Introduction to GPU programming using CUDA

Felice Pantaleo
EP-CMG-CO

felice@cern.ch

Prerequisites

- You need experience with C or C++
- You don’t need GPU experience
- You don’t need parallel programming experience
- You don’t need graphics experience

Content of the theoretical session

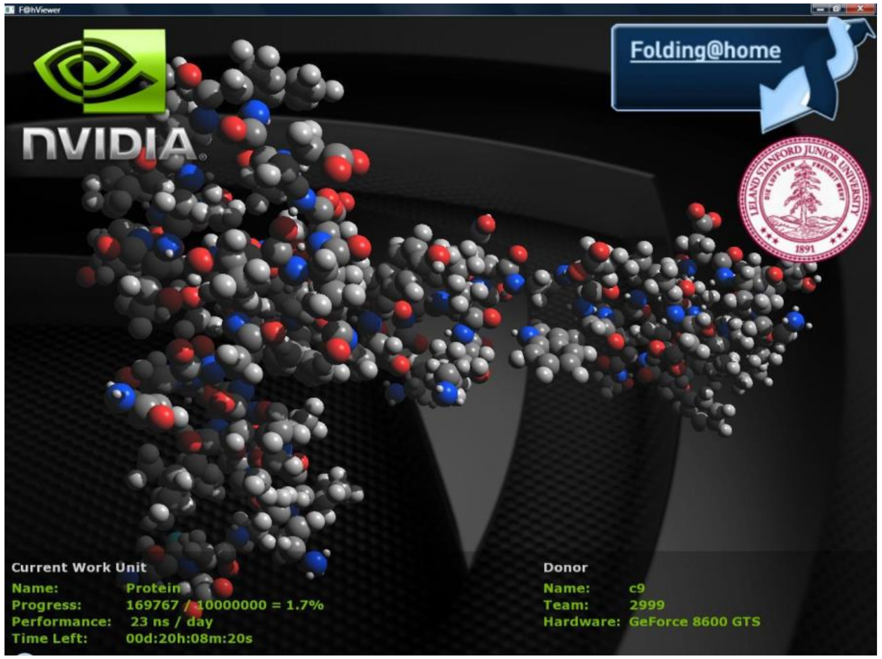
- Heterogeneous Parallel computing systems
- CUDA Basics
- Parallel constructs in CUDA
- Shared Memory
- Device Management
- Thrust

Content of the tutorial session

- Write and launch CUDA C/C++ kernels
- Manage GPU memory
- Manage communication and synchronization
- …Think parallel

Accelerators

- Exceptional raw power wrt CPUs
- Higher energy efficiency
- Plug & Accelerate
- Massively parallel architecture
- Low Memory/core



Current Work Unit

Name:	Protein
Progress:	169767 / 10000000 = 1.7%
Performance:	23 ns / day
Time Left:	00d:20h:08m:20s

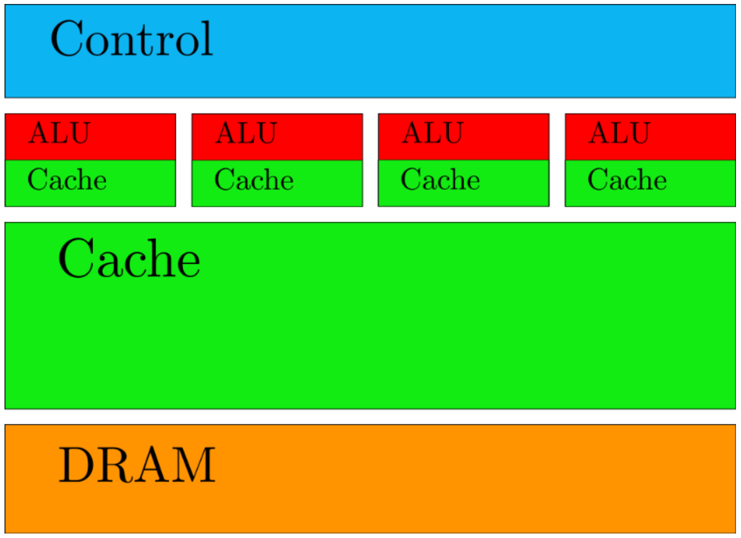
Donor

Name:	c9
Team:	2999
Hardware:	GeForce 8600 GTS

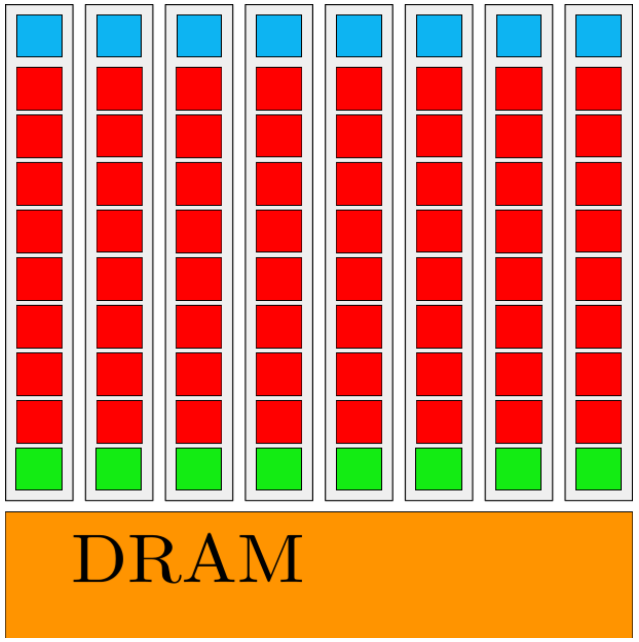
Accelerators

- GPUs were traditionally used for real-time rendering. NVIDIA & AMD main manufacturers.
- Intel introduced the coprocessor Xeon Phi (MIC)

CPU vs GPU architectures

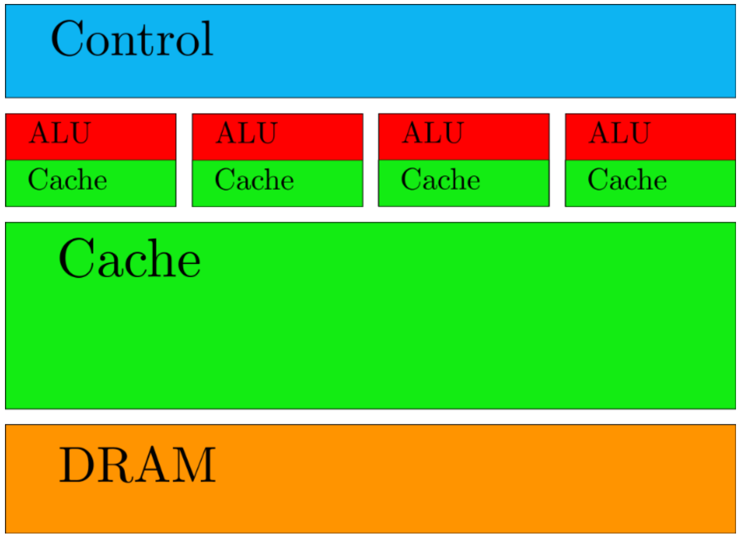


CPU



GPU

CPU vs GPU architectures

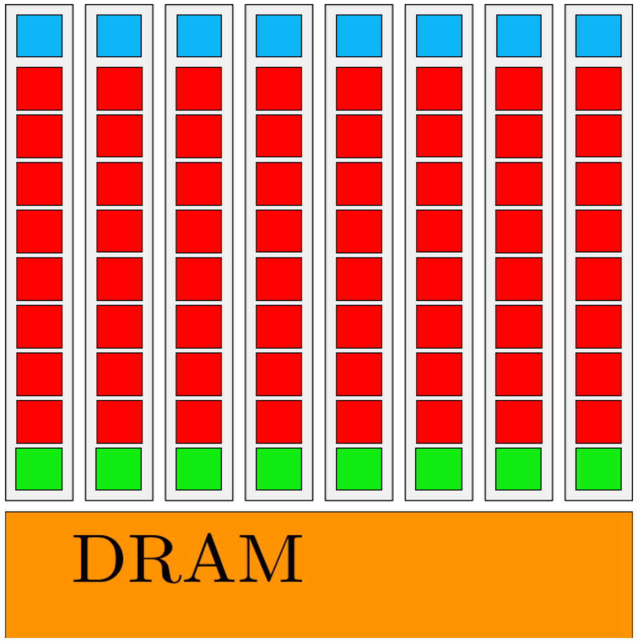


CPU

- Large caches (slow memory accesses to quick cache accesses)
- SIMD
- Branch prediction
- Data forwarding
- Powerful ALU
- Pipelining

CPU vs GPU architectures

- SMX* executes kernels (aka functions) using hundreds of threads concurrently.
- SIMT (Single-Instruction, Multiple-Thread)
- Instructions pipelined
- Thread-level parallelism
- Instructions issued in order
- No Branch prediction
- Branch predication
- Cost ranging from few hundreds to a thousand euros depending on features (e.g. NVIDIA GTX680 400 euros)



GPU

*SMX = Streaming multiprocessor

Throughput

Theoretical peak throughput: the maximum amount of data that a kernel can read **and** produce in the unit time.

Throughput

Theoretical peak throughput: the maximum amount of data that a kernel can read **and** produce in the unit time.

$$\text{Throughput}_{\text{peak}}\ (\text{GB/s}) = 2 \times \text{access width (byte)} \times \text{mem\_freq (GHz)}$$

Throughput

Theoretical peak throughput: the maximum amount of data that a kernel can read **and** produce in the unit time.

$$\text{Throughput}_{\text{peak}}\ (\text{GB/s}) = 2 \times \text{access width (byte)} \times \text{mem\_freq (GHz)}$$

This means that if your device comes with a memory clock rate of 1GHz DDR (double data rate) and a 384-bit wide memory interface, the amount of data that a kernel can process and produce in the unit time is at most:

Throughput

Theoretical peak throughput: the maximum amount of data that a kernel can read **and** produce in the unit time.

$$\text{Throughput}_{\text{peak}} \text{ (GB/s)} = 2 \text{ x access width (byte) x mem\_freq (GHz)}$$

This means that if your device comes with a memory clock rate of 1GHz DDR (double data rate) and a 384-bit wide memory interface, the amount of data that a kernel can process and produce in the unit time is at most:

$$\text{Throughput}_{\text{peak}} \text{ (GB/s)} = 2 \text{ x } (384/8)\text{(byte) x 1 (GHz)} = 96\text{GB/s}$$

Global memory

Kepler Architecture:

- 1.3 TFLOPS DPFP peak throughput
- 250 GB/s peak off-chip memory access bandwidth
- 31 G DPFP operands per second
- To achieve peak throughput, a program must perform 1,300/31 = ~42 FP arithmetic operations for each operand value fetched from off-chip memory

Bandwidth

Bandwidth

Conflicting data

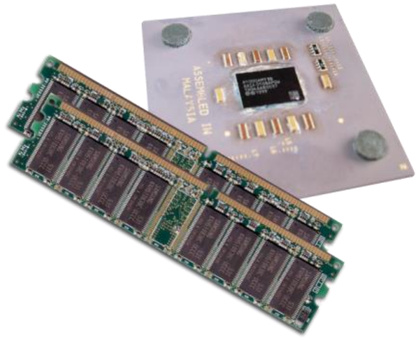
Conflicting Data Updates Cause Serialization and Delays:

- Massively parallel execution cannot afford serialization
- Contentions in updating critical data causes serialization

Heterogeneous Parallel Computing Systems

Heterogeneous Computing

- Terminology
  - Host   The CPU and its memory space
  - Device   The GPU and its memory space


Host

Device

Heterogeneous Computing

```
#include <iostream>
#include <algorithm>

using namespace std;

#define N          1024
#define RADIUS     3
#define BLOCK_SIZE  16

__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();

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

            // Store the result
            out[gindex] = result;
}

void fill_ints(int *x, int n) {
            fill_n(x, n, 1);
}

int main(void) {
            int *in, *out;              // host copies of a, b, c
            int *d_in, *d_out;          // device copies of a, b, c
            int size = (N + 2*RADIUS) * sizeof(int);

            // Alloc space for host copies and setup values
            in  = (int *)malloc(size); fill_ints(in,  N + 2*RADIUS);
            out = (int *)malloc(size); fill_ints(out, N + 2*RADIUS);

            // Alloc space for device copies
            cudaMalloc((void **)&d_in,  size);
            cudaMalloc((void **)&d_out, size);

            // Copy to device
            cudaMemcpy(d_in,  in,  size, cudaMemcpyHostToDevice);
            cudaMemcpy(d_out, out, size, cudaMemcpyHostToDevice);

            // Launch stencil_1d() kernel on GPU
            stencil_1d<<<N/BLOCK_SIZE,BLOCK_SIZE>>>(d_in + RADIUS,
d_out + RADIUS);

            // Copy result back to host
            cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);

            // Cleanup
            free(in); free(out);
            cudaFree(d_in); cudaFree(d_out);
            return 0;
}
```

Heterogeneous Computing

```
#include <iostream>
#include <algorithm>

using namespace std;

#define N         1024
#define RADIUS     3
#define BLOCK_SIZE  16

__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();

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

        // Store the result
        out[gindex] = result;
}

void fill_ints(int *x, int n) {
        fill_n(x, n, 1);
}

int main(void) {
        int *in, *out;           // host copies of a, b, c
        int *d_in, *d_out;       // device copies of a, b, c
        int size = (N + 2*RADIUS) * sizeof(int);

        // Alloc space for host copies and setup values
        in  = (int *)malloc(size); fill_ints(in,  N + 2*RADIUS);
        out = (int *)malloc(size); fill_ints(out, N + 2*RADIUS);

        // Alloc space for device copies
        cudaMalloc((void **)&d_in,  size);
        cudaMalloc((void **)&d_out, size);

        // Copy to device
        cudaMemcpy(d_in,  in,  size, cudaMemcpyHostToDevice);
        cudaMemcpy(d_out, out, size, cudaMemcpyHostToDevice);

        // Launch stencil_1d() kernel on GPU
        stencil_1d<<<N/BLOCK_SIZE,BLOCK_SIZE>>>(d_in + RADIUS,
d_out + RADIUS);

        // Copy result back to host
        cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);

        // Cleanup
        free(in); free(out);
        cudaFree(d_in); cudaFree(d_out);
        return 0;
}
```

serial code

Heterogeneous Computing

```
#include <iostream>
#include <algorithm>

using namespace std;

#define N         1024
#define RADIUS     3
#define BLOCK_SIZE  16

__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();

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

          // Store the result
          out[gindex] = result;
}

void fill_ints(int *x, int n) {
          fill_n(x, n, 1);
}

int main(void) {
          int *in, *out;            // host copies of a, b, c
          int *d_in, *d_out;          // device copies of a, b, c
          int size = (N + 2*RADIUS) * sizeof(int);

          // Alloc space for host copies and setup values
          in  = (int *)malloc(size); fill_ints(in,  N + 2*RADIUS);
          out = (int *)malloc(size); fill_ints(out, N + 2*RADIUS);

          // Alloc space for device copies
          cudaMalloc((void **)&d_in,  size);
          cudaMalloc((void **)&d_out, size);

          // Copy to device
          cudaMemcpy(d_in,  in,  size, cudaMemcpyHostToDevice);
          cudaMemcpy(d_out, out, size, cudaMemcpyHostToDevice);

          // Launch stencil_1d() kernel on GPU
          stencil_1d<<<N/BLOCK_SIZE,BLOCK_SIZE>>>(d_in + RADIUS,
d_out + RADIUS);

          // Copy result back to host
          cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);

          // Cleanup
          free(in); free(out);
          cudaFree(d_in); cudaFree(d_out);
          return 0;
}
```

parallel fn

serial code

parallel code

Heterogeneous Computing

```
#include <iostream>
#include <algorithm>

using namespace std;

#define N         1024
#define RADIUS    3
#define BLOCK_SIZE  16

__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();

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

        // Store the result
        out[gindex] = result;
}

void fill_ints(int *x, int n) {
        fill_n(x, n, 1);
}

int main(void) {
        int *in, *out;            // host copies of a, b, c
        int *d_in, *d_out;         // device copies of a, b, c
        int size = (N + 2*RADIUS) * sizeof(int);

        // Alloc space for host copies and setup values
        in  = (int *)malloc(size); fill_ints(in,  N + 2*RADIUS);
        out = (int *)malloc(size); fill_ints(out, N + 2*RADIUS);

        // Alloc space for device copies
        cudaMalloc((void **)&d_in,  size);
        cudaMalloc((void **)&d_out, size);

        // Copy to device
        cudaMemcpy(d_in,  in,  size, cudaMemcpyHostToDevice);
        cudaMemcpy(d_out, out, size, cudaMemcpyHostToDevice);

        // Launch stencil_1d() kernel on GPU
        stencil_1d<<<N/BLOCK_SIZE,BLOCK_SIZE>>>(d_in + RADIUS,
d_out + RADIUS);

        // Copy result back to host
        cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);

        // Cleanup
        free(in); free(out);
        cudaFree(d_in); cudaFree(d_out);
        return 0;
}
```

parallel fn

serial code

parallel code

serial code

Heterogeneous Computing

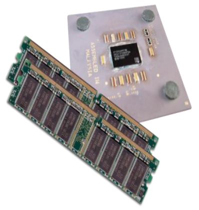
```
#include <iostream>
#include <algorithm>

using namespace std;

#define N         1024
#define RADIUS     3
#define BLOCK_SIZE  16

__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();

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

        // Store the result
        out[gindex] = result;
}

void fill_ints(int *x, int n) {
        fill_n(x, n, 1);
}

int main(void) {
        int *in, *out;          // host copies of a, b, c
        int *d_in, *d_out;      // device copies of a, b, c
        int size = (N + 2*RADIUS) * sizeof(int);

        // Alloc space for host copies and setup values
        in  = (int *)malloc(size); fill_ints(in,  N + 2*RADIUS);
        out = (int *)malloc(size); fill_ints(out, N + 2*RADIUS);

        // Alloc space for device copies
        cudaMalloc((void **)&d_in,  size);
        cudaMalloc((void **)&d_out, size);

        // Copy to device
        cudaMemcpy(d_in,  in,  size, cudaMemcpyHostToDevice);
        cudaMemcpy(d_out, out, size, cudaMemcpyHostToDevice);

        // Launch stencil_1d() kernel on GPU
        stencil_1d<<<N/BLOCK_SIZE,BLOCK_SIZE>>>(d_in + RADIUS,
d_out + RADIUS);

        // Copy result back to host
        cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);

        // Cleanup
        free(in); free(out);
        cudaFree(d_in); cudaFree(d_out);
        return 0;
}
```

parallel fn

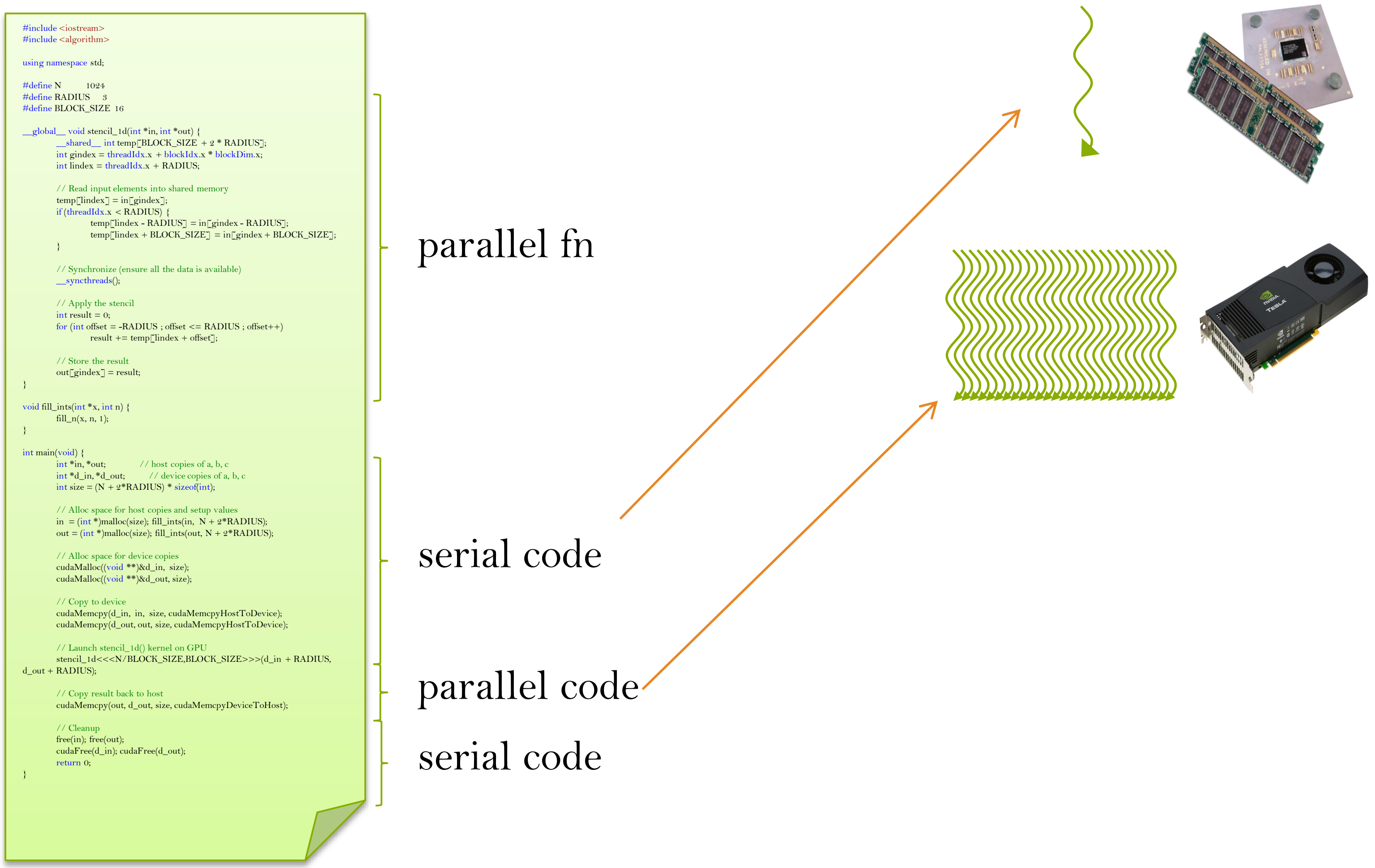
serial code

parallel code

serial code

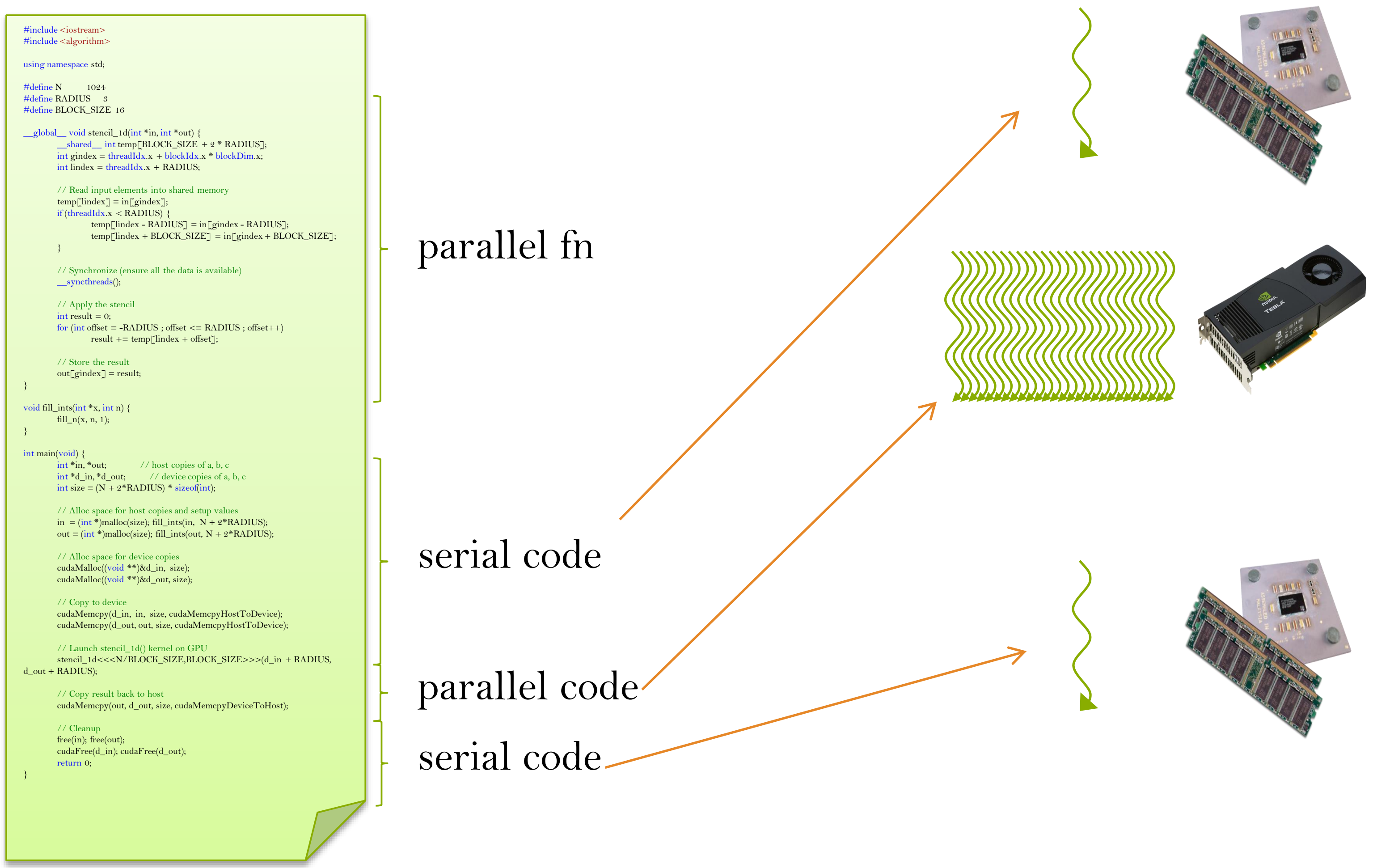


Heterogeneous Computing



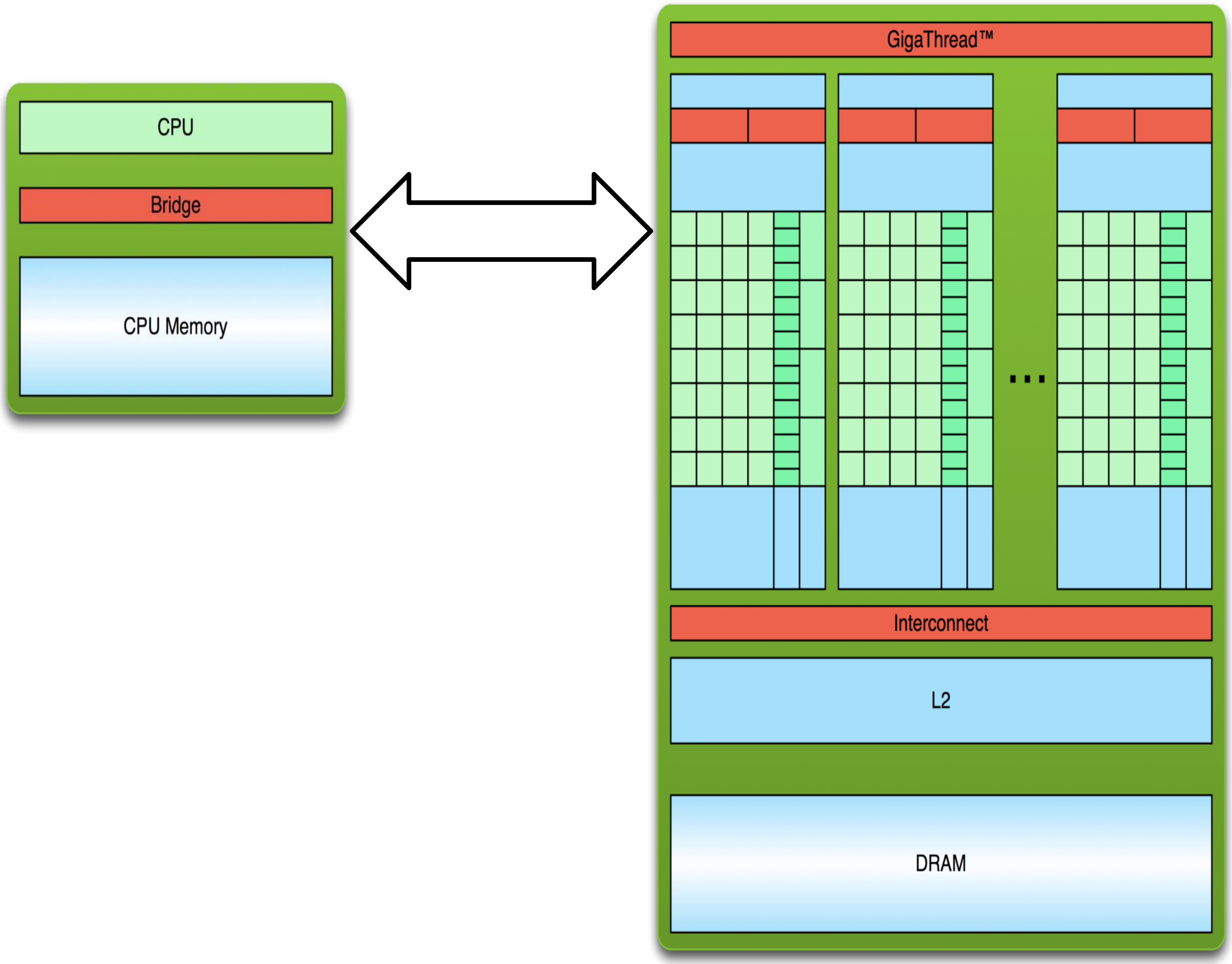
```
#include <iostream>
#include <algorithm>

using namespace std;

#define N        1024
#define RADIUS    3
#define BLOCK_SIZE 16

__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();

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

          // Store the result
          out[gindex] = result;
}

void fill_ints(int *x, int n) {
          fill_n(x, n, 1);
}

int main(void) {
          int *in, *out;            // host copies of a, b, c
          int *d_in, *d_out;        // device copies of a, b, c
          int size = (N + 2*RADIUS) * sizeof(int);

          // Alloc space for host copies and setup values
          in  = (int *)malloc(size); fill_ints(in,  N + 2*RADIUS);
          out = (int *)malloc(size); fill_ints(out, N + 2*RADIUS);

          // Alloc space for device copies
          cudaMalloc((void **)&d_in,  size);
          cudaMalloc((void **)&d_out, size);

          // Copy to device
          cudaMemcpy(d_in,  in,  size, cudaMemcpyHostToDevice);
          cudaMemcpy(d_out, out, size, cudaMemcpyHostToDevice);

          // Launch stencil_1d() kernel on GPU
          stencil_1d<<<N/BLOCK_SIZE,BLOCK_SIZE>>>(d_in + RADIUS,
d_out + RADIUS);

          // Copy result back to host
          cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);

          // Cleanup
          free(in); free(out);
          cudaFree(d_in); cudaFree(d_out);
          return 0;
}
```

Heterogeneous Computing

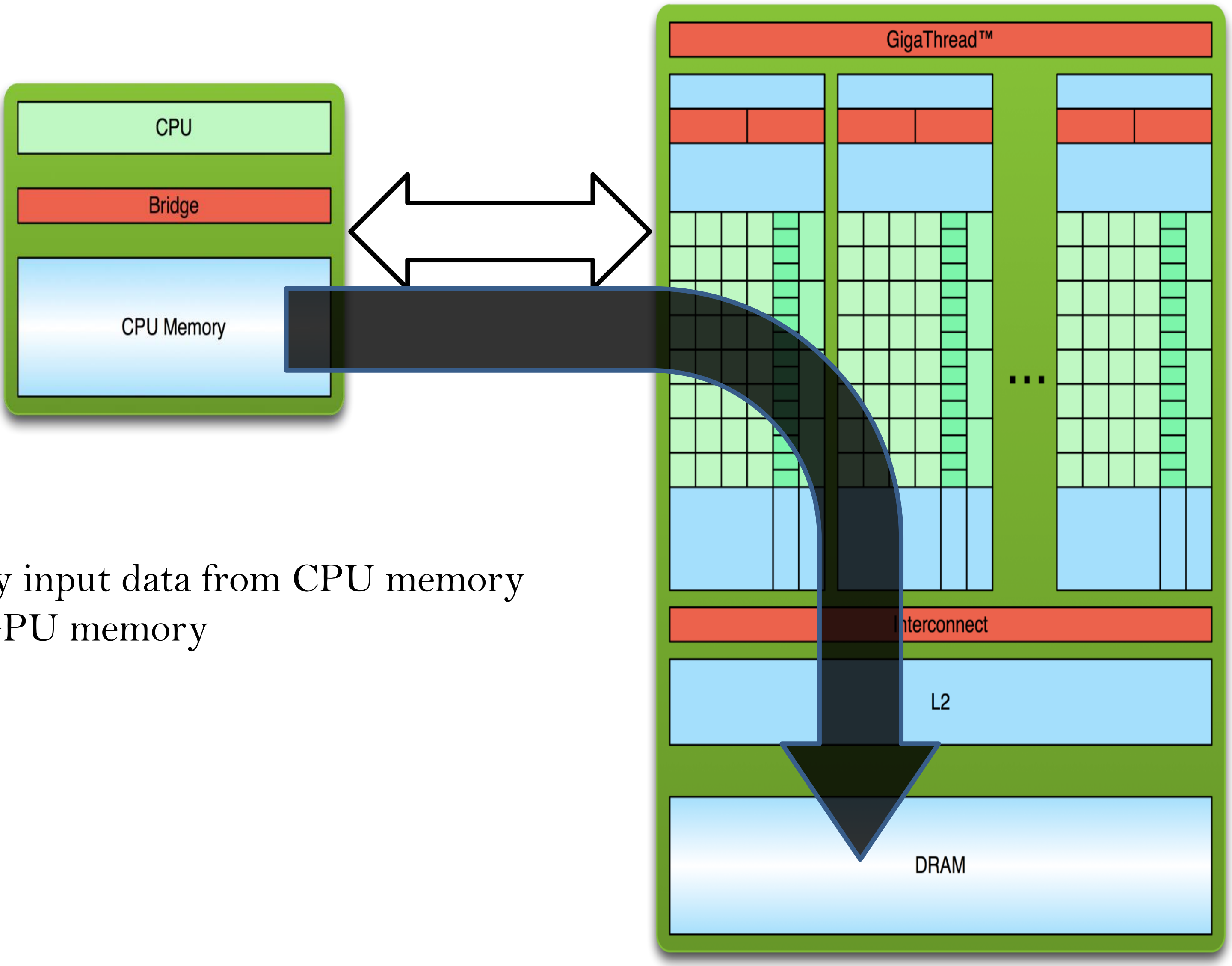


```
#include <iostream>
#include <algorithm>

using namespace std;

#define N          1024
#define RADIUS     3
#define BLOCK_SIZE  16

__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();

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

        // Store the result
        out[gindex] = result;
}

void fill_ints(int *x, int n) {
        fill_n(x, n, 1);
}

int main(void) {
        int *in, *out;            // host copies of a, b, c
        int *d_in, *d_out;        // device copies of a, b, c
        int size = (N + 2*RADIUS) * sizeof(int);

        // Alloc space for host copies and setup values
        in  = (int *)malloc(size); fill_ints(in,  N + 2*RADIUS);
        out = (int *)malloc(size); fill_ints(out, N + 2*RADIUS);

        // Alloc space for device copies
        cudaMalloc((void **)&d_in,  size);
        cudaMalloc((void **)&d_out, size);

        // Copy to device
        cudaMemcpy(d_in,  in,  size, cudaMemcpyHostToDevice);
        cudaMemcpy(d_out, out, size, cudaMemcpyHostToDevice);

        // Launch stencil_1d() kernel on GPU
        stencil_1d<<<N/BLOCK_SIZE,BLOCK_SIZE>>>(d_in + RADIUS,
d_out + RADIUS);

        // Copy result back to host
        cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);

        // Cleanup
        free(in); free(out);
        cudaFree(d_in); cudaFree(d_out);
        return 0;
}
```

Simple Processing Flow

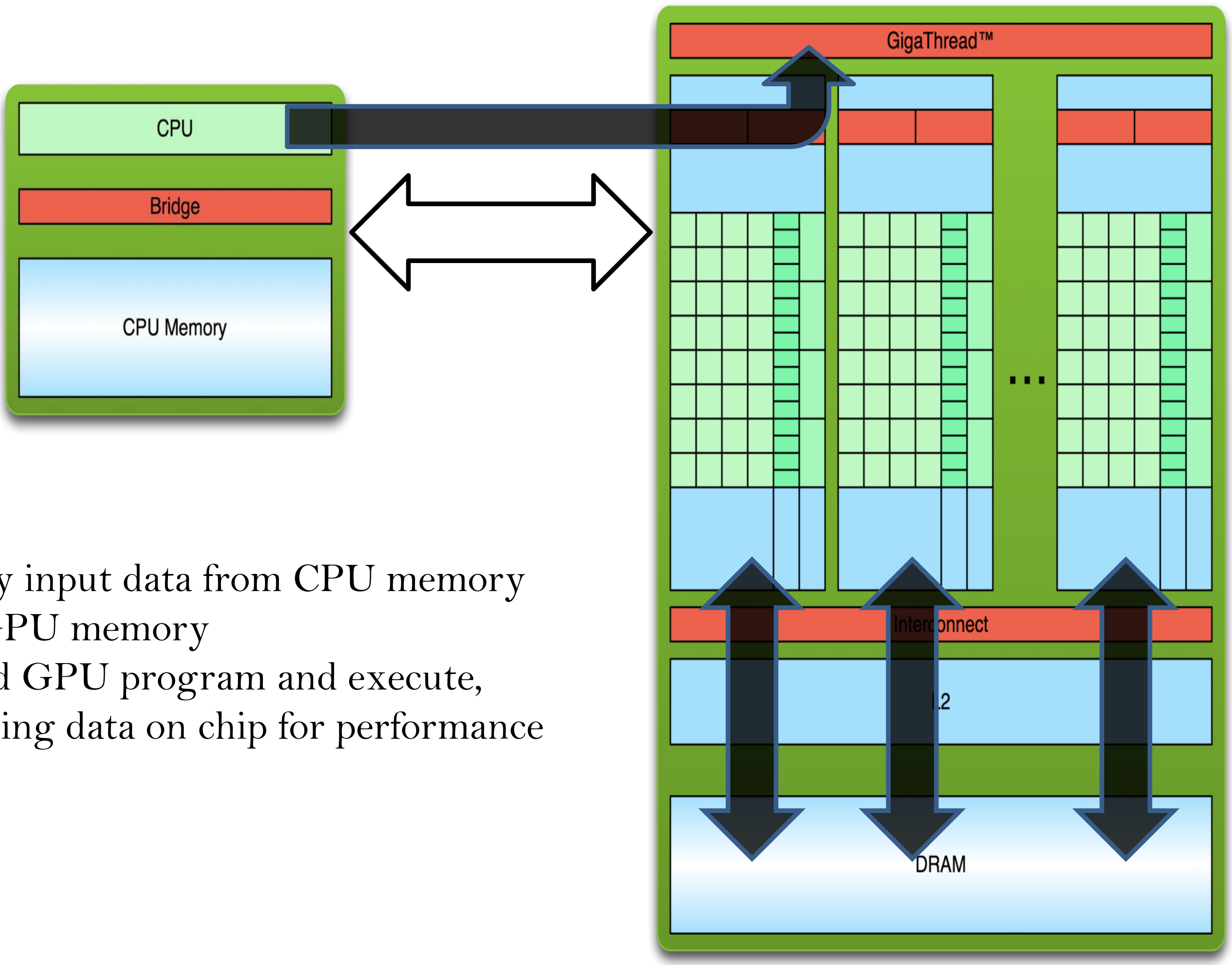


Simple Processing Flow



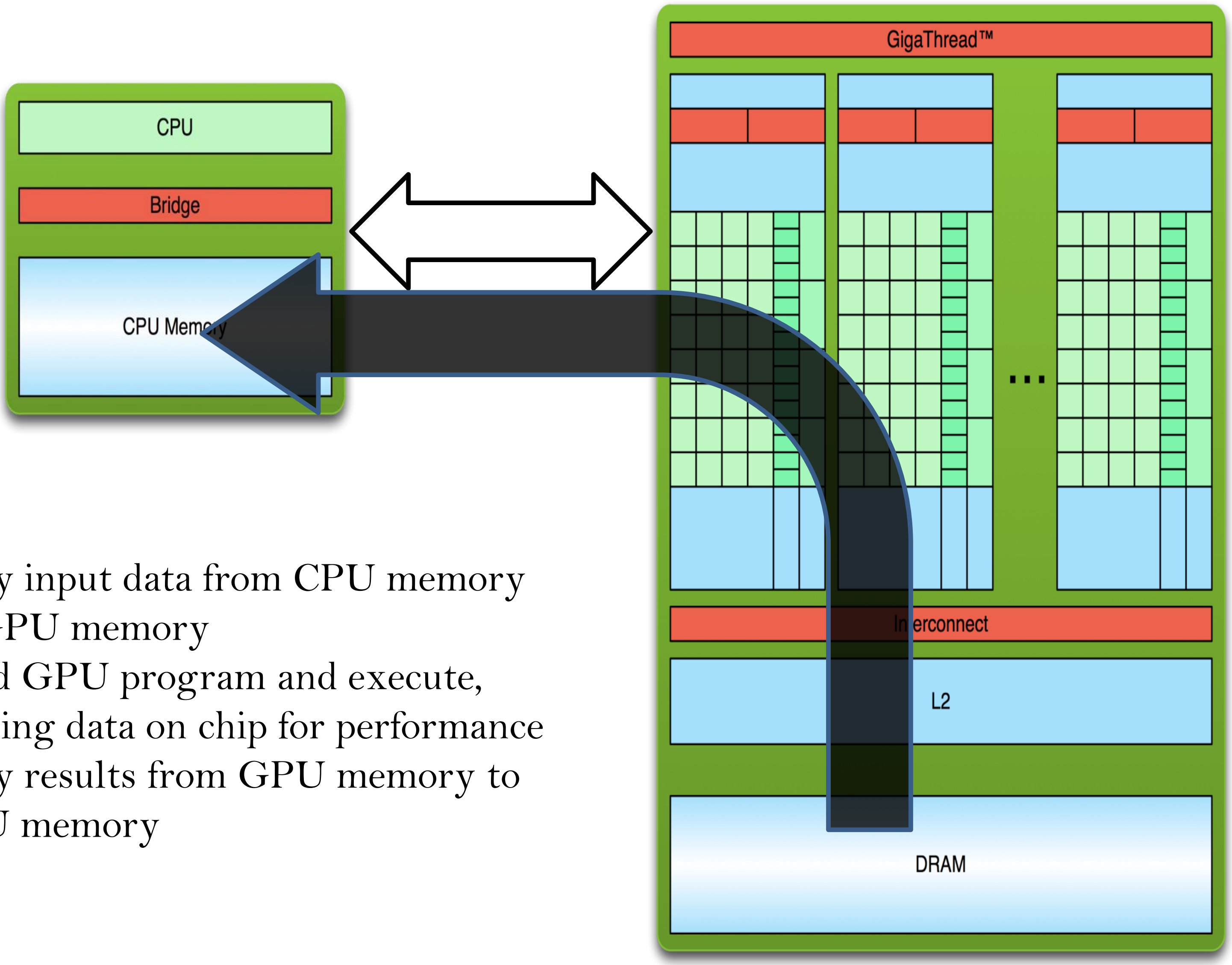
1. Copy input data from CPU memory to GPU memory

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

CUDA Basics

What is CUDA?

- 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.

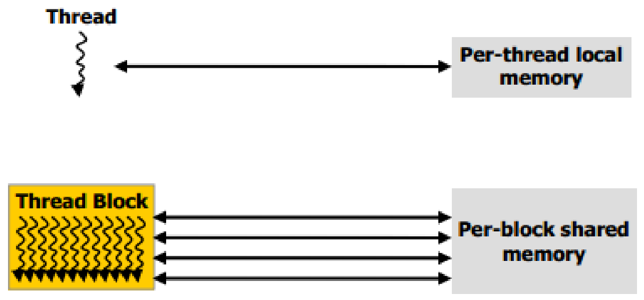
*Compute Unified Device Architecture

SPMD Phases

- Initialize
  - Establish localized data structure and communication channels
- Obtain a unique identifier
  - Each thread acquires a unique identifier, typically range from 0 to N-1, where N is the number of threads
- Distribute Data
  - Decompose global data into chunks and localize them, or
  - Sharing/replicating major data structure using thread ID to associate subset of the data to threads
- Run the core computation
- Finalize
  - Reconcile global data structure, prepare for the next major iteration

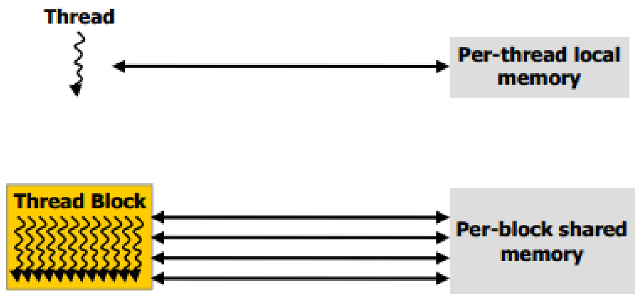
Memory Hierarchy in CUDA

- Registers/Shared memory:
  - Fast
  - Only accessible by the thread/block
  - Lifetime of the thread/block



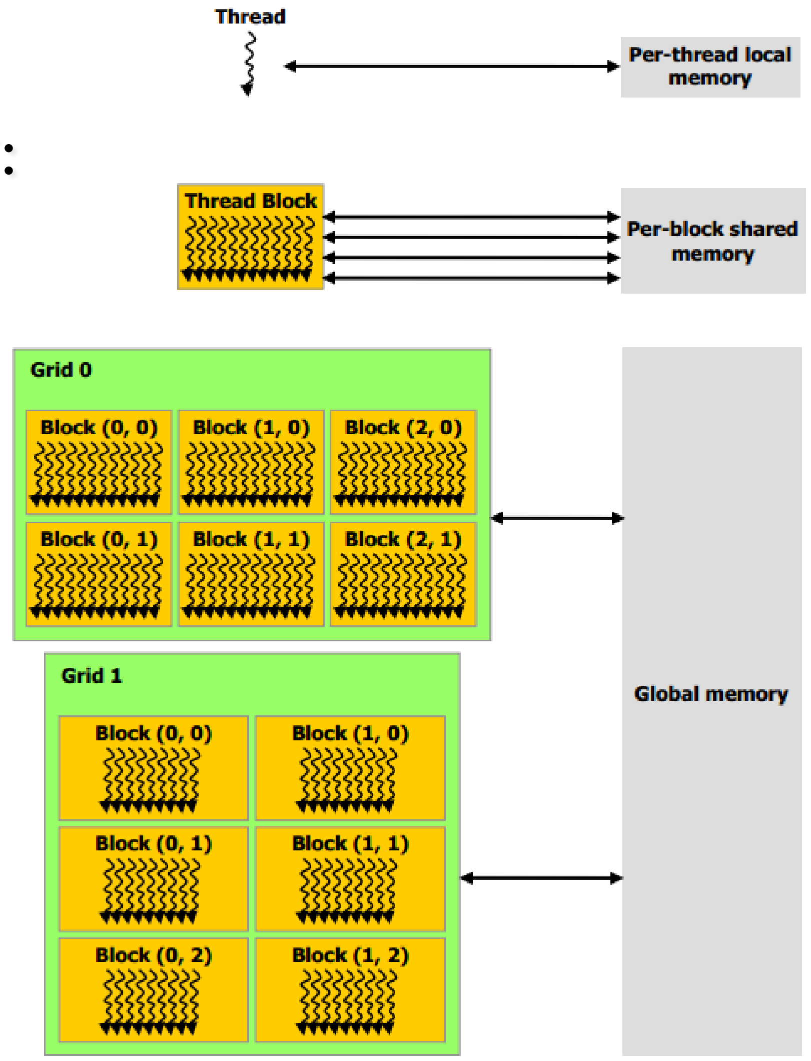
Memory Hierarchy in CUDA

- Registers/Shared memory:
  - Fast
  - Only accessible by the thread/block
  - Lifetime of the thread/block
- Global memory:



Memory Hierarchy in CUDA

- Registers/Shared memory:
  - Fast
  - Only accessible by the thread/block
  - Lifetime of the thread/block
- Global memory:
  - Potentially 150x slower than register or shared memory
  - Accessible from either the host or device
  - Lifetime of the application



Hello World!

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

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;
}
```

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) {
}
```

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

Hello World! with Device Code

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

Hello World! with Device Code

```
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;
}
```

Hello World! with Device Code

```
__global__ void mykernel(void){
}

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

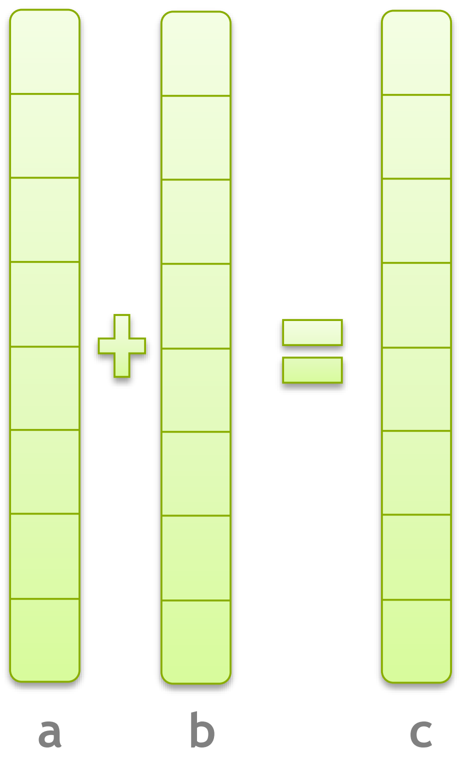
Output:

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

Parallel constructs in CUDA

Parallel Programming in CUDA

- 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;
}
```

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;
}
```

Addition on the Device

- Note that we use pointers for the variables

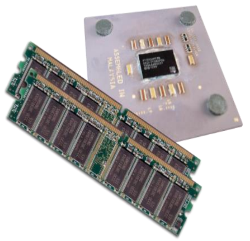
```
__global__ void add(int *a, int *b, int *c) {
      *c = *a + *b;
}
```

- `add()` runs on the device, so `a`, `b` and `c` must point to device memory

- We need to allocate memory on the GPU

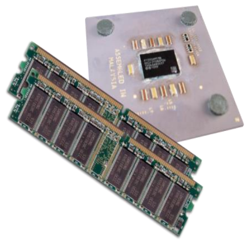
Memory Management

- 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



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()`

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
```

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);
```

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);
```

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);
```

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);
```

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;
    }
```

Moving to Parallel

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

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

Vector Addition on the Device

- With `add()` running in parallel we can do vector addition
- Terminology: each parallel invocation of `add()` is referred to as a block

Vector Addition on the Device

- With `add()` running in parallel we can do vector addition
- Terminology: 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

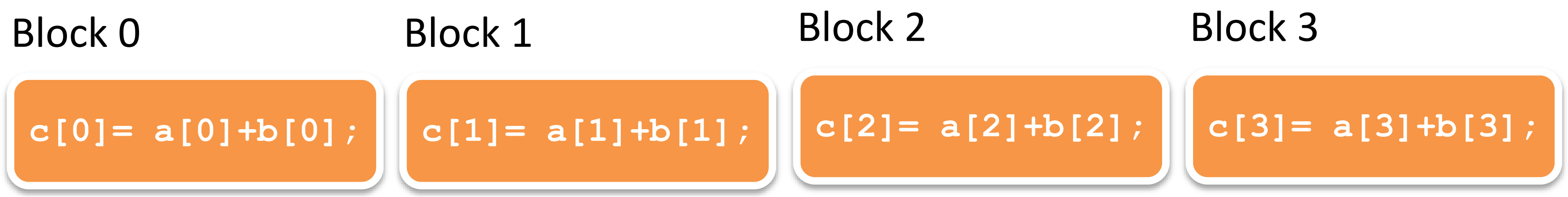
Vector Addition on the Device

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

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
```

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);
```

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
```

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:

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

Vector Addition on the Device:

```
// 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);
```

Vector Addition on the Device:

```
// 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);
```

Vector Addition on the Device:

```
        // 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;
    }
```

CUDA Threads

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

CUDA Threads

- Terminology: 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:

```
#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 <u>*and*</u> Threads

- We've seen parallel vector addition using:
  - Many blocks with one thread each
  - One block with many threads

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*

Combining Blocks *and* Threads

- We've seen parallel vector addition using:
  - Many blocks with one thread each
  - One block with many threads

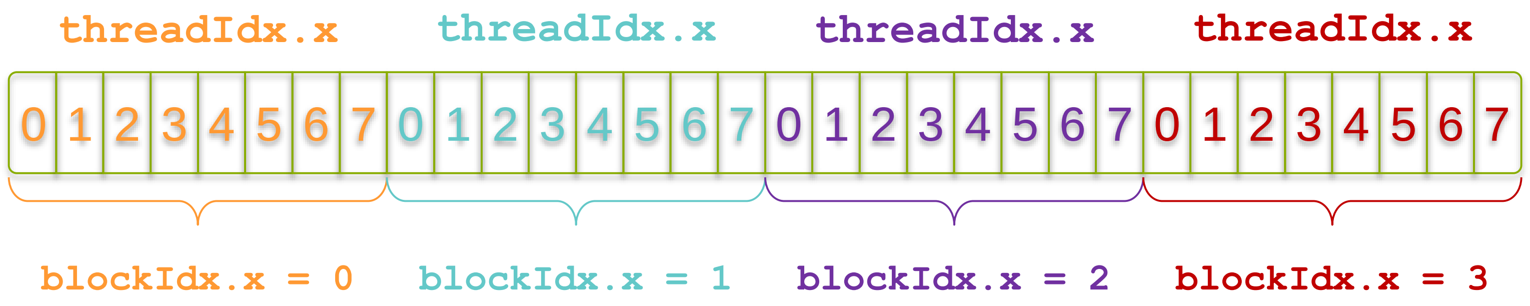
- Let's adapt vector addition to use both *blocks* and *threads*

- Why? We'll come to that…

- First let's discuss data indexing…

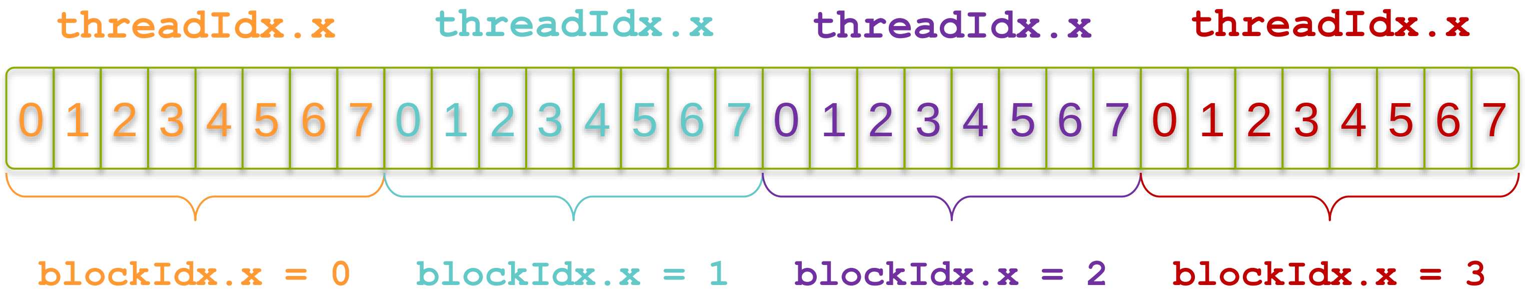
Indexing Arrays with Blocks and Threads

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



Indexing Arrays with Blocks and 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:

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

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;
```

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];
}
```

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 with Blocks and Threads:

```
#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 with Blocks and Threads:

```
    // 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`

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];
}
```

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:

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?

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

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

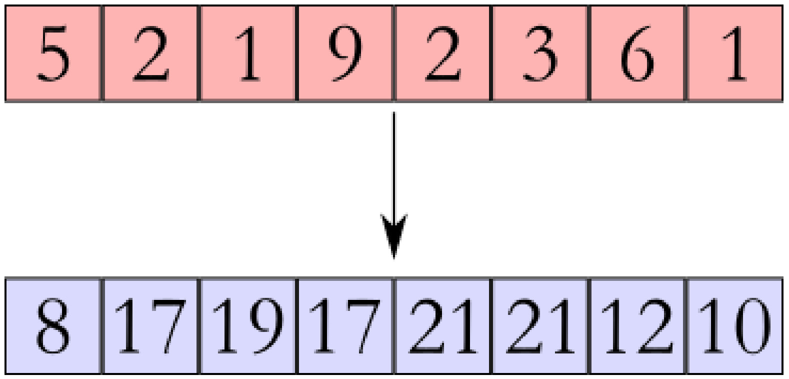
Why Bother with Threads?

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

Shared Memory

1D Stencil

- Consider applying a 1D stencil sum to a 1D array of elements
  - Each output element is the sum of input elements within a radius
  - Example of stencil with radius 2:

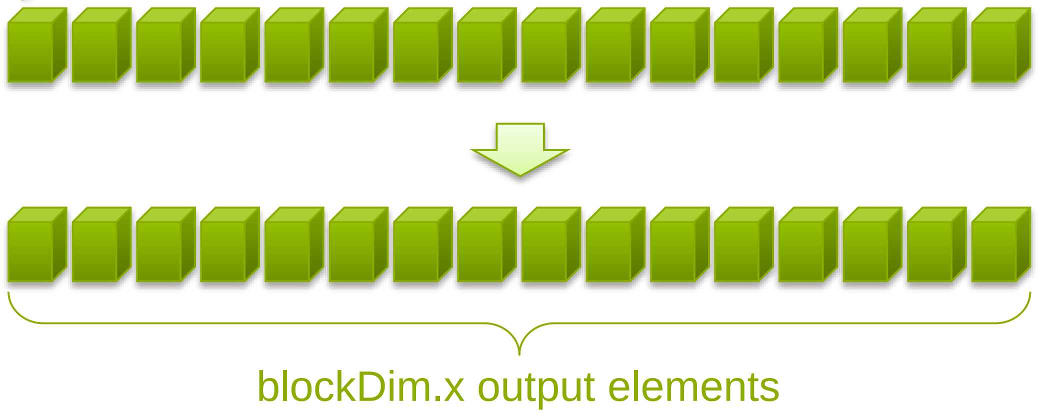


Sharing Data Between Threads

- Terminology: 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

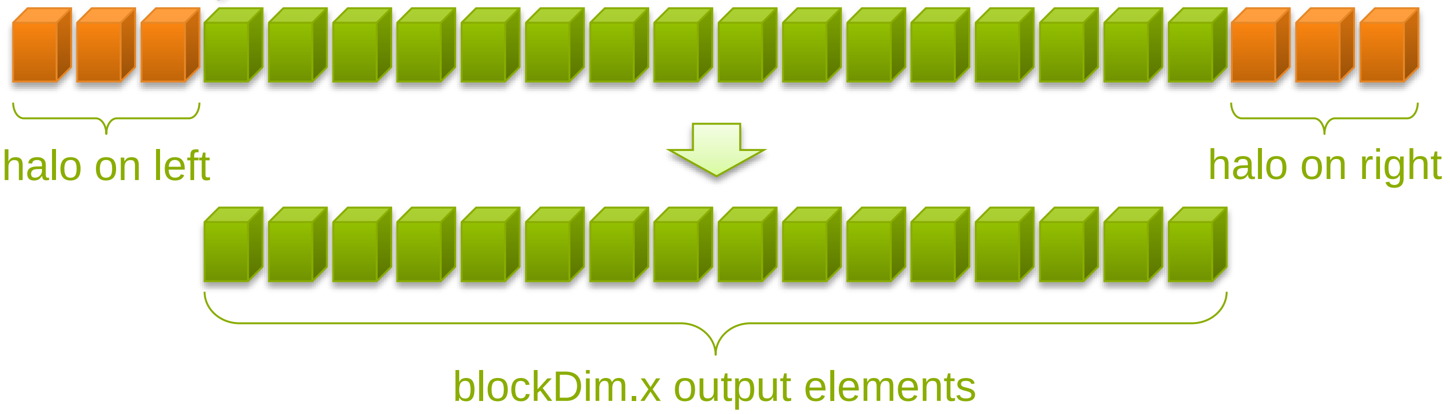
Implementing With Shared Memory

- Cache data in shared memory
  - Read `(blockDim.x + 2 * radius)` input elements from global memory to shared memory
  - Compute `blockDim.x` output elements
  - Write `blockDim.x` output elements to global memory



Implementing With Shared Memory

- Cache data in shared memory
  - Read `(blockDim.x + 2 * 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) {
```

Stencil Kernel

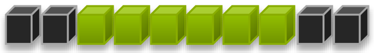
```
__global__ void stencil_1d(int *in, int *out) {
  __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
```

Stencil Kernel

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

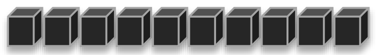
Stencil Kernel

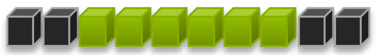
```
__global__ void stencil_1d(int *in, int *out) {
    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
    int g_index = threadIdx.x + blockIdx.x * blockDim.x;
    int s_index = threadIdx.x + RADIUS;

    // Read input elements into shared memory
    temp[s_index] = in[g_index];
```

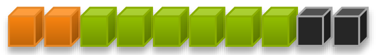


Stencil Kernel

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

  // Read input elements into shared memory
  temp[s_index] = in[g_index];
  if (threadIdx.x < RADIUS) {
    temp[s_index - RADIUS] = in[g_index - RADIUS];
```

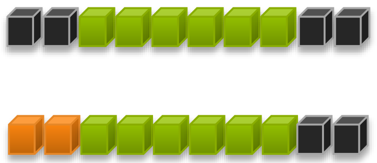






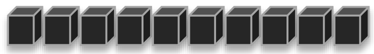
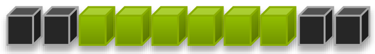
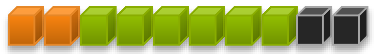
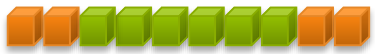
Stencil Kernel

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

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



Stencil Kernel

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

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

Stencil Kernel

```
    // Apply the stencil
    int result = 0;
    for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
      result += temp[s_index + offset];

    // Store the result
    out[g_index] = result;
  }
```

Data Race!

- The stencil example will not work…

Data Race!

- The stencil example will not work…

- Suppose thread 15 reads the halo before thread 0 has fetched it…

```
temp[s_index] = in[g_index];
 if (threadIdx.x < RADIUS) {
      temp[s_index - RADIUS = in[g_index - RADIUS];
      temp[s_index + BLOCK_SIZE] = in[g_index + BLOCK_SIZE];
 }
 int result = 0;
 result += temp[s_index + 1];
```

__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 g_index = threadIdx.x + blockIdx.x * blockDim.x;
    int s_index = threadIdx.x + radius;

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

Stencil Kernel

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

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

    // Store the result
    out[g_index] = 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 to prevent data hazards

Device Management

Compute Capability

- The compute capability of a device describes its architecture, e.g.
  - Number of registers
  - Sizes of memories
  - Features & capabilities
- By running the application `deviceQuery` in the practical part you will be able to know useful information like
  - The maximum number of threads per block
  - The amount of shared memory
  - The frequency of the DDR memory
- The compute capability is given as a major.minor version number (i.e: 2.0, 2.1, 3.0, 3.5)

Coordinating Host & Device

- Kernel launches are asynchronous
  - control is returned to the host thread before the device has completed the requested task
  - 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

Pinned memory

- Pinned (or page-locked memory) is a main memory area that is not pageable by the operating system
- Ensures faster transfers (the DMA engine can work without CPU intervention)
- The only way to get closer to PCI peak bandwidth
- Allows CUDA asynchronous operations to work correctly

```
// allocate page-locked memory
cudaMallocHost(&area, sizeof(double) * N);
// free page-locked memory
cudaFreeHost(area);
```

Asynchronous GPU Operations: CUDA Streams

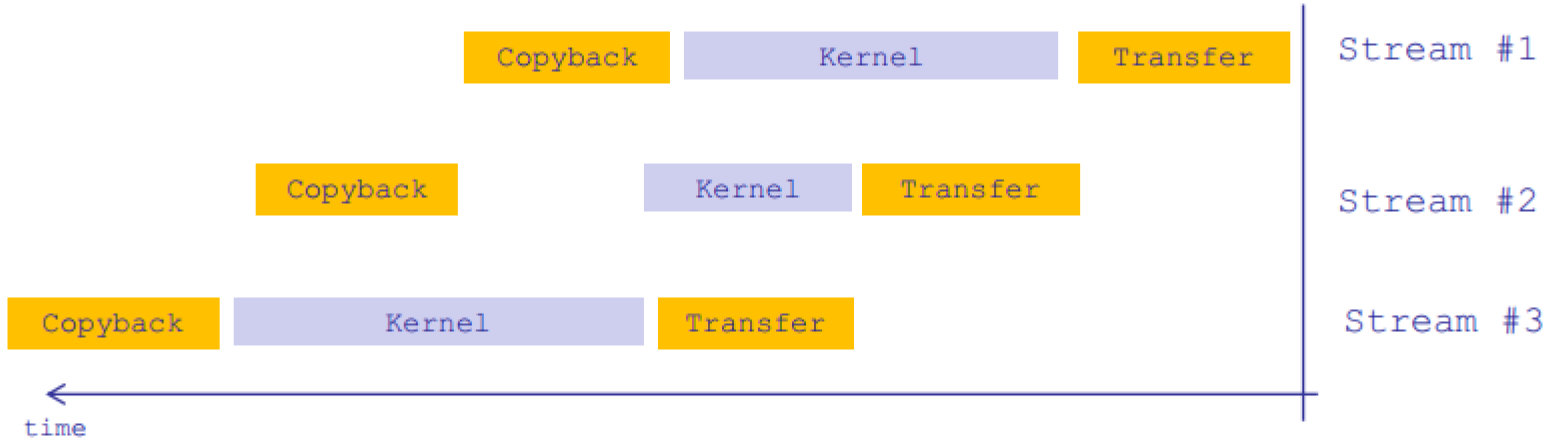
- A stream is a FIFO command queue;
  - Default stream (aka stream '0'): Kernel launches and memory copies that do not specify any stream (or set the stream to zero) are issued to the default stream.****
- A stream is independent to every other active stream;
- Streams are the main way to exploit concurrent execution and I/O operations
- Explicit Synchronization:
  - cudaDeviceSynchronize()
    - blocks host until all issued CUDA calls are complete
  - cudaStreamSynchronize(streamId)
    - blocks host until all CUDA calls in streamid are complete
  - cudaStreamWaitEvent(streamId, event)
    - all commands added to the stream delay their execution until the event has completed
- Implicit Synchronization:
  - any CUDA command to the default stream,
  - a page-locked host memory allocation,
  - a device memory set or allocation,

CUDA streams enable concurrency

- Concurrency: the ability to perform multiple CUDA operations simultaneously.
- Starting from compute capability 2.0, simultaneous support:
  - Up to 16 CUDA kernels on GPU
  - 2 cudaMemcpyAsyncs (in opposite directions)
  - Computation on the CPU
- Requirements for Concurrency:
  - CUDA operations must be in different, non-0, streams
  - cudaMemcpyAsync with host from 'pinned' memory

CUDA Streams

```
cudaStream_t stream[3];
for (int i=0; i<3; ++i) cudaStreamCreate(&stream[i]);

float* hPtr; cudaMallocHost((void**)&hPtr, 3 * size);

for (int i=0; i<3; ++i) {
  cudaMemcpyAsync(d_inp + i*size, hPtr + i*size,
                        size, cudaMemcpyHostToDevice, stream[i]);

  MyKernel<<<100, 512, 0, stream[i]>>>(d_out+i*size, d_inp+i*size, size);

  cudaMemcpyAsync(hPtr + i*size, d_out + i*size,
                        size, cudaMemcpyDeviceToHost, stream[i]);
}
cudaDeviceSynchronize();

for (int i=0; i<3; ++i) cudaStreamDestroy(&stream[i]);
```



****CUDA 7

- The default stream is useful where concurrency is not crucial to performance.
- CUDA 7 introduces a new option, the *per-thread default stream*
  - it gives each host thread its own default stream
    - commands issued to the default stream by different host threads can run concurrently
  - these default streams are regular streams
    - commands in the default stream may run concurrently with commands in non-default streams.
- Compile with the nvcc command-line option --default-stream per-thread

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)

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)
```

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()));
```

Timing

- You can use the standard timing facilities (host side) in an almost standard way…
  - but remember: CUDA calls can be asynchronous!

```
start = clock();
my_kernel<<< blocks, threads>>>();
cudaDeviceSynchronize();
end = clock();
```

Timing

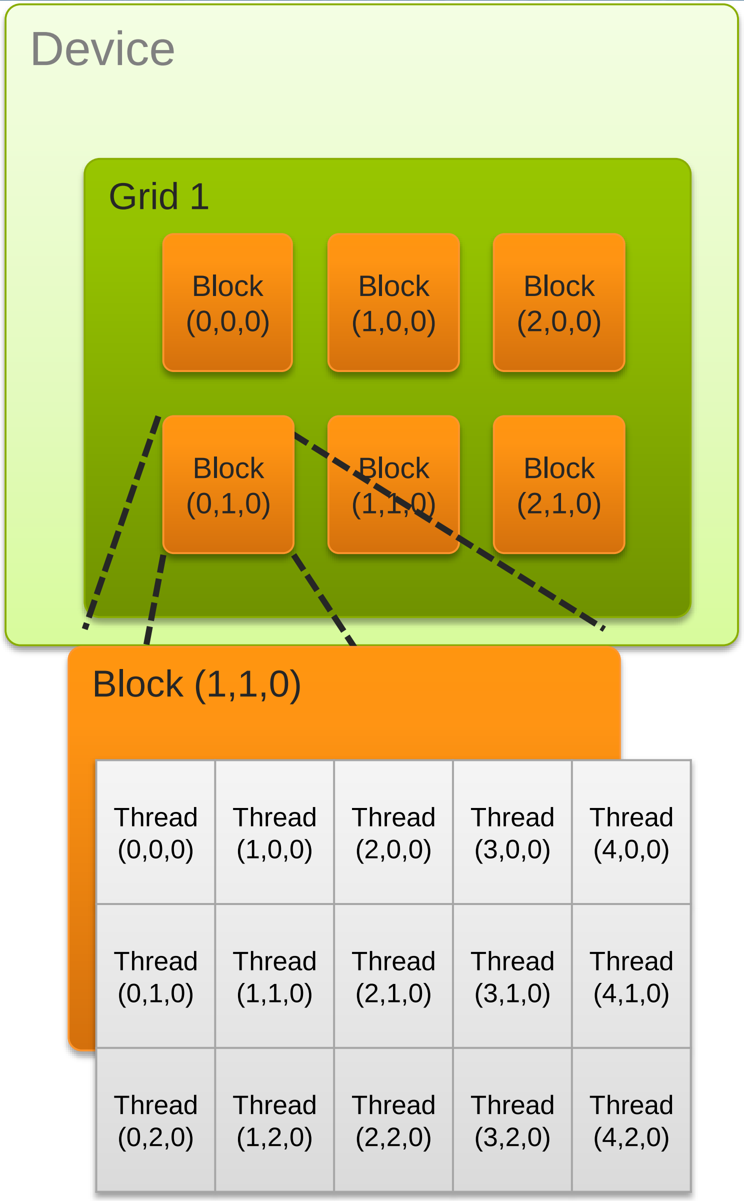
- CUDA provides the `cudaEvents` facility. They grant you access to the GPU timer.
  - Needed to time a single stream without loosing Host/Device concurrency.

```
cudaEvent_t start, stop;
cudaEventCreate(start);
cudaEventCreate(stop); cudaEventRecord(start,
0);
My_kernel<<<blocks, threads>>> ();
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
float ElapsedTime;
cudaEventElapsedTime(&elapsedTime, start,
stop);
cudaEventDestroy(start);
cudaEventDestroy(stop);
```

IDs and Dimensions

– A kernel is launched as a grid of blocks of threads
  - `blockIdx` and `threadIdx` are 3D (dim3)
  - We showed only one dimension (x)

- Built-in variables:
  – `threadIdx`
  – `blockIdx`
  – `blockDim`
  – `gridDim`

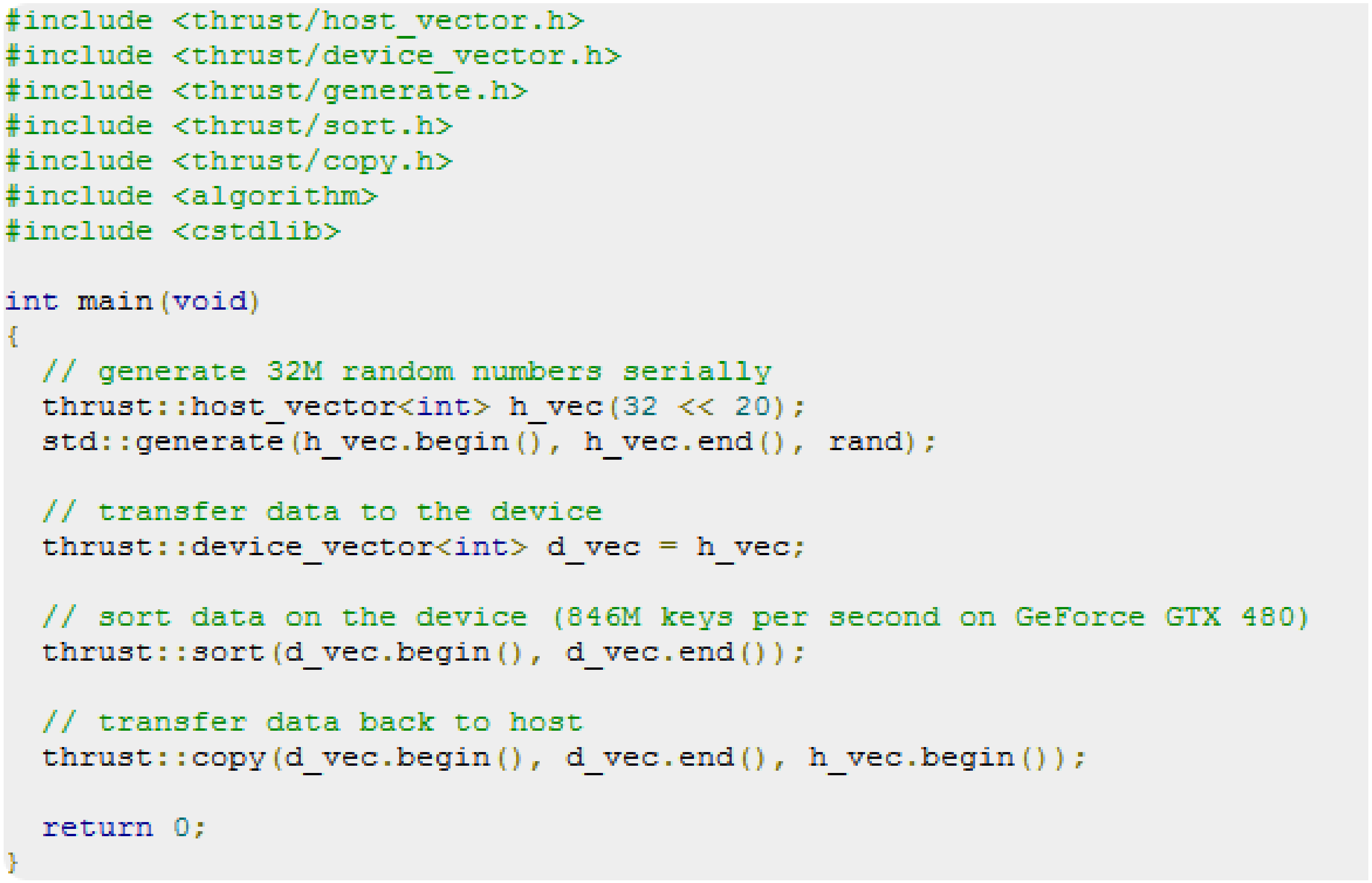


Thrust

Thrust

- Thrust is a C++ template library for CUDA based on the Standard Template Library (STL).
- Thrust allows you to implement high performance parallel applications with minimal programming effort through a high-level interface that is fully interoperable with CUDA C.
- Rich collection of data parallel primitives such as scan, sort, and reduce
  Read more at: http://docs.nvidia.com/cuda/thrust/index.html#ixzz3ypRKV2Zt

Sorting using Thrust



```
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/generate.h>
#include <thrust/sort.h>
#include <thrust/copy.h>
#include <algorithm>
#include <cstdlib>

int main(void)
{
  // generate 32M random numbers serially
  thrust::host_vector<int> h_vec(32 << 20);
  std::generate(h_vec.begin(), h_vec.end(), rand);

  // transfer data to the device
  thrust::device_vector<int> d_vec = h_vec;

  // sort data on the device (846M keys per second on GeForce GTX 480)
  thrust::sort(d_vec.begin(), d_vec.end());

  // transfer data back to host
  thrust::copy(d_vec.begin(), d_vec.end(), h_vec.begin());

  return 0;
}
```

https://thrust.github.io/

Thrust and CUDA Interoperability

- You can easily launch your custom CUDA Kernels on thrust device vectors, by passing the pointer to the first element (you can use the `thrust::device_vector::data()` method or `&vector[0]`)
- Improves readability
- Memory managed automatically
- CUDA code generated behind the scenes

```
thrust::device_vector<int> d_vector;
int* d_array= thrust::raw_pointer_cast(&d_vector[0]);

myCUDAKernel<<< x, y >>>(d_array, d_vector.size());
```

Questions?

References

- CUDA Training – NVIDIA
- Programming Massively Parallel Processors – D. B. Kirk, W. W. Hwu