CPSC 4240/5240: Parallel Programming Techniques

Lecture 19: Programming on GPUs

Lecturer: Quanquan C. Liu

Spring 2026

Final Project

- If you'd like for me to comment on your final project ideas:
  - Come to office hours Wednesday at 3pm in AKW 111
  - Post a private message on Ed Discussion
- Fill out the form for your final project sent via Canvas Announcement **by Friday**!

Parallelization on GPUs

Challenges of Parallelization

- On CPUs, **hardware-supported atomic operations** are used to enable concurrency

Challenges of Parallelization

- On CPUs, **hardware-supported atomic operations** are used to enable concurrency
  - Atomic operations allow data to be **read and written without intervention from another thread**

Challenges of Parallelization

- On CPUs, **hardware-supported atomic operations** are used to enable concurrency
  - Atomic operations allow data to be **read and written without intervention from another thread**
- Some GPUs support system-wide atomic operations, but with a large performance trade-off

Challenges of Parallelization

- On CPUs, **hardware-supported atomic operations** are used to enable concurrency
  - Atomic operations allow data to be **read and written without intervention from another thread**
- Some GPUs support system-wide atomic operations, but with a large performance trade-off
  - Usually code that requires **global synchronization** is not well suited for GPUs

Challenges of Parallelization

- On CPUs, **hardware-supported atomic operations** are used to enable concurrency
  - Atomic operations allow data to be **read and written without intervention from another thread**
- Some GPUs support system-wide atomic operations, but with a large performance trade-off
  - Usually code that requires **global synchronization** is not well suited for GPUs
  - Any problem that is decomposed using **input data partitioning** (i.e., requires results to be combined at the end) will likely need to be restructured to execute well on a GPU

General-Purpose Computing on GPUs (GPGPU)

Concept of GPGPU (General-Purpose Computing on GPUs)

- Idea:
  - Potential for very high performance at low cost

Concept of GPGPU (General-Purpose Computing on GPUs)

- Idea:
  - Potential for very high performance at low cost
  - Architecture well suited for certain kinds of parallel applications ***(data parallel)***

Concept of GPGPU (General-Purpose Computing on GPUs)

- Idea:
  - Potential for very high performance at low cost
  - Architecture well suited for certain kinds of parallel applications ***(data parallel)***
  - Demonstrations of ***30-100x*** speedup over CPU

Concept of GPGPU (General-Purpose Computing on GPUs)

- Idea:
  - Potential for very high performance at low cost
  - Architecture well suited for certain kinds of parallel applications ***(data parallel)***
  - Demonstrations of ***30-100x*** speedup over CPU
- Early challenges:
  - Architectures very customized to graphics problems (e.g., vertex and fragment processors)

Concept of GPGPU (General-Purpose Computing on GPUs)

- Idea:
  - Potential for very high performance at low cost
  - Architecture well suited for certain kinds of parallel applications ***(data parallel)***
  - Demonstrations of ***30-100x*** speedup over CPU
- Early challenges:
  - Architectures very customized to graphics problems (e.g., vertex and fragment processors)
  - Programmed using graphics-specific programming models or libraries

Concept of GPGPU (General-Purpose Computing on GPUs)

- Idea:
  - Potential for very high performance at low cost
  - Architecture well suited for certain kinds of parallel applications ***(data parallel)***
  - Demonstrations of ***30-100x*** speedup over CPU
- Early challenges:
  - Architectures very customized to graphics problems (e.g., vertex and fragment processors)
  - Programmed using graphics-specific programming models or libraries
- Recent trends:
  - Some convergence between commodity and GPUs and their associated parallel programming models

Memory Wall

- **Memory wall** refers to disparity between how quickly processors (CPUs or GPUs) can perform computations versus how quickly data can be moved to and from main memory

Memory Wall

- **Memory wall** refers to disparity between how quickly processors (CPUs or GPUs) can perform computations versus how quickly data can be moved to and from main memory
- As processors get faster, ability to process data **outpaces ability of memory systems** to supply it

Memory Wall

- **Memory wall** refers to disparity between how quickly processors (CPUs or GPUs) can perform computations versus how quickly data can be moved to and from main memory
- As processors get faster, ability to process data **outpaces ability of memory systems** to supply it
- Leaves the CPU/GPU idle or underutilized while it waits for data to arrive from memory

Memory Wall

- **Memory wall** refers to disparity between how quickly processors (CPUs or GPUs) can perform computations versus how quickly data can be moved to and from main memory
- As processors get faster, ability to process data **outpaces ability of memory systems** to supply it
- Leaves the CPU/GPU idle or underutilized while it waits for data to arrive from memory
- Insufficient memory bandwidth or high memory latency can negate the benefits of massive parallelism in GPUs

Memory Wall

- **Memory wall** refers to disparity between how quickly processors (CPUs or GPUs) can perform computations versus how quickly data can be moved to and from main memory
- As processors get faster, ability to process data **outpaces ability of memory systems** to supply it
- Leaves the CPU/GPU idle or underutilized while it waits for data to arrive from memory
- Insufficient memory bandwidth or high memory latency can negate the benefits of massive parallelism in GPUs
- GPU-based computing (e.g., CUDA) often needs **careful attention to memory optimizations**

Memory Wall

- **Memory wall** refers to disparity between how quickly processors (CPUs or GPUs) can perform computations versus how quickly data can be moved to and from main memory
- As processors get faster, ability to process data **outpaces ability of memory systems** to supply it
- Leaves the CPU/GPU idle or underutilized while it waits for data to arrive from memory
- Insufficient memory bandwidth or high memory latency can negate the benefits of massive parallelism in GPUs
- GPU-based computing (e.g., CUDA) often needs **careful attention to memory optimizations**
  - **Efficient data layouts, caching, shared memory usage**

Overcoming the Memory Wall

- **Cache Hierarchies**: Multiple levels of caches to keep data closer to the core and reduce round trips to main memory

Overcoming the Memory Wall

- **Cache Hierarchies**: Multiple levels of caches to keep data closer to the core and reduce round trips to main memory
- **On-Chip Memory/Shared Memory**: GPUs often provide small, fast on-chip memories that threads can share

Overcoming the Memory Wall

- **Cache Hierarchies**: Multiple levels of caches to keep data closer to the core and reduce round trips to main memory
- **On-Chip Memory/Shared Memory**: GPUs often provide small, fast on-chip memories that threads can share
- **Prefetching and Data Locality**: Software and hardware techniques try to fetch data into caches before it's needed, taking advantage of predictable access patterns

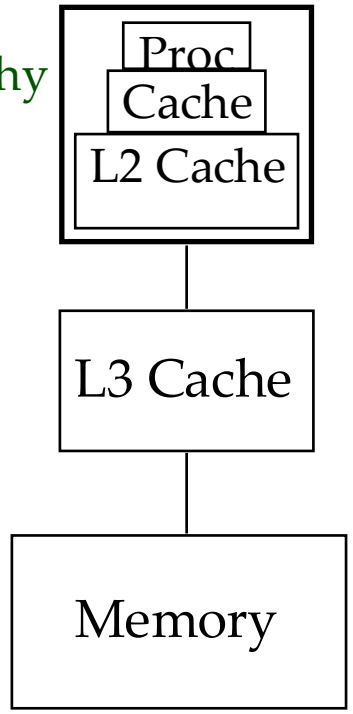
Overcoming the Memory Wall

- **Cache Hierarchies**: Multiple levels of caches to keep data closer to the core and reduce round trips to main memory
- **On-Chip Memory/Shared Memory**: GPUs often provide small, fast on-chip memories that threads can share
- **Prefetching and Data Locality**: Software and hardware techniques try to fetch data into caches before it's needed, taking advantage of predictable access patterns
- **High-Bandwidth Memory (HBM):** New memory technologies aim to increase memory bandwidth to better keep pace with processor speeds

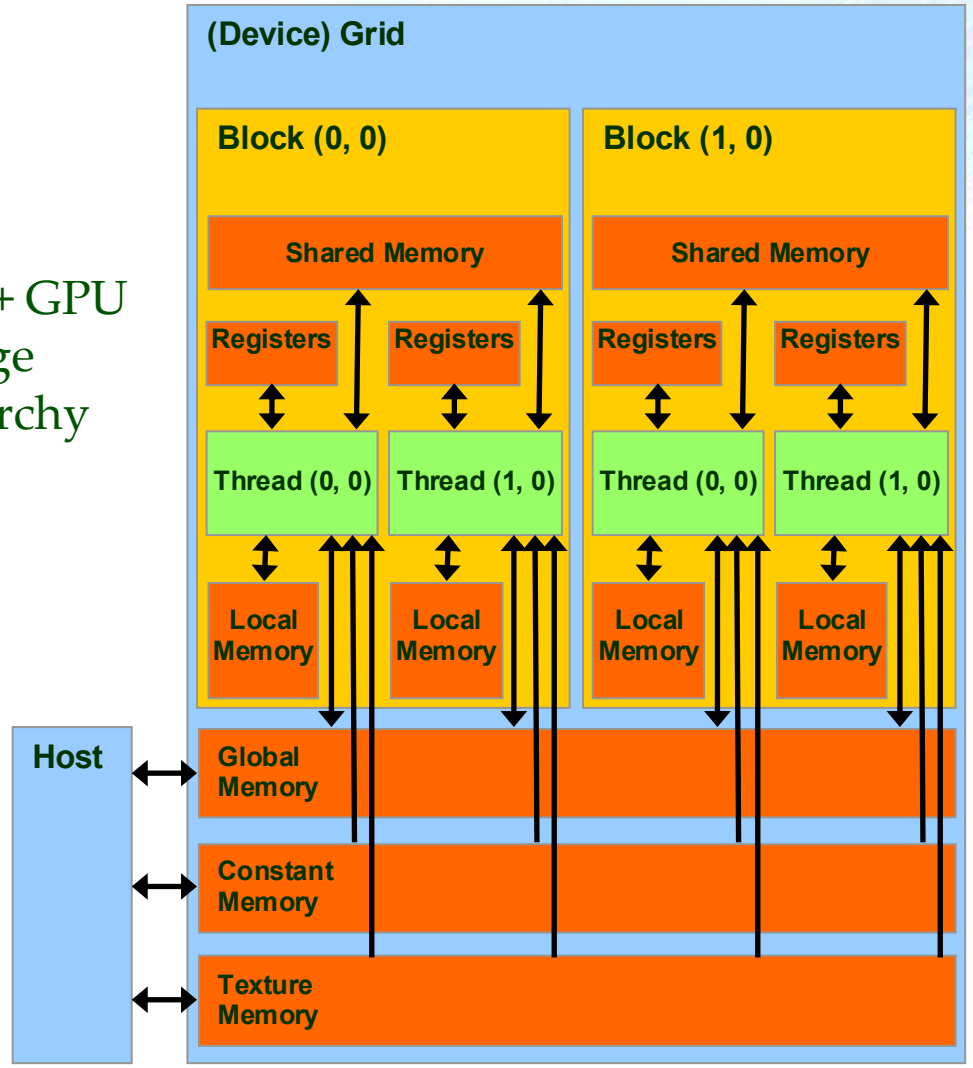
Locality and Parallelism

- Large memories are slow, fast memories are small
- Storage hierarchies are large and fast on average
- Algorithm should do most work on nearby data

Conventional Storage Hierarchy



Host + GPU Storage Hierarchy



Courtesy NVIDIA

CUDA: Programming Interface (Nvidia)

CUDA (Compute Unified Device Architecture)

- ***Data-parallel*** programming interface to GPU

CUDA (Compute Unified Device Architecture)

- ***Data-parallel*** programming interface to GPU
  - Data to be operated on is discretized into independent partition of memory

CUDA (Compute Unified Device Architecture)

- ***Data-parallel*** programming interface to GPU
  - Data to be operated on is discretized into independent partition of memory
  - Each thread performs roughly same computation to different partition of data

CUDA (Compute Unified Device Architecture)

- ***Data-parallel*** programming interface to GPU
  - Data to be operated on is discretized into independent partition of memory
  - Each thread performs roughly same computation to different partition of data
  - When appropriate, easy to express and very efficient parallelization

CUDA (Compute Unified Device Architecture)

- ***Data-parallel*** programming interface to GPU
  - Data to be operated on is discretized into independent partition of memory
  - Each thread performs roughly same computation to different partition of data
  - When appropriate, easy to express and very efficient parallelization
- Programmer expresses

CUDA (Compute Unified Device Architecture)

- ***Data-parallel*** programming interface to GPU
  - Data to be operated on is discretized into independent partition of memory
  - Each thread performs roughly same computation to different partition of data
  - When appropriate, easy to express and very efficient parallelization
- Programmer expresses
  - Thread programs to be launched on GPU, and how to launch

CUDA (Compute Unified Device Architecture)

- ***Data-parallel*** programming interface to GPU
  - Data to be operated on is discretized into independent partition of memory
  - Each thread performs roughly same computation to different partition of data
  - When appropriate, easy to express and very efficient parallelization
- Programmer expresses
  - Thread programs to be launched on GPU, and how to launch
  - Data organization and movement between CPU and GPU

CUDA (Compute Unified Device Architecture)

- ***Data-parallel*** programming interface to GPU
  - Data to be operated on is discretized into independent partition of memory
  - Each thread performs roughly same computation to different partition of data
  - When appropriate, easy to express and very efficient parallelization
- Programmer expresses
  - Thread programs to be launched on GPU, and how to launch
  - Data organization and movement between CPU and GPU
  - Synchronization, memory management, testing, …

CUDA (Compute Unified Device Architecture)

- ***Data-parallel*** programming interface to GPU
  - Data to be operated on is discretized into independent partition of memory
  - Each thread performs roughly same computation to different partition of data
  - When appropriate, easy to express and very efficient parallelization
- Programmer expresses
  - Thread programs to be launched on GPU, and how to launch
  - Data organization and movement between CPU and GPU
  - Synchronization, memory management, testing, …
- CUDA is one of first to support ***heterogeneous*** architectures
  - Can work on CPUs and GPUs

CUDA (Compute Unified Device Architecture)

- ***Data-parallel*** programming interface to GPU
  - Data to be operated on is discretized into independent partition of memory
  - Each thread performs roughly same computation to different partition of data
  - When appropriate, easy to express and very efficient parallelization
- Programmer expresses
  - Thread programs to be launched on GPU, and how to launch
  - Data organization and movement between CPU and GPU
  - Synchronization, memory management, testing, …
- CUDA is one of first to support ***heterogeneous*** architectures
  - Can work on CPUs and GPUs
- CUDA environment
  - Compiler, run-time utilities, libraries, emulation, performance

CUDA Programming Model: A Highly Multithreaded Coprocessor

- The GPU is viewed as a compute device that:
  - Is a coprocessor to the CPU or host
  - Has its own DRAM (device memory)
  - Runs many threads in parallel

CUDA Programming Model: A Highly Multithreaded Coprocessor

- The GPU is viewed as a compute device that:
  - Is a coprocessor to the CPU or host
  - Has its own DRAM (device memory)
  - Runs many threads in parallel
- Data-parallel portions of an application are executed on the device as kernels which run in parallel on many threads

CUDA Programming Model: A Highly Multithreaded Coprocessor

- The GPU is viewed as a compute device that:
  - Is a coprocessor to the CPU or host
  - Has its own DRAM (device memory)
  - Runs many threads in parallel
- Data-parallel portions of an application are executed on the device as kernels which run in parallel on many threads
- Differences between GPU and CPU threads
  - GPU threads are extremely lightweight
    - Very little creation overhead

CUDA Programming Model: A Highly Multithreaded Coprocessor

- The GPU is viewed as a compute device that:
  - Is a coprocessor to the CPU or host
  - Has its own DRAM (device memory)
  - Runs many threads in parallel
- Data-parallel portions of an application are executed on the device as kernels which run in parallel on many threads
- Differences between GPU and CPU threads
  - GPU threads are extremely lightweight
    - Very little creation overhead
  - GPU needs 1000s of threads for full efficiency
    - Multi-core CPU needs only a few

Computation Partitioning in CUDA

- In **heterogeneous systems** using CUDA, you **partition computation** between the **host** (CPU) and the **device** (GPU)

Computation Partitioning in CUDA

- In **heterogeneous systems** using CUDA, you **partition computation** between the **host** (CPU) and the **device** (GPU)
- **host**
  - **Where It Runs:** On the CPU.

Computation Partitioning in CUDA

- In **heterogeneous systems** using CUDA, you **partition computation** between the **host** (CPU) and the **device** (GPU)
- **host**
  - **Where It Runs:** On the CPU.
  - **Usage:** A function marked with __host__ executes on the host. (This is the default if no annotation is given)

Computation Partitioning in CUDA

- In **heterogeneous systems** using CUDA, you **partition computation** between the **host** (CPU) and the **device** (GPU)
- **host**
  - **Where It Runs:** On the CPU.
  - **Usage:** A function marked with __host__ executes on the host. (This is the default if no annotation is given)
  - **When to Use:** For code that is not performance-critical on the GPU or for handling operations that are inherently sequential or require system-level control (like I/O operations).

Computation Partitioning in CUDA

- In **heterogeneous systems** using CUDA, you **partition computation** between the **host** (CPU) and the **device** (GPU)
- **device**

Computation Partitioning in CUDA

- In **heterogeneous systems** using CUDA, you **partition computation** between the **host** (CPU) and the **device** (GPU)
- **device**
  - **Where It Runs:** On the GPU.

Computation Partitioning in CUDA

- In **heterogeneous systems** using CUDA, you **partition computation** between the **host** (CPU) and the **device** (GPU)
- **device**
  - **Where It Runs:** On the GPU.
  - **Usage:** A function declared as __device__ is executed on the GPU.

Computation Partitioning in CUDA

- In **heterogeneous systems** using CUDA, you **partition computation** between the **host** (CPU) and the **device** (GPU)
- **device**
  - **Where It Runs:** On the GPU.
  - **Usage:** A function declared as __device__ is executed on the GPU.
  - **Restrictions:**
    - It **cannot** be called directly from host code.

Computation Partitioning in CUDA

- In **heterogeneous systems** using CUDA, you **partition computation** between the **host** (CPU) and the **device** (GPU)
- **device**
  - **Where It Runs:** On the GPU.
  - **Usage:** A function declared as __device__ is executed on the GPU.
  - **Restrictions:**
    - It **cannot** be called directly from host code.
    - It is used as a helper function for other GPU code.

Computation Partitioning in CUDA

- In **heterogeneous systems** using CUDA, you **partition computation** between the **host** (CPU) and the **device** (GPU)
- **device**
  - **Where It Runs:** On the GPU.
  - **Usage:** A function declared as __device__ is executed on the GPU.
  - **Restrictions:**
    - It **cannot** be called directly from host code.
    - It is used as a helper function for other GPU code.
  - **When to Use:** For operations that are closely tied to the GPU's parallel execution and that you want to call from within kernels (GPU functions).

Computation Partitioning in CUDA

- In **heterogeneous systems** using CUDA, you **partition computation** between the **host** (CPU) and the **device** (GPU)
- **global**
  - **Where It Runs:** On the GPU.

Computation Partitioning in CUDA

- In **heterogeneous systems** using CUDA, you **partition computation** between the **host** (CPU) and the **device** (GPU)
- **global**
  - **Where It Runs:** On the GPU.
  - **Usage:** A __global__ function is a *kernel* that is launched from the host and executed in parallel on the GPU.

Computation Partitioning in CUDA

- In **heterogeneous systems** using CUDA, you **partition computation** between the **host** (CPU) and the **device** (GPU)
- **global**
  - **Where It Runs:** On the GPU.
  - **Usage:** A __global__ function is a *kernel* that is launched from the host and executed in parallel on the GPU.
  - **Requirements:**
    - Must have a void return type.
    - It is invoked using a special kernel launch syntax.

Computation Partitioning in CUDA

- In **heterogeneous systems** using CUDA, you **partition computation** between the **host** (CPU) and the **device** (GPU)
- **global**
  - **Where It Runs:** On the GPU.
  - **Usage:** A __global__ function is a *kernel* that is launched from the host and executed in parallel on the GPU.
  - **Requirements:**
    - Must have a void return type.
    - It is invoked using a special kernel launch syntax.
  - **When to Use:** When you want to execute large-scale parallel operations; the host calls the kernel, which then executes concurrently across many threads on the GPU.

CUDA Code Example

```
#include <iostream>
#include <cuda_runtime.h>

// __host__ function: Runs on the CPU
__host__ void cpuFunction() {
    std::cout << "Running on host (CPU)" << std::endl;
}

// __device__ function: Runs on the GPU; can only be called from GPU code
__device__ int deviceAdd(int a, int b) {
    return a + b;
}

// __global__ kernel: Runs on the GPU, callable from the host
__global__ void kernelFunction(int* d_result, int a, int b) {
    // Each thread can compute its own index if processing array data, for example:
    int threadId = threadIdx.x + blockIdx.x * blockDim.x;
    // For this simple example, we let one thread do the addition
    if (threadId == 0) {
        // Call the __device__ function to perform the addition
        *d_result = deviceAdd(a, b);
    }
}
```

CUDA Code Example

```
#include <iostream>
#include <cuda_runtime.h>

// __host__ function: Runs on the CPU
__host__ void cpuFunction() {
    std::cout << "Running on host (CPU)" << std::endl;
}

// __device__ function: Runs on the GPU; can only be called from GPU code
__device__ int deviceAdd(int a, int b) {
    return a + b;
}

// __global__ kernel: Runs on the GPU, callable from the host
__global__ void kernelFunction(int* d_result, int a, int b) {
    // Each thread can compute its own index if processing array data, for example:
    int threadId = threadIdx.x + blockIdx.x * blockDim.x;
    // For this simple example, we let one thread do the addition
    if (threadId == 0) {
        // Call the __device__ function to perform the addition
        *d_result = deviceAdd(a, b);
```

CUDA Code Example

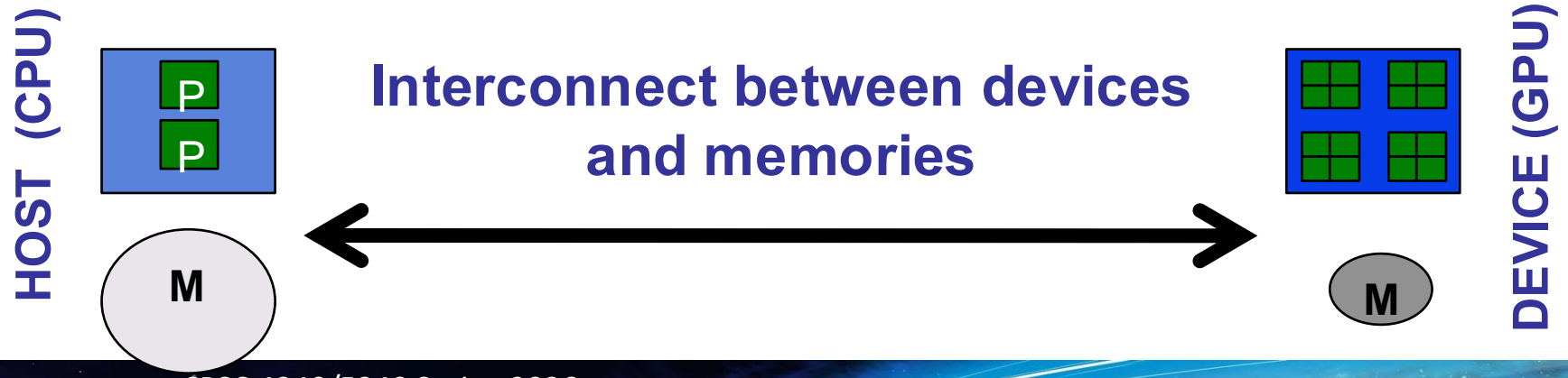
```
#include <iostream>
#include <cuda_runtime.h>

// __host__ function: Runs on the CPU
__host__ void cpuFunction() {
    std::cout << "Running on host (CPU)" << std::endl;
}

// __device__ function: Runs on the GPU; can only be called from GPU code
__device__ int deviceAdd(int a, int b) {
    return a + b;
}

// __global__ kernel: Runs on the GPU, callable from the host
__global__ void kernelFunction(int* d_result, int a, int b) {
    int threadId = threadIdx.x + blockIdx.x * blockDim.x;
    if (threadId == 0) {
        // Call the __device__ function to perform the addition
        *d_result = deviceAdd(a, b);
    }
}
```

```
int main() {
    // Print a message from the CPU.
    cpuFunction();

    // Host-side variable to receive the result.
    int h_result;

    // Declare a pointer for device memory.
    int* d_result;

    // Allocate memory on the GPU (device)
    cudaMalloc(&d_result, sizeof(int));

    // Launch the kernel:
    // Here we use 1 block with 1 thread for simplicity.
    kernelFunction<<<1, 1>>>(d_result, 3, 4);

    // Wait for the GPU to finish its work (ensure kernel execution is complete).
    cudaDeviceSynchronize();

    // Copy the result from device memory to host memory.
    cudaMemcpy(&h_result, d_result, sizeof(int),
cudaMemcpyDeviceToHost);

    // Free the allocated device memory.
    cudaFree(d_result);

    // Display the result.
    std::cout << "Result (3 + 4): " << h_result << std::endl;

    return 0;
}
```

```
int main() {
    // Print a message from the CPU.
    cpuFunction();

    // Host-side variable to receive the result.
    int h_result;

    // Declare a pointer for device memory.
    int* d_result;

    // Allocate memory on the GPU (device)
    cudaMalloc(&d_result, sizeof(int));

    // Launch the kernel:
    // Here we use 1 block with 1 thread for
    kernelFunction<<<1, 1>>>(d_result, 3, 4);

    // Wait for the GPU to finish its work (ensure kernel execution is complete).
    cudaDeviceSynchronize();

    // Copy the result from device memory to host memory.
    cudaMemcpy(&h_result, d_result, sizeof(int),
cudaMemcpyDeviceToHost);

    // Free the allocated device memory.
    cudaFree(d_result);

    // Display the result.
    std::cout << "Result (3 + 4): " << h_result << std::endl;

    return 0;
}
```

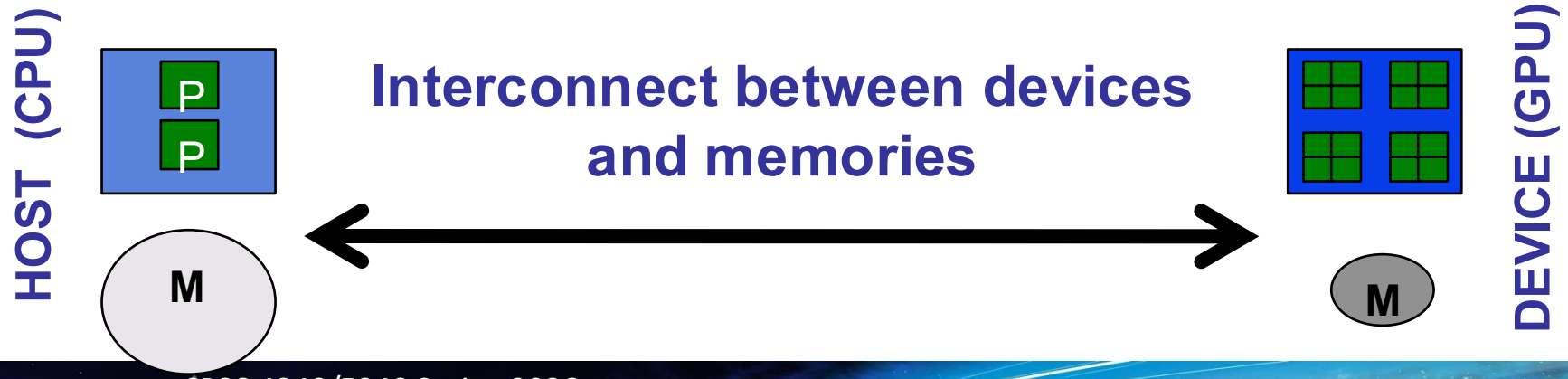
What Programmer Expresses in CUDA

- Where does computation occur?
  - **__host__**: Runs on the CPU (host)



What Programmer Expresses in CUDA

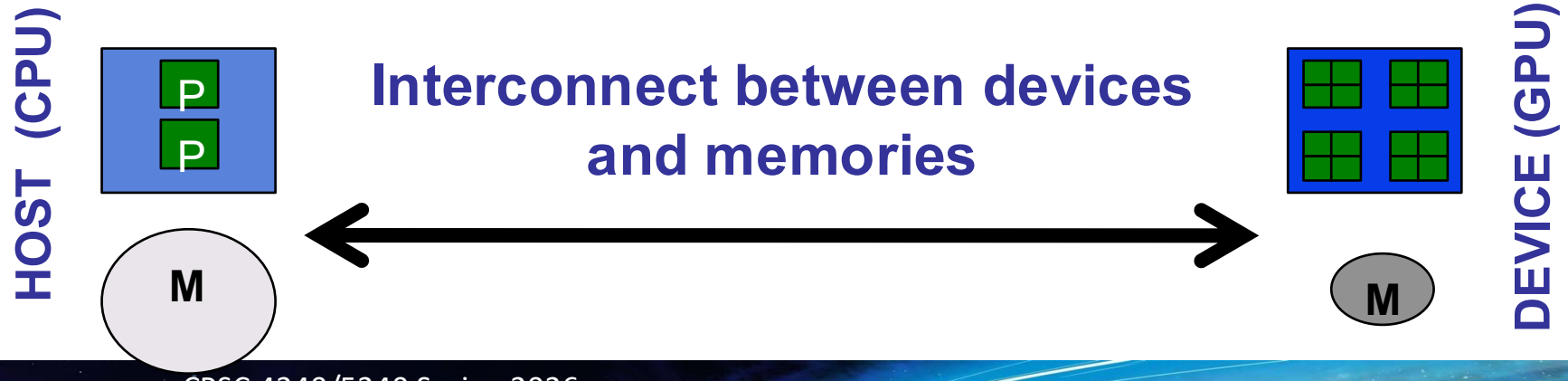
- Where does computation occur?
  - **__host__**: Runs on the CPU (host)
  - **__global__**: A kernel function that runs on the GPU (device), callable from the host



What Programmer Expresses in CUDA

- Where does computation occur?
  - **__host__**: Runs on the CPU (host)
  - **__global__**: A kernel function that runs on the GPU (device), callable from the host
  - **__device__**: A device function that runs on the GPU and can be called only from other device or global functions

```
kernel_function<<<gridSize, blockSize>>>(args);
```



What Programmer Expresses in CUDA

- Where does data reside and who may access it? **Data partitioning**

What Programmer Expresses in CUDA

- Where does data reside and who may access it? **Data partitioning**
  - **__shared__**: Shared memory, accessible by all threads within a single block
  - **__device__**: Global memory on the GPU, accessible by all threads
  - **__constant__**: Read-only constant memory on the GPU, accessible by all threads
  - **(Others)**: Local (per-thread) or texture memory, depending on usage

What Programmer Expresses in CUDA

- Where does data reside and who may access it? **Data partitioning**
  - **__shared__**: Shared memory, accessible by all threads within a single block
  - **__device__**: Global memory on the GPU, accessible by all threads
  - **__constant__**: Read-only constant memory on the GPU, accessible by all threads
  - **(Others)**: Local (per-thread) or texture memory, depending on usage

Explicit Copying between Host (CPU) and Device (GPU)

```
cudaMemcpy(host_data, device_data, size, cudaMemcpyDeviceToHost);
cudaMemcpy(device_data, host_data, size, cudaMemcpyHostToDevice);
```

```
int main() {
    // Print a message from the CPU.
    cpuFunction();

    // Host-side variable to receive the result
    int h_result;

    // Declare a pointer for device memory.
    int* d_result;

    // Allocate memory on the GPU (device)
    cudaMalloc(&d_result, sizeof(int));

    // Launch the kernel:
    // Here we use 1 block with 1 thread for simplicity.
    kernelFunction<<<1, 1>>>(d_result, 3, 4);

    // Wait for the GPU to finish its work (ensure kernel execution is complete).
    cudaDeviceSynchronize();

    // Copy the result from device memory to host memory.
    cudaMemcpy(&h_result, d_result, sizeof(int),
cudaMemcpyDeviceToHost);

    // Free the allocated device memory.
    cudaFree(d_result);

    // Display the result.
    std::cout << "Result (3 + 4): " << h_result << std::endl;

    return 0;
}
```

```
int main() {
    // Print a message from the CPU.
    cpuFunction();

    // Host-side variable to receive the result.
    int h_result;

    // Declare a pointer for device memory.
    int* d_result;

    // Allocate memory on the GPU (device)
    cudaMalloc(&d_result, sizeof(int));

    // Launch the kernel:
    // Here we use 1 block with 1 thread for simplicity.
    kernelFunction<<<1, 1>>>(d_result, 3, 4);

    // Wait for the GPU to finish its work (ensure kernel execution is complete).
    cudaDeviceSynchronize();

    // Copy the result from device memory to host memory.
    cudaMemcpy(&h_result, d_result, sizeof(int),
cudaMemcpyDeviceToHost);

    // Free the allocated device memory.
    cudaFree(d_result);

    // Display the result.
    std::cout << "Result (3 + 4): " << h_result << std::endl;

    return 0;
}
```

```
int main() {
    // Print a message from the CPU.
    cpuFunction();

    // Host-side variable to receive the result.
    int h_result;

    // Declare a pointer for device memory.
    int* d_result;

    // Allocate memory on the GPU (device)
    cudaMalloc(&d_result, sizeof(int));

    // Launch the kernel:
    // Here we use 1 block with 1 thread for simplicity.
    kernelFunction<<<1, 1>>>(d_result, 3, 4);

    // Wait for the GPU to finish its work (ensure kernel execution is complete).
    cudaDeviceSynchronize();

    // Copy the result from device memory to host memory.
    cudaMemcpy(&h_result, d_result, sizeof(int),
cudaMemcpyDeviceToHost);

    // Free the allocated device memory.
    cudaFree(d_result);

    // Display the result.
    std::cout << "Result (3 + 4): " << h_result << std::endl;

    return 0;
}
```

```
int main() {
    // Print a message from the CPU.
    cpuFunction();

    // Host-side variable to receive the result.
    int h_result;

    // Declare a pointer for device memory.
    int* d_result;

    // Allocate memory on the GPU (device)
    cudaMalloc(&d_result, sizeof(int));

    // Launch the kernel:
    // Here we use 1 block with 1 thread for simplicity.
    kernelFunction<<<1, 1>>>(d_result, 3, 4);

    // Wait for the GPU to finish its work (ensure kernel execution is complete).
    cudaDeviceSynchronize();

    // Copy the result from device memory to host memory.
    cudaMemcpy(&h_result, d_result, sizeof(int), cudaMemcpyDeviceToHost);

    // Free the allocated device memory.
    cudaFree(d_result);

    // Display the result.
    std::cout << "Result (3 + 4): " << h_result << std::endl;

    return 0;
}
```

```
int main() {
    // Print a message from the CPU.
    cpuFunction();

    // Host-side variable to receive the result.
    int h_result;

    // Declare a pointer for device memory.
    int* d_result;

    // Allocate memory on the GPU (device)
    cudaMalloc(&d_result, sizeof(int));

    // Launch the kernel:
    // Here we use 1 block with 1 thread for simplicity.
    kernelFunction<<<1, 1>>>(d_result, 3, 4);

    // Wait for the GPU to finish its work (ensure kernel execut  }
    cudaDeviceSynchronize();
```

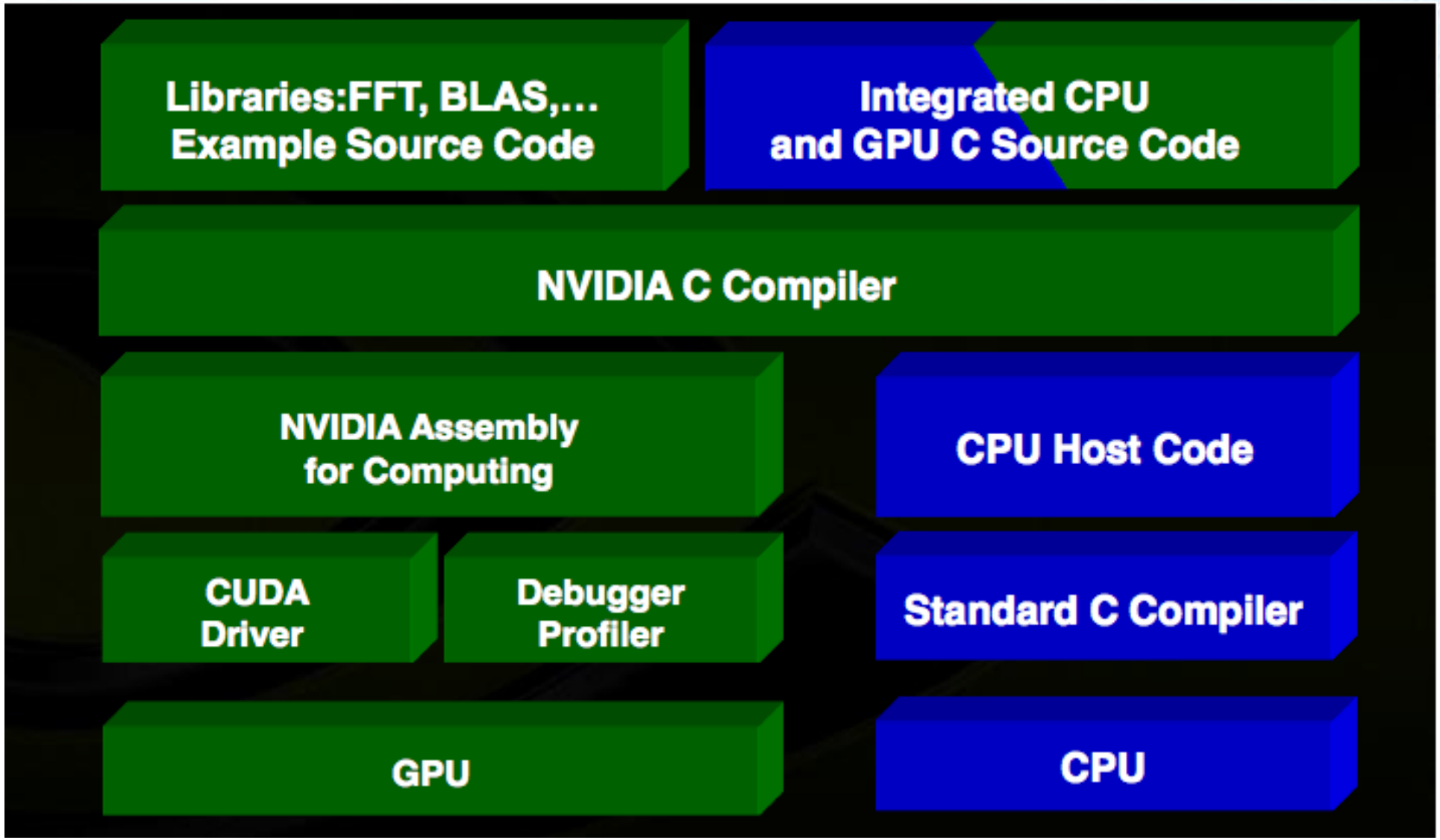
// Copy the result from device memory to host memory.

cudaMemcpy(&h_result, d_result, sizeof(int), cudaMemcpyDeviceToHost);

```
    // Free the allocated device memory.
    cudaFree(d_result);

    // Display the result.
    std::cout << "Result (3 + 4): " << h_result << std::endl;

    return 0;
```

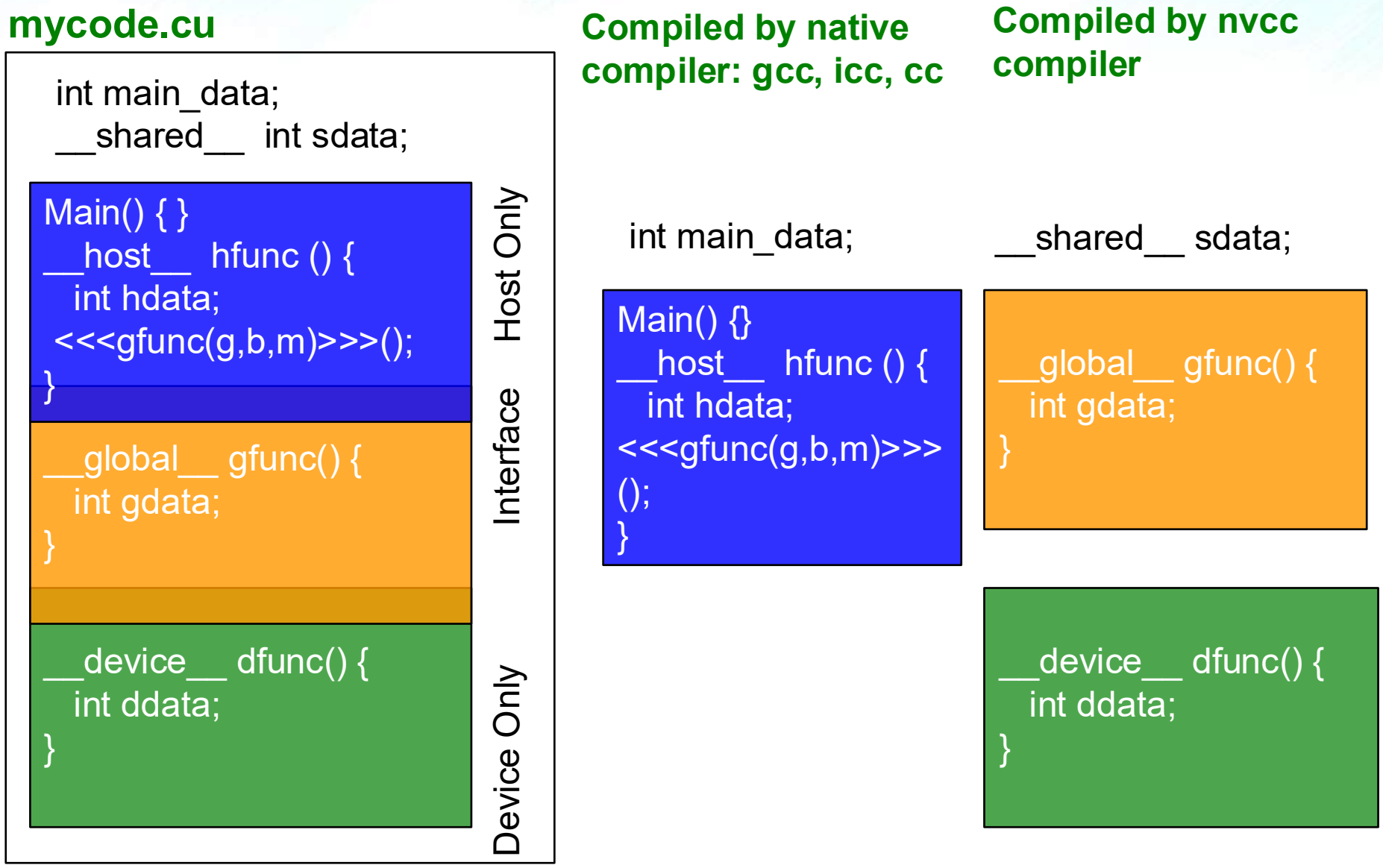
```
int main() {
    // Print a message from the CPU.
    cpuFunction();

    // Host-side variable to receive the result.
    int h_result;

    // Declare a pointer for device memory.
    int* d_result;

    // Allocate memory on the GPU (device)
    cudaMalloc(&d_result, sizeof(int));

    // Launch the kernel:
    // Here we use 1 block with 1 thread for simplicity.
    kernelFunction<<<1, 1>>>(d_result, 3, 4);

    // Wait for the GPU to finish its work (ensure kernel execution is complete).
    cudaDeviceSynchronize();

    // Copy the result from device memory to host memory.
    cudaMemcpy(&h_result, d_result, sizeof(int),
cudaMemcpyDeviceToHost);

    // Free the allocated device memory.
    cudaFree(d_result);

    // Display the result.
    std::cout << "Result (3 + 4): " << h_result << std::endl;

    return 0;
}
```

CUDA Software Developer's Kit (SDK)



Slide source: Austin Robison (NVIDIA)

NVCC Compiler's Role: Partition Code and Compile for Device



Compiling CUDA

- You cannot compile CUDA code directly using gcc

Compiling CUDA

- You cannot compile CUDA code directly using gcc
- Instead, NVIDIA provides the CUDA Compiler (nvcc) as part of the CUDA Toolkit,

Compiling CUDA

- You cannot compile CUDA code directly using gcc
- Instead, NVIDIA provides the CUDA Compiler (nvcc) as part of the CUDA Toolkit,
- **nvcc as a Wrapper:**
  - nvcc processes CUDA code (usually in .cu files) and separates the portions meant for the GPU and CPU (host code).

Compiling CUDA

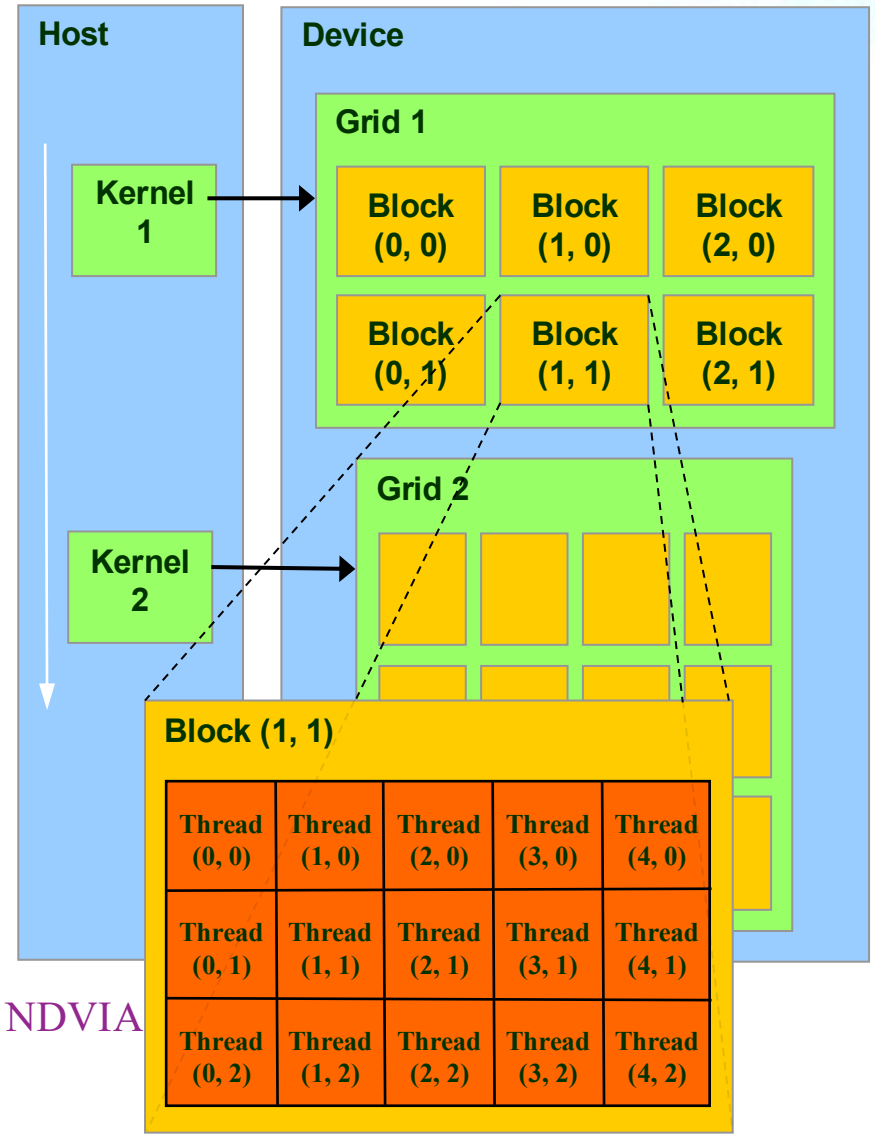
- You cannot compile CUDA code directly using gcc
- Instead, NVIDIA provides the CUDA Compiler (nvcc) as part of the CUDA Toolkit,
- **nvcc as a Wrapper:**
  - nvcc processes CUDA code (usually in .cu files) and separates the portions meant for the GPU and CPU (host code).
- **Host Compiler Integration:**
  - For the host (CPU) parts, nvcc typically invokes gcc

Compiling CUDA

- You cannot compile CUDA code directly using gcc
- Instead, NVIDIA provides the CUDA Compiler (nvcc) as part of the CUDA Toolkit,
- **nvcc as a Wrapper:**
  - nvcc processes CUDA code (usually in .cu files) and separates the portions meant for the GPU and CPU (host code).
- **Host Compiler Integration:**
  - For the host (CPU) parts, nvcc typically invokes gcc
- **Device Code Compilation:**
  - The device code gets compiled using NVIDIA's own compilation tools

Thread Batching: Grids and Blocks

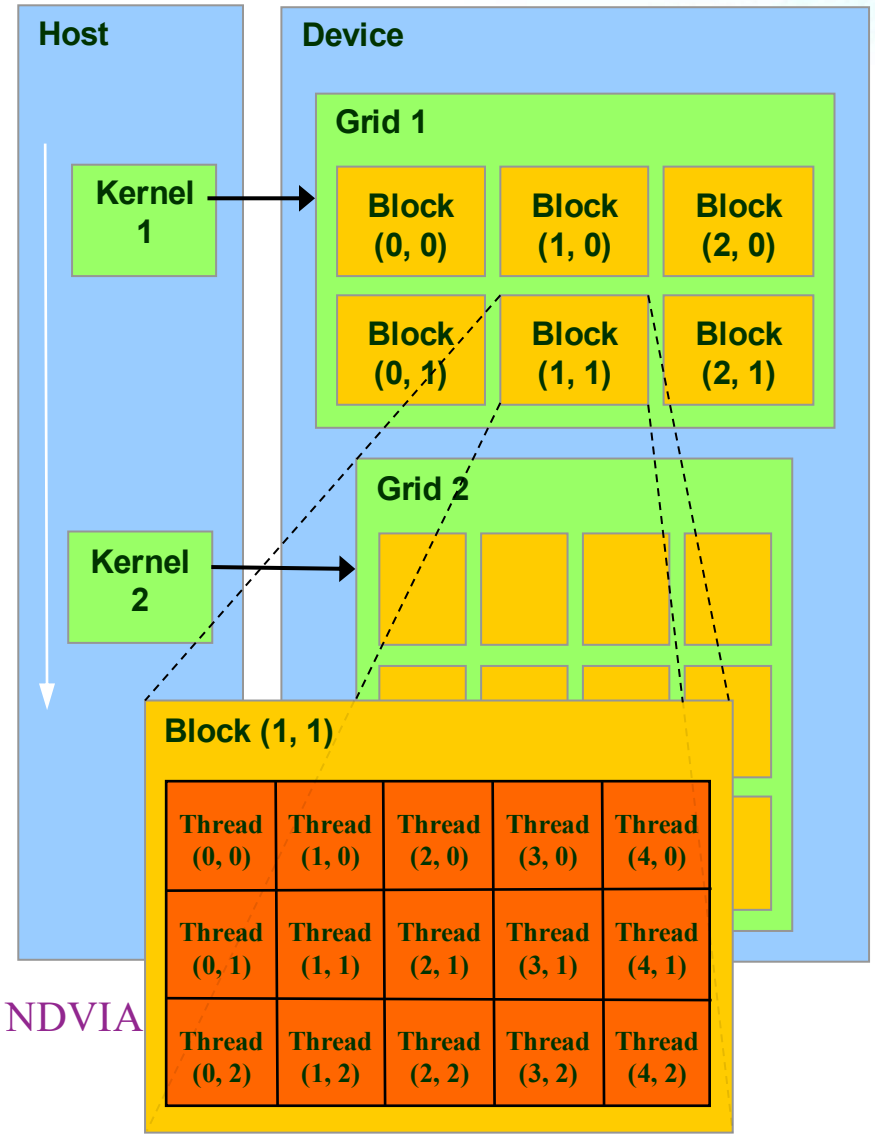
- A kernel is executed as a grid of thread blocks
  - All threads share data memory space



Courtesy: NDVIA

Thread Batching: Grids and Blocks

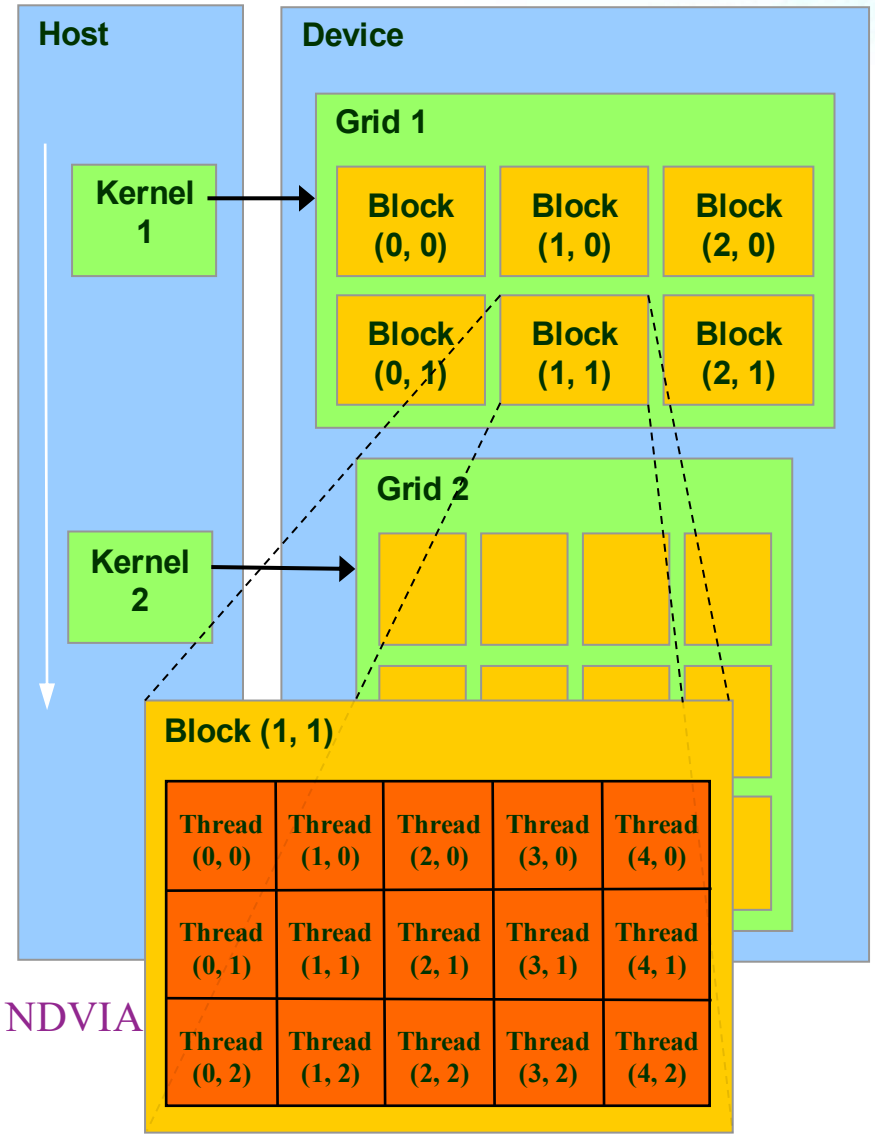
- A kernel is executed as a grid of thread blocks
  - All threads share data memory space
- A thread block is a batch of threads that can cooperate with each other by:



Courtesy: NDVIA

Thread Batching: Grids and Blocks

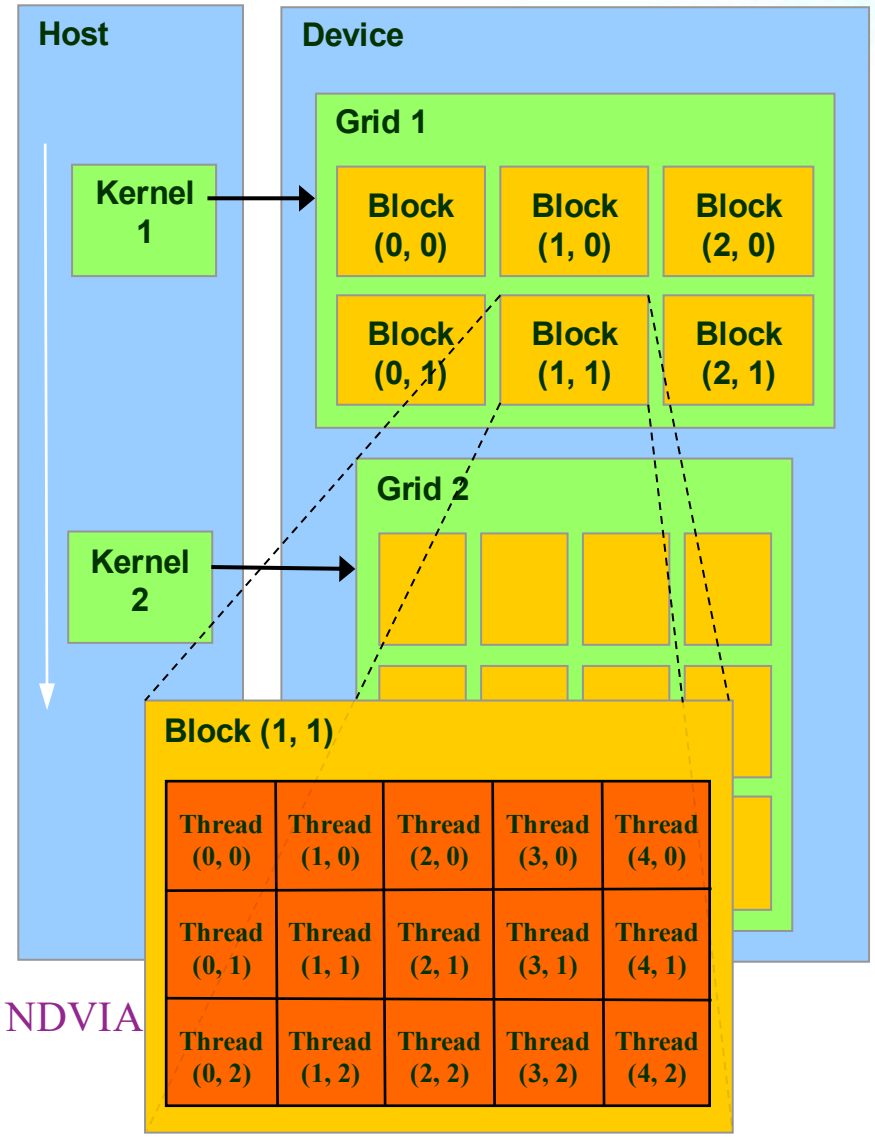
- A kernel is executed as a grid of thread blocks
  - All threads share data memory space
- A thread block is a batch of threads that can cooperate with each other by:
  - Synchronizing their execution
    - For hazard-free shared memory accesses
  - Efficiently sharing data through a low latency shared memory



Courtesy: NDVIA

Thread Batching: Grids and Blocks

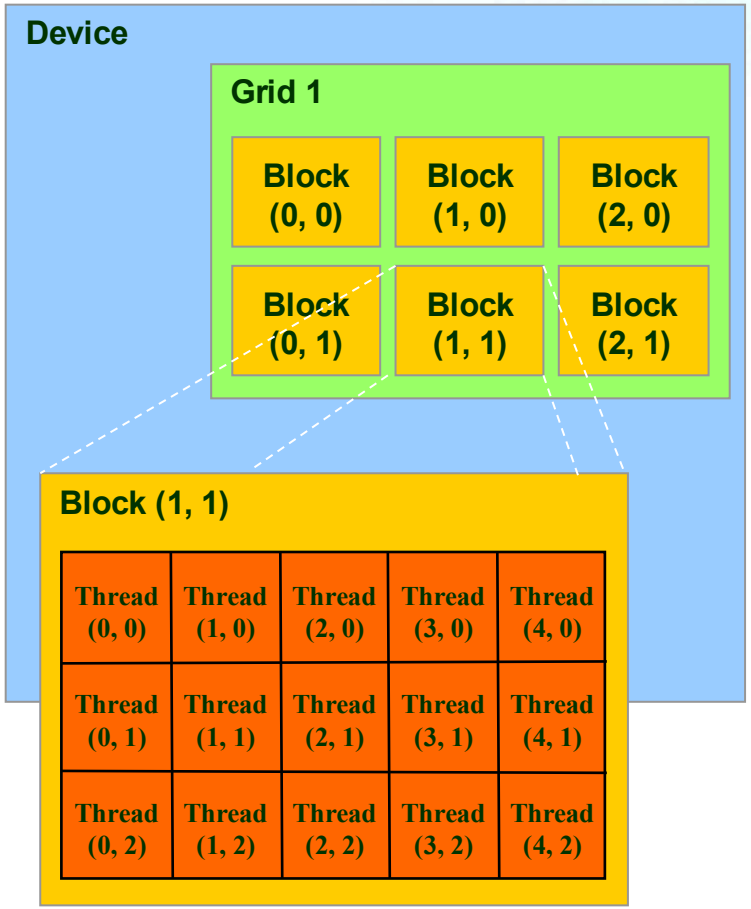
- A kernel is executed as a grid of thread blocks
  - All threads share data memory space
- A thread block is a batch of threads that can cooperate with each other by:
  - Synchronizing their execution
    - For hazard-free shared memory accesses
  - Efficiently sharing data through a low latency shared memory
- Two threads from two different blocks cannot cooperate



Courtesy: NDVIA

Block and Thread IDs

- Threads and blocks have IDs
  - So each thread can decide what data to work on
  - Block ID: 1D or 2D (blockIdx.x, blockIdx.y)
  - Thread ID: 1D, 2D, or 3D (threadIdx.{x,y,z})
- Simplifies memory addressing when processing multidimensional data
  - Image processing
  - Solving PDEs on volumes
  - …



Courtesy: NDVIA

Closer Inspection of Computation and Data Partitioning on GPU

- Define 2-d set of blocks, and 2-d set of threads per block

```
dim3 dimBlock(blocksize,blocksize);
dim3 dimGrid(N/dimBlock.x,N/dimBlock.y);
```

Closer Inspection of Computation and Data Partitioning on GPU

- Define 2-d set of blocks, and 2-d set of threads per block

```
dim3 dimBlock(blocksize,blocksize);
dim3 dimGrid(N/dimBlock.x,N/dimBlock.y);
```

- Each thread identifies what element of the matrix it operates on

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

Closer Inspection of Computation and Data Partitioning on GPU

- Define 2-d set of blocks, and 2-d set of threads per block

```
dim3 dimBlock(blocksize,blocksize);
dim3 dimGrid(N/dimBlock.x,N/dimBlock.y);
```

- Each thread identifies what element of the matrix it operates on

```
int i=blockIdx.x*blockDim.x+threadIdx.x;
int j=blockIdx.y*blockDim.y+threadIdx.y;
```

Closer Inspection of Computation and Data Partitioning on GPU

- Define 2-d set of blocks, and 2-d set of threads per block

```
dim3 dimBlock(blocksize,blocksize);
dim3 dimGrid(N/dimBlock.x,N/dimBlock.y);
```

- Each thread identifies what element of the matrix it operates on

```
int i=blockIdx.x*blockDim.x+threadIdx.x;
int j=blockIdx.y*blockDim.y+threadIdx.y;
int index =i+j*N;
```

Closer Inspection of Computation and Data Partitioning on GPU

- Define 2-d set of blocks, and 2-d set of threads per block

```
dim3 dimBlock(blocksize,blocksize);
dim3 dimGrid(N/dimBlock.x,N/dimBlock.y);
```

- Each thread identifies what element of the matrix it operates on

```
int i=blockIdx.x*blockDim.x+threadIdx.x;
int j=blockIdx.y*blockDim.y+threadIdx.y;
int index =i+j*N;
if( i <N && j <N)
  c[index]=a[index]+b[index];
```

Simple working code example

- What does it do?
  - Scan elements of array of numbers (any of 0 to 9)

Simple working code example

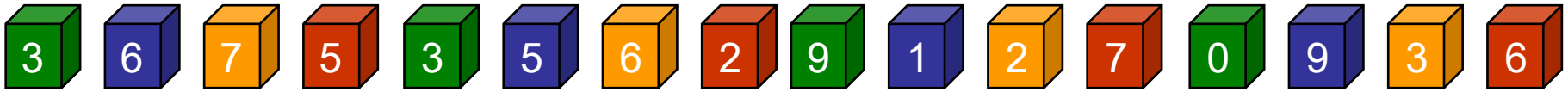
- What does it do?
  - Scan elements of array of numbers (any of 0 to 9)
  - How many times does “6” appear?

Simple working code example

- What does it do?
  - Scan elements of array of numbers (any of 0 to 9)
  - How many times does “6” appear?
  - Array of 16 elements, each thread examines 4 elements, 1 block in grid, 1 grid

Simple working code example

- What does it do?
  - Scan elements of array of numbers (any of 0 to 9)
  - How many times does “6” appear?
  - Array of 16 elements, each thread examines 4 elements, 1 block in grid, 1 grid



threadIdx.x = 0 examines in_array elements 0, 4, 8, 12
threadIdx.x = 1 examines in_array elements 1, 5, 9, 13
threadIdx.x = 2 examines in_array elements 2, 6, 10, 14
threadIdx.x = 3 examines in_array elements 3, 7, 11, 15

} Known as a cyclic data distribution

CUDA Pseudo-Code

MAIN PROGRAM:

Initialization

- Allocate memory on host for input and output
- Assign random numbers to input array

Call ***host*** function

Calculate final output from per-thread output

Print result

CUDA Pseudo-Code

MAIN PROGRAM:

Initialization

- Allocate memory on host for input and output
- Assign random numbers to input array

Call ***host*** function

Calculate final output from per-thread output

Print result

HOST FUNCTION:

Allocate memory on device for copy of ***input*** and ***output***

Copy input to ***device***

Set up grid/block

Call ***global*** function

Copy ***device*** output to host

CUDA Pseudo-Code

MAIN PROGRAM:

Initialization

- Allocate memory on host for input and output
- Assign random numbers to input array

Call ***host*** function

Calculate final output from per-thread output

Print result

GLOBAL FUNCTION:

Thread scans subset of array elements

Call ***device*** function to compare with “6”

Compute local result

HOST FUNCTION:

Allocate memory on device for copy of ***input*** and ***output***

Copy input to ***device***

Set up grid/block

Call ***global*** function

Copy ***device*** output to host

CUDA Pseudo-Code

MAIN PROGRAM:

Initialization

- Allocate memory on host for input and output
- Assign random numbers to input array

Call ***host*** function

Calculate final output from per-thread output

Print result

HOST FUNCTION:

Allocate memory on device for copy of ***input*** and ***output***

Copy input to ***device***

Set up grid/block

Call ***global*** function

Copy ***device*** output to host

GLOBAL FUNCTION:

Thread scans subset of array elements

Call ***device*** function to compare with “6”

Compute local result

DEVICE FUNCTION:

Compare current element and “6”

Return 1 if same, else 0

Main Program: Preliminaries

MAIN PROGRAM:

Initialization

- Allocate memory on host for input and output
- Assign random numbers to input array

Call ***host*** function

Calculate final output from per-thread output

Print result

```
#include <stdio.h>
#define SIZE 16
#define BLOCKSIZE 4

int main(int argc, char **argv)
{
  int *in_array, *out_array;
  …
}
```

Main Program: Invoke Global Function

MAIN PROGRAM:

Initialization (OMIT)

- Allocate memory on host for input and output
- Assign random numbers to input array

Call ***host*** function

Calculate final output from per-thread output

Print result

```
#include <stdio.h>
#define SIZE 16
#define BLOCKSIZE 4
__host__ void outer_compute
        (int *in_arr, int *out_arr);
int main(int argc, char **argv)
{
  int *in_array, *out_array;
  /* initialization */ …
  outer_compute(in_array,
   out_array);
   …
}
```

Main Program

MAIN PROGRAM:

Initialization (OMIT)

- Allocate memory on host for input and output
- Assign random numbers to input array

Call ***host*** function

Calculate final output from per-thread output

Print result

```
#include <stdio.h>
#define SIZE 16
#define BLOCKSIZE 4
__host__ void outer_compute     (int *in_arr,
  int *out_arr);
int main(int argc, char **argv)
{
  int *in_array, *out_array;
  int sum = 0;
  /* initialization */ …
  outer_compute(in_array, out_array);
  for (int i=0; i<BLOCKSIZE; i++) {
    sum+=out_array[i];
  }
  printf (”Result = %d\n",sum);
}
```

Host Function: Preliminaries & Allocation

HOST FUNCTION:

Allocate memory on device for copy of ***input*** and ***output***

Copy input to ***device***

Set up grid/block

Call ***global*** function

Copy ***device*** output to host

```
__host__ void outer_compute (int
  *h_in_array, int *h_out_array) {

   int *d_in_array, *d_out_array;


   cudaMalloc((void **) &d_in_array,
         SIZE*sizeof(int));

   cudaMalloc((void **) &d_out_array,
         BLOCKSIZE*sizeof(int));


   …
}
```

Host Function: Copy Data To/From Host

HOST FUNCTION:

Allocate memory on device for copy of ***input*** and ***output***

Copy input to ***device***

Set up grid/block

Call ***global*** function

Copy ***device*** output to host

```
__host__ void outer_compute (int
*h_in_array, int *h_out_array) {
int *d_in_array, *d_out_array;

cudaMalloc((void **) &d_in_array,
      SIZE*sizeof(int));

cudaMalloc((void **) &d_out_array,
      BLOCKSIZE*sizeof(int));

cudaMemcpy(d_in_array, h_in_array,
      SIZE*sizeof(int),
      cudaMemcpyHostToDevice);

… do computation …

cudaMemcpy(h_out_array,d_out_array,
      BLOCKSIZE*sizeof(int),
      cudaMemcpyDeviceToHost);

}
```

Host Function: Setup & Call Global Function

HOST FUNCTION:

Allocate memory on device for copy of ***input*** and ***output***

Copy input to ***device***

Set up grid/block

Call ***global*** function

Copy ***device*** output to host

```
__host__ void outer_compute (int
*h_in_array, int *h_out_array) {
int *d_in_array, *d_out_array;


cudaMalloc((void **) &d_in_array,
      SIZE*sizeof(int));

cudaMalloc((void **) &d_out_array,
      BLOCKSIZE*sizeof(int));
cudaMemcpy(d_in_array, h_in_array,
      SIZE*sizeof(int),
      cudaMemcpyHostToDevice);
msize = (SIZE+BLOCKSIZE) *
      sizeof (int);
compute<<<1,BLOCKSIZE,msize)>>>
      (d_in_array, d_out_array);
cudaMemcpy(h_out_array, d_out_array,
      BLOCKSIZE*sizeof(int),
      cudaMemcpyDeviceToHost);

}
```

Global Function

GLOBAL FUNCTION:

Thread scans subset of array elements

Call ***device*** function to compare with “6”

Compute local result

```
__global__ void compute(int *d_in,int *d_out)
 {
 d_out[threadIdx.x] = 0;
 for (int i=0; i<SIZE/BLOCKSIZE;        i++)
  {
     int val = d_in[i*BLOCKSIZE +
 threadIdx.x];
    d_out[threadIdx.x] += compare(val, 6);
  }
}
```

Device Function

DEVICE FUNCTION:

Compare current element and “6”

Return 1 if same, else 0

```
__device__ int
  compare(int a, int b)
  {
  if (a == b) return 1;
  return 0;
}
```

Reductions

- This type of computation is called a ***parallel reduction***
  - Operation is applied to large data structure

Reductions

- This type of computation is called a ***parallel reduction***
  - Operation is applied to large data structure
  - Computed result represents the aggregate solution across the large data structure

Reductions

- This type of computation is called a ***parallel reduction***
  - Operation is applied to large data structure
  - Computed result represents the aggregate solution across the large data structure
  - Large data structure → computed result (perhaps single number) **[dimensionality reduced]**

Reductions

- This type of computation is called a ***parallel reduction***
  - Operation is applied to large data structure
  - Computed result represents the aggregate solution across the large data structure
  - Large data structure → computed result (perhaps single number) **[dimensionality reduced]**
- Why might parallel reductions be well-suited to GPUs?

Reductions

- This type of computation is called a ***parallel reduction***
  - Operation is applied to large data structure
  - Computed result represents the aggregate solution across the large data structure
  - Large data structure → computed result (perhaps single number) **[dimensionality reduced]**
- Why might parallel reductions be well-suited to GPUs?
- What if we tried to compute the final sum on the GPUs?

Standard Parallel Construct

- Sometimes called “embarassingly parallel” or “pleasingly parallel”

Standard Parallel Construct

- Sometimes called “embarassingly parallel” or “pleasingly parallel”
- Each thread is completely independent of the others

Standard Parallel Construct

- Sometimes called “embarassingly parallel” or “pleasingly parallel”
- Each thread is completely independent of the others
- Final result copied to CPU

Standard Parallel Construct

- Sometimes called “embarassingly parallel” or “pleasingly parallel”
- Each thread is completely independent of the others
- Final result copied to CPU
- Another example, adding two matrices:
  - A more careful examination of decomposing computation into grids and thread blocks

Best Practices for Writing CUDA Programs

Approach to Working with New Systems

- Approach with **caution**

Approach to Working with New Systems

- Approach with **caution**
  - May not behave as expected.

Approach to Working with New Systems

- Approach with **caution**
  - May not behave as expected.
  - Learn about behavior through observation and measurement.

Approach to Working with New Systems

- Approach with **caution**
  - May not behave as expected.
  - Learn about behavior through observation and measurement.
  - Realize that **documentation is not always clear or accurate**

Approach to Working with New Systems

- Approach with **caution**
  - May not behave as expected.
  - Learn about behavior through observation and measurement.
  - Realize that **documentation is not always clear or accurate**
- Crawl, then walk, then run

Approach to Working with New Systems

- Approach with **caution**
  - May not behave as expected.
  - Learn about behavior through observation and measurement.
  - Realize that **documentation is not always clear or accurate**
- Crawl, then walk, then run
  - Start with something known to work.

Approach to Working with New Systems

- Approach with **caution**
  - May not behave as expected.
  - Learn about behavior through observation and measurement.
  - Realize that **documentation is not always clear or accurate**
- Crawl, then walk, then run
  - Start with something known to work.
  - Only change one “variable” at a time.