Introduction to CUDA
UNSW/NCI Training Week

Stephen Sanderson

National Computational Infrastructure, Australia

Acknowledgement of Country

The National Computational Infrastructure acknowledges, celebrates and pays our respects to the Ngunnawal and Ngambri people of the Canberra region and to all First Nations Australians on whose traditional lands we meet and work, and whose cultures are among the oldest continuing cultures in human history.

Outline

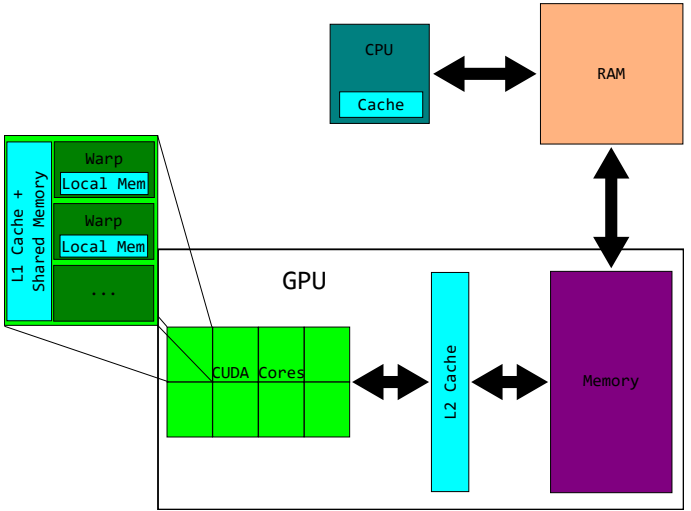
CUDA is an extension to the C, C++ and Fortran languages that allows general purpose computing on GPUs. This can give significant speed up for some problems that benefit from massive parallelism.

This short course presents an introduction to GPU programming in CUDA C.

It covers:

- Mental model of a GPU
- Core concepts of CUDA
- Syntax of CUDA C
- Basic kernel implementation
- Overlapping compute and data transfer
- Modular compilation
- Key performance considerations

Your mental model of a GPU



Hello, world!

```
#include <stdio.h>

__global__
void hello() {
  printf("Hello, world!\n");
}

int main(void) {
  hello<<<1,64>>>();
  cudaDeviceSynchronize();
  return 0;
}
```

Compile and run with:

```
$ nvcc main.cu -o hello
$ ./hello
```

Hello, world!

```c
#include <stdio.h>

__global__
void hello() {
  printf("Hello, world!\n");
}

int main(void) {
  hello<<<1,64>>>();
  cudaDeviceSynchronize();
  return 0;
}
```

Compile and run with:

```
$ nvcc main.cu -o hello
$ ./hello
```

Prints "Hello, world!" 64 times.

SIMT

GPUs operate under a single instruction, multiple thread (SIMT) model.

- 32 threads (1 warp) all execute the same instruction at the same time.
- Branching (e.g. `if` statements) can cause some threads to sit idle, particularly on pre-Volta architectures.
- Multiple warps within one *block* can synchronize with each other by calling `__syncthreads()`.
- Not mandatory, but best to use block sizes that are multiples of 32.
- Different blocks cannot synchronize—there could be more blocks than launched than available threads!

	Block 0							
Instruction	**Warp 0**				**Warp 1**			
`int a = 10;`								
`printf("Hi\n");`								
`__syncthreads();`								
`printf("Bye\n");`								

SIMT

GPUs operate under a single instruction, multiple thread (SIMT) model.

- 32 threads (1 warp) all execute the same instruction at the same time.
- Branching (e.g. `if` statements) can cause some threads to sit idle, particularly on pre-Volta architectures.
- Multiple warps within one *block* can synchronize with each other by calling `__syncthreads()`.
- Not mandatory, but best to use block sizes that are multiples of 32.
- Different blocks cannot synchronize—there could be more blocks than launched than available threads!

	Block 0							
Instruction	**Warp 0**				**Warp 1**			
`int a = 10;`	x	x	x	x	x	x	x	x
`printf("Hi\n");`								
`__syncthreads();`								
`printf("Bye\n");`								

SIMT

GPUs operate under a single instruction, multiple thread (SIMT) model.

- 32 threads (1 warp) all execute the same instruction at the same time.
- Branching (e.g. `if` statements) can cause some threads to sit idle, particularly on pre-Volta architectures.
- Multiple warps within one *block* can synchronize with each other by calling `__syncthreads()`.
- Not mandatory, but best to use block sizes that are multiples of 32.
- Different blocks cannot synchronize—there could be more blocks than launched than available threads!

	Block 0							
Instruction	**Warp 0**				**Warp 1**			
`int a = 10;`	x	x	x	x	x	x	x	x
`printf("Hi\n");`	x	x	x	x				
`__syncthreads();`								
`printf("Bye\n");`								

SIMT

GPUs operate under a single instruction, multiple thread (SIMT) model.

- 32 threads (1 warp) all execute the same instruction at the same time.
- Branching (e.g. `if` statements) can cause some threads to sit idle, particularly on pre-Volta architectures.
- Multiple warps within one *block* can synchronize with each other by calling `__syncthreads()`.
- Not mandatory, but best to use block sizes that are multiples of 32.
- Different blocks cannot synchronize—there could be more blocks than launched than available threads!

	Block 0							
Instruction	**Warp 0**				**Warp 1**			
`int a = 10;`	x	x	x	x	x	x	x	x
`printf("Hi\n");`	x	x	x	x	x	x	x	x
`__syncthreads();`								
`printf("Bye\n");`								

SIMT

GPUs operate under a single instruction, multiple thread (SIMT) model.

- 32 threads (1 warp) all execute the same instruction at the same time.
- Branching (e.g. `if` statements) can cause some threads to sit idle, particularly on pre-Volta architectures.
- Multiple warps within one *block* can synchronize with each other by calling `__syncthreads()`.
- Not mandatory, but best to use block sizes that are multiples of 32.
- Different blocks cannot synchronize—there could be more blocks than launched than available threads!

	Block 0							
Instruction	**Warp 0**				**Warp 1**			
`int a = 10;`	X	X	X	X	X	X	X	X
`printf("Hi\n");`	X	X	X	X	X	X	X	X
`__syncthreads();`	X	X	X	X	X	X	X	X
`printf("Bye\n");`								

SIMT

GPUs operate under a single instruction, multiple thread (SIMT) model.

- 32 threads (1 warp) all execute the same instruction at the same time.
- Branching (e.g. `if` statements) can cause some threads to sit idle, particularly on pre-Volta architectures.
- Multiple warps within one *block* can synchronize with each other by calling `__syncthreads()`.
- Not mandatory, but best to use block sizes that are multiples of 32.
- Different blocks cannot synchronize—there could be more blocks than launched than available threads!

	Block 0							
Instruction	**Warp 0**				**Warp 1**			
`int a = 10;`	X	X	X	X	X	X	X	X
`printf("Hi\n");`	X	X	X	X	X	X	X	X
`__syncthreads();`	X	X	X	X	X	X	X	X
`printf("Bye\n");`	X	X	X	X				

SIMT

GPUs operate under a single instruction, multiple thread (SIMT) model.

- 32 threads (1 warp) all execute the same instruction at the same time.
- Branching (e.g. `if` statements) can cause some threads to sit idle, particularly on pre-Volta architectures.
- Multiple warps within one *block* can synchronize with each other by calling `__syncthreads()`.
- Not mandatory, but best to use block sizes that are multiples of 32.
- Different blocks cannot synchronize—there could be more blocks than launched than available threads!

	Block 0							
Instruction	**Warp 0**				**Warp 1**			
`int a = 10;`	X	X	X	X	X	X	X	X
`printf("Hi\n");`	X	X	X	X	X	X	X	X
`__syncthreads();`	X	X	X	X	X	X	X	X
`printf("Bye\n");`	X	X	X	X	X	X	X	X

Divergent Warps

	Block 0							
Instruction	**Warp 0**				**Warp 1**			
`int i = threadIdx.x;`								
`if (i % 2 == 0) printf("Hi\n");`								
`else printf("Bye\n");`								
`printf("Goodnight\n");`								

Divergent Warps

	Block 0							
Instruction	**Warp 0**				**Warp 1**			
`int i = threadIdx.x;`	X	X	X	X	X	X	X	X
`if (i % 2 == 0) printf("Hi\n");`								
`else printf("Bye\n");`								
`printf("Goodnight\n");`								

Divergent Warps

	Block 0							
Instruction	**Warp 0**				**Warp 1**			
`int i = threadIdx.x;`	x	x	x	x	x	x	x	x
`if (i % 2 == 0) printf("Hi\n");`	x	-	x	-	x	-	x	-
`else printf("Bye\n");`								
`printf("Goodnight\n");`								

Divergent Warps

	Block 0							
Instruction	**Warp 0**				**Warp 1**			
`int i = threadIdx.x;`	x	x	x	x	x	x	x	x
`if (i % 2 == 0) printf("Hi\n");`	x	-	x	-	x	-	x	-
`else printf("Bye\n");`	-	x	-	x	-	x	-	x
`printf("Goodnight\n");`								

Divergent Warps

	Block 0							
Instruction	**Warp 0**				**Warp 1**			
`int i = threadIdx.x;`	x	x	x	x	x	x	x	x
`if (i % 2 == 0) printf("Hi\n");`	x	-	x	-	x	-	x	-
`else printf("Bye\n");`	-	x	-	x	-	x	-	x
`printf("Goodnight\n");`	x	x	x	x	x	x	x	x

Avoiding Divergence

Planning division of work around expected branching can help avoid divergent warps and give better performance.

This is less important for Volta or newer GPUs, but in that case keep in mind that threads within a warp are no longer guaranteed to be synchronised.

	Block 0							
Instruction	**Warp 0**				**Warp 1**			
`int i = threadIdx.x;`								
`if (i < 4) printf("Hi\n");`								
`else printf("Bye\n");`								
`printf("Goodnight\n");`								

Avoiding Divergence

Planning division of work around expected branching can help avoid divergent warps and give better performance.

This is less important for Volta or newer GPUs, but in that case keep in mind that threads within a warp are no longer guaranteed to be synchronised.

	Block 0							
Instruction	**Warp 0**				**Warp 1**			
`int i = threadIdx.x;`	x	x	x	x	x	x	x	x
`if (i < 4) printf("Hi\n");`								
`else printf("Bye\n");`								
`printf("Goodnight\n");`								

Avoiding Divergence

Planning division of work around expected branching can help avoid divergent warps and give better performance.

This is less important for Volta or newer GPUs, but in that case keep in mind that threads within a warp are no longer guaranteed to be synchronised.

	Block 0							
Instruction	**Warp 0**				**Warp 1**			
`int i = threadIdx.x;`	x	x	x	x	x	x	x	x
`if (i < 4) printf("Hi\n");`	x	x	x	x	-	-	-	-
`else printf("Bye\n");`	-	-	-	-	x	x	x	x
`printf("Goodnight\n");`								

Avoiding Divergence

Planning division of work around expected branching can help avoid divergent warps and give better performance.

This is less important for Volta or newer GPUs, but in that case keep in mind that threads within a warp are no longer guaranteed to be synchronised.

	Block 0							
Instruction	**Warp 0**				**Warp 1**			
`int i = threadIdx.x;`	x	x	x	x	x	x	x	x
`if (i < 4) printf("Hi\n");`	x	x	x	x	-	-	-	-
`else printf("Bye\n");`	-	-	-	-	x	x	x	x
`printf("Goodnight\n");`	x	x	x	x	x	x	x	x

Launching kernels

When launching a GPU kernel, we specify the dimensions of the *grid* on which it will execute, and the size of each *block* in that grid.

For example, `some_func<<<10,32>>>();` calls the kernel `some_func` on a 1D grid of ten 1D blocks, where each block contains 32 threads.

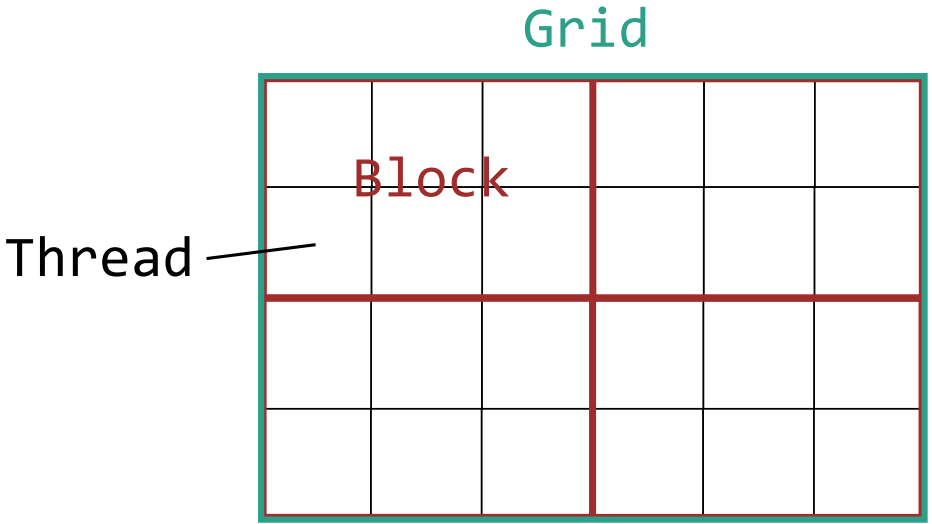
Launching kernels

When launching a GPU kernel, we specify the dimensions of the *grid* on which it will execute, and the size of each *block* in that grid.

For example, `some_func<<<10,32>>>();` calls the kernel `some_func` on a 1D grid of ten 1D blocks, where each block contains 32 threads.

This expands to up to 3 dimensions!

Launching kernels



Launching kernels

CUDA includes intrinsics for 2D, 3D, and 4D datatypes, including **`int2`**, **`int3`**, **`int4`**, **`float2`**, **`float3`**, **`float4`**, **`double2`**, ...

These have up to 4 members, named `.x`, `.y`, `.z`, and `.w`

Launching kernels

CUDA includes intrinsics for 2D, 3D, and 4D datatypes, including **int2**, **int3**, **int4**, **float2**, **float3**, **float4**, **double2**, ...

These have up to 4 members, named `.x`, `.y`, `.z`, and `.w`

The datatype for specifying grid and block sizes is **dim3**.
This can be constructed in a few ways. For example:

```
dim3 xyz = {.x = 10, .y = 20, .z = 2};
dim3 xyz = {10, 20, 2};
dim3 xyz = dim3(10, 20, 2);
```

Launching kernels

CUDA includes intrinsics for 2D, 3D, and 4D datatypes, including **int2**, **int3**, **int4**, **float2**, **float3**, **float4**, **double2**, ...

These have up to 4 members, named `.x`, `.y`, `.z`, and `.w`

The datatype for specifying grid and block sizes is **dim3**.
This can be constructed in a few ways. For example:

```
dim3 xyz = {.x = 10, .y = 20, .z = 2};
dim3 xyz = {10, 20, 2};
dim3 xyz = dim3(10, 20, 2);
```

For example, when launching launching kernels:

```
// Run kernel over 10*20*2 * 4*4*4 = 25,600 threads
some_func<<<dim3(10,20,2), dim3(4,4,4)>>>();
some_func<<<dim3(10,20), dim3(4,4)>>>(); // Dims can be left out
```

Launching kernels

```
some_func<<<dim3(10,20,2), dim3(4,4,4)>>>();
```

Within `some_func`, we can access the first argument (the grid size) via the variable `gridDim`, and the 2nd argument (the block size) via `blockDim`.

Launching kernels

```
some_func<<<dim3(10,20,2), dim3(4,4,4)>>>();
```

Within `some_func`, we can access the first argument (the grid size) via the variable `gridDim`, and the 2nd argument (the block size) via `blockDim`.

The unique index of a given thread is specified by the combination of `blockIdx` and `threadIdx`.

```
__global__ void some_func() {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    int idy = blockIdx.y * blockDim.y + threadIdx.y;
    int idz = blockIdx.z * blockDim.z + threadIdx.z;
    int tid = idx + idy * gridDim.x + idz * gridDim.x * gridDim.y;
    printf("Unique thread id: %i @ {%i, %i, %i} of {%i, %i, %i}\n",
           tid, idx, idy, idz, gridDim.x, gridDim.y, gridDim.z);
}
```

Choosing block size

Block size can have a significant impact on performance. 256 threads per block is a common choice, but it's worth testing for particular algorithms, and for particular hardware.

The size of a block is limited to 1024 threads, and the z dimension of the block is limited to 64 threads.

The overall grid size (total number of threads launched) is limited to $2^{31} - 1$ in the x dimension, and 65535 in the y and z dimensions.

There are a number of other hardware limitations dependent on the *compute capability* of the GPU, which you can look up here: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#features-and-technical-specifications

Choosing block size

For easy tweaking of the block size, it can be convenient to use a macro:

```
// integer division rounding up
#define NBLOCKS(N, BLOCK_SIZE) (((N) + (BLOCK_SIZE) - 1)/(BLOCK_SIZE))
// example:
kernel<<<NBLOCKS(N, 1024),1024>>>(some_array_d, N);

// Helper for typical default block size
#define DEF_BLK 256
#define DEF_GRID(N) NBLOCKS((N), DEF_BLK)
// example:
kernel<<<DEF_GRID(N), DEF_BLK>>>(some_array_d, N);
```

Choosing block size

For easy tweaking of the block size, it can be convenient to use a macro:

```
// integer division rounding up
#define NBLOCKS(N, BLOCK_SIZE) (((N) + (BLOCK_SIZE) - 1)/(BLOCK_SIZE))
// example:
kernel<<<NBLOCKS(N, 1024),1024>>>(some_array_d, N);

// Helper for typical default block size
#define DEF_BLK 256
#define DEF_GRID(N) NBLOCKS((N), DEF_BLK)
// example:
kernel<<<DEF_GRID(N), DEF_BLK>>>(some_array_d, N);
```

Keep in mind that since more threads may be launched than elements in the data, the kernel should check whether a thread has a valid index in the array:

```
__global__ void kernel(int* data, const int N) {
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    if (idx >= N) return;
    // ...
```

Launching kernels

Questions about launching kernels?

Managing Memory

Unlike a standard C program, we now have two separate memory address spaces to worry about.

- One for the CPU (the host).
  - Allocate with `ptr = malloc(BYTES);`
  - Deallocate with `free(ptr);`
- One for the GPU (the device).
  - Allocate with `cudaMalloc((void**)&ptr_d, BYTES);`
  - Deallocate with `cudaFree(ptr_d);`

Managing Memory

Unlike a standard C program, we now have two separate memory address spaces to worry about.

- One for the CPU (the host).
  - Allocate with `ptr = malloc(BYTES);`
  - Deallocate with `free(ptr);`
- One for the GPU (the device).
  - Allocate with `cudaMalloc((void**)&ptr_d, BYTES);`
  - Deallocate with `cudaFree(ptr_d);`

Example: Let's create an array of numbers from 0 to N-1

Example: Array of numbers 0 to N-1

In C, we would do this as:

```c
int main(void) {
    const int N = 1024;
    int* arr = malloc(sizeof(*arr) * N);
    for (int i = 0; i < N; ++i)
        arr[i] = i;
    /* Do stuff with arr */
    free(arr);
}
```

Example: Array of numbers 0 to N-1

In CUDA, we would do this as:

```
__global__ void kernel(const int N);

int main(void) {
    const int N = 1024;
    int* arr_d;
    cudaMalloc((void**)&arr_d, sizeof(*arr_d) * N);
    kernel<<<4,256>>>(N, arr_d);
    /* Do stuff with arr_d */
    cudaFree(arr);
}

__global__ void kernel(const int N, int* arr) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    arr[i] = i;
}
```

Managing Memory

What can we do with `arr_d`?

Managing Memory

What can we do with `arr_d`?
Accessing device memory directly on the host is undefined behaviour!

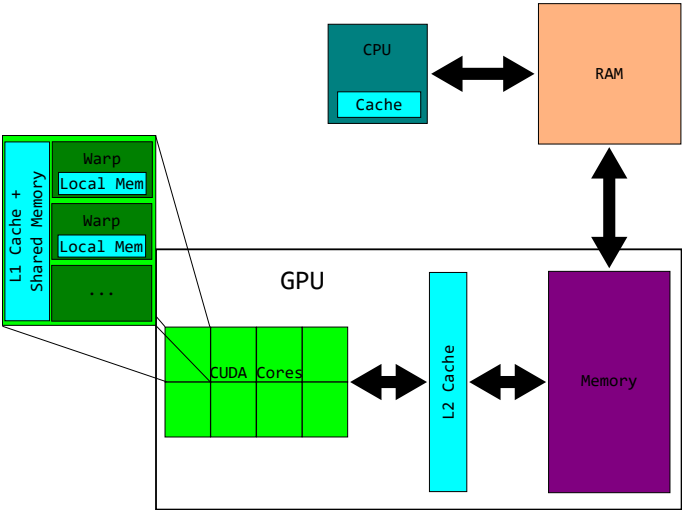
```
int* arr_d;
cudaMalloc((void**)&arr_d, sizeof(int) * N);
kernel<<<4, 256>>>(N, arr_d);

// Make sure device kernel has finished
cudaDeviceSynchronize();

for (int i = 0; i < N; ++i)
    printf("%i\n", arr_d[i]); // Undefined behaviour!
```

This tries to access host memory at the address pointed to by `arr_d`, which could be anything!

Managing memory



Managing Memory

It helps to keep in mind that our code and variables must be executed and stored in one of two places.

CPU Memory

Address	Value
0x00	10
0x04	4.3f
0x08	
0x0C	
0x10	0x08
...	

GPU Memory

Address	Value
0x00	3.0f
0x04	2.0f
0x08	1.0f
0x0C	
0x10	
...	

Managing Memory

It helps to keep in mind that our code and variables must be executed and stored in one of two places.

CPU Memory

Address	Value
0x00	10
0x04	4.3f
0x08	
0x0C	
0x10	0x08
...	

GPU Memory

Address	Value
0x00	3.0f
0x04	2.0f
0x08	1.0f
0x0C	
0x10	
...	

To work with results from the GPU, we first need to copy it back to host memory.

Managing Memory

Chunks of memory can be moved back and forth with `cudaMemcpy()`

- Copy from host to device:

```
cudaMemcpy(arr_d, arr, sizeof(*arr)*N, cudaMemcpyHostToDevice);
```

- Copy from device to host:

```
cudaMemcpy(arr, arr_d, sizeof(*arr)*N, cudaMemcpyDeviceToHost);
```

- Copy between chunks of device memory:

```
cudaMemcpy(dest_d, src_d, sizeof(*arr)*N,
                          cudaMemcpyDeviceToDevice); // no overlap!
```

Managing Memory

Chunks of memory can be moved back and forth with `cudaMemcpy()`

- Copy from host to device:

```
cudaMemcpy(arr_d, arr, sizeof(*arr)*N, cudaMemcpyHostToDevice);
```

- Copy from device to host:

```
cudaMemcpy(arr, arr_d, sizeof(*arr)*N, cudaMemcpyDeviceToHost);
```

- Copy between chunks of device memory:

```
cudaMemcpy(dest_d, src_d, sizeof(*arr)*N,
                          cudaMemcpyDeviceToDevice); // no overlap!
```

Memory copies have an overhead, so better to move larger chunks at once.

Host to device and device to host copies are particularly slow.

Managing Memory

Chunks of memory can be moved back and forth with `cudaMemcpy()`

- Copy from host to device:
```
cudaMemcpy(arr_d, arr, sizeof(*arr)*N, cudaMemcpyHostToDevice);
```
- Copy from device to host:
```
cudaMemcpy(arr, arr_d, sizeof(*arr)*N, cudaMemcpyDeviceToHost);
```
- Copy between chunks of device memory:
```
cudaMemcpy(dest_d, src_d, sizeof(*arr)*N,
                          cudaMemcpyDeviceToDevice); // no overlap!
```

Memory copies have an overhead, so better to move larger chunks at once.

Host to device and device to host copies are particularly slow.

Copies with `cudaMemcpy()` are synchronous between the device and host.
To overlap host computation with memory movement, use:
```
cudaMemcpyAsync(); /* host compute */ cudaDeviceSynchronize();
```

Managing Memory

```
int* arr_d;
cudaMalloc((void**)&arr_d, sizeof(*arr_d) * N);
kernel<<<4, 256>>>(N, arr_d);

// Allocate host memory while kernel is executing
int* arr = malloc(sizeof(*arr) * N);

// Synchronize with device and copy results back to host memory
cudaMemcpy(arr, arr_d, sizeof(*arr) * N, cudaMemcpyDeviceToHost);

// Work with results in host memory
for (int i = 0; i < N; ++i)
    printf("%i\n", arr[i]); // This is OK!
```

Managing Memory

```
int* arr_d;
cudaMalloc((void**)&arr_d, sizeof(*arr_d) * N);
kernel<<<4, 256>>>(N, arr_d);

// Allocate host memory while kernel is executing
int* arr = malloc(sizeof(*arr) * N);

// Copy data back to host asynchronously.
// Won't start until previous kernel launch finishes
cudaMemcpyAsync(arr, arr_d, sizeof(*arr) * N, cudaMemcpyDeviceToHost

/* Do more things on the host while data copies */

cudaDeviceSynchronize(); // Need to sync explicitly with async copy
for (int i = 0; i < N; ++i)
    printf("%i\n", arr[i]);
```

Questions about memory management?

Error Checking

Notice the difference between `malloc` and `cudaMalloc`.

`malloc` returns a pointer to new memory, while `cudaMalloc` takes a pointer to the address of the pointer to new memory. This allows it to return an exit status of type `cudaError_t`.

Almost all CUDA library functions return an exit code of this type.

To save having to write detailed error-handling code for every function call, it's handy to wrap this in a macro.

Error Checking

```
#define cuda_check_impl(cmd, abort) { \
    cudaError_t status = (cmd); \
    if (status != cudaSuccess) { \
        fprintf(stderr, "CUDA Error: %s (%s:%d)\n", \
            cudaGetErrorString(status), __FILE__, __LINE__);  \
        if (abort) exit(status); \
    } }
#define cuda_check(cmd) cuda_check_impl((cmd), 1)
#define cuda_check_noabort(cmd) cuda_check_impl((cmd), 0)
// ...

int* arr_d; cuda_check(cudaMalloc((void**)&arr_d, sizeof(*arr_d) * N
kernel<<<4, 256>>>(N, arr_d);
int* arr = malloc(sizeof(*arr) * N);
cuda_check(cudaMemcpy(arr, arr_d, sizeof(*arr) * N, \
    cudaMemcpyDeviceToHost));
```

Demo: let's make something fail.

Error Checking

Demo: let's make something fail.

Use `cudaGetLastError()` to check kernel execution!

Error Checking

Demo: let's make something fail.

Use `cudaGetLastError()` to check kernel execution!
There are also other handy error-related functions such as `cudaPeekAtLastError()`.

Exercise: axpy

Try writing a CUDA program that evaluates $aX+Y$ and stores the result back into X.

Execution spaces

As we've seen, any CUDA functions that execute on the GPU, but are called from the CPU, have been marked `__global__`.

This decorator tells `nvcc` that the code in the function should be compiled to GPU code, but the function could be called from anywhere. (Including from within another kernel!)

Execution spaces

As we've seen, any CUDA functions that execute on the GPU, but are called from the CPU, have been marked `__global__`.

This decorator tells `nvcc` that the code in the function should be compiled to GPU code, but the function could be called from anywhere. (Including from within another kernel!)

There are two other options:

- `__device__`: a function that should be compiled to GPU code, and be callable from other code executing on the GPU.
- `__host__`: a function that should be compiled to CPU code, and should be callable from other code executing on the CPU (the default in the case of no decorator).

`__device__` functions can be handy when you need to repeat code inside GPU kernels.

Execution spaces

As we've seen, any CUDA functions that execute on the GPU, but are called from the CPU, have been marked `__global__`.

This decorator tells `nvcc` that the code in the function should be compiled to GPU code, but the function could be called from anywhere. (Including from within another kernel!)

There are two other options:

- `__device__`: a function that should be compiled to GPU code, and be callable from other code executing on the GPU.
- `__host__`: a function that should be compiled to CPU code, and should be callable from other code executing on the CPU (the default in the case of no decorator).

`__device__` functions can be handy when you need to repeat code inside GPU kernels.

Functions can also be marked as both `__device__` and `__host__`, in which case two versions of the function will be compiled—one to GPU instructions, and one to CPU instructions.

Execution spaces

Global variables can also be marked as `__device__`, in which case they are only accessible from GPU code.

```
__device__ int value_d;

__global__ void kernel() {
    printf("%i\n", value_d); // value_d is accessible here
}

int main(void) {
    printf("%i\n", value_d); // This is a compile error!
}
```

Execution spaces

To access __device__ variables from the host, use:

```
__device__ int value_d = 1;
int main(void) {
    int* d_ptr_to_value_d;
    cudaGetSymbolAddress((void**)&d_ptr_to_value_d, value_d);

    // Get value of value_d
    int value;
    cudaMemcpy(&value, d_ptr_to_value_d, sizeof(int), \
               cudaMemcpyDeviceToHost);

    // Set value of value_d
    value = 10;
    cudaMemcpy(d_ptr_to_value_d, &value, sizeof(int),
               cudaMemcpyHostToDevice);
}
```

Execution spaces

Alternatively, there are helper functions for working directly with `__device__` variables:

```
__device__ int value_d = 1;
int main(void) {
    // Get value of value_d
    int value;
    cudaMemcpyFromSymbol(&value, value_d, sizeof(int), 0, \
                         cudaMemcpyDeviceToHost);
    // Set value of value_d
    value = 10;
    cudaMemcpyToSymbol(value_d, &h_value, sizeof(int), 0, \
                       cudaMemcpyHostToDevice);
}
```

In this case, the extra argument (currently set to 0) is a pointer offset in bytes, which is needed if `value_d` were an array.

Execution spaces

For example:

```
__device__ int values_d[] = {1, 2};
int main(void) {
    int one, two;

    // Copy first element from values_d
    cudaMemcpyFromSymbol(&one, values_d, sizeof(int), \
                         0, cudaMemcpyDeviceToHost);
    // Copy 2nd element from values_d
    cudaMemcpyFromSymbol(&two, values_d, sizeof(int), \
                         1*sizeof(int), cudaMemcpyDeviceToHost);
}
```

Execution spaces

Questions about execution spaces?

`__shared__` memory

Sometimes we want to transform data through a few steps in a kernel before writing it back to GPU memory.

If that data only needs to be worked on within a block, the algorithm can often be sped up by first loading the initial data into *shared memory*, transforming it there, and then writing it back to global memory once at the end. Shared memory is partitioned in L1 Cache.

__shared__ memory

Sometimes we want to transform data through a few steps in a kernel before writing it back to GPU memory.

If that data only needs to be worked on within a block, the algorithm can often be sped up by first loading the initial data into *shared memory*, transforming it there, and then writing it back to global memory once at the end. Shared memory is partitioned in L1 Cache.

A reduction algorithm is a good example of this.
Consider the naive approach (just handling reduction within each block for simplicity):

```
__global__ reduce(int* data) {
    int idx = blockIdx.x * blockDim.x * 2 + threadIdx.x;
    for (int i = blockDim.x; i > 0; i /= 2) {
        if (threadIdx.x < i)
            data[idx] =  data[idx] + data[idx + i];
        __syncthreads();
    }
}
```

`__shared__` memory

We can speed this up by using shared memory:

```
__global__ reduce(int* data) {
    extern __shared__ int s_mem[];
    int idx = blockIdx.x * blockDim.x * 2 + threadIdx.x;
    s_mem[threadIdx.x] = data[idx];    // Want consecutive reads
    s_mem[threadIdx.x] += data[idx + blockDim.x];
    for (int i = blockDim.x/2; i > 0; i /= 2) {
        __syncthreads();
        if (threadIdx.x < i)
            s_mem[threadIdx.x] =  s_mem[threadIdx.x] +
                                  s_mem[threadIdx.x + i];
    }
    if (threadIdx.x == 0) data[idx] = s_mem[0];
}
```

Launch with enough shared memory allocated (one `int` per thread in a block):

```
reduce<<<10, 256, 256*sizeof(int)>>>(data_to_reduce_d);
```

Questions about shared memory?

Streams

CUDA has a concept of *streams*, which represent a sequence of calculations.

By default, kernels and library calls are sent to the *default stream*.

This means that the code below is entirely synchronous:

```
cudaMemcpy(data_d, data, data_size, cudaMemcpyHostToDevice);
kernel<<<BLOCKS,THREADS>>>(data_d);
cudaMemcpy(data, data_d, data_size, cudaMemcpyDeviceToHost);
```

Streams

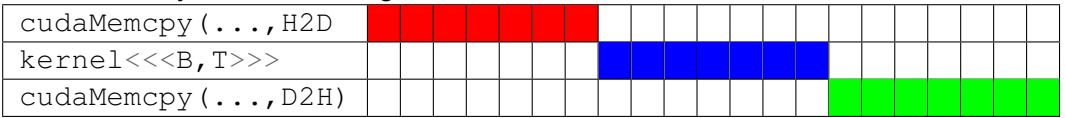
CUDA has a concept of *streams*, which represent a sequence of calculations.

By default, kernels and library calls are sent to the *default stream*.

This means that the code below is entirely synchronous:

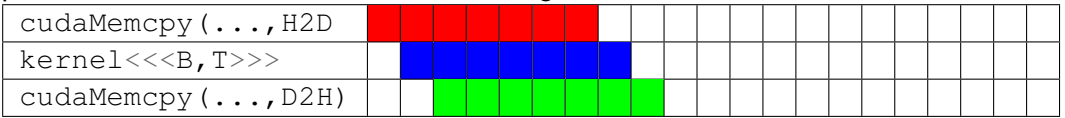
```
cudaMemcpy(data_d, data, data_size, cudaMemcpyHostToDevice);
kernel<<<BLOCKS,THREADS>>>(data_d);
cudaMemcpy(data, data_d, data_size, cudaMemcpyDeviceToHost);
```

Execution may look something like this:



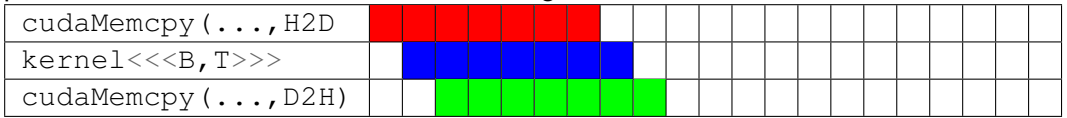
Streams

Often, blocks of computation within a kernel are independent, so in theory we could gain performance if execution looked something like this:



Streams

Often, blocks of computation within a kernel are independent, so in theory we could gain performance if execution looked something like this:



We can achieve this with streams!

Streams

We can create and use a new stream with the code below:

```
// Create stream
cudaStream_t stream1;
cuda_check(cudaStreamCreate(&stream1));

// Run kernel in stream
kernel<<<BLOCKS,BLOCK_SIZE,0,stream1>>>();

// Make sure kernel has finished executing
cuda_check(cudaStreamSynchronize(stream1));

// Clean up stream
cuda_check(cudaStreamDestroy(stream1));
```

Streams

For the previous example, we can achieve overlapped execution with:

```
int *data, *data_d;
cuda_check(cudaMalloc((void**)&data_d, sizeof(int) * N));
cuda_check(cudaMallocHost(&data, sizeof(int) * N)); // Pinned memory!

// ...

int n_streams = (N + STREAM_SIZE - 1) / STREAM_SIZE;
cudaStream_t* streams = malloc(sizeof(cudaStream_t) * n_streams);
for (int i = 0; i < n_streams; ++i)
    cuda_check(cudaStreamCreate(&streams[i]));

// contd..
```

Streams

```
dim3 GRID_SIZE = STREAM_SIZE/BLOCK_SIZE;
for (int i = 0; i < n_streams; ++i) {
    int offset = STREAM_SIZE * i;
    cudaMemcpyAsync(&data_d[offset], &data[offset], \
                    sizeof(int) * STREAM_SIZE, \
                    cudaMemcpyHostToDevice, streams[i]);
    kernel<<<GRID_SIZE, BLOCK_SIZE, 0, streams[i]>>>(&data_d[offset]);
    cudaMemcpyAsync(&data[offset], &data_d[offset], \
                    sizeof(int) * STREAM_SIZE, \
                    cudaMemcpyDeviceToHost, streams[i]);
}
// Possible overlapped CPU compute

// contd..
```

Streams

```
// Clean up:

// Check errors
cuda_check(cudaGetLastError());

// Synchronize
// Could replace with cudaDeviceSynchronize(); if no other streams
for (int i = 0; i < n_streams; ++i)
    cuda_check(cudaStreamSynchronize(streams[i]));

// Destroy streams
for (int i = 0; i < n_streams; ++i)
    cuda_check(cudaStreamDestroy(stream[i]));
```

Exercise: Modify your axpy code to use streams for overlapped data transfer and computation.

Modular code

It's often becomes necessary to link CUDA code into existing C code.
One option is to compile everything with `nvcc`, but this can lead to all sorts of issues, particularly in a large existing codebase.

Modular code

It's often becomes necessary to link CUDA code into existing C code.
One option is to compile everything with `nvcc`, but this can lead to all sorts of issues, particularly in a large existing codebase.

Alternatively, the CUDA code can be compiled to have a C interface so that it can be linked in:

gpu.h

```
#ifndef GPU_H
#define GPU_H
void gpu_func();
#endif
```

main.c

```
#include "gpu.h"
int main(void) {
    gpu_func();
    return 0;
}
```

gpu.cu

```
extern "C" {
#include "gpu.h"
}
#include <stdio.h>
__global__ void kernel() {
    printf("tid: %d\n", threadIdx.x);
}
extern "C" void gpu_func() {
    kernel<<<1,256>>>();
    cudaDeviceSynchronize();
}
```

Modular code

To compile:

```
$ nvcc -c gpu.cu
$ gcc main.c gpu.o -L/path/to/cuda/lib/ -lcudart
```

This first command compiles the GPU code into `gpu.o`, in which `gpu_func()` has C linkage (because of `extern "C"`).

The C linkage is required, since C and CUDA C are not the same language.

Modular code

To compile:

```
$ nvcc -c gpu.cu
$ gcc main.c gpu.o -L/path/to/cuda/lib/ -lcudart
```

This first command compiles the GPU code into `gpu.o`, in which `gpu_func()` has C linkage (because of `extern "C"`).

The C linkage is required, since C and CUDA C are not the same language.

The second command then compiles the C code into the executable, linking against `gpu.o` and the CUDA runtime library (`cudart`).

Modular code

To compile:

```
$ nvcc -c gpu.cu
$ gcc main.c gpu.o -L/path/to/cuda/lib/ -lcudart
```

This first command compiles the GPU code into `gpu.o`, in which `gpu_func()` has C linkage (because of `extern "C"`).

The C linkage is required, since C and CUDA C are not the same language.

The second command then compiles the C code into the executable, linking against `gpu.o` and the CUDA runtime library (`cudart`).

You can call CUDA runtime functions (e.g. `cudaMalloc()`) directly from C code by including `<cuda_runtime.h>` and adding `-I/path/to/cuda/include/`

Modular code

To compile:

```
$ nvcc -c gpu.cu
$ gcc main.c gpu.o -L/path/to/cuda/lib/ -lcudart
```

This first command compiles the GPU code into `gpu.o`, in which `gpu_func()` has C linkage (because of `extern "C"`).

The C linkage is required, since C and CUDA C are not the same language.

The second command then compiles the C code into the executable, linking against `gpu.o` and the CUDA runtime library (`cudart`).

You can call CUDA runtime functions (e.g. `cudaMalloc()`) directly from C code by including `<cuda_runtime.h>` and adding `-I/path/to/cuda/include/`

Note

Loading the `cuda` or `nvidia-hpc-sdk` modules on Gadi will set environment variables so that the `-L` and `-I` flags are not needed.

Modular code

It can also be necessary to cross-compile to a GPU architecture not on the compiling machine, or to multiple architectures to support different GPUs from the same binary. `CMake` can help with this, but to do it manually:

```
$ nvcc -c gpu.cu -gencode=arch=compute_XY,code=sm_XY
```

Replace `XY` with the compute capability of the card you're using. For example, compute capability 7.2 would use `arch=compute_72,code=sm_72`

Multiple `-gencode` flags can be passed to support multiple compute capabilities. `arch` and `code` can also be specified separately with the `-arch` and `-code` flags.

Modular code

It can also be necessary to cross-compile to a GPU architecture not on the compiling machine, or to multiple architectures to support different GPUs from the same binary. `CMake` can help with this, but to do it manually:

```
$ nvcc -c gpu.cu -gencode=arch=compute_XY,code=sm_XY
```

Replace `XY` with the compute capability of the card you're using. For example, compute capability 7.2 would use `arch=compute_72,code=sm_72`

Multiple `-gencode` flags can be passed to support multiple compute capabilities. `arch` and `code` can also be specified separately with the `-arch` and `-code` flags.

`arch=...` specifies the *virtual* architecture for which PTX code is generated. PTX code can be dynamically compiled to supported architectures at runtime.

Runtime compilation can be avoided by specifying *real* architectures to compile for with `code=...`

Exercise

Try abstracting your axpy code to a `.o` file and linking it into some C code.

Conclusion

While this is enough to get started with CUDA, it only scratches the surface on many of the details. There are more advanced features that can be used to speed up your code, and it can help to consider the particular GPU architecture the code will run on.

There are also a number of useful CUDA libraries which we haven't covered. For example:

- cuRAND: Random number generation
- cuBLAS: Basic linear algebra in CUDA
- cuSPARSE: Basic linear algebra for sparse data
- cuSOLVER: Based on cuBLAS and cuSPARSE, provides LAPACK-like features
- cuFFT: GPU accelerated fast Fourier transforms
- CUB: C++ library for common CUDA algorithms (e.g. reduction, scan, much more)

Conclusion

While this is enough to get started with CUDA, it only scratches the surface on many of the details. There are more advanced features that can be used to speed up your code, and it can help to consider the particular GPU architecture the code will run on.

There are also a number of useful CUDA libraries which we haven't covered. For example:

- cuRAND: Random number generation
- cuBLAS: Basic linear algebra in CUDA
- cuSPARSE: Basic linear algebra for sparse data
- cuSOLVER: Based on cuBLAS and cuSPARSE, provides LAPACK-like features
- cuFFT: GPU accelerated fast Fourier transforms
- CUB: C++ library for common CUDA algorithms (e.g. reduction, scan, much more)

For debugging and profiling, check out `cuda-gdb`, `nvprof`, and Nsight

Resources

CUDA runtime API documentation for details of the available functions:
https://docs.nvidia.com/cuda/cuda-runtime-api/index.html

Best practices guide:
https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html

Table of compute capability support of GPU models:
https://en.wikipedia.org/wiki/CUDA#GPUs_supported

NCI TechTake talk on some of the nuances of shared memory:
https://www.youtube.com/watch?v=9QEvmIQnmlw