Topics: GPUs

Graphics co-processors have grown immensely in power in the last few years

Topics: GPUs

Graphics co-processors have grown immensely in power in the last few years

Originally intended to offload graphical work from the main CPU they have become recognised as powerful processors in their own right and people have tried to tap into their potential

Topics: GPUs

Graphics co-processors have grown immensely in power in the last few years

Originally intended to offload graphical work from the main CPU they have become recognised as powerful processors in their own right and people have tried to tap into their potential

General-Purpose computing on Graphics Processing Units (GPGPU) has emerged as an important example of parallel processing

Topics: GPUs

Graphics co-processors have grown immensely in power in the last few years

Originally intended to offload graphical work from the main CPU they have become recognised as powerful processors in their own right and people have tried to tap into their potential

General-Purpose computing on Graphics Processing Units (GPGPU) has emerged as an important example of parallel processing

So hardware, originally intended to support gamers, is now being used in general purpose computations

Topics: GPUs

Graphics co-processors have grown immensely in power in the last few years

Originally intended to offload graphical work from the main CPU they have become recognised as powerful processors in their own right and people have tried to tap into their potential

General-Purpose computing on Graphics Processing Units (GPGPU) has emerged as an important example of parallel processing

So hardware, originally intended to support gamers, is now being used in general purpose computations

GPU-based computing appears strongly in the Top 500 largest computers in the world

GPUs

GPUs naturally do certain things very well: in particular data-parallel pixel rendering (colouring, shading and so on)

GPUs

GPUs naturally do certain things very well: in particular data-parallel pixel rendering (colouring, shading and so on)

The computations you typically do on pixels can be quite intensive, but are fairly restricted in nature

GPUs

GPUs naturally do certain things very well: in particular data-parallel pixel rendering (colouring, shading and so on)

The computations you typically do on pixels can be quite intensive, but are fairly restricted in nature

And the data-parallel nature of the computations on the millions of pixels on your screen is very relevant

GPUs

GPUs naturally do certain things very well: in particular data-parallel pixel rendering (colouring, shading and so on)

The computations you typically do on pixels can be quite intensive, but are fairly restricted in nature

And the data-parallel nature of the computations on the millions of pixels on your screen is very relevant

Over time GPUs became more and more programmable as they needed to do more and more complex manipulations

GPUs

GPUs naturally do certain things very well: in particular data-parallel pixel rendering (colouring, shading and so on)

The computations you typically do on pixels can be quite intensive, but are fairly restricted in nature

And the data-parallel nature of the computations on the millions of pixels on your screen is very relevant

Over time GPUs became more and more programmable as they needed to do more and more complex manipulations

Graphics libraries (like OpenGL and DirectX) that were originally developed to draw pictures eventually supported programmable sequences of operations via *shader languages* such as GLSL and HLSL (aka Cg)

GPUs

So people soon realised that GPUs are powerful multicore SIMD processors, but just tuned for certain intensive data-parallel computations

GPUs

So people soon realised that GPUs are powerful multicore SIMD processors, but just tuned for certain intensive data-parallel computations

GPU companies like NVIDIA and AMD/ATI have seen the possibilities of using this power and now put hardware into their GPUs specifically to help GPGPU computations

GPUs

So people soon realised that GPUs are powerful multicore SIMD processors, but just tuned for certain intensive data-parallel computations

GPU companies like NVIDIA and AMD/ATI have seen the possibilities of using this power and now put hardware into their GPUs specifically to help GPGPU computations

This means putting in hardware to support generic computation, not just graphics oriented stuff

GPUs

And NVIDIA have also produced a language, *Compute Unified Device Architecture* (CUDA), to aid in the general programming of these devices

GPUs

And NVIDIA have also produced a language, *Compute Unified Device Architecture* (CUDA), to aid in the general programming of these devices

There is also an open standard, *Open Computing Language* (OpenCL), that is not vendor based

GPUs

And NVIDIA have also produced a language, *Compute Unified Device Architecture* (CUDA), to aid in the general programming of these devices

There is also an open standard, *Open Computing Language* (OpenCL), that is not vendor based

CUDA is quite popular right now, but only runs on NVIDIA cards

GPUs

And NVIDIA have also produced a language, *Compute Unified Device Architecture* (CUDA), to aid in the general programming of these devices

There is also an open standard, *Open Computing Language* (OpenCL), that is not vendor based

CUDA is quite popular right now, but only runs on NVIDIA cards

OpenCL is strong, and is supported by NVIDIA, AMD, Intel and ARM amongst others

GPUs
CUDA

CUDA looks a lot like C and C++

GPUs
CUDA

CUDA looks a lot like C and C++

Dangerously close, as there are several important differences between CUDA and these languages

GPUs

CUDA

CUDA looks a lot like C and C++

Dangerously close, as there are several important differences between CUDA and these languages

CUDA is a modified C/C++ with a *syntactic* addition to notate parallel execution and various semantic additions to support parallelism

GPUs

CUDA

CUDA looks a lot like C and C++

Dangerously close, as there are several important differences between CUDA and these languages

CUDA is a modified C/C++ with a *syntactic* addition to notate parallel execution and various semantic additions to support parallelism

It requires a special compiler, provided by Nvidia

GPUs

CUDA

CUDA looks a lot like C and C++

Dangerously close, as there are several important differences between CUDA and these languages

CUDA is a modified C/C++ with a *syntactic* addition to notate parallel execution and various semantic additions to support parallelism

It requires a special compiler, provided by Nvidia

In contrast, OpenCL is a library that runs on plain C or C++ (and any other language that can call C functions)

GPUs
Architecture

The language reflects the hardware architecture

GPUs

Architecture

The language reflects the hardware architecture

A GPU has several *multiprocessors* each containing a bunch of SIMD cores: thus a GPU is a MIMD of SIMD

GPUs

Architecture

The language reflects the hardware architecture

A GPU has several *multiprocessors* each containing a bunch of SIMD cores: thus a GPU is a MIMD of SIMD

It works best when there are thousands of threads, even if there are only hundreds of cores

GPUs

Architecture

The language reflects the hardware architecture

A GPU has several *multiprocessors* each containing a bunch of SIMD cores: thus a GPU is a MIMD of SIMD

It works best when there are thousands of threads, even if there are only hundreds of cores

This is to overlap communications with computation: a core that would be waiting for some data can pick up another thread and work on it instead on doing nothing

GPUs

Architecture

The language reflects the hardware architecture

A GPU has several *multiprocessors* each containing a bunch of SIMD cores: thus a GPU is a MIMD of SIMD

It works best when there are thousands of threads, even if there are only hundreds of cores

This is to overlap communications with computation: a core that would be waiting for some data can pick up another thread and work on it instead on doing nothing

Memory access in GPUs is relatively **very slow**, so there would be a lot of waiting otherwise

GPUs

Architecture

Threads in a GPU are hardware managed and extremely lightweight, meaning they have tiny creation and scheduling overhead

GPUs

Architecture

Threads in a GPU are hardware managed and extremely lightweight, meaning they have tiny creation and scheduling overhead

Thus there is no need to worry about making and destroying large numbers of threads

GPUs

Architecture

Threads in a GPU are hardware managed and extremely lightweight, meaning they have tiny creation and scheduling overhead

Thus there is no need to worry about making and destroying large numbers of threads

Very different from normal CPU threads

GPUs

Architecture

Threads in a GPU are hardware managed and extremely lightweight, meaning they have tiny creation and scheduling overhead

Thus there is no need to worry about making and destroying large numbers of threads

Very different from normal CPU threads

Exercise Why don't normal CPUs do the same: have hardware support for threads?

GPUs

Architecture

GPUs have very complicated architectures, both for threading and memory

GPUs

Architecture

GPUs have very complicated architectures, both for threading and memory

We shall describe them using CUDA terminology

GPUs

Architecture

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

GPUs

CUDA

There is a hierarchical management of the threads

GPUs

CUDA

There is a hierarchical management of the threads

- A *kernel* is some code running on the *device* (GPU)

GPUs

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

GPUs

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*

GPUs

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

GPUs
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

GPUs

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

GPUs

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)

GPUs
CUDA

For example, threads 0–31 are in one warp and 32–63 are in another warp

GPUs
CUDA

For example, threads 0–31 are in one warp and 32–63 are in another warp

Warps are the basic SIMD chunk

GPUs

CUDA

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:

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

is better than

```
if (threadid % 2 == 0) {...} else {...}
```

GPUs
CUDA

A block (of multiple warps) is the basic chunk that gets scheduled on a multiprocessor; the multiprocessor then executes the warps, as many as it can at a time as the hardware permits

GPUs
CUDA

A block (of multiple warps) is the basic chunk that gets scheduled on a multiprocessor; the multiprocessor then executes the warps, as many as it can at a time as the hardware permits

While threads within a warp are SIMD, separate blocks of threads might be executed at different times: a kind of SPMD of SIMD, though the SPMD nature is generally not really usable

GPUs
CUDA

A block (of multiple warps) is the basic chunk that gets scheduled on a multiprocessor; the multiprocessor then executes the warps, as many as it can at a time as the hardware permits

While threads within a warp are SIMD, separate blocks of threads might be executed at different times: a kind of SPMD of SIMD, though the SPMD nature is generally not really usable

Warps within a block might be executed at the same time or at different times 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

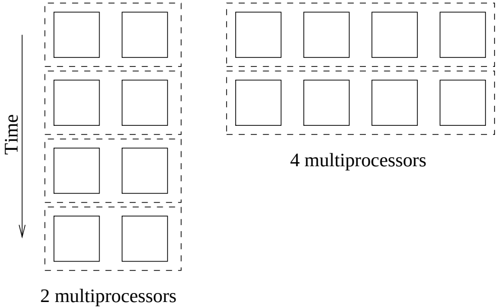
GPUs
CUDA

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 grid

GPUs

CUDA



Processing CUDA blocks

This naturally and automatically obtains more parallelism when there are more multiprocessors. So it makes sense to have lots more blocks than multiprocessors

GPUs
CUDA

All the blocks in a given grid have the same number of threads

GPUs

CUDA

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)

GPUs

CUDA

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

GPUs

CUDA

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

GPUs
CUDA

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

GPUs
CUDA

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

GPUs
CUDA

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`

GPUs

CUDA

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

GPUs

CUDA

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

GPUs

CUDA

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

Each thread has a chunk of **slow** local memory (`__local__`)

GPUs

CUDA

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

Each thread has a chunk of **slow** local memory (`__local__`)

This is accessible only by the thread

GPUs

CUDA

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

Each thread has a chunk of **slow** local memory (`__local__`)

This is accessible only by the thread

Registers are what you need to use if you want fast access, but registers are limited in number, and `__local__` memory might be needed if the compiler can't fit the data into registers

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

GPUs

CUDA

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

GPUs
CUDA

A grid has a big chunk of **slow** global shared memory

GPUs

CUDA

A grid has a big chunk of **slow** 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

GPUs

CUDA

A grid has a big chunk of **slow** 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

GPUs
CUDA

A grid has a big chunk of **slow** 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)

GPUs
CUDA

A grid has a big chunk of **slow** 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

GPUs

CUDA

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

GPUs

CUDA

A typical CUDA source 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__`

GPUs
CUDA

A typical CUDA source 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

GPUs
CUDA

A typical CUDA source 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

Note, when executing, code and data on the GPU are separate from code and data on the CPU

GPUs

CUDA

A typical CUDA source 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

Note, when executing, code and data on the GPU are separate from code and data on the CPU

Values are passed from CPU to GPU as arguments of CUDA kernel calls; or 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

`dim3 B(w, h, d)` defines `B` to be a 3D $w \times h \times d$ shape object

GPUs
CUDA

CUDA has dimension types that are used to specify sizes and shapes of grids and blocks

`dim3 B(w, h, d)` defines `B` to be a 3D $w \times h \times d$ shape object

`dim3 G(n, m)` defines `G` to be a 2D $n \times m$ shape object

GPUs
CUDA

CUDA has dimension types that are used to specify sizes and shapes of grids and blocks

`dim3 B(w, h, d)` defines `B` to be a 3D $w \times h \times d$ shape object

`dim3 G(n, m)` defines `G` to be a 2D $n \times m$ shape object

Just use an integer for 1D

GPUs

CUDA

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

```
fun<<<G,B>>>(arg1, arg2, ...);
```

to run `fun` on a grid containing blocks arranged as `G`; the blocks containing threads arranged as `B`

GPUs
CUDA

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

```
fun<<<G,B>>>(arg1, arg2, ...);
```

to run `fun` on a grid containing blocks arranged as `G`; the blocks containing threads arranged as `B`

This creates $n \times m \times w \times h \times d$ threads, each running `fun`

GPUs
CUDA

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

```
fun<<<G,B>>>(arg1, arg2, ...);
```

to run `fun` on a grid containing blocks arranged as `G`; the blocks containing threads arranged as `B`

This creates $n \times m \times w \times h \times d$ threads, each running `fun`

(And copies the code for the kernel to the GPU; copies the argument values to the GPU; starts the GPU scheduler; and so on)

GPUs

CUDA

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

GPUs
CUDA

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

GPUs

CUDA

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

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

For example, the amount of shared memory per block is very limited, so this may affect how you choose blocks

GPUs

Properties of a typical gamer's card (2020):

name	'GeForce RTX 3080'
totalGlobalMem	10GB
maxThreadsPerBlock	1024
maxRegistersPerBlock	65536
clockRate	1.44 GHz
multiProcessorCount	68 processors
CoreCount	8704 (128 per multiprocessor)
warp size	32 threads
processing:	25 TFlop single
	783 GFlop double (1/32)
power	320W

GPUs

Properties of a compute oriented GPU card (2015):

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

GPUs

December 2017: NVIDIA Titan V

CUDA Cores	5120
Tensor Cores	640
Transistors	21.1 billion
Power	250W
Single precision	12.4 TFLOPS
Double precision	6.1 TFLOPS
Half precision	24.6 TFLOPS

Half precision they call “deep learning FLOPS”

Tensor cores are specialised to 4×4 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

GPUs

CUDA

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)

GPUs

CUDA

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)

GPUs

CUDA

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 (`__constant__`), which is read-only but faster to access than the read-write global memory

GPUs
CUDA

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 (`__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 (to make it fast access) 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 (to make it fast access) 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 (to make it fast access) 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

GPUs

CUDA

The weird stuff:

GPUs

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

GPUs

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

GPUs

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

GPUs

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

GPUs

CUDA

	Speed	Access	Scope	Size	Lifetime
register	v fast	r/w	thread	10s	thread
local	slow	r/w	thread	GBs	thread
shared	fast	r/w	block	KBs	block
global	slow	r/w	grid	GBs	application
constant	cached	r	grid	KBs	application
texture	cached	r	grid	KBs	application

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