GPU Programming with CUDA

A brief overview

Filipo Novo Mór
Postgraduate Program in Computer Science
Pontifical Catholic University of Rio Grande do Sul
Porto Alegre, RS, Brazil
filipo.mor@gmail.com

Plauto de Abreu Neto
Postgraduate Program in Computer Science
Pontifical Catholic University of Rio Grande do Sul
Porto Alegre, RS, Brazil
plauto.a@gmail.com

Abstract— In this paper we describe the architecture of a NVIDIA GPU, as well as the CUDA programming model. The basic statements are explained. We also provide an example of CUDA code, explaining its execution workflow in a GPU device.

Keywords—CUDA; GPU; NVIDIA; programming;

I. INTRODUCTION

The first graphic accelerator cards were developed during the 90’s by a group of companies, each one of them seeking to conquer its own piece of the growing game market. NVIDIA Corporation was among those companies and launched its first graphic card in 1995, named NV1. In 1996, Microsoft Corporation unveiled the first DirectX drivers that were designed to take advantage of the graphic accelerators of the time, what helped to spread out the use of dedicated hardware for graphic acceleration in gaming software.

The basic idea of a graphic accelerator card is to free the CPU from massive processing tasks directly related to the image texturing and video rendering, as showed on Fig. 1.

![Image](image-url)

*Fig. 1.:* image texture processing. [1]

Due to that, normally this hardware keeps its own memory banks, apart of the main CPU memory space addressing. This way, these devices actually implement independent processing units, being able to deal with a huge amount of numeric data [1]. And by releasing the CPU from the charge of massive video processing, the use of a graphic accelerator may improve the overall performance of a game. At the same time, by having a specialized and dedicated hardware, its use will result in a better image quality [2].

After a while, some developers realized that the processing power available on those accelerators could be used to improve the performance of other types of applications, and not only for gaming. However, to access those processing resources, it was mandatory to code using very complex graphic statements and data structures, since the hardware had been designed to adhere to game programming constraints.

But the situation suddenly changed in August of 1999 when the NVIDIA Corporation launched its first GPU (Graphic Processing Unit). The NVIDIA GeForce 256 had around 22 million transistors and was able to process 10 million polygons per second [3]. It had versions with 32MB and 64MB of memory, and was the first card to implement on hardware the transform and light engine. The T&L [4] engine (vertex shader) is now a pattern on the industry. For the first time, a graphic accelerator card had reached that processing power level. And thanks to its architecture, the GeForce GPU family quickly became the most accepted and wide spread pattern for graphic cards.

But at that time, GPU coding was still a challenge, because it was compulsory to keep coding by using complex graphic primitives as statements [1]. It was only in 2006, after the developing of many GPU generations, that NVIDIA once again gave a big step forward by launching the first CUDA (Compute Unified Device Architecture) version. For the first time it was possible to access the GPU resources by using simple statements, pretty similar to those already used in third generation programming languages. Actually, the first CUDA version was a simply extension of the C language. But its advent brought the concept of GPGPU (general-purpose computing on graphics processing units)

II. THE NVIDIA GPU ARCHITECTURE

A. Multicore CPU vs many-core GPU

Specifically designed for massive parallel data processing, the GPU architecture allows a performance overshoot over the multicore CPUs performance when the problem to be solved relies on strong data parallelism. In fact, the original problem for which the graphic accelerators were designed (the image processing and video rendering) is based on transformation of a
huge amount of independent numeric data, which in turns make problems based on this kind of data organization ideal candidates for GPU processing [1]. The Fig. 2 shows a comparison of the recent evolution of computational power of commercial CPUs and GPUs.

![Comparison of CPU and GPU performance](image)

**Fig. 2:** comparative performance between commercial GPU and CPU [1].

But it's important to understand the reasons that lead the GPU to this outstanding performance when compared to a conventional CPU. First, the GPU design focus the processing of big bunches of massive parallel data also named as stream processing [5]. This is possible because a wider area on the silicon chip is reserved for actual processing units with allocation of a bigger amount of transistors on those areas, while cache and control structures are restricted to minimum size possible, as shown by the Fig. 3.

![CPU and GPU architectures](image)

**Fig. 3:** transistor allocation on CPU and GPU architectures [1].

This feature directly reflects on the ability to deal with a big amount of parallel data, but, by the other hand, it makes harder to the GPU to manage non-linear workloads, as, for example, to manage recursion, queuing or context changing.

The second reason relies on the problem to be solved. If the problem is easily parallelizable, then it may be a good candidate for processing on GPU. By the other hand, if the problem is hard to parallelize, a hybrid implementation (CPU and GPU) may be a good option. In fact, conventional CPUs use task parallelism, and not data parallelism. Multiple tasks are mapped into multiple threads. Each thread runs different instructions, whilst each of them requires to be explicitly managed and scheduled [6]. It means that each thread demands to be controlled by software and have to be individually programmed. By the other hand, a GPU performs data parallelism by having a SIMD (Single Instruction Multiple Data) structure. It means that a GPU is able to run the same instruction over different data. Also, all the threads are managed and scheduled directly by the hardware. The GPU programming model relies on coding batches of independent threads.

Finally, GPUs are “latency tolerant” [6], while on conventional CPUs a huge effort is made in order to hide or minimize latency. In fact, caches are needed to minimize latency, and also, a CPU normally performs instruction pre-fetch, flow control and out-of-order execution. On the other hand, GPUs abandoned the pipeline paradigm, avoiding thread synchronization issues by making them totally independent from each other.

### B. NVIDIA GPU Roadmap

Since the launching of its first GeForce GPU, NVIDIA Company has established a roadmap covering the evolution of its GPU products [7]. Each step on this roadmap covers (or will cover) one specific set of capabilities, as shown by Fig 4.

![NVIDIA GPU roadmap](image)

**Fig. 4:** NVIDIA GPU roadmap [7].

The GeForce and Tesla families made part of the first GPU generation that was covered by the CUDA programming framework. But although both product families belong to the same GPU generation, GeForce was designed for gaming and home markets, while the Tesla family was designed for HPC market.

The next generation, Fermi architecture, has provided a series of features, mainly its support to floating point with double precision and also the increase of the amount of shared memory available per each block thread.

The Kepler architecture arrived supporting the Dynamic Parallelism, which allowed a considerable increase in the autonomy of the CUDA kernel codes (performed on devices), in relation to its past dependency of the CPU task control.

The future Maxwell architecture (yet to be launched by NVIDIA) promises a unified memory space between GPU and CPU, which should improve the performance of applications that need to communicate with processes running on the CPU. This feature also should make easy to implement hybrid codes (that uses CPU and GPU collaboration) that are common nowadays on cluster applications.
Finally, the future Volta architecture should come with improvements on the hardware design, mainly making use of stacked DRAM. This probably will represent improvements on efficiency of energy consumption, but not necessarily on execution time reduction.

C. Compute Capability

The Compute Capability describes the technical features supported by a CUDA compliant GPU. Devices from the same GPU family may have different compute capabilities. That occurs because different sets of features were implemented throughout the NVIDIA roadmap. Each compute capability differentiates from others by implementing a different group of technical capabilities, as shown by Fig 5.

![Compute Capability List](image)

**Fig. 5.:** compute capability list [8].

It is possible to query the device by its compute capability number by running a specific CUDA statement, which allows the implementation of a code that will dynamically adapt during run time to the available GPU board.

III. CUDA PROGRAMMING MODEL

The CUDA programming model treats the GPU as a compute device that serves as a coprocessor for the host CPU. The GPU has its own device memory on the card and it is able to execute many threads in parallel.

The basic execution unit of a CUDA program is named **kernel**. A kernel runs a single program by starting many different concurrent threads. Accordingly to the CUDA programming model, expects thousands of concurrent threads to be fully used.

A kernel can be started in synchronous or asynchronous mode by its host process on the CPU. But depending on the GPU architecture, a kernel may be launched only by a CPU process (Tesla, Fermi and Kepler with compute capability up to 3.0) or be launched by a previous kernel instance (it is possible from the Kepler devices with compute capability from 3.5 or more recent). The fig 6 illustrates this behavior.

![Kernel Launching](image)

**Fig. 6.:** kernel launching from Fermi and Kepler architectures [7].

In practice, the ability to launch a second kernel from a first kernel independently from the host process on the CPU (named as Dynamic Parallelism by NVIDIA Corporation) makes possible to run different code versions directly into the device. This feature, in fact, turns the GPU into a MIMD (Multiple Instruction, Multiple Data) device, since now it is possible to run different instructions over different data sets. At the same time, Dynamic Parallelism allows the use the GPU resources in a much more effective way, since it makes possible to focus the processing effort on areas where it is more needed, as shown by Fig. 7.

![Dynamic Parallelism](image)

**Fig. 7.:** Dynamic Parallelism as explained by NVIDIA [7].

A CUDA kernel is executed by an array of threads. Each thread has its own numeric ID, which may be read by a CUDA statement. But there are two extra organization types available for the thread group. Beyond the default array mode, the thread block may be set as matrix (a two dimensions thread array) or even as a cube (a three dimension array). Choosing one of other will not affect execution time of the application, and this choice depends only on developer preferences. On the other hand, there are certain limits on matrix or cube dimensions. These limitations are specific for each board and may be directly queried to the device on runtime.

Due to its hardware design, CUDA threads are extremely lightweight, with a very little creation overhead and allowing
instant switching. Fig. 8 shows a representation of CUDA thread block.

![Cuda Block](image)

Fig. 8.: representation of a CUDA thread block.

The kernel launches a grid of thread blocks, where threads within a block can share data through shared memory and synchronize their execution. But threads in different blocks cannot cooperate directly. In order to cooperate, these threads may access the global memory. However, this access is usually more costly than a direct access to shared memory. Accordingly to NVIDIA documentation, an IO operation on shared memory may cost 4 clock cycles, while the same operation on the global memory may cost between 300 and 400 clock cycles. Thus, the lack of synchronization between threads from different thread blocks may easily rise into racing conditions.

The Fig. 9 shows the basic representation of a NVIDIA GPU and its memory hierarchy.

![Cuda Memory Hierarchy](image)

Fig. 9.: CUDA memory hierarchy.

Due to is memory hierarchy, the amount of shared memory available for each thread block, and the quantity of free registers for each thread should be take into consideration by the developer. For example, if he declares more local variables than the free registers to be used for each thread may contain, the CUDA compiler will allocate extra space on the global memory to place this variables (NVIDIA call this special allocation as Local Memory usage). This operation will only generate a warning message during the building process, but will decisively affect the overall application performance.

Each thread has access to some CUDA language identifiers, which allow the developer to identify specifically the thread location within a thread block or block grid:

- __blockIdx.x__ – block ID within grid.
- __clockDim.x__ – number of threads per block.

For example, to map a data array placed in the global memory into a specific thread within a thread block, both identifiers may be used, as shown by Fig 10.

![Cuda Block Coordination](image)

Fig. 10.: mapping global memory addresses into individual threads.

By using threadIdx.x, threadIdx.y and threadIdx.z CUDA statements, it is possible to group the threads in multidimensional structures, as Fig. 11 illustrates.

![Cuda Multidimensional Ids](image)

Fig. 11.: multidimensional thread IDs.

A. CUDA Programming Basis

CUDA C extends the default ANSI C language by appending some new statements and data types. For example, there are new function qualifiers:

- __global__ : called from the host (CPU) code, but runs on GPU;
- __device__ : called from other GPU functions and runs only on GPU;
- __host__ : called from the host and runs only on CPU;

Also, a new set of variable qualifiers were introduced:

- __device__ : stored in global memory (these data will be no cached, which means high latency);
• **__constant__** : stored in global memory, but cached. This memory will be read-only for threads and written by the host process only (on CPU);

• **__shared__** : stored in shared memory. Accessible by all threads in the same block, but invisible for threads from other blocks. Each thread block has its own amount of private shared memory.

Unqualified variables will be stored in local memory, as follows: scalar and built-in vector types will be stored in registers when possible, otherwise, they will be placed by the compiler on the device global memory. Arrays are stored on the global memory by default.

**B. Study Case: dot product**

Dot product is a reduction operation from two vectors to a scalar value and it is defined as shown by the equations 1 and 2.

\[
\vec{A} \cdot \vec{B} = \sum_{i=1}^{n} a_i b_i
\]

Where:

\[
A = (a_1, a_2, ..., a_n)
\]

\[
B = (b_1, b_2, ..., b_n)
\]

In fact, this is a good matching problem for a SIMD machine as a GPU. The Fig 12 shows the workflow for the parallel algorithm that implements the solution for this problem.

![Fig. 12.: dot product parallel workflow.](image)

The pair multiplication step may be completely parallel, and if there are processing units enough to cover all array elements, the speed up for this step may reach the \( n \) over \( p \) (\( n \) for array elements and \( p \) for processing units). However, the final sum step has to be atomized in order to avoid racing conditions. This constraint can be implemented in CUDA by using two different ruses at once: first, all the threads may be synchronized by forcing them to stop by a common barrier; second, once all threads had proceeded the summation of its own pair of array elements and all intermediate values are stored on the memory, the final summation can be performed in a sequential way by one of the threads. This thread will be the “master thread”. The Fig 13 shows the CUDA kernel code that implements this reduction.

![Fig. 13.: kernel code for dot product.](image)

All the threads will perform its private pair summation and then will stop (synchronize) by the line 7, forced by the statement **_syncthreads()**. Once all threads have reached that point, the thread “zero” (identified by threadIdx.x equals to zero) will still be running, while the remaining threads on the same block will simply die. The command loop performed between the code lines 12 and 15 will accumulate the value of the array pairs summed by all the threads of the current thread block. At the end of this phase, an atomic operation at line 16 will perform the final summation step, guaranteeing the correctness of the final summation value by preventing racing conditions between all threads “zero” from all thread blocks. It means that in the final phase, the CUDA runtime will sequence the summation threads for those temporary values in order to reach the correct final value.

The host code is shown by the Fig. 14.

![Fig. 14.: host code for dot product.](image)

In the lines between 11 and 13 the memory on the device (in the GPU global memory) will be allocated. The CUDA statement has a quite similar syntax to the original malloc from the C language. It is mandatory to inform the memory pointer and memory length (size) of the desired memory space.

The lines 15, 16 and 17 allocate memory for the local data arrays on the host memory (CPU RAM). The functions on the lines 19 and 20 simply fill those arrays with random integer values. Once these data arrays are fully filled, the
The cudaMemcpy() statement on the lines 23 and 24 will copy the memory space allocated on the local host (now filled with integer random values from the both data arrays) to the corresponding memory space on the device memory. These transfer operations are synchronous, which means that the second transfer only will start after the first operation has been finished.

The kernel code is finally launched by the statement on the line 27. The parameters informed are:
- The amount of threads per block;
- The amount of thread blocks;
- The memory pointers (corresponding for the A and B arrays and the third memory space where will be placed the final result).

After the kernel perform the calculation, the result will be placed at the GPU global memory at the address indicated by the dev_c pointer. So, the statement on the line 30 will copy the content of that memory address from the GPU back to the CPU RAM. The statement cudaMemcpy() is used again, but this time, for the reverse direction, as indicated by the parameter cudaMemcpyDeviceToHost. After perform this operation, the result of the dot product will be contained by the local host variable “c”. The Fig. 15 shows the workflow for this algorithm.

![Fig. 15. Workflow for the CUDA dot product algorithm.](image)

**III. CONCLUSIONS**

Raised from the game industry demands for better graphic quality, Graphic Processing Units proved to offer big computational capacity for lower costs. Because of this, the turn into GPGPU was a natural step for industry. Today, most of top HPC (High Performance Computing) machines are equipped with GPUs. Until now, its architecture remained based on SIMD machines, but the latest NVIDIA generations, as the Kepler family and the future Maxwell family will implement actual MIMD machines. This way, GPU programming tends to increase their level of specificity, demanding more specialization from the developers. As an immediate result of this tendency, the distance between conventional sequential programming will increase even more. And this may open space on the market for other philosophies of lateral accelerators, as for example the Intel Xeon Phi, that is based on the X86 architecture and tends to be more accessible to sequential code developers. On the other hand, maybe these new GPU families will be able to adhere to a wider range of problems, which today demand hybrid implementations with collaborative CPU and GPU approaches.

**REFERENCES**