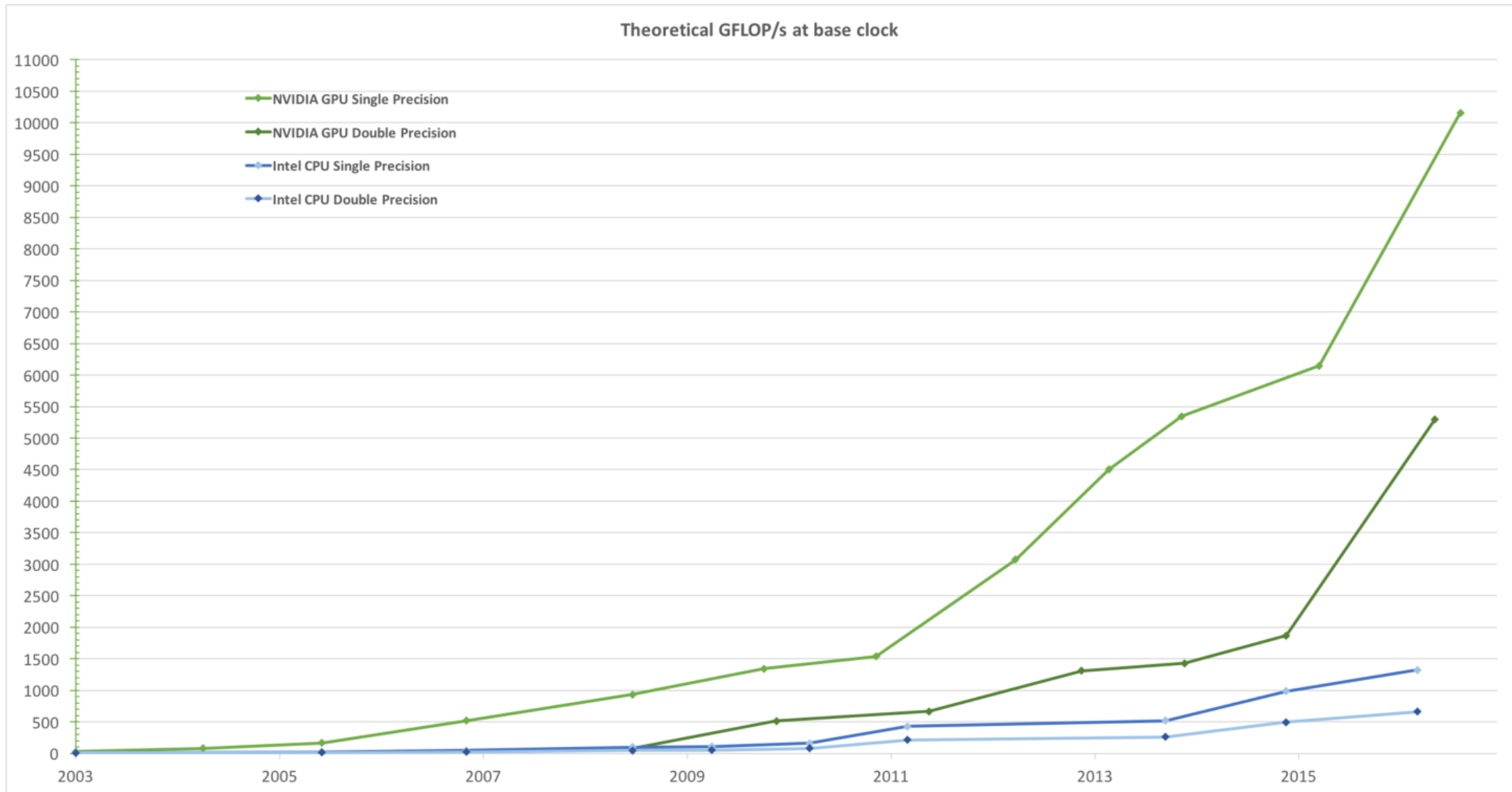
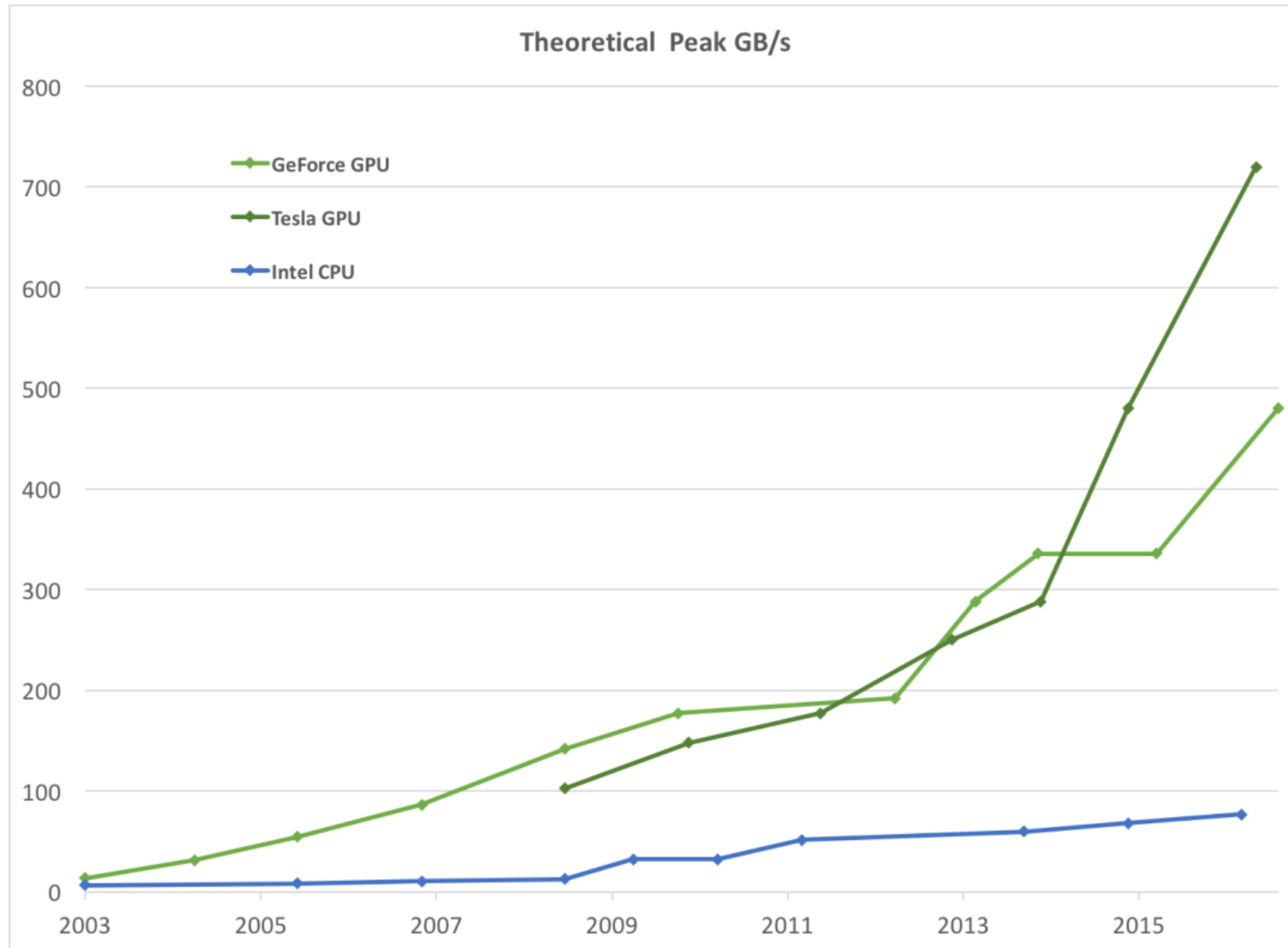


GPU的CUDA编程方法

Floating-Point Operations per Second for the CPU and GPU



Memory Bandwidth for the CPU and GPU

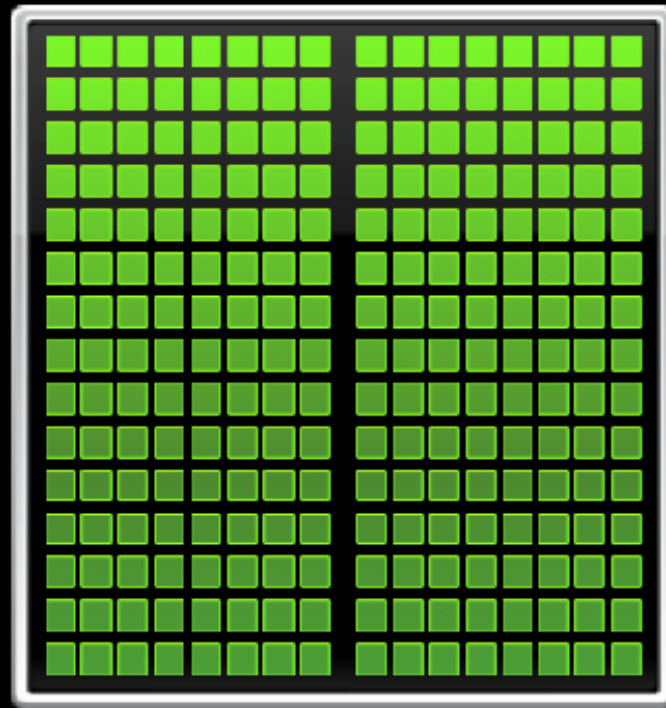


Why a GPU?

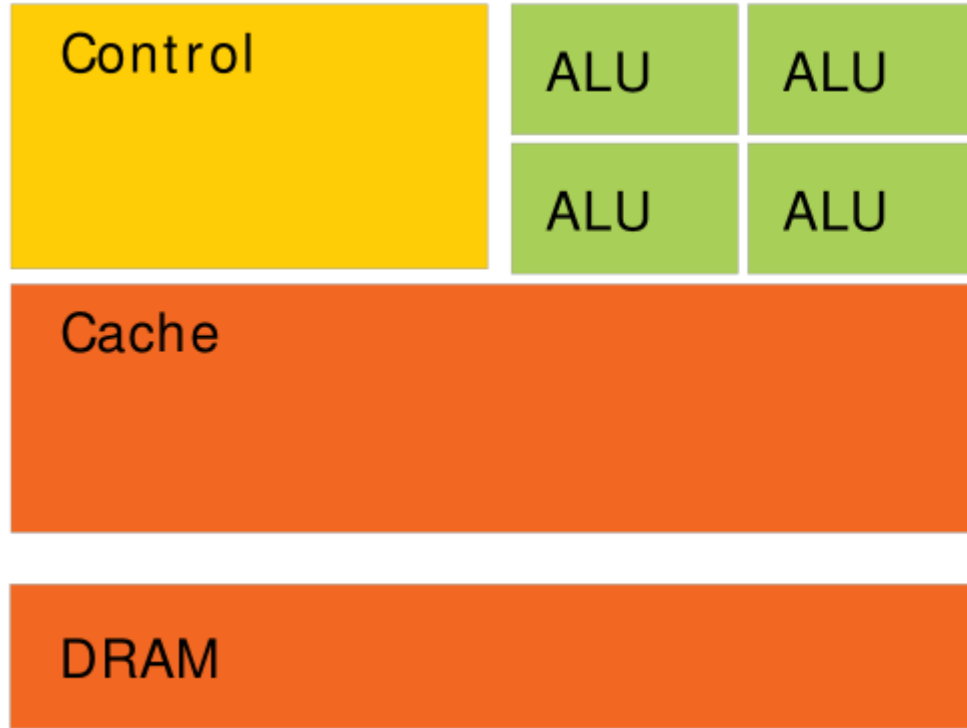
CPU
Optimized for
Serial Tasks



GPU
Optimized for Many
Parallel Tasks



The GPU Devotes More Transistors to Data Processing

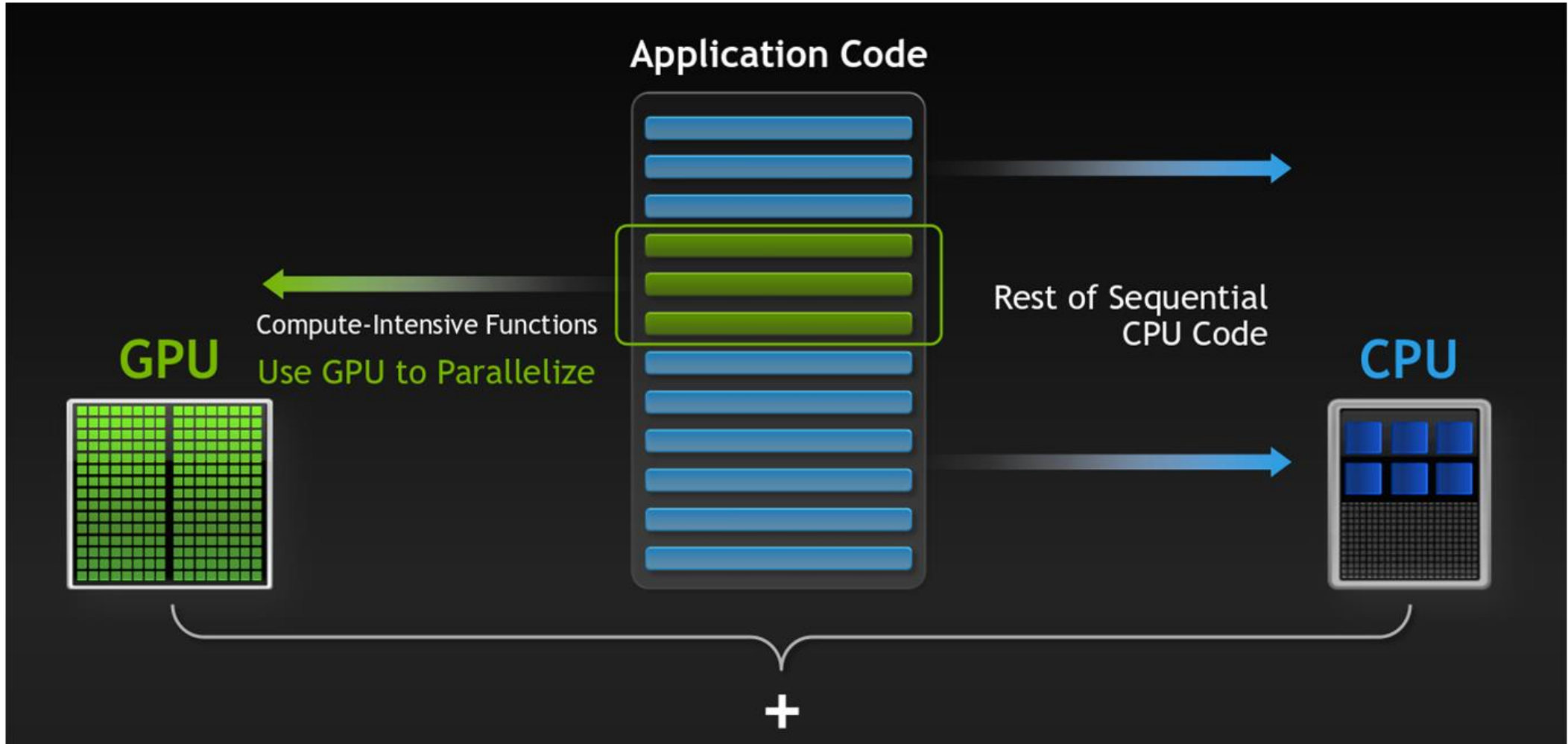


CPU



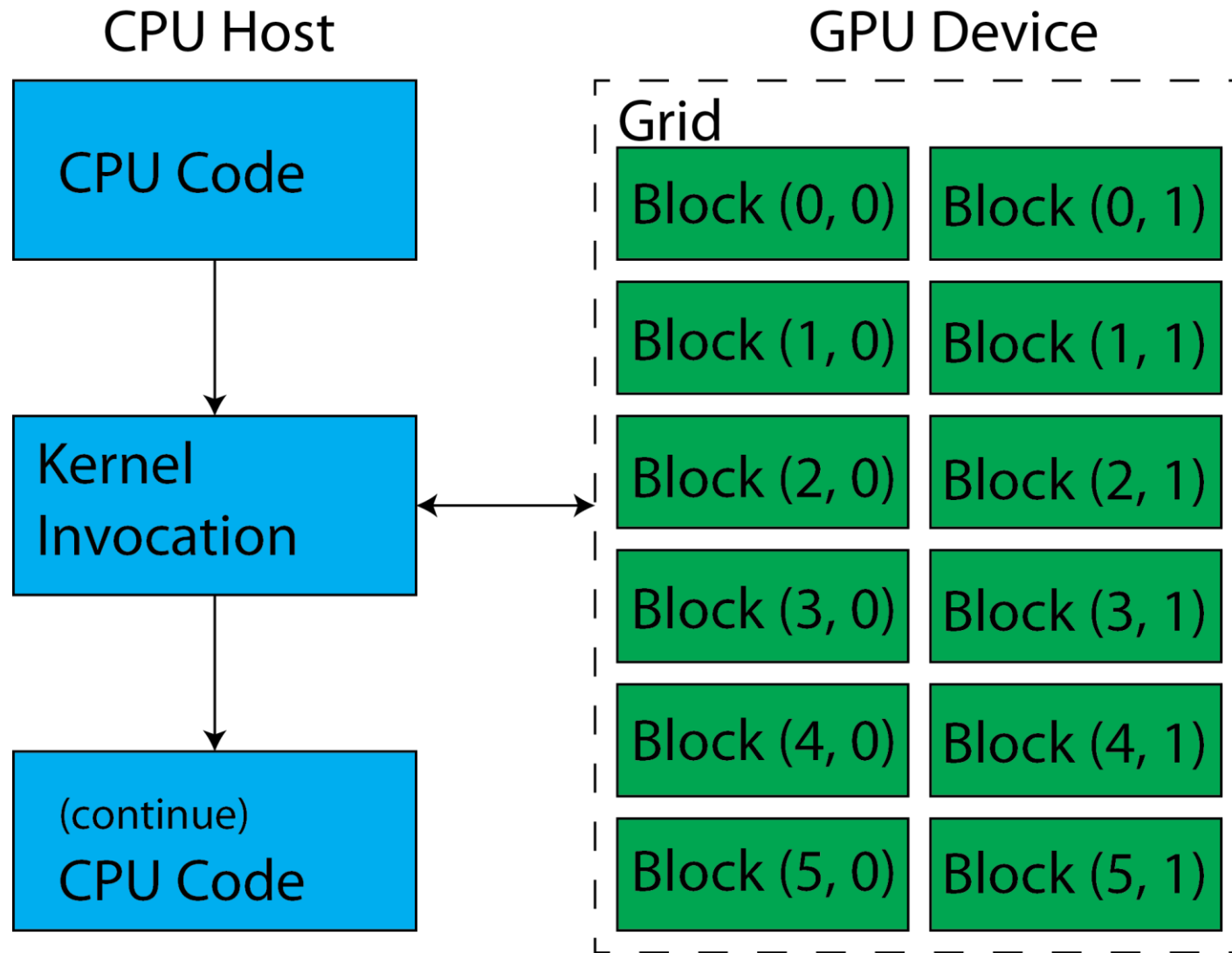
GPU

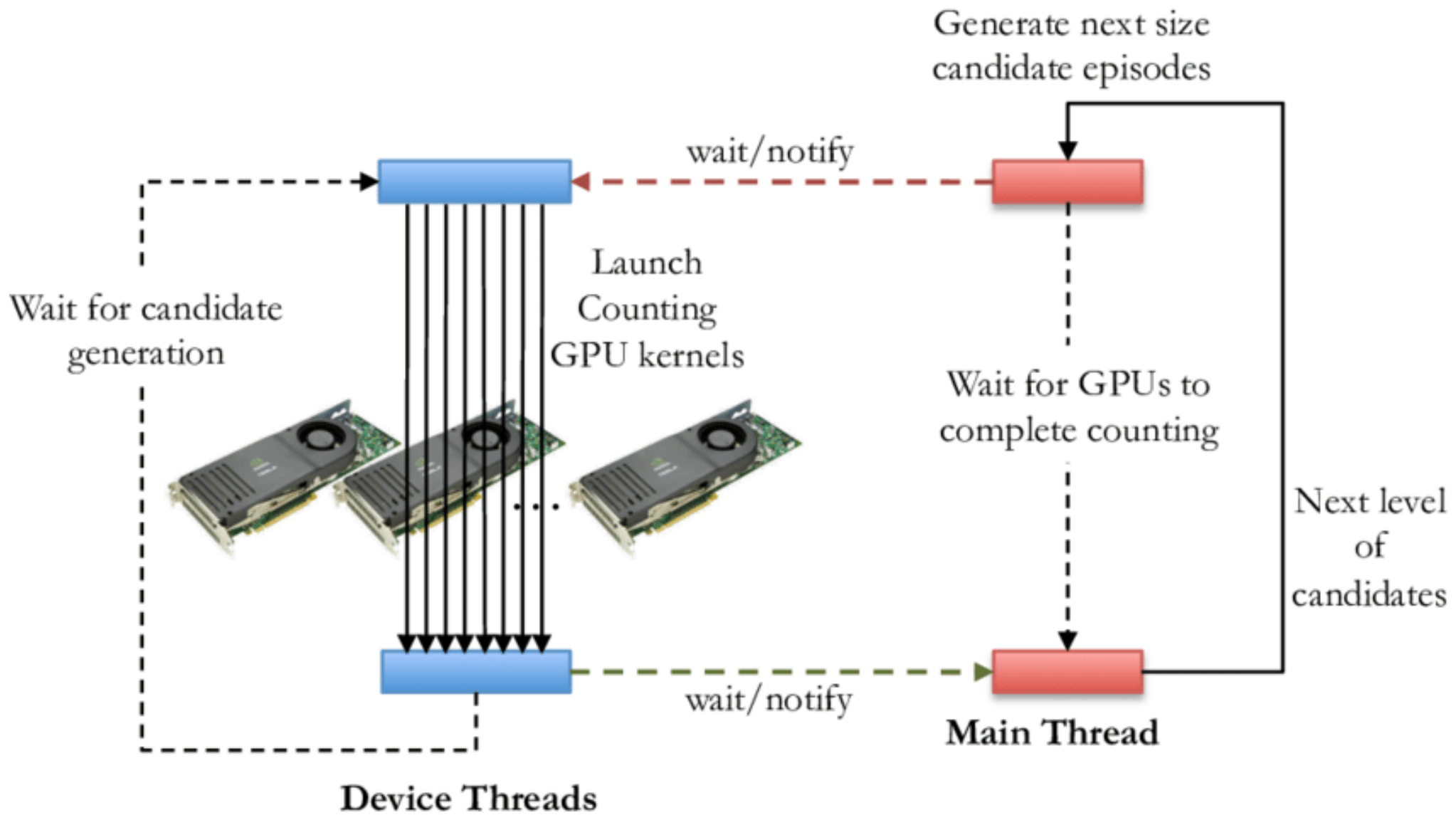
CUDA编程模型是一个异构模型，需要CPU和GPU协同工作

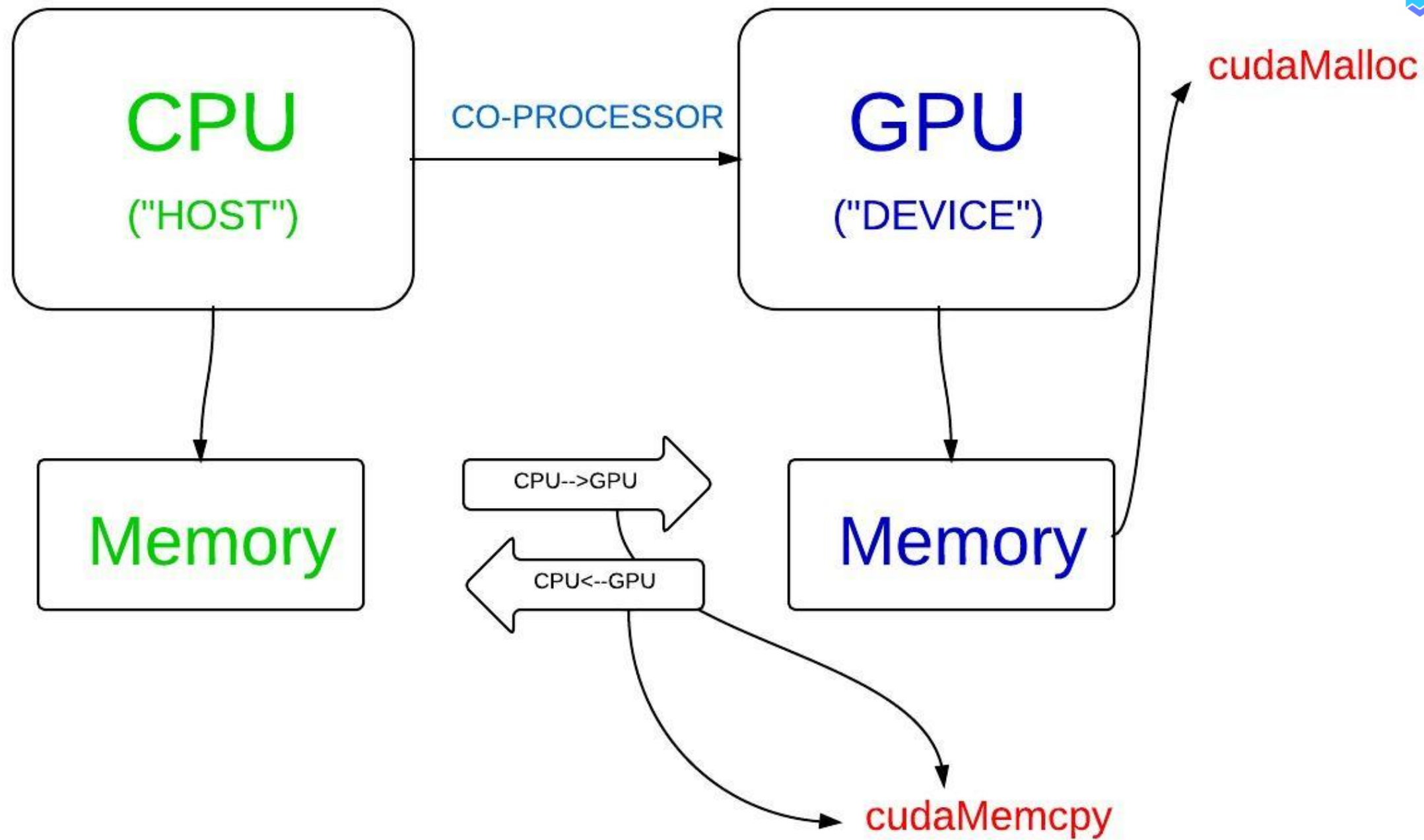


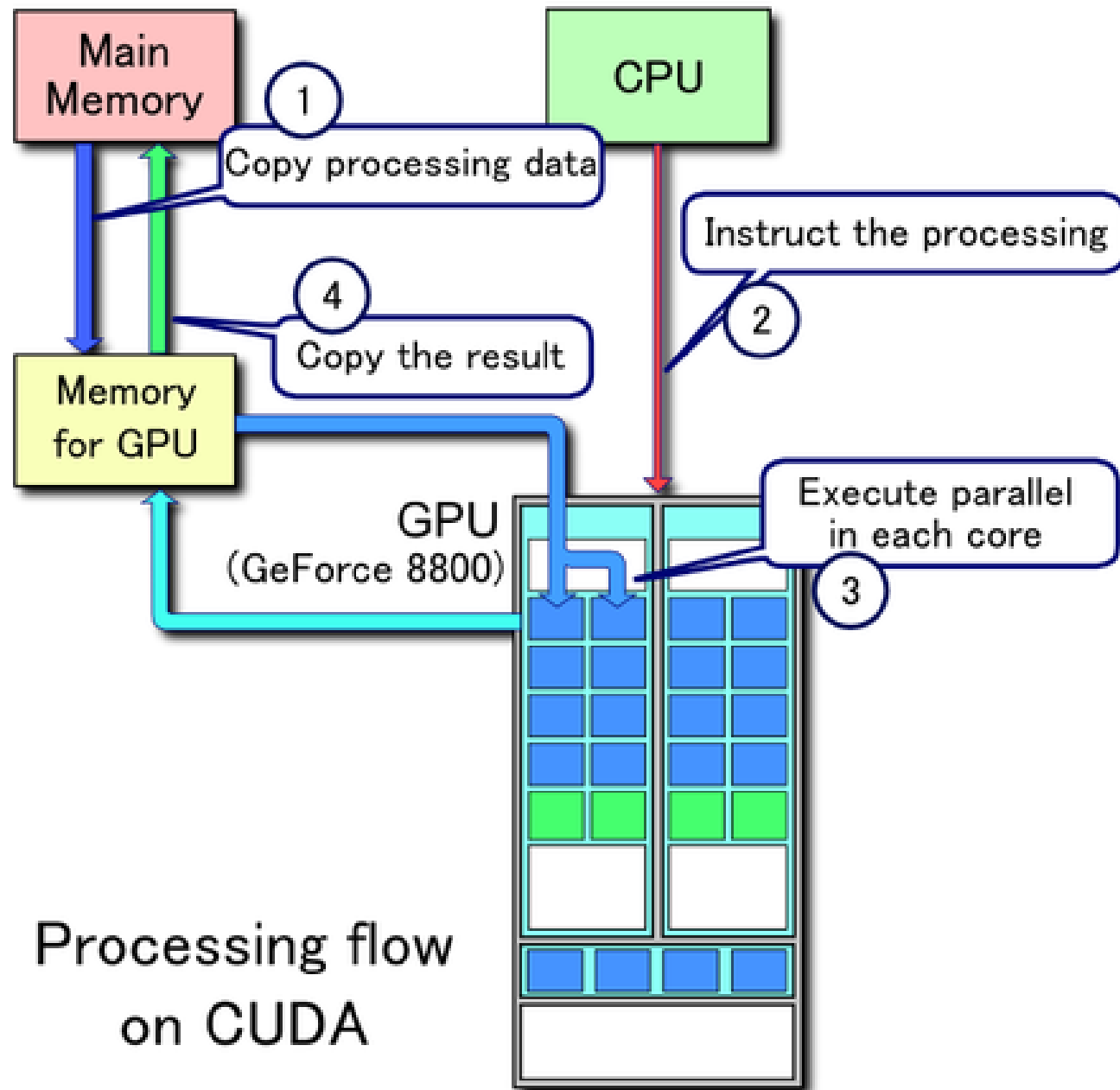
CUDA ® : A General-Purpose Parallel Computing Platform and Programming Model

| GPU Computing Applications | | | | | | |
|--|------------------------------------|--|---------------------------|-----------------------------|-------------------------------|-----------------------|
| Libraries and Middleware | | | | | | |
| cuDNN TensorRT | cuFFT, cuBLAS, cuRAND, cuSPARSE | CULA MAGMA | Thrust NPP | VSIPL, SVM, OpenCurrent | PhysX, OptiX, iRay | MATLAB Mathematica |
| Programming Languages | | | | | | |
| C | C++ | Fortran | Java, Python, Wrappers | DirectCompute | Directives (e.g., OpenACC) | |
| CUDA-enabled NVIDIA GPUs | | | | | | |
| Turing Architecture (Compute capabilities 7.x) | DRIVE/JETSON AGX Xavier | GeForce 2000 Series | | Quadro RTX Series | Tesla T Series | |
| Volta Architecture (Compute capabilities 7.x) | DRIVE/JETSON AGX Xavier | | | | Tesla V Series | |
| Pascal Architecture (Compute capabilities 6.x) | Tegra X2 | GeForce 1000 Series | | Quadro P Series | Tesla P Series | |
| Maxwell Architecture (Compute capabilities 5.x) | Tegra X1 | GeForce 900 Series | | Quadro M Series | Tesla M Series | |
| Kepler Architecture (Compute capabilities 3.x) | Tegra K1 | GeForce 700 Series GeForce 600 Series | | Quadro K Series | Tesla K Series | |
| | EMBEDDED | CONSUMER DESKTOP, LAPTOP | | PROFESSIONAL WORKSTATION | DATA CENTER | |

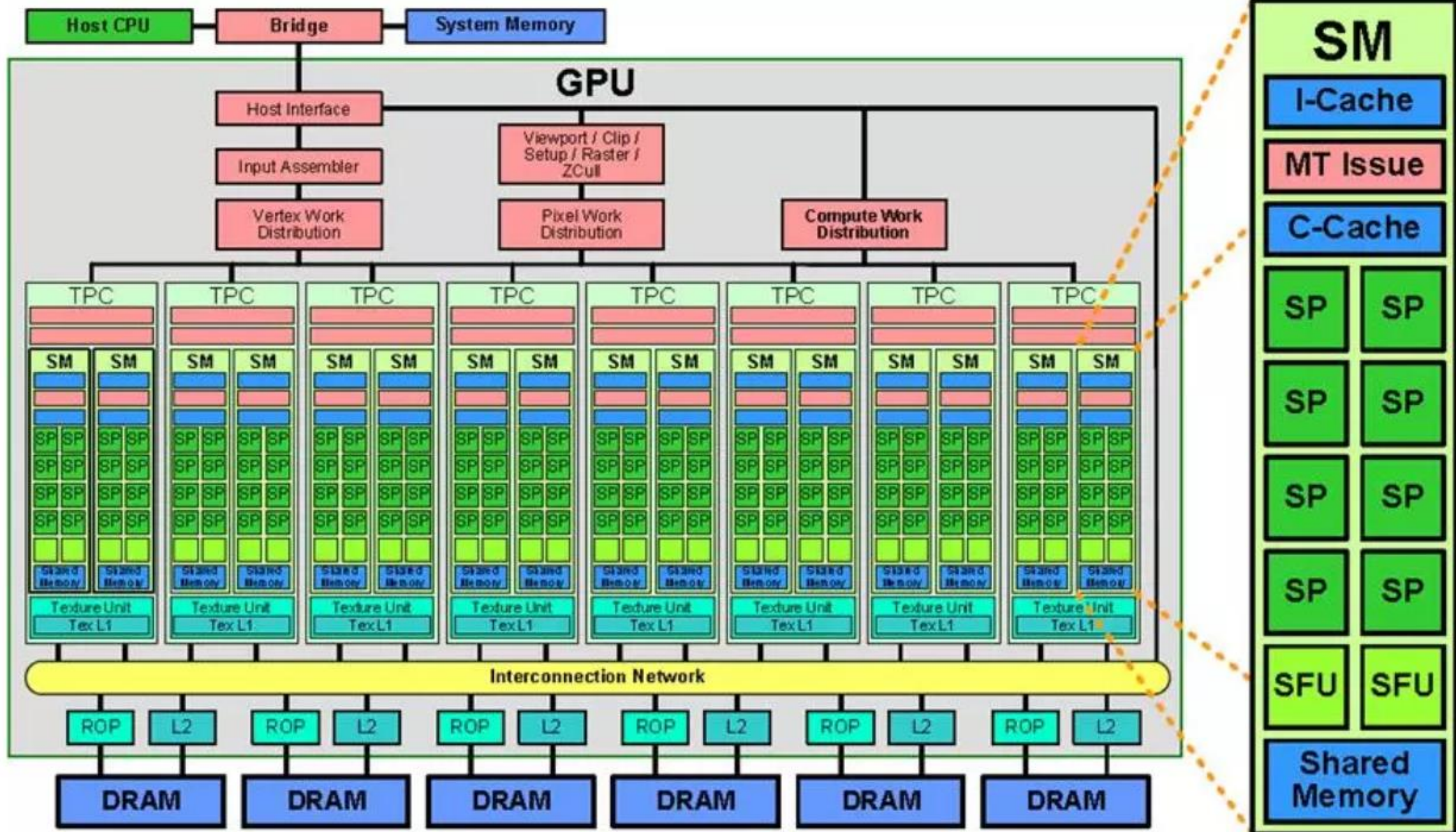








Processing flow
on CUDA

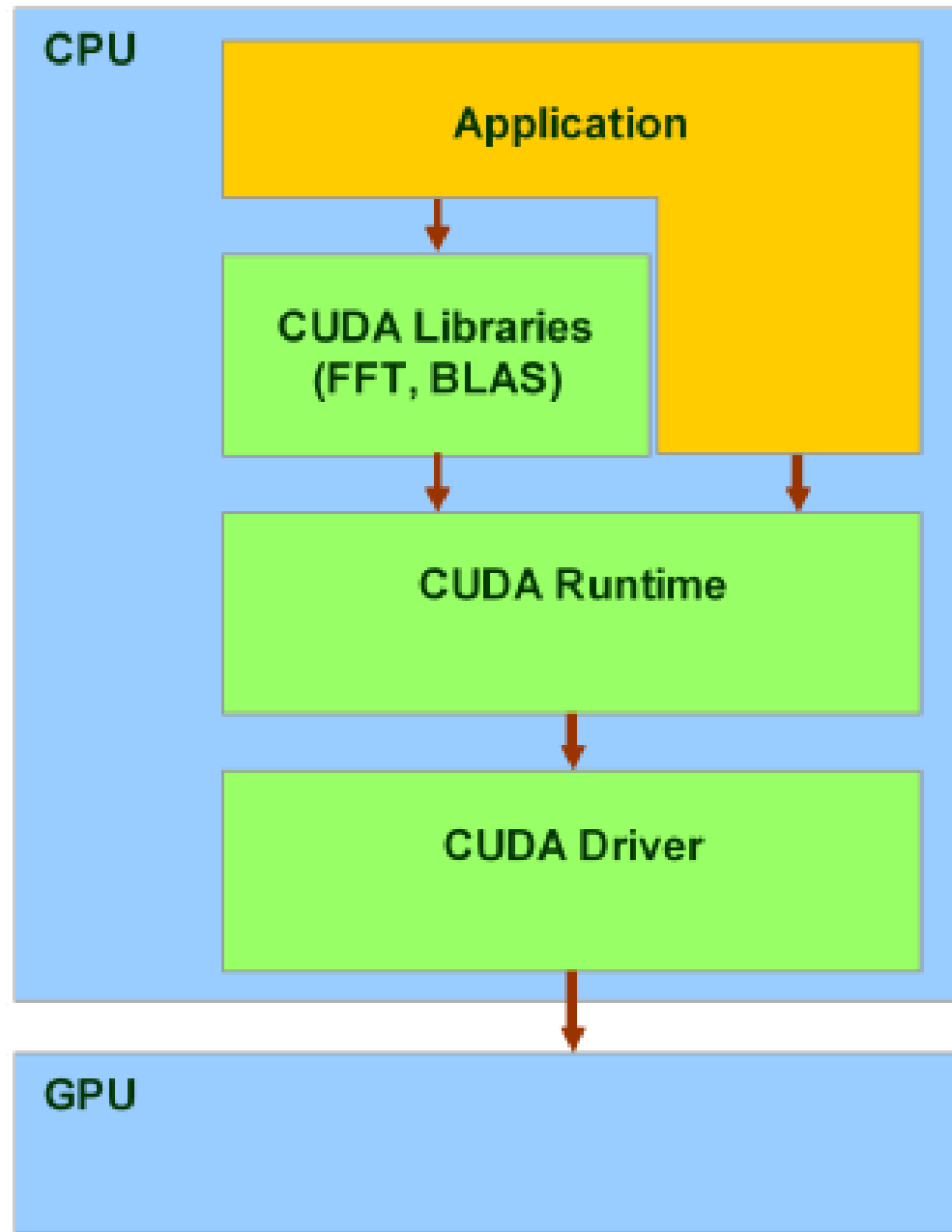


| CPU | GPU | 层次 |
|-----------|-----------|----|
| 算术逻辑和控制单元 | 流处理器(SM) | 硬件 |
| 算术单元 | 批量处理器(SP) | 硬件 |
| 进程 | Block | 软件 |
| 线程 | thread | 软件 |
| 调度单位 | Warp | 软件 |

SP最基本的处理单元, Streaming Processor, 也称为CUDA core。

SM是英文名是 Streaming Multiprocessor, 翻译过来就是流式多处理器。

SM采用的是SIMT (Single-Instruction, Multiple-Thread, 单指令多线程)架构, 基本的执行单元是warps, 一个warp包含32个线程。



CUDA Software Development

CUDA Optimized Libraries:
math.h, FFT, BLAS, ...

Integrated CPU + GPU
C Source Code

NVIDIA C Compiler

NVIDIA Assembly
for Computing (PTX)

CPU Host Code

CUDA
Driver

Profiler

Standard C Compiler

GPU

CPU

```
void serial_function(... ) {  
    ...  
}  
void other_function(int ... ) {  
    ...  
}
```

```
void saxpy_serial(float ... ) {  
    for (int i = 0; i < n; ++i)  
        y[i] = a*x[i] + y[i];  
}
```

```
void main( ) {  
    float x;  
    saxpy_serial(..);  
    ...  
}
```

Write Parallel
CUDA code

C CUDA
Key Kernels

NVCC

CUDA object
files

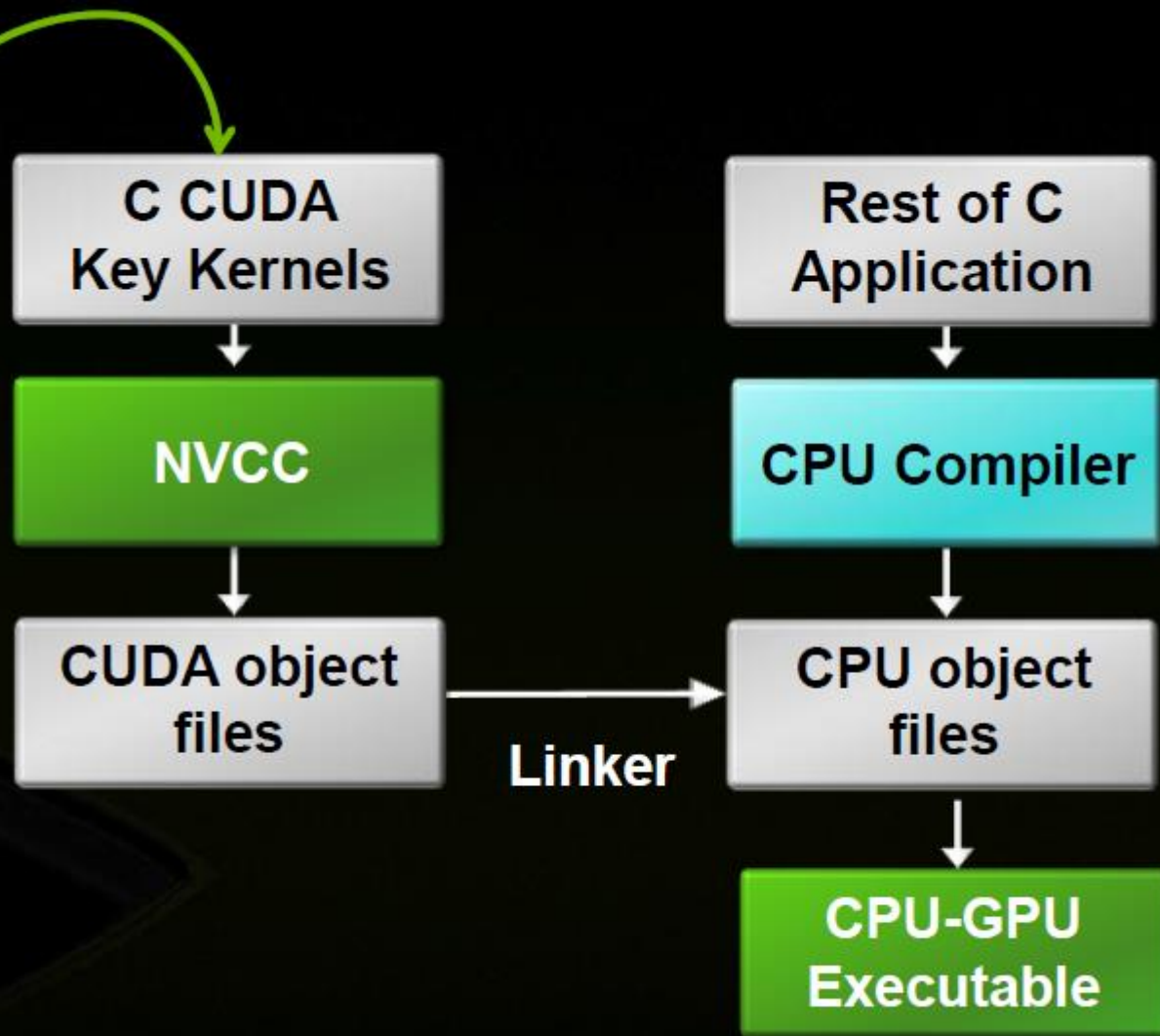
Rest of C
Application

CPU Compiler

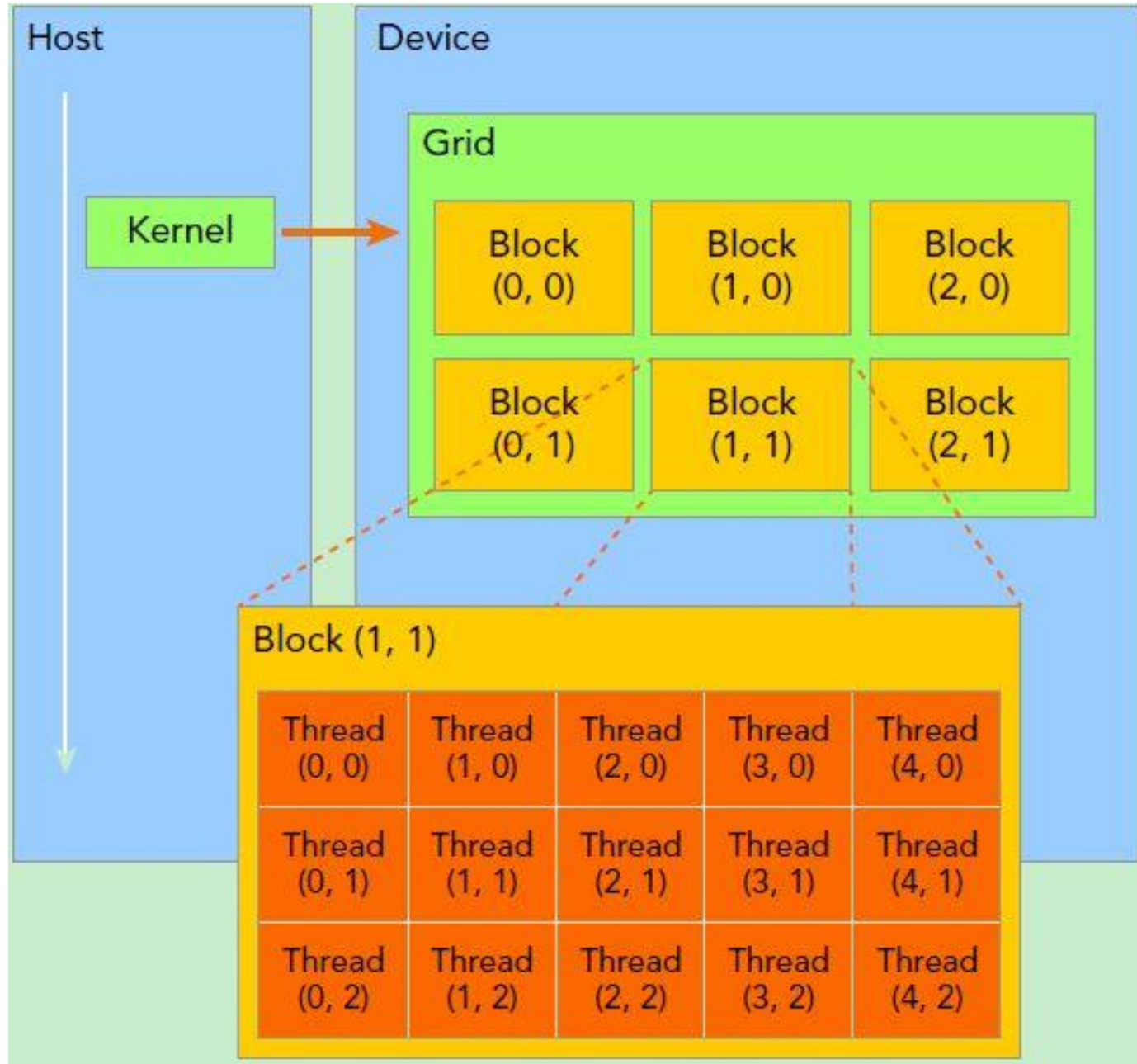
CPU object
files

Linker

CPU-GPU
Executable

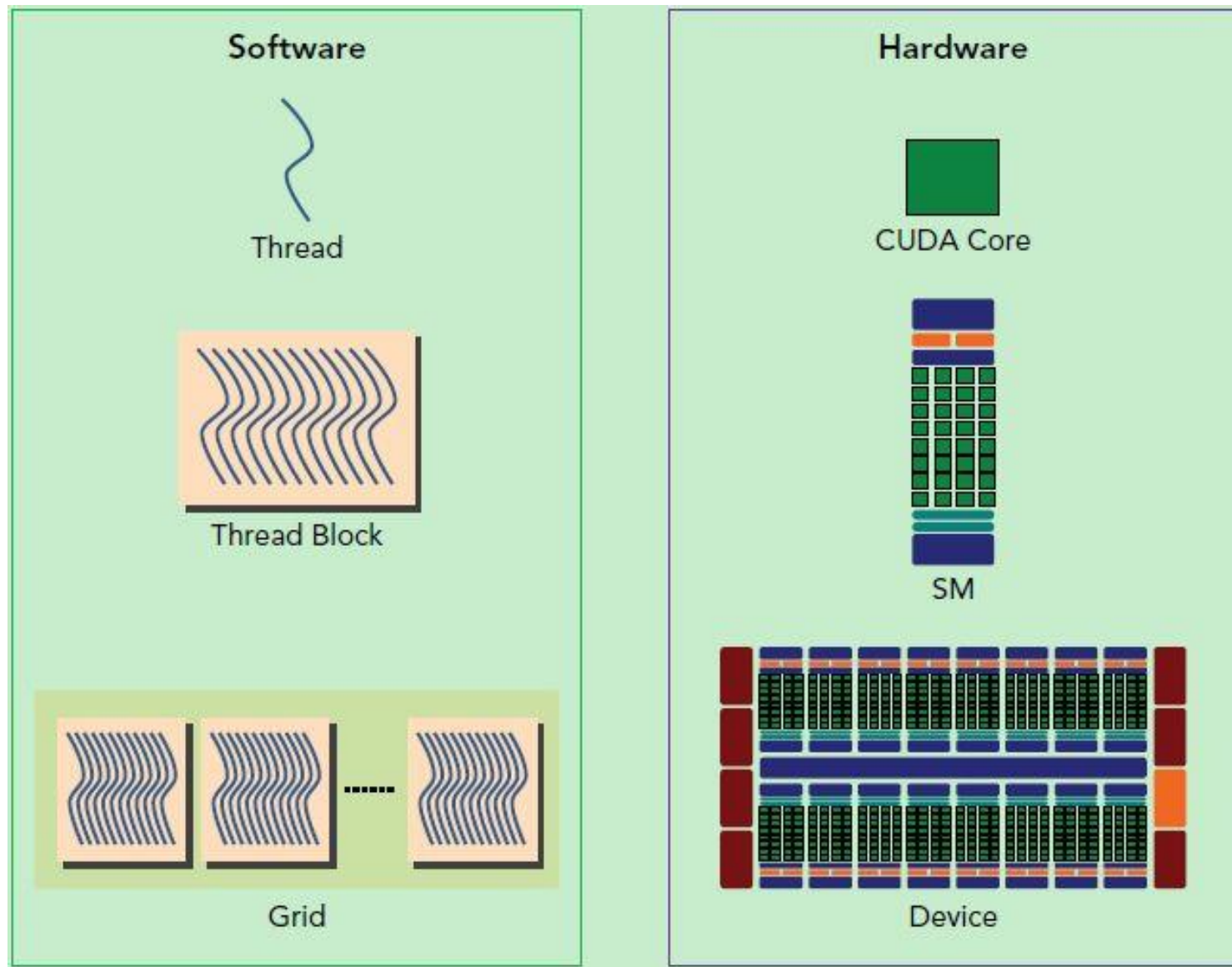


Grid of Thread Blocks



Kernel上的两层线程组织结构 (2-dim)


CUDA编程的逻辑层和物理层



- 每个thread由每个SP执行
- 每个thread block由SM执行

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
    ...
}
```



指定kernel要执行的线程数量

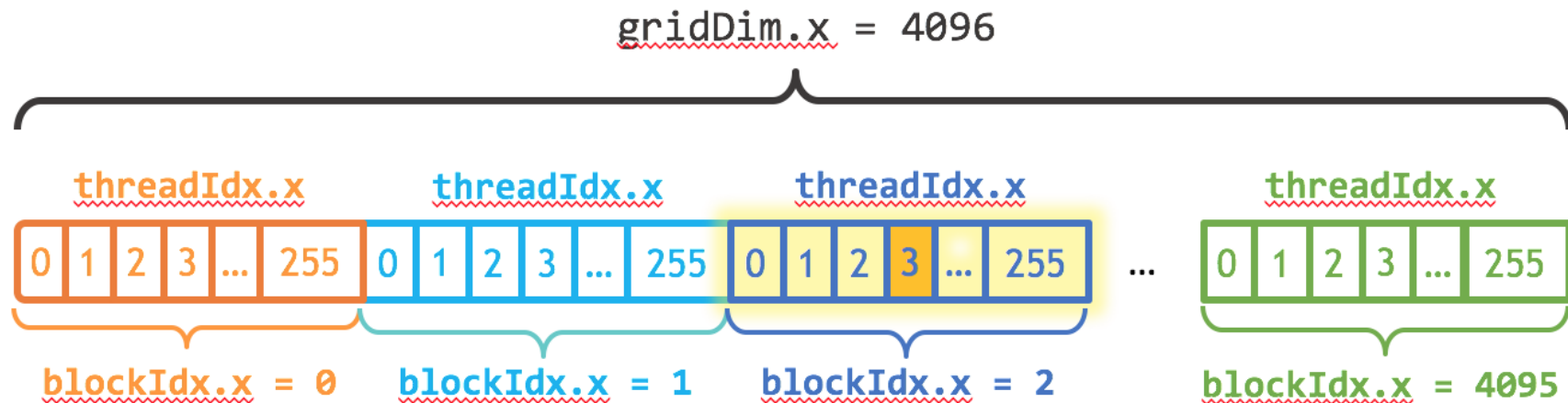
<<<1, size>>>, 表示分配了一个线程块 (Block), 每个线程块有分配了size个线程

“<<<>>>” 中的 参数并不是传递给设备代码的参数, 而是定义主机代码运行时如何启动设备代码。

CUDA C语言对C语言的扩展之一就是加入了一些函数前缀

| | Executed on the: | Only callable from the: |
|--------------------------------------|------------------|-------------------------|
| __device__ float DeviceFunc() | device | device |
| __global__ void KernelFunc() | device | host |
| __host__ float HostFunc() | host | host |

若向量大小为 $1 < \ll 20$ ，而block大小为256，那么grid大小是4096，kernel的线程层级结构如下图所示：腾讯课堂

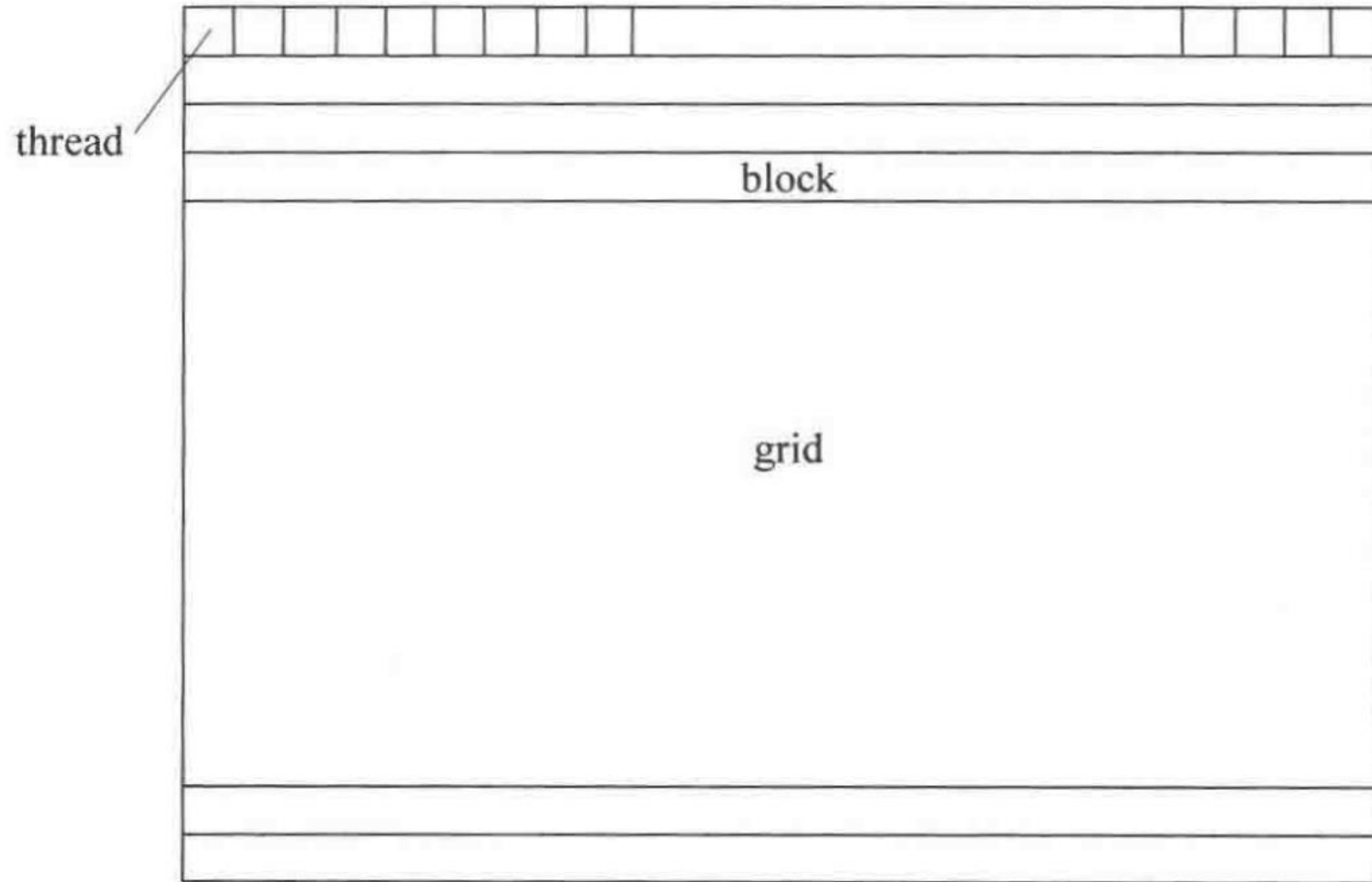


$$\text{index} = \text{blockIdx.x} * \text{blockDim.x} + \text{threadIdx.x}$$

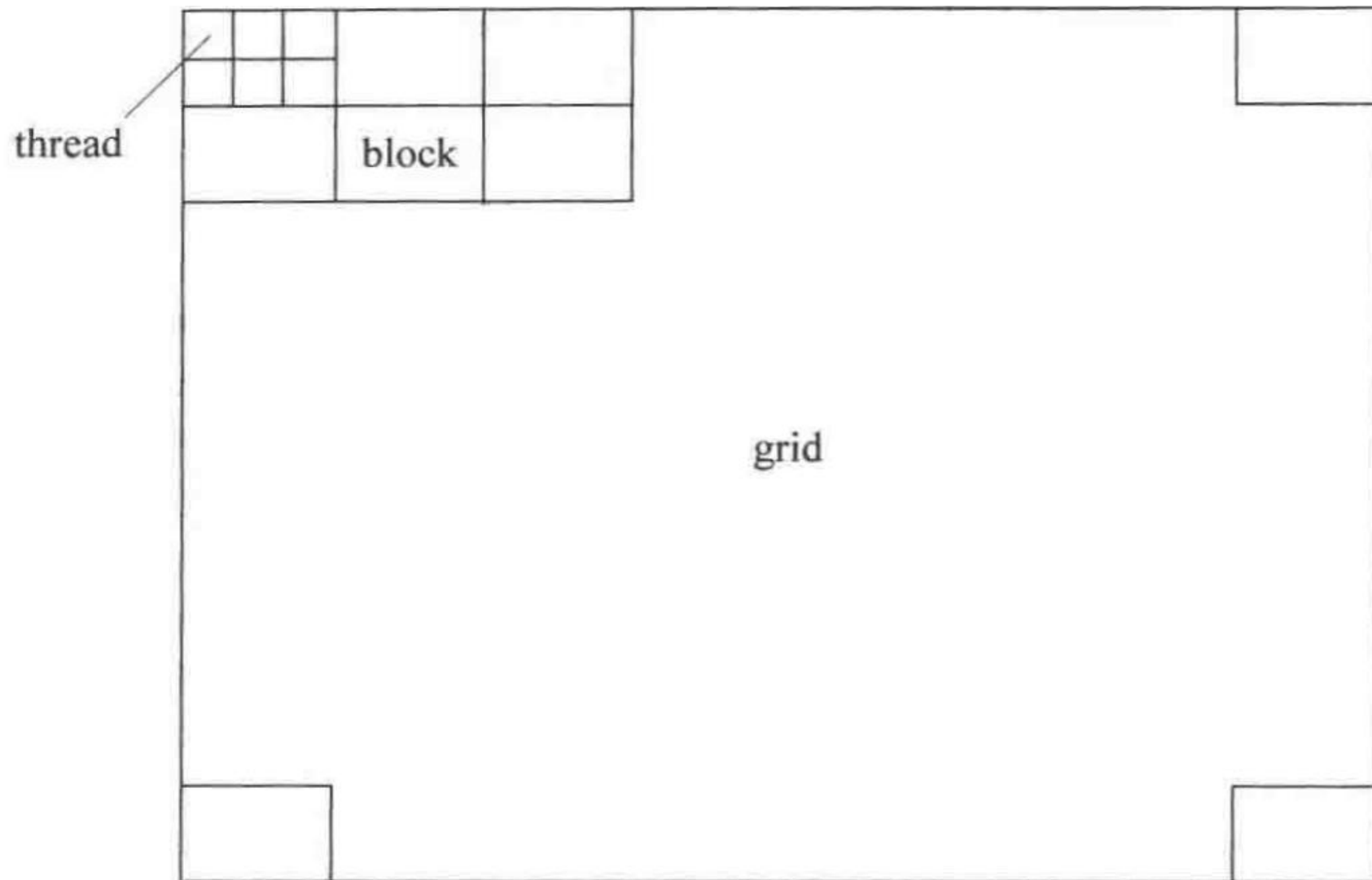
$$\text{index} = (2) * (256) + (3) = 515$$

- 一个线程需要两个内置的坐标变量 (blockIdx, threadIdx) 来唯一标识，它们都是dim3类型变量，其中 **blockIdx** 指明线程所在grid中的位置，而 **threadIdx** 指明线程所在block中的位置。
- 对于一个2-dim的block(Dx,Dy)，线程(x,y)的ID值为(x+y*Dx)，如果是3-dim的block(Dx,Dy,Dz)，线程(x,y,z)的ID值为(x+y*Dx+z*Dx*Dy)。
- 不是block越大越好，而要适当选择。目前GPU可以达到 **1024** threads/block

1D线程维度



2D线程维度



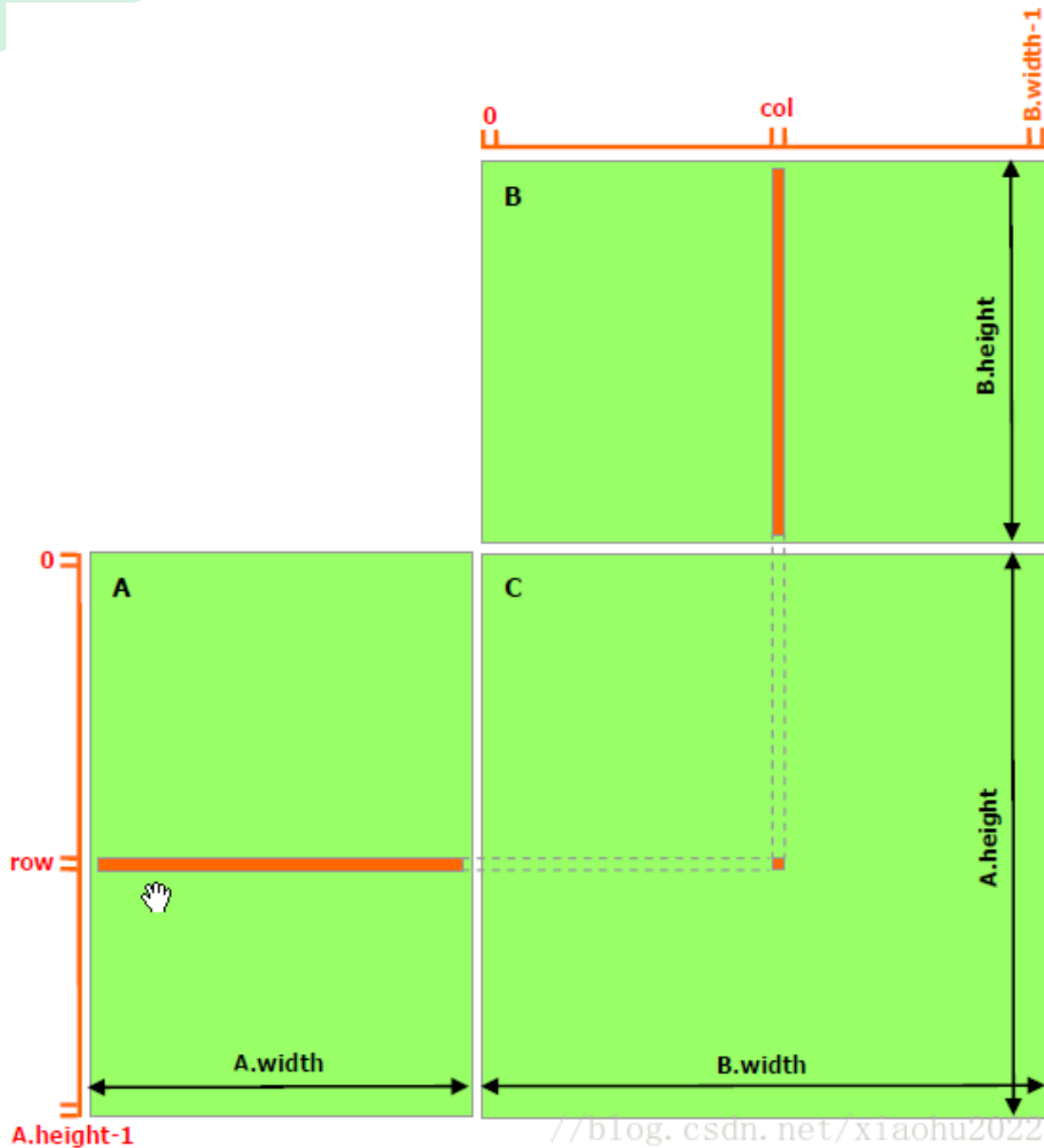
```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
                      float C[N][N])
{
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    // Kernel invocation with one block of N * N * 1 threads
    int numBlocks = 1;
    dim3 threadsPerBlock(N, N);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```


kernel的这种线程组织结构天然适合vector, matrix等运算, 如利用2-dim结构实现两个矩阵的加法, 每个线程负责处理每个位置的两个元素相加, 代码如下所示。线程块大小为(16, 16), 然后将 $N \times N$ 大小的矩阵均分为不同的线程块来执行加法运算。

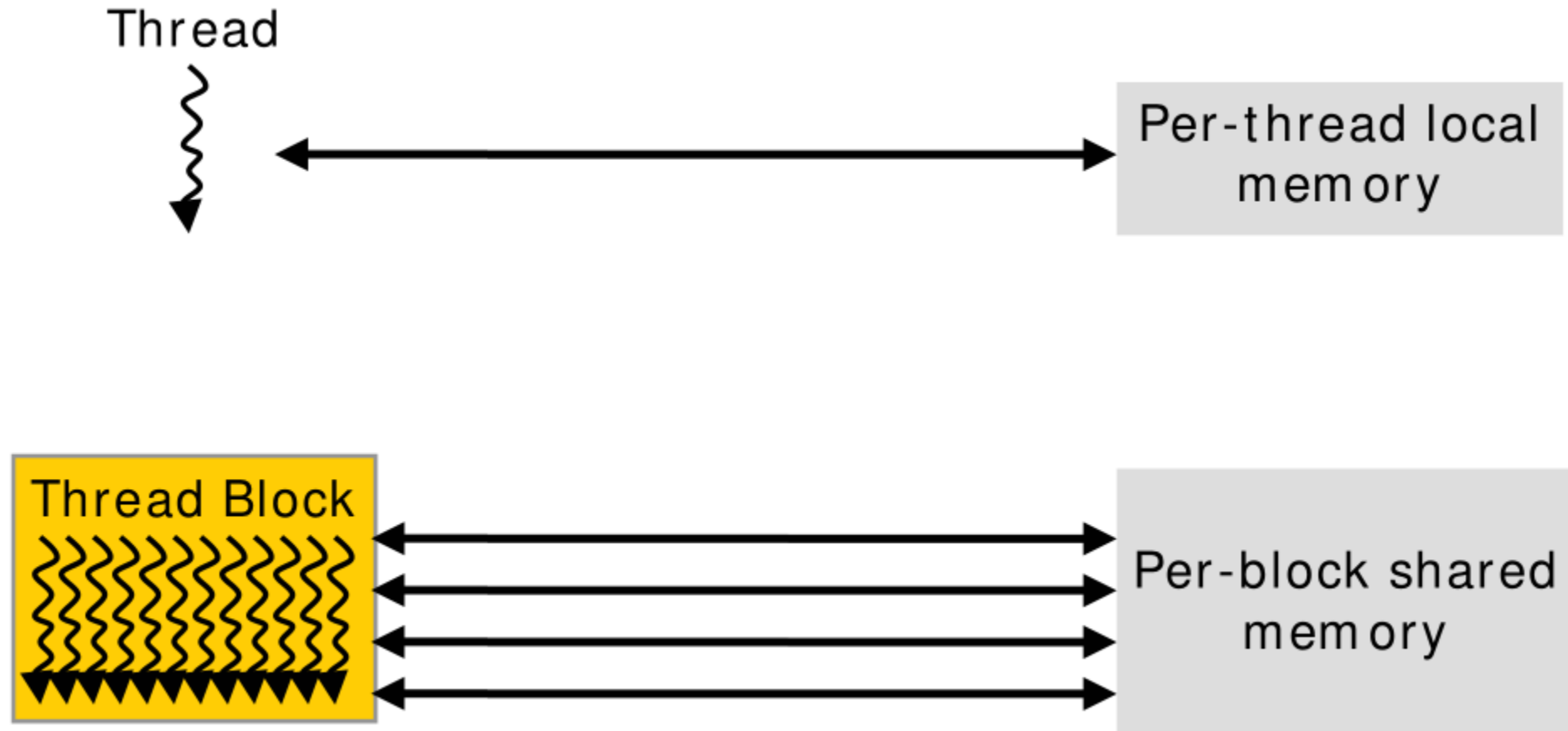
```
// Kernel定义
__global__ void MatAdd(float A[N][N], float B[N][N], float C[N][N])
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j < N)
        C[i][j] = A[i][j] + B[i][j];
}

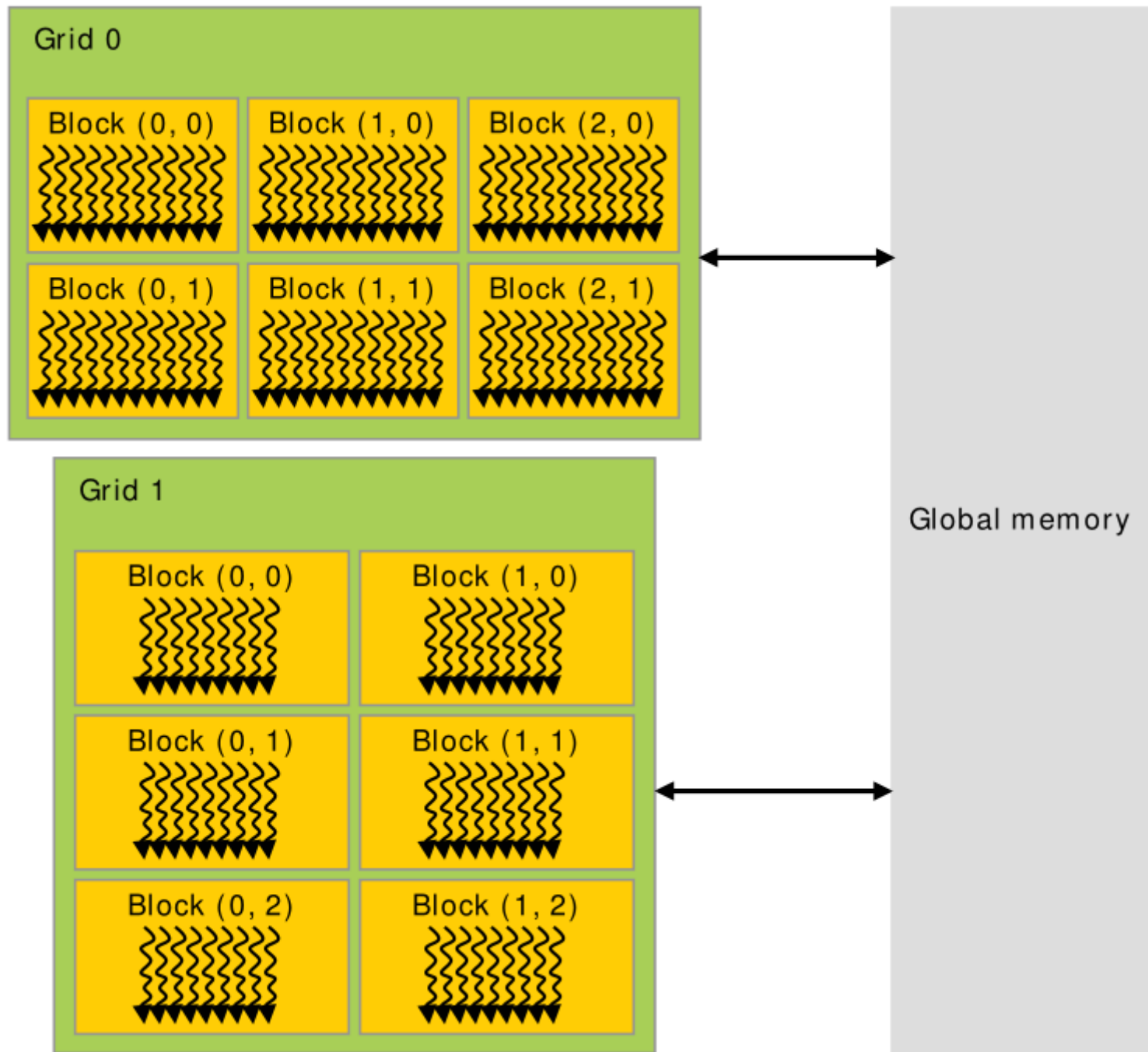
int main()
{
    ...
    // Kernel 线程配置
    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
    // kernel调用
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

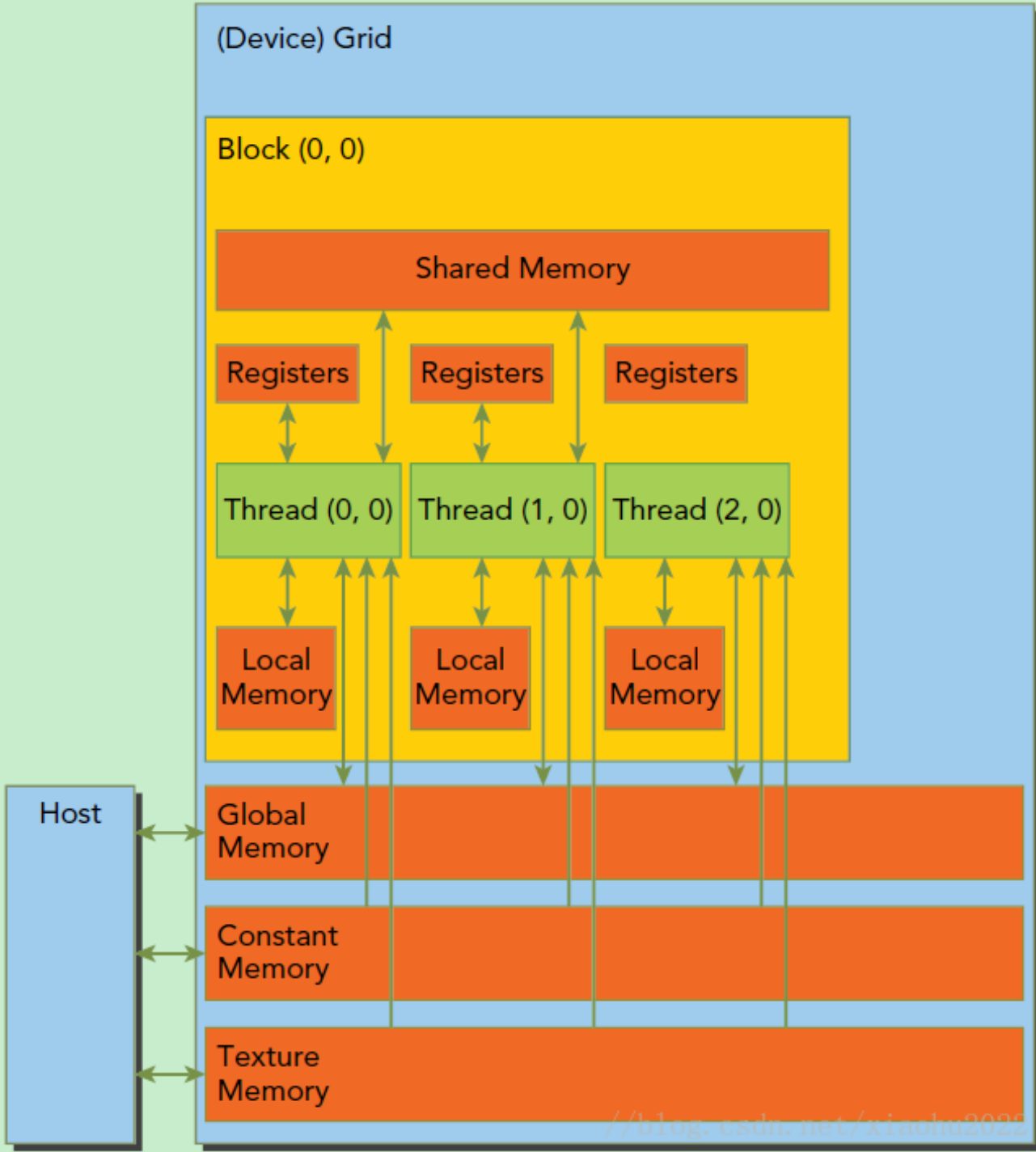


这里矩阵大小为 1024×1024 ，设计的线程的block大小为(32, 32)，那么grid大小为(32, 32)。

CUDA的内存模型







每个线程有自己的私有本地内存（Local Memory），而每个线程块有包含共享内存（Shared Memory），可以被线程块中所有线程共享，其生命周期与线程块一致。此外，所有的线程都可以访问全局内存（Global Memory）。还可以访问一些只读内存块：常量内存（Constant Memory）和纹理内存（Texture Memory）。

C Program Sequential Execution

Serial code

Parallel kernel

Kernel0<<<>>>()

Serial code

Parallel kernel

Kernel1<<<>>>()

Host



Device

Grid 0

Block (0, 0)



Block (1, 0)



Block (2, 0)



Block (0, 1)



Block (1, 1)



Block (2, 1)



Host



Device

Grid 1

Block (0, 0)



Block (1, 0)



Block (0, 1)



Block (1, 1)



Block (0, 2)



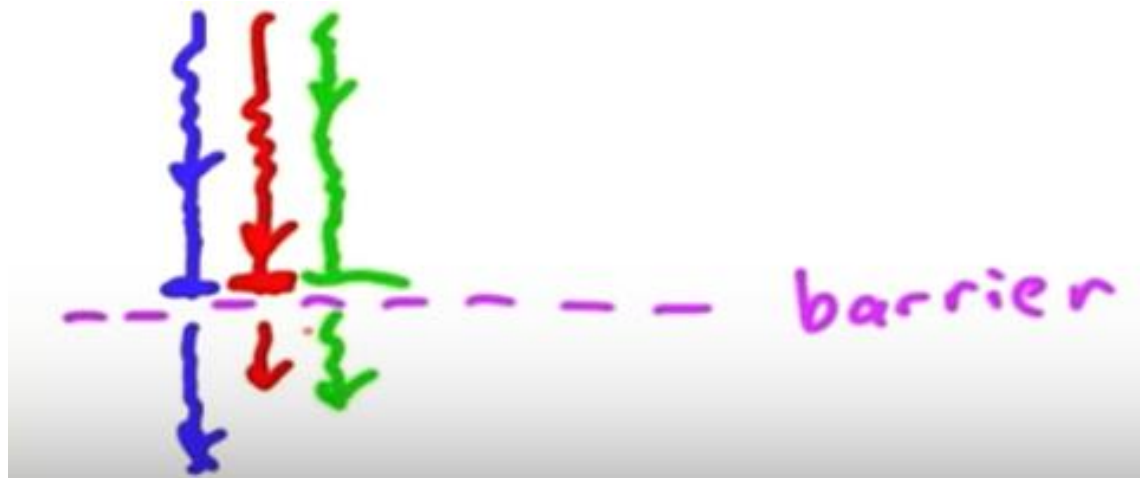
Block (1, 2)



Threads的同步

Barrier – point in the program where threads stop and wait.

When all threads have reached the barrier, they can proceed.



```
Int idx = threadIdx.x;
__shared__ int array[128];
array[idx] = threadIdx.x;
__syncthreads();
If (idx < 127) {
    int temp = array[idx+1];
    __syncthreads();
    array[idx] = temp;
    __syncthreads();
}
```


CUDA synchronization

CUDA provides a synchronization **barrier** routine for those threads within each block
`__syncthreads()`

This routine would be used **within a kernel**.

Threads would wait at this point until all threads in the block have reached it and they are all released.

Note: **only synchronizes with other threads in block**.

七个步骤

`cudaSetDevice(0);` //获取设备; 只有一个GPU时或默认使用0号GPU时可以省略

`cudaMalloc((void**) &d_a, sizeof(float)*n);` //分配显存

`cudaMemcpy(d_a, a, sizeof(float)*n, cudaMemcpyHostToDevice);` //数据传输 host to device

`gpu_kernel<<<blocks, threads>>>(**);` //kernel函数

`cudaMemcpy(a, d_a, sizeof(float)*n, cudaMemcpyDeviceToHost);` //数据传输 device to host

`cudaFree(d_a);` //释放显存空间

`cudaDeviceReset();` //重置设备; 可以省略

GPU的axpy的实现

```
__global__ void axpy_kernel(int N, float ALPHA, float *X, int OFFX, int INCX, float *Y, int OFFY, int INCY)
{
    int i = (blockIdx.x + blockIdx.y*gridDim.x) * blockDim.x + threadIdx.x;
    if(i < N) Y[OFFY+i*INCY] += ALPHA*X[OFFX+i*INCX];
}
```

```
extern "C" void axpy_gpu(int N, float ALPHA, float * X, int INCX, float * Y, int INCY)
{
    axpy_gpu_offset(N, ALPHA, X, 0, INCX, Y, 0, INCY);
}
```

```
extern "C" void axpy_gpu_offset(int N, float ALPHA, float * X, int OFFX, int INCX, float * Y, int OFFY, int INCY)
{
    axpy_kernel<<<cuda_gridsize(N), BLOCK>>>(N, ALPHA, X, OFFX, INCX, Y, OFFY, INCY);
    check_error(cudaPeekAtLastError());
}
```