CS 758: Advanced Topics in Computer Architecture

Lecture #3: Parallelism + first GPU program

Professor Matthew D. Sinclair

Some of these slides were developed by Tim Rogers at the Purdue University, Tor Aamodt at the University of British Columbia, and Wen-mei Hwu & David Kirk at the University of Illinois at Urbana-Champaign.

Slides enhanced by Matt Sinclair

Announcements

- HW0 Released
  - Let me know if you have any problems accessing euler
- Adjusted HW Due Dates to reflect added HW0
- Adjusted Reviews
  - One fewer reviews – no review of Hower paper
- Office Hours
  - Mondays 1-2
  - Fridays 4-5 – no office hours this Friday (9/13)
  - My office: 6369

Today’s Objectives

- Talk about Parallelism
- Introduce the GPU Programming model
- Some “Nuts and Bolts” of CUDA C
- By the end of today, you should be able to write a simple CUDA kernel (i.e. **vectorAdd**) within a couple hours

Parallelism in general

Many different definitions/types of parallelism

- Instruction Level Parallelism
  - Different machine instructions in the same thread can execute in parallel
- Task Level Parallelism
  - Higher level tasks can run concurrently
- Bit level Parallelism
  - In VHDL exploit the ability to do level bit-level computation in parallel (i.e. longer words, carry-lookahead adders)
- Data Level Parallelism
  - Identical computation just on different data
  - Single Instruction Multiple Data (SIMD) instructions exploit data parallelism
  - Single Program Multiple Data (SPMD) applications exploit data parallelism

GPUs are designed to exploit DLP

Remember: Can’t get around Ahmadl's Law

$$S_{\text{latency}}(s) = \frac{1}{(1-p) + \frac{p}{s}}$$

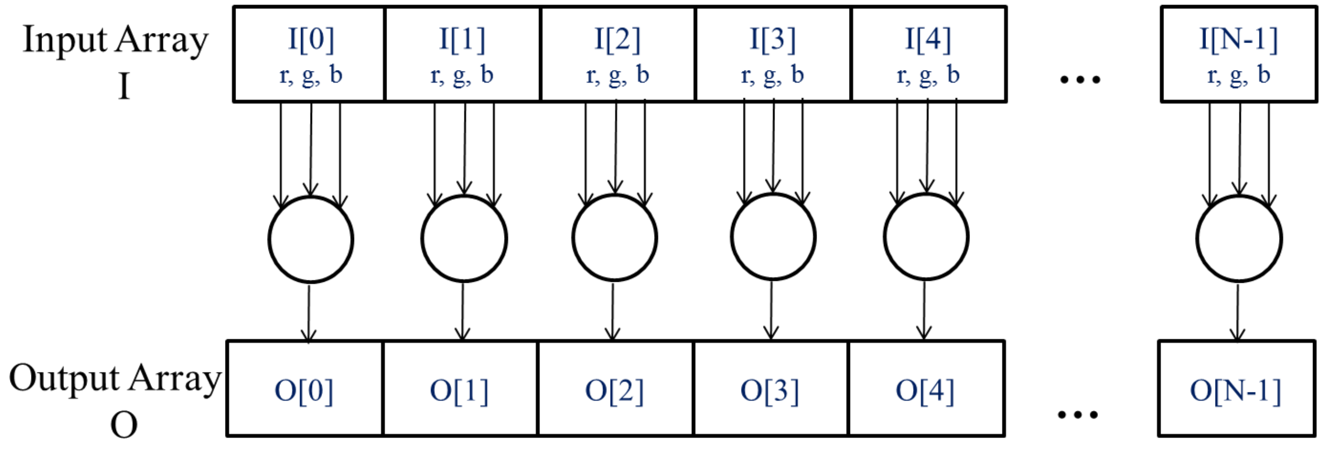
- There will always be some serial work that needs to be done
- CPUs are much better designed to handle serial work
- CPU and a parallel accelerator will almost certainly always work together.
  - OoO, superscalar CPU = Serial Accelerator
  - GPU = Parallel Accelerator

Bottom line:
Without parallelism in the program, GPUs are useless

Example Application: Conversion to grey-scale

- Every pixel has 3 values to determine the color (R,G,B)
- Compute the **Luminance** value of the pixel
  - Embarrassingly **data-parallel** operation
  - Same operation on every pixel, all independent
  - Parallelism scales 1:1 with input data

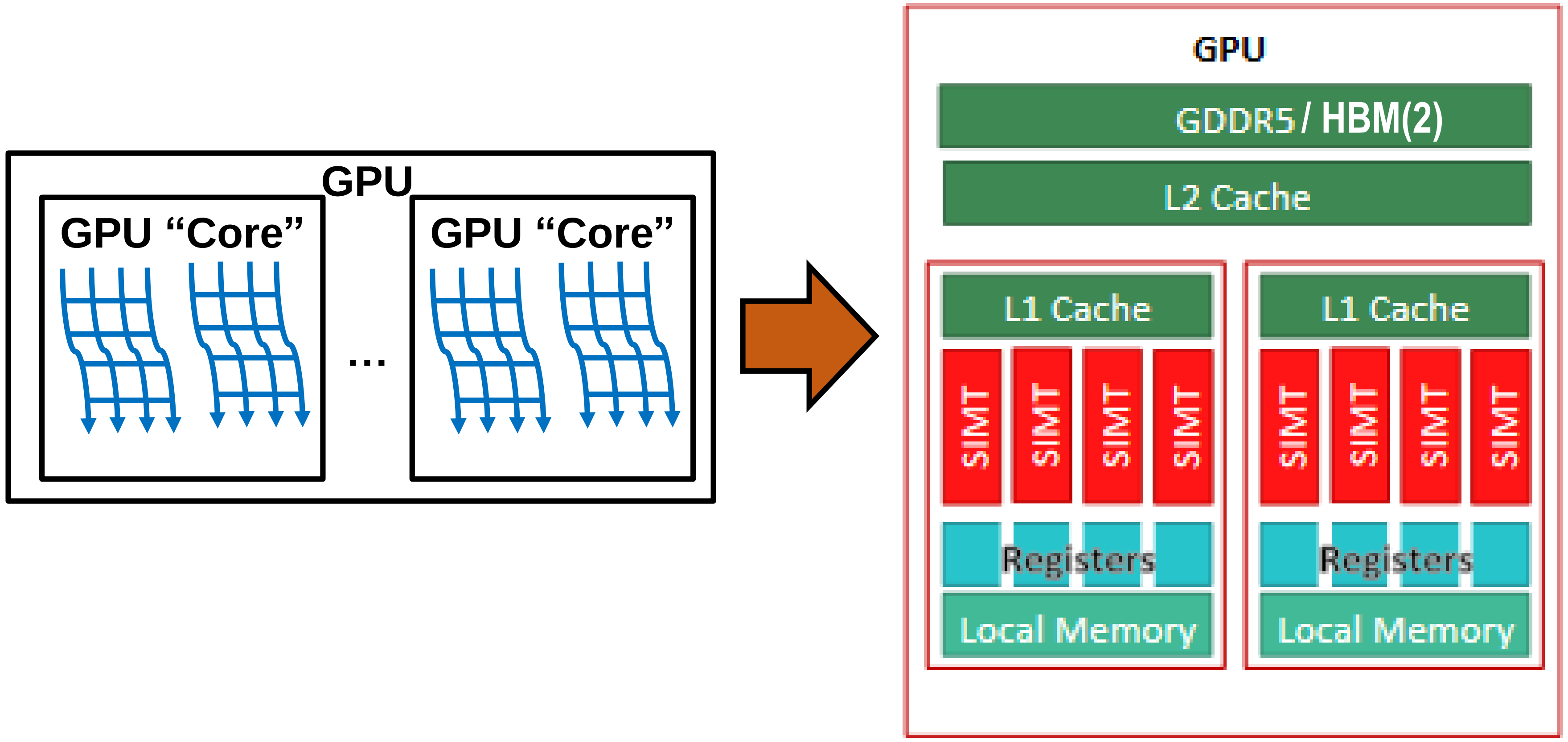
Example of data parallelism



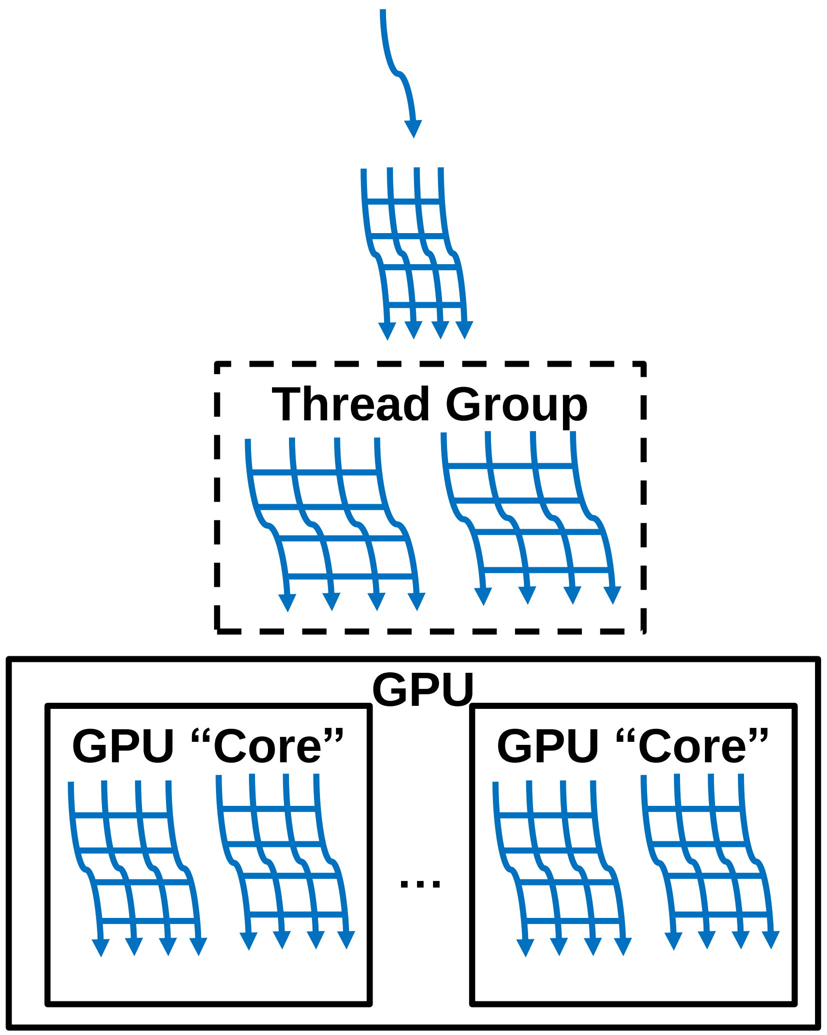
Why Data Parallelism?

- Easy to build efficient hardware to capture it
- The **regularity** in the computation can be exploited to reduce control hardware and make effective use of memory bandwidth

GPU Hardware Overview



GPU Component Names



CUDA/HIP	OpenCL
Thread	Work-item
Warp	Wavefront
Thread Block/CTA	Workgroup
Grid (Kernel)	NDRange (Kernel)

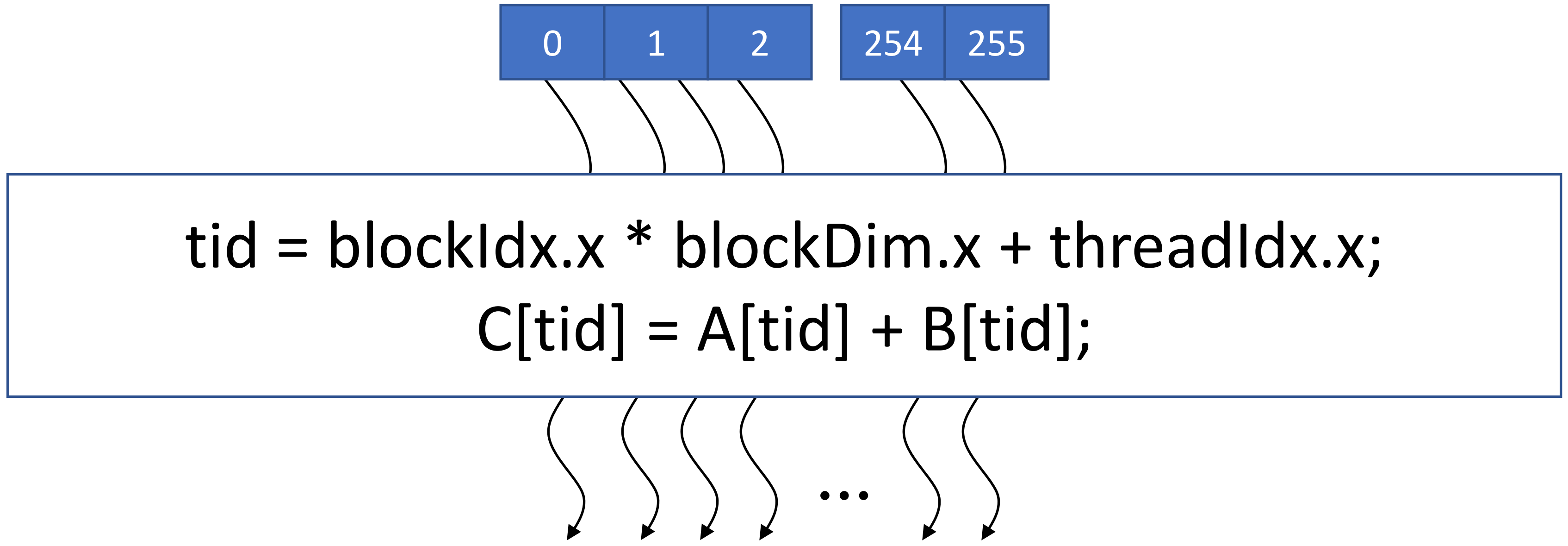
Programming GPUs (**CS/ECE/ME/EMA 759**)

- Program it with CUDA, HIP, or OpenCL
  - CUDA = Compute Unified Device Architecture
    - NVIDIA's proprietary solution
  - OpenCL = Open Computing Language
    - Open, industry wide standard
  - HIP = Heterogeneous interface for portability
    - AMD's open source solution, its successor to OpenCL
  - Extensions to C
  - Perform a "shader task" (a snippet of scalar computation) over many elements
  - Internally, GPU uses scatter/gather and vector mask operations
- Other solutions:
  - C++ AMP (Microsoft), OpenACC (extension to OpenMP)

Note: CUDA is not the only way to program an accelerator.
However it is arguably the most mature and full-featured

A CUDA “Kernel” is a Grid (Array) of threads

- All the threads run the same kernel code (Single Program Multiple Data -- SPMD)
- Each thread has a unique index



```
tid = blockIdx.x * blockDim.x + threadIdx.x;
C[tid] = A[tid] + B[tid];
```

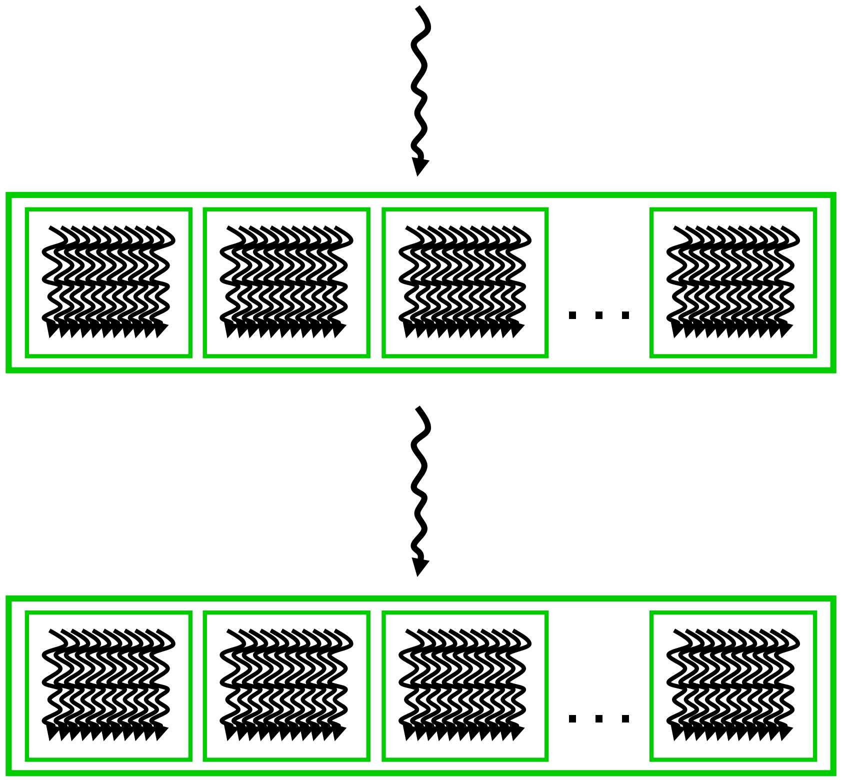
Execution model

Serial Code (host)

Parallel Kernel (device)
KernelA<<< nBlk, nTid >>>(args);

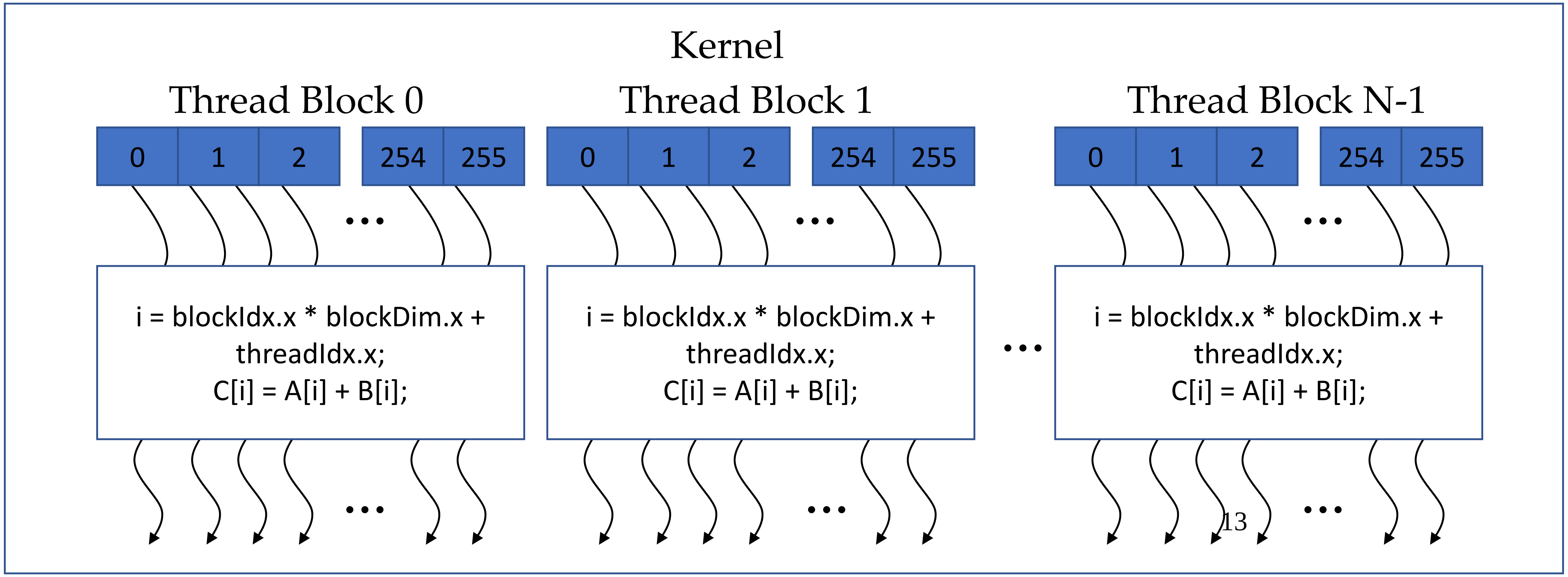
Serial Code (host)

Parallel Kernel (device)
KernelB<<< nBlk, nTid >>>(args);



Thread Blocks

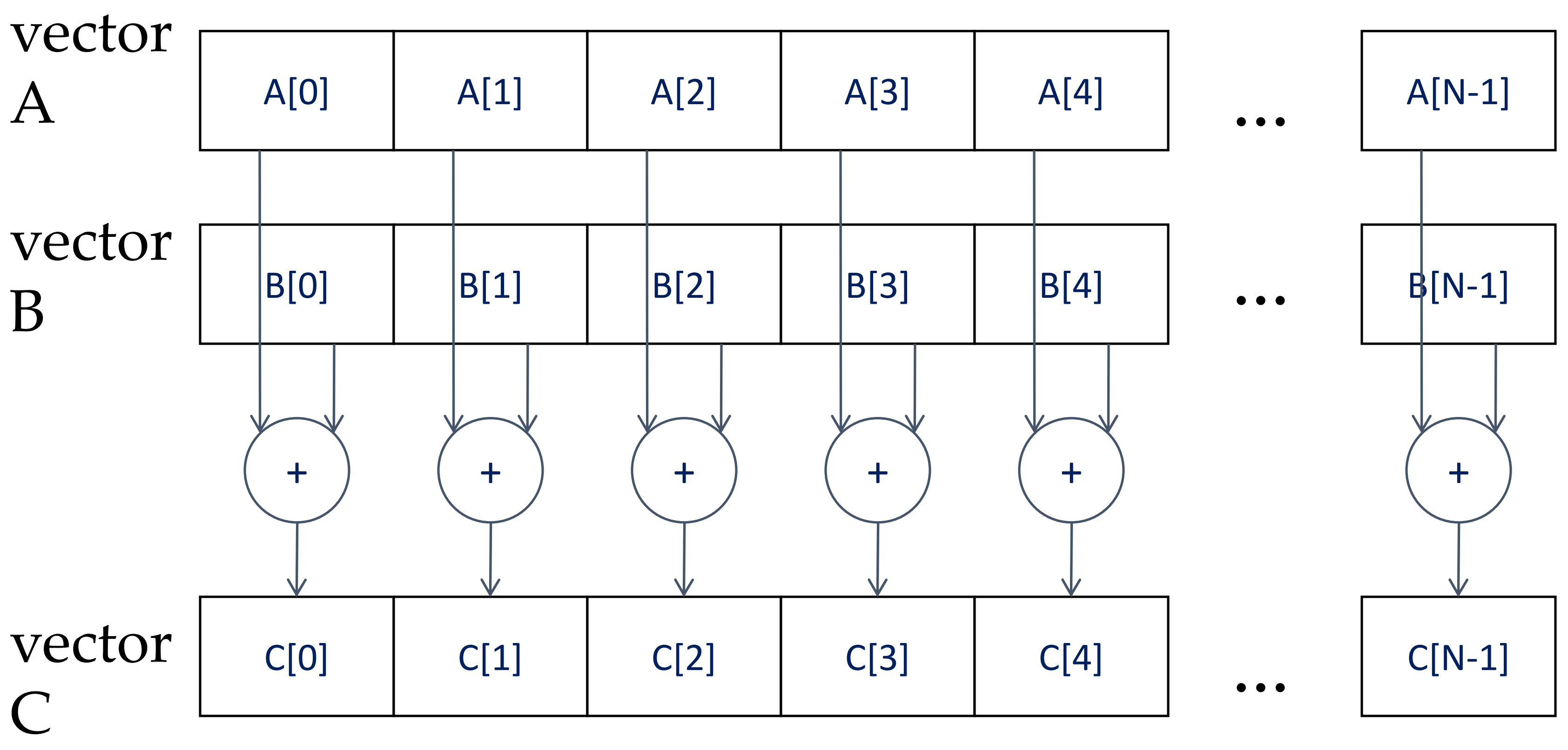
- Divide the kernel into chunks
- Threads within a thread block can cooperate
  - On-chip shared memory, atomic operations and barriers
- Threads in different thread blocks generally do not communicate



Why threadIdx.x?

- Thread ids (and block ids) are actually 3 dimensional
- Simplifies the addressing in code when accessing multi-dimensional data
    - Consider an image with (x,y) for each pixel: if you want to do one operation/pixel, you can simply launch (x,y) threads.
    - Easy to map the parallelism directly to the data.
- In reality these threads are linearized by the hardware – but it makes programming simpler.

Simplest Kernel: Vector Addition



Vector Addition in C

```
// 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);
}
```

Outline of vector addition with the GPU

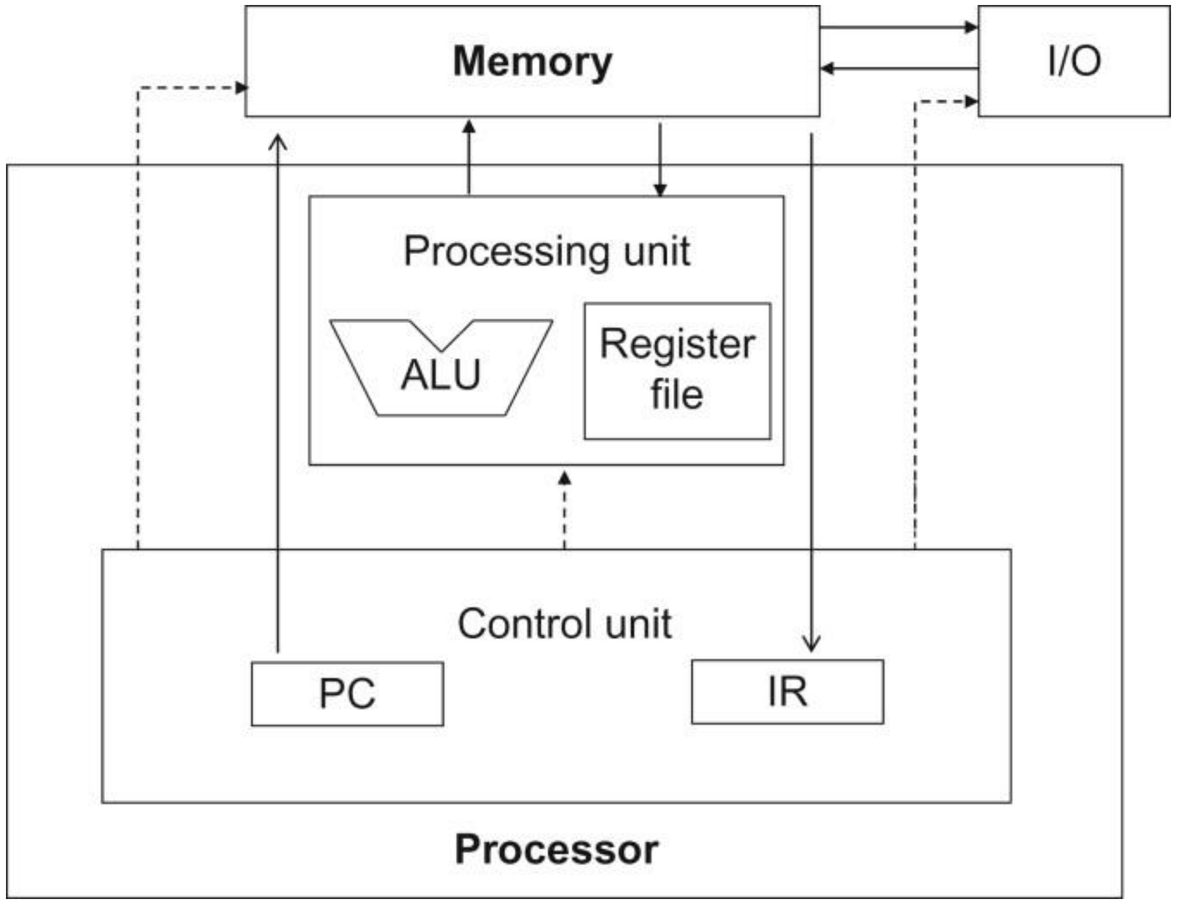
```
#include <cuda.h>
void vecAdd(float* A, float* B, float* C, int n)
{
     int size = n* sizeof(float);
    float* A_d, B_d, C_d;
    …
1. // Allocate device memory for A, B, and C
   // copy A and B to device memory

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

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

All run on the CPU

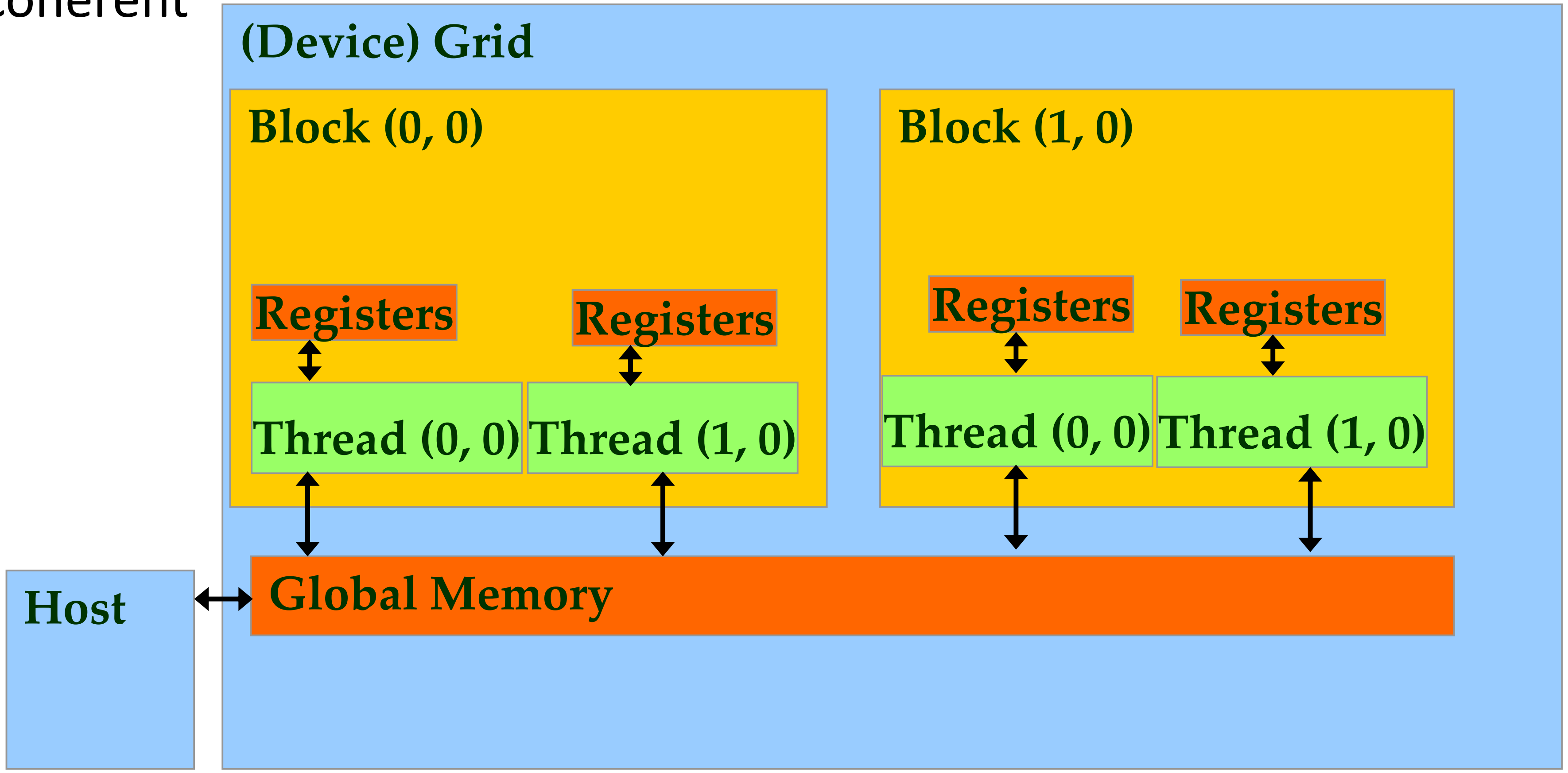
Conceptual model of a Von Neumann thread



Conceptually you can think of a thread this way:
In reality – the hardware does not look like this.

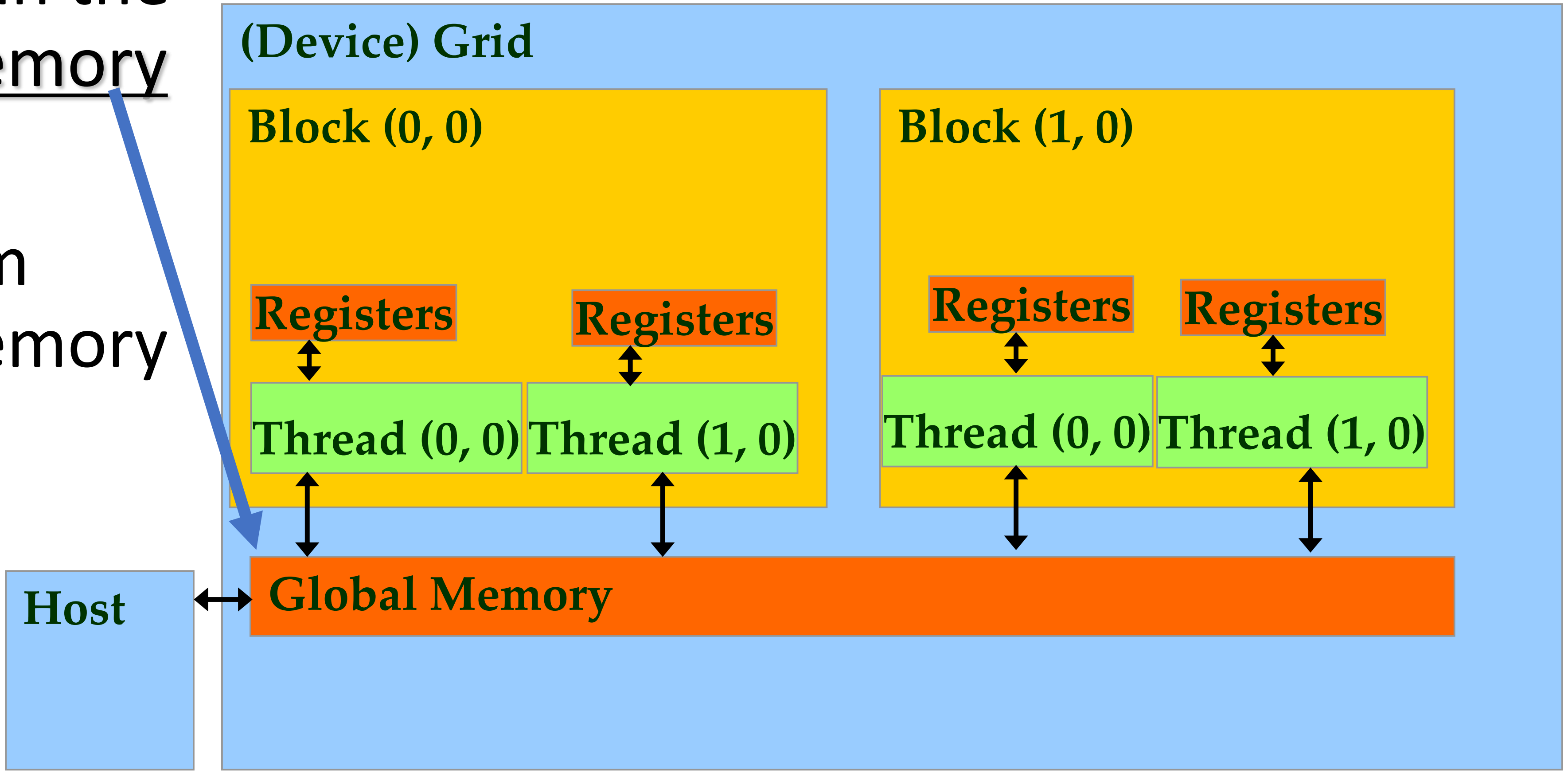
Memory Basics in CPU/GPU systems

- Each thread has it’s own private registers
  - We will see that these are really per-warp “vector registers” in the HW
- Every thread + CPU has R/W access to global device memory
  - Generally non-coherent



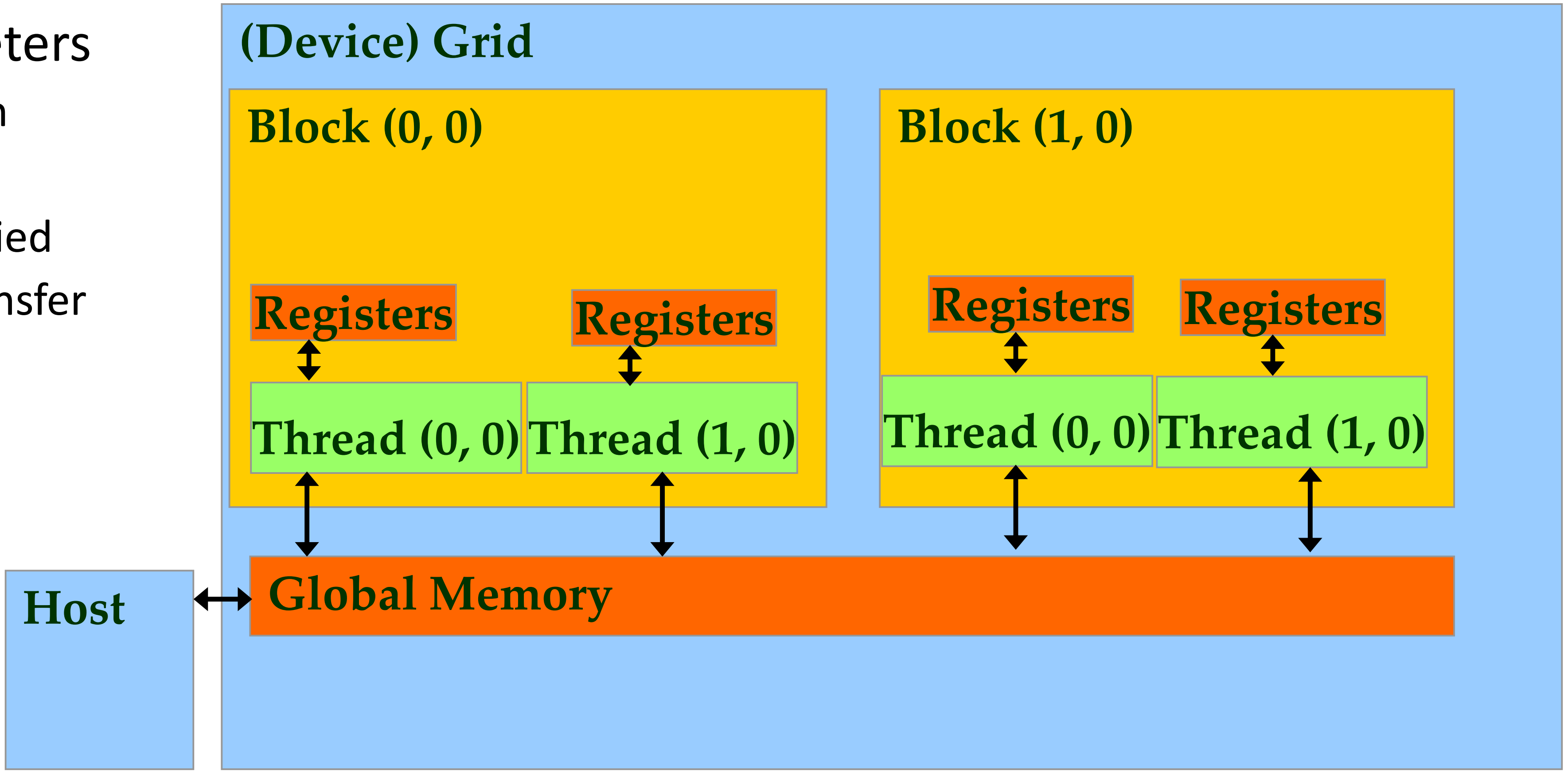
CUDA Memory Management Functions (called from CPU)

- cudaMalloc()
  - Allocates object in the device global memory
- cudaFree()
  - Frees object from device global memory



Host/Device Data transfer functions (called from CPU)

- cudaMemcpy()
  - memory data transfer
  - Requires four parameters
    - Pointer to destination
    - Pointer to source
    - Number of bytes copied
    - Type/Direction of transfer
  - Transfer to device is synchronous



CPU code with memory management

```
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 C
    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);
}
```

The Kernel

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/256.0), 256>>>(A_d, B_d, C_d, n);
}
```

The Kernel

```
// 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];
}
```

Host Code

```
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/256.0), 256>>>(A_d, B_d, C_d, n);
}
```

A little more on Kernel Launch

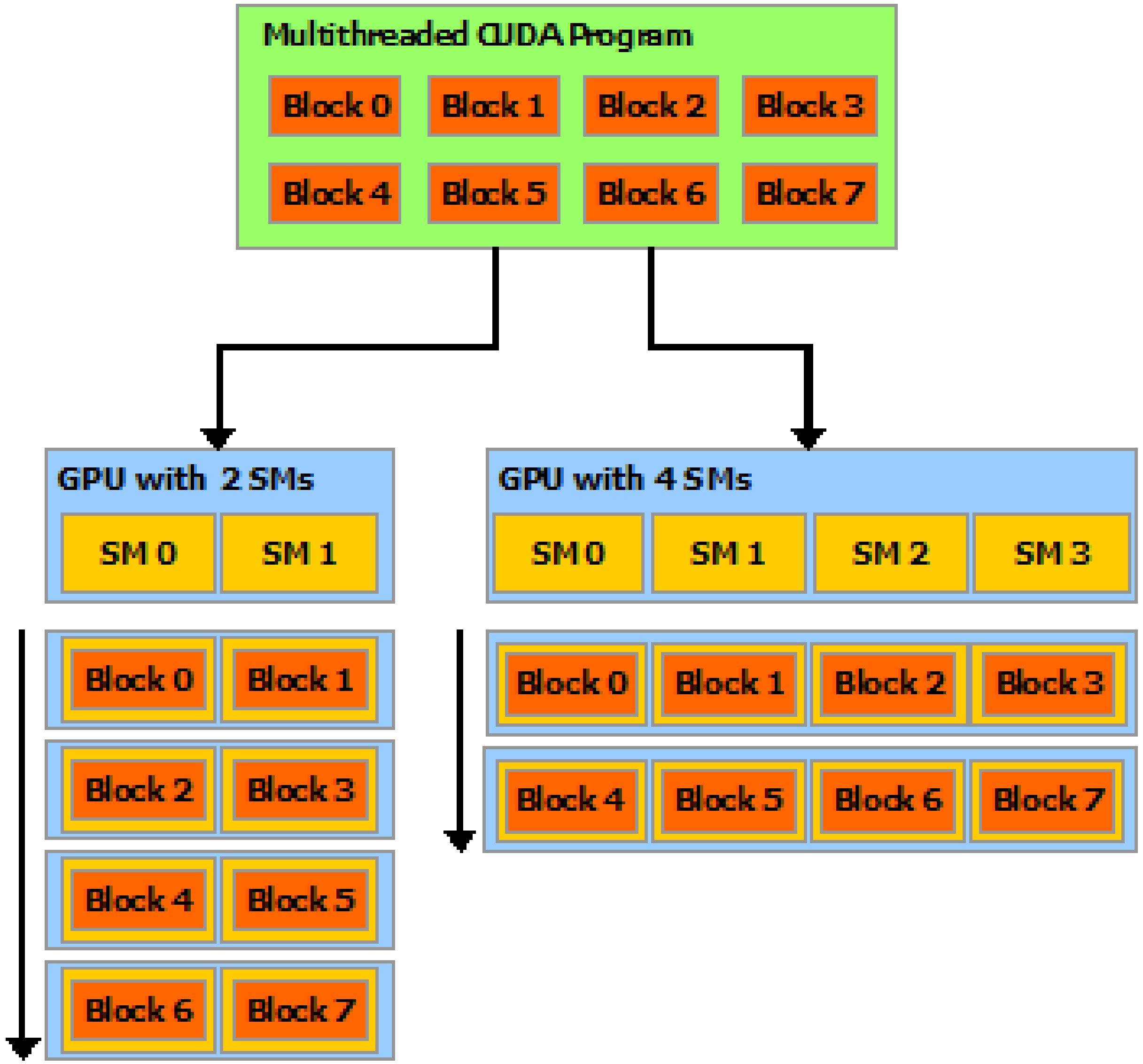
Host Code

```
int vecAdd(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
  dim3 DimGrid(n/256, 1, 1);
  if (n%256) DimGrid.x++;
  dim3 DimBlock(256, 1, 1);

  vecAddKernel<<<DimGrid,DimBlock>>>(A_d, B_d, C_d, n);
}
```

- Any call to a kernel function is asynchronous from CUDA 1.0 on, explicit synch needed for blocking

How kernel maps to a GPU



A Little More on Kernel Launch

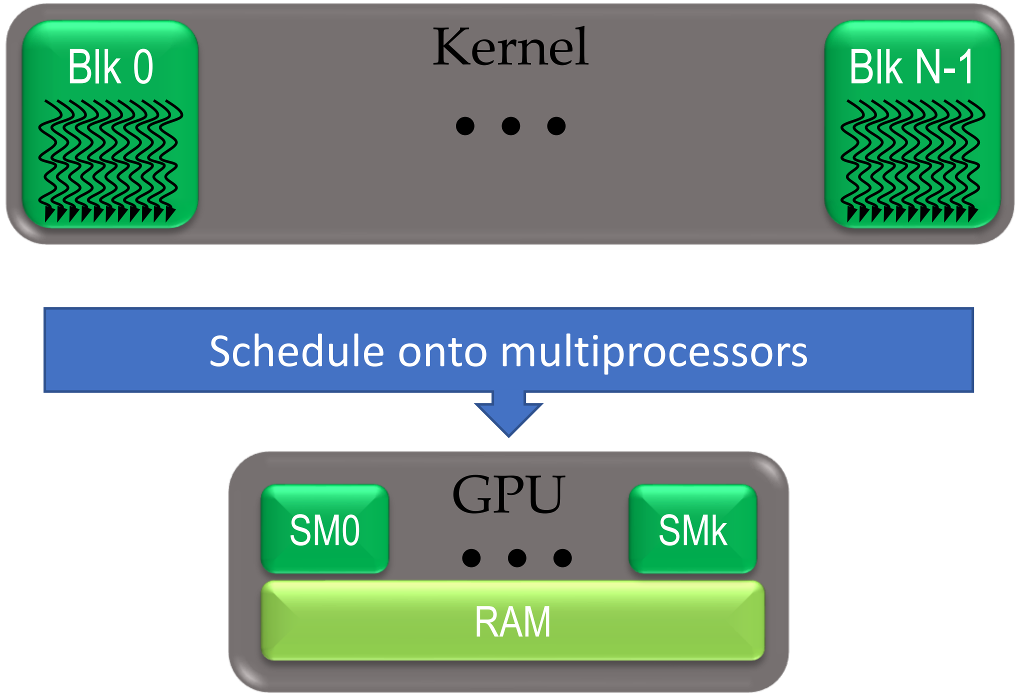
```
__host__
void vecAdd()
{
  dim3 DimGrid(ceil(n/256.0),1,1);
  dim3 DimBlock(256,1,1);

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];
}
```

The runtime software (user-level + device driver) will initiate the kernel on the GPU

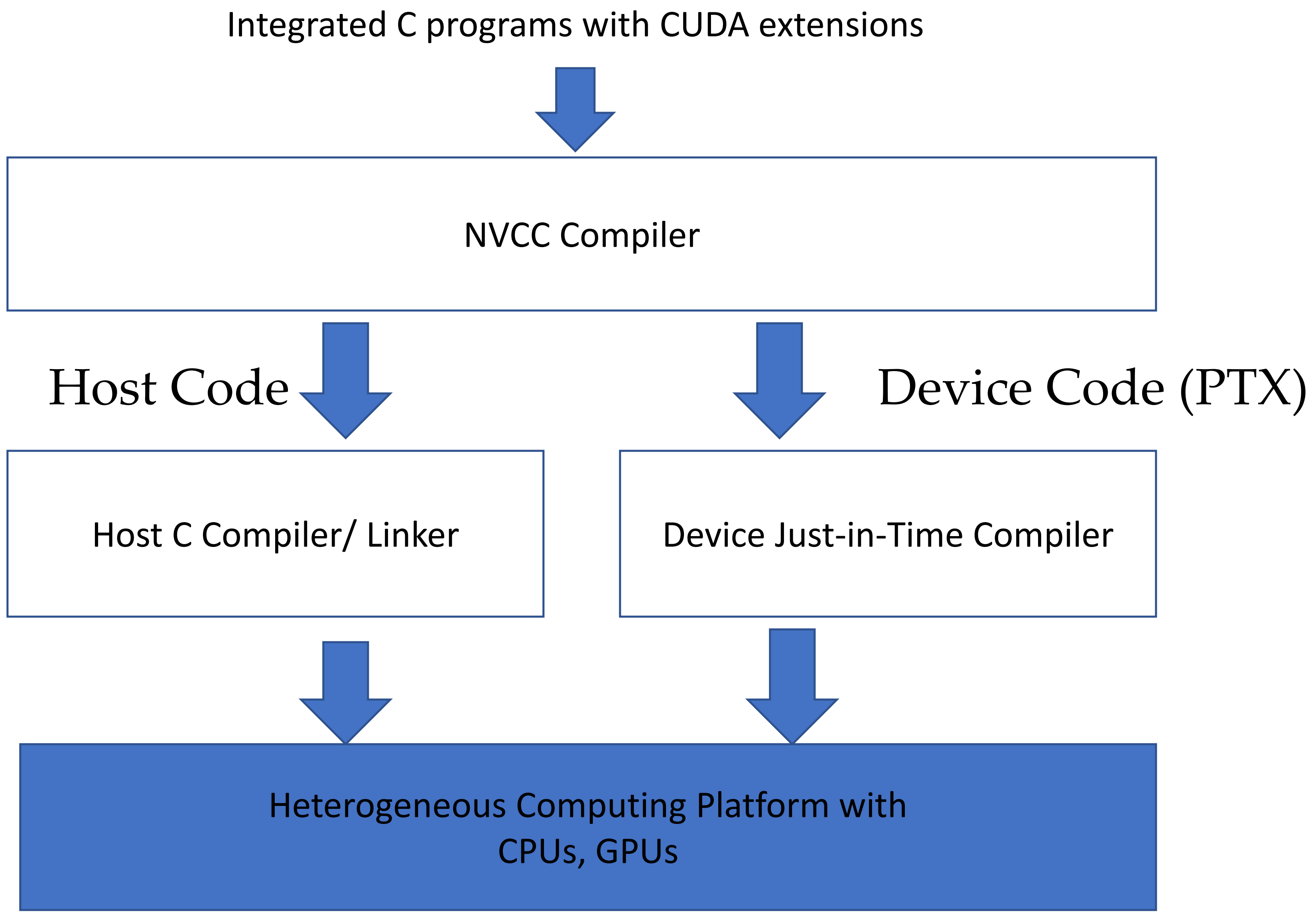


Declaring Functions in CUDA programs

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

- `__global__` defines a kernel function
  - Each "__" consists of two underscore characters
  - A kernel function must return `void`
- `__device__` and `__host__` can be used together

How the programs are compiled


Integrated C programs with CUDA extensions
NVCC Compiler
Host Code
Device Code (PTX)
Host C Compiler/ Linker
Device Just-in-Time Compiler
Heterogeneous Computing Platform with CPUs, GPUs

For Next Class

- GPU Simulation
  - Review Gutierrez 2018
  - Read Khairy 2018
  - References: Lew 2018, Bakhoda 2009
- HW0 Released
  - Let me know if you have issues accessing euler

Parting Thought

- Sources of irregularity in vectorAdd?
  - Thoughts on the implications of this?

```
__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];
}
```