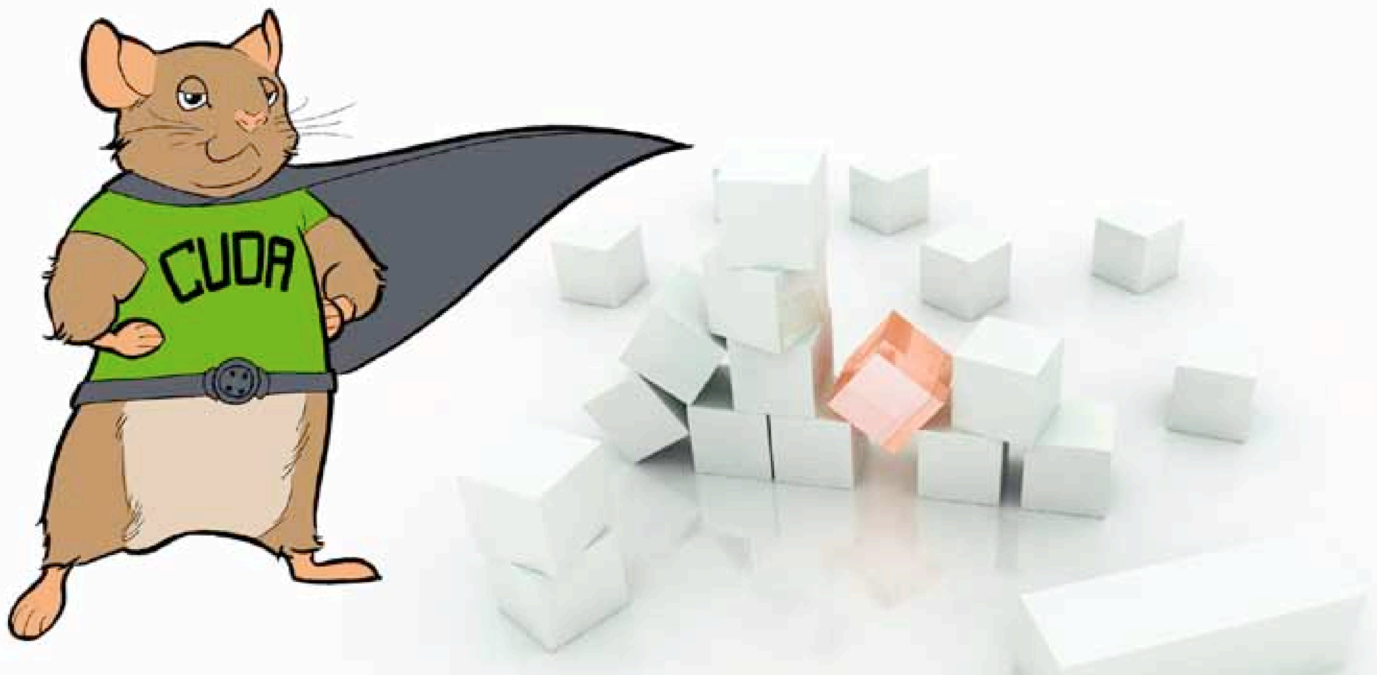
Introducing NVIDIA's Compute Unified Device Architecture (CUDA)

This article, the first in a series, introduces readers to the NVIDIA CUDA architecture, as good programming requires a decent amount of knowledge about the architecture.



Jack Dongarra, professor at the University of Tennessee and author of Linpack has said, "Graphics Processing Units have evolved to the point where many real-world applications are easily implemented on them, and run significantly faster than on multi-core systems. Future computing architectures will be hybrid systems with parallel-core GPUs working in tandem with multi-core CPUs."

Project managers often instruct developers to improve their algorithm so that their compute efficiency of their application increases. We all know parallel processing is faster, but there was always a doubt whether it would be worth the effort and time—but not any more! Graphics Processing Units (GPUs) have evolved to flexible and powerful processors, which are now programmable using high-level languages supporting 32-bit and 64-bit floating-point precision and do not require programming in assembly. They offer a lot of computational power, and this is the primary reason that developers today are focussing on getting the maximum benefit of this extreme scalability.

In the last few years, mass marketing of multi-core GPUs has brought terascale computing power to laptops and petascale computing power to clusters. A CPU + GPU is a powerful combination, because CPUs consist of a few cores optimised for serial processing, while GPUs consist of thousands of smaller, more efficient cores designed for parallel performance. Serial portions of the code run on the CPU, while parallel portions run on the GPU.

The Compute Unified Device Architecture (CUDA) is a parallel programming architecture developed by NVIDIA. CUDA is the computing engine in NVIDIA GPUs that gives developers access to the virtual instruction set and memory of the parallel computational elements in the CUDA GPUs, through variants of industry-standard programming languages. Exploiting data parallelism on the GPU has become significantly easier with newer programming models like OpenACC, which provides developers with simple compiler directives to run their applications in parallel on the GPU.

Recently, at the 19th IEEE HiPC conference held in Pune, I met several delegates from academia and industry who wanted to make use of this extreme computing power, to run their programs in parallel and get faster results than they would normally get using multi-core CPUs. Graphics rendering is all about compute-intensive, highly parallel computation, such that more transistors can be devoted to processing of data rather than data caching and flow control. You can simply take traditional C code that runs on a CPU and offload the data parallel sections of the code to the GPU. Functions executed on the GPU are referred to as computer kernels.

Each NVIDIA GPU has hundreds of cores, where each core has a floating point unit, logic unit, move, compare unit and a branch unit. Cores are managed by the thread manager, which can manage and spawn thousands of threads per core. There is no overhead in thread switching.

CUDA is C for parallel processors. You can write a program for one thread, and then instantiate it on many parallel threads, exploiting the inherent data parallelism of your algorithm. CUDA C code can run on any number of processors without the need for recompilation, and you can map CUDA threads to GPU threads or to CPU vectors. CUDA threads express fine-grained data parallelism and virtualise the processors. On the other hand, CUDA thread blocks express coarse-grained parallelism, as blocks hold arrays of GPU threads.

Kernels

CUDA C extends C by allowing the programmer to define C functions called kernels, which, when called, are executed N times in parallel by N different CUDA threads, as opposed to only once like regular C functions. A kernel is executed by a grid, which contains blocks.

The CUDA logical hierarchy (Figure 2) explains the points discussed above with respect to grids, blocks and threads.

A block contains a number of threads. A thread block or 'warp' is a collection of threads that can use shared data through shared memory and synchronise their execution. Threads from different blocks operate independently, and can be used to perform different functions in parallel. Each block and each thread is identified by a 'build-in' block index and thread index accessible within the kernel. The configuration placement is determined by the programmer when launching the kernel on the device, specifying blocks per grid and threads per block. Probably, this would be a lot of data to take in for someone who has just been introduced to the world of CUDA, but trust me, this is much more interesting once you sit down and start programming with CUDA.

Well, I believe that by now, you have a basic understanding of CUDA thread hierarchy and the memory hierarchy. One important point to consider here is that all applications won't scale well on the CUDA device. It is well suited for problems that can be broken down into thousands of smaller chunks, to make use of the intensive threads in the architecture. CUDA can take the best advantage of C, one of the most widely used

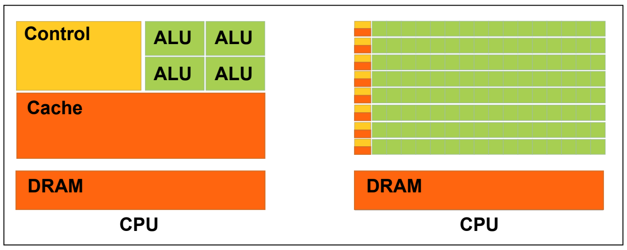


Figure 1: Basic CUDA Architecture

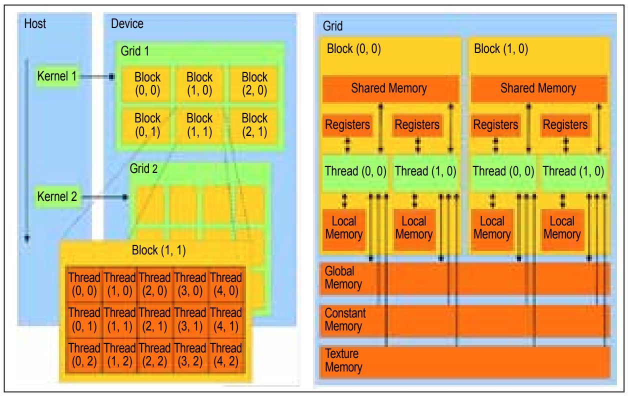


Figure 2: CUDA logical hierarchy

programming languages. You do not need to write the entire code in CUDA. Only when performing something computationally expensive, you could write a CUDA snippet and integrate it with your existing code, thus providing the required speedup.

NVIDIA has sold more than 100 million CUDA devices since 2006. With massive parallel programming reaching the end users and becoming a commodity technology, it is essential for a developer to understand the architecture and programming.

I will cover the basics of CUDA programming in an upcoming article. Till then, it would be worthwhile to put on your thinking caps and start thinking about algorithms in parallel. With massive parallel programming reaching the end users and becoming a commodity technology, it is essential for a developer to understand the architecture and programming of these devices which has redefined the world of parallel computing.

Resources

[1] *http://www.nvidia.com/cuda*
[2] *CUDA Wikipedia: http://en.wikipedia.org/wiki/CUDA*
[3] *Antonino Tumeo (Politecnico di Milano), "Massively Parallel Computing with CUDA"*

By: Tejaswi Agarwal

The author is passionate about computing power utilisation and often spends his time computing the run-time and memory utilisation of various algorithms. Computer architecture, parallel programming and performance engineering are some of his areas of research. He can be reached at *tejaswi.agarwal2010@vit.ac.in.*