



中山大學
SUN YAT-SEN UNIVERSITY



国家超级计算广州中心
NATIONAL SUPERCOMPUTER CENTER IN GUANGZHOU

CUDA编程模型

任课教师：吴迪、杜云飞

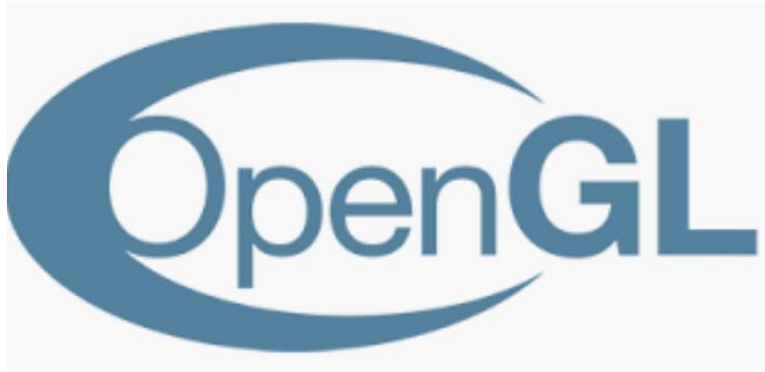
Roadmap

- Why CUDA?
- CUDA Programming Model
- The First CUDA Program
- Run in Parallel
- CUDA Threads
- More on CUDA Parallel Computing
- Cooperating Threads
- Managing the Device

Why CUDA?

Problems of Traditional GPUs - 1

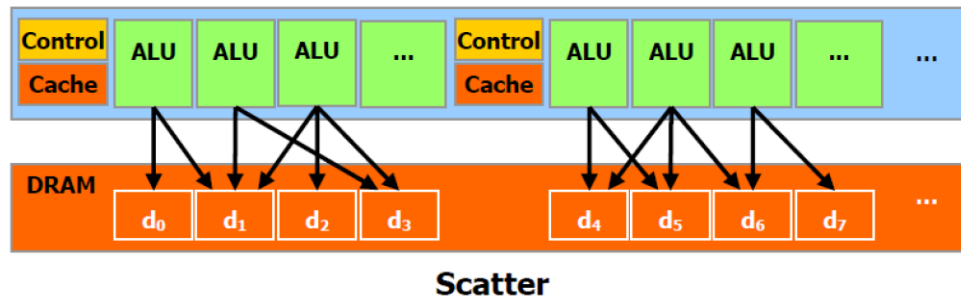
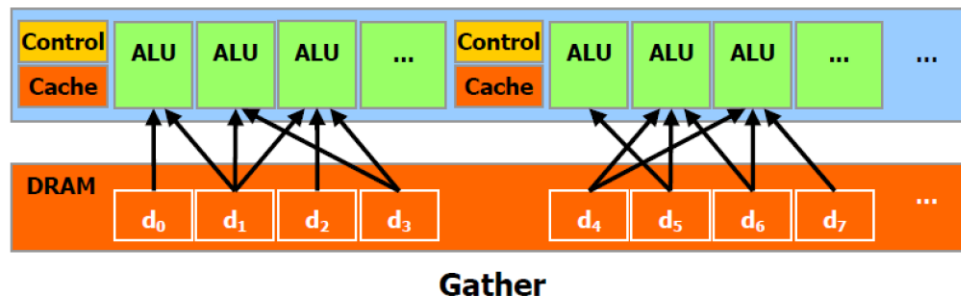
- Traditional GPUs could only be programmed through a **graphics API**
 - E.g., **OpenGL**, **DirectX**
 - imposing a high learning curve to the novice
 - overhead to non-graphics applications



Microsoft®
DirectX®

Problems of Traditional GPUs -2

- GPU programs **can gather** data elements from any part of DRAM
- But **cannot scatter** information to any part of DRAM



Removing a lot of the programming flexibility readily available on the CPU.

Problems of Traditional GPUs - 3

- Some applications were **bottlenecked** by the DRAM **memory bandwidth**
 - Underutilizing the GPU's computational power

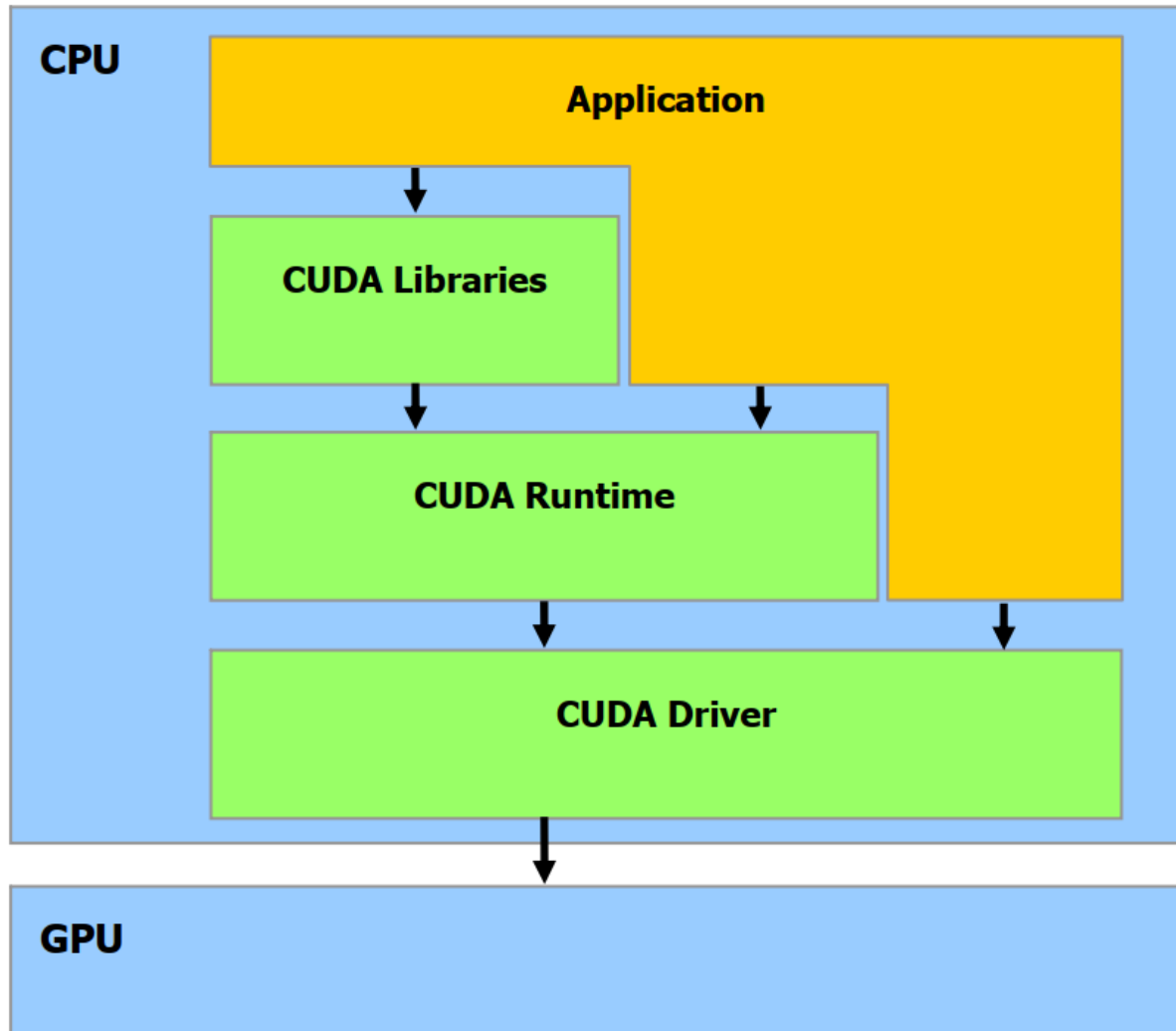


What is CUDA?



- **CUDA: Compute Unified Device Architecture**
- CUDA Architecture
 - Expose GPU parallelism for **general-purpose computing**
 - Retain performance
- CUDA C/C++
 - Based on **industry-standard C/C++**
 - Small set of extensions to enable heterogeneous programming
 - **Straightforward APIs** to manage devices, memory etc.

CUDA Software Stack



CUDA Major Components

Compiler (bin/)

The CUDA-C and CUDA-C++ compiler, `nvcc`, is built on top of the NVVM optimizer, which is itself built on top of the LLVM compiler infrastructure. Developers who want to target NVVM directly can do so using the Compiler SDK, which is available in the `nvvm/` directory.

Tools (bin/)

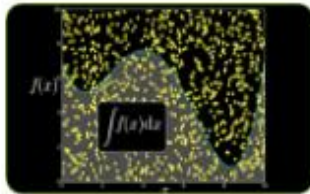
IDEs: nsight (Linux, Mac)
Debuggers: cuda-memcheck, cuda-gdb (Linux, Mac)
Profilers: nvprof, nvvp
Utilities: cuobjdump, nvdasm, gwiz

Libraries (lib/)

- | | |
|---|--|
| <ul style="list-style-type: none">▸ cublas (BLAS)▸ cuda_occupancy (Kernel Occupancy Calculation [header file implementation])▸ cudadevrt (CUDA Device Runtime)▸ cudart (CUDA Runtime)▸ cufft (Fast Fourier Transform [FFT])▸ cupti (Profiling Tools Interface) | <ul style="list-style-type: none">▸ curand (Random Number Generation)▸ cusolver (Dense and Sparse Direct Linear Solvers and Eigen Solvers)▸ cusparse (Sparse Matrix)▸ npp (NVIDIA Performance Primitives [image and signal processing])▸ nvcuvid (CUDA Video Decoder [Windows, Linux])▸ nvgraph (CUDA nvGRAPH [accelerated graph analytics])▸ nvml (NVIDIA Management Library) |
|---|--|

CUDA Libraries & Tools

Libraries



cuRAND



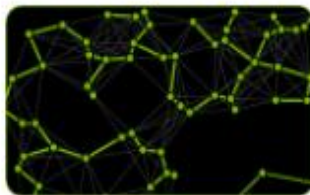
NPP



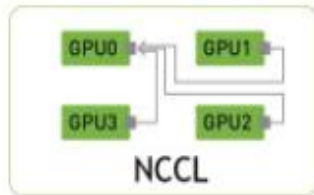
Math Library



cuFFT



nvGRAPH

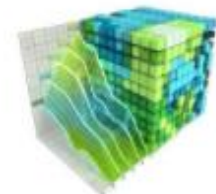


NCCL

Tools and Integrations



Nsight



Visual Profiler



CUDA GDB



CUDA MemCheck

OpenACC

OpenACC



CUDA Profiling Tools

CUDA Installation (1)

- Step 1: CUDA-capable GPU
- If your GPU is an NVIDIA card that is listed on the [CUDA-supported GPUs page](#), your GPU is CUDA-capable



CUDA-Enabled Tesla Products



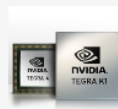
CUDA-Enabled Quadro Products



CUDA-Enabled NVS Products



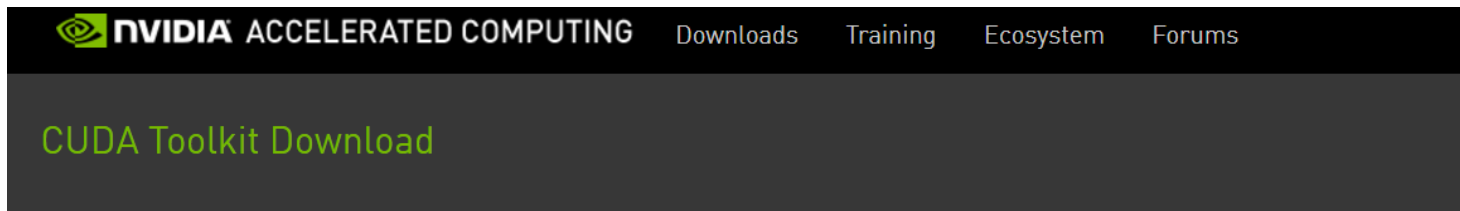
CUDA-Enabled GeForce Products




CUDA-Enabled TEGRA /Jetson Products

CUDA Installation (2)



- Step 2: Select platform



[Home](#) > [ComputeWorks](#) > [CUDA Toolkit](#) > [CUDA Toolkit Download](#)

Select Target Platform 

Click on the green buttons that describe your target platform. Only supported platforms will be shown.

Operating System	Windows	Linux	Mac OSX			
Architecture 	x86_64	ppc64le				
Distribution	Fedora	OpenSUSE	RHEL	CentOS	SLES	Ubuntu
Version	16.04	14.04				
Installer Type 	runfile (local)	deb (local)	deb (network)	cluster (local)		

CUDA Installation (3)

- Step 3: Install CUDA

1. Install the repository meta-data, update the apt-get cache, and install CUDA:

```
$ sudo dpkg --install cuda-repo-<distro>-<version>.<architecture>.deb  
$ sudo apt-get update  
$ sudo apt-get install cuda
```

2. Reboot the system to load the NVIDIA drivers.
3. Set up the development environment by modifying the PATH and LD_LIBRARY_PATH variables:

```
$ export PATH=/usr/local/cuda-8.0/bin${PATH:+:${PATH}}  
$ export LD_LIBRARY_PATH=/usr/local/cuda-8.0/lib64\  
${LD_LIBRARY_PATH:+:${LD_LIBRARY_PATH}}
```

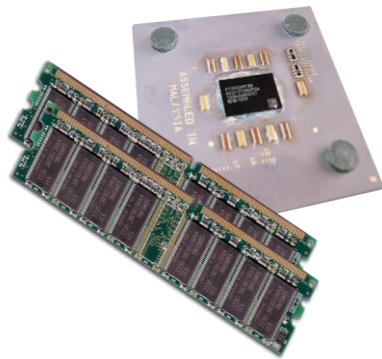
4. Install a writable copy of the samples then build and run the nbody sample:

```
$ cuda-install-samples-8.0.sh ~  
$ cd ~/NVIDIA_CUDA-8.0_Samples/5_Simulations/nbody  
$ make  
$ ./nbody
```

CUDA Programming Model

Heterogeneous Computing

- Terminology:
 - *Host* The **CPU** and its memory (host memory)
 - *Device* The **GPU** and its memory (device memory)



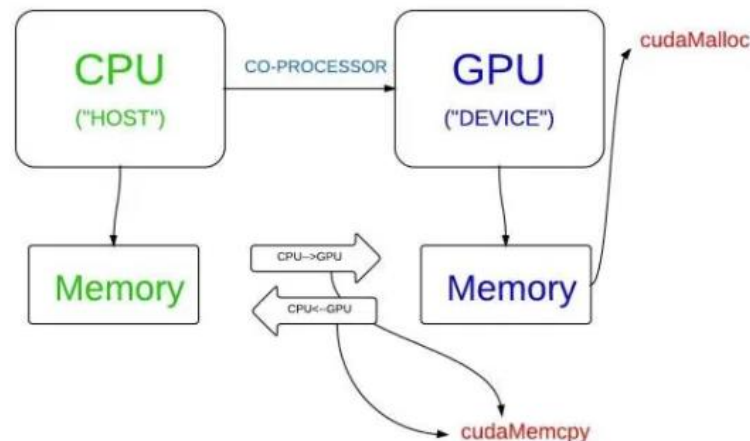
Host



Device

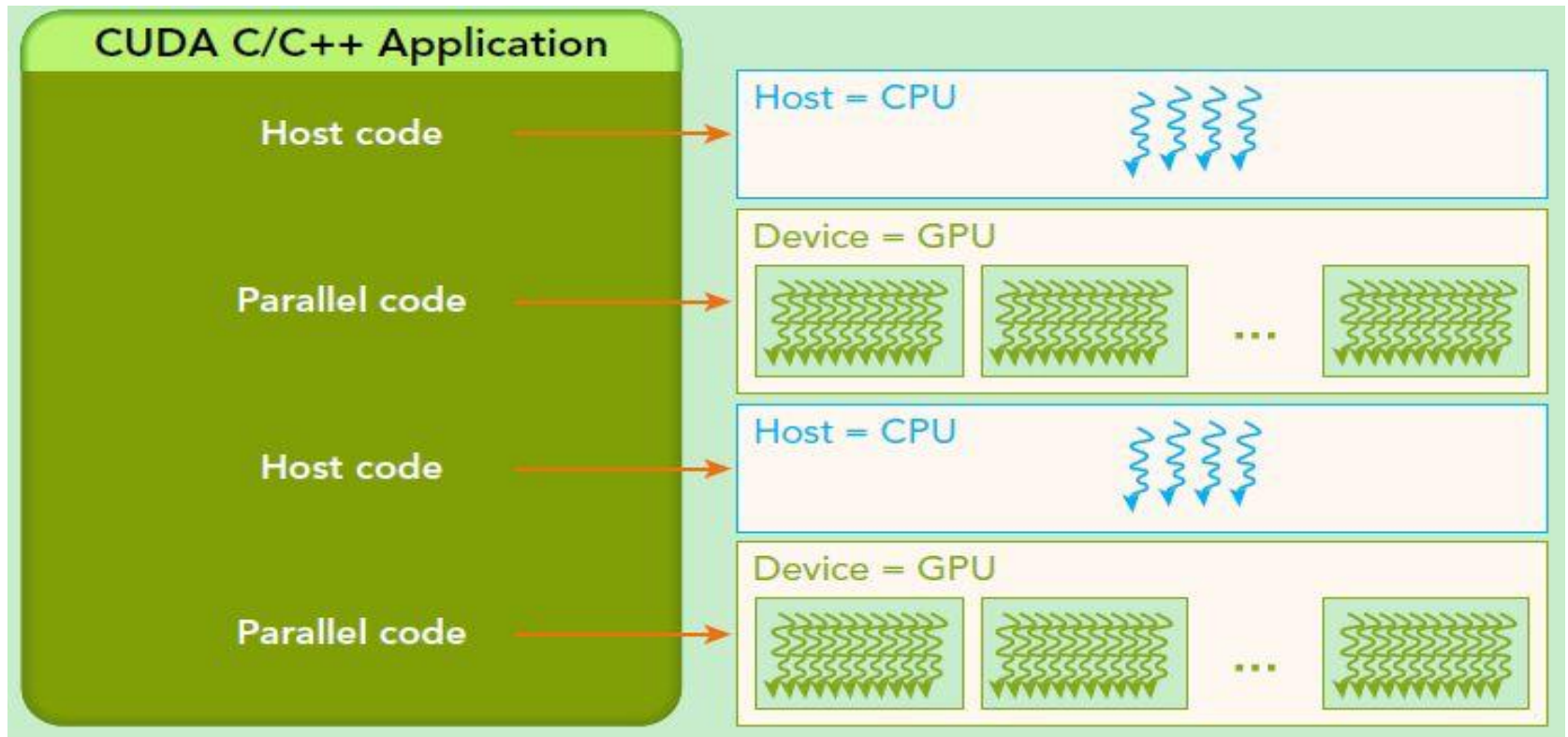
CUDA Programming Model

- In CUDA, **GPU** is viewed as *compute device* capable of executing a very high number of threads in parallel
- GPU operates as a coprocessor to the main **CPU** (*called host*)
- Compute-intensive portions of applications running on the host are **off-loaded** onto the device.

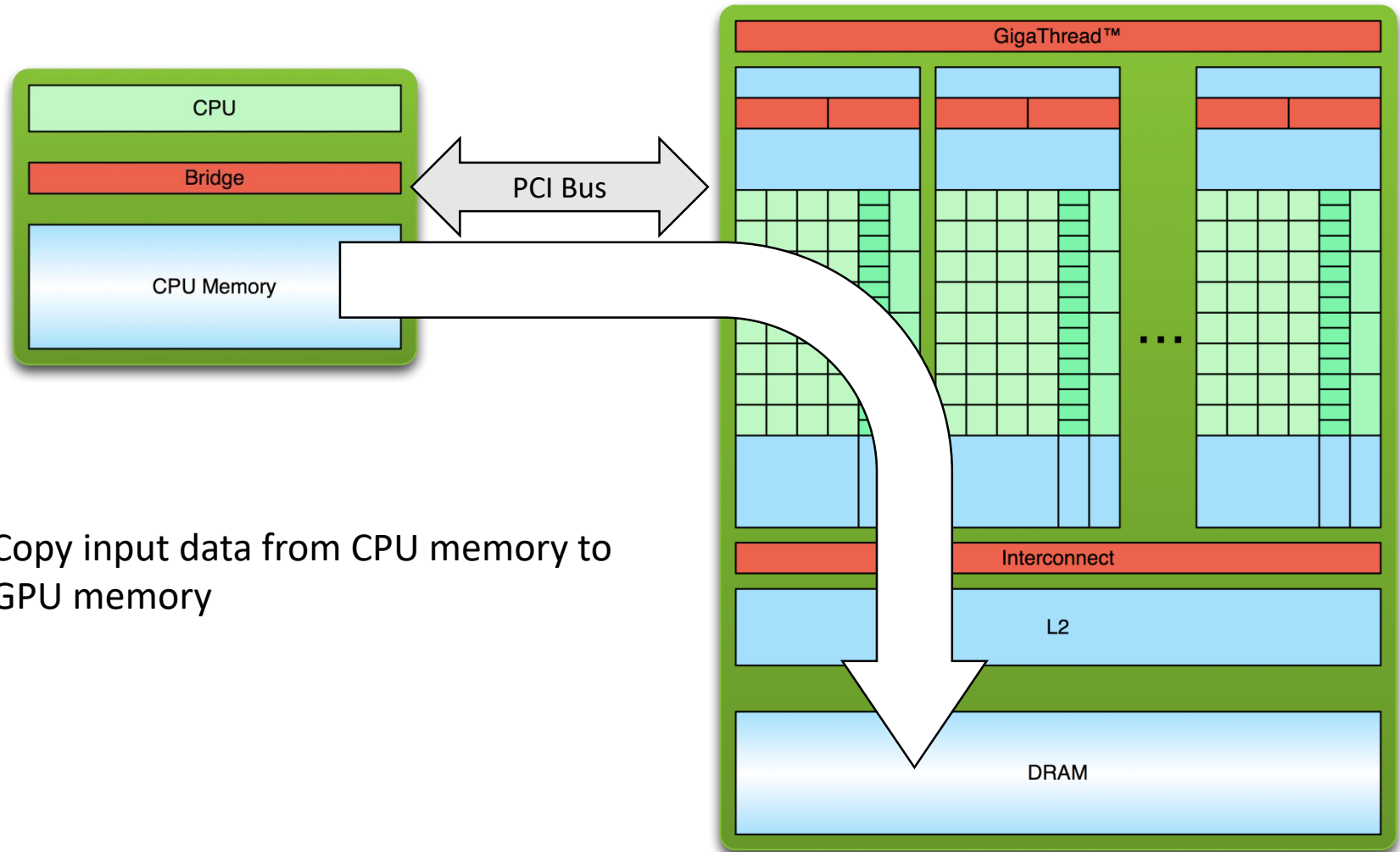


CUDA Programming Model

- CUDA program: **host code + device code**

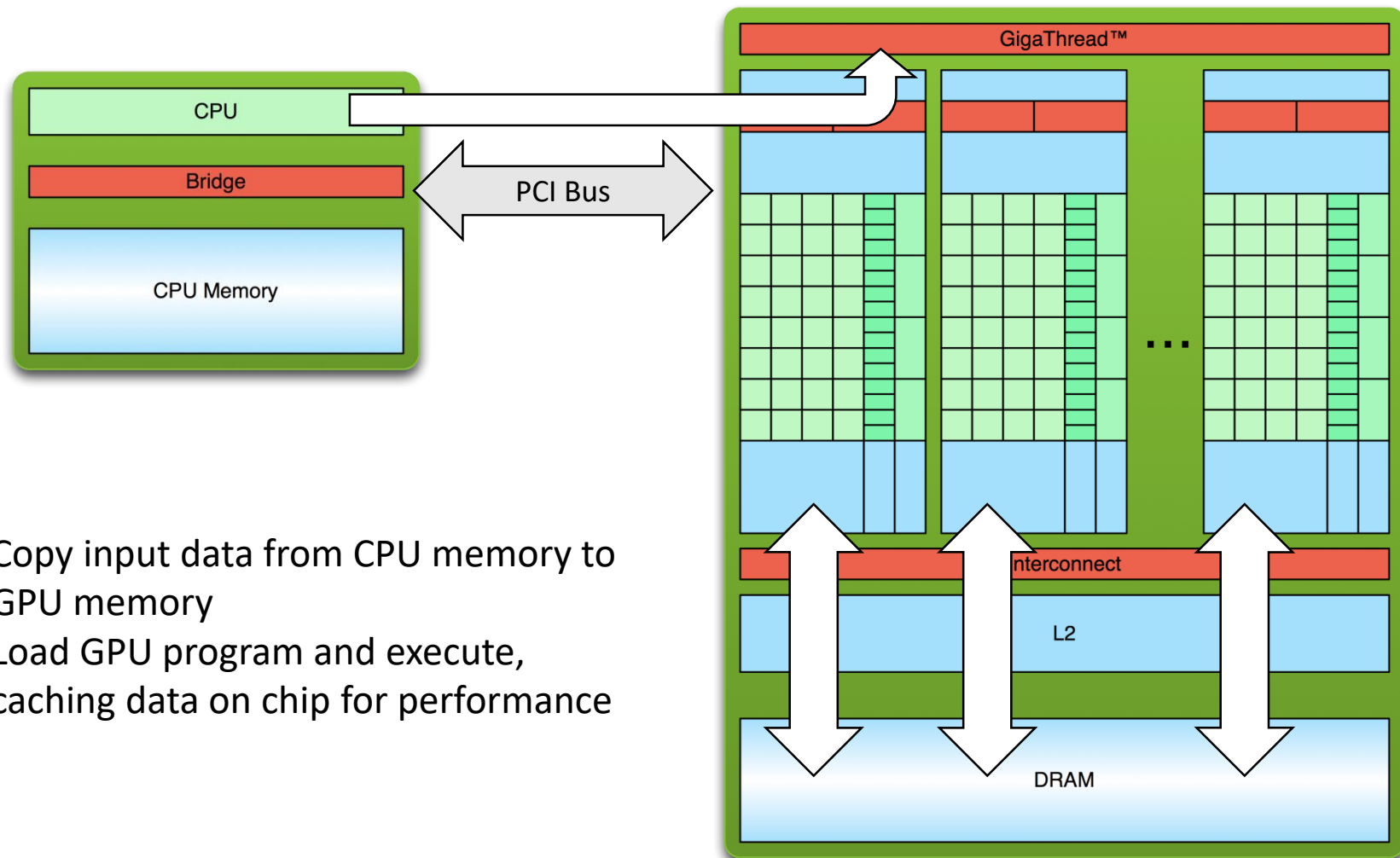


Simple Processing Flow



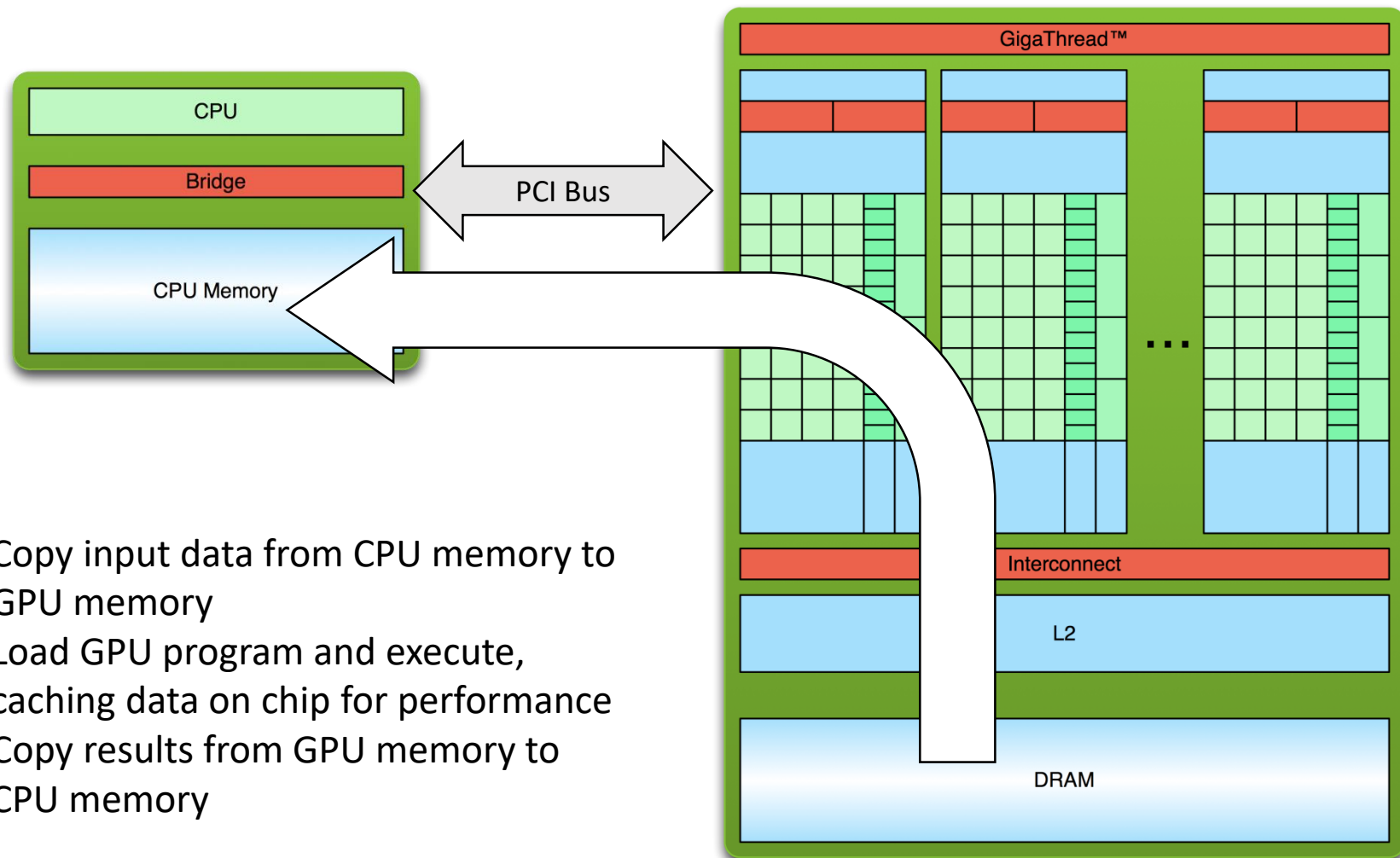
1. Copy input data from CPU memory to GPU memory

Simple Processing Flow



1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance

Simple Processing Flow



1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
3. Copy results from GPU memory to CPU memory

The First CUDA Program

Hello World!

```
int main(void) {  
    printf("Hello World!\n");  
    return 0;  
}
```

- Standard C that runs on the host
- NVIDIA compiler (**nvcc**) can be used to compile programs with no *device* code

Output:

```
$ nvcc  
hello_world.cu  
$ a.out  
Hello World!  
$
```

Hello World! with Device Code

```
__global__ void mykernel(void) {  
}
```

```
int main(void) {  
    mykernel<<<1,1>>>();  
    printf("Hello World!\n");  
    return 0;  
}
```

- Two new syntactic elements...

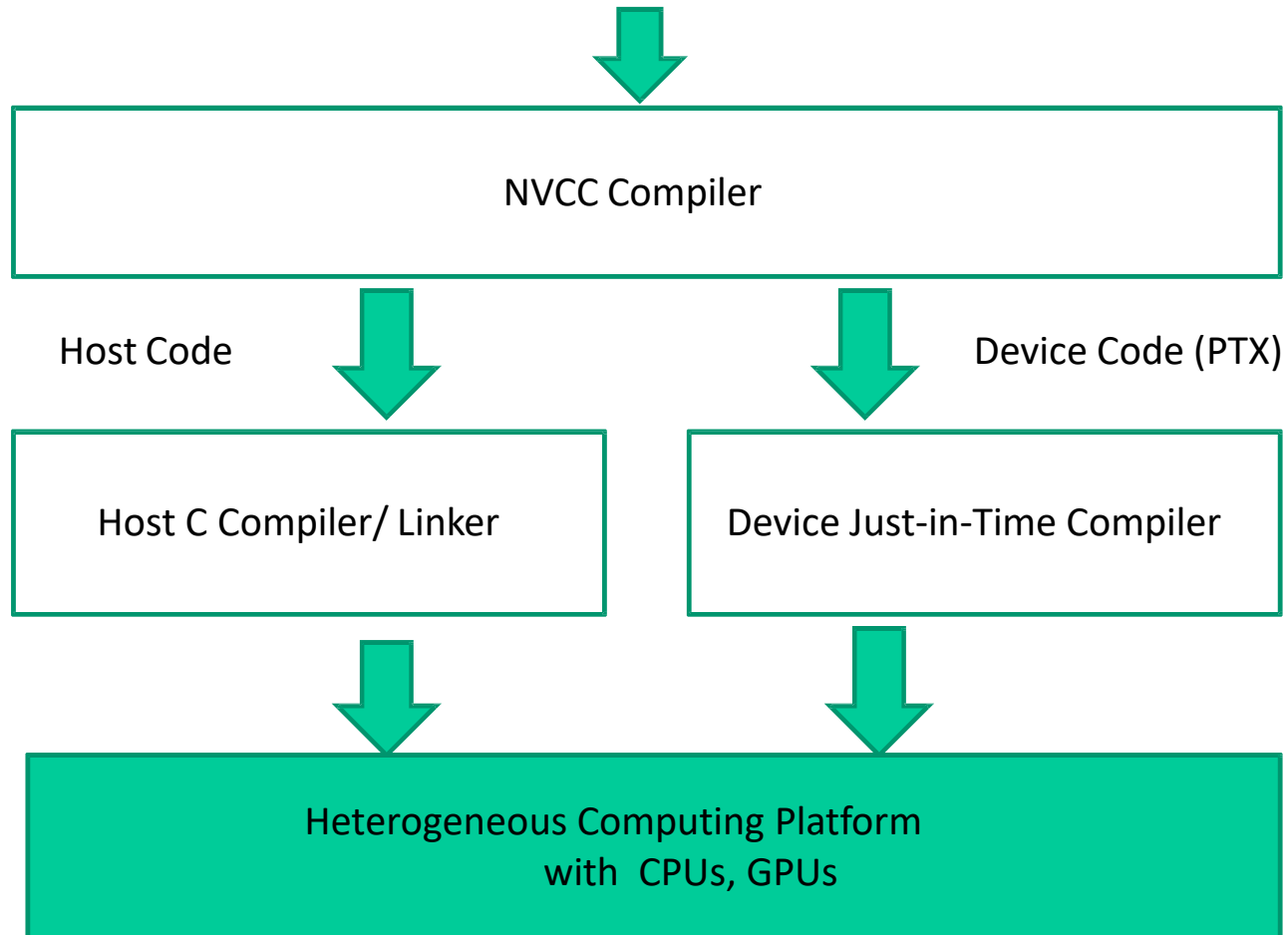
Hello World! with Device Code

```
__global__ void mykernel(void) {  
}
```

- CUDA C/C++ keyword `__global__` indicates a function that:
 - Runs on the **device**
 - Is called from host code
- `nvcc` **separates** source code into host and device components
 - Device functions (e.g. `mykernel()`) processed by NVIDIA compiler
 - Host functions (e.g. `main()`) processed by standard host compiler
 - `gcc, cl.exe`

CUDA Compiler

Integrated C programs with CUDA extensions
(***.cu** files)



Hello World! with Device C0de

```
mykernel<<<1,1>>>() ;
```

- Triple angle brackets mark a call from *host* code to *device* code
 - Also called a “**kernel launch**”
 - We’ll return to the parameters (1,1) in a moment
- That’s **all** that is required to execute a function on the GPU!

Hello World! with Device Code

```
__global__ void mykernel(void) {  
}
```

```
int main(void) {  
    mykernel<<<1,1>>>();  
    printf("Hello World!\n");  
    return 0;  
}
```

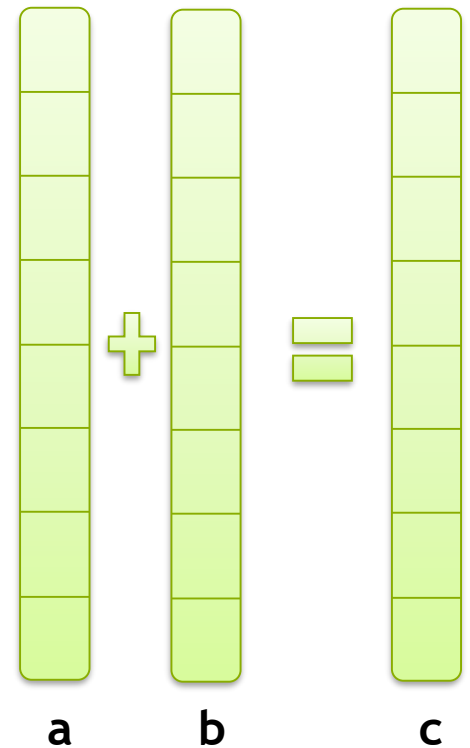
Output:

```
$ nvcc  
hello.cu  
$ a.out  
Hello World!  
$
```

- `mykernel()` does nothing,
somewhat **anticlimactic (虎头蛇尾)!**

Parallel Programming in CUDA

- But wait... GPU computing is about **massive parallelism**!
- We need a more **interesting** example...
- We'll start by adding two integers and build up to **vector addition**



Addition on the Device

- A simple kernel to add two integers

```
__global__ void add(int *a, int *b, int *c) {  
    *c = *a + *b;  
}
```

- As before `__global__` is a CUDA C/C++ keyword meaning
 - `add()` will execute on the device
 - `add()` will be called from the host

Addition on the Device

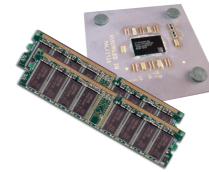
- Note that we use pointers for the variables

```
__global__ void add(int *a, int *b, int *c) {  
    *c = *a + *b;  
}
```

- **add()** runs on the device, so *a*, *b* and *c* must point to **device memory**
- We need to allocate memory on the GPU

Memory Management

- Host and device memory are separate entities
 - *Device* pointers point to GPU memory
 - May be passed to/from host code
 - May *not* be dereferenced in host code
 - *Host* pointers point to CPU memory
 - May be passed to/from device code
 - May *not* be dereferenced in device code
- Simple CUDA API for handling device memory
 - `cudaMalloc()`, `cudaFree()`, `cudaMemcpy()`
 - Similar to the C equivalents `malloc()`, `free()`, `memcpy()`



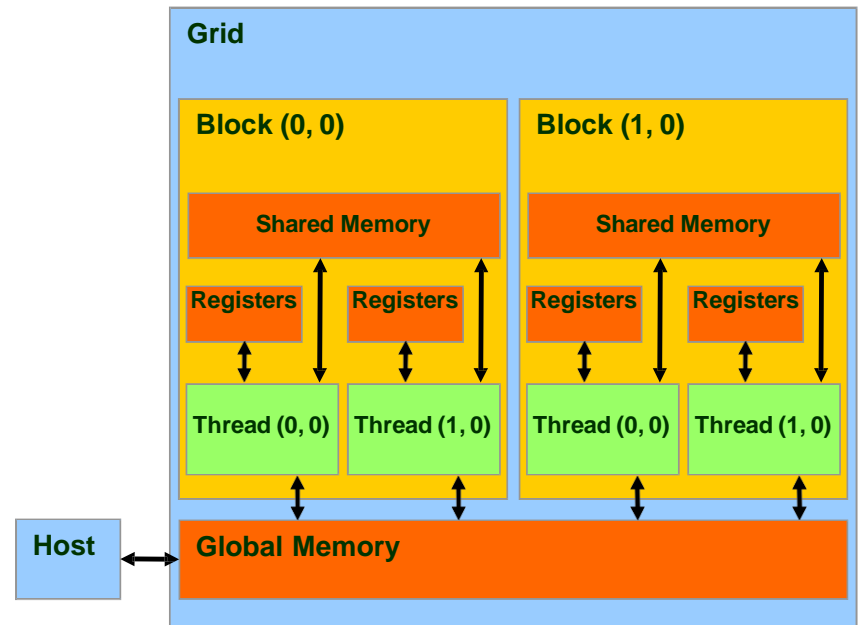
CUDA Memory Model

Global memory

- Main means of communicating R/W data between **host** and **device**
- Contents visible to all threads
- Long latency access

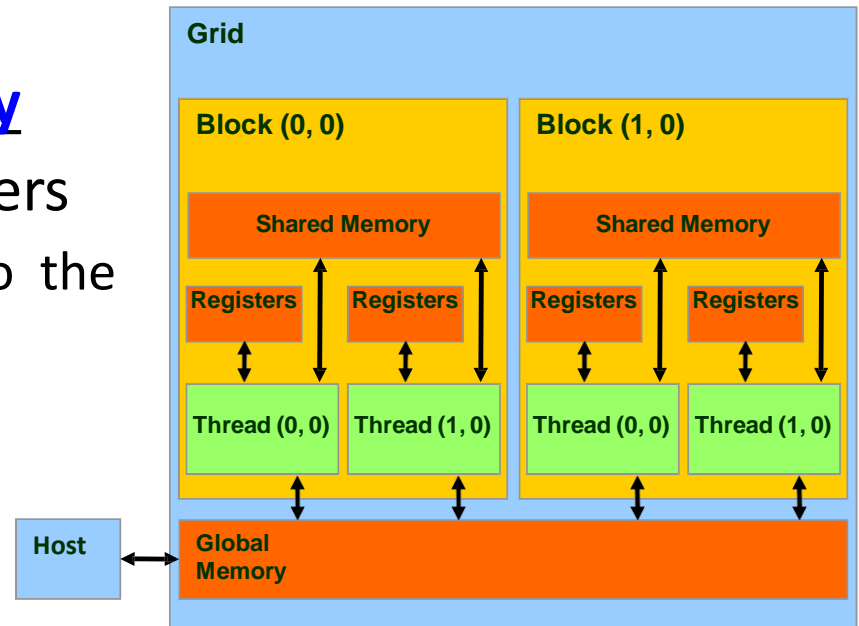
Device code can:

- R/W per-thread registers
- R/W per-grid global memory



CUDA Device Memory Allocation

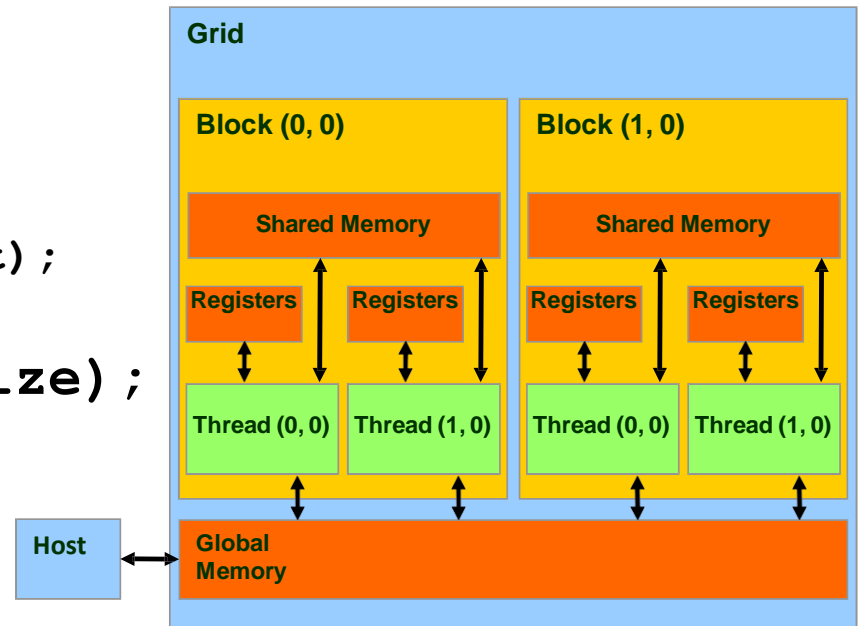
- **cudaMalloc()**
 - Allocates object in the device **Global Memory**
 - Requires two parameters
 - **Address of a pointer** to the allocated object
 - **Size of** allocated object
- **cudaFree()**
 - Free object from device Global Memory
 - Pointer to freed object



CUDA Device Memory Allocation

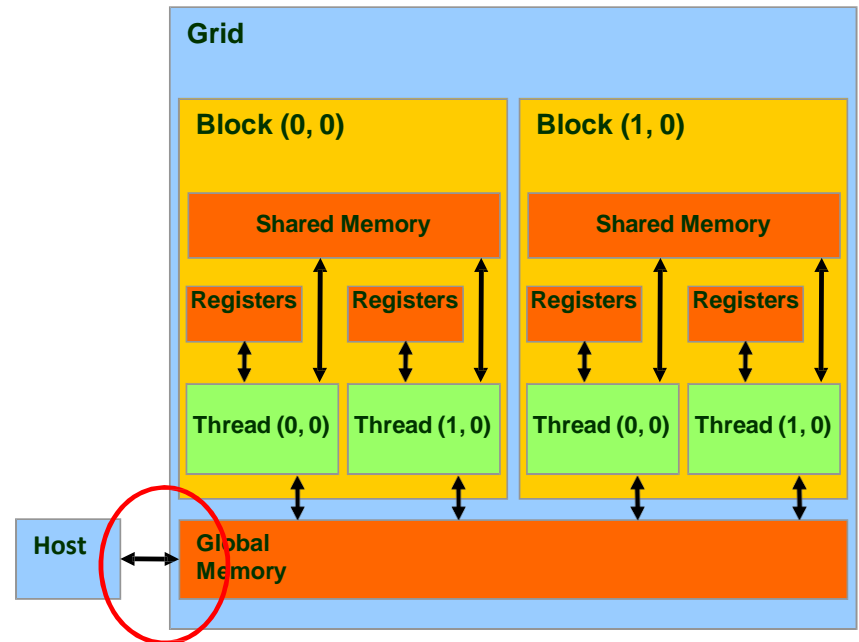
Example:

```
WIDTH = 64;  
float *      Md;  
int size = WIDTH * sizeof(float);  
  
cudaMalloc( (void**) &Md, size);  
cudaFree (Md);
```



CUDA Device Memory Allocation

- **cudaMemcpy()**
 - memory data transfer
 - Requires four parameters
 - Pointer to destination
 - Pointer to source
 - Number of bytes copied
 - Type of transfer
 - Host to Host
 - Host to Device
 - Device to Host
 - Device to Device
- Asynchronous transfer



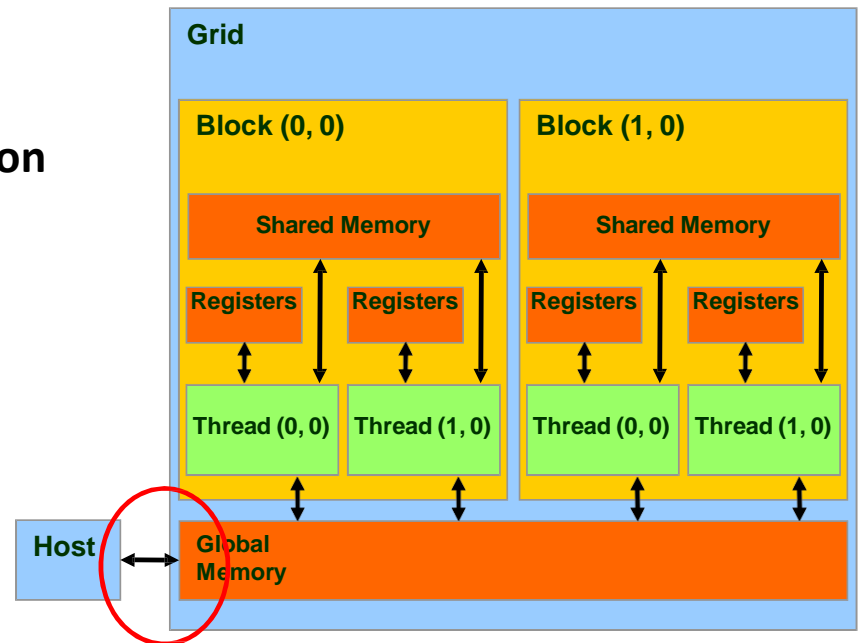
CUDA Device Memory Allocation

Example:

Destination pointer Source pointer Size in bytes Direction

`cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);`

`cudaMemcpy(M, Md, size, cudaMemcpyDeviceToHost);`



Addition on the Device: add ()

- Returning to our **add ()** kernel

```
__global__ void add(int *a, int *b, int *c) {  
    *c = *a + *b;  
}
```

- Let's take a look at main()...

Addition on the Device: main ()

```
int main(void) {  
    int a, b, c;           // host copies of a, b, c  
    int *d_a, *d_b, *d_c; // device copies of a, b, c  
    int size = sizeof(int);  
  
    // Allocate space for device copies of a, b, c  
    cudaMalloc((void **)&d_a, size);  
    cudaMalloc((void **)&d_b, size);  
    cudaMalloc((void **)&d_c, size);  
  
    // Setup input values  
    a = 2;  
    b = 7;
```

Addition on the Device: `main()`

```
// Copy inputs to device
```

```
cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);
```

```
cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);
```

```
// Launch add() kernel on GPU
```

```
add<<<1,1>>>(d_a, d_b, d_c);
```

```
// Copy result back to host
```

```
cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);
```

```
// Cleanup
```

```
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
```

```
return 0;
```

```
}
```

Steps in a Typical CUDA program

A typical CUDA program contains 5 steps:

1. Allocate GPU memory space
2. Copy data from CPU to GPU
3. Dispatch a kernel to perform calculation
4. Copy results from GPU back to CPU
5. Release GPU memory space

Run in Parallel

Moving to Parallel

- GPU computing is about massive parallelism
 - So how do we run code in parallel on the device?

```
add<<< 1, 1 >>> () ;
```



```
add<<< N, 1 >>> () ;
```

- Instead of executing `add ()` once, execute N times in parallel

Vector Addition on the Device

- With **add()** running in parallel we can do vector addition
- Each parallel invocation of **add()** is referred to as a **block**
 - The set of blocks is referred to as a **grid**
 - Each invocation can refer to its block index using **blockIdx.x**

```
__global__ void add(int *a, int *b, int *c) {  
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];  
}
```

- By using **blockIdx.x** to index into the array, each block handles a different index

Vector Addition on the Device

```
__global__ void add(int *a, int *b, int *c) {  
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];  
}
```

- On the device, each block can execute in parallel:

Block 0

```
c[0] = a[0] + b[0];
```

Block 1

```
c[1] = a[1] + b[1];
```

Block 2

```
c[2] = a[2] + b[2];
```

Block 3

```
c[3] = a[3] + b[3];
```

Vector Addition on the Device: `add()`

- Returning to our parallelized **`add()`** kernel

```
__global__ void add(int *a, int *b, int *c) {  
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];  
}
```

- Let's take a look at `main()`...

Vector Addition on the Device: `main()`

```
#define N 512
int main(void) {
    int *a  *b  *c           // host copies of a, b, c
    int *d_a, *d_b, *d_c;    // device copies of a, b, c
    int size = N * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    // Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);
```

Vector Addition on the Device: `main()`

```
// Copy inputs to device
```

```
cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
```

```
cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);
```

```
// Launch add() kernel on GPU with N blocks
```

```
add<<<N,1>>>(d_a, d_b, d_c);
```

```
// Copy result back to host
```

```
cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);
```

```
// Cleanup
```

```
free(a); free(b); free(c);
```

```
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
```

```
return 0;
```

```
}
```

Review (1 of 2)

- Difference between *host* and *device*
 - *Host* CPU
 - *Device* GPU
- Using `__global__` to declare a function as device code
 - Executes on the device
 - Called from the host
- Passing parameters from host code to a device function

Review (2 of 2)

- Basic device memory management
 - `cudaMalloc()`
 - `cudaMemcpy()`
 - `cudaFree()`
- Launching parallel kernels
 - Launch **N** copies of `add()` with `add<<<N, 1>>>(...)` ;
 - Use `blockIdx.x` to access block index

CUDA Threads

CUDA Threads

- A block can be split into parallel *threads*
- Let's change `add()` to use parallel *threads* instead of parallel *blocks*

```
__global__ void add(int *a, int *b, int *c) {  
    c[threadIdx.x] = a[threadIdx.x] + b[threadIdx.x];  
}
```

- We use `threadIdx.x` instead of `blockIdx.x`
- Need to make one change in `main()` ...

Vector Addition Using Threads: main ()

```
#define N 512

int main(void) {
    int *a, *b, *c;           // host copies of a, b, c
    int *d_a, *d_b, *d_c;    // device copies of a, b, c
    int size = N * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    // Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);
```

Vector Addition Using Threads: `main()`

```
// Copy inputs to device
cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

// Launch add() kernel on GPU with N threads
add<<<1,N>>>(d_a, d_b, d_c);

// Copy result back to host
cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

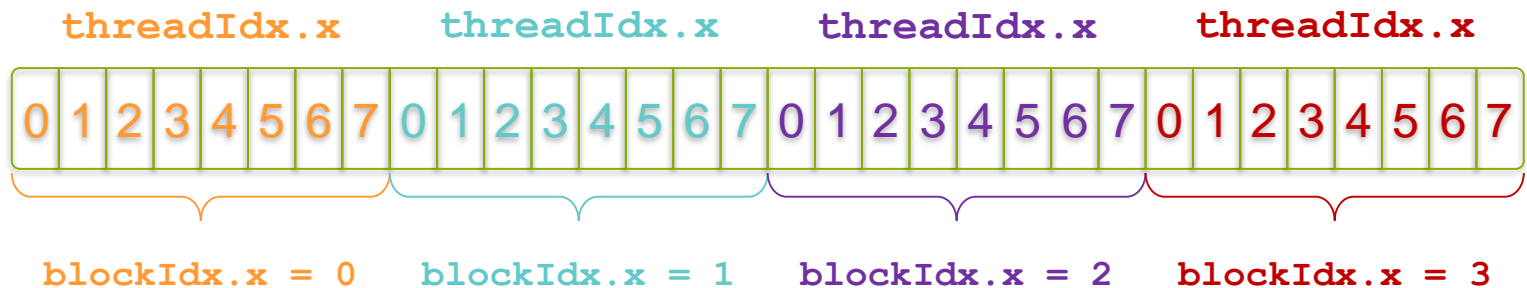
// Cleanup
free(a); free(b); free(c);
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
return 0;
}
```

Combining Blocks and Threads

- We've seen parallel vector addition using:
 - Many blocks with one thread each
 - One block with many threads
- Let's adapt vector addition to use both blocks and threads
- Why? We'll come to that...
- First let's discuss data indexing...

Indexing Arrays with Blocks/Threads

- No longer as simple as using `blockIdx.x` and `threadIdx.x`
 - Consider indexing an array with one element per thread (8 threads/block)

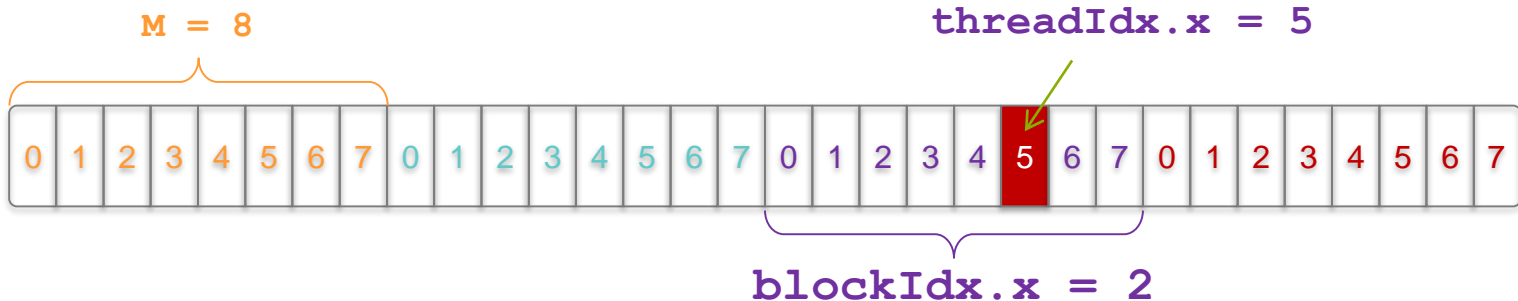


- With **M** threads/block a unique index for each thread is given by:

```
int index = threadIdx.x + blockIdx.x * M;
```

Indexing Arrays: Example

- Which thread will operate on the red element?



```
int index = threadIdx.x + blockIdx.x * M;  
          =           5      +           2      * 8;  
          = 21;
```


Vector Addition with Blocks and Threads

- Use the **built-in variable** `blockDim.x` for threads per block

```
int index = threadIdx.x + blockIdx.x * blockDim.x;
```

- Combined version of `add()` to use parallel threads *and* parallel blocks

```
__global__ void add(int *a, int *b, int *c) {  
    int index = threadIdx.x + blockIdx.x * blockDim.x;  
    c[index] = a[index] + b[index];  
}
```

- What changes need to be made in `main()`?

Addition w/ Blocks & Threads: main()

```
#define N (2048*2048)
```

```
#define THREADS_PER_BLOCK 512
```

```
int main(void) {
```

```
    int *a, *b, *c;           // host copies of a, b, c
```

```
    int *d_a, *d_b, *d_c;     // device copies of a, b, c
```

```
    int size = N * sizeof(int);
```

```
    // Alloc space for device copies of a, b, c
```

```
    cudaMalloc((void **)&d_a, size);
```

```
    cudaMalloc((void **)&d_b, size);
```

```
    cudaMalloc((void **)&d_c, size);
```

```
    // Alloc space for host copies of a, b, c and setup input values
```

```
    a = (int *)malloc(size); random_ints(a, N);
```

```
    b = (int *)malloc(size); random_ints(b, N);
```

```
    c = (int *)malloc(size);
```

Addition w/ Blocks & Threads: main()

```
// Copy inputs to device
```

```
cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
```

```
cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);
```

```
// Launch add() kernel on GPU
```

```
add<<<N/THREADS_PER_BLOCK, THREADS_PER_BLOCK>>>(d_a, d_b, d_c);
```

```
// Copy result back to host
```

```
cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);
```

```
// Cleanup
```

```
free(a); free(b); free(c);
```

```
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
```

```
return 0;
```

```
}
```

Handling Arbitrary Vector Sizes

- Typical problems are **not friendly multiples** of **blockDim.x**
- **Avoid accessing beyond** the end of the arrays:

```
__global__ void add(int *a, int *b, int *c, int n) {  
    int index = threadIdx.x + blockIdx.x * blockDim.x;  
    if (index < n)  
        c[index] = a[index] + b[index];  
}
```

- Update the kernel launch:

```
add<<<(N + M-1) / M, M >>>(d_a, d_b, d_c, N);
```

Why Bother with Threads?

- Threads seem unnecessary
 - They add a level of complexity
 - What do we gain?
- Unlike parallel blocks, **threads** have mechanisms to:
 - Communicate
 - Synchronize
- To look closer, we need a new example...

Review

- Launching parallel kernels
 - Launch N copies of `add()` with `add<<<N/M,M>>>(...)` ;
 - Use `blockIdx.x` to access block index
 - Use `threadIdx.x` to access thread index within block
- Allocate elements to threads:

```
int index = threadIdx.x + blockIdx.x * blockDim.x;
```

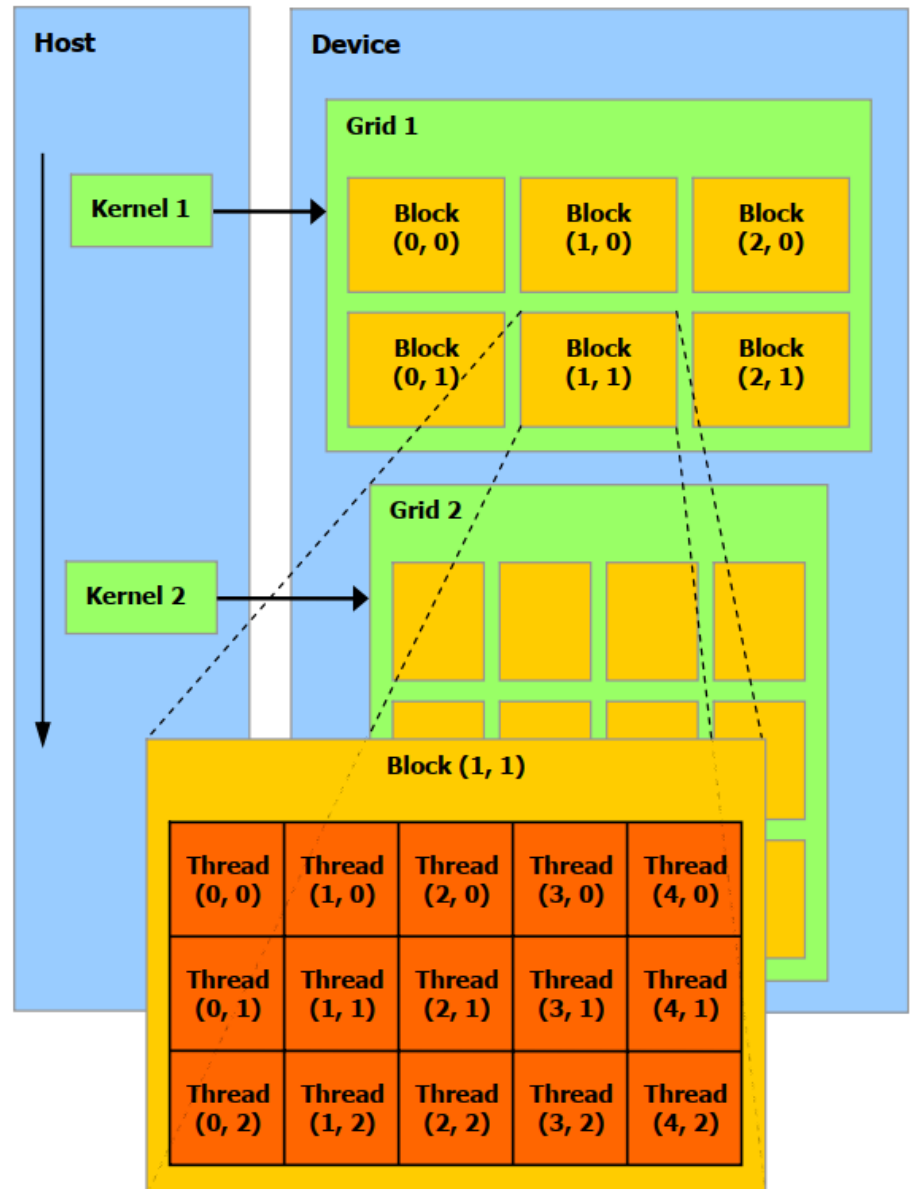
More on CUDA Parallel Computing

Recall: CUDA Kernel

- The parallel portion of an **application** can be isolated into a ***function*** that is executed on the device as many different ***threads***.
- Such a ***function*** is compiled to **the instruction set of the device** and the ***resulting program***, called a ***kernel***
- The batch of threads that executes a kernel is organized as a **grid** of thread blocks

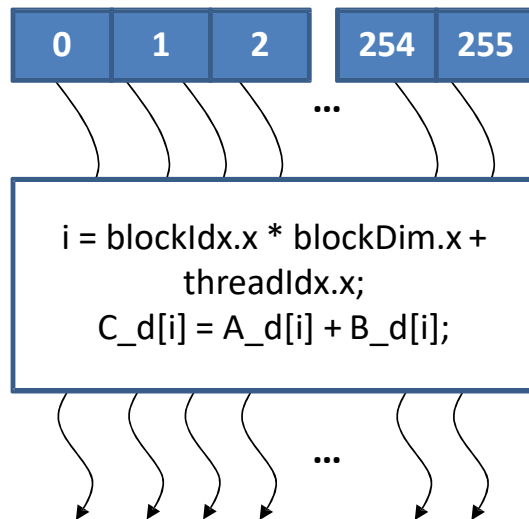
Thread Batching

- The host issues a succession of kernel invocations to the device.
- Each kernel is executed as a batch of threads organized as a grid of thread blocks



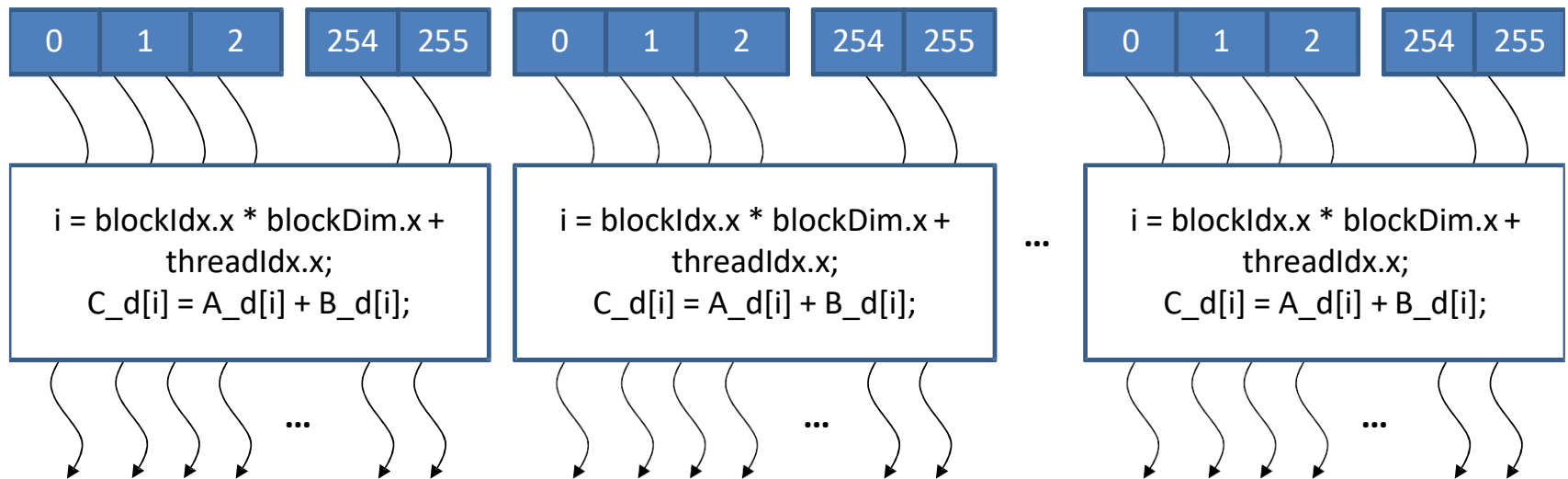
Block

- **Block**: a batch of threads that can **cooperate** together by **sharing data** and **synchronizing** their execution.
 - All threads run the same code (the SP in SPMD)
 - Each thread has an ID that it uses to compute memory addresses and make control decisions



Grid

- **Grid**: a set of blocks with same dimensionality and size that execute the same kernel can be **batched together** into a **grid**



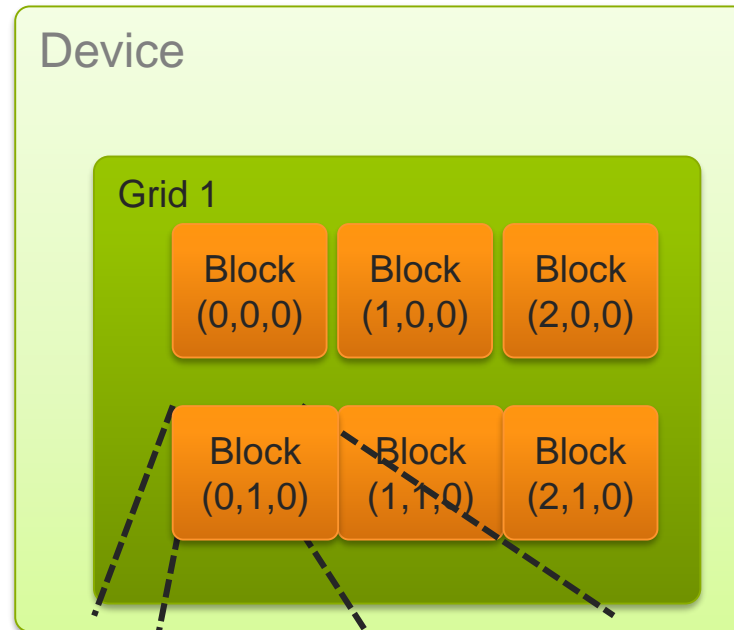
gridDim and blockDim

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

- **A collection of blocks form a grid (1D, 2D or 3D)**
 - Built-in variable `gridDim` specifies the size (or dimension) of the grid.
 - Each copy of the kernel can determine which block it is executing with the built-in variable `blockIdx`
- **Threads in a block are arranged in 1D, 2D, or 3D arrays**
 - Built-in variable `blockDim` specifies the size (or dimensions) of block.
 - `threadIdx` index (or 2D/3D indices) thread within a block
 - `maxThreadsPerBlock`: The limit is 512 threads per block

IDs and Dimensions

- A kernel is launched as a grid of blocks of threads
 - `blockIdx` and `threadIdx` are 3D
 - We showed only one dimension (x)



- Built-in variables:

- `threadIdx`
- `blockIdx`
- `blockDim`
- `gridDim`

Block (1,1,0)

Thread ad (0,0, 0)	Thread ad (1,0, 0)	Thread ad (2,0, 0)	Thread ad (3,0, 0)	Thread ad (4,0, 0)
Thread ad (0,1, 0)	Thread ad (1,1, 0)	Thread ad (2,1, 0)	Thread ad (3,1, 0)	Thread ad (4,1, 0)
Thread ad (0,2, 0)	Thread ad (1,2, 0)	Thread ad (2,2, 0)	Thread ad (3,2, 0)	Thread ad (4,2, 0)

Kernel



Grid



Block



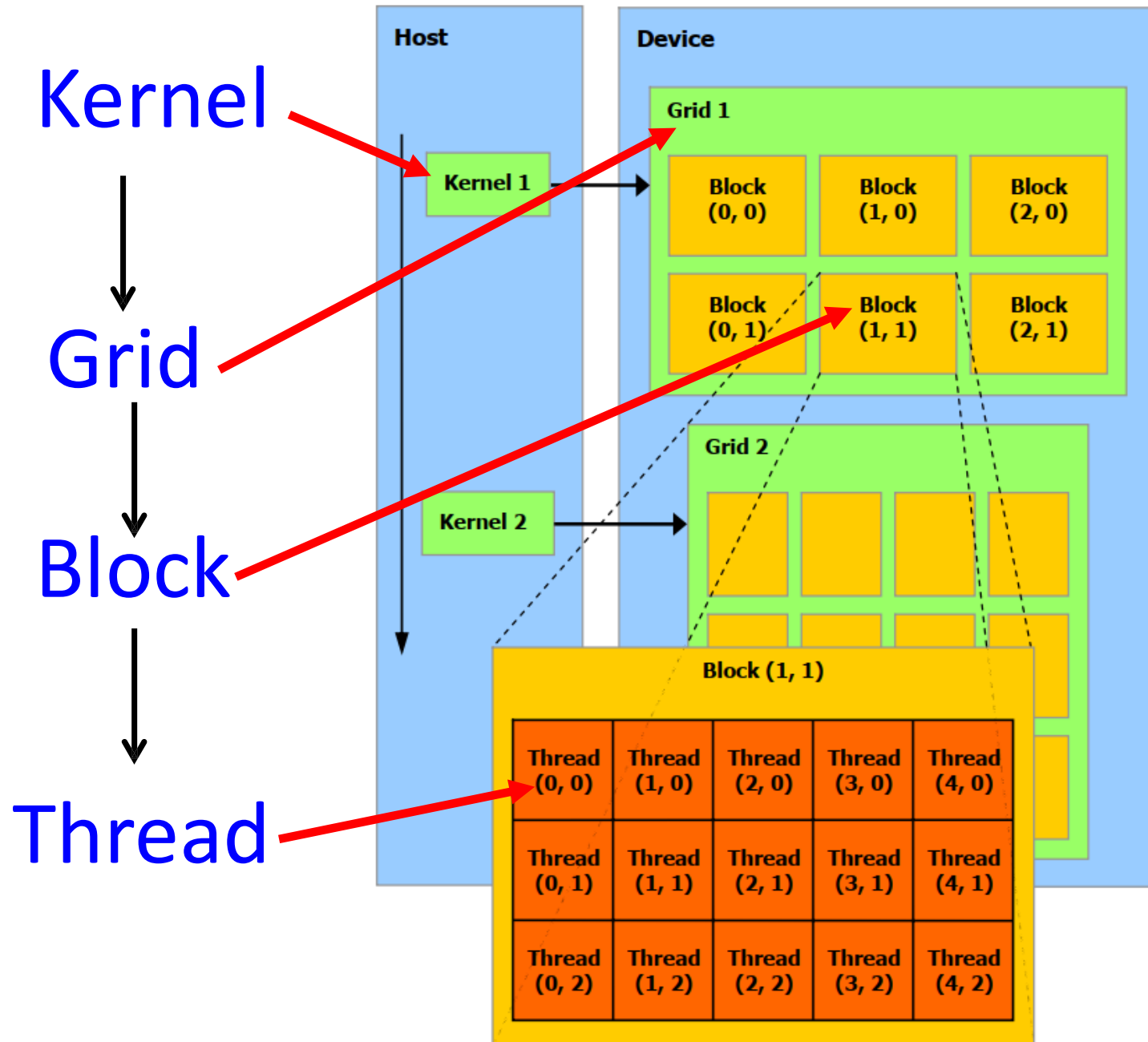
Thread

- Launched by the **host**
- Very similar to a C function
- To be executed on **device**
- All threads will execute that same code in the kernel.

- 1D or 2D (or 3D) organization of a block
- **blockDim.x** and **blockDim.y**
- **gridDim.x** and **gridDim.y**

- 1D, 2D, or 3D organization of a block
- Block is assigned to an SM
- **blockIdx.x**, **blockIdx.y**, and **blockIdx.z**

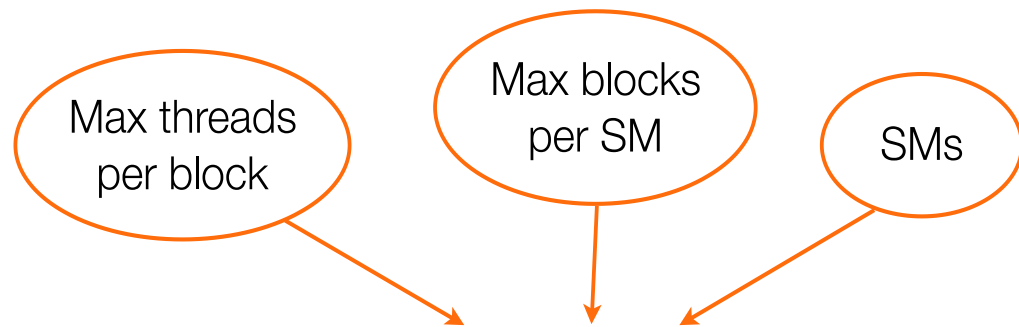
- **threadIdx.x**, **threadIdx.y**, and **threadIdx.z**



Warps

- Inside the SM, threads are launched in groups of 32 called **warps**
 - Warps share the control part (**warp scheduler**)
 - At any time, only **one warp is executed per SM**
 - Threads in a **warp** will be executing the **same instruction**

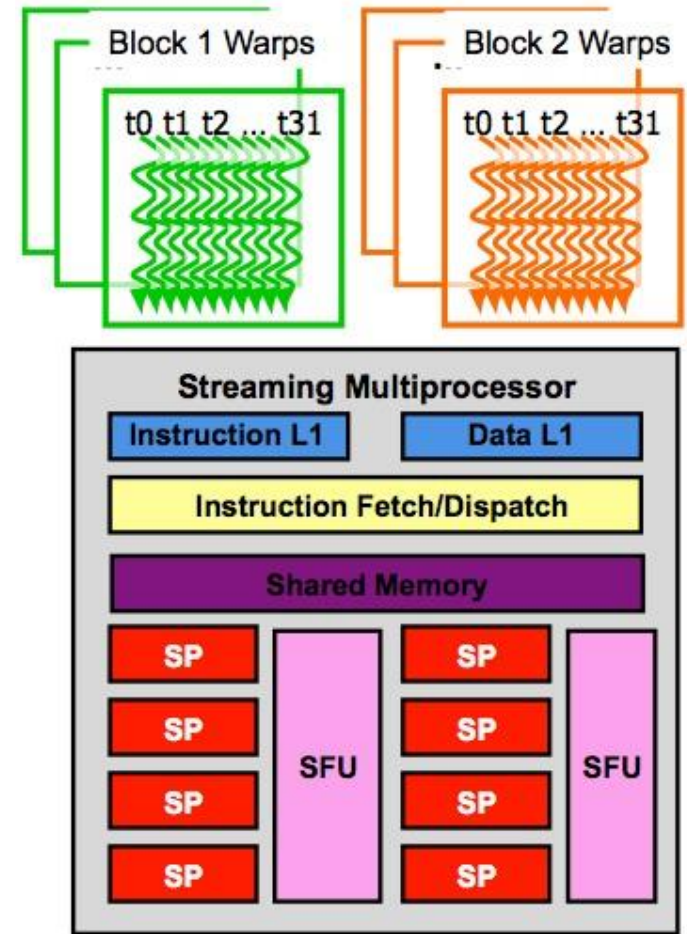
- Fermi:



► Maximum number of active threads $1024 * 8 * 32 = 262144$

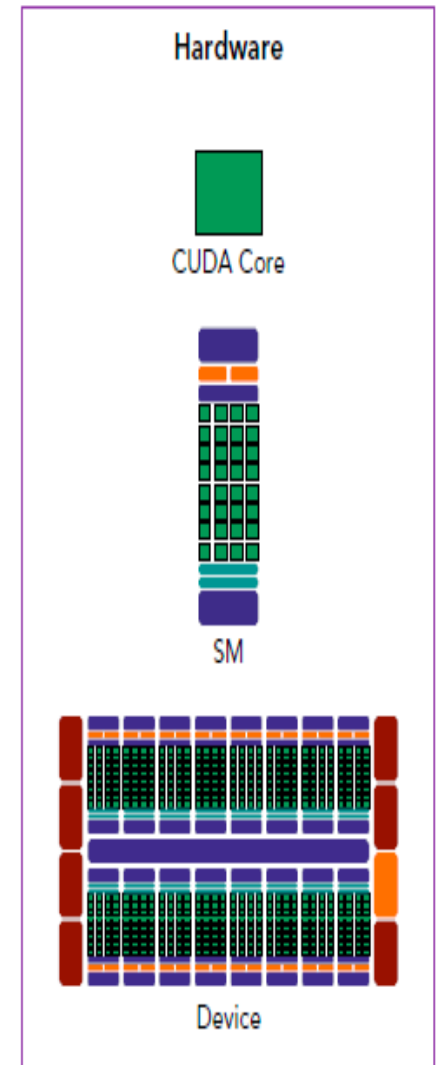
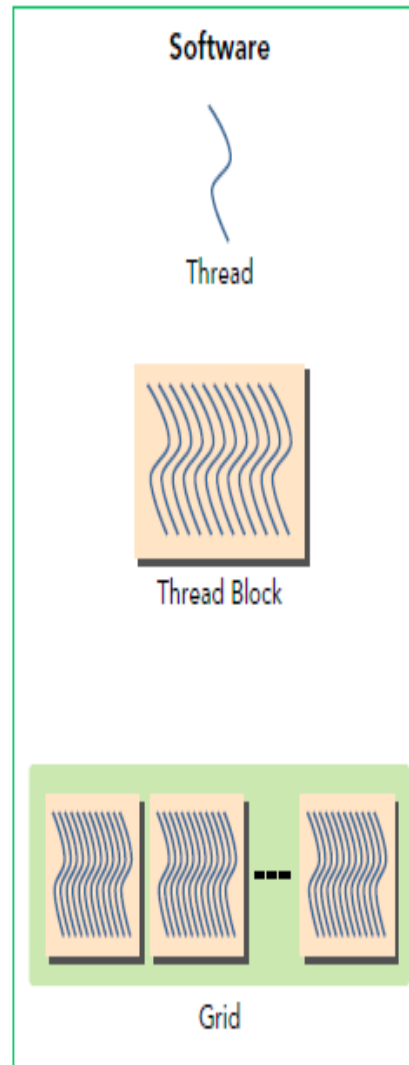
Warp designation

- Hardware separates threads of a block into warps
 - All threads in a warp correspond to the same thread block
 - Threads are placed in a warp sequentially
 - Threads are scheduled in warps

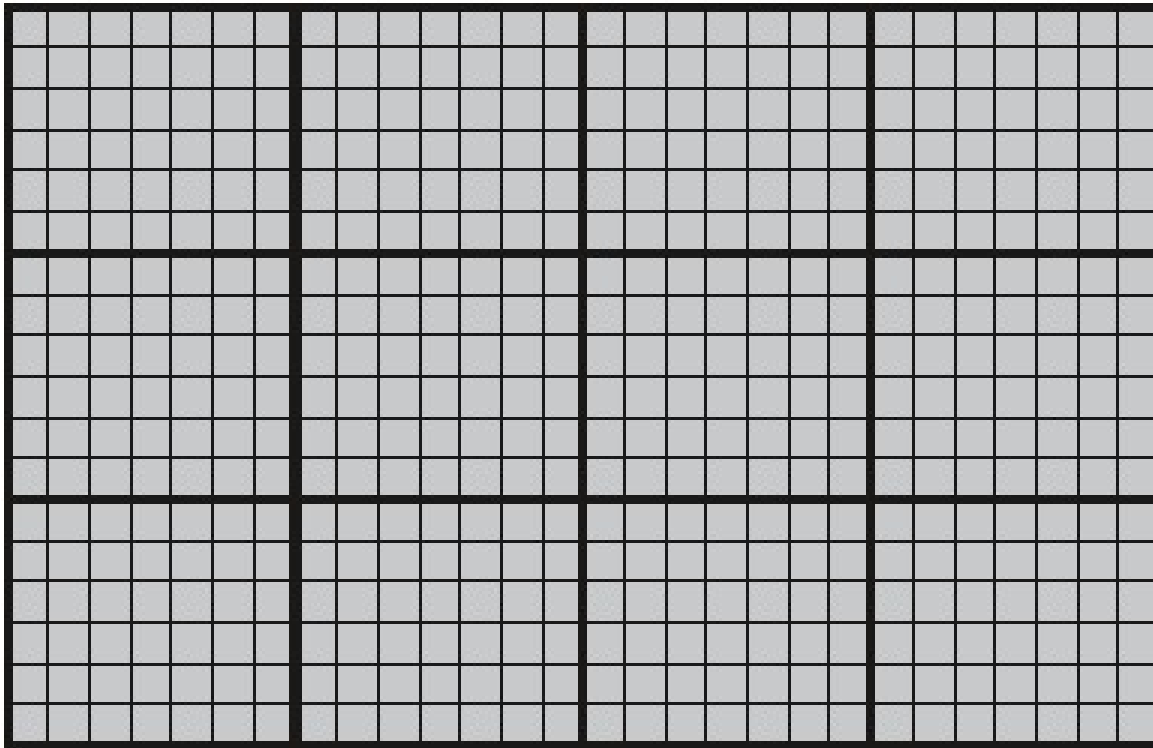


Mappings

- **Grids** map to GPUs
- **Blocks** map to the MultiProcessors (SM)
- **Threads** map to Stream Processors (SP)
- **Warps** are groups of (32) threads that execute simultaneously



Connecting the hardware and the software

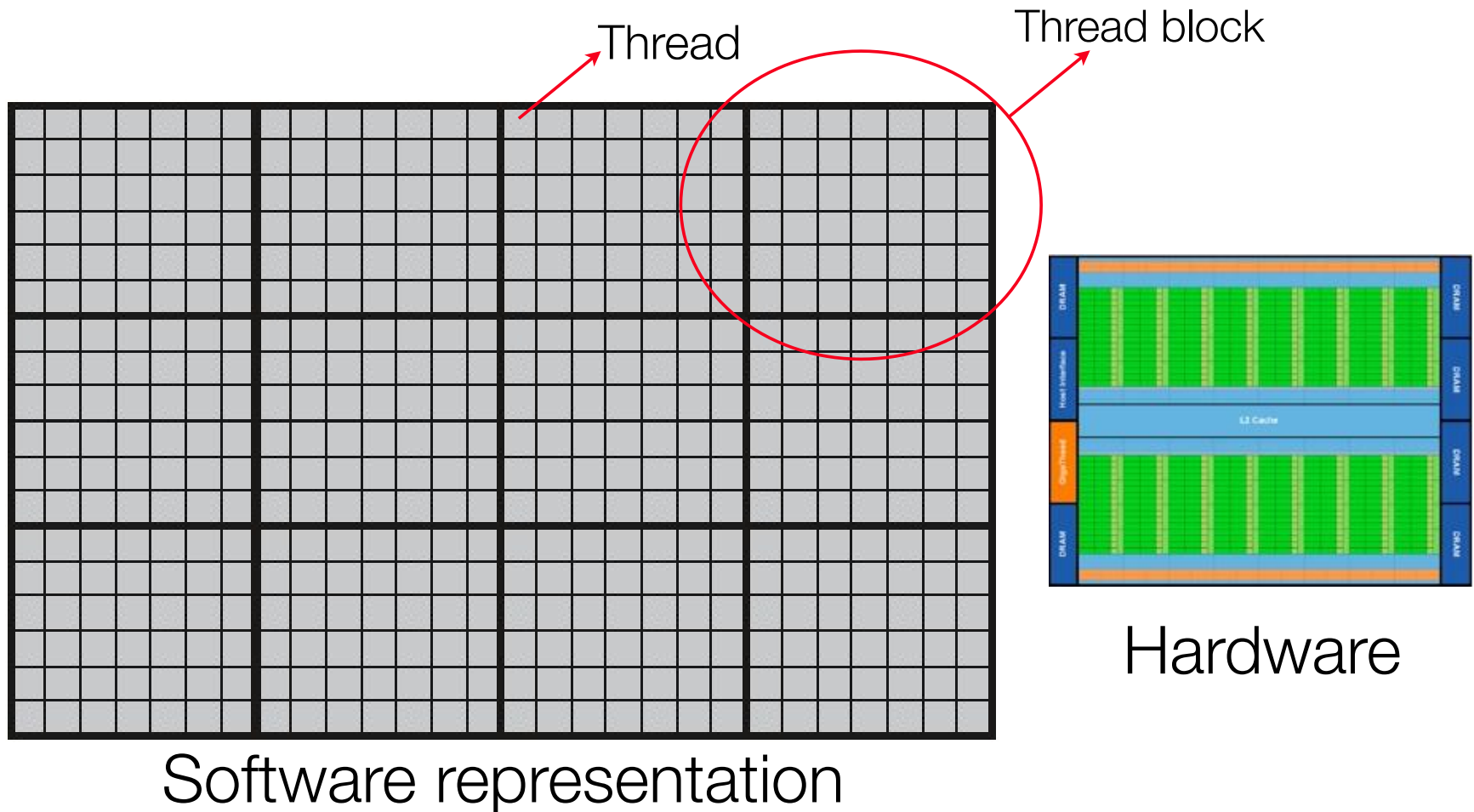


Software representation

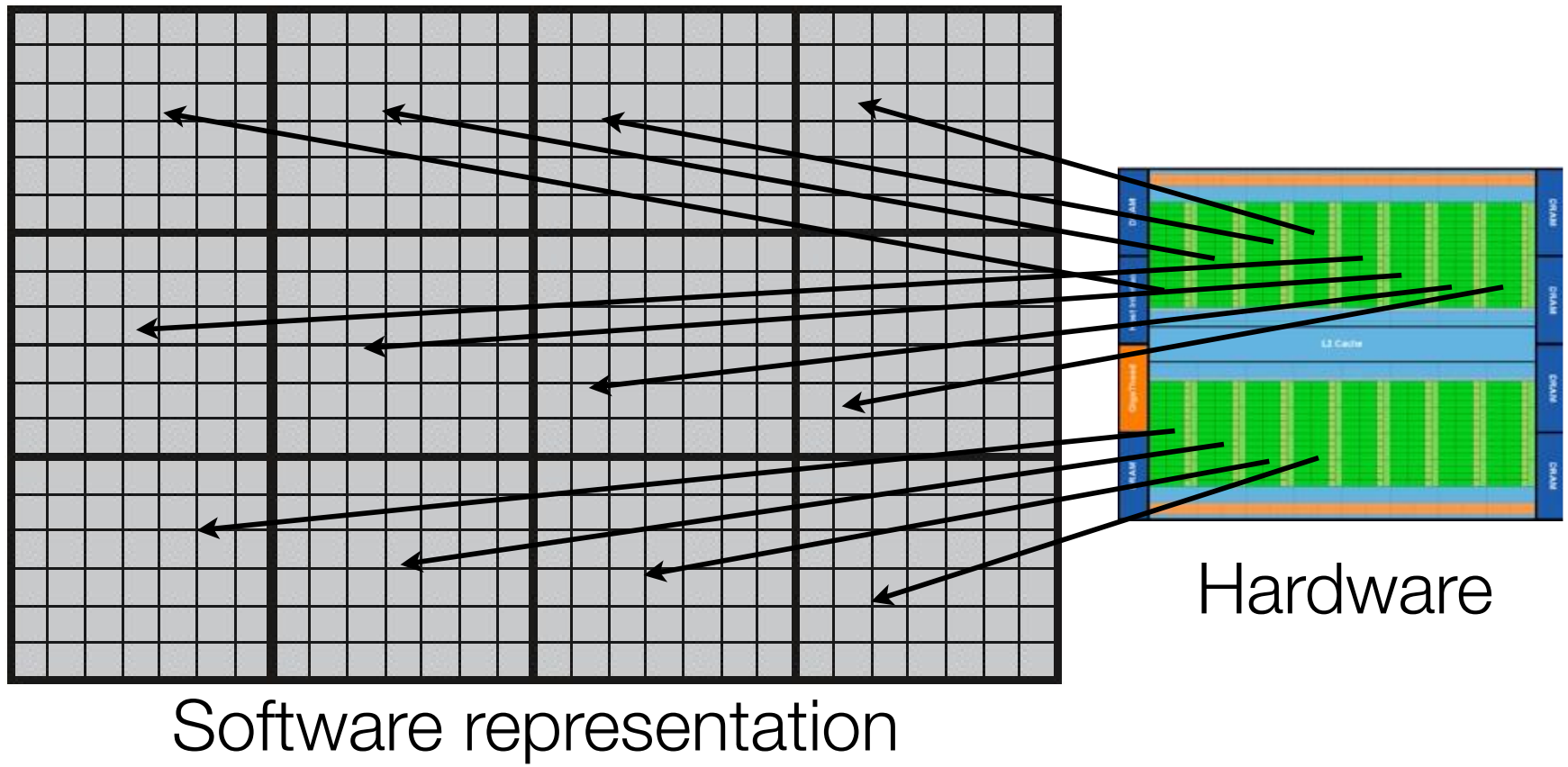


Hardware

Connecting the hardware and the software



Connecting the hardware and the software

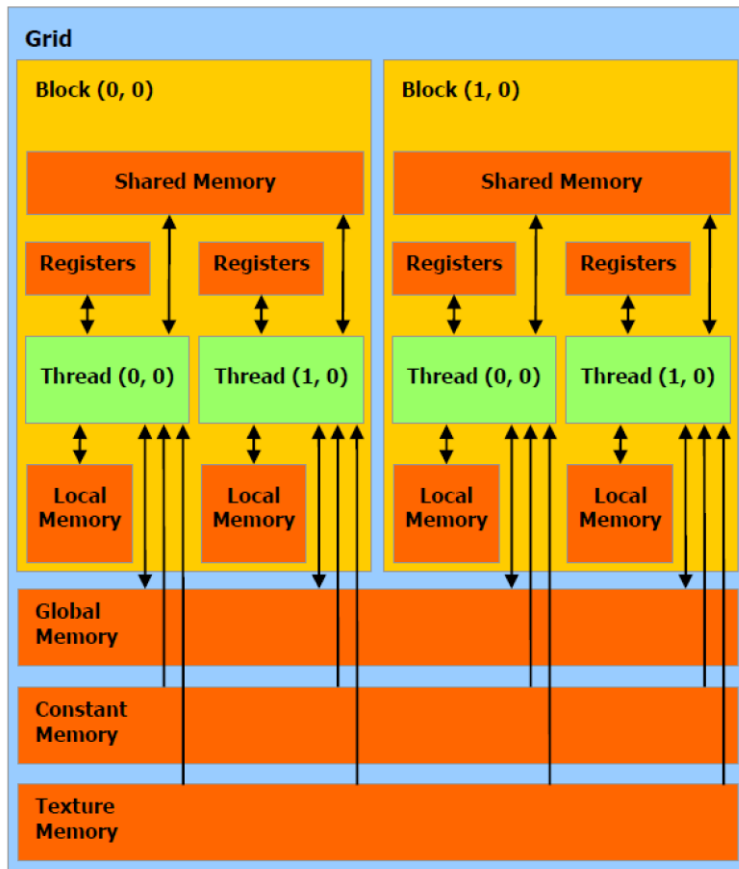


How a Streaming Multiprocessor works

- All threads of the same thread block are executed in the same SM at the same time
- SMs have shared memory, then threads within a thread block can communicate
- The entirety of the threads of a thread block must be executed before there is space to schedule another thread block

More on CUDA Memory Model

- A thread has access to the device's DRAM and on-chip memory through a set of memory spaces



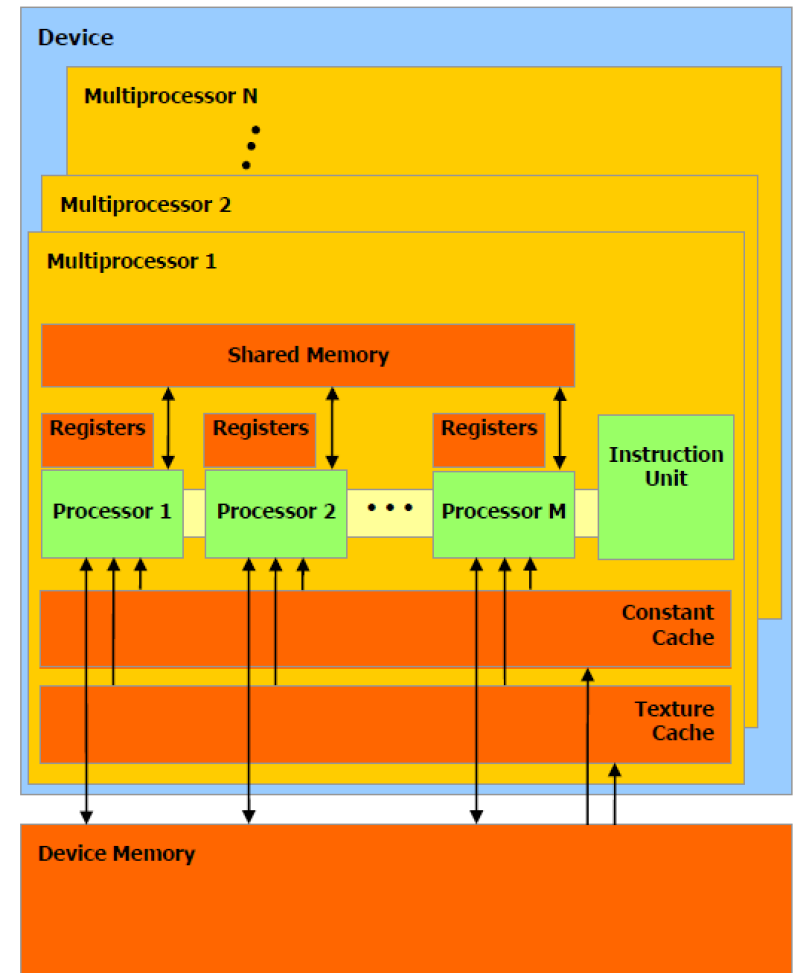
- Read-write per-thread registers,
- Read-write per-thread local memory,
- Read-write per-block shared memory,
- Read-write per-grid global memory,
- Read-only per-grid constant memory,
- Read-only per-grid texture memory.

Global, constant, and texture memory spaces can be read from (or written to) **by the host** and are **persistent** across kernel launches by the same application

More on CUDA Memory Model

Each multiprocessor has **on-chip memory** of four types:

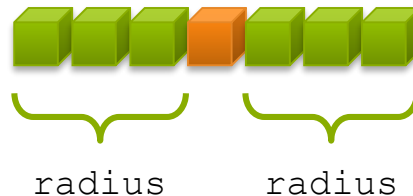
- One set of local 32-bit **registers per processor**,
- **shared memory** that is shared by all the processors
- A read-only **constant cache** that is shared by all the processors
- A read-only **texture cache** that is shared by all the processors



Cooperating Threads

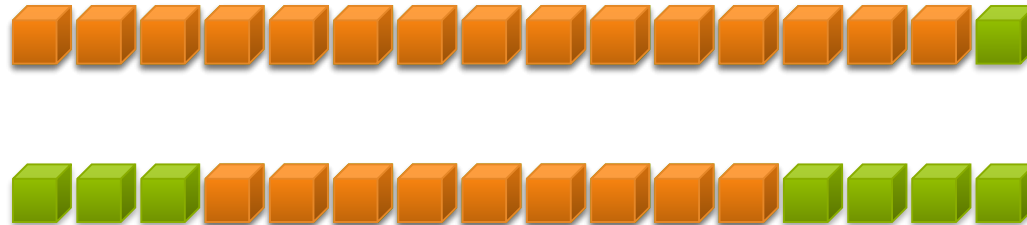
1D Stencil (一维数组环)

- Consider applying a 1D stencil to a 1D array of elements
- Each **output** element is the **sum of input elements within a radius**
- If radius is 3, then each output element is the sum of 7 input elements:



Implementing Within a Block

- Each **thread** processes one **output element**
 - `blockDim.x` elements per block
- Input elements are **read several times**
 - With radius 3, each input element is read seven times

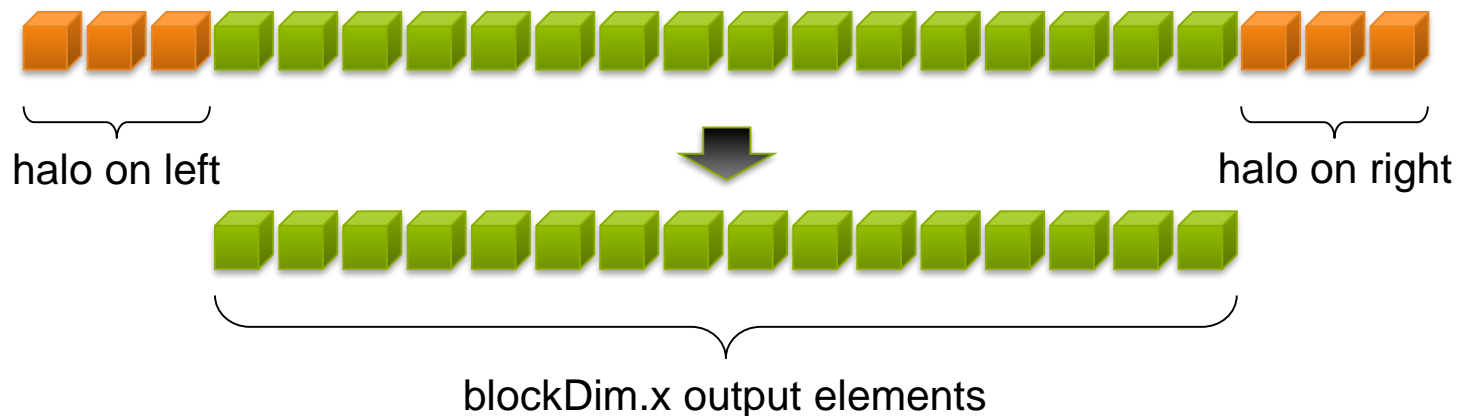


Sharing Data Between Threads

- Within a block, threads share data via **shared memory**
- Extremely fast on-chip memory, user-managed
- Declare using **__shared__**, allocated per block
- Data is not visible to threads in other blocks

Implementing With Shared Memory

- Cache data in shared memory
 - Read ($\text{blockDim.x} + 2 * \text{radius}$) input elements from global memory to shared memory
 - Compute blockDim.x output elements
 - Write blockDim.x output elements to global memory
- Each block needs a **halo** of radius elements at each boundary



Stencil Kernel

```
__global__ void stencil_1d(int *in, int *out) {  
    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];  
    int gindex = threadIdx.x + blockIdx.x * blockDim.x;  
    int lindex = threadIdx.x + RADIUS;  
  
    // Read input elements into shared memory  
    temp[lindex] = in[gindex];  
    if (threadIdx.x < RADIUS) {  
        temp[lindex - RADIUS] = in[gindex - RADIUS];  
        temp[lindex + BLOCK_SIZE] =  
            in[gindex + BLOCK_SIZE];  
    }  
}
```



Stencil Kernel

```
// Apply the stencil
```

```
int result = 0;
```

```
for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
```

```
    result += temp[lindex + offset];
```



```
// Store the result
```

```
out[gindex] = result;
```

```
}
```

Data Race!

- The stencil example **will not work**...
- Suppose thread 15 reads the halo before thread 0 has fetched it...

```
temp[lindex] = in[gindex];           Store at temp[18]   
if (threadIdx.x < RADIUS) {  
    temp[lindex - RADIUS] = in[gindex - RADIUS];    Skipped, threadIdx > RADIUS  
    temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];  
}  
  
int result = 0;  
result += temp[lindex + 1];          Load from temp[19] 
```


__syncthreads()

- **void __syncthreads () ;**
- Synchronizes all threads within a block
 - Used to prevent RAW / WAR / WAW hazards
- All threads must reach the barrier
 - In conditional code, the condition must be uniform across the block

Stencil Kernel

```
__global__ void stencil_1d(int *in, int *out) {
    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
    int gindex = threadIdx.x + blockIdx.x * blockDim.x;
    int lindex = threadIdx.x + radius;

    // Read input elements into shared memory
    temp[lindex] = in[gindex];
    if (threadIdx.x < RADIUS) {
        temp[lindex - RADIUS] = in[gindex - RADIUS];
        temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
    }

    // Synchronize (ensure all the data is available)
    __syncthreads();
}
```

Stencil Kernel

```
// Apply the stencil  
int result = 0;  
for (int offset = -RADIUS ; offset <= RADIUS ; offset++)  
    result += temp[lindex + offset];  
  
// Store the result  
out[gindex] = result;  
}
```

Review (1 of 2)

- Launching parallel threads
 - Launch N blocks with M threads per block with `kernel<<<N,M>>> (...)` ;
 - Use `blockIdx.x` to access block index within grid
 - Use `threadIdx.x` to access thread index within block
- Allocate elements to threads:

```
int index = threadIdx.x + blockIdx.x * blockDim.x;
```

Review (2 of 2)

- Use `__shared__` to declare a variable/array in shared memory
 - Data is shared between threads in a block
 - Not visible to threads in other blocks
- Use `__syncthreads ()` as a barrier
 - Use to prevent data hazards

Managing the Device

Coordinating Host & Device

- Kernel launches are **asynchronous**
 - Control returns to the CPU immediately
- CPU needs to synchronize before consuming the results

cudaMemcpy()	Blocks the CPU until the copy is complete Copy begins when all preceding CUDA calls have completed
cudaMemcpyAsync()	Asynchronous, does not block the CPU
cudaDeviceSynchronize()	Blocks the CPU until all preceding CUDA calls have completed

Reporting Errors

- All CUDA API calls return an error code (`cudaError_t`)
 - Error in the API call itself
 - OR
 - Error in an earlier asynchronous operation (e.g. kernel)

- Get the error code for the last error:

```
cudaError_t cudaGetLastError(void)
```

- Get a string to describe the error:

```
char *cudaGetErrorString(cudaError_t)
```

```
printf("%s\n", cudaGetErrorString(cudaGetLastError()));
```


Device Management

- Application can query and select GPUs

`cudaGetDeviceCount(int *count)`

`cudaSetDevice(int device)`

`cudaGetDevice(int *device)`

`cudaGetDeviceProperties(cudaDeviceProp *prop, int device)`

- Multiple threads can share a device
- A single thread can manage multiple devices

`cudaSetDevice(i)` to select current device

`cudaMemcpy(...)` for peer-to-peer copies[†]

[†] requires OS and device support

Compute Capability

- The **compute capability** of a device describes its architecture, e.g.
 - Number of registers
 - Sizes of memories
 - Features & capabilities

Compute Capability	Selected Features (see CUDA C Programming Guide for complete list)	Tesla models
1.0	Fundamental CUDA support	870
1.3	Double precision, improved memory accesses, atomics	10-series
2.0	Caches, fused multiply-add, 3D grids, surfaces, ECC, P2P, concurrent kernels/copies, function pointers, recursion	20-series

Conclusions

- **Data parallelism** is the main source of scalability for parallel programs
- Each CUDA source file can have a **mixture** of both host and device code.
- What we learned about CUDA:
 - `KernelA<<< nBlk, nTid >>>(args)`
 - `cudaMalloc()`
 - `cudaFree()`
 - `cudaMemcpy()`
 - `gridDim` and `blockDim`
 - `threadIdx.x` and `threadIdx.y`