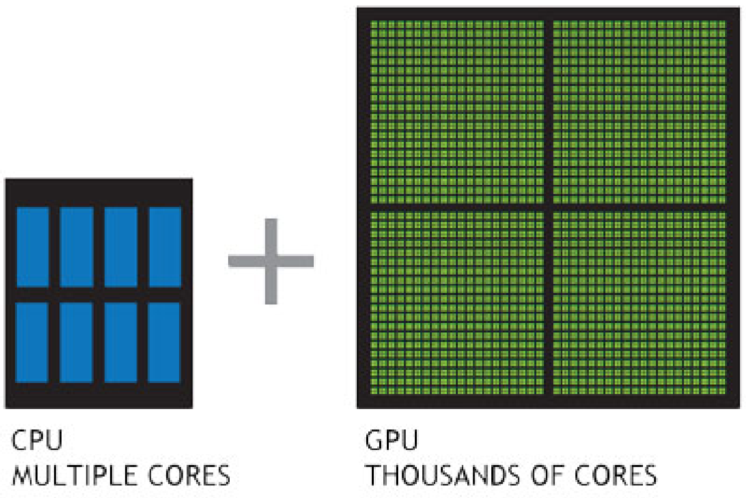
GPU Tutorial 1: Introduction to GPU Computing



Summary

This tutorial introduces the concept of GPU computation. CUDA is employed as a framework for this, but the principles map to any vendor's hardware. We provide an overview of GPU computation, its origins and development, before presenting both the CUDA hardware and software APIs.

New Concepts

GPU Computation, CUDA Hardware, CUDA Software

Introduction

In this pair of tutorials, we shall discuss in some depth the nature of GPU computation. This is not to be confused with rendering, which you have covered in the graphics module, but rather the exploitation of the GPUs vast floating-point throughput as a means of speeding up certain elements of our software.

This is an area of growing interest in video game development. Two of the three current generation consoles selected AMD SoC solutions with unified memory architectures, allowing the GPU and CPU to readily communicate and update data, in order to leverage the computational power of the GPU portion of the chip. Indeed, the fact that both of these hardware solutions featured octa-core set-ups backed up with multi-compute-unit graphical solutions strongly suggests that multi- and many-core computation will be a significant area of games-related research for years to come.

Our first tutorial shall discuss the origins of GPU computation, before introducing GPU architecture, focusing upon the specific hardware you shall be programming. Once we have introduced the nature of the hardware model, we shall discuss its strengths, limitations, and the philosophies which underpin the deployment of a problem to the GPU. Lastly, in this session, we shall discuss the variable types specific to CUDA programming, and how they map to the hardware, before moving on to implement some simple CUDA functions to test our understanding.

GPU Computation Overview

The concept of number-crunching on the GPU is, almost, as old as the GPU itself. Early solutions revolved around the idea of manipulating pixel data through shader language, as a means of performing simple floating-point calculations in a dummy graphics shell. Essentially, where in rendering we perform per-pixel operations in the context of colour space, early GPU computation used those colour components to conceal the numerical data which needed processing. In some cases, just to make this function the researchers then had to force the GPU to render *something* to get the results out the other end (generally two triangles).

Around 2004, researchers began taking this idea very seriously. A lot of problems, particularly simulation problems, have significant amounts of physical data to consider; in computing terms, physical points are often handled as three element vectors. It is not difficult to see how this mapped conveniently to the colour variables in rendering. Similarly, many of the problems research focused upon were relatively straightforward mathematics and, where scale was a problem rather than complexity, it was believed that the GPU offered a cost-effective improvement to performance.

Windows Vista changed the playing field with DirectX 10s unified shader model. Prior to this, shader cores had very specific tasks and were largely incapable of performing any other task (different instruction sets for different shader types). With this move towards unified shaders came an industry sea-change in favour of more generally capable shaders all-round if the instruction sets needed to be generalised in terms of vertex, pixel and geometry shader need, why not generalise them as far as possible beyond that?

With the advent of CUDA, and later FireStream, researchers gained access to easily programmable APIs (relative to performing GPU computation using shader language) and ever-more-capable hardware. The issue then became one of identifying problems that the GPU could solve well, and deploying those solutions; similarly, avoiding deploying problems to the GPU which did not lend themselves to its strengths.

Now, there are several well-established APIs for GPU computation. We list a sample of these below, and categorise their more important features:

Table 1: GPU Computation 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

* C++ AMP has received ongoing investigation from Intel (See: Shevlin Park) which suggested it could be made far quicker than current benchmarks suggest (and OS-agnostic) with compiler optimisations that redirect to OpenCL/GLSL from DirectCompute. If that work is ever made public, C++ AMP might have claim to be both the most accessible cross-platform API available, but three years on it seems unlikely.

In these tutorials we focus on CUDA as it is the most straightforward API through which to implement GPU computation without completely abstracting the GPU hardware (C++ AMP is easier to write in, but does not require us to think about the machine were deploying our code on, OpenCL is less accessible to the novice GPU programmer, though you're invited to explore the API on your own time). The principles discussed in this lecture series, however, map to all contemporary GPU computation APIs, as the issues faced in deploying code to the GPU do not change with vendor.

CUDA

Hardware

In this tutorial we outline the Kepler CUDA hardware architecture, which maps to the GTX 780Ti graphics processors present in most of the MSc machines. Some of you are using GTX 970 cards, which have a Maxwell-architecture chip in them - the principles do not change in the context of the tutorial, only cache ratios, and so on. Figure 1 illustrates an abstract overview of the Kepler architecture.

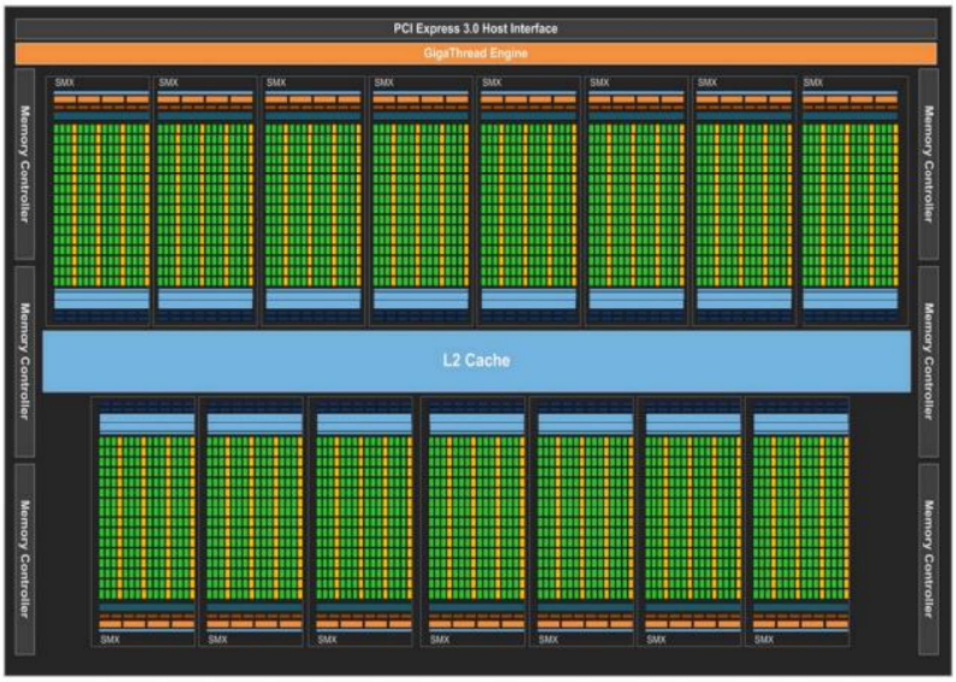


Figure 1: The Kepler Architecture

You can see from Figure 1 (credit: NVIDIA, Kepler Whitepaper), that the GPU is subdivided into several units (referred to in NVIDIA literature as streaming multiprocessors, or SMX). These units share the L2 Cache and, through that, access to the VRAM (analogous to system memory when programming for the GPU). Figure 2 (credit: NVIDIA, Kepler Whitepaper), illustrates the layout of the SMX itself.

An SMX features 192 single-precision cores and 64 double-precision, along with 32 special function units (SFUs units optimised for common mathematical functions). You will note also the memory architecture. 48KB of Read-Only Data Cache, and 64KB of memory labelled "Shared Memory/L1 Cache".

This 64KB is a pool of memory that you can, through the CUDA API, control to favour one or the other (L1 Cache, or Shared Memory) 16KB L1 and 48KB Shared; 16KB Shared and 48KB L1; or, 32KB of each. Shared Memory is a store for variables that can be accessed and updated by any core in the SMX, at any time. The L1 Cache pool is a shared cache pool which is used by every core in an SMX.

The instruction cache for a single SMX is used by all cores in that SMX (meaning that all cores will execute the same set of instructions). The Warp Scheduler handles the initiation of cores to execute their 'instance' of the instruction (the kernel instance, discussed later). If instruction sets branch significantly (if-then conditions which make their completion time varied), the warp scheduler will not be able to leverage maximum efficiency from the cores in the SMX.

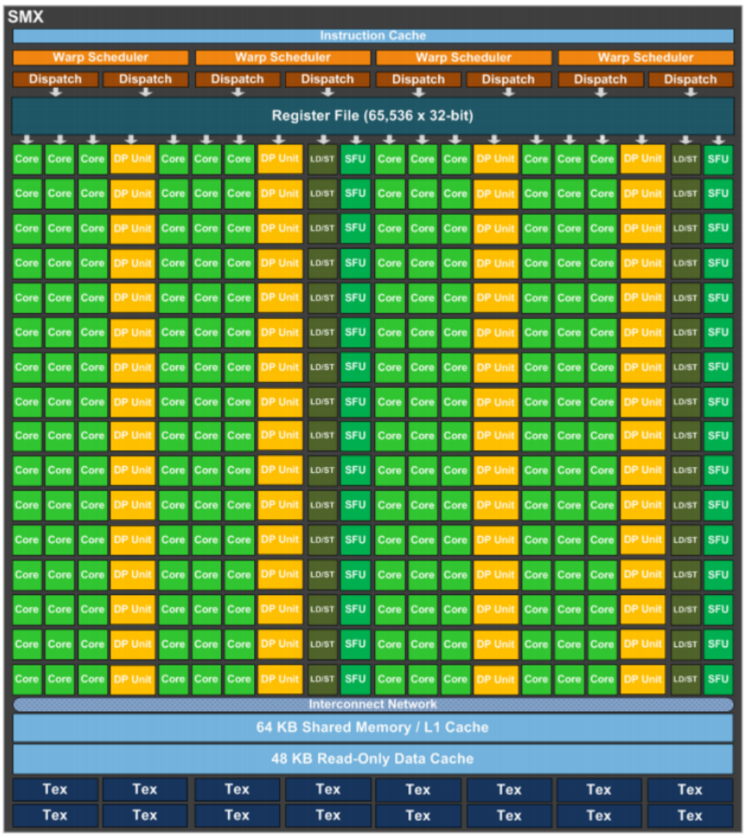


Figure 2: The Kepler Architecture

It should be obvious at this point that the architecture of the GPU is a very different beast to that of the CPU. The CPUs in your desktop have as much L1 cache per core as is allocated by default to all 192 single precision cores in the SMX combined. They also enjoy more versatile instruction cache, optimised for resolving cache misses more rapidly (not something the GPU can claim, regrettably).

This makes sense, however, when we consider exactly what the GPU is intended to do: it executes shaders, which are themselves very simple functions (in terms of instructions if not theory), across all cores simultaneously. Its memory architecture is optimised towards that purpose. And if we are going to leverage this hardware to perform computationally intensive tasks for us, we need to keep that firmly in mind.

Software

CUDA is NVIDIAs hardware and software architecture; when we refer to CUDA in these tutorials, we are normally referring to the software API. In that context, CUDA is a C-styled language that permits the deployment of programs on the GPU. CUDAs syntax is relatively straightforward (and documented in the CUDA API).

You can integrate your CUDA functions with your existing C++ projects through the use of external functions (the extern compiler instruction). This enables you to add CUDA functionality to your codebase, rather than rewriting your codebase into a VS2012 CUDA project.

The CUDA programming model is built on the idea of a grid execution; within the grid are a number of blocks; within a block, are a number of threads. A thread is a single instance of a kernel. It accepts a set of variables, and performs a set of instructions using those variables. A thread has a block ID within its thread block and grid; this is used to determine the threads unique ID, which normally maps to the data element it is accessing. IE, threadID 103 accesses the 103rd element of the arrays that have been sent to the GPU.

A block is a set of concurrently executing threads. These threads cooperate with each other through barrier synchronisation and shared memory. A block as a block ID within its grid. A grid is an array of thread blocks that execute the same kernel. The grid reads in inputs from global memory, writes results back out to global memory, and synchronises between multiple, dependent kernel calls.

You can consider initiating a kernel function as generating a grid, whose size is determined by the number of elements you have instructed the GPU to process. A constant in CUDA is stored in constant memory accessible by all threads. Arrays cannot be stored in constant memory. Shared memory is accessible to all threads in a block; arrays can be stored there. Similarly, read-only memory is accessible to all threads in a block.

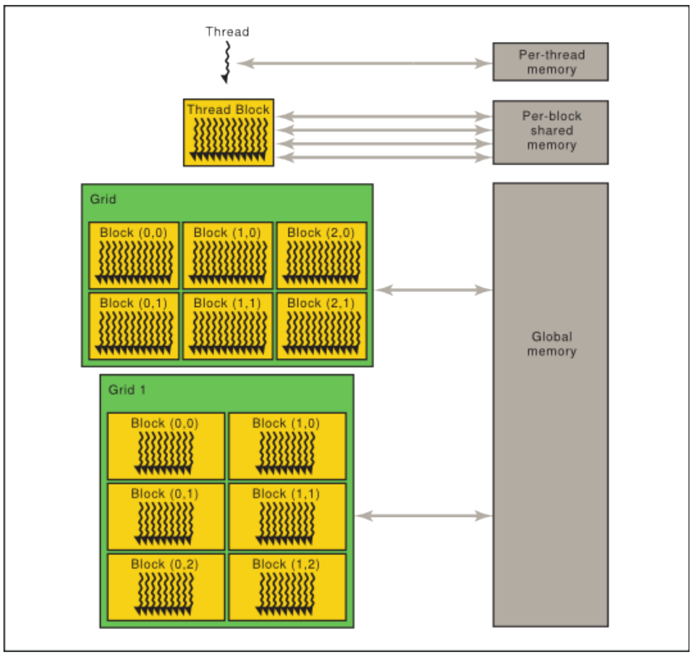


Figure 3: Memory Hierarchy - Grids, Blocks, Threads

Figure 3 (credit: Nvidia) summarises this graphically. It also helps illustrate the hierarchy of threads, blocks and grids. In this figure, multiple grids will be executed; communication between them, as indicated, can only occur via global memory.

Program Flow

A CUDA program requires the declaration of memory on the GPU (Video Memory). The size of this memory chunk is determined by the function you intend to execute, and is declared through cudaMalloc at the beginning of your program loop. As in C programming, that memory must be freed (using cudaFree) when your program loop ends.

When you call an externalised CUDA function, you will pass in array references to the variables you wish to be processed by the kernel. This data will be copied to the GPU memory using cudaMemcpy (of `kind` cudaMemcpyHostToDevice), before the kernel is executed. The kernel will execute on this data. On completion of the kernels execution, you will copy (cudaMemcpyDeviceToHost) the results back to system memory.

This emphasises the role of the GPU as a batch-based number-cruncher. You send it a chunk of data from system memory, perform a parallelisable operation on that data, and it kicks updated information back to system memory (or feeds it forward into some other, GPU-related process, such as rendering).

Paradigms

When we consider a problem for deployment to the GPU, there are four factors we need to keep in mind:

- Memory footprint per instruction set execution. Our GPU has limited cache resources shared between a large number of cores. It is far more vulnerable to cache misses than any CPU architecture. If our problem has a large memory footprint per execution (such as heuristic path planning), we might need to restructure it to best fit the GPU programming model, or not deploy it to the GPU. Of course, a large memory footprint overall poses no issue - so long as the memory footprint per execution is small.
- Parallelisation. The GPU excels at solving embarrassingly parallel problems (problems with no communication between threads, and no perfect execution order). If we add communication between threads, our program will slow down. We should be mindful of this, when selecting algorithms for deployment on the GPU.
- Host-Device Communication. The GPU can only act on variables we pass to the GPU. If variables are stored in system memory, they cannot form part of the kernels instructions. Instead, they must be duplicated to the GPU.
- Overhead. Every CUDA call requires memcopy operations these are costly, and can slow down our program significantly. Also, if we use our GPU for rendering, as well as computation, we should try to avoid deploying draw instructions at the same time as we execute CUDA kernels. This triggers context switching, which can be a costly process in terms of frame-rate as each context switch can cost around 10 microseconds.

In the context of game engineering, this fourth issue is of key importance - because, in a game, our GPU is meant to be rendering an attractive scene. If we're shunting work to it that distracts from that task, it must be for some meaningful reason - not simply because we want to use GPU computation. Normally the sorts of problems you would outsource to the GPU are those where a quality improvement overall makes the loss of GPU cycles acceptable - or a situation where a CPU solution creates such a bottleneck that outsourcing the task to the GPU actually increases frame-rate.

Implementation

Explore the sample software in the CUDA SDK, to understand the demarcation between tasks performed by the Host, tasks instigated by the Host but performed on the Device, and tasks instigated by the Device and performed by the Device.