



# University of Zurich<sup>UZH</sup>

## High Performance Computing Lecture 13

Douglas Potter

Jozef Bucko

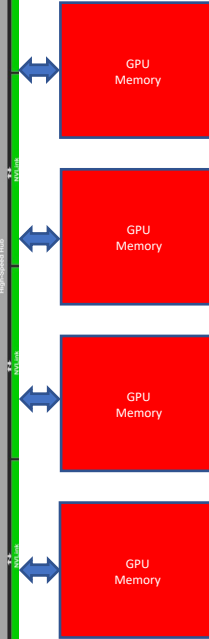
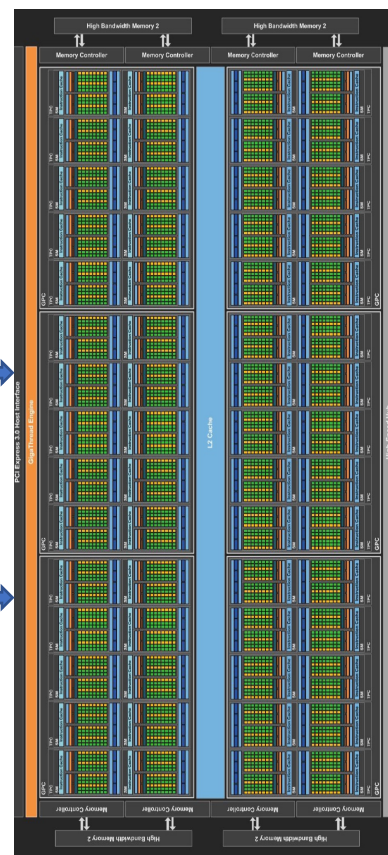
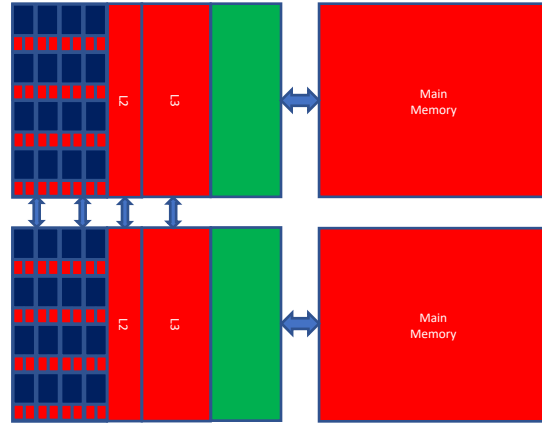
Noah Kubli

# What is CUDA?



- CUDA Architecture
  - Exposes the GPU parallelism for general-purpose computing
  - Retains performance
- CUDA C/C++
  - Based on C/C++
  - Small set of GPU specific extensions
  - API to manage devices, memory, etc.

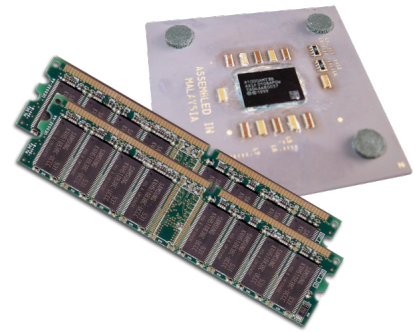
# Terminology



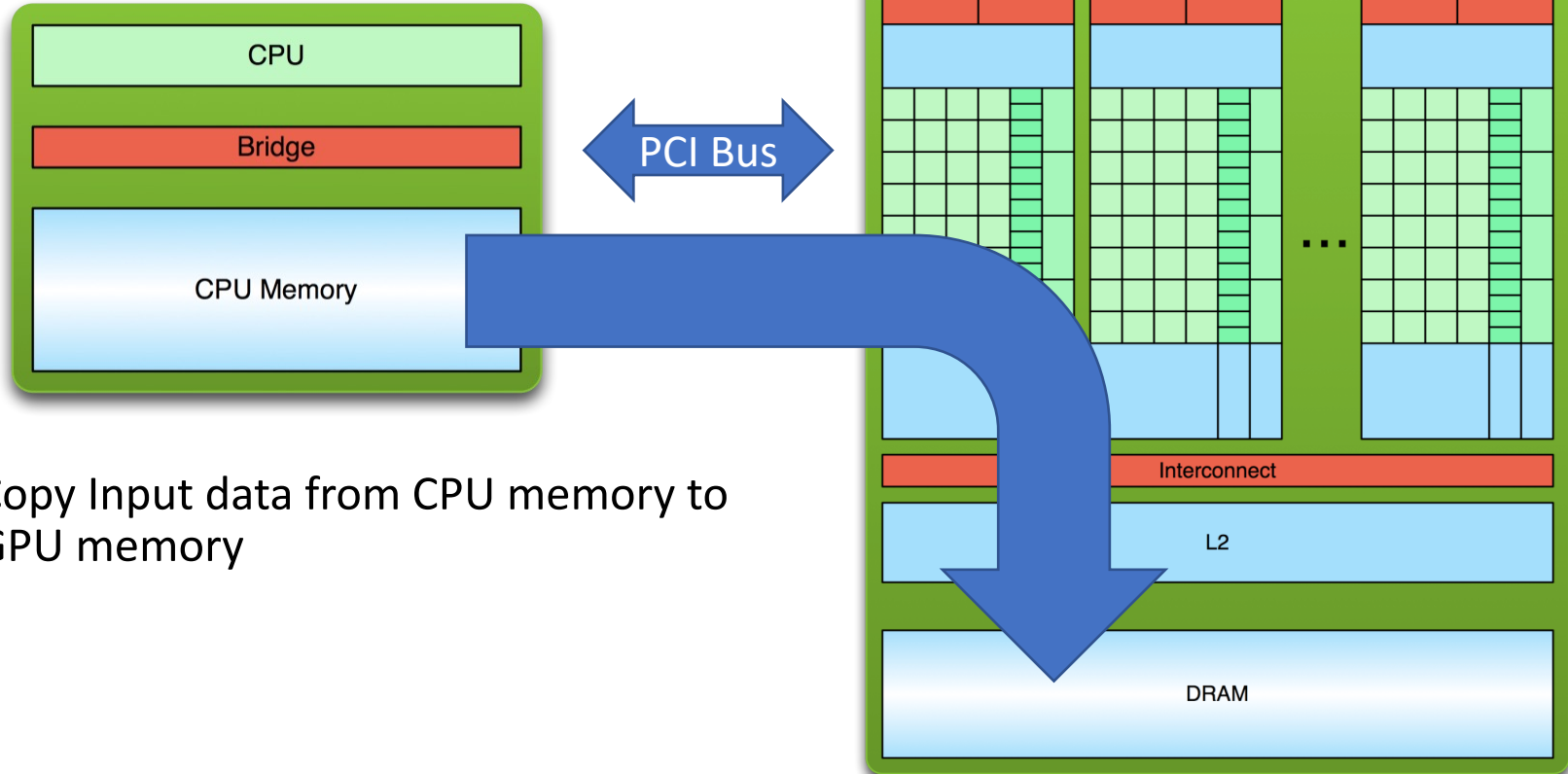
Host

Device

High Performance Computing

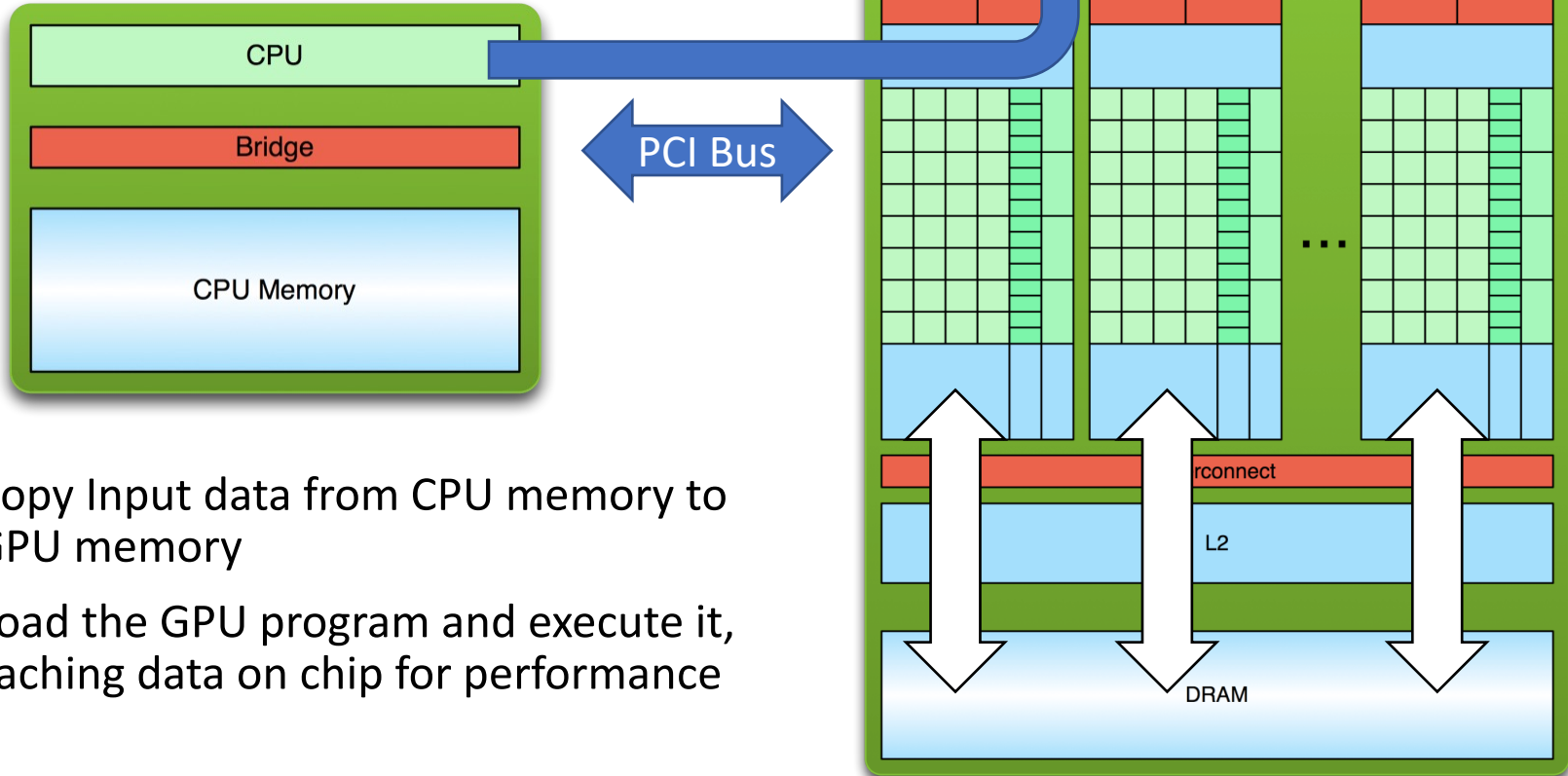


# Simple processing flow



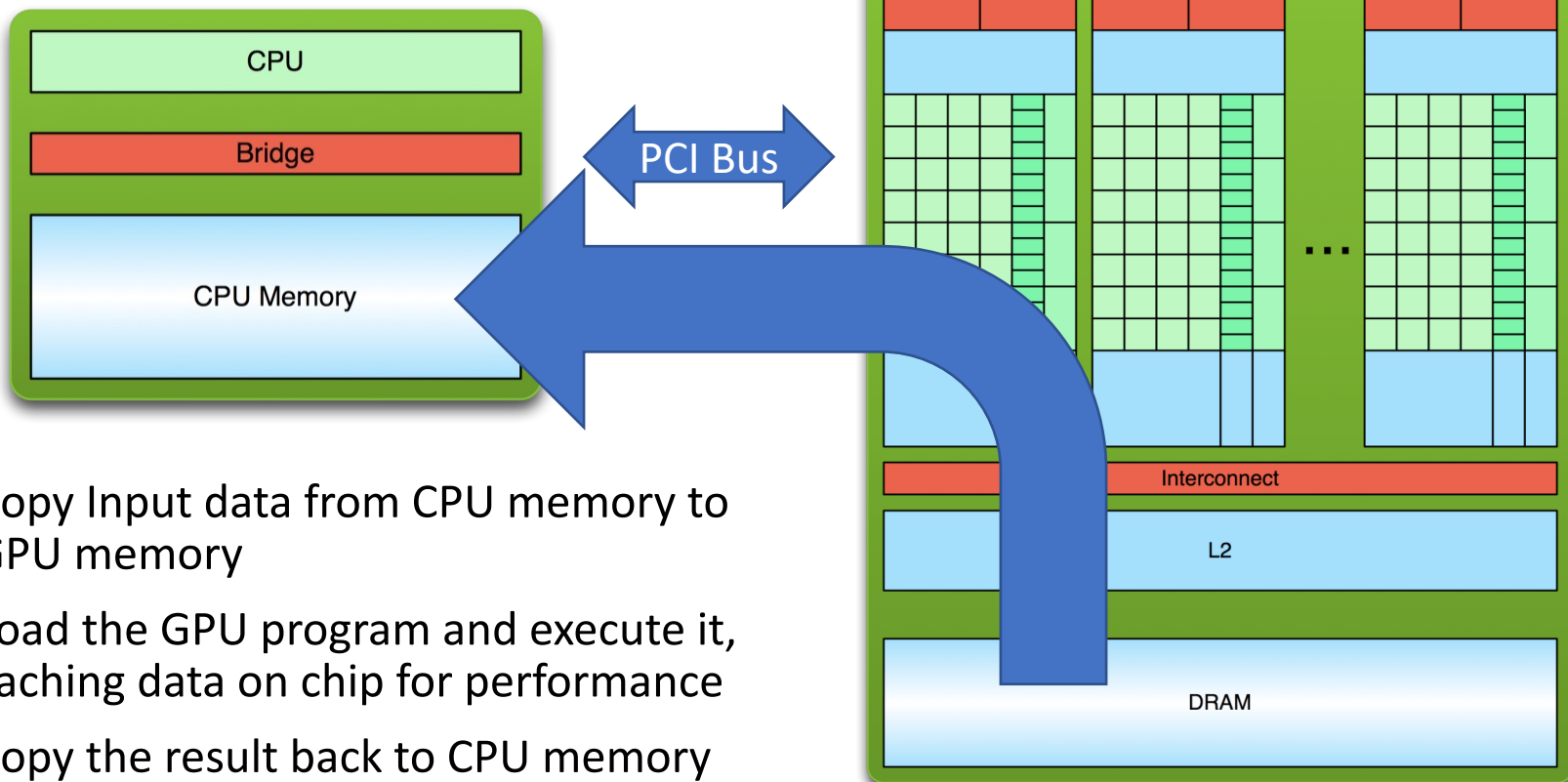
1. Copy Input data from CPU memory to GPU memory

# Simple processing flow



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

# Simple processing flow



1. Copy Input data from CPU memory to GPU memory
2. Load the GPU program and execute it, caching data on chip for performance
3. Copy the result back to CPU memory

# Hello World!

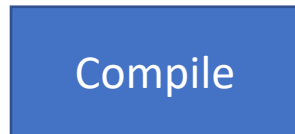
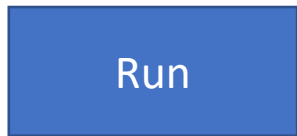
```
int main(void) {  
    printf("Hello world!\n");  
    return 0;  
}
```

- File extension is “.cu”
- Standard C/C++ code that is run on the host
- Can compile C/C++ programs without any device code

```
$ 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 syntactical elements
- The `__global__` keyword is used to indicate that a function runs on the **device** and is called from the **host**.
- The nVidia compiler (nvcc) separates **host** and **device** code.
  - The **device** code is processed by the nVidia compiler.
  - The **host** code is passed to **gcc**.



# Hello world! with device code

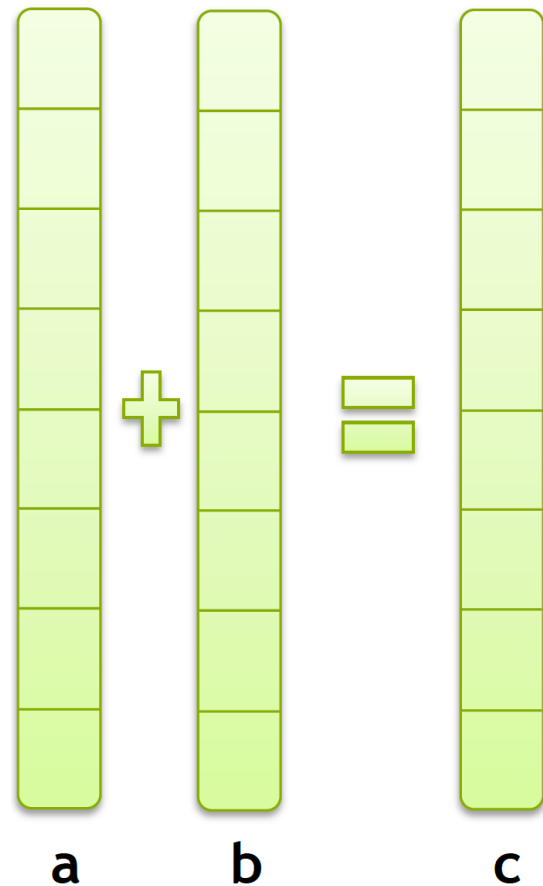
```
__global__ void
mykernel(void) {

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

- The triple angle brackets “<<<” and “>>>” mark a call from **host** code to **device** code.
  - Called a “kernel launch”
  - The numbers inside correspond to the number of **blocks** and **threads per block**
- This will execute the function “mykernel” on the GPU!
- It doesn’t do much yet...

# Parallel programming with CUDA

- GPU Programming is about massive parallelism
- Let's start by adding two numbers on the GPU
- Later, we will modify this to add two vectors
- This starts to get more complicated



# Addition on the device

- We copy data to the GPU memory so that's where we will find it!

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

- The `__global__` keyword means that the “add” function will be run on the **device** and called from the **host**.
- The kernel “add” will compute:  $c = a + b$
- The variables “a”, “b” and “c” are pointers to integers on the **device**.
  - Of course we have to manage this memory ourselves!

# Memory Management

- **Host** and **Device** memory are separate entities
  - **Device** pointers point to GPU memory
    - May be passed between **host** functions
    - May **NOT** be dereferenced in **host** functions
  - **Host** pointers point to CPU memory
    - May be passed to/from **device** code (but why?)
    - May **NOT** be dereferenced in **device** code.
- We will use the CUDA API to manage **device** memory
  - `cudaMalloc()`, `cudaFree()`, `cudaMemcpy()`
  - Similar to the C functions: `malloc()`, `free()`, `memcpy()`

# Back to main() – allocating GPU memory

```
int main(void) {  
    int a, b, c; // host copies of a, b, c  
    int *a_d, *b_d, *c_d; // device copies  
    int size = sizeof(int);  
  
    // Allocate device copies of a, b, c  
    cudaMalloc((void **)&a_d, size);  
    cudaMalloc((void **)&b_d, size);  
    cudaMalloc((void **)&c_d, size);  
  
    // Setup input values  
    a = 2;  
    b = 7;
```

- We have a copy of the data on the **host**
  - The variables “a”, “b” and “c”
- We need to allocate space on the **device**
  - The pointers “a\_d”, “b\_d”, “c\_d”
  - These will point to memory on the **device**
- We need “size” bytes (one integer)
- Calling **cudaMalloc** will allocate **device** memory
  - Allocates “size” bytes
  - Returns the address on the **device**
  - cudaMalloc is “slow” – normally used at startup
- Memory is allocated, but we still need to transfer!

# Back to main() – data transfer and kernel

```
// Copy inputs to device
cudaMemcpy(a_d, &a, size, cudaMemcpyHostToDevice);
cudaMemcpy(b_d, &b, size, cudaMemcpyHostToDevice);
// Launch add() kernel on GPU
add<<<1,1>>>(a_d, b_d, c_d);
// Copy result back to host
cudaMemcpy(&c, c_d, size, cudaMemcpyDeviceToHost);
// Print the result
printf("c = %d\n", c);
// Cleanup
cudaFree(a_d); cudaFree(b_d); cudaFree(c_d);
return 0;
}
```

- We first copy “a” and “b” to the **device**
  - cudaMemcpy() does this
  - Need to specify the source and destination addresses
  - Also need a direction: HostToDevice
- Now we launch the kernel on the **device**
- And copy the result “c” back to the **host**
  - Again with cudaMemcpy()
  - Direction is: DeviceToHost
- Whew! Lots of work to add two numbers!
- Next let’s switch to parallel

# Parallel “add” kernel

- GPU computing is all about **massive** parallelism
- To run in parallel we just change the kernel launch parameters

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



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

- Instead of executing **add()** once, it now executes “N” times in parallel

# Back to main() – Memory for “N” integers

```
#define N 512

int main(void) {
    int *a, *b, *c;           // host copies of a, b, c
    int *a_d, *b_d, *c_d;    // device copies
    int size = N * sizeof(int);

    // Allocate device copies of a, b, c
    cudaMalloc((void **)&a_d, size);
    cudaMalloc((void **)&b_d, size);
    cudaMalloc((void **)&c_d, size);

    // Allocate host copies of a, b, c
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);
```

- We have a copy of the data on the **host**
  - The variables “a”, “b” and “c”
  - These are now pointers to an array
- We need to allocate space on the **device**
  - The pointers “a\_d”, “b\_d”, “c\_d”
  - These will point to memory on the **device**
- We need “size” bytes (N integers)
- Calling **cudaMalloc** will allocate **device** memory
  - Allocates “size” bytes
  - Returns the address on the **device**
  - cudaMalloc is “slow” – normally used at startup



# Back to main() – data transfer and kernel

```
// Copy inputs to device
cudaMemcpy(a_d, a, size, cudaMemcpyHostToDevice);
cudaMemcpy(b_d, b, size, cudaMemcpyHostToDevice);
// Launch add() kernel on GPU
add<<<N,1>>>(a_d, b_d, c_d);
// Copy result back to host
cudaMemcpy(c, c_d, size, cudaMemcpyDeviceToHost);
// Cleanup
free(a); free(b); free(c);
cudaFree(a_d); cudaFree(b_d); cudaFree(c_d);
return 0;
}
```

- We first copy “a” and “b” to the **device**
  - cudaMemcpy() does this
  - Need to specify the source and destination addresses
  - Also need a direction: HostToDevice
- Now we launch the kernel on the **device**
- And copy the result “c” back to the **host**
  - Again with cudaMemcpy()
  - Direction is: DeviceToHost
- This time we launched “N” threads to process the data in parallel.
- How does the “kernel” change?

# Vector addition on the device

- We requested that 512 blocks be run on the device

```
#define N 512  
add<<<N, 1>>> (a_d, b_d, c_d) ;
```

- Terminology: A set of **blocks** is called a **grid**.
- **Blocks** are numbered sequentially from zero, e.g., 0 to 511
- Each thread can use **blockIdx.x** to get its **block** number

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

- 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 be executed in parallel

| Block 0              | Block 1              | Block 2              | Block 3              |
|----------------------|----------------------|----------------------|----------------------|
| $c[0] = a[0] + b[0]$ | $c[1] = a[1] + b[1]$ | $c[2] = a[2] + b[2]$ | $c[3] = a[3] + b[3]$ |

# Summary so far

- **Host** versus **Device**

- **Host**                      CPU

- **Device**                   GPU

- Use **\_\_global\_\_** to declare a function as **device** code
  - Executes on the **device**
  - Called from the **host**
- Passing parameters from **host** code to a **device** function.

- **Device** memory management

- cudaMalloc()
  - cudaFree()
  - cudaMemcpy()

- Launching parallel kernels

- Launch N blocks of add() with

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

- Use blockIdx.x for the block index

# CUDA Threads

- Terminology: A **block** can be split into parallel **threads**.
- Here is **add()** using parallel **threads** instead of **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**
- We need to make one change to the kernel launch...

# Parallel “add” kernel with “N” threads

- Now we run with **1 thread block** with **N threads** instead of N thread blocks with 1 thread each.

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



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

# Combining Blocks and Threads

- We've seen parallel vector addition using:
  - **Many blocks** with **one thread** each `add<<<N, 1>>>`
  - **One block** with **many threads** `add<<<1, N>>>`
- For optimal performance we need to use both at the same time
  - Threads and blocks have a “hardware” interpretation (more later)
  - Only by using both can you achieve full performance
  - There are hardware limitations that come into play
- Our simple thread indexing (`blockIdx.x` or `threadIdx.x`) gets complicated
  - We need to use both at the same time.
  - Let's see how...

# Indexing

- A kernel is launched as a **grid** of **blocks** of **threads**

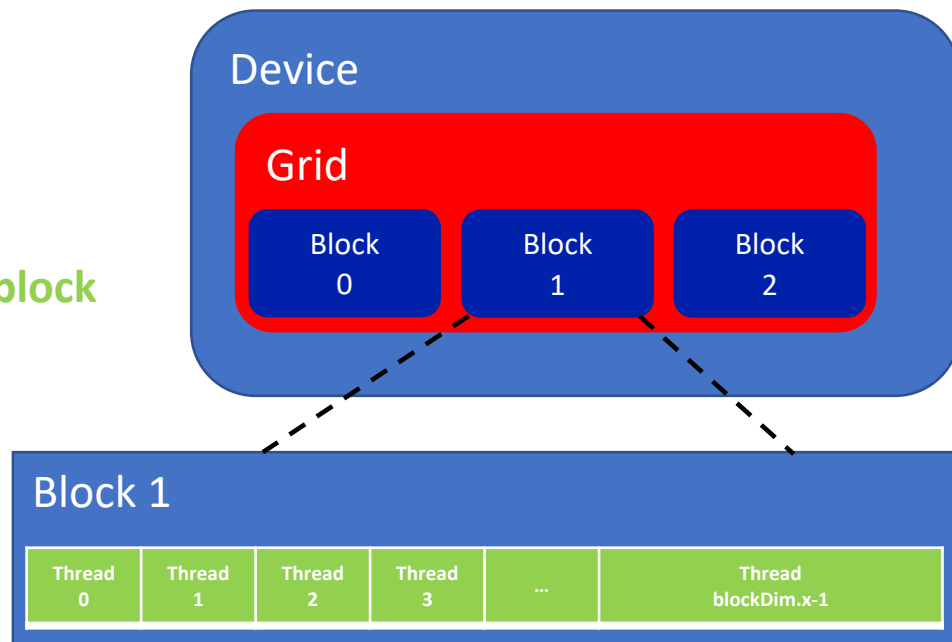
## Built-in variables:

**threadIdx.x** Thread inside the block

**blockDim.x** Threads per block

**blockIdx.x** Block index

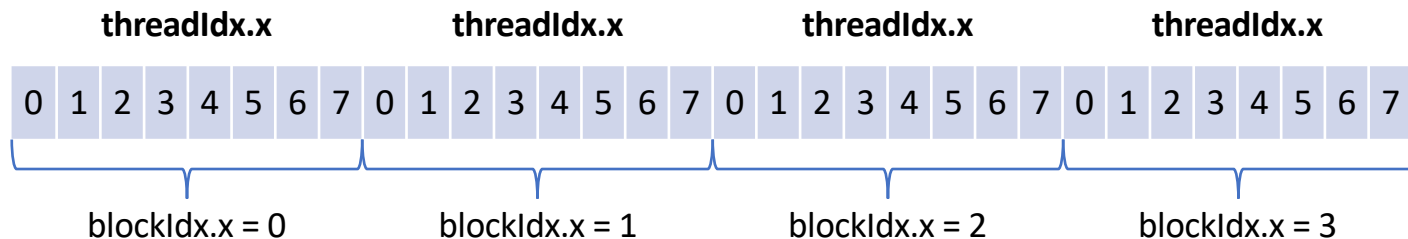
**gridDim.x** Blocks in the grid





# Indexing arrays

- We can no longer use just **blockIdx.x** or **threadIdx.x**
- Consider 4 **blocks** with 8 **threads** per block

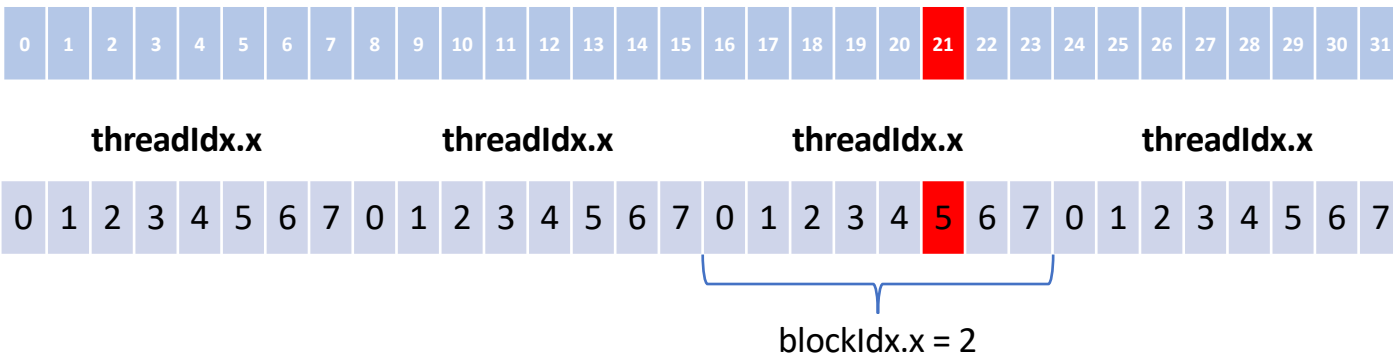


- Here **gridDim.x** == 4 (4 blocks) and **blockDim.x** == 8 (8 threads), so  

$$\text{int index} = \text{threadIdx.x} + \text{blockIdx.x} * \text{blockDim.x}$$

# Indexing arrays: Example

- Which thread will operate on the red element?



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

# Vector addition with both threads and blocks

- Here our **add()** kernel uses both parallel **threads** and **blocks**
- We use the formula we just discussed

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

- We use **threadIdx.x**, **blockIdx.x**, and **blockDim.x**
- We need to adjust our main() program again...

# Back to main() – Blocks and Threads

```
#define N (2048*2048)
#define BLOCKSIZE 512
int main(void) {
    int *a, *b, *c;          // host copies of a, b, c
    int *a_d, *b_d, *c_d;    // device copies
    int size = N * sizeof(int);

    // Allocate device copies of a, b, c
    cudaMalloc((void **)&a_d, size);
    cudaMalloc((void **)&b_d, size);
    cudaMalloc((void **)&c_d, size);

    // Allocate host copies of a, b, c
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);
```

- Increase the total array length
  - $N$  is now  $2048^2 = 4'194'304$
- The block size is 512
  - This is the number of threads per block
- How many blocks do we need?
  - $N / \text{BLOCKSIZE}$
  - 8'192

# Back to main() – **Blocks** and **Threads**

```
// Copy inputs to device
cudaMemcpy(a_d, a, size, cudaMemcpyHostToDevice);
cudaMemcpy(b_d, b, size, cudaMemcpyHostToDevice);
// Launch add() kernel on GPU
add<<<N/BLOCKSIZE, BLOCKSIZE>>>(a_d, b_d, c_d);
// Copy result back to host
cudaMemcpy(c, c_d, size, cudaMemcpyDeviceToHost);
// Cleanup
free(a); free(b); free(c);
cudaFree(a_d); cudaFree(b_d); cudaFree(c_d);
return 0;
}
```

# How to handle arbitrary vector sizes

- The previous example works if “N” is an exact multiple of BLOCKSIZE
  - What happens if N = 560?
- Change the kernel to avoid accessing beyond the end of the array

```
__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];  
}
```

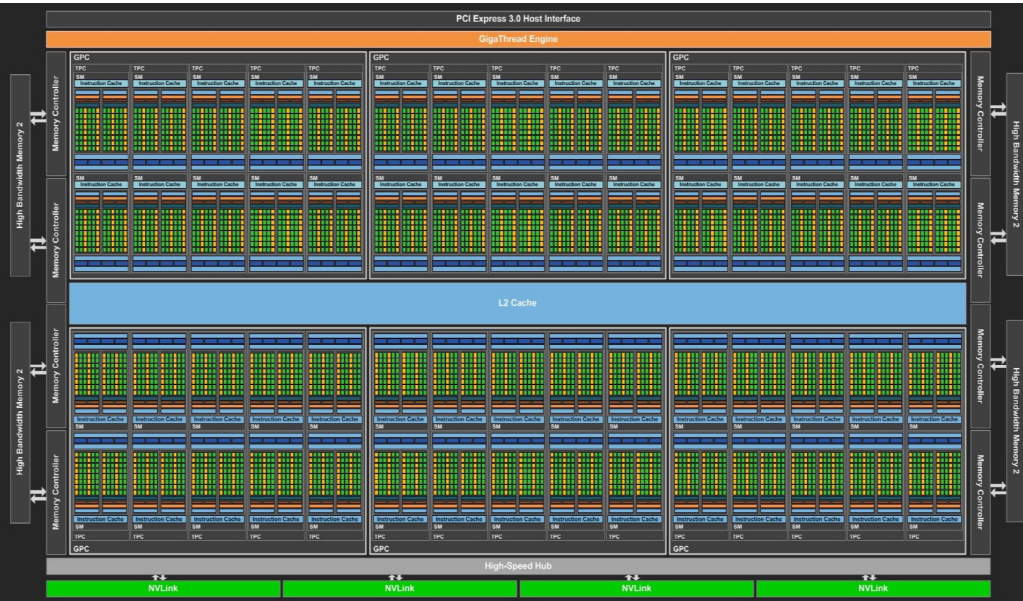
- Update the kernel launch –  $\lceil N \div BLOCKSIZE \rceil$  (ceiling)

```
add<<< (N+BLOCKSIZE-1)/BLOCKSIZE, BLOCKSIZE >>>(a_d, b_d, c_d, N);
```

# Why bother with Threads and Blocks

- Using both seems unnecessarily complicated
  - We have to match our problem to this additional layer
  - Do we really gain anything?
- Threads in a block can communicate through shared memory
  - Threads in different memory can only communicate through global memory
- There is a limit on the number of threads per block
  - The limit is very small – usually 1024
- Basically we have to or we won't get good performance

# Pascal P100 block diagram





# Compute Capability

- Each device has a compute capability level
  - Higher numbers are not necessarily “better”.
- You can query
  - the level at runtime, or,
  - individual device attributes.
- You often don’t really care, except:
  - You probably want to specify the correct level when compiling.

## NVIDIA Data Center Products

| GPU         | Compute Capability |
|-------------|--------------------|
| NVIDIA A100 | 8.0                |
| NVIDIA A40  | 8.6                |
| NVIDIA A30  | 8.0                |
| NVIDIA A10  | 8.6                |
| NVIDIA A16  | 8.6                |
| NVIDIA A2   | 8.6                |
| NVIDIA T4   | 7.5                |
| NVIDIA V100 | 7.0                |
| Tesla P100  | 6.0                |
| Tesla P40   | 6.1                |
| Tesla P4    | 6.1                |
| Tesla M60   | 5.2                |
| Tesla M40   | 5.2                |
| Tesla K80   | 3.7                |
| Tesla K40   | 3.5                |
| Tesla K20   | 3.5                |
| Tesla K10   | 3.0                |

High Performance Computing

## GeForce and TITAN Products

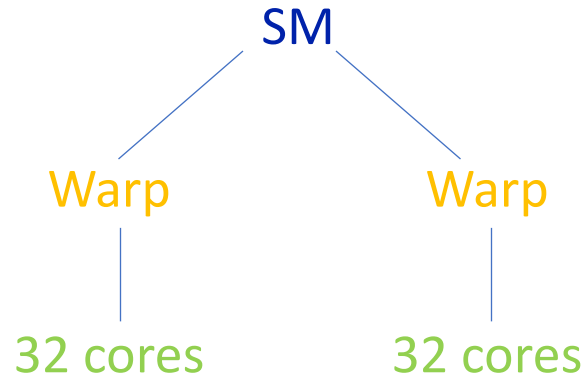
| GPU                 | Compute Capability |
|---------------------|--------------------|
| Geforce RTX 3060 Ti | 8.6                |
| Geforce RTX 3060    | 8.6                |
| GeForce RTX 3090    | 8.6                |
| GeForce RTX 3080    | 8.6                |
| GeForce RTX 3070    | 8.6                |
| GeForce GTX 1650 Ti | 7.5                |
| NVIDIA TITAN RTX    | 7.5                |
| Geforce RTX 2080 Ti | 7.5                |
| Geforce RTX 2080    | 7.5                |
| Geforce RTX 2070    | 7.5                |
| Geforce RTX 2060    | 7.5                |
| NVIDIA TITAN V      | 7.0                |
| NVIDIA TITAN Xp     | 6.1                |
| NVIDIA TITAN X      | 6.1                |
| GeForce GTX 1080 Ti | 6.1                |
| GeForce GTX 1080    | 6.1                |
| GeForce GTX 1070 Ti | 6.1                |
| GeForce GTX 1070    | 6.1                |
| GeForce GTX 1060    | 6.1                |
| GeForce GTX 1050    | 6.1                |
| GeForce GTX TITAN X | 5.2                |
| GeForce GTX TITAN Z | 3.5                |

# Compute Capability

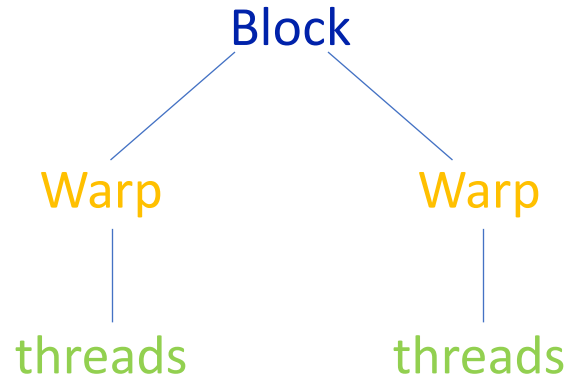
Table 15. Technical Specifications per Compute Capability

|  | Compute Capability |        |       |       |       |      |       |       |       |       |       |        |        |        |
|--|--------------------|--------|-------|-------|-------|------|-------|-------|-------|-------|-------|--------|--------|--------|
| Technical Specifications   | 3.5                | 3.7    | 5.0   | 5.2   | 5.3   | 6.0  | 6.1   | 6.2   | 7.0   | 7.2   | 7.5   | 8.0    | 8.6    | 8.7    |
| Maximum number of resident grids per device<br>(Concurrent Kernel Execution) | 32                 |        |       |       | 16    | 128  | 32    | 16    | 128   | 16    | 128   |        |        |        |
| Maximum dimensionality of grid of thread blocks                              | 3                  |        |       |       |       |      |       |       |       |       |       |        |        |        |
| Maximum x-dimension of a grid of thread blocks                               | 2 <sup>31</sup> -1 |        |       |       |       |      |       |       |       |       |       |        |        |        |
| Maximum y- or z-dimension of a grid of thread blocks                         | 65535              |        |       |       |       |      |       |       |       |       |       |        |        |        |
| Maximum dimensionality of a thread block                                     | 3                  |        |       |       |       |      |       |       |       |       |       |        |        |        |
| Maximum x- or y-dimension of a block   | 1024               |        |       |       |       |      |       |       |       |       |       |        |        |        |
| Maximum z-dimension of a block   | 64                 |        |       |       |       |      |       |       |       |       |       |        |        |        |
| Maximum number of threads per block  | 1024               |        |       |       |       |      |       |       |       |       |       |        |        |        |
| Warp size  | 32                 |        |       |       |       |      |       |       |       |       |       |        |        |        |
| Maximum number of resident blocks per SM                                     | 16                 |        | 32    |       |       |      |       |       |       |       | 16    | 32     | 16     |        |
| Maximum number of resident warps per SM                                      | 64                 |        |       |       |       |      |       |       |       |       | 32    | 64     | 48     |        |
| Maximum number of resident threads per SM                                    | 2048               |        |       |       |       |      |       |       |       |       | 1024  | 2048   | 1536   |        |
| Number of 32-bit registers per SM  | 64 K               | 128 K  | 64 K  |       |       |      |       |       |       |       |       |        |        |        |
| Maximum number of 32-bit registers per thread block                          | 64 K               |        |       |       | 32 K  | 64 K |       | 32 K  | 64 K  |       |       |        |        |        |
| Maximum number of 32-bit registers per thread                                | 255                |        |       |       |       |      |       |       |       |       |       |        |        |        |
| Maximum amount of shared memory per SM                                       | 48 KB              | 112 KB | 64 KB | 96 KB | 64 KB |      | 96 KB | 64 KB | 96 KB |       | 64 KB | 164 KB | 100 KB | 164 KB |
| Maximum amount of shared memory per thread block <sup>33</sup>               | 48 KB              |        |       |       |       |      |       |       | 96 KB | 96 KB | 64 KB | 163 KB | 99 KB  | 163 KB |


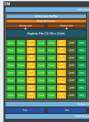
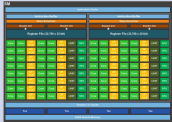
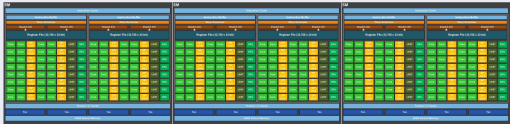
# Streaming Multiprocessor (SM)



# Streaming Multiprocessor (SM)



# Terminology

| Concept | Hardware  | Description  |
|---------|---|--|
| thread  |  | Each thread is executed by one core.<br>Cores have multiple threads resident at one time.<br>Only one thread is executing at one time.                         |
| warp    |  | A warp consists of 32 cores that share register memory and operate on the same instruction currently by construction.  |
| block   |  | Each block is assigned to a SM (streaming multiprocessor), consisting of warps. Each SM shares 64 KB of memory that can be used to share data between threads. |
| grid    |  | The kernel is run on a grid of blocks.   |

$$F = G \frac{m_1 m_2}{d^2}$$

$$\phi(x) = \frac{1}{\sqrt{2\pi\sigma}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

$$i\hbar \frac{\partial}{\partial t} \psi = \hat{H} \psi$$

Some advanced topics

$$\frac{\partial^2 u}{\partial t^2} = c^2 \frac{\partial^2 u}{\partial x^2}$$

$$\frac{df}{dt} = \lim_{h \rightarrow 0} \frac{f(t+h) - f(t)}{h}$$

# Synchronising the Host and Device

- Kernel launches are asynchronous
  - The host can continue processing while the kernel is launched and executed
- The host needs to synchronise (wait) for the results to be ready

|                                      |  |
|--------------------------------------|--|
| <code>cudaMemcpy()</code>            | Blocks the host until the copy is complete. The copy begins when all preceding kernels calls have completed. |
| <code>cudaMemcpyAsync()</code>       | As above, but does not block the host. Copies occur after all preceding kernel calls have completed.         |
| <code>cudaDeviceSynchronize()</code> | Waits for all transfers and kernel calls to complete.  |

# Detecting Errors

- The CUDA API calls all return an error code (`cudaError_t`)
  - An error in the API call, or,
  - An error in an earlier asynchronous operation (e.g., a kernel launch)

- You can always retrieve the last error with

```
cudaError_t cudaGetLastError(void);
```

- You can convert the error code into a human friendly string with

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



# Multiple Device Management

- A host may have more than one device and this must be managed explicitly with the following API calls

`cudaGetDeviceCount(int *count)`

`cudaSetDevice(int device)`

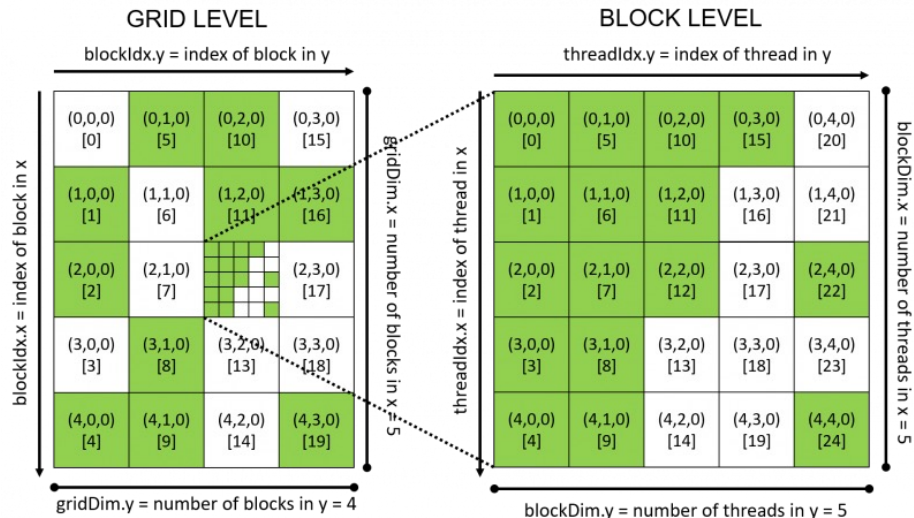
`cudaGetDevice(int *device)`

`cudaGetDeviceProperties(cudaDeviceProp *prop, int device)`

- Multiple host threads can share a device – “thread safe”
- A single host thread can access multiple devices  
`cudaSetDevice(i)` to select the current device for kernel launches and API calls

# 3D Indexing

- A kernel is launched as a grid of blocks of threads.
- blockIdx and threadIdx are 3D
  - We only used “x”
- Builtin variables have x,y and z
  - threadIdx
  - blockDim
  - blockIdx
  - gridDim
- Some problems map well to 3D
- Total threads:  $\text{gridDim.x} \times \text{gridDim.y} \times \text{gridDim.z} \times \text{blockDim.x} \times \text{blockDim.y} \times \text{blockDim.z}$



# Sharing Data between threads

- Within a block, threads can share data via shared memory.
- Extremely fast on-chip memory, user (you) managed.
- Declare using `__shared__`, allocated per block.
- Data is not visible to threads in different blocks

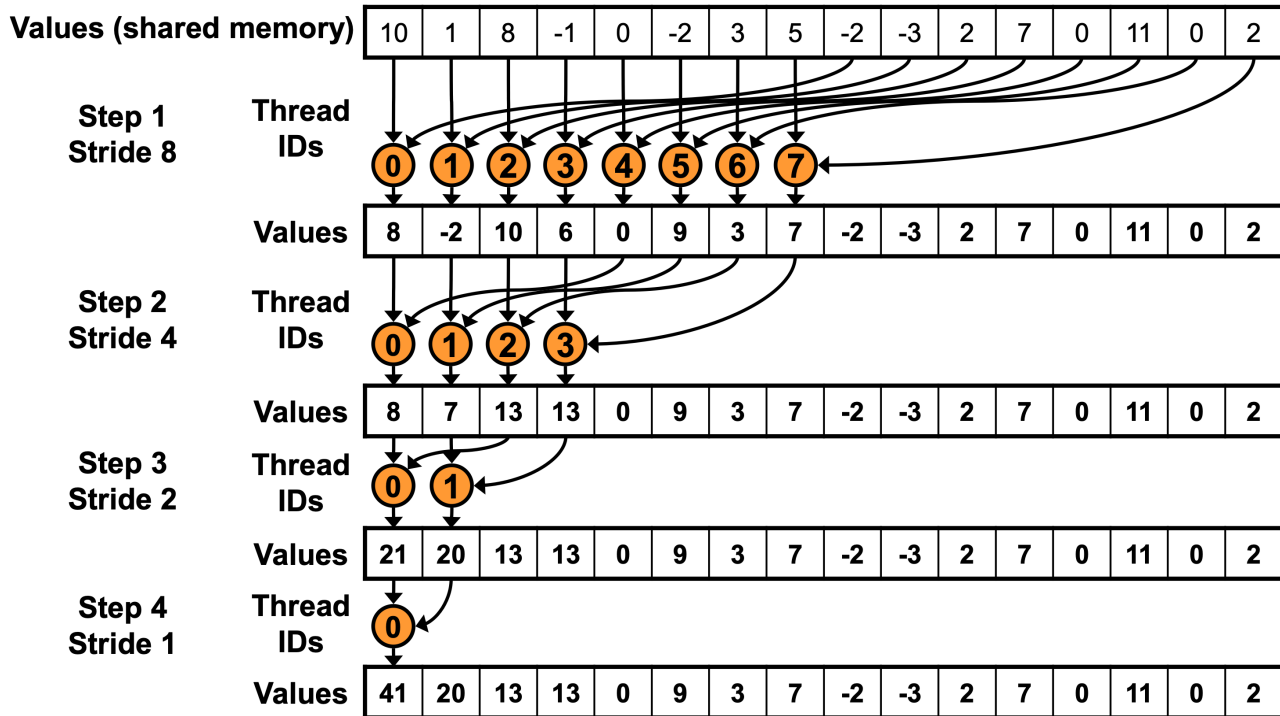
- Threads can execute in any order within a thread block, so to synchronize use

```
void __syncthreads();
```

- This is like a “barrier”; all threads will execute the call before continuing.
- Careful: do not put inside conditional code unless it is uniform across the block.

# Shared Memory Reduction

- See [Optimizing Parallel Reduction in CUDA](#) by nVidia for details.
- Idea is that you can take 32 (or more!) values and reduce them on the GPU
- Each block must be done separately, and the results sent to the CPU (one result per block).



# Importance of Optimising

|  | Time (2 <sup>22</sup> ints) | Bandwidth   | Step Speedup | Cumulative Speedup |
|--|-----------------------------|-------------|--------------|--------------------|
| <b>Kernel 1:</b><br>interleaved addressing<br>with divergent branching | 8.054 ms                    | 2.083 GB/s  |              |                    |
| <b>Kernel 2:</b><br>interleaved addressing<br>with bank conflicts      | 3.456 ms                    | 4.854 GB/s  | 2.33x        | 2.33x              |
| <b>Kernel 3:</b><br>sequential addressing                              | 1.722 ms                    | 9.741 GB/s  | 2.01x        | 4.68x              |
| <b>Kernel 4:</b><br>first add during global load                       | 0.965 ms                    | 17.377 GB/s | 1.78x        | 8.34x              |
| <b>Kernel 5:</b><br>unroll last warp                                   | 0.536 ms                    | 31.289 GB/s | 1.8x         | 15.01x             |
| <b>Kernel 6:</b><br>completely unrolled                                | 0.381 ms                    | 43.996 GB/s | 1.41x        | 21.16x             |
| <b>Kernel 7:</b><br>multiple elements per thread                       | 0.268 ms                    | 62.671 GB/s | 1.42x        | 30.04x             |

- Results are from [Optimizing Parallel Reduction in CUDA](#).
- It is “easy” to write a CUDA kernel.
- Hard to write optimal code.
- Kernels can be limited by
  - Compute (number of FLOPS)
  - Memory bandwidth
- Max for G80: 86.4 GB/s
- Experience and practice!