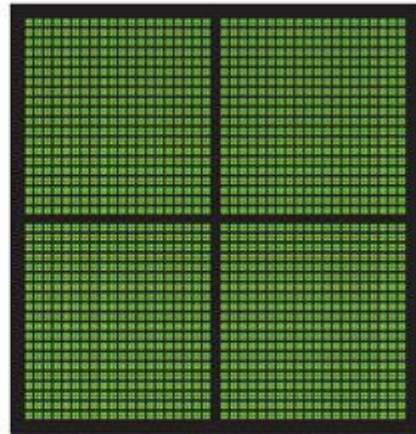


# GPU Tutorial 1: Introduction to GPU Computing



CPU  
MULTIPLE CORES



GPU  
THOUSANDS OF CORES

# New Concepts

- ▶ GPU Computation: Overview
- ▶ CUDA
  - ▶ Hardware
  - ▶ Software
- ▶ Program Flow
- ▶ Paradigms

# A Note on Nomenclature

- ▶ In GPU Computing, it's common to refer to the DEVICE and the HOST
  - ▶ Host means the Motherboard
    - ▶ Host is the CPU
    - ▶ Host memory is system memory (RAM)
- ▶ Device means the Graphics Card
  - ▶ Device is the GPU
  - ▶ Device memory is graphics memory (VRAM)

# GPU Computation: Overview

# GPU Computation: History

- ▶ Can be charted back to the first discrete graphics processing units.
- ▶ Early solutions employed GLSL to act on data abstracted to pixel data - dummy graphics shell.
- ▶ Per-pixel operations subverted the idea of colour data, and embedded other floating point data, which was operated on as though colours were being changed
- ▶ Often, researchers had to ensure their system rendered something, just to get the execution to complete - “engineering solution”

# GPU Computation: History

- ▶ Began to take off around 2004 in a limited fashion.
- ▶ Verdesca et al some of the earliest adopters - used it to accelerate a simulation for the Pentagon - realised that the GPU could be used to speed up line-of-sight checks between AI agents
- ▶ GPU hardware and API development, though, remained focused on graphics

# GPU Computation: History

- ▶ DirectX 10 - Unified Shader Architectures
- ▶ If you have to go there, you might as well go all the way - General Purpose
- ▶ Identifying issues the GPU would be good at solving
- ▶ NVIDIA CUDA
- ▶ AMD FireStream/“Close-to-Metal”/Stream SDK...
- ▶ One of these is still around...

# CPU Computation: Here and Now

- ▶ Diverged into two areas:
  - ▶ GPU Computing
  - ▶ Heterogeneous Computing
- ▶ From a researcher's perspective, their focus is different
  - ▶ GPU Computing research is using the GPU to accelerate solution of a specific problem
  - ▶ Heterogeneous Computing research is deploying code optimally across multiple different types of core
- ▶ From a game engineer's perspective, they're the same thing
  - ▶ If we're deploying something to the GPU, it's to free up needed CPU cycles/avoid taking CPU cycles
  - ▶ We'll always be using those free CPU cycles for something else - if we weren't, we'd not bother shunting stuff to the GPU, because graphics.



# CPU Computation: Here and Now

- ▶ Multiple contemporary APIs
- ▶ APIs have different focus, depending on purpose (purpose is often commercially driven rather than technologically driven - ‘keeping relevance’)
- ▶ Available APIs:

Name	Ease of Programming	Cross-Platform?	Performance (Guide)
CUDA	High	No (Hardware)	High
OpenCL	Medium	Yes	Medium-High
DirectCompute	Medium	No (Software)	Low
C++ AMP	Highest	No (Software)*	Lowest
GLSL Compute Shaders	Lowest	Yes	Highest

# CPU Computation: Here and Now

- ▶ CUDA vs OpenCL - GPU vs Heterogeneous
- ▶ We cover CUDA for three reasons:
  - ▶ You're learning the principles, not an API
  - ▶ The hardware you have is better suited to feature-full CUDA
  - ▶ CUDA is more accessible to a C++ programmer

Name	Ease of Programming	Cross-Platform?	Performance (Guide)
CUDA	High	No (Hardware)	High
OpenCL	Medium	Yes	Medium-High
DirectCompute	Medium	No (Software)	Low
C++ AMP	Highest	No (Software)*	Lowest
GLSL Compute Shaders	Lowest	Yes	Highest

CUDA



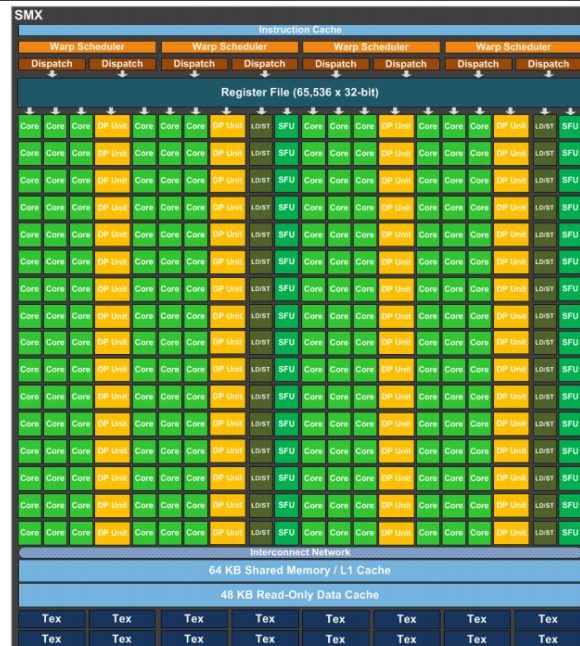
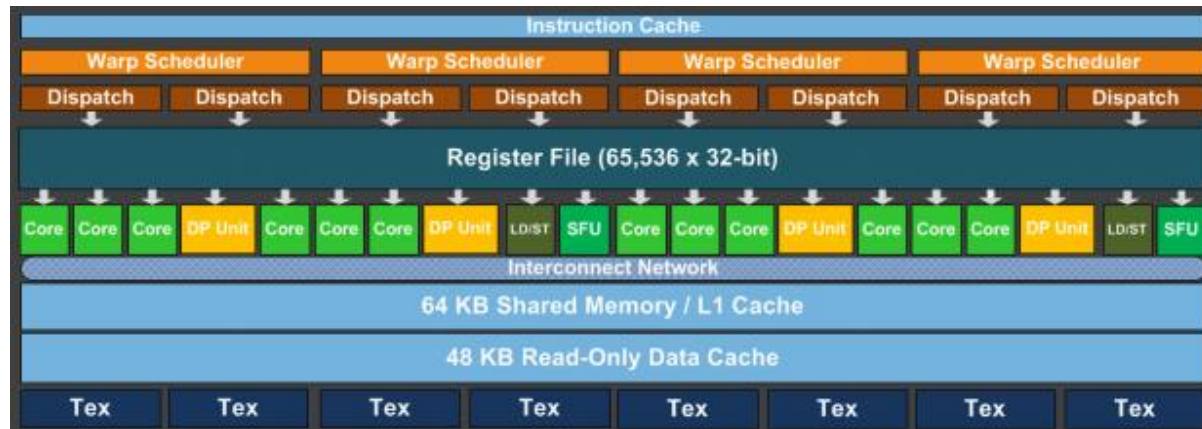
# CUDA: What is it?

- ▶ Compute Unified Device Architecture
- ▶ API for writing functions which execute on the GPU
- ▶ Syntax focused on highly parallel executions
- ▶ Both a Hardware and a Software API

# CUDA Hardware



# CUDA Hardware



# CUDA Hardware

- ▶ The GPU's architecture is completely different to the CPU's
- ▶ One 1.5MB L2 cache pool divided between hundreds of cores
- ▶ Compare that with 256KB per core on an i7-4770K CPU
- ▶ Far more vulnerable to cache misses

# CUDA Hardware

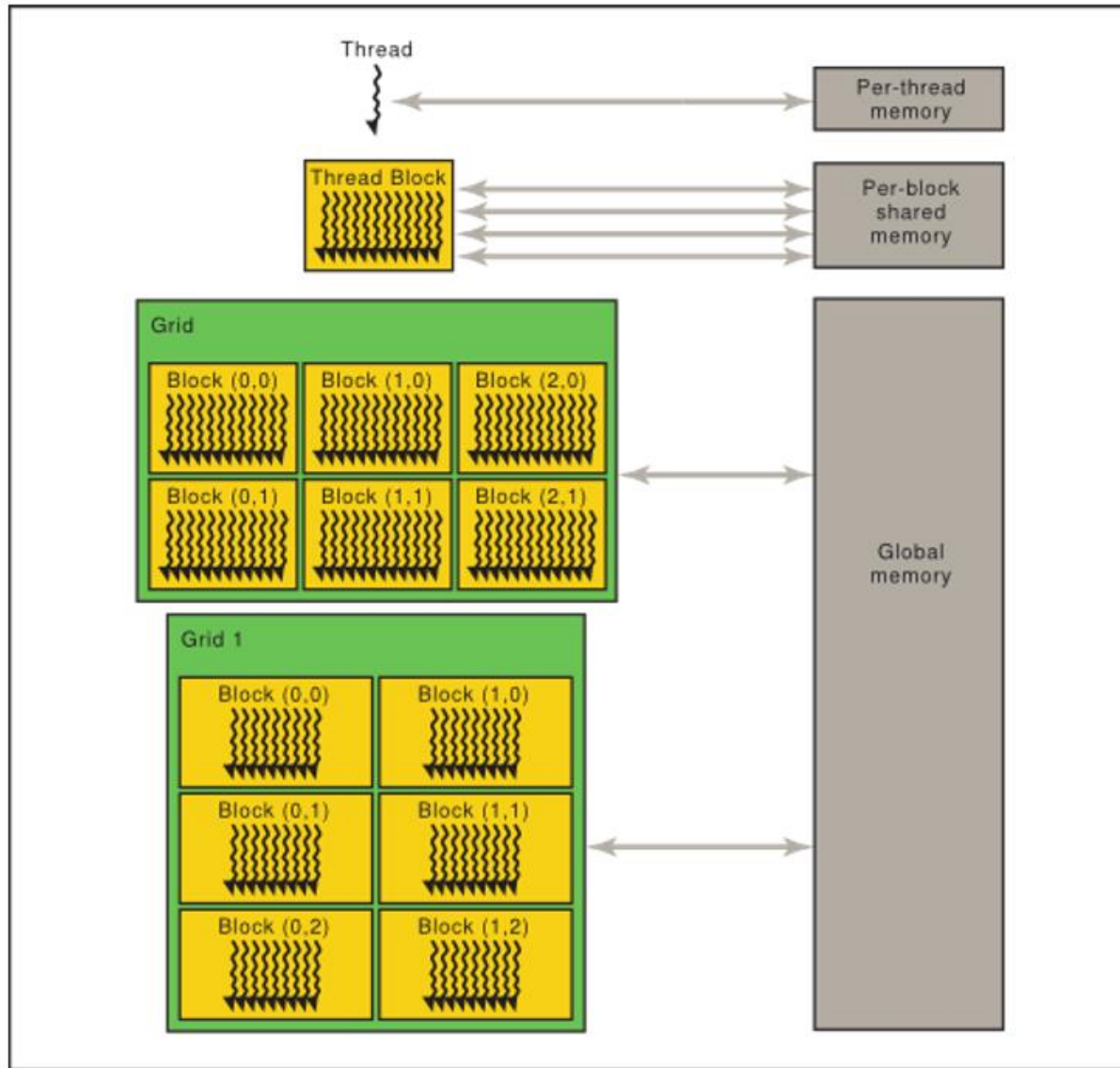
- ▶ These weaknesses make sense, though, given they relate to the GPU's strengths
- ▶ GPUs are designed to execute the same shader thousands of times
- ▶ Shaders are small, simple (in terms of computation) programs
- ▶ Don't need massive cache pools, don't need to communicate between shaders (much), don't need the versatile architecture of the CPU
- ▶ Couldn't build a GPU with the architecture of a CPU



# CUDA Software

- ▶ C/C++ Style
  - ▶ Can use classes/structs
  - ▶ Function calls use similar syntax
  - ▶ Compiles in the same IDE
- ▶ Interface C++ code with .cu files through external functions (extern)
- ▶ Compatible with MSc Hardware

# CUDA Software



# CUDA Software: Thread

- ▶ Instance of a Kernel. Thread with ThreadID 103 accesses array index 103 to obtain its data.
- ▶ Can access per-thread memory
- ▶ Can access per-block shared memory, to communicate with threads in the same block
- ▶ Can access global memory if needed

# CUDA Software: Block

- ▶ Group of threads
- ▶ Block defines which threads can communicate via shared memory
- ▶ All threads in a block notionally execute in parallel
- ▶ But they don't really, if the number of threads in the block exceeds the occupancy of the GPU
- ▶ So communication between specific threads can be problematic

# CUDA Software: Grid

- ▶ Grid is an array of blocks which is triggered by a specific kernel function execution
- ▶ Grid reads inputs from global memory
- ▶ Grid outputs data back to global memory

# CUDA Software: General

- ▶ Kernel function generates a grid, with dimensions of the grid passed in as parameters.
- ▶ Constants are stored in Device's constant memory accessible by all threads
  - ▶ Can't be changed during execution, because constant
- ▶ Arrays cannot be stored in constant memory
  - ▶ Arrays can be stored in shared memory

# Program Flow

# CUDA Program Flow

- ▶ CUDA programs require declaration of memory
- ▶ Similar to C rather than C++ in this regard
- ▶ Rather than newing a variable, we declare an allocation of memory and assign it a name (and a size)
- ▶ `cudaMalloc`
- ▶ Generally done before CUDA function call (allocate at startup)



# CUDA Program Flow

- ▶ Call the externalised CUDA function, passing in reference to data stored on Host memory
- ▶ Copy that data to Device memory (`cudaMemcpyHostToDevice`)
- ▶ Execute kernel on data
- ▶ On completion of kernel execution, copy results back to Host memory (`cudaMemcpyDeviceToHost`)

# CUDA Program Flow

- ▶ The process reflects the nature of the GPU as a batch-based number cruncher
- ▶ Having performed an embarrassingly parallel operation on a load of data, we give that data back to the CPU so it can do something useful with it

OR

- ▶ We keep it on the GPU to do something useful with it (such as directly rendering the results)
- ▶ GPU computation in real-time systems is, therefore, akin to heterogeneous computing - we want something done fast, and often, with batches being passed regularly to the GPU

# CUDA Program Flow

- ▶ Reason we expect batches to be passed regularly is simple: scheduling
- ▶ If we're randomly occupying the GPU with some non-rendering task, that's a fraction of the frame where the GPU cannot be rendering
- ▶ We cannot 'render more' on other frames (take up the slack when not using GPU compute) because that means on the frames in which we do compute we'll get a performance hit
- ▶ So if we don't do a GPU compute update every frame, we're wasting GPU cycles every frame where we're not GPU-computing

# Paradigms



# CUDA Paradigms

- ▶ Memory footprint
- ▶ Parallelisation
- ▶ Host-Device Communication
- ▶ Overhead

# Memory Footprint

- ▶ GPU has limited cache resources shared between a large number of cores
- ▶ More vulnerable to cache misses than CPU architecture
- ▶ Follows that we need a low memory footprint per kernel-instance
- ▶ Algorithms which have a large per-instance memory footprint may need restructuring
- ▶ Algorithms which have a lazy per-instance memory footprint may need unlazifying

# Parallelisation

- ▶ GPU excels at solving embarrassingly parallel problems
- ▶ Embarrassingly parallel problems require little work to separate into parallel, independent tasks (no dependency/communication between threads)
- ▶ Adding communication between threads is possible (shared memory) but can really slow down our execution
- ▶ Something to bear in mind when selecting algorithms to deploy to the GPU

# Host-Device Communication

- ▶ GPU can only act on variables it's been told about
- ▶ Declaring a constant on the Host does NOT mean the Device can see it
- ▶ Need to declare such variables twice, once for Host, once for Device
- ▶ Non-constant variables need passing to the GPU explicitly with each CUDA function call - they don't need mallocing every frame, but they do need memcpying



# Overhead

- ▶ On the subject of memcpying...
- ▶ Every CUDA function will require memcpying (either to the GPU, from the GPU, or both)
- ▶ These instructions are expensive - but become cheaper per element the more elements you send (depending on memory architecture, but usually true)
- ▶ Need to ensure that whatever performance gain we get from the GPU execution outstrips the real-time cost of the overheads, or no point

# Overhead

- ▶ Also, context switching
- ▶ When we change the GPU from compute tasks to rendering tasks, its cache needs flushing
- ▶ This isn't a -very- expensive operation, but it needs to happen every time the context switches
- ▶ Thus, important that we do every frame's GPU computation sequentially, BEFORE we trigger any rendering tasks, or we'll multiply this overhead significantly
- ▶ This has consequences for threading on the Host - separating renderer thread from GPU compute can lead to Bad Things

# Summary

- ▶ Introduced GPU Computation
  - ▶ History
  - ▶ Current API options
  - ▶ GPU Compute vs. Heterogeneous Compute
- ▶ Introduced CUDA
  - ▶ Hardware
  - ▶ Software
- ▶ Described flow of a CUDA function
- ▶ Discussed paradigms of GPU computation (as applicable to CUDA as anything else, except maybe Unified Memory Architecture)

# CUDA Implementation

- ▶ Launch the standard CUDA project
- ▶ Explore the variable types listed in the CUDA API
- ▶ Attempt to expand the test CUDA function to cover arrays of thousands of randomly generated floating point numbers.