GPUs have very complicated architectures, both for threading and memory
GPUs have very complicated architectures, both for threading and memory

We shall describe them using CUDA terminology
GPUs have very complicated architectures, both for threading and memory.

We shall describe them using CUDA terminology.

OpenCL has a separate set of words for the same things.
There is a hierarchical management of the threads
There is a hierarchical management of the threads

- A *kernel* is some code running on the *device* (GPU)
There is a hierarchical management of the threads

- A *kernel* is some code running on the *device* (GPU)
- A *grid* is the collection of all threads in a kernel
There is a hierarchical management of the threads

- A *kernel* is some code running on the *device* (GPU)
- A *grid* is the collection of all threads in a kernel
- A grid contains one or more *thread blocks*
Topics
CUDA

There is a hierarchical management of the threads

- A *kernel* is some code running on the *device* (GPU)
- A *grid* is the collection of all threads in a kernel
- A grid contains one or more *thread blocks*
- A thread block contains a number of threads: all blocks in a grid contain the same number of threads
There is a hierarchical management of the threads:

- A *kernel* is some code running on the *device* (GPU).
- A *grid* is the collection of all threads in a kernel.
- A grid contains one or more *thread blocks*.
- A thread block contains a number of threads: all blocks in a grid contain the same number of threads.

All threads in a grid execute the same kernel.
There is a hierarchical management of the threads

- A *kernel* is some code running on the *device* (GPU)
- A *grid* is the collection of all threads in a kernel
- A grid contains one or more *thread blocks*
- A thread block contains a number of threads: all blocks in a grid contain the same number of threads

All threads in a grid execute the same kernel

These are not all SIMD, but are arranged in bunches, called *warps*, of SIMD threads within the blocks
Topics
CUDA

There is a hierarchical management of the threads

- A *kernel* is some code running on the *device* (GPU)
- A *grid* is the collection of all threads in a kernel
- A grid contains one or more *thread blocks*
- A thread block contains a number of threads: all blocks in a grid contain the same number of threads

All threads in a grid execute the same kernel

These are not all SIMD, but are arranged in bunches, called *warps*, of SIMD threads within the blocks

Nvidia calls this “Single Instruction Multiple Thread” (SIMT)
For example, threads 0–31 are in one warp and 32–63 are in another warp
For example, threads 0–31 are in one warp and 32–63 are in another warp

Warps are the basic SIMD chunk
For example, threads 0–31 are in one warp and 32–63 are in another warp.

Warps are the basic SIMD chunk.

This means it is better to gather threads that take the same branches of an if or loop as they will be processed together:

```c
if (threadid < 32) {...} else {...}
```

is better than

```c
if (threadid % 2 == 0) {...} else {...}
```
A block (of multiple warps) is the basic chunk that gets scheduled on a multiprocessor; the multiprocessor then executes the warps.
A block (of multiple warps) is the basic chunk that gets scheduled on a multiprocessor; the multiprocessor then executes the warps.

While threads within a warp are SIMD, threads in separate blocks are independent: a kind of SPMD of SIMD.
A block (of multiple warps) is the basic chunk that gets scheduled on a multiprocessor; the multiprocessor then executes the warps.

While threads within a warp are SIMD, threads in separate blocks are independent: a kind of SPMD of SIMD.

Warps within a block might be independent or SIMD depending on the number of cores per multiprocessor and the number of schedulers per multiprocessor.
A block (of multiple warps) is the basic chunk that gets scheduled on a multiprocessor; the multiprocessor then executes the warps.

While threads within a warp are SIMD, threads in separate blocks are independent: a kind of SPMD of SIMD.

Warps within a block might be independent or SIMD depending on the number of cores per multiprocessor and the number of schedulers per multiprocessor.

Having many warps and many blocks means the system can adapt at runtime to the number of multiprocessors available in the hardware.
A block (of multiple warps) is the basic chunk that gets scheduled on a multiprocessor; the multiprocessor then executes the warps.

While threads within a warp are SIMD, threads in separate blocks are independent: a kind of SPMD of SIMD.

Warps within a block might be independent or SIMD depending on the number of cores per multiprocessor and the number of schedulers per multiprocessor.

Having many warps and many blocks means the system can adapt at runtime to the number of multiprocessors available in the hardware.

Suppose we have 8 blocks in our program kernel.
This naturally and automatically obtains more parallelism when there are more multiprocessors. So it makes sense to have lots more blocks than multiprocessors.
All the blocks in a given grid have the same number of threads
All the blocks in a given grid have the same number of threads.

Blocks are indexed in the grid in one, two or three dimensions (programmer’s choice).
All the blocks in a given grid have the same number of threads.

Blocks are indexed in the grid in one, two or three dimensions (programmer’s choice)

`blockIdx.x` returns the block index for a 1D arrangement.
All the blocks in a given grid have the same number of threads.

Blocks are indexed in the grid in one, two or three dimensions (programmer’s choice).

`blockIdx.x` returns the block index for a 1D arrangement.

`blockIdx.x` and `blockIdx.y` return the block indices for a 2D arrangement.
All the blocks in a given grid have the same number of threads.

Blocks are indexed in the grid in one, two or three dimensions (programmer’s choice).

`blockIdx.x` returns the block index for a 1D arrangement.

`blockIdx.x` and `blockIdx.y` return the block indices for a 2D arrangement.

`blockIdx.x`, `blockIdx.y` and `blockIdx.z` return the block indices for a 3D arrangement.
All the blocks in a given grid have the same number of threads.

Blocks are indexed in the grid in one, two or three dimensions (programmer’s choice).

`blockIdx.x` returns the block index for a 1D arrangement.

`blockIdx.x` and `blockIdx.y` return the block indices for a 2D arrangement.

`blockIdx.x`, `blockIdx.y` and `blockIdx.z` return the block indices for a 3D arrangement.

You specify the size and number of dimensions when creating the grid.
The threads within a block are indexed in one, two or three dimensions
The threads within a block are indexed in one, two or three dimensions

- threadIdx.x
- threadIdx.x, threadIdx.y
- threadIdx.x, threadIdx.y, threadIdx.z
The threads within a block are indexed in one, two or three dimensions

- `threadIdx.x`
- `threadIdx.x, threadIdx.y`
- `threadIdx.x, threadIdx.y, threadIdx.z`

You specify the size and number of dimensions of the blocks when creating the grid
Each thread has its own CPU-style state and registers used in the normal way for function local variables and temporary results; the block has a number of registers (32768, say) which are shared amongst its threads.
Each thread has its own CPU-style state and registers used in the normal way for function local variables and temporary results; the block has a number of registers (32768, say) which are shared amongst its threads.

Each thread has a chunk of local memory (\texttt{local})
Each thread has its own CPU-style state and registers used in the normal way for function local variables and temporary results; the block has a number of registers (32768, say) which are shared amongst its threads.

Each thread has a chunk of local memory (__local__).

This is accessible only by the thread.
Each thread has its own CPU-style state and registers used in the normal way for function local variables and temporary results; the block has a number of registers (32768, say) which are shared amongst its threads.

Each thread has a chunk of local memory (`__local__`) this is accessible only by the thread.

Each block has a chunk of fast shared memory (`__shared__`).
Each thread has its own CPU-style state and registers used in the normal way for function local variables and temporary results; the block has a number of registers (32768, say) which are shared amongst its threads.

Each thread has a chunk of local memory (__local__).

This is accessible only by the thread.

Each block has a chunk of fast shared memory (__shared__).

This is accessible by all the threads in the block and can be used to communicate between threads in a block.
A grid has a big chunk of global shared memory
A grid has a big chunk of global shared memory

This is accessible to all the threads in all the blocks and is the way to communicate between threads in different blocks
A grid has a big chunk of global shared memory

This is accessible to all the threads in all the blocks and is the way to communicate between threads in different blocks

Importantly, access to each of these areas of memory is at radically different speeds
A grid has a big chunk of global shared memory

This is accessible to all the threads in all the blocks and is the way to communicate between threads in different blocks

Importantly, access to each of these areas of memory is at radically different speeds

Access to registers is a bit faster than block shared memory (a few cycles to access); both are much faster than global shared and thread local memory (hundreds of cycles to access)
A grid has a big chunk of global shared memory

This is accessible to all the threads in all the blocks and is the way to communicate between threads in different blocks

Importantly, access to each of these areas of memory is at radically different speeds

Access to registers is a bit faster than block shared memory (a few cycles to access); both are much faster than global shared and thread local memory (hundreds of cycles to access)

So you need to take care on where you place data
A typical CUDA program contains a mix of code to be run on the CPU and code to be run on the GPU.
A typical CUDA program contains a mix of code to be run on the CPU and code to be run on the GPU

This can be in the same source file: GPU kernels are marked by `__global__`
Topics
CUDA

A typical CUDA program contains a mix of code to be run on the CPU and code to be run on the GPU.

This can be in the same source file: GPU kernels are marked by `__global__`

The code is pretty much normal C/C++, but with some restrictions.
A typical CUDA program contains a mix of code to be run on the CPU and code to be run on the GPU. This can be in the same source file: GPU kernels are marked by `__global__`. The code is pretty much normal C/C++, but with some restrictions. Namespaces for variables are a bit tricky, but usually the value of a variable in the CPU is inaccessible to the GPU, and vice versa.
A typical CUDA program contains a mix of code to be run on the CPU and code to be run on the GPU.

This can be in the same source file: GPU kernels are marked by `__global__`

The code is pretty much normal C/C++, but with some restrictions.

Namespaces for variables are a bit tricky, but usually the value of a variable in the CPU is inaccessible to the GPU, and vice versa.

Values are passed across as arguments of CUDA kernel calls; and as explicit cpu-memory-to-gpu-memory copies.
CUDA has dimension types that are used to specify sizes and shapes of grids and blocks.
CUDA has dimension types that are used to specify sizes and shapes of grids and blocks

\[
\text{dim3 } B(w, h, d) \text{ defines } B \text{ to be a 3D } w \times h \times d \text{ object}
\]
CUDA has dimension types that are used to specify sizes and shapes of grids and blocks

\[ \text{dim3 } B(w, h, d) \text{ defines } B \text{ to be a 3D } w \times h \times d \text{ object} \]

\[ \text{dim3 } G(n, m) \text{ defines } G \text{ to be a 2D } n \times m \text{ object} \]
CUDA has dimension types that are used to specify sizes and shapes of grids and blocks

\[
dim3 \ B(w, h, d) \ 	ext{defines} \ B \ 	ext{to be a 3D} \ w \times h \times d \ 	ext{object}
\]

\[
dim3 \ G(n, m) \ 	ext{defines} \ G \ 	ext{to be a 2D} \ n \times m \ 	ext{object}
\]

If \( \text{fun} \) is a kernel (i.e., GPU function), we can call it from the CPU code by

\[
\text{fun}^{<<<G,B>>>}(\text{arg1, arg2, ...});
\]

to run \( \text{fun} \) on a grid containing blocks arranged as \( G \); the blocks containing threads arranged as \( B \)
CUDA has dimension types that are used to specify sizes and shapes of grids and blocks

\[
\text{dim3 } B(w, h, d) \text{ defines } B \text{ to be a 3D } w \times h \times d \text{ object}
\]

\[
\text{dim3 } G(n, m) \text{ defines } G \text{ to be a 2D } n \times m \text{ object}
\]

If \texttt{fun} is a kernel (i.e., GPU function), we can call it from the CPU code by

\[
\texttt{fun}<<\texttt{G,B}>>>(\texttt{arg1}, \texttt{arg2}, \ldots);
\]

to run \texttt{fun} on a grid containing blocks arranged as \texttt{G}; the blocks containing threads arranged as \texttt{B}

\texttt{G} and \texttt{B} are also allowed to be simple integers in the 1D case
This creates $n \times m \times w \times h \times d$ threads, each running $\text{fun}$.
This creates \( n \times m \times w \times h \times d \) threads, each running `fun`.

Each thread is uniquely indexed by `threadIdx` and `blockIdx` and can use these values to decide what to do.
This creates $n \times m \times w \times h \times d$ threads, each running fun.

Each thread is uniquely indexed by threadIdx and blockIdx and can use these values to decide what to do.

You can choose dimensions and sizes of grids and blocks to suit your problem: you should not be shy of 1000s of threads.
This creates $n \times m \times w \times h \times d$ threads, each running fun

Each thread is uniquely indexed by threadIdx and blockIdx and can use these values to decide what to do

You can choose dimensions and sizes of grids and blocks to suit your problem: you should not be shy of 1000s of threads

In fact, one of the issues when writing a CUDA program is figuring how to choose your blocks and distribute your data amongst them
This creates $n \times m \times w \times h \times d$ threads, each running \texttt{fun}.

Each thread is uniquely indexed by \texttt{threadIdx} and \texttt{blockIdx} and can use these values to decide what to do.

You can choose dimensions and sizes of grids and blocks to suit your problem: you should not be shy of 1000s of threads.

In fact, one of the issues when writing a CUDA program is figuring how to choose your blocks and distribute your data amongst them.

For example, the amount of shared memory per block is very limited, so this may affect how you choose blocks.
# Properties of a typical gamer’s card (2020):

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>’GeForce RTX 3080’</td>
</tr>
<tr>
<td>totalGlobalMem</td>
<td>10GB</td>
</tr>
<tr>
<td>maxThreadsPerBlock</td>
<td>1024</td>
</tr>
<tr>
<td>maxRegistersPerBlock</td>
<td>65536</td>
</tr>
<tr>
<td>clockRate</td>
<td>1.44 GHz</td>
</tr>
<tr>
<td>multiProcessorCount</td>
<td>68 processors</td>
</tr>
<tr>
<td>CoreCount</td>
<td>8704 (128 per multiprocessor)</td>
</tr>
<tr>
<td>warp size</td>
<td>32 threads</td>
</tr>
<tr>
<td>processing:</td>
<td>25 TFlop single</td>
</tr>
<tr>
<td></td>
<td>783 GFlop double (1/32)</td>
</tr>
<tr>
<td>power</td>
<td>320W</td>
</tr>
</tbody>
</table>
Properties of a compute oriented GPU card (Balena, 2015):

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>'GeForce GTX K20X'</td>
</tr>
<tr>
<td>totalGlobalMem</td>
<td>6039339008</td>
</tr>
<tr>
<td>sharedMemPerBlock</td>
<td>49152</td>
</tr>
<tr>
<td>maxThreadsPerBlock</td>
<td>1024</td>
</tr>
<tr>
<td>maxRegistersPerBlock</td>
<td>65536</td>
</tr>
<tr>
<td>maxThreadsDim</td>
<td>1024 x 1024 x 64</td>
</tr>
<tr>
<td>maxGridSize</td>
<td>2147483647 x 65535 x 65535</td>
</tr>
<tr>
<td>clockRate</td>
<td>0.73 GHz</td>
</tr>
<tr>
<td>multiProcessorCount</td>
<td>14 processors</td>
</tr>
<tr>
<td>CoreCount</td>
<td>2688 (192 per multiprocessor)</td>
</tr>
<tr>
<td>warp size</td>
<td>32 threads</td>
</tr>
<tr>
<td>processing:</td>
<td>3935 GFlop single</td>
</tr>
<tr>
<td></td>
<td>1310 GFlop double (1/3)</td>
</tr>
<tr>
<td>power</td>
<td>235W</td>
</tr>
</tbody>
</table>
Topics

December 2017: Nvidia Titan V

<table>
<thead>
<tr>
<th>Feature</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA Cores</td>
<td>5120</td>
</tr>
<tr>
<td>Tensor Cores</td>
<td>640</td>
</tr>
<tr>
<td>Transistors</td>
<td>21.1 billion</td>
</tr>
<tr>
<td>Power</td>
<td>250W</td>
</tr>
<tr>
<td>Single precision</td>
<td>12.4 TFLOPS</td>
</tr>
<tr>
<td>Double precision</td>
<td>6.1 TFLOPS</td>
</tr>
<tr>
<td>Half precision</td>
<td>24.6 TFLOPS</td>
</tr>
</tbody>
</table>

Half precision they call “deep learning FLOPS”

Tensor cores are specialised to $4 \times 4$ matrix half-precision fused multiply add ($AB + C$) computations, also for AI.
The main point of GPUs is they have a large number of cores: the RTX 3080 above has 8704 cores in 68 multiprocessors
So there is a lot of global memory, but this is substantially slower (100s of cycles to access) than the block shared memory (maybe 2 cycles)
So there is a lot of global memory, but this is substantially slower (100s of cycles to access) than the block shared memory (maybe 2 cycles)

Though modern GPUs do cache global shared memory: access time is a couple of cycles for a cache hit (though the cache is of limited size, of course)
So there is a lot of global memory, but this is substantially slower (100s of cycles to access) than the block shared memory (maybe 2 cycles)

Though modern GPUs do cache global shared memory: access time is a couple of cycles for a cache hit (though the cache is of limited size, of course)

There is also a chunk of global \textit{constant} memory (\_\_constant\_\_), which is read-only but faster to access than the read-write global memory
So there is a lot of global memory, but this is substantially slower (100s of cycles to access) than the block shared memory (maybe 2 cycles)

Though modern GPUs do cache global shared memory: access time is a couple of cycles for a cache hit (though the cache is of limited size, of course)

There is also a chunk of global constant memory (\texttt{__constant__}), which is read-only but faster to access than the read-write global memory

And some read-only texture memory, whose development arose from the needs of graphics
Constant memory is actually a different way of accessing global memory, but the mechanism limits the amount of constant memory available, e.g., to 64K bytes
Constant memory is actually a different way of accessing global memory, but the mechanism limits the amount of constant memory available, e.g., to 64K bytes.

Similarly, texture memory is global memory accessed in a strange way, via a *texture reference* object.
Constant memory is actually a different way of accessing global memory, but the mechanism limits the amount of constant memory available, e.g., to 64K bytes.

Similarly texture memory is global memory accessed in a strange way, via a *texture reference* object.

A texture reference can be associated with an area of global memory and then that memory is read via the reference.
The weird stuff:

• the index into the texture memory is a floating point number: the value at index 3.14142, say, is interpolated appropriately by the hardware between the values for indices 3 and 4

• the index can be normalised to the interval 0.0 to 1.0. Then the index 0.5 corresponds to the index half-way along the array

• this can be done for 1, 2 or 3 dimensional arrays

It is possible to ignore the clever stuff and just use textures as a fast(er) way to read global memory
Topics
CUDA

The weird stuff:

- the index into the texture memory is a floating point number: the value at index 3.14142, say, is interpolated appropriately by the hardware between the values for indices 3 and 4

• the index can be normalised to the interval 0.0 to 1.0. Then the index 0.5 corresponds to the index half-way along the array

• this can be done for 1, 2 or 3 dimensional arrays

It is possible to ignore the clever stuff and just use textures as a fast(er) way to read global memory
The weird stuff:

- the index into the texture memory is a floating point number: the value at index 3.14142, say, is interpolated appropriately by the hardware between the values for indices 3 and 4
- the index can be *normalised* to the interval 0.0 to 1.0. Then the index 0.5 corresponds to the index half-way along the array
The weird stuff:

- the index into the texture memory is a floating point number: the value at index 3.14142, say, is interpolated appropriately by the hardware between the values for indices 3 and 4
- the index can be *normalised* to the interval 0.0 to 1.0. Then the index 0.5 corresponds to the index half-way along the array
- this can be done for 1, 2 or 3 dimensional arrays
The weird stuff:

- the index into the texture memory is a floating point number: the value at index 3.14142, say, is interpolated appropriately by the hardware between the values for indices 3 and 4
- the index can be *normalised* to the interval 0.0 to 1.0. Then the index 0.5 corresponds to the index half-way along the array
- this can be done for 1, 2 or 3 dimensional arrays

It is possible to ignore the clever stuff and just use textures as a fast(er) way to read global memory
<table>
<thead>
<tr>
<th>Topics</th>
<th>Speed</th>
<th>Access</th>
<th>Scope</th>
<th>Size</th>
<th>Lifetime</th>
</tr>
</thead>
<tbody>
<tr>
<td>register</td>
<td>v fast</td>
<td>r/w</td>
<td>thread</td>
<td>10s</td>
<td>thread</td>
</tr>
<tr>
<td>local</td>
<td>slow</td>
<td>r/w</td>
<td>thread</td>
<td>GBs</td>
<td>thread</td>
</tr>
<tr>
<td>shared</td>
<td>fast</td>
<td>r/w</td>
<td>block</td>
<td>KBs</td>
<td>block</td>
</tr>
<tr>
<td>global</td>
<td>slow</td>
<td>r/w</td>
<td>grid</td>
<td>GBs</td>
<td>application</td>
</tr>
<tr>
<td>constant</td>
<td>cached</td>
<td>r</td>
<td>grid</td>
<td>KBs</td>
<td>application</td>
</tr>
<tr>
<td>texture</td>
<td>cached</td>
<td>r</td>
<td>grid</td>
<td>KBs</td>
<td>application</td>
</tr>
</tbody>
</table>

N.B. the thread, block and grid/kernel lifetimes are typically all the same; a typical application will have many kernel calls