



CUDA Part I

Chris Rossbach and Calvin Lin

cs380p

Outline

Over the next few classes:

Background from many areas

Architecture

Vector processors

Hardware multi-threading

Graphics

Graphics pipeline

Graphics programming models

Algorithms

parallel architectures → parallel algorithms



This
lecture

Advanced: making it perform

Acknowledgements:

http://developer.download.nvidia.com/compute/developertrainingmaterials/presentations/cuda_language/Introduction_to_CUDA_C.pptx

Review

Thread block scheduler warp (thread) scheduler



Each SM has multiple vector units (4)
32 lanes wide → warp size

Vector units use *hardware multi-threading*

Execution → a grid of thread blocks (TBs)

Each TB has some number of threads

Programming Model

GPUs are I/O devices, managed by user-code

“kernels” == “shader programs”

1000s of HW-scheduled threads per kernel

Threads grouped into independent blocks.

Threads in a block can synchronize (barrier)

This is the **only** synchronization

“Grid” == “launch” == “invocation” of a kernel

a group of blocks (or warps)

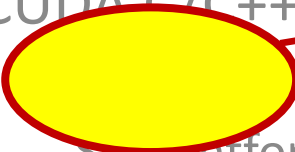
***Details of architecture are
exposed to the programmer***

CUDA

Architecture/Goals

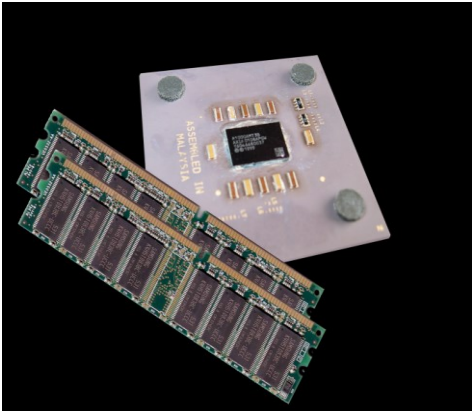
Expose GPU parallelism for general-purpose computing

Retain performance

CUDA C/C++ *Small?*
 of extensions to enable heterogeneous programming
Straightforward APIs to manage devices, memory etc.

Heterogeneous Computing

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



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);

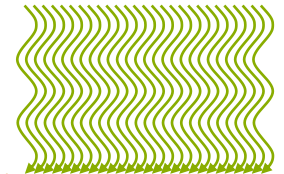
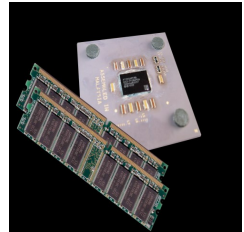
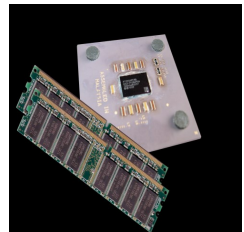
    // Cleanup
    free(in); free(out);
    cudaFree(d_in); cudaFree(d_out);
    return 0;
}
```

parallel function

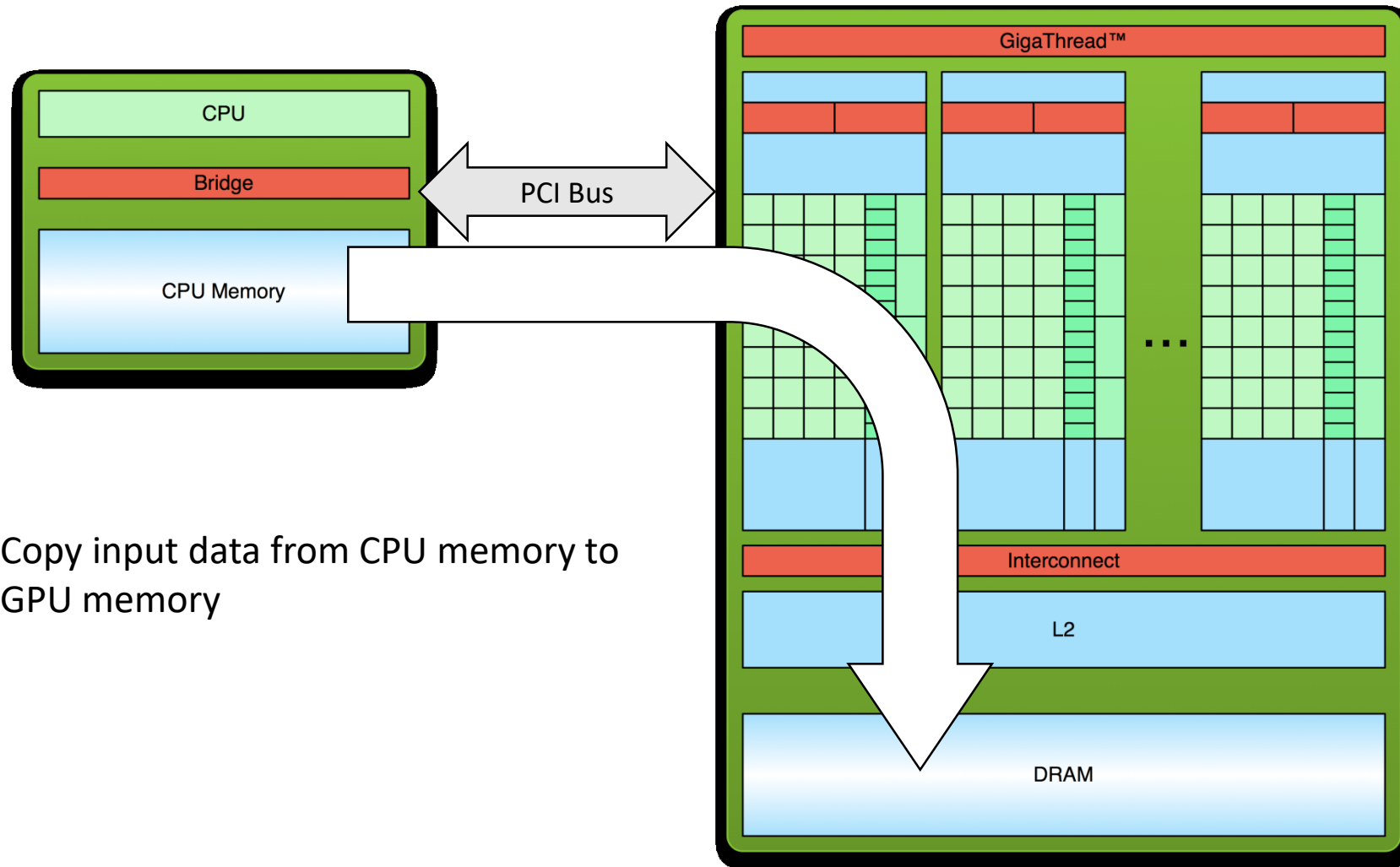
serial code

parallel code

serial code

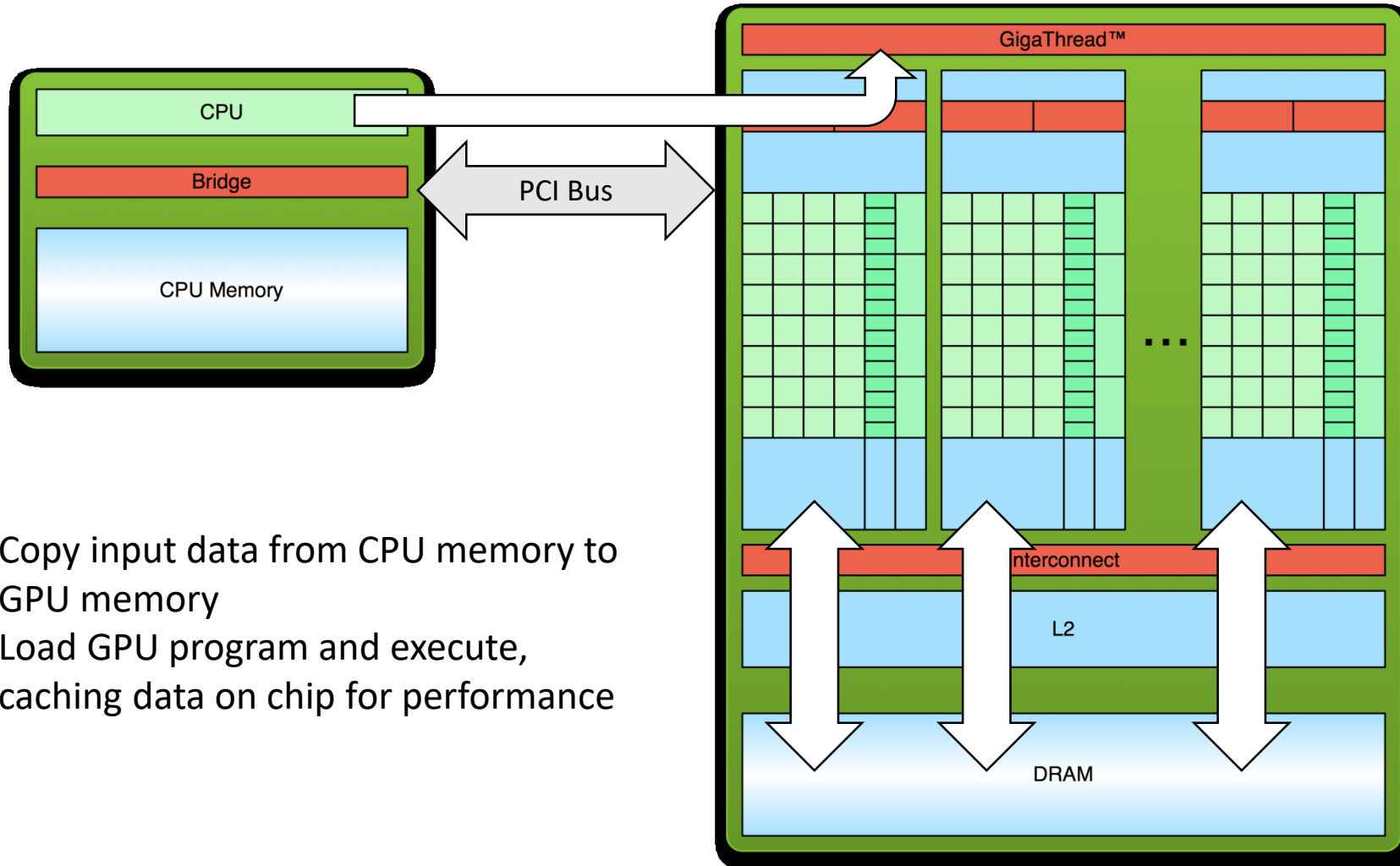


Processing Flow



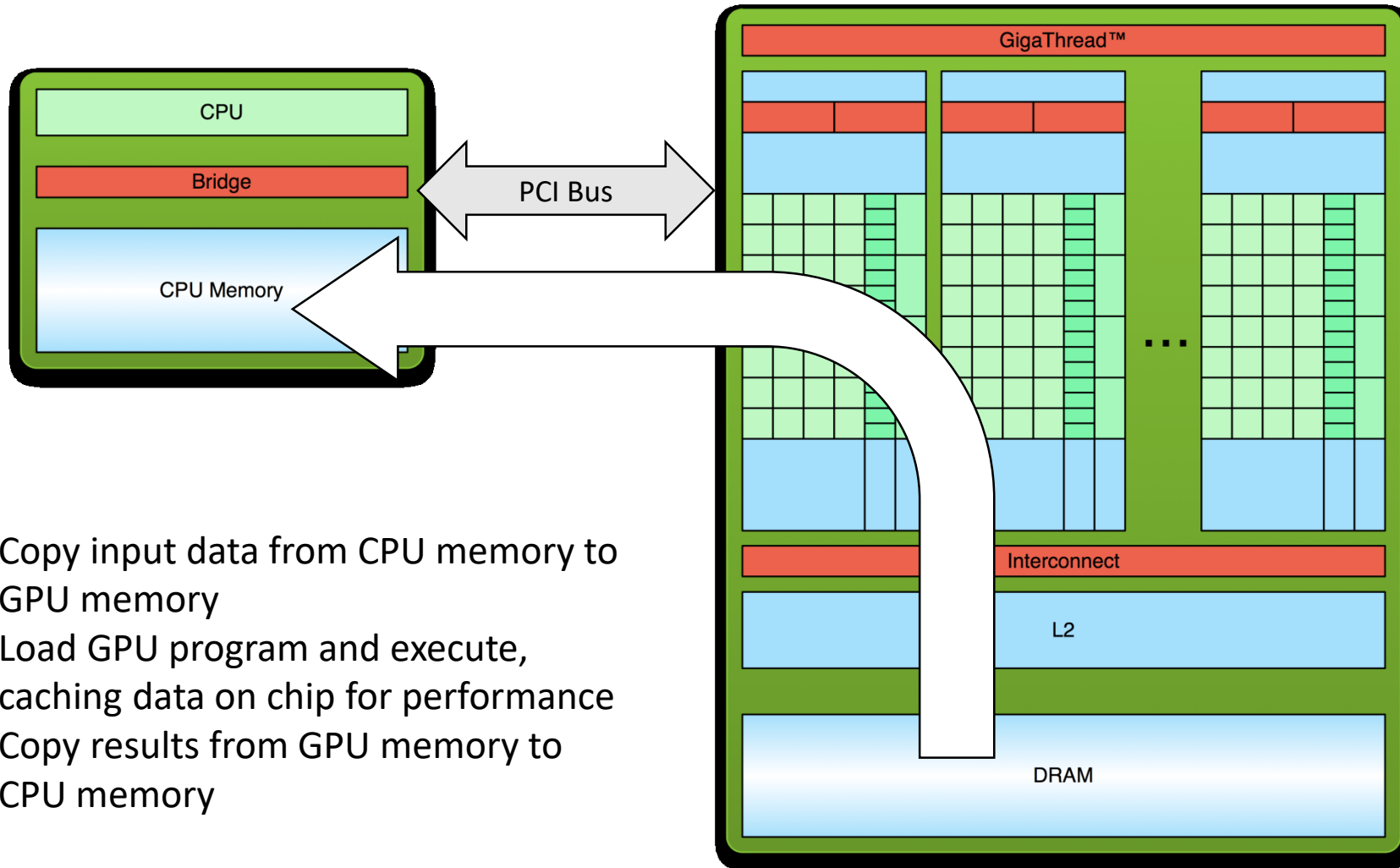
1. Copy input data from CPU memory to GPU memory

Processing Flow



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

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

Hello World

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

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

Output:

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

Hello World! with Device Code

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

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

- Two new syntactic elements...

Hello World! with Device Code

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

CUDA C/C++ keyword `__global__` indicates a function that:

- Runs on the device

- Is called from host code

`nvcc` separates source code into host and device components

- Device functions (e.g. `mykernel()`) processed by NVIDIA compiler

- Host functions (e.g. `main()`) processed by standard host compiler

 - `gcc, cl.exe`

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”

What do parameters <<<1,1>>> mean?

stay tuned

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;  
}
```

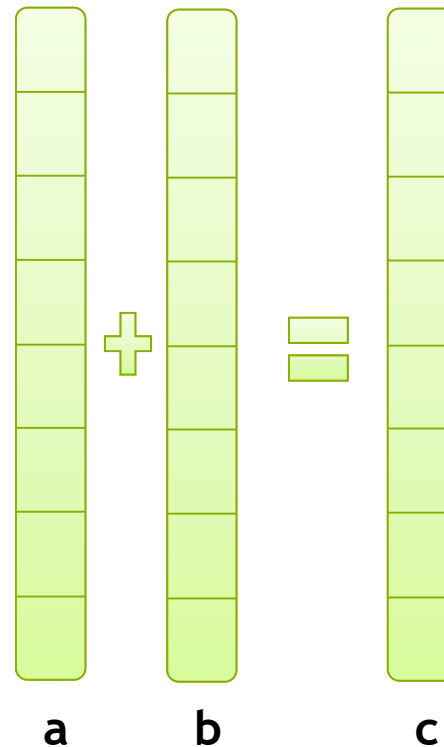
- `mykernel()` does nothing

Output:

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

Parallel Programming in CUDA C/C++

- But wait... GPUs are massively parallel!
- We need a more interesting example...
- Start with integers addition
 - 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;  
}
```

Same as before `__global__` → CUDA C/C++ keyword:

`add()` will *execute* on the device

`add()` will be *called* from the host

Addition on the Device

Note that we use pointers for the variables

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

`add()` runs on the device

so `a`, `b` and `c` must point to device memory

Must 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

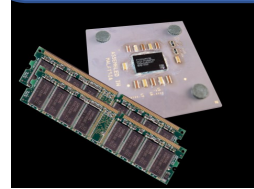
Host pointers point to CPU memory

May be passed to/from device code

May *not* be dereferenced in device code

Truth in advertising:

- This is changing (UVM)
- More on this later



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

Addition on the Device: `main()`

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

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

// Copy result back to host
cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);
// Cleanup
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
return 0;
```

```
}
```

Getting 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

Terminology: each parallel invocation of `add()` is a **block**

The set of blocks is referred to as a **grid**

Each invocation can refer to its block index using **`blockIdx.x`**

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

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

Vector Addition on the Device

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

On the device, each block can execute in parallel:

Block 0

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

Block 1

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

Block 2

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

Block 3

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

Vector Addition on the Device: `add()`

Returning to our parallelized `add()` kernel

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

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

Vector Addition on the Device: `main()`

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

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

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

Vector Addition on the Device: `main()`

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

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

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

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

CUDA Threads

- Terminology: a block can be split into parallel **threads**
- 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];  
}
```

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

Vector Addition Using Threads: `main()`

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

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

    // Alloc space for host copies of a, b, c and setup 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);
```

Traditional CPU	Graphics Shaders	CUDA	OpenCL
SIMD lane	thread	thread	work-item
~thread	-	warp	-
	thread group	block	work group
	-	grid	N-D range

Combining Blocks and Threads

We've seen parallel vector addition using:

Many blocks with *one thread* each (M:1)

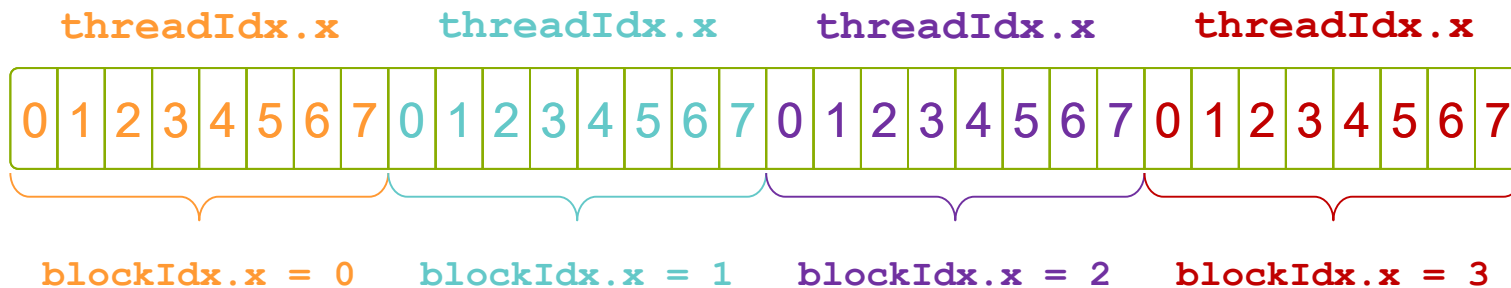
One block with *many threads* (1:M)

How to make vector addition to use both blocks and threads?

How to deal with `blockIdx.*` vs `threadIdx.*`?

Indexing Arrays with Blocks and Threads

- Most kernels use **both** `blockIdx.x` and `threadIdx.x`
 - Index an array with one elem. per thread (8 threads/block)



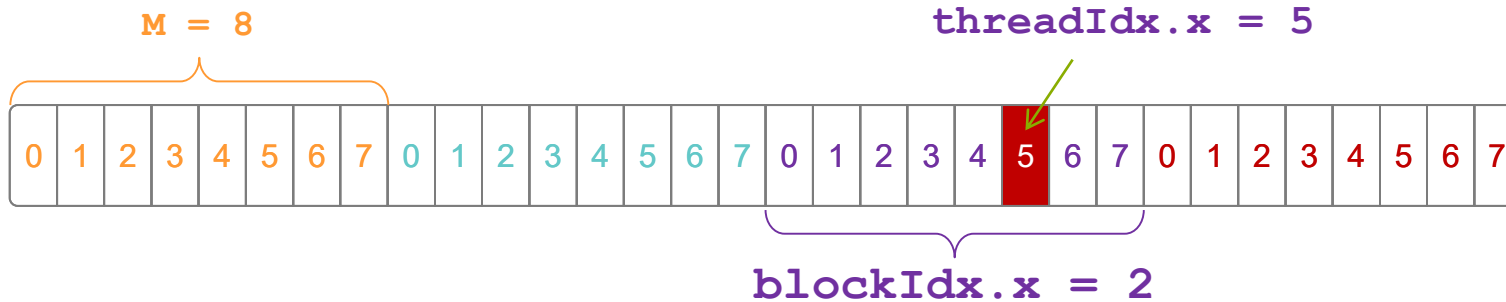
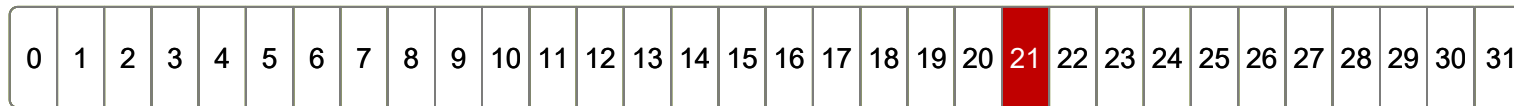
- With M threads/block, unique index per thread is :

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

Indexing Arrays: Example

Which thread will operate on the red element?

M=8 Threads, 4 blocks



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

Vector Addition with Blocks and Threads

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

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

- Combined `add()` using parallel threads *and* 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

```
main()
```

```
#define N (2048*2048)
#define THREADS_PER_BLOCK 512
int main(void) {
    int *a, *b, *c;           // host copies of a, b, c
    int *d_a, *d_b, *d_c;    // device copies of a, b, c
    int size = N * sizeof(int);

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

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

```
main()
```

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

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

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

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

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

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

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

```
// Cleanup
```

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

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

```
return 0;
```

```
}
```

Anyone see a
problem?

Handling Arbitrary Vector Sizes

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

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

- Update the kernel launch:

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

Why Bother with Threads?

Threads seem unnecessary

They add a level of complexity

What do we gain?

Unlike parallel blocks, threads have mechanisms to:

Communicate

Synchronize

To look closer, we need a new example...stay tuned

Summary + Review

Heterogeneous Computing
BSP-like Programming Model
Host and Device Code

Launching parallel kernels

Launch **N** copies of **add ()** with **add<<<N/M,M>>> (...)** ;

Use **blockIdx.x** to access block index

Use **threadIdx.x** to access thread index within block

Allocate elements to threads:

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