、原子操作和屏障同步进行协作（shared memory, atomic operations and barrier synchronization）
- 不同块中的线程不能协作



网格（grid）：并行线程块组合

CUDA 核函数由线程网格（数组）执行

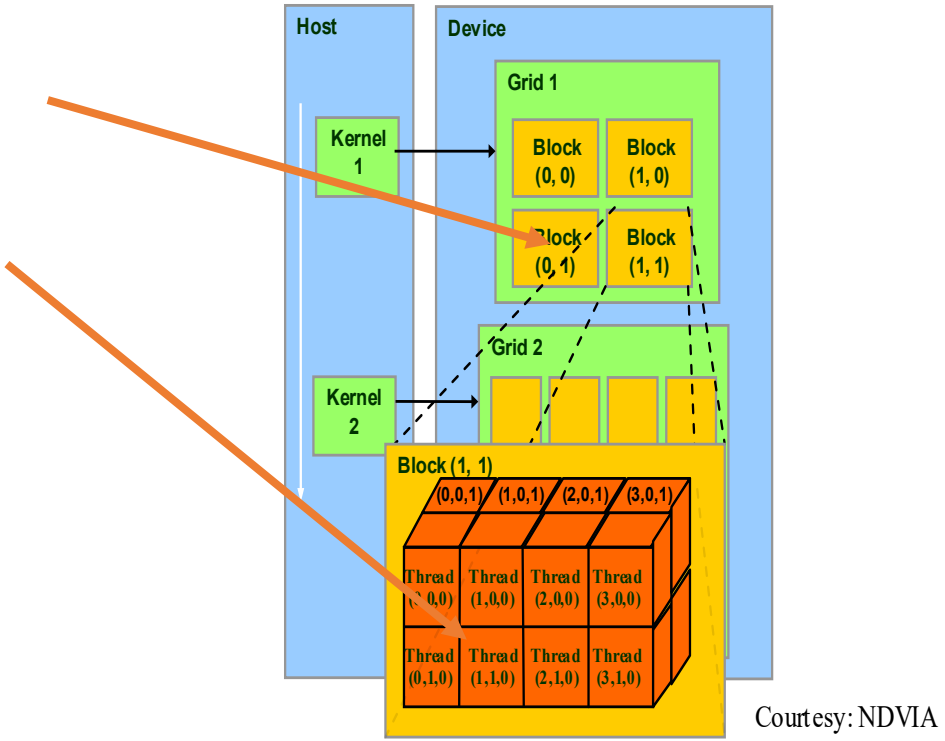
– 每个线程都有一个索引，用于计算内存地址和做出控制决策



线程块id&线程id：定位独立线程的门牌号

每个线程使用索引来决定要处理的数据

- `blockIdx: 1D, 2D, or 3D`
- `threadIdx: 1D, 2D, or 3D`

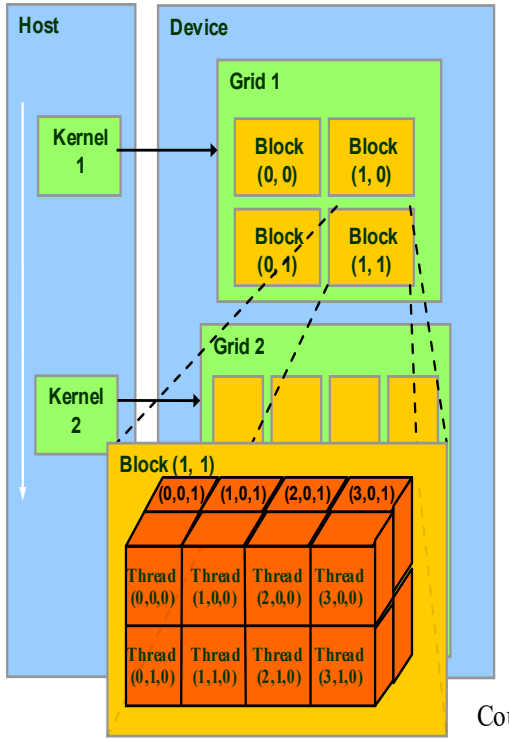


Courtesy: NDVIA

线程id计算

- `dim3 dimGrid(M, N);`
- `dim3 dimBlock(P, Q, S);`

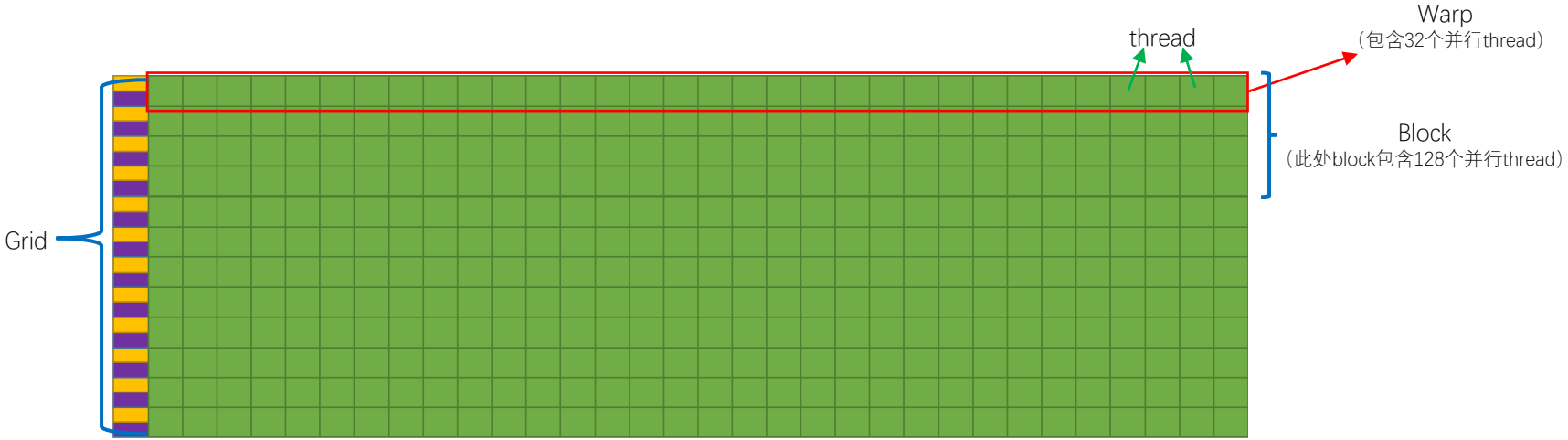
- `threadId.x = blockIdx.x*blockDim.x+ threadIdx.x;`
- `threadId.y = blockIdx.y*blockDim.y+ threadIdx.y;`



Courtesy: NDVIA

线程束（warp）

- SM采用的SIMT(Single-Instruction, Multiple-Thread，单指令多线程)架构，warp(线程束)是最基本的执行单元，一个warp包含32个并行thread，这些thread**以不同数据资源执行相同的指令。**warp本质上是线程在GPU上运行的最小单元。
- 当一个kernel被执行时，grid中的线程块被分配到SM上，**一个线程块的thread只能在一个SM上调度**，SM一般可以调度多个线程块，大量的thread可能被分到不同的SM上。每个thread拥有它自己的程序计数器和状态寄存器，并且用该线程自己的数据执行指令，这就是所谓的Single Instruction Multiple Thread(SIMT)。
- **由于warp的大小为32，所以block所含的thread的大小一般要设置为32的倍数。**

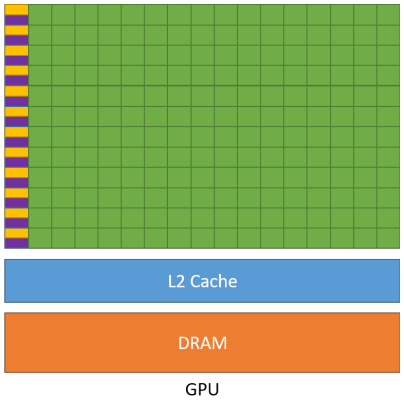


作业 1.2

GPU

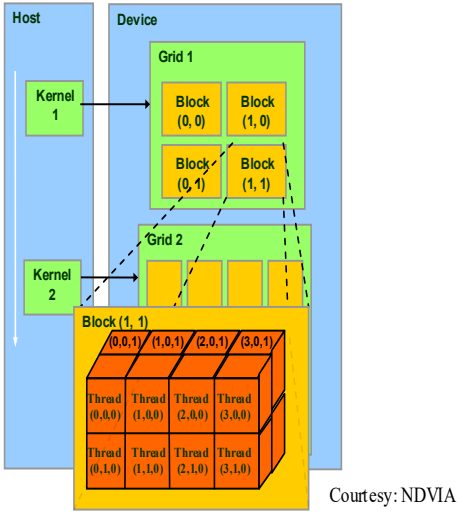
（1）线程束warp

GPU的控制单元与计算单元是如何结合的，或者说warp线程束是如何在软件和硬件端被执行的，为什么说线程束是执行核函数的最基本单元。

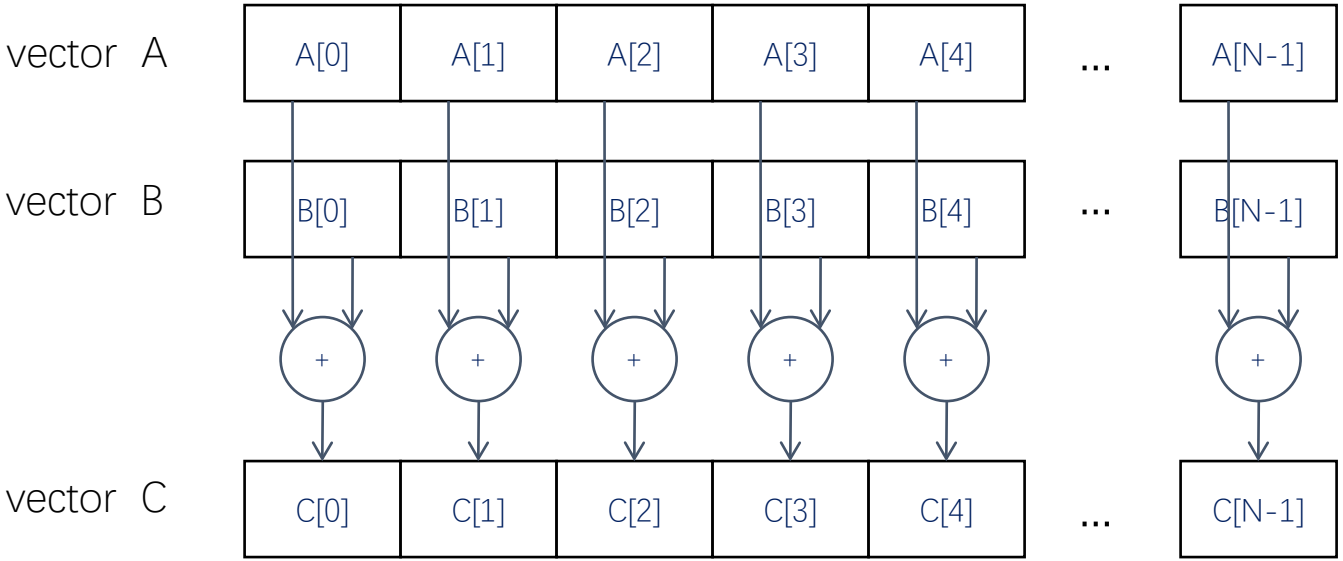


（2）线程ID

计算下图中Thread(3,0,0)的线程ID。



并行计算实例：向量相加


vector A
A[0]
A[1]
A[2]
A[3]
A[4]
...
A[N-1]
vector B
B[0]
B[1]
B[2]
B[3]
B[4]
...
B[N-1]
+
+
+
+
+
+
vector C
C[0]
C[1]
C[2]
C[3]
C[4]
...
C[N-1]

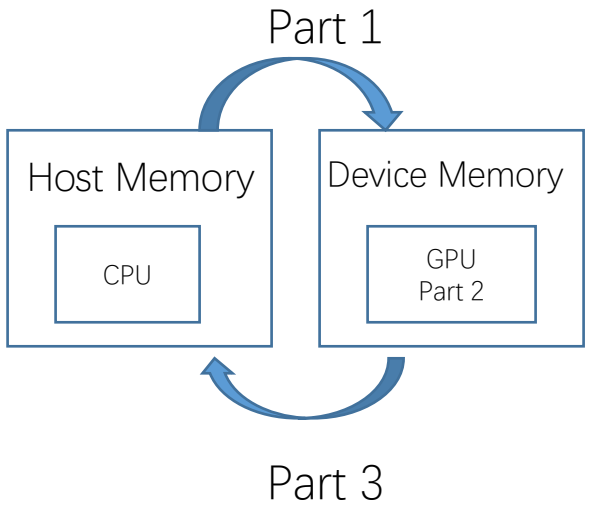
并行计算实例：向量相加

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

int main()
{
  // Memory allocation for A_h, B_h, and C_h
    // I/O to read A_h and B_h, N elements
    …
    vecAdd(A_h, B_h, C_h, N);
}
```

CPU中数组相加

并行计算实例：向量相加

```
#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;
    …
1.  // Allocate device memory for A, B, and C
    // copy A and B to device memory

2.  // Kernel launch code – to have the device
    // to perform the actual vector addition

3.  // copy C from the device memory
    // Free device vectors
}
```



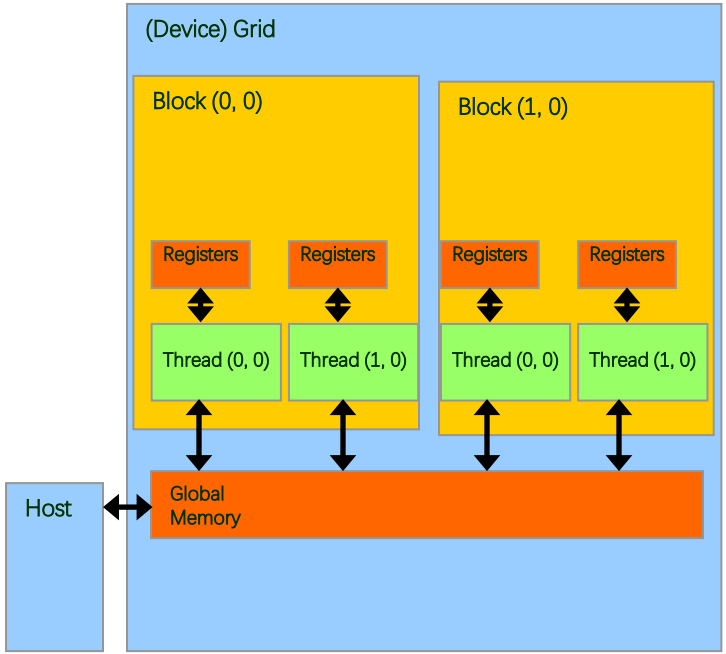
并行计算实例：向量相加

设备端代码:

- 读写线程寄存器
- 读写Grid中全局内存
- 读写block中共享内存

主机端代码:

- Grid中全局内存拷贝转移



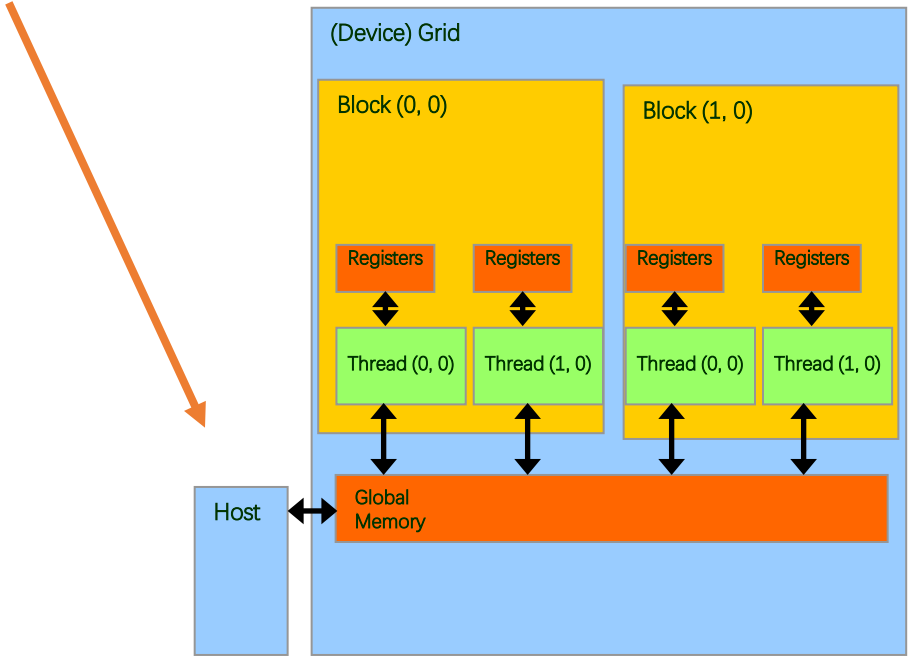
并行计算实例：向量相加

cudaMalloc( )

- cudaError_t cudaMalloc (void **devPtr, size_t size)
- 在设备全局内存中分配对象
- 两个参数
  - 地址
  - 申请内存大小

cudaFree( )

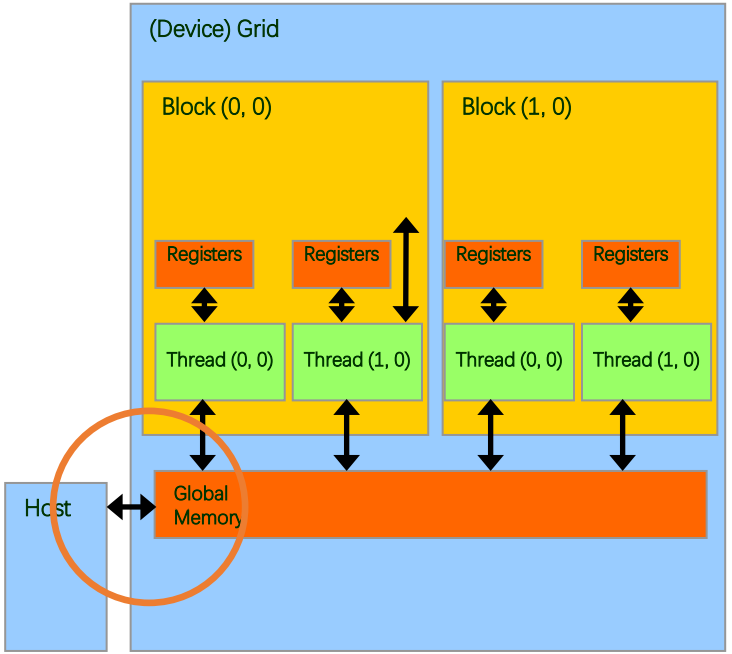
- cudaError_t cudaFree (void* devPtr)
- 从设备全局内存中释放对象
- 指向释放对象的指针



并行计算实例：向量相加

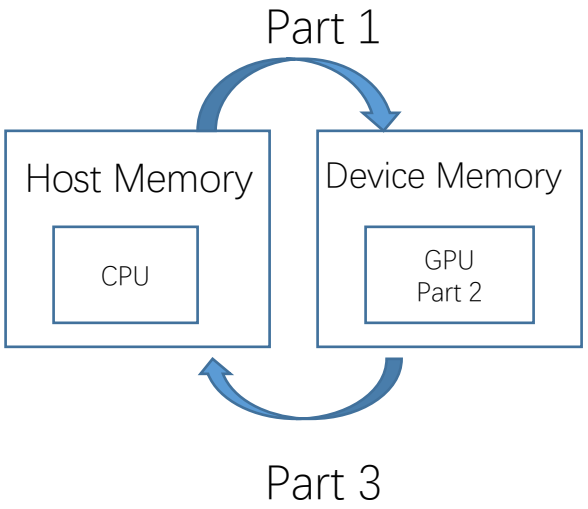
cudaMemcpy( )

- cudaError_t cudaMemcpy (void *dst, const void *src, size_t count, cudaMemcpyKind kind)
- 内存数据复制传递
- 目前支持的四种选项
  - cudaMemcpyHostToDevice
  - cudaMemcpyDeviceToHost
  - cudaMemcpyDeviceToDevice
  - cudaMemcpyDefault
- 调用cudaMemcpy( )传输内存是同步的

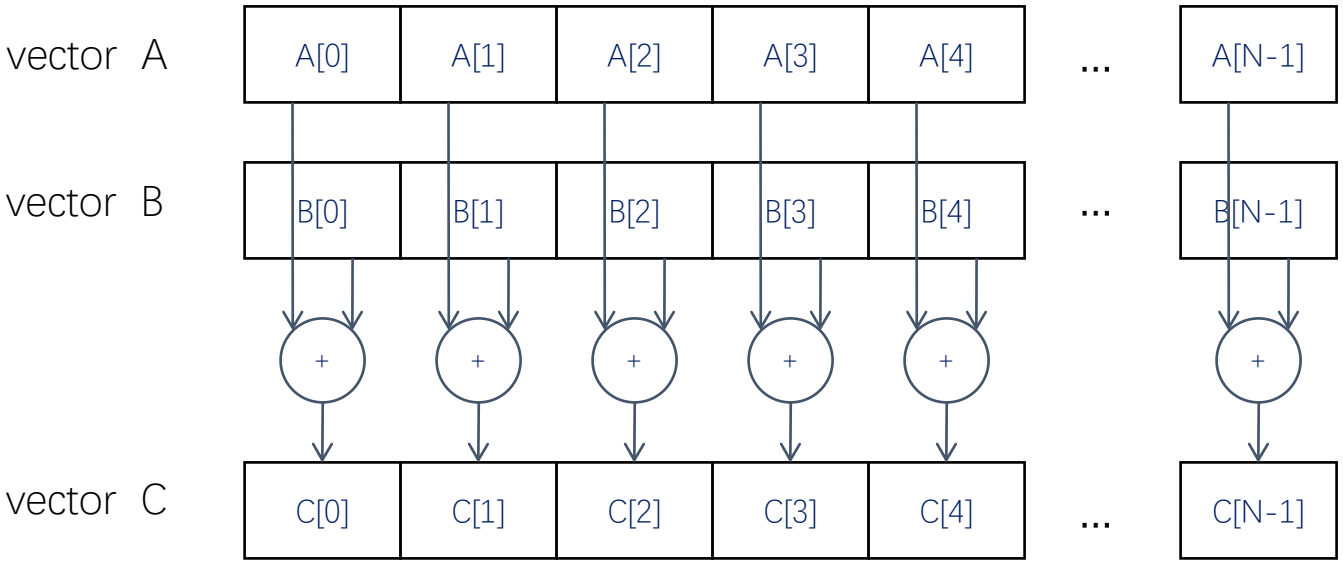


并行计算实例：向量相加

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

1. // Transfer A and B to device memory
     cudaMalloc((void **) &A_d, size);
     cudaMemcpy(A_d, A, size, cudaMemcpyHostToDevice);
     cudaMalloc((void **) &B_d, size);
     cudaMemcpy(B_d, B, size, cudaMemcpyHostToDevice);

      // Allocate device memory for
      cudaMalloc((void **) &C_d, size);

2.    // Kernel invocation code – to be shown later
      …
3.     // Transfer C from device to host
      cudaMemcpy(C, C_d, size, cudaMemcpyDeviceToHost);
        // Free device memory for A, B, C
      cudaFree(A_d); cudaFree(B_d); cudaFree (C_d);
}
```



并行计算实例：向量相加


vector A
A[0]
A[1]
A[2]
A[3]
A[4]
...
A[N-1]
vector B
B[0]
B[1]
B[2]
B[3]
B[4]
...
B[N-1]
+
vector C
C[0]
C[1]
C[2]
C[3]
C[4]
...
C[N-1]

并行计算实例：向量相加

核函数调用

- 在GPU上执行的函数。
- 一般通过标识符__global__修饰。
- 调用通过<<<参数1,参数2>>>，用于说明内核函数中的线程数量，以及线程是如何组织的。
- 以网格（Grid）的形式组织，每个线程格由若干个线程块（block）组成，而每个线程块又由若干个线程（thread）组成。
- 调用时必须声明内核函数的执行参数。
- 在编程时，必须先为kernel函数中用到的数组或变量分配好足够的空间，再调用kernel函数，否则在GPU计算时会发生错误。

CUDA编程标识符号

__global__ 标志核函数

- 核函数必须返回 void

__device__ & __host__ 可以一起用

	工作地点	被调用地点
`__device__ float DeviceFunc()`	device	device
`__global__ void KernelFunc()`	device	host
`__host__ float HostFunc()`	host	host

并行计算实例：向量相加

Device Code

```
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition
__global__
void vecAddKernel(float* A_d, float* B_d, float* C_d, int n)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if(i<n) C_d[i] = A_d[i] + B_d[i];
}

int vectAdd(float* A, float* B, float* C, int n)
{
    // A_d, B_d, C_d allocations and copies omitted
    // Run ceil(n/256) blocks of 256 threads each
    vecAddKernel<<<ceil(n/256), 256>>>(A_d, B_d, C_d, n);
}
```

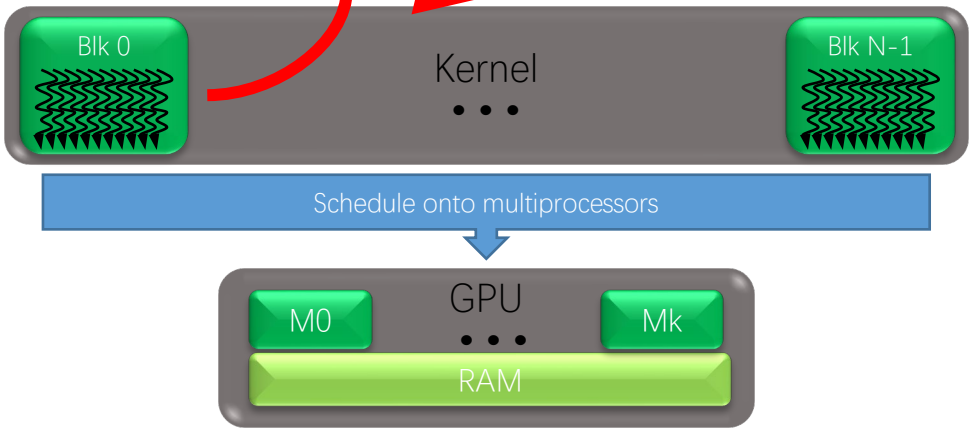
GPU中thread id
可类比CPU中数组位置

Host Code

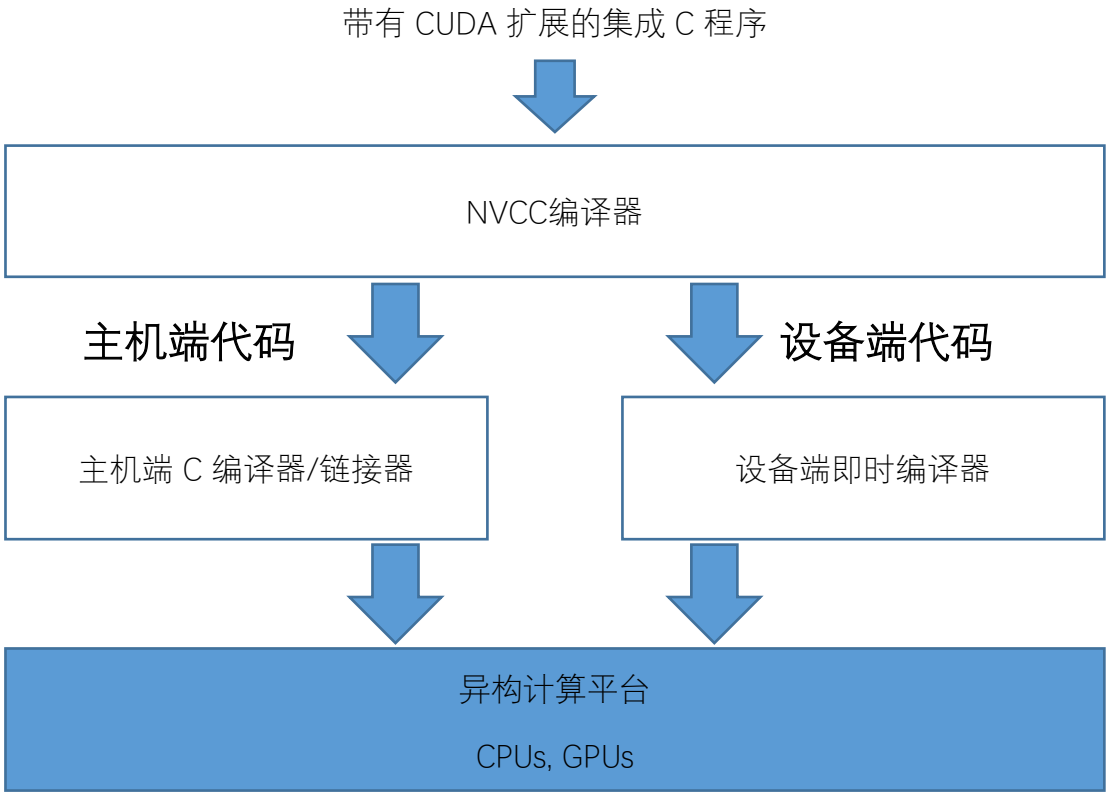
并行计算实例：向量相加

```
__host__
Void vecAdd()
{
  dim3 DimGrid = (ceil(n/256),1,1);
  dim3 DimBlock = (256,1,1);
 vecAddKernel<<<DimGrid,DimBlock>>>
(A_d,B_d,C_d,n);
}
```

```
__global__
void vecAddKernel(float *A_d,
     float *B_d, float *C_d, int n)
{
   int i = blockIdx.x * blockDim.x
           + threadIdx.x;

   if( i<n ) C_d[i] = A_d[i]+B_d[i];
}
```



CUDA编程流程


带有 CUDA 扩展的集成 C 程序
NVCC编译器
主机端代码
设备端代码
主机端 C 编译器/链接器
设备端即时编译器
异构计算平台
CPUs, GPUs

参考资料

1. D. Kirk and W. Hwu, “Programming Massively Parallel Processors – A Hands-on Approach, Second Edition”

2. CUDA by example, *Sanders and Kandrot*

3. *Nvidia CUDA C Programming Guide*

   https://docs.nvidia.com/cuda/cuda-c-programming-guide/

4. CS/EE217 GPU Architecture and Programming

深蓝学院
shenlanxueyuan.com

感谢聆听！

Thanks for Listening