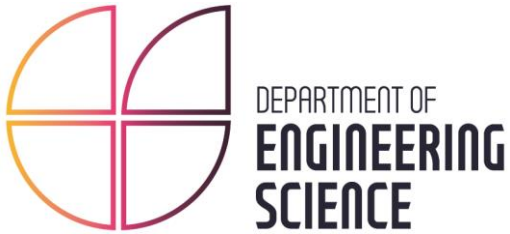


An Introduction to HPC and Scientific Computing

Lecture seven: An introduction to GPUs and how to use them.

Wes Armour

Oxford e-Research Centre,
Department of Engineering Science

Overview

In this lecture you will learn about:

- Different hardware generations of GPUs.
- How GPUs execute code.
- A more detailed look at the differences between GPUs and CPUs
- How GPUs hide memory latencies.
- The GPU software environment and useful tools.
- Finally we will cover the basics of how to use GPU libraries in your codes.

Hardware overview

CUDA capable hardware

NVIDIA has produced six generations of GPU architectures that are capable of executing CUDA code.

Each generation has a different *compute capability.* Typically as the compute capability increases the functionality of the card increases.

Over the six generations memory capacity has increased by four times, bandwidth to main memory by nine times, but compute throughput by 14 times.

Hardware generation	Card	# CUDA cores	Main memory (GB)	Bandwidth (GB/s)	Peak compute performance (Tflops)	CUDA compute capability
Tesla	C1060	240	4	103	0.93	1.3
Fermi	M2090	512	6	178	1.3	2
Kepler	K40	2880	12	288	4.2	3.5
Maxwell	M40	3072	12	288	5.8	5.2
Pascal	P100	3584	16	732	8.1	6
Volta	V100	5120	16	900	14	7

https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#compute-capabilities

CUDA capable hardware

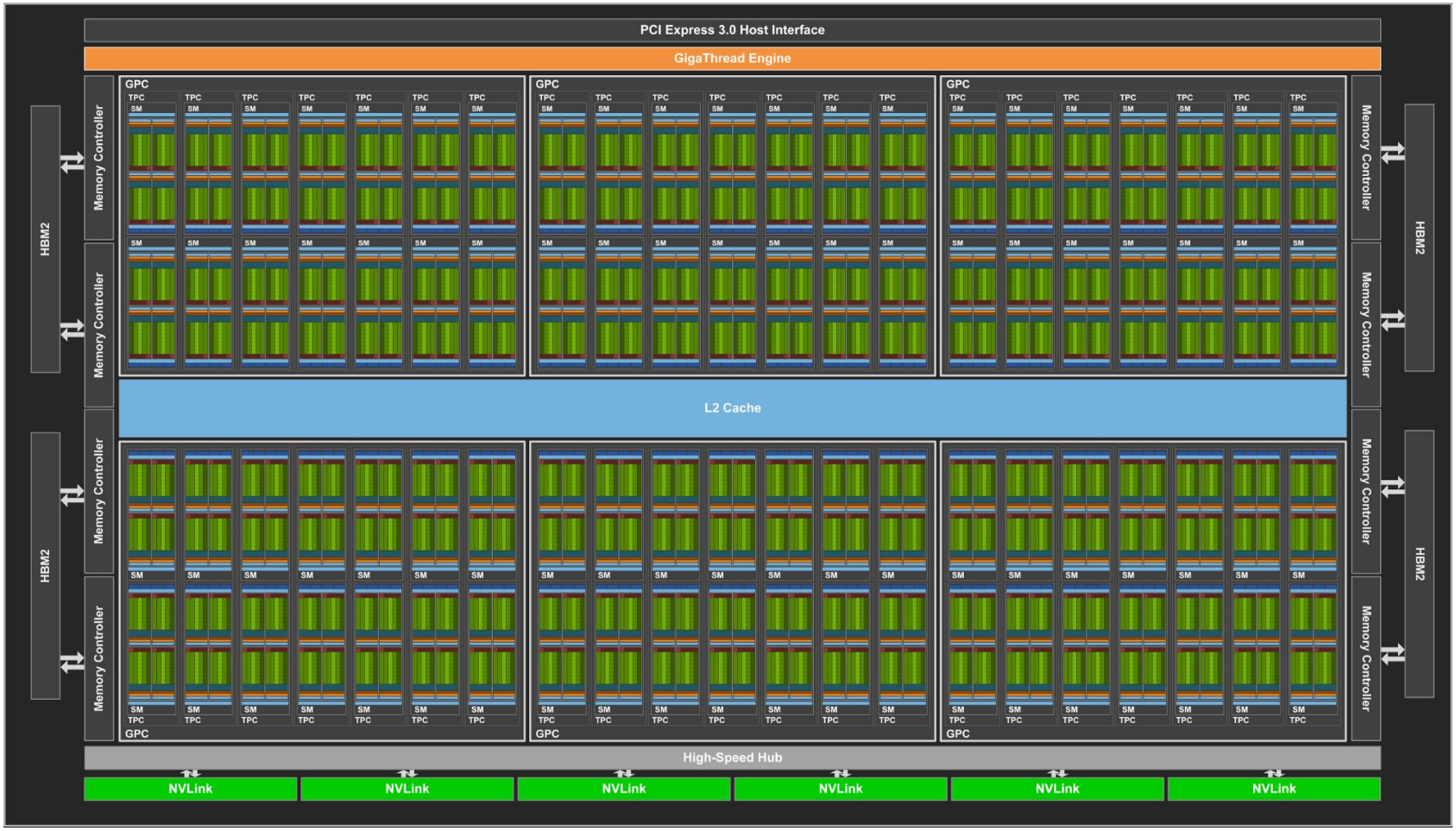
Each generation has slightly different technical specifications, for example the maximum number of thread blocks per SM (multiprocessor). These specifications are often what limits the eventual performance of your GPU code.

Technical Specifications	Compute Capability										
	3	3.2	3.5	3.7	5	5.2	5.3	6	6.1	6.2	7
Maximum dimensionality of grid of thread blocks	3										
Maximum x-dimension of a grid of thread blocks	$2^{31}-1$										
Maximum y- or z-dimension of a grid of thread blocks	65535										
Maximum dimensionality of 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 multiprocessor	16				32						
Maximum number of resident warps per multiprocessor	64										
Maximum number of resident threads per multiprocessor	2048										
Number of 32-bit registers per multiprocessor	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		32 K	64 K
Maximum number of 32-bit registers per thread	63	255									
Maximum amount of shared memory per multiprocessor	48 KB			112 KB	64 KB	96 KB	64 KB		96 KB	64 KB	96 KB
Maximum amount of shared memory per thread block	48 KB										96 KB
Number of shared memory banks	32										

UNIVERSITY OF OXFORD

Oxford e-Research Centre

NVIDIA Volta V100



UNIVERSITY OF OXFORD

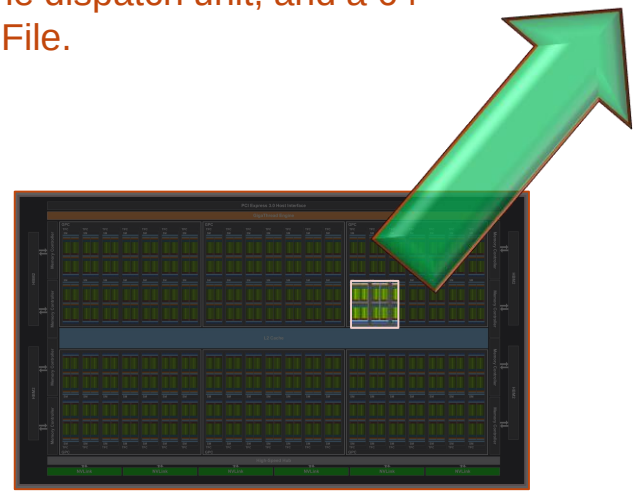
Oxford e-Research Centre

NVIDIA Volta V100 – Streaming Multiprocessor (SM)

A V100 has 80 SMs (see right).

The GV100 SM incorporates 64 FP32 cores and 32 FP64 cores per SM.

The SM is partitioned into four processing blocks, each with 16 FP32 Cores, 8 FP64 Cores, 16 INT32 Cores, a new L0 instruction cache, one warp scheduler, one dispatch unit, and a 64 KB Register File.



NVIDIA Volta V100 - SM

Lets look at this in more detail

We've talked about a Streaming Multiprocessor, this is a collection of hardware that our thread block executes on.

A FP32 core is the execution unit that performs single precision floating point arithmetic (floats).

A FP64 core performs double precision arithmetic (doubles).

NVIDIA Volta V100 - SM

Lets look at this in more detail

A INT32 Core performs integer arithmetic

The warp scheduler selects which warp (group of 32 threads) to send to which execution units.

64 KB Register File – lots of transistors used for vary fast memory.

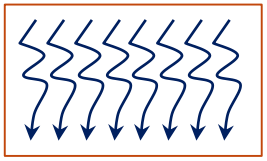
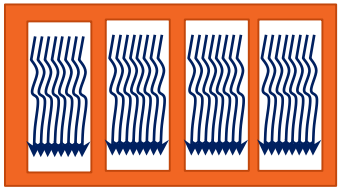
128KB of configurable L1 (data) cache or shared memory

Shared memory is a used manged cache.

LD/ST units load and store data from cores

SFU – special function units compute things like transcendentals.

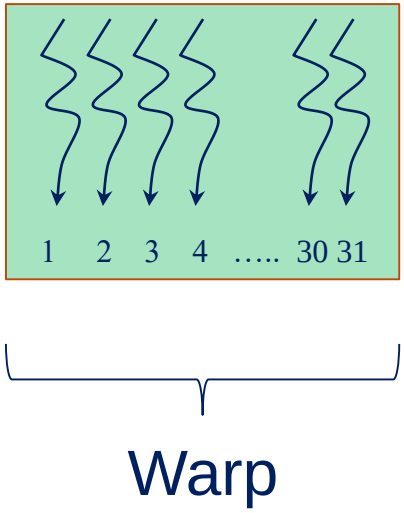
Code execution model

	Software	Hardware	
A thread executes on a core.	Thread			Core
A group of threads (thread block) is executed on a SM. Thread blocks don't migrate between SMs. Several concurrent thread blocks can reside on a SM.	Thread Block			SM
A group of thread blocks form a grid and a grid runs on the device. *In CUDA this is a kernel launch.*	Grid			Device

Warps

Code executes on a GPU in groups of threads, the threads are naturally grouped into lots of 32, this is called a warp.

The reason that a warp has 32 elements is that the SIMD (recall lecture one) units on a GPU have 32 “lanes”.



Warps

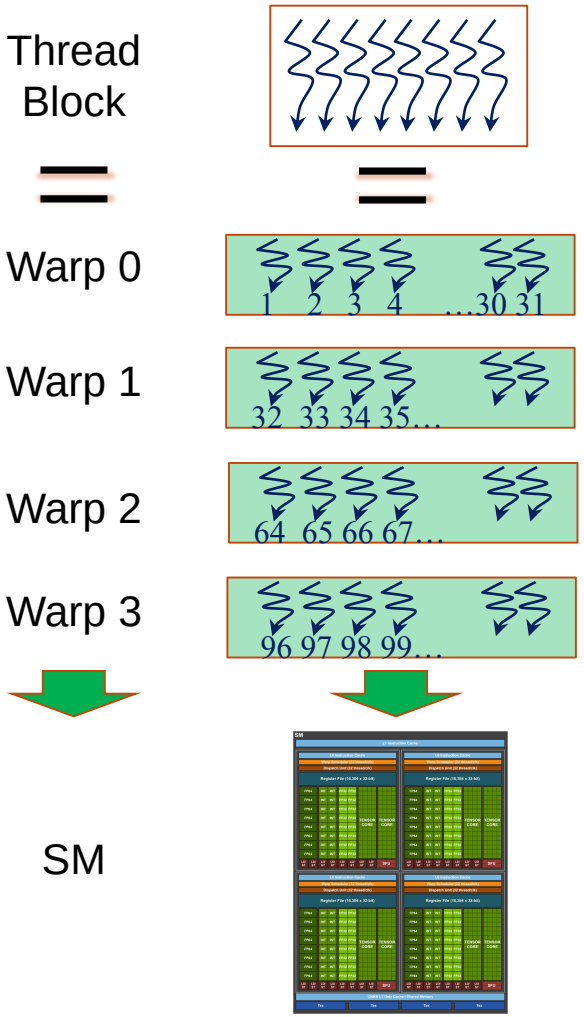
Thread blocks are formed from warps.

The warp is executed in parallel on the SM.

By this we mean that everything that happens within a warp is lock-step.

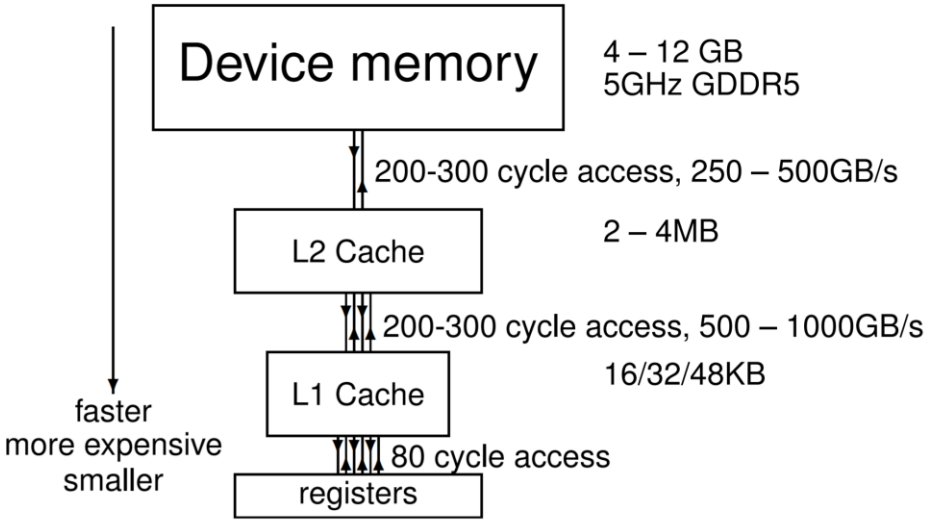
So the same operation (instruction) in thread 2, 3, 8 occurs in thread 7, 1, 4, 11… (from 0 to 31) at the same time.

So we have a Single Instruction Multiple Thread (SIMT) architecture.

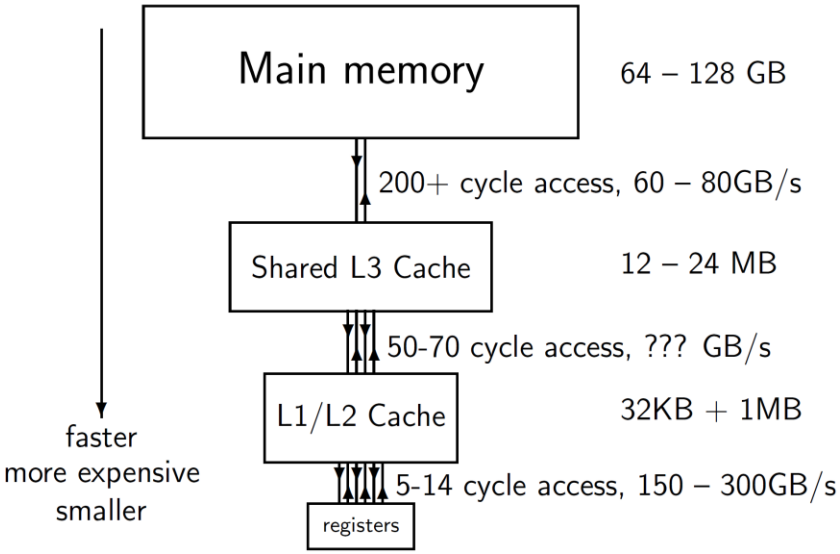


Memory architecture

In lecture one we looked a the memory architecture of CPUs. Lets compare this to that of GPUs. We see that although GPUs have greater bandwidth, they have less cache and (importantly) grater latencies (the time taken to get data from memory to the processing cores).



GPU



CPU

Images courtesy of Mike Giles

Hiding memory latencies - SIMT

CPUs are optimised for low latency access to cached data sets.

Lots of transistors spent on control logic for out-of-order (OoO) and speculative execution.

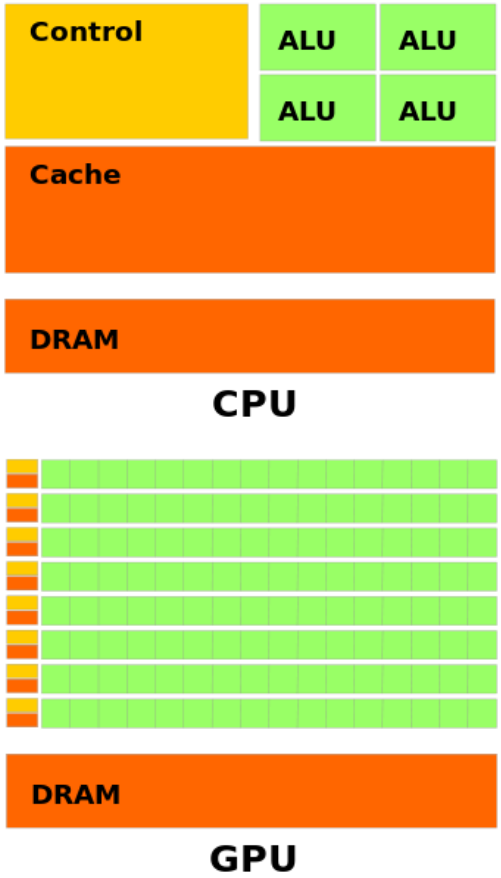
CPUs support 10's of threads.

GPUs are optimised for data parallel high throughput computation.

By having many thousands of memory requests *in flight* at any one time the larger latencies can be hidden.

Lots of transistors dedicated to computation.

GPUs support 1000's of threads.



Hiding memory latencies - SIMT

Lets think about our GPU memory pipe as a conveyor belt.

Every two seconds a container becomes available in which we can put a single piece of data.

Bandwidth is one container per two seconds =
0.5 containers per second

Our conveyor belt has 8 containers on it at any one time.

Latency is 8 containers / 0.5 containers per second =
16 seconds

http://on-demand.gputechconf.com/gtc/2018/presentation/s81006-volta-architecture-and-performance-optimization.pdf

Hiding memory latencies - SIMT

A CPU has only 10's of threads to make memory requests to hide it's memory latencies.

However a GPU has thousands of threads which can all make memory requests and so can hide larger latencies.

A GPU hides the memory latencies of one warp with the computation of another warp

Hiding memory latencies - SIMT

A warp is executed on an SM, up to the point it stalls on a memory request.

The memory request is made and the warp removed.

The warp scheduler then looks to see what data has arrived through the memory pipe and then schedules the warp associated with that data for execution.

Eventually the data arrives for our stalled warp and then this is scheduled for execution.

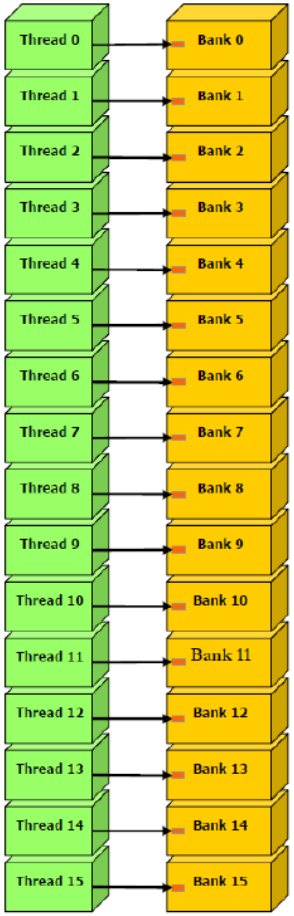
Memory coalescing

Global memory access happens in transactions of 32 or 128 bytes (cache line).

So to achieve peak memory bandwidth we need to use all of the data stored in the cache line.

Coalesced memory access
A warp of threads access adjacent data (see right) in a cache line. In the best case this results in one memory transaction (best bandwidth).

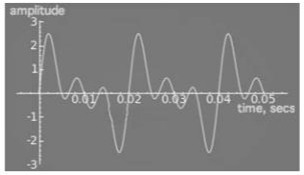
Uncoalesced memory access
A warp of threads access scattered data all in different cache lines.
This would result in 32 different memory transactions (giving poor bandwidth).



Software overview

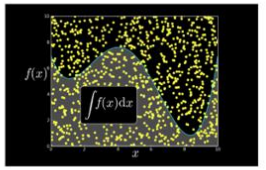
NVIDIA provides a rich ecosystem of software tools that allow you to easily utilise GPUs in your project.

During this course we will focus on the CUDA programming language, we will also look at some of the NVIDIA libraries that can be used to make programming GPUs easier.



cuFFT

GPU-accelerated library for Fast Fourier Transforms



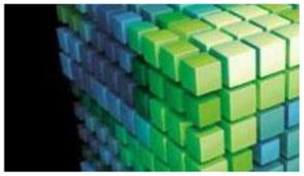
cuRAND

GPU-accelerated random number generation (RNG)

NVIDIA CUDA

CUDA Toolkit

Provides a comprehensive environment for C/C++ developers building GPU-accelerated applications.



cuBLAS

GPU-accelerated standard BLAS library

UNIVERSITY OF OXFORD

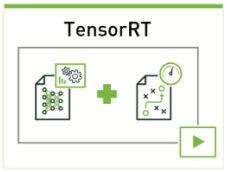
Oxford e-Research Centre

Libraries

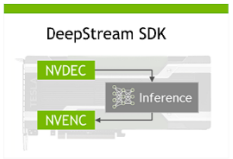
Deep Learning Libraries

cuDNN

GPU-accelerated library of primitives for deep neural networks

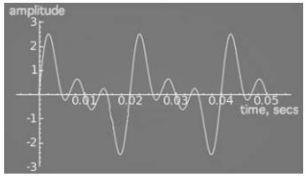


GPU-accelerated neural network inference library for building deep learning applications



Advanced GPU-accelerated video inference library

Signal, Image and Video Libraries



cuFFT

GPU-accelerated library for Fast Fourier Transforms

NVIDIA Performance Primitives

GPU-accelerated library for image and signal processing

NVIDIA Codec SDK

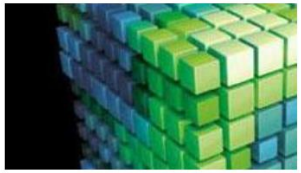
High-performance APIs and tools for hardware accelerated video encode and decode

UNIVERSITY OF OXFORD

Oxford e-Research Centre

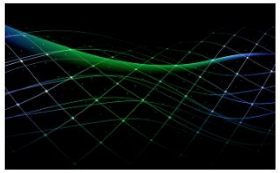
Libraries

Linear Algebra and Math Libraries



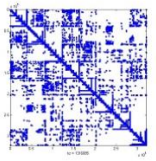
cuBLAS

GPU-accelerated standard BLAS library



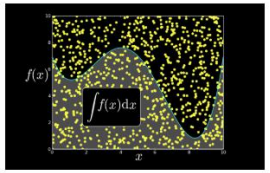
CUDA Math Library

GPU-accelerated standard mathematical function library



cuSPARSE

GPU-accelerated BLAS for sparse matrices



cuRAND

GPU-accelerated random number generation (RNG)

cuSOLVER

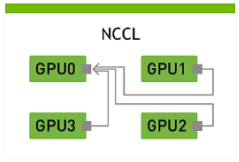
Dense and sparse direct solvers for Computer Vision, CFD, Computational Chemistry, and Linear Optimization applications

AmgX

GPU accelerated linear solvers for simulations and implicit unstructured methods

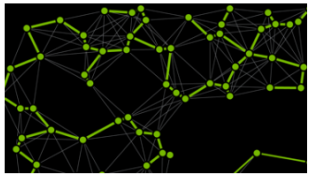
Libraries

Parallel Algorithm Libraries



NCCL

Collective Communications Library for scaling apps across multiple GPUs and nodes



nvGRAPH

GPU-accelerated library for graph analytics

Thrust

GPU-accelerated library of parallel algorithms and data structures

Languages

CUDA Toolkit

Provides a comprehensive environment for C/C++ developers building GPU-accelerated applications.

OpenACC

Directives for parallel computing, is a new open parallel programming standard designed to enable all scientific and technical programmers.

PGI Accelerator Fortran and C Compilers

Accelerate applications on GPU platforms by adding compiler directives to existing code.

The PGI CUDA C/C++ compiler for x86

Compile and optimize their CUDA applications to run on x86-based workstations, servers and clusters.

CUDA FORTRAN

Enjoy GPU acceleration directly from your Fortran program using CUDA Fortran from The Portland Group.

Anaconda Accelerate

Enables acceleration on your GPU or multi-core processor using Python.

PyCUDA

Gives you access to CUDA fuctionality from your Python code.

Altimesh Hybridizer™

An advanced productivity tool that generates vectorized C++ (AVX) and CUDA C code from .NET assemblies (MSIL) or Java archives (bytecode)

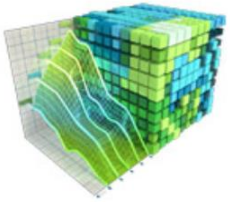
OpenCL™

OpenCL is a low-level API for GPU computing that can run on CUDA-powered GPUs.

Alea GPU

This is a novel approach to develop GPU applications on .NET, combining the CUDA with Microsoft's F#.

Tools



NVIDIA Visual Profiler

This is a cross-platform performance profiling tool that delivers developers vital feedback for optimizing CUDA C/C++ applications. First introduced in 2008, Visual Profiler supports all CUDA capable NVIDIA GPUs shipped since 2006 on Linux, Mac OS X, and Windows.

NVIDIA® Nsight™

The ultimate development platform for heterogeneous computing. Work with powerful debugging and profiling tools, optimize the performance of your CPU and GPU code. Find out about the Ecilipse Edition and the graphics debugging enabled Visual Studio Edition.

CUDA-GDB

Delivers a seamless debugging experience that allows you to debug both the CPU and GPU portions of your application simultaneously. Use CUDA-GDB on Linux or MacOS, from the command line, DDD or EMACS.

CUDA-MEMCHECK

Identifies memory access errors in your GPU code and allows you to locate and resolve problems quickly. CUDA-MEMCHECK also reports runtime execution errors, identifying situations that could result in an "unspecified launch failure" error while your application is running.

CUDA

CUDA (Compute Unified Device Architecture) provides a set of programming extensions based on the C/C++ family of languages.

If you have a basic understanding of C and understand the concept of threads and SIMD execution, then CUDA is easy to pick up.

An introduction to the CUDA programming language will be given in lecture eight.

CUDA Toolkit

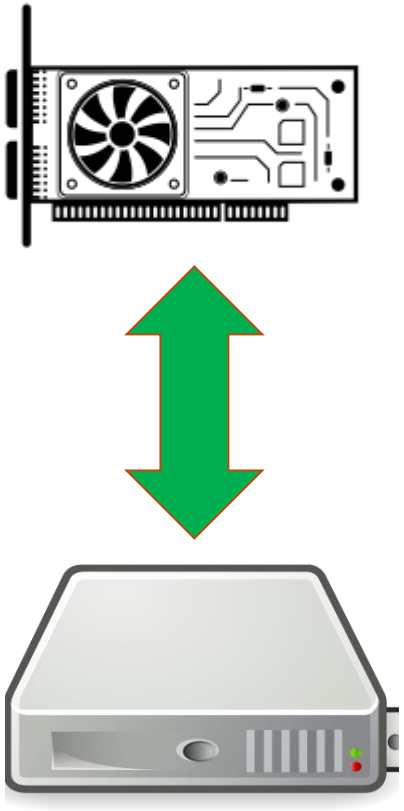
Provides a comprehensive environment for C/C++ developers building GPU-accelerated applications.

Working with GPUs

Recall for our first lecture (and earlier in these slides), a GPU is attached to the computer by a PCIe bus.

It also has it's own memory (for example a V100 has 16GB of HBM2 memory). So the GPU has a separate address space.

This means that for us to work with the GPU we need to allocate memory on the card (device) and then transfer data to and from the device.



Working with GPUs

To allocate memory on the device CUDA provides a function just like `malloc()` called `cudaMalloc()`

To free memory CUDA provides a function like `free()` called `cudaFree()`

Examples are given in the code snippit on the right.

```
// Allocate pointers for host and device memory
float *h_input, *h_output;
float *d_input, *d_output;

// malloc() host memory (this is in your RAM)
h_input  = (float*) malloc(mem_size);
h_output = (float*) malloc(mem_size);

// allocate device memory input and output arrays
cudaMalloc((void**)&d_input,  mem_size);
cudaMalloc((void**)&d_output, mem_size);

// Do something here!

// cleanup memory
free(h_input);
free(h_output);
cudaFree(d_input);
cudaFree(d_output);
```

Working with GPUs

To transfer data to and from the GPU CUDA provides a function to do this called `cudaMemcpy()`

As arguments `cudaMemcpy()` takes:

```
cudaMemcpy(destination pointer, origin pointer, size of memory to move, direction)
```

Examples below

```
// Copy host memory to device input array
cudaMemcpy(d_input, h_input, mem_size, cudaMemcpyHostToDevice);

// Do something on the GPU

// copy result from device to host
cudaMemcpy(h_output, d_output, mem_size, cudaMemcpyDeviceToHost);
```

An example - cuRAND

The example on the right shows how to generate a distribution of random normal number using cuRAND.

This code could replace

// Do something here

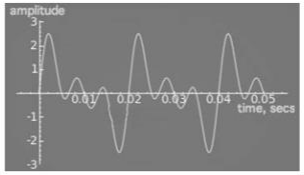
In the previous slides. Combining this will `cudaMemcpy()` allows you to start building a GPU program.

```
// random number generation
#include <cuda.h>
#include <curand.h>

// Declare variable
curandGenerator_t gen;

// Create random number generator
curandCreateGenerator(&gen, CURAND_RNG_PSEUDO_DEFAULT) );

// Set the generator options
curandSetPseudoRandomGeneratorSeed(gen, 1234ULL) );

// Generate the randoms
curandGenerateNormal(gen, d_input, NUM_ELS, 0.0f, 1.0f) );
```

https://docs.nvidia.com/cuda/curand/introduction.html#introduction

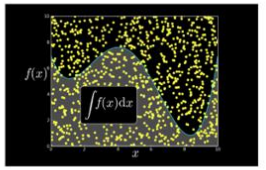
Practical 5

In practical 5 you will work with different codes to familiarise yourselves with cuRAND, cuFFT and cuBLAS.



cuFFT

GPU-accelerated library for Fast Fourier Transforms



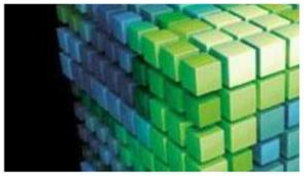
cuRAND

GPU-accelerated random number generation (RNG)

NVIDIA CUDA

CUDA Toolkit

Provides a comprehensive environment for C/C++ developers building GPU-accelerated applications.



cuBLAS

GPU-accelerated standard BLAS library

UNIVERSITY OF OXFORD

Oxford e-Research Centre

What have we learnt?

In this lecture you have learnt about different hardware generations of GPUs and some of there differences.

We've looked at how GPUs execute code.

We have looked at the differences between GPUs and CPUs in greater detail, we have also looked at how GPUs hide memory latencies.

We've looked at the GPU software environment and useful tools.

Finally we have covered the basics of how to use GPU libraries in your codes.

Further reading

In the next lecture…

In the next lecture you will learn about