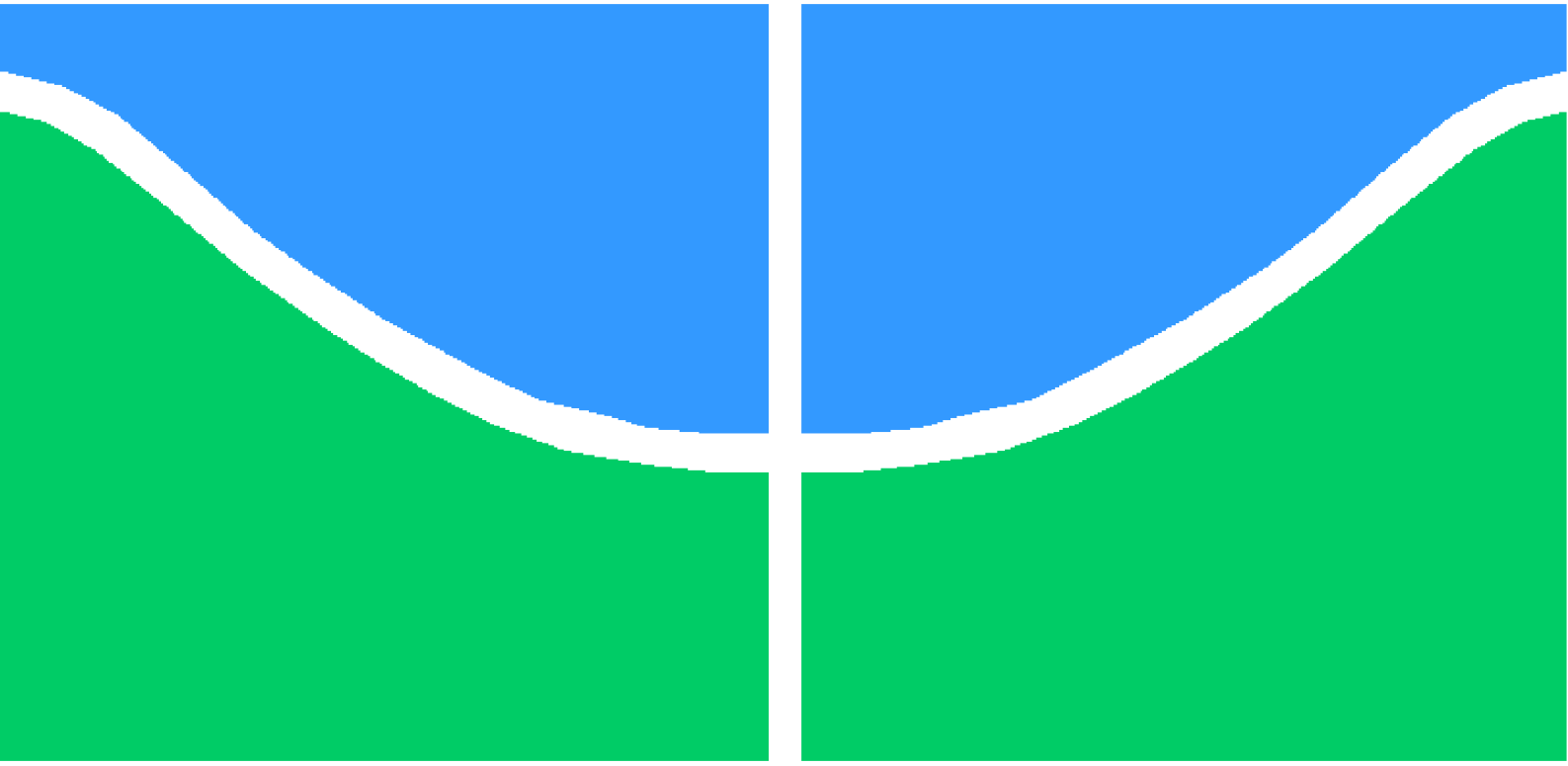

Universidade de Brasília - UnB
Faculdade UnB Gama - FGA
Software Engineering

A study of replacing CUDA by OpenCL in KGPU

Author: Rodrigo Siqueira de Melo
Advisor: Dr. Edson Alves da Costa Júnior

Brasília, DF
2015

Rodrigo Siqueira de Melo

A study of replacing CUDA by OpenCL in KGPU

Monograph submitted to the undergraduate course of Software Engineering in the Universidade de Brasília, as a partial requirement for obtaining the title of Bachelor of Software Engineering.

Universidade de Brasília - UnB

Faculdade UnB Gama - FGA

Supervisor: Dr. Edson Alves da Costa Júnior

Brasília, DF

2015

Rodrigo Siqueira de Melo
A study of replacing CUDA by OpenCL in KGPU/ Rodrigo Siqueira de Melo. – Brasília, DF, 2015-
68 p. : il. (algumas color.) ; 30 cm.

Supervisor: Dr. Edson Alves da Costa Júnior

Completion of Course Work – Universidade de Brasília - UnB
Faculdade UnB Gama - FGA , 2015.

1. GPU. 2. embedded System. I. Dr. Edson Alves da Costa Júnior. II. Universidade de Brasília. III. Faculdade UnB Gama. IV. A study of replacing CUDA by OpenCL in KGPU

CDU 02:141:005.6

Rodrigo Siqueira de Melo

A study of replacing CUDA by OpenCL in KGPU

Monograph submitted to the undergraduate course of Software Engineering in the Universidade de Brasília, as a partial requirement for obtaining the title of Bachelor of Software Engineering.

Approved work. Brasília, DF, July 10, 2015:

Dr. Edson Alves da Costa Júnior
Advisor

Dr. Evandro Leonardo Silva Teixeira
Guest 1

Dr. Renato Coral Sampaio
Guest 2

Brasília, DF
2015

I dedicate this work for all the Brazilians that still believe,
in our country, and still fighting to build a better nation.

Acknowledgements

Prior of anything, I need to acknowledge two special women that have taught me the value of working hard, honesty, study, ethics, and dreaming. They are my lovely mother and grandmother, whom I have given all my respect, admiration, and love. Furthermore, I have to thank to my little adorable sister that renew my energy and makes me remember to keep dreaming.

Moreover, I need to acknowledge three special professors that have crossed with me in my journey and have changed my life. Firstly, I want to say thank to my teacher Fernando, he always trusted on my capacity and gave me many precious advices when I was in the high school. Secondly, I need to say thank for my teacher Rogerio (as known as, Basilio) that had introduced me the beauty of mathematics. Finally, I have to say thanks to Professor Edson. He has influenced me with his knowledge, way of thinking, working, and humbleness. I have to especially thank him, for always believe in me and in my project, even when I started to not believe.

Throughout my journey, I have met great people. I have made awesome friends that always helped and provided me a good time together. I'm talking about: Ronei that studied many hours with me in the library on Sundays, Ramy that always received me really well and have many enjoyable conversations, Kamilla whose many times shared nice ideas, Rafael that always drink coffee with me in the Library and indroduced me many different points of view, Matheus, Rubens that always provided good ideas, Bruno that always discuss politics with me, Aniderson Fraga who I have shared few cups of the most joyful and inspiring words, moments and sake. Additionally I need to say a special thank to my friend Charles, that always gave me great advices and inspired with his knowledge. Finally, I have to say an extra thanks to Charles due to all the valuable corrections provided by him for this work.

Finally, I need to say a huge thank to Sunmin Song. She is a very special woman that always inspired me a lot with her hard work and her way to see the world. Thanks for have been patient with me and take care of me many times in Korea. I will remember you forever.

"What is really good is to fight with determination, embrace life and live it with passion. Lose your battles with class and dare to win because the world belongs to those who dare to live."
(Charlie Chaplin)

Abstract

GPU is a very high parallel device which became popular. Nowadays, many processors already coming with a minimal GPU in the same die, this characteristic creates a new and unexplored application area for this device. CUDA and OpenCL are two non-graphics libraries commonly used for take advantages of GPU. CUDA was created by NVidia, and it was designed to run on NVidia's GPUs. On the other hand, OpenCL was created to run on many different devices. Those libraries, interacts with the operating system by using device drivers, and usually this is the unique connection between them. A group of researchers from Utah proposed the use of GPU as a coprocessor, they developed a device driver based on CUDA for achieving this goal (they called it as KGPU). In this work we improved KGPU's code, added support to OpenCL, and we analyzed the possibility of use this project as a mature solution.

Key-words: GPU. OpenCL. Coprocessor. Embedded System. Device Driver. KGPU.

Lista de ilustrações

Lista de tabelas

List of abbreviations and acronyms

SGI	Silicon Graphic Inc.
OS	Operating System.
SIMD	Single Instruction Multiple Data.
VGA	Video Graphic Array.
PGA	Professional Graphics Controller.
AGP	Accelerated Graphics Port.
GPU	Graphic Process Unit.
DMA	Direct Memory Access.
APU	Accelerated Process Unit.
SM	Streaming Multiprocessor.
ALU	Arithmetic Logic Unit.
SP	Streaming Processors.
RF	Register File.
SFU	Special Function Unit.
DRAM	Dynamic Random Access Memory.
DDR	Double-Data Rate.
MMU	Memory Management Unit.
DRI	Direct Rendering infrastructure.
PCI	Peripheral Component Interconnect.
NSK	Non Stop Kernel.
PRT	Predictable-Response-Time.
HT	High-Throughput.
AE	Apriori Enforcement.

PE	Posterior Enforcement.
DMIPS	Dhrystone MIPS.
IT	Information Technology.
SoC	System-on-a-Chip.

Sumário

Introduction

In 1965 Gordon E. Moore noticed that the number of transistors by chip doubles every two years, all of this growth made the electronics devices evolve very fast and made the users want to get the most out of their devices. This constant greed has inspired engineers to create the Graphics Processor Unit (GPU) to provide graphic powered device for common users. The unit became incredibly fast, nowadays it can exceed 1000 GFLOPS [1] (ADRIAENS et al., 2011). It is equivalent to 10 times the capacity of the traditional microprocessors. All of this power attracts people to use GPU for other purposes, thus, thousands GPU-ready applications have emerged during the last years. As an example, researchers used GPU for network packet processing (HAN; JANG; MOON, 2010), cryptography (HARRISON; WALDRON, 2008), or fundamental algorithm (OWENS; LUEBKE; GOVINDARAJU, 2007), etc.

Green computing is a new and curious area in which the GPU became objects of study. This subject is related to computation concerning the environment. Some researchers estimated that 2% of global carbon dioxide emission can be attributed to IT systems. Based on that, many researches defend the use of GPU for reducing the execution time of some tasks and reduce the energy consumption (SCANNIELLO et al., 2013).

All of GPU application area, raises a reasonable question: how to take some advantage of the GPU? The answer is: using some specialized libraries. When this device was made, OpenGL and DirectX were created just to make use of it. Both of those libraries concern about graphic generation, and they aim to provide an easy way to create it. However, the main use of libraries was not intended to use GPU in a general purpose area. NVidia noticed that limitation and created the Compute Unified Device Architecture (CUDA) [2]. This new library makes GPU usage easier, it has a huge problem though: CUDA is well supported just for NVidia devices. A few years ago this limitation was not being considered, nevertheless today there are many different manufacturers producing different GPU architectures incompatible with CUDA.

Many companies noted GPU potential and created their own chips. Whereas CUDA was well supported for NVidia devices, the other companies did not have the same background. Therefore, many groups have started to support a project called OpenCL, which is a library maintained by Khronos Group. There are big companies associated with the project, such as Samsung, Apple, ARM, Nokia, Sony, NVidia and others (GROUP, 2014b). The main feature of OpenCL is the portability, it means that a code written for GPU created by Samsung can run in a chip made by NVidia.

[1] FLOPS: FLoating-point Operations Per Second
[2] http://www.nvidia.com/object/cuda_home_new.html

Many companies tried to produce better or cheaper GPUs, and suddenly this kind of device began to be put on the same die of the microprocessor. For instance, ARM added the Mali GPU in some of their chips. This kind of GPU focused on energy efficiency and provided a good performance in a smaller silicon area (GROUP, 2014a). This new kind of SoC (System-on-Chip) aligned with OpenCL opened a new opportunity for research in embedded systems.

With all of the ideas stated before, this work will discuss about the integration of GPU into the Operating System with the intention of using OpenCL instead of CUDA. This work brought many questions that we will answer, however, two inquiry represent the main focus of this work:

1. Is it possible to use OpenCL instead of CUDA and get similar results?
2. Is it possible to use GPU more close to kernel context?

To achieve the expected results, will use a device driver to make GPU work as a coprocessor. That driver uses OpenCL 1.23 [3], because this library can be used in an embedded systems.

[3] Until now, this is the more recent version that is supported on embedded systems.

1 Theoretical rationale

1.1 A brief history of GPU

In the beginning of 1980's the computer market experienced an enormous technical evolution and sales growths. In this period IBM released the Professional Graphics Controller (PGA). This device made use of Intel 8088 to handle some video tasks, and reduce the overhead in the CPU. PGA had two problems: high cost (around $5500) and the lack of compatibility with many programs (MCCLANAHAN, 2014). The hugest contribution from that device was the introduction of the new idea of splitting graphics manipulation from CPU. At this decade IBM coined the term Video Graphic Array (VGA) for the device responsible control and display images (PATTERSON, 2014).

During the years some companies added and improved the idea of VGA. Silicon Graphic Inc. (SGI) improved the 2D/3D manipulation and added the pioneer concept of graphic pipeline. In 1989 some graphics API like OpenGL emerged and contributed for making the graphic devices evaluation faster.

In the 90s, companies as NVidia and ATI began to be recognized by their graphics devices. In this period the VGAs were connected to PCI bus, each one offering millions of transistors, around 4 Mb of 64-bit DRAM, and 50 MHz. Nevertheless the CPU still had to perform vertex transformations and another calculations, limiting the overall performance. Figure 1 presents the referred architecture:

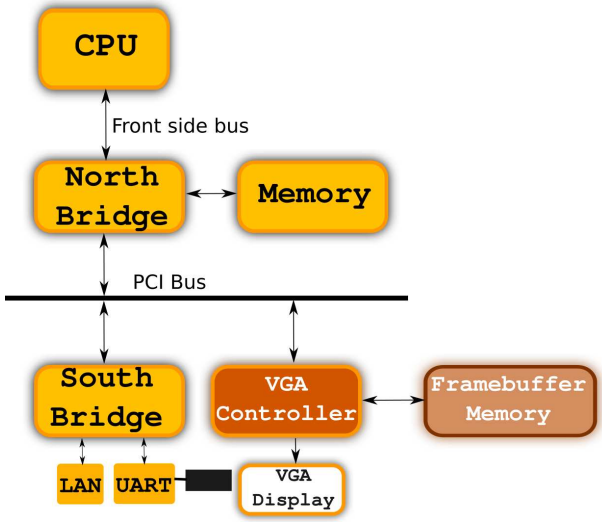

Figure 1 – VGA architecture

The VGA was treated as a normal peripheral, connected in the North Bridge and accessed by PCI (PATTERSON, 2014).

In 1999 NVidia released the GeForce 256 with 23 million transistors, 32 MB with 128-bit DRAM, 120 MHz of clock. It was connected to the Accelerated Graphics Port (AGP) and offered new features (e.g.: multi-texturing, bump maps, light maps, and hardware geometry transform and lighting (MCCLANAHAN, 2014)). All of these characteristics made this devices to receive a different name, and for the first time the term Graphic Processor Unit (GPU) was used by NVidia. The GPU made use of a sophisticated pipeline, but non-programmable, just configurable. See Figure 2:

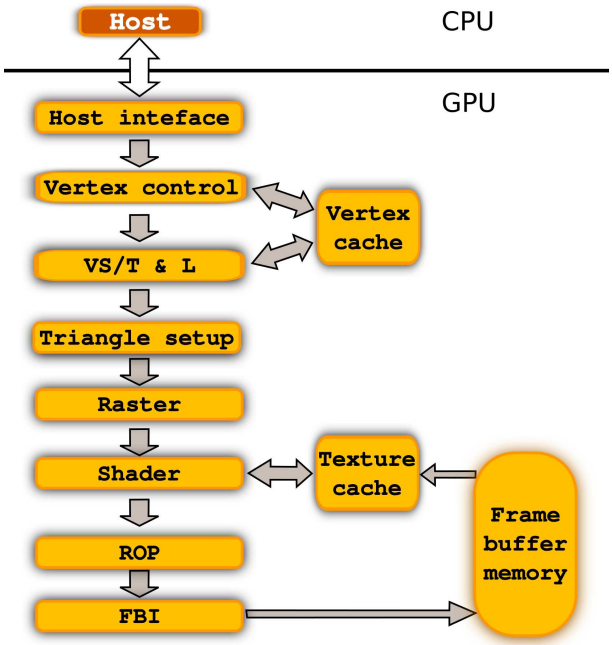

Figure 2 – NVIDIA pipeline

The first step of this pipeline is the *Host Interface* that is responsible for receiving data from the CPU. The common approach for handling this situation is the use of Direct Memory Access (DMA). The second stage is the *vertex control*, translating the triangle data [1] (received from Host Interface) into a form that the hardware understands and placing the prepared data into the vertex cache. The third step is the *vertex shading, transforming, and lighting (summarize as VS/T&L)*. This stage is responsible for transforming vertex and assigning a color for each vertex. The fourth stage, *Triangle Setup*, is responsible for creating edges equations that are used to interpolate color. The fifth stage determines which pixel is contained in each triangle. The sixth stage handling the final color. The seventh stage is the raster operation (ROP), which define the visible object for transparency and anti-aliasing, and a given viewpoint, discards the occluded pixel, and blend the color. The last stage is the *Frame Buffer Interface (FBI)* that manages memory read from and written to the display frame buffer memory (KIRK, 2014).

The model above mentioned worked well for some years, however generation after generation engineers have introduced more and more features. Then, the developers

[1] Usually, the surface of an object can be divided into a collection of triangles.

started to use ostensibly those features and they asked for more functions inside GPU. This situation made the GPU evolve into a new step: it began to offer a programmable pipeline. In 2002 NVIDIA released GeForce FX and ATI the Radeon 9700 the first fully programmable graphics cards (MCCLANAHAN, 2014). This new kind of GPU started a new trend toward to unifying the functionality of different stages as seen by the application programmer.

The GPU technology developed really fast, and in 2004 GeForce 6 and Radeon X800 were the first GPU using PCI-express bus. See Figure 3:

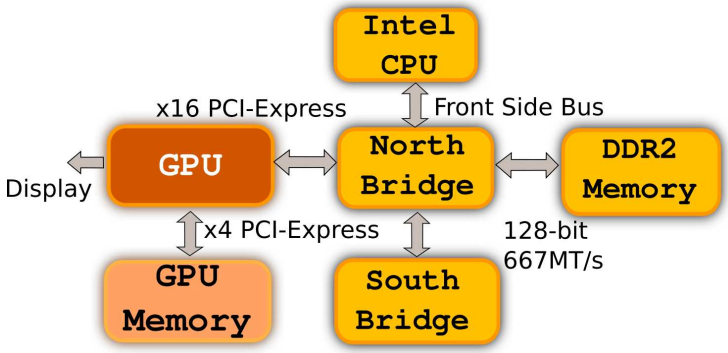

Figure 3 – GPU architecture overview

Compare the GPU architecture with VGA and notice that the graphical devices evolved a lot in the sense of CPU and GPU communication. The fully programmable unified process called a Streaming Multiprocessor (SM) that handle vertex, pixel, and geometry commutation.

Nowadays the new trend is the mix between CPU and GPU on the same chip. This kind of device is known as Accelerated Process Unit (APU).

1.2 Processor Array

At a high abstraction level, the basic CPU is composed by: control unit, ALU, some function units, and memory (Figure 4):

Figure 4 – CPU

Nowadays, engineers want to build a processor massively parallel. In order to achieve this aim they are trying to create many different architectures. One of the ideas is related to using one control unit connected with many ALUs, trying to create a parallelism by data division. See Figure 5.

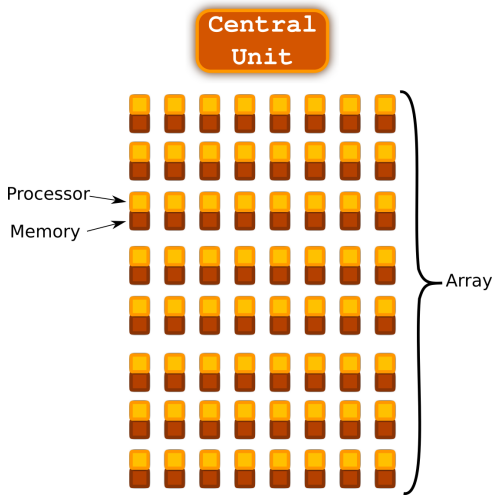

Figure 5 – Processor Array

The processor vector has some unique characteristics that make it special, mainly because it can process many data in parallel. Some of the characteristics are (PACHECO, 2014):

- *Vector register*: Registers are capable of storing vector of operands and operate simultaneously on their contents.
- *Vectorized and pipelined function unit*: The same operation is applied to each element in the vector.
- *Vector instruction*: Instructions that operate on vector rather than scalar.
- *Interleaved memory*: The memory system is composed of multiple "banks" of memory, which can be accessed independently.
- *Strided Memory Access and hardware scatter/gatter*: In strided memory access, the program accesses elements of a vector located at fixed intervals.

The characteristics above mentioned are aligned to the idea of logical graphics pipeline, because physically it is a recirculating path that visits the processor more than once with many of the fixed-function revisited. The characteristics of the vector processor allow the dynamic partitioning of the array to vertex shading, geometry processing, and pixel processing (KIRK, 2014).

In 2006 NVIDIA released the GeForce 8800 that applied the idea of unified processor. After that the engineers accepted the challenge of creating an unified processor. Figure 6 presents 8800 architecture.

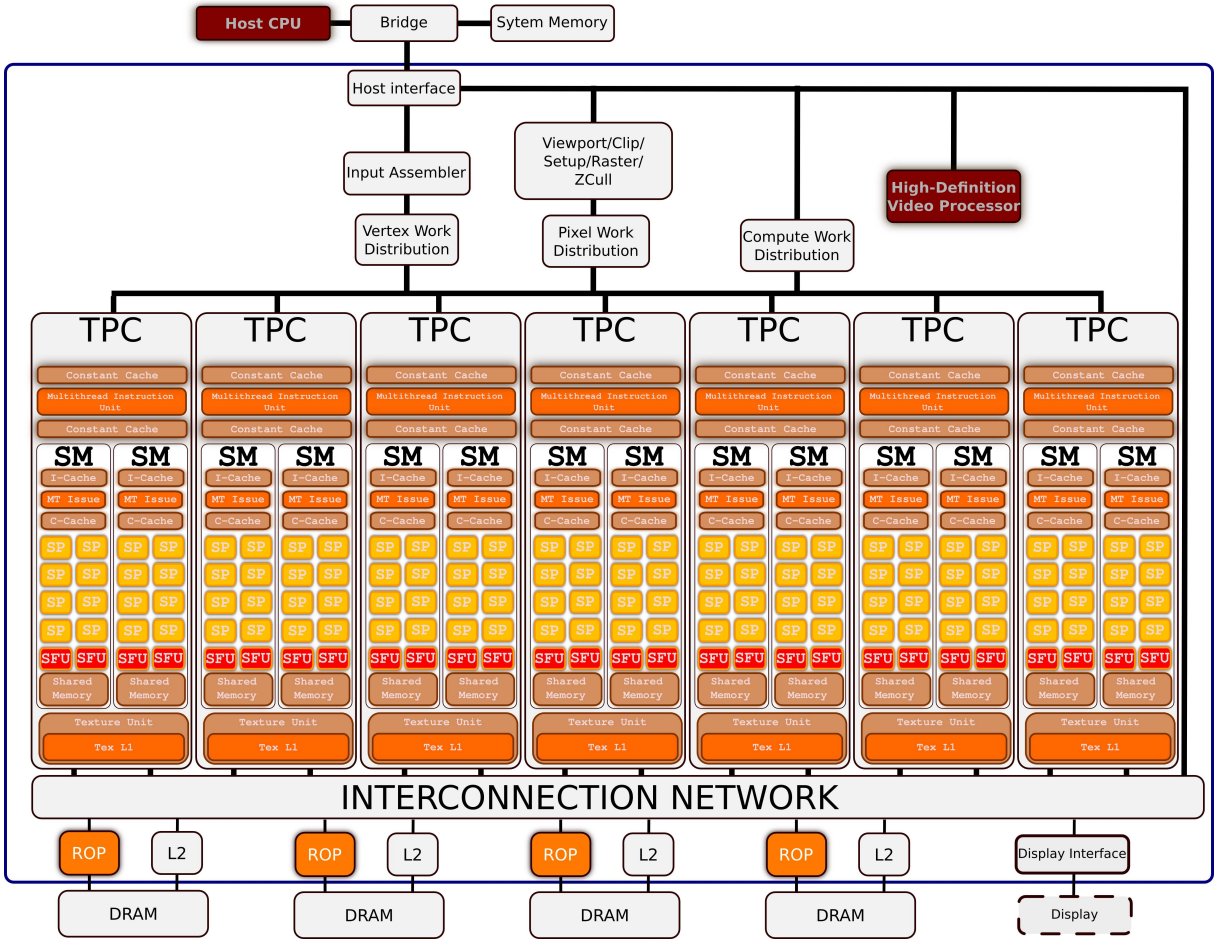

Figure 6 – GeForce 8800 Architecture reference(PATTERSON, 2014)

The architecture will be better explained in the following section.

1.3 GPU Architecture

GPU follows the vector processor architecture. It means that GPU is composed by many multiprocessors. The total amount of multiprocessor inside of the GPU may vary based on the purpose of use. For example, a GPU dedicated for games can have dozens of processor whilst a GPU in a Smartphone can have fewer than ten.

There are many reasons for building a highly multithreaded processor, and it has been tried to achieve several goals with that processor. Patterson and Hennessy highlight five of them (PATTERSON, 2014):

1. Cover the latency of memory load and texture fetches from DRAM.

2. Support fine-grained parallel graphics shader programming models.
3. Support fine-grained parallel computing programming model.
4. Virtualize the physical processor as thread and thread block to provide transparent scalability.
5. Simplify parallel programming model to write a serial program for one thread.

The subsequent subsections will introduce more deeply the GeForce 8800 architecture. See Figure 7 carefully, because all of the explanations in this section will refer to it.

Figure 7 – Detailed SM (PATTERSON, 2014)

1.3.1 Streaming Multiprocessor (SM)

The streaming processor (SP) is the first element in the multithread processor. Figure 8 shows that the multiprocessor has many SPs. It is responsible for creating and managing many threads (for example, GForce 8800 spawns up to 64 threads in some GPU) (PATTERSON, 2014).

Each SP has its own Register File (RF) associated with it (see Figure 8). The advantage of using this resource is the increase in speed. This RF can have hundreds of registers in some GPU's (in opposite of CPU's).

1.3.2 Special Function Unit (SFU)

This unit, as the name indicates, makes some special calculations such as float-point approximation to reciprocal, reciprocal square root, sine, cosine, logarithm, and some special cases of multiplication. The SFU has a special type of multiplication for handling floating point number. It can be executed concurrently with the SPs. This situation can increase the peak of computation rate up to 50% for threads with suitable instruction mixture (PATTERSON, 2014).

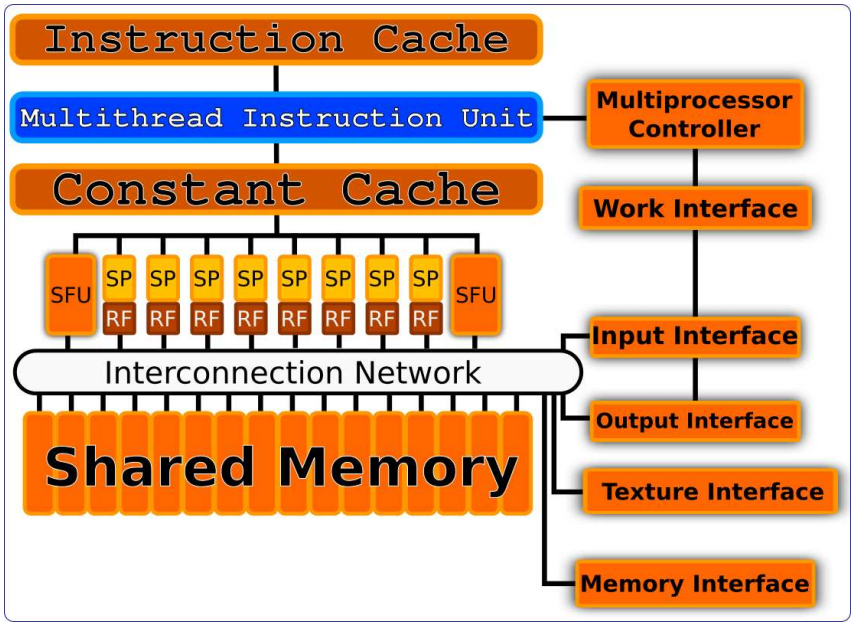

Figure 8 – Detailed GPU (PATTERSON, 2014)

1.3.3 GPU Memory Architecture

So far, it was discussed unit process and how much data it generates. Thus, a new question is raised: how to load and store the data effectively? This situation becomes clear when think about write and read pixels, the GPU can calculate thousands of pixels that will be accessed hundreds of times. If the operation in those pixels occur slowly, the GPU will lose performance (something similar to Von Neumann Bottleneck[2]).

The GPU memory system must consider five important characteristics (PATTERSON, 2014):

1. The memory must be wide, meaning there are a large number of pins to convey data between the GPU and its memory devices.

2. The memory must be as fast as possible.

3. GPU must use every available cycle to transfer data to or from the memory array.

4. Use of compression techniques.

5. Reduce the total amount of off-chip traffic needed (for it usually uses cache).

The memory system in a GPU is complex and must be considered in each part separately. This following subsection will discuss the most common elements one by one.

[2] The separation of memory and CPU (PACHECO, 2014)

1.3.3.1 Dynamic Random Access Memory (DRAM)

It is possible to find many different kinds of memories, nevertheless many concepts are the same between all of them. All memory types needs a way to receive and send data, in this sense five operations can be abstracted for of them (TOCCI; WIDMER; MOSS, 2007):

1. References a memory address by an operation for read or write.
2. Select one operation of read or write to be executed.
3. Provides the data to be stored in memory during a write operation.
4. Keep the output data in the read operation.
5. Enable or disable the memory, for making possible the read and write operation.

Random Access Memory (RAM) is one kind of memory that makes easy the access to any part of the memory. The content into the memory can be written or read based on the execution program needs. One special type of RAM is the Dynamic RAM (DRAM). The main feature of this kind of memory is the data storage density, and the low power consumption aligned with moderate speed operations (TOCCI; WIDMER; MOSS, 2007).

Usually DRAM is organized in many blocks, because of this, multiple timing requirements are imposed to make it possible. Based on that the Double-Data Rate (DDR) was introduced as a technique that transfers data on both rising and falling edges of the interface clock, this feature makes the access to DRAM more effective. This last behavior is very import for GPU, due to the huge amount of memory access which this device needs (PATTERSON, 2014).

As commented previously, the GPU has many elements that generate thousands of accesses to the memory. GPU pipeline is one of those tasks responsible for the huge memory access. Notice that each stage of the pipeline can generate more and more access to the memory. This situation causes an enormous number of uncorrelated requests. In order to minimizing this problem, GPU's memory controller keeps separated heaps of traffic bound for different DRAM banks. Next, it waits until enough traffic for a particular DRAM row is pending before activating that row and transferring all the traffic at once. Finally it is necessary to take care with this approach either, because it can generate starvation.

1.3.3.2 Cache and MMU

In order to reduce the off-chip traffic, GPU designers introduced caches. Cache has two main ideas associated with it (PATTERSON, 2014).

- Temporal Locality: Data recently used can be used again soon.

- Spatial Locality: If some data is referenced, a data near to it has a higher probability to be referenced also.

Those principles in a regular the CPU correspond to 99,99% of cache hit [3], which represents a very expressive amount of hits. On CPU, when there is a cache miss [4], one approach is just to wait for the data to be retrieved.

When the cache is inserted in the context of the GPU some experiments (PATTERSON, 2014) have shown that the total amount of cache hit can be reduced to 90%. It is a problem, because the GPU cannot wait so much. In the opposite to the CPU that can wait, the GPU must to proceed even with the cache miss. This technique is known as streaming cache architecture.

Finally, some modern GPU provide a memory management unit (MMU). It performs the translation to virtual from physical address.

1.3.3.3 Global Memory

The global memory is an external memory from SM with the main objective of offering a way for communicating different SM. For accessing this kind of memory the programmer must take care with synchronization.

1.3.3.4 Shared Memory

The shared memory is located inside of SM and it is visible to the local threads. This kind of memory provides a huge advantage for reducing the memory traffic and it is practical to build very high-bandwidth memory structures on-chip to support the read/write demands of each SM.

1.3.3.5 Local and Constant Memory

This kind of memory is visible only by a thread and it is bigger than register file, sometimes, the thread needs to support allocation for big data, because of this local memory is located on external DRAM. Constant Memory is a read-only memory located at DRAM. The constants are designed to broadcast scalar values.

[3] Find data inside of the cache
[4] Data is not in cache, and must be sought on main memory

1.4 GPU Pipeline

One of the most important characteristics related to GPU is known as a pipeline. In the section 1.1, it was introduced an old style of pipeline. In this section the pipeline issue will be addressed in more detail.

It is important to highlight that each API can handle the pipeline in a different manner. Figure 9 shows DirectX pipeline, basically, OpenCL is similar.

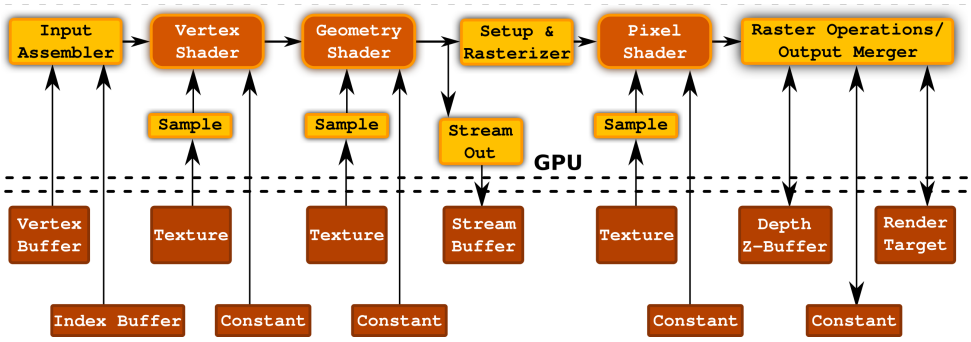

Figure 9 – DirectX pipeline

1.4.1 Input Assembler Stage

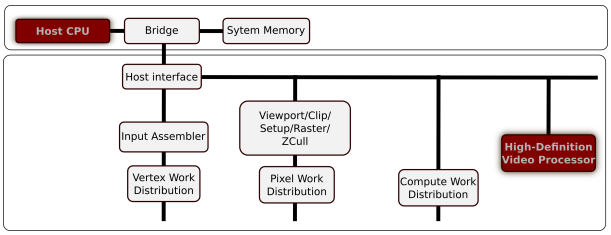

Figure 10 – Assembler decomposes instruction to GPU

The first step is responsible for receiving the data from the CPU. Usually the communication is carried out by the DMA (Direct Memory Access) (KIRK, 2014). This stage receives a sequence of vertices grouped in a primitive like points, line, triangles, and polygons (PATTERSON, 2014). Afterwards this stage translates the data for another type of data that the GPU can understand.

1.4.2 GPU stages

On the Vertex Shader Stage, the color transformation and light will be calculated vertex by vertex. This stage is a programmable one and it makes use of parallel behavior in the GPU. On the Geometry Shader stage it is possible to add and drop primitives.

The Setup & Rasterization stage generates a pixel fragment that is covered by a geometric primitive. Pixel Shader, performs for each fragment parameter, texturing, and coloring.

1.5 GPU computing and GPGPU

DirectX and OpenGL were the first graphical libraries of common use, that supports the use of GPUs. In the beginning, those libraries and GPU were designed for handling graphics. Nevertheless year after year, GPUs became more and more powerful. It is evolving faster than Moore's law predictions (KIRK, 2014). Hence many scientists tried to use all the GPU power for doing some special computations, such as simulation or particles.

When someone wants to use the GPU for another purpose different of graphics, they need to use graphics API, as OpenGL. For example, to run many instances of a program or function in the GPU it is necessary to write it like a Pixel Shader [5]. All the necessary input used by the application, must be filled with a texture image. Additionally it is necessary to cast the output. Finally it is possible to conclude, that the GPU computing was not prepared for handling this kind of application (KIRK, 2014).

The new trend of using the GPU for computing, made NVIDIA focus in this new area. The company tried to find a way to make the use of GPU easier without the need to use graphics libraries. NVidia initiated a new movement called GPGPU (General-Purpose Computation on GPU), the main idea is to use a a GPU for general-purpose computation via traditional graphics API and graphics pipeline without any need of using any graphical library (PATTERSON, 2014). As a result of many efforts, NVIDIA developed the CUDA C/C++ compiler libraries, and runtime software to allow programmers to access the resources of GPUs easily.

1.6 CPU and GPU

So far, it was described many features related to the GPU and its architecture, and how powerful it is. Someone that compares the GPU architecture with a computer architecture can notice the similarity between them. The GPU is a very complex system with many features, and it can overcome easily a normal CPU in a processing of float point (around 10 times faster (KATO et al., 2011)).

On the other hand, CPU offers many other characteristics that GPU cannot provide like: virtual memory, interruption, preemption, the controllable context of switching, and the ability to interact directly with I/O devices (KATO et al., 2011). Those characteristics are essential for developing an operating system or an embedded system. Another difference is the way that data is handled by the GPU and CPU. In the case of GPU the focus is data independence.

Based on the differences and in the advantages of each device some companies

[5] compute color and other attributes of each fragment

tried to mix GPU and CPU in the same die. Some CPUs nowadays come with GPU on the same chip. This is called Accelerated Processing Unit (APU [6]).

[6] APU a computer's main processing unit that includes additional processing capability designed to accelerate one or more types of computation outside of CPU (SMITH, 2011)

2 CUDA and OpenCL

2.1 Overview CUDA and OpenCL framework

CUDA and OpenCL are the most popular GPGPU framework, both APIs are SIMD, thus they have similar structures. Despite the similarities, each one has their own advantages and disadvantages. This section introduces each API, and a comparison between them.

2.1.1 Basic CUDA organization

Compute Unified Device Architecture (CUDA) was created by NVIDIA to provide an easy way to use GPU features in a non-graphical problems. This framework is based on the idea of "Single Instruction, Multiple Data"(SIMD), in other words, the same piece of code can executes into many threads. Basically, the user writes a code in a C-like language which will run on the GPU (ARROYO, 2011), this piece of code running on the GPU is named kernel.

All programmers who wish to take advantage of parallel computing with in GPU using CUDA, have to keep in mind two different concepts: host and device. Host is related with code that runs on the CPU and has little data parallelism. For example a code with intensive control behaviour (searching, sorting, and so on). The second one, is a device with many of massive parallel processors and with intensive data (for example, image processing). Conceptually, CUDA provides one phase with low data parallelism (CPU) and another phase with heavy data parallelism (device) (KIRK, 2014). Figure 11 shows an example of CUDA cycle, notice the top of the figure shows a serial code running on CPU, and next launching a kernel running in GPU. It is important to recognize that all threads generated by the kernel are organized in a grid.

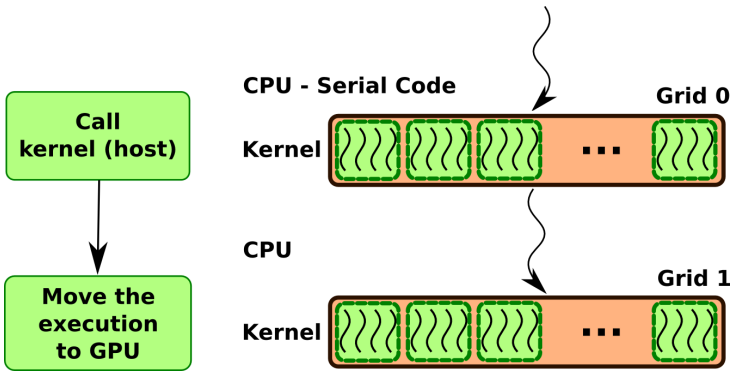

Figure 11 – CUDA cycle

When a CUDA kernel is executed on the GPU, it generates a thousands of threads. In opposite to thread in CPU, threads in GPU are much more lightweight. The reason for this characteristic is the hardware support for thread implemented inside the GPU.

As explained in the last Section 1.3.3, GPUs have their own memory. The device and host memory are separated as a result, the programmer has to allocate space on GPU and copy data from host to device. To facilitate working with device memory, CUDA creates an abstraction for representing data memory. Figure 12 shows the memory used by CUDA, and the basic flow to use it. Notice that global memory is connected between host and device. For make the allocation of memory in GPU and copy data from host to GPU, it is used cudaMalloc and cudaMemcpy. Those functions are two simple examples to illustrate how the CUDA API works.

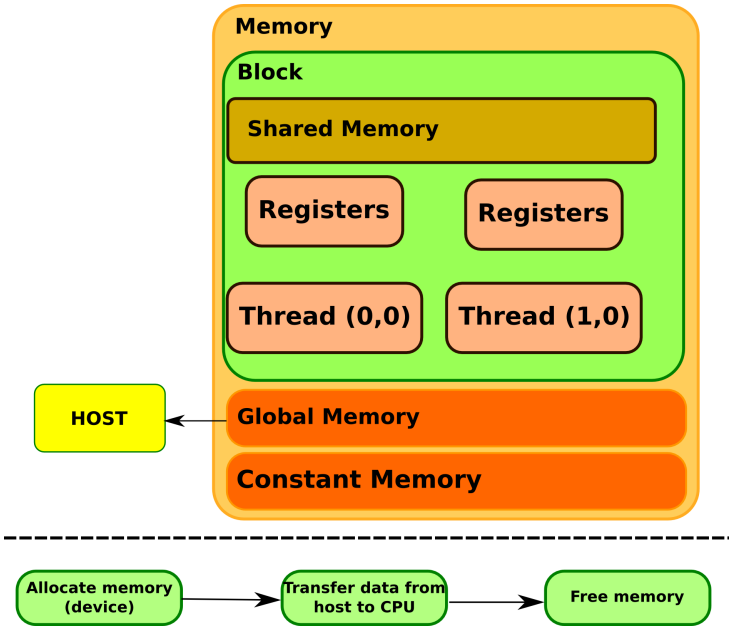

Figure 12 – CUDA memory model

Figure 13 shows how CUDA organize their threads. Each kernel, has their own grid, each one subdivided into block, and finally it is possible to access the thread part.

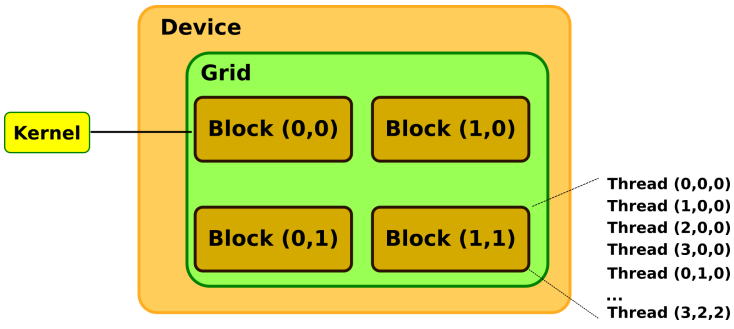

Figure 13 – CUDA thread organization

2.1.2 Basic OpenCL organization

CUDA is a framework that works only on NVidea devices. This is an important restriction that makes this API limited from the portability view point. In this sense, some years ago Apple tried to explore GPU-acceleration independently of the device. In order to achieve this goal they started the OpenCL project. Nowadays, Khronos Group maintains the specification of this API (ARROYO, 2011).

OpenCL defines a variant of C99. This means that it is not needed to use a specific compiler, however, it is required to link it with the correct OpenCL library. The most powerful characteristics of this API are the multi-platform, as a result, OpenCL can run on many different devices.

Figure 14 illustrates the basic execution flow of OpenCL code. Firstly, it is the "setup", this step responsible to configure the platform, device and the context whose host will use interact with the device (BENEDICT et al., 2013). This is an important difference between CUDA and OpenCL, because this kind of configuration is driven by the device driver and happens transparently for CUDA. These configurations allow OpenCL to know which device will be used at runtime, and then execute specific code for the current device vendor. This low-level approach gives more flexibility in the sense of running the same code into many different platforms, on the other hand it makes the API much more verbose (ARROYO, 2011). The second part of setup creates an abstraction to communicate host with device, this is named "context". The third part, defines the memory model hierarchy whose kernel will use. The last chunk defines how to map the concurrent model of the hardware.

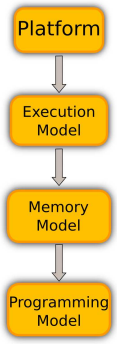

Figure 14 – OpenCL Specification

Similar to CUDA, OpenCL defines the idea of kernel that will execute on device. The smaller granularity of parallelism in OpenCL is called work-item (equivalent to thread in CUDA) and it is responsible for executing a snippet of code that should be executed in parallel.

Context has the main objective of coordinating the mechanism between host and device, manage memory objects and keep track of registered programs (BENEDICT et al.,

2013)). OpenCL has an abstraction named command queue, which is responsible for the mechanism of communicating host action with device.

To manage memory, OpenCL API implements two kinds of memory: buffer and image. The first one is a simple memory in the device, located at the Global memory. Image is another kind of memory optimized for fast access. The drawback of this type of memory is related to the that of some devices cannot support this feature. As a comparison, buffer memory has a similar behavior of cudaMalloc and the image is similar to pinned memory.

OpenCL has a special abstraction of memory. This abstraction was designed to embrace the differences between all vendors memories. Figure 15 illustrates the overview of OpenCL memory model, notice the similarity with CUDA memory model. Global memory is responsible for handling data transfer between host and device, and only has a small piece inside of it to handle constants. All work-items are organized into the work-group, which has local memory visible for all threads. Finally, all work-items have their own private memory. This granularity, provides great flexibility to OpenCL API and allows it to run into a different kind of devices from different vendors.

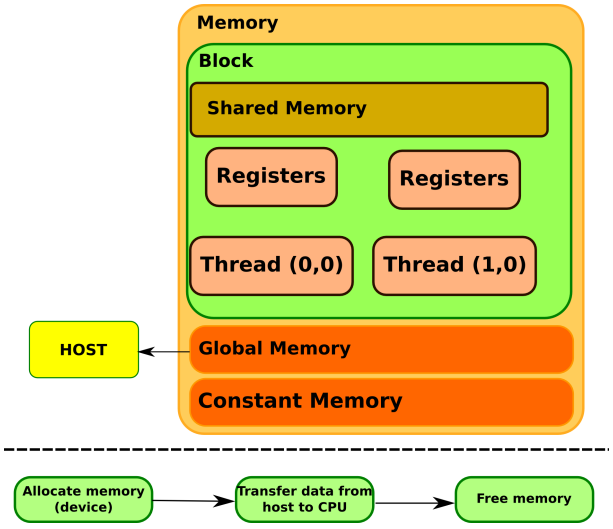

Figure 15 – OpenCL memory abstraction

One of the most interesting characteristic of OpenCL is the creation and build process. First of all, kernel code is loaded as a string with a function called clCreate-ProgramWithSource. If code is correct, OpenCL creates an object file. Afterwards, the program is "compiled"at runtime by clBuildProgram, and finally it is ready to execute.

Figure 16 shows the basic flow to run the kernel. The first is to create a kernel reference, cl_kernel is the data structure used to handle it. Secondly, if the kernel has a parameter the programmer has to explicitly set them one by one with the function clSetKernelArg. Finally, for launching the kernel it is necessary to initialize it with clEqueueNDRangeKernel.

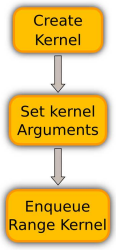

Figure 16 – Kernel execution flow

3 Delimitation of the subject

3.1 Purpose

Firstly, this section aims to introduce the related works that inspired this project and later it overall idea. Secondly, it is possible to verify an analysis of the proposal feasibility. Finally, it is possible to find the scope delimitation that the author wants to achieve.

3.1.1 Related work

3.1.1.1 Augmenting Operating System with the GPU

Weibin Sun and Robert Ricci wrote a paper titled *Augmenting Operating System with the GPU*(SUN, 2011). Basically they argued that GPU can be used to speed up some kernel functions. For this reason, they had a great idea of using GPU as a coprocessor from an OS kernel. They proved their thoughts by implementing a device driver for make GPU behave as a coprocessor (better known as KGPU).

As a result, there are new subtle issues that demand extra attention. Indeed, problems related with the time elapsed to transfer data from memory to GPU and vice-versa becomes a problem to KGPU. To put it in another way, it is fundamental to realize that this action needs a bus access. Actually, for handling this situation it is used DMA [1] and it is a huge bottleneck. See Figure 17

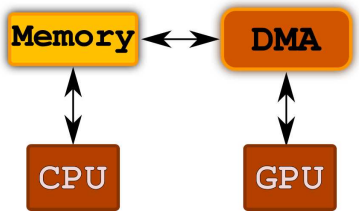

Figure 17 – Bottleneck: CPU and GPU communication

For handling the problem of data transfer between CPU and GPU, the authors addressed the issue using *pinned memory* which is a CUDA feature for faster memory access. This kind of memory is faster because it is a mmap [2] resource, however it has the disadvantage of locks the specific physical page.

By the same token other problems by using GPU as a coprocessor arises, but at this time relates to the Launch Overhead. Any kind of device needs to be set-up before

[1] Direct Memory Access, for more information going back to theoretical rationale.
[2] http://man7.org/linux/man-pages/man2/mmap.2.html

using, and GPU follows the same idea. It based on that, it is easy to see that GPU takes some time to be ready for use. This waiting time, creates an overhead that degraded a lot the performance and it must be handled.

Based on launch overhead, the paper suggested the creation of new GPU Kernel execution model which the authors called Non Stop Kernel (NSK). The main idea behind the design of NSK is keeping it small and launch it only once without terminating it. See the Figure 18 that illustrates the NSK:

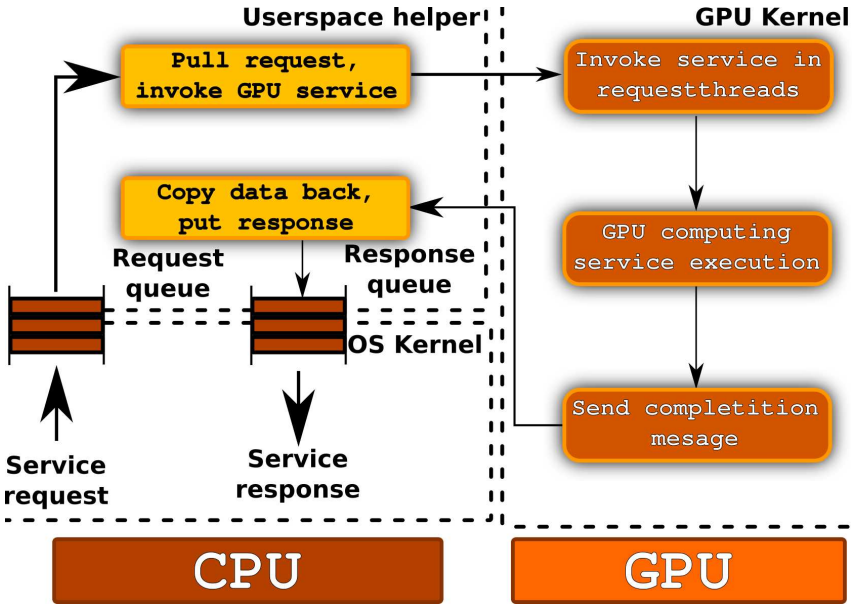

Figure 18 – KGPU architecture

The paper described on the current section has given the main motivation for the start of this work.

3.1.1.2 Timegraph, GPU scheduling

The GPU was originally designed to be used by electronic games. However new applications have began to make use of this device, e.g., many GUI and multimedia applications are processed on GPU. Additionally, many engineers and scientists had become more interested in the parallelism provided by it. In this sense, the idea of sharing GPU with many softwares begins to make sense and took the attention to the problem of improving the scheduler for this device. Inspired by this issue, a group of research tried to solve this problem by creating a new GPU scheduler called *Timegraph* (KATO et al., 2011).

The idea of *Timegraph* can be summarized in the creation of a new GPU scheduler. For real-time environment that adds prioritization to the processes running on it, and provides a better separation between applications. Nevertheless, for implementing this

idea, a device driver was created that handles the communication to GPU and one module connected to it that has the *Timegraph* algorithm. See Figure 19.

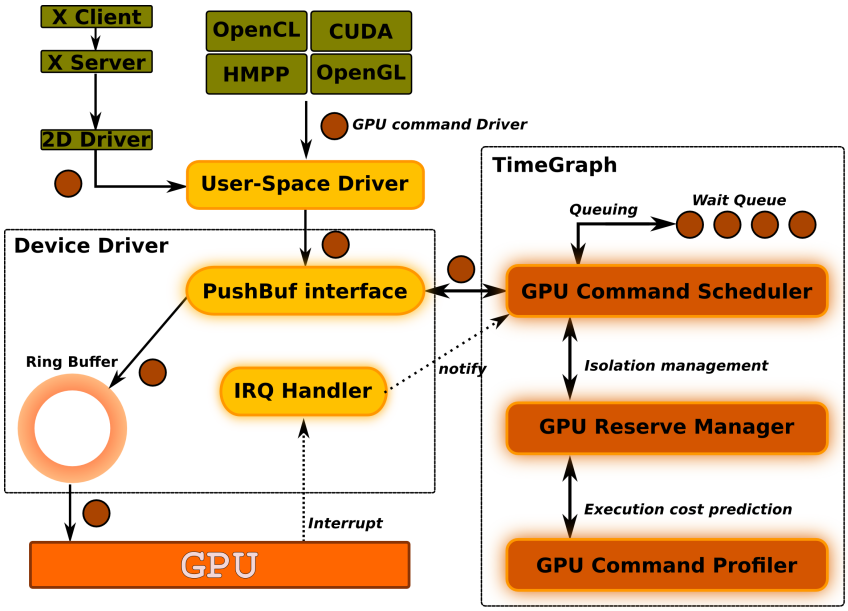

Figure 19 – Timegraph architecture (KATO et al., 2011)

At the top of the Figure 19 it is possible to see many applications that generate GPU commands. Afterward those instructions are grouped and delivered to the Device Driver. It passes through *Pushbuf* interface, which is connected to *GPU command scheduling* within *Timegraph*. In this module is handled the policy of scheduling and reserving GPU. *Timegraph* provides two types of scheduler algorithms, one specialized to increase the predictable response time and another one for high-throughput. See the Figure 20 and 21 below that illustrates both schedulers:

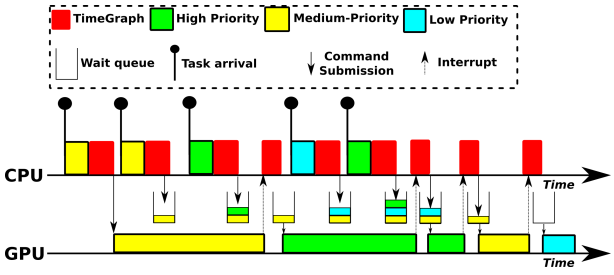

Figure 20 – PRT

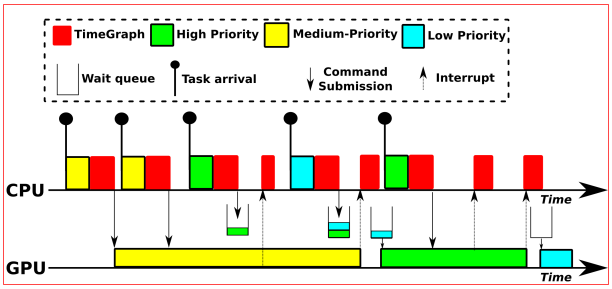

Figure 21 – HT

Figure 20 illustrates the *Predictable-Response-Time* (PRT) policies. It receives tasks with different priorities to be scheduled; all the processes arrived are queued, and based on their priorities the queue is rearranged. Notice that only one task is sent to GPU at time. Whenever high priority task arrives it is put on top of the queue, however this operation has a price: the time caught for decision making. This schedule has a good performance and it is very predictable, but it suffers with system degradation caused by the decision making.

The Figure 21 shows the *High-Throughput* (HT) policy: it focuses on providing high-throughput. PRT and HT policy are similar, but HT allows GPU command group to be submitted to GPU immediately. It increases the throughput, on the other hand, it degrades the predicting response because some high priority task needs to wait.

Another characteristic that *Timegraph* provides is the reservation mechanism. It provides two strategies: *Posterior Enforcement* (PE) and *Apriori Enforcement* (AE). PE enforces GPU resource usage after GPU command groups are completed without sacrificing throughput. AE enforces GPU resource usage before GPU command groups are submitted using prediction of GPU execution cost at the expense of additional overhead.

3.1.1.3 GPGPU spatial time

The case for GPGPU spatial multitasking (ADRIAENS et al., 2011) is a paper that discusses the current techniques for GPU scheduling, and it proposes a different approach to the scheduler GPU. Usually when one application runs on GPU it has 100% of GPU dedicated only to it, and it just releases the device if it explicitly yield the GPU. Bearing it in mind, some malicious applications may take the control of GPU and never free it. *Windows Vista* and *Windows 7* addressed this issue by imposing time limits to GPU computation: if the application fails to yield the Operating System eliminates GPU computation.

Usually GPU scheduler is based on time and preemption policy, and it is possible to notice that this approach has some overhead associated with context switches (saving and restoring). Additionally, this approach is not good for this kind of device because of their nature. As introduced in the second section, GPU is highly parallel with many processor working together. In order to take some advantage of this device, it is necessary to use a different way to schedule the processes passed to it.

To address the issue of temporal multi task on GPU, it was proposed the *Spatial Multitasking*. Basically this technique allows many GPU kernels[3] executing at the same time using in a subset of the GPU resource. Figure 22 represents three different methods. The first and the second image doesn't share the resource and represents the traditional

[3] The kernel term is used in the CUDA context here.

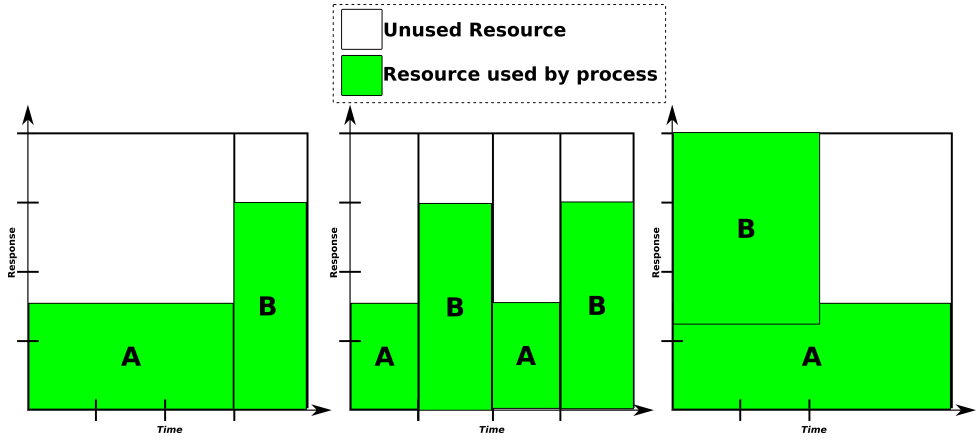

Figure 22 – Cooperative, Preemptive and Spatial

way of GPU scheduling. In opposite way, the last image shows one approach that shares the resource based on spatial.

3.1.2 Purpose

The ideas described in the previous section mixed with many other acquired in a long time of study have guided this work. This project has two main objectives:

1. Make KGPU work with CUDA and OpenCL.
2. Organize KGPU code, and verify the architecture.

KGPU provides the main idea for the current project. The concept of GPU as a coprocessor is the core idea of this project, nonetheless two things were changed: the use of CUDA and the software engineering techniques. Firstly, instead of use CUDA it is used OpenCL because of the portability. Secondly, this project tried to apply many software engineering techniques to make KGPU more readable and to create the bases to build other projects. Finally, this work expects to verify practicability in the use of the KGPU architecture.

4 Methodology Procedures and Technical

4.1 Methodology design

With the intention of classifying the current research based on the used methods, this section describes the methodology. Before analysing the methodology, it is necessary to define methods and research. Lakatos and Marconi (MARCONI; LAKATOS, 2003) argued that research methods, is a set of systematic activities and arguments that guided to the generation of legitimate knowledge. Finally Silva and Menezes (SILVA; MENEZES, 2005) said that research means finding an answer to proposed questions.

Based on Table 1, this work can be classified as applied, quantitative, exploratory, bibliographies, and experimental.

4.2 Questions and objectives

The aim of this research, is the attempt to answer some or all the following questions:

1. Is it possible to use OpenCL instead of CUDA in KGPU?
2. Is it possible to keep CUDA and OpenCL in KGPU?
3. KGPU architecture is good enough to be adopted as an alternative to use GPU as a coprocessor?

For help to answer those questions, some objectives were defined:

1. Study and understand GPU architecture.
2. Reduce KGPU services to simple example.
3. Refectory KGPU code.
4. Translate CUDA code to OpenCL code.

Classification	Type of research	Description
Nature	Basic	Research that aims to generate new knowledge and useful advance in science without concerned with practical applications.
	Applied	Aims to generate knowledge for practical applications guided by specific solution for problems.
Approach	**Quantitative**	Research with a focus on statistical studies target to the quantification of the study object.
	Qualitative	In this kind of research, the process of interpretation of the phenomenon and the passing of meaning are basic in the research process. The data are analysed in intuitively.
Objectives	**Exploratory**	Aims to provide a deeply into the studied subject, it tries to make explicit the issue or build hypothesis.
	Descriptive	Aims to describe the characteristics of some specific population or phenomenon, or the relationship between variables.
	Explanatory	Aims to identify or determine the contribution for the phenomenon occur. Dive into the knowledge because it explain the "why"of the things.
Technical Procedures	**Bibliographic**	When it is created based on the materials already published, mainly by books, articles, and materials provided on the Internet.
	Document	When it is created based on materials, that have not received the analytic treatment.
	Experimental	When it is determined an object of study. Firstly it is selected a variable capable of influence, secondly defines the way of controlling it, and finally take a look at the effects made by the variable.
	Collection	When the research is focused on asking for people, whose the objective is know their behaviour.
	Case study	Deeply study of one or a few objects, that allows a wide knowledge details about the object.
	Action research	The researchers and the members of the experiment are involved in a cooperative way.
	Partner Research	When the research is developed based on the interaction between the researchers and the members of the investigated situation.

Table 1 – Classification of Researches (Based on (SILVA; MENEZES, 2005))

4.3 Hypothesis

This work aims to study the feasibility of replace CUDA to OpenCL in KGPU and verify the possible adoption of KGPU as a coprocessor. For it, two hypotheses were defined:

1. ***OpenCL can replace CUDA with similar performance***. OpenCL has similar characteristics with CUDA, however it has much more advantages because of the portability.

2. ***The use of KGPU as an API to make use of the GPU as a coprocessor***. If KGPU has a good architecture, it can be extend for future works.

4.4 Hypothesis test

KGPU was written using CUDA, and already has a few services implemented. Thus, to verify if the translation of CUDA to OpenCL is right, it is possible to compare the output generated by the original service with CUDA and compare it with the KGPU output with OpenCL. This test aims to verify, the correctness of the output without taking care of performance issues. KGPU has many complex services, because of this it was selected a set of simple tests whose the objective is to verify the correct behaviour of the KGPU functions.

The same hypothesis can be applied to the case of applying software engineering techniques to the KGPU code. However, to verify this hypothesis the use of CUDA or OpenCL does not affect the comparison.

4.5 Devices

This project was developed in a computer, with following the configuration:

- 4Gb of RAM memory;
- Intel Core i7 2.93GHz;
- Hard dist sata, 1.4TB.

NVidia 9500 GT was adopted for using in the project. Figure 23 illustrates this device. Follow the main characteristics of 9500GT:

- 32 processor core;

- 550 MHz of Graphics clock;
- 1400 MHz of processor clock;
- Memory clock of 800MHz (GDDR3) and 500MHz (DDR2);
- 256/512 MB (GDDR3) and 512 MB (DDR2) of standard memory.
- 128 bits Memory Interface Width

Figure 23 – GeForce 9500 GT

4.6 Environment Configuration

All the project was developed in GNU/Linux, following the software the specification of system:

- Operating System Kubuntu 14.04, 64 bits;
- Kernel 3.13.0-48;
- NVIDIA driver 331.113;
- nvcc version 5.5;
- gcc version 4.8.2;
- Make 3.81;
- C99;

Finally, it is possible to see all changes made in KGPU code in the URL:
<https://www.github.com/rodrigosiqueira/kgpu_fga>
This repository was forked from the original KGPU code, and used as a base to develop this work.

5 Achieved Result

5.1 Results

This section presents the result of the research in details. Firstly, it is introduced the KGPU code structure and the current state of the original project. Afterward, it explains the software engineering techniques applied to it. Finally, it is described the essential points of the insertion of OpenCL.

5.1.1 State-of-art of KGPU

KGPU was a part of the Ph.D project developed by Weibin in the Utah University, and it had many of the updates in the period of his studies. Nowadays, the project is stopped mostly because Webin was the only programmer on the project. To be more accurate, it has more than two years that the repository does not have any update. Just for illustration, Figure 24 shows the last commits on the master.

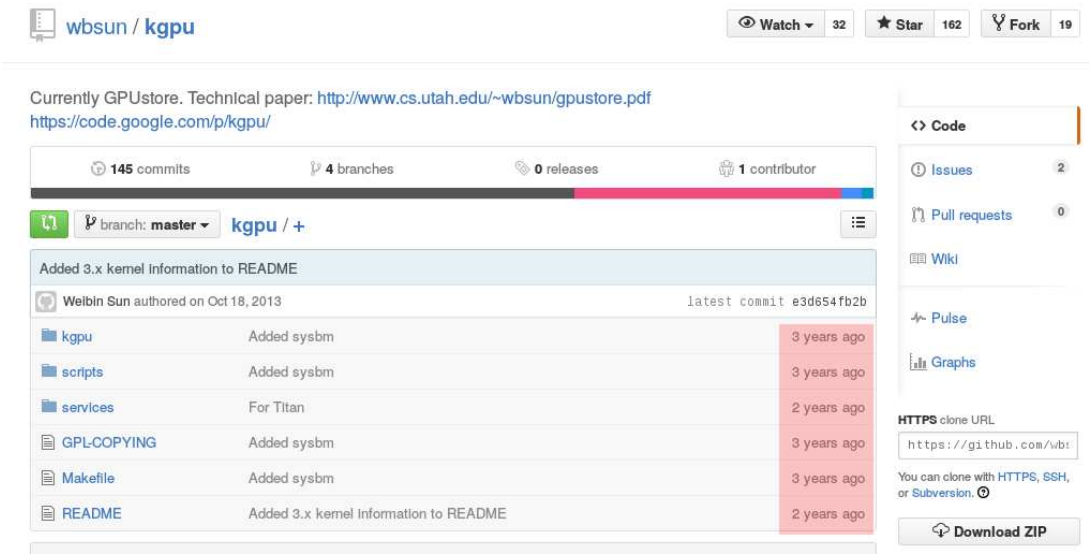

Figure 24 – Last commit KGPU

Look at the top of Figure 24, it is possible to notice that 162 people "stared"the repository, and Figure 25 shows the total forks of the project, which indicates that many people are interested in KGPU. Notwithstanding, after verifying all the user's forks, it is possible to notice that nobody made many change in the code. This raises the question: why so much people are interested in the project, but no one did a change? We believe that part of this, is related to the project structure.

Figure 25 – Forks

After a deeply study at the code to try to identify the all the major problems, it was possible to realize that KGPU are not organized. As a simple example of the bad organization, all the code was put together in the same folder (Figure 26). The code does not have any logical separation and it is very hard to read and understand. See below, the top 5 list of code problems found in KGPU.

1. Dreadful variable and functions names;
2. No documentation of any kind;
3. Many structures spread around many files;
4. Thousands of macros;
5. Many code dependencies to CUDA.

Other weak point of the project is the procedure of launching KGPU. Figure 27 describes all the required steps for making KGPU run and waiting for execute services. One thing that make all this process painful and dangerous, it is the need of execute all of those steps as a super user. The first step for run KGPU, is inserting the kgpu.ko module

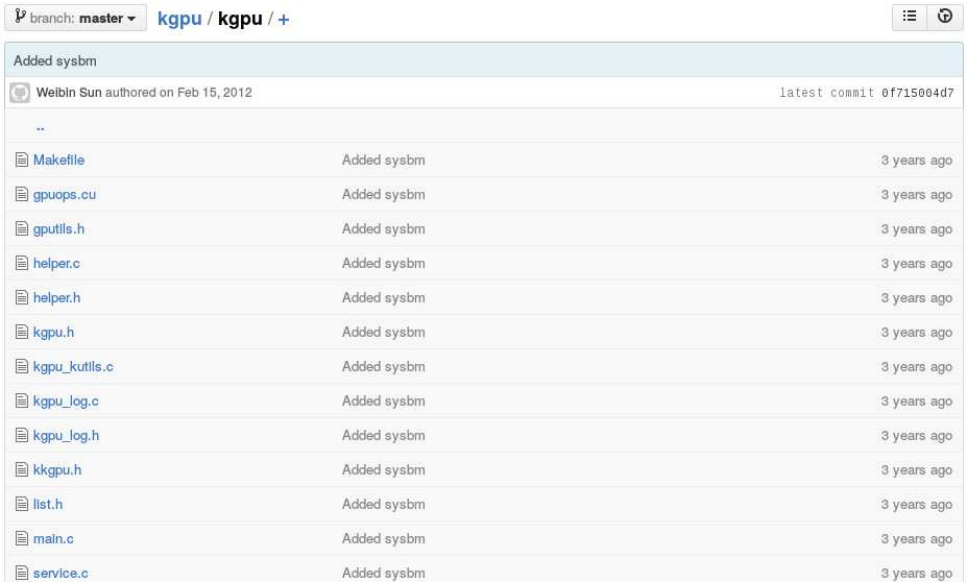

Figure 26 – Code organization

by the command:

```
1 insmod ./kgpu.ko
```

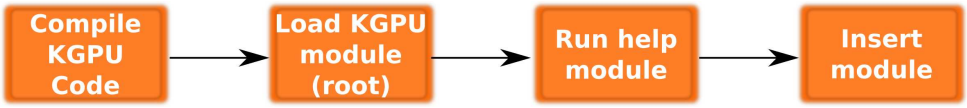

Figure 27 – KGPU launching process

Next, it is required to initialize the helper service (user mode). To do it, the command below have to be executed:

```
1 ./helper -l ‘pwd‘
```

The last step it is insert the service on the memory, this step can vary based on the name of the service. The command below, show one simple example:

```
1 insmod ./callgpu.ko
```

Finally, the last critical problem with KGPU is related to its compilation. KGPU is a very old project without updates, and when it was designed, the kernel 2.7 was the main version. Nowadays kernel 3.x (and upper) became more popular, and latest versions does not have many support anymore. In this sense, it was really important to port the project for the new versions.

5.1.2 Refactoring

The process of refactoring KGPU brought a challenge, mostly because of dependencies between files and compilation problems to run the project in kernel 3.x. The first step of refactoring, consisted to make KGPU compile and run on new versions of the GNU/Linux kernel, to achieve this goal it was needed to update some pieces of code. Code 5.1 shows a piece of code written by Weibin which run on kernel 2.6, but not in the new version. Code 5.2 shows the needed changes to make KGPU run on newer kernel versions.

Code 5.1 Those flags not exists on new versions of kernel

```
1     vma->vm_flags |= VM_RESERVED;
```

Code 5.2 Changed flag to run in new kernel version

```
1   vma->vm_flags |= (VM_IO | VM_LOCKED | (VM_DONTEXPAND | ↩
        VM_DONTDUMP));
```

Make KGPU capable to run on newer Kernel version created the basic requirement to change the code, once it is possible to test the changes in the code and verify if the output still correct. The second step, was renamed variables and functions with meaningful names and correct the indentations to make easier to understand the code.

The third step consisted to organize all files in a logical way, because KGPU did not have any kind of logical separation between files and folder. To make simple to understand the code, it was created one folder for each part of the architecture described on Section 3.1.1.1. KGPU has three main parts: Kernel GPU, user space, and operating system service (SUN, 2011). Despite of this conceptual division, the code does not reflect it as Figure 26 can illustrates. In order to improve the implementation, it was separated all the header files and changed all the folder organization for making it more consistent with the theory. Figure 28 illustrates the new folder arrangement of KGPU, notice the clear division of the code based on the KGPU architecture. Additionally, it was created, new files to make the division of the code better.

Finally, it was added some simple features (but powerful) to create the bases for the code grows in the future. The first one, is the addition of continuous integration with Travis CI [1], this is an important feature for keeping the project always working right. Secondly, it was registered the project in waffle [2] website for help to manage the project.

[1] <https://travis-ci.org/rodrigosiqueira/kgpu_fga>
[2] <https://waffle.io/rodrigosiqueira/kgpu_fga>

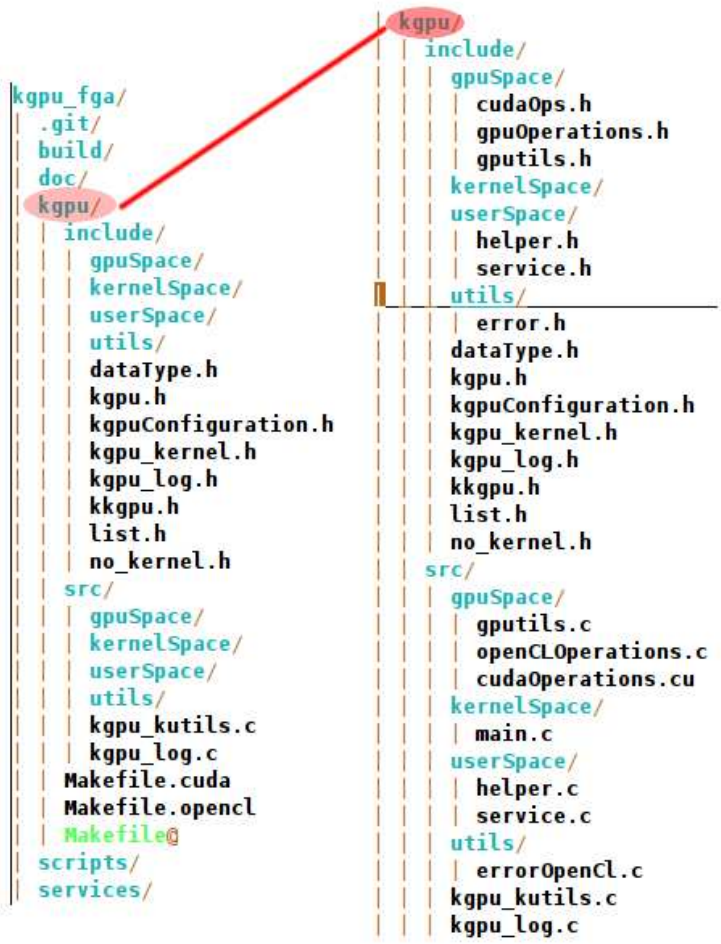

Figure 28 – New KGPU organization

Lastly, it was added Doxygen to the project and made the current code API description online via gh-pages in github [3].

5.1.3 Add OpenCL API

5.1.3.1 Overview of GPU code in KGPU

KGPU was designed to use CUDA framework, and originally the author did not have any intention to use OpenCL. Hence, the code was not prepared to receive others APIs, and this characteristic made the task to add new a library complicated.

Code 5.3 is the header file which keeps function's signature used by KGPU, the implementation of those functions can be found in "cudaOperations.cu"with CUDA implementation of each of those signatures. All the main function of GPU space was implemented inside this file, follows a brief description of each one:

[3] Gh-pages is a simple way to make the code API available in Internet

Code 5.3 All KGPU CUDA operations

```
1  extern "C" void gpu_init();
2  extern "C" void gpu_finit();
3
4  extern "C" void * gpu_alloc_pinned_mem (unsigned long size);
5  extern "C" void gpu_free_pinned_mem (void * p);
6
7  extern "C" void gpu_pin_mem (void * p, size_t sz);
8  extern "C" void gpu_unpin_mem (void * p);
9
10 extern "C" int gpu_alloc_device_mem (struct kgpu_service_request
       * sreq);
11 extern "C" void gpu_free_device_mem (struct kgpu_service_request
       * sreq);
12 extern "C" int gpu_alloc_stream (struct kgpu_service_request *
      sreq);
13 extern "C" void gpu_free_stream (struct kgpu_service_request *
      sreq);
14
15 extern "C" int gpu_execution_finished (struct
      kgpu_service_request * sreq);
16 extern "C" int gpu_post_finished (struct kgpu_service_request *
      sreq);
17
18 extern "C" unsigned long gpu_get_stream (int sid);
```

1. gpu_init: Initialize KGPU elements. Called to start KGPU.
2. gpu_finit: It is called to free memory after the service finish.
3. gpu_alloc_pinned_mem: Allocated a pinned memory based on size required. Usually, this size is defined at KGPU_BUF_SIZE located at "kgpuConfigurations.h".
4. gpu_free_pinned_mem: Release previously memory allocated by gpu_alloc_pinned_mem.
5. gpu_pin_mem: Attach address to memory.
6. gpu_unpin_mem: Detach pinned memory.
7. gpu_alloc_device_mem: Allocate device memory.
8. gpu_free_device_mem: Release device memory previously allocated.

9. gpu_alloc_stream: Take the index of the last stream allocated.
10. gpu_free_stream: Release the stream.
11. gpu_execution_finished: Remove stream from GPU.
12. gpu_post_finished: Verify if the processing in GPU finished.
13. gpu_get_stream: Get the last stream used.

To add OpenCL in KGPU without change all the original code, it is necessary to find a way to reimplement the functions above with a little impact on the other parts of the code. The next sections describes with details some functions and how the translation was made.

5.1.3.2 Changing data structures

One important step to correctly convert CUDA code to OpenCL is changing the data structure used by CUDA. The configure kernel dimension is the easier one to be replaced, CUDA use dim3 as a data structure that can be replaced by size_t[3] in the case of OpenCL (ARROYO, 2011). Secondly, KGPU uses cudaStream_t data structure to keep track of each stream is in use. The equivalent of it on OpenCL is cl_command_queue, see Code 5.4 with CUDA implementation and 5.5 with new OpenCL code.

Code 5.4 CUDA data structure on KGPU

```
1 #define MAX_STREAM_NR 8
2 static cudaStream_t streams[MAX_STREAM_NR];
3 static int streamuses[MAX_STREAM_NR];
4
5 static const dim3 default_block_size (32, 1);
6 static const dim3 default_grid_size (512, 1);
```

Code 5.5 OpenCL data structure on KGPU

```
1 #define MAX_STREAM_NR 8
2
3 openCLRuntimeData * openCLData = NULL;
4
5 float tmp_memory_test[KGPU_BUF_SIZE * 2];
6
```

```
7 cl_command_queue streams[MAX_STREAM_NR];
8 static int streamuses[MAX_STREAM_NR];
9
10 static const size_t default_block_size[3]; // 32, 1
11 static const size_t default_grid_size[3]; // 512, 1
```

Another important step for OpenCL code is the platform configuration, device, and context (See 2.1.2). Unfortunately, KGPU does not have any configuration step for handling GPU setup, because CUDA drivers already take care of all low level details in a transparent way. Hence, it was created a new data structure for handling OpenCL information. Code 5.6 shows the new data structure, basically it keeps the current context and device to be used in the implementation.

Code 5.6 OpenCL data structure

```
1 #define PAGE_SIZE 4096
2 #endif
3
4 typedef struct _openCLRuntimeData
5 {
6   cl_context  context;
7   cl_device_id * devices;
8 }openCLRuntimeData;
9
10 typedef struct
11 {
12   cl_mem userVirtualAddress;
```

5.1.3.3 Changing functions

After map the data structure and change it for use OpenCL, it was necessary to replace CUDA code inside functions. However, as explained in Section 2, OpenCL needs a special initialization before trying to use the device. In order to configure the environment, it was created a static function whose the main objective was prepared OpenCL to execute. Code 5.7, show the implementation of the start point of OpenCL. This function should be called just once and have to be invoked before any other KGPU operation.

Code 5.7 Initialize OpenCL

```
1  static int initializeOpenCL()
2  {
3    cl_int status = 0;
4
5    if (openCLData)
6    {
7      free(openCLData);
8    }
9
10   openCLData = (openCLRuntimeData *) malloc(sizeof(↩
       openCLRuntimeData));
11   if (!openCLData)
12   {
13     printErrorMessage(NO_SPACE_ON_HOST);
14     return NO_SPACE_ON_HOST;
15   }
16
17   cl_platform_id * platforms = NULL;
18
19   status = initializePlatform(&platforms);
20   if(status != CL_SUCCESS)
21   {
22     goto firstLevelOfClean;
23   }
24
25   cl_uint numDevices = 0;
26   status = initializeDevice(platforms, openCLData, &numDevices);
27   if (status != CL_SUCCESS)
28   {
29     goto secondLevelOfClean;
30   }
31
32   status = initializeContext(openCLData, numDevices);
33   if (status != CL_SUCCESS)
34   {
35     goto thirdLevelOfClean;
36   }
37
38   return status;
39
40   thirdLevelOfClean:
41     free(openCLData->devices);
```

```
42
43   secondLevelOfClean:
44     free(platforms);
45
46   firstLevelOfClean:
47     free(openCLData);
48
49   printErrorMessage(status);
50   return status;
51 }
```

The gpu_init was one of the first functions to be translated. Figure 5.8 show the original implementation, and Code 5.9 presents the OpenCL code for this function. This function creates the necessary streams and memory to be used in GPU. Figure 5.8 shows function alloc_dev_mem, whose the main task is allocating memory inside GPU by calling cudaMalloc. A similar function in OpenCL to reserve space is called clCreateBuffer, and as can be verified in Figure 5.9 it was replaced in KGPU code.

Code 5.8 Initialize CUDA OpenCL

```
1  void gpu_init()
2  {
3    int i;
4
5    devbuf.uva = alloc_dev_mem(KGPU_BUF_SIZE);
6    devbuf4vma.uva = alloc_dev_mem(KGPU_BUF_SIZE);
7
8    //TODO: Improve it.
9    fprintf(stdout, ">>>>> gpuops.cu: GPU INIT.\n");
10
11   for (i = 0; i < MAX_STREAM_NR; i++)
12   {
13     csc( cudaStreamCreate(&streams[i]) );
14     streamuses[i] = 0;
15   }
16 }
```

Code 5.9 Initialize KGPU OpenCL

```
1  void gpu_init()
2  {
3    int i = 0;
4    cl_int status = 0;
5
6    if (!initialized)
7    {
8      status = initializeOpenCL ();
9      if (status != CL_SUCCESS)
10     {
11       return;
12     }
13     initialized = 1;
14   }
15
16   deviceBuffer.userVirtualAddress = clCreateBuffer(openCLData->↩
       context, CL_MEM_READ_WRITE, KGPU_BUF_SIZE, NULL, &status);
17
18   if (status != CL_SUCCESS)
19   {
20     printErrorMessage(status);
21     return;
22   }
23
24   deviceBufferForVMA.userVirtualAddress = clCreateBuffer(↩
       openCLData->context, CL_MEM_READ_WRITE, KGPU_BUF_SIZE, NULL↩
       , &status);
25
26   if (status != CL_SUCCESS)
27   {
28     printErrorMessage(status);
29     return;
30   }
31
32   for (i = 0; i < MAX_STREAM_NR; i++)
33   {
34     streams[i] = clCreateCommandQueue (openCLData->context, ↩
         openCLData->devices[0], ↩
         CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE, NULL);
35     streamuses[i] = 0;
36   }
37
```

```
38   return;
39 }
40
41 void gpu_finit()
42 {
43   int i = 0;
44   cl_int status = 0;
45
46   //CL_INVALID_MEM_OBJECT
47   status = clReleaseMemObject(deviceBuffer.userVirtualAddress);
48   if (status != CL_SUCCESS)
```

After finishing the working with KGPU, it is extremely recommended releases the memory and GPU. To do it, gpu_finit is called and release all memory previously allocated. The translation of this function was straightforward, because cudaFree and clReleaseBuffer doing exactly the same thing.

One important point about KGPU, is the pinned memory. Code 5.10 shows how to allocate pinned memory in a CUDA style, and one important thing to realize about this step is the returned pointer. Usually, CUDA uses an *float ** to reference memory and return this kind of pointer. However, OpenCL works with a special data structure named cl_mem. Due to the difference between those styles, it is required to work with cl_mem in some way that function signature keeps the same. The solution for this problem, was the creation of a new buffer with the flag CL_MEM_USE_HOST_PTR and afterward the use clGetMemObjectInfo function for retrieving the address pointer to the function. Figure 5.11 shows the OpenCL implementation of the same function.

Code 5.10 Initialize CUDA OpenCL

```
1 void * gpu_alloc_pinned_mem(unsigned long size)
2 {
3   void * h;
4
5   //TODO: IMPROVE IT
6   fprintf(stdout, ">>>>> gpuops.cu: GPU ALLOC PINNED MEMORY\n");
7   csc( cudaHostAlloc(&h, size, 0));//cudaHostAllocWriteCombined)↩
        );
8   return h;
9 }
```

Code 5.11 Initialize KGPU OpenCL

```
1  if (!openCLData)
2  {
3    if (!initialized)
4    {
5      status = initializeOpenCL ();
6      if (status != CL_SUCCESS)
7      {
8        return;
9      }
10     initialized = 1;
11   }
12 }
13
14 host = clCreateBuffer(openCLData->context, CL_MEM_READ_WRITE |↩
       CL_MEM_USE_HOST_PTR, sizeof(memory), memory, &status);
15
16 if (status != CL_SUCCESS)
17 {
18   printErrorMessage(status);
19   return host;
20 }
21
22 status = clGetMemObjectInfo(host, CL_MEM_HOST_PTR, sizeof(↩
      hostPointer), &hostPointer, NULL);
23 if (status != CL_SUCCESS)
24 {
25   printErrorMessage(status);
26   return NULL;
27 }
28
29 return hostPointer;
30 }
31
32 //TODO: It is wrong the parameter. Fix IT!
33 void gpu_free_pinned_mem (cl_mem memory)
34 {
35 fprintf(stdout, ">>>>> openCLOperation.c: GPU FREE PINNED ↩
      MEMORY.\n");
36 cl_int status = clReleaseMemObject(memory);
```

The interesting thing about adopting the approach described above is the low impact in the other functions.

5.1.4 Final output

Originally, KGPU has some interesting services for cryptography and other optimizations characteristics. The problem with those services is the complexity of each one, some of them has more than 1500 lines of code. In this sense, it was selected a set of simple service whose the objective is only to call the KGPU function and test the behaviour.

Figure 29 shows the first time whose KGPU with OpenCL executed correctly. This output, means that kgpu.ko was loaded correctly and is already prepared to execute. The second line of the output, means that helper was running and waiting for a service. Notice that two equal address are displayed, this means that memory was allocated in the device returned correctly. After, it displays the memory address mapped from GPU and CPU. The last lines, shows the service be loaded by KGPU, executing a simple command and finish. Finally, KGPU stopped and wait for new services.

```
rodrigo@kgpu:~/Documents/Code/kgpu_fga/build$ sudo ./runkgpu
[helper] helper.c::99 kh_init(): KH_INIT
________ 1 _______________
Before return: 0x6052e0
________ 2 _______________
pinnedMemory: 0x6052e0
[helper] helper.c::108 kh_init() DEBUG: -->0x6052e0
_________ 3 _______________
_________ 4 _______________
[helper] helper.c::129 kh_init(): mmap start 0x7F6438BFB000
***** service.c: LOAD ALL SERVICES.
***** service.c: LOAD SERVICE.
[libsrv_test] Info: init test service
***** service.c: Register service.
[helper] helper.c::426 kh_main_loop(): Main loop.
```

Figure 29 – First output

6 Conclusions

6.1 Questions and answers

All the questions raised in Section 4.2 will be analysed and discussed in this section.

6.1.1 Is it possible to use OpenCL instead of CUDA in KGPU?

It is possible to completely remove all CUDA code, and replace it by OpenCL in KGPU. This project replaced all functions related with CUDA, and created the bases for working use OpenCL inside KGPU. All the translation was possible, because of the similarity between both APIs, and the previous refectory.

It is interesting to notice, that this work create the possibility of compare KGPU using CUDA with KGPU using OpenCL. This comparison is better than the one used by the author, because in the original work KGPU performance was compared with CPU and this not provide good conclusions about performance of KGPU. The current work already translate many parts of the CUDA code, however it is required to improve the translation for take more advantages of OpenCL characteristics.

6.1.2 Is it possible to keep CUDA and OpenCL in KGPU?

Yes, it is. This project, not only translated CUDA function to OpenCL but added both libraries to KGPU. If someone wants to use CUDA or OpenCL, they just need to use the correct Makefile.

6.1.3 Can we adopt KGPU architecture?

No, it isn't. KGPU architecture is really complex to understand and use, it is more easy to use CUDA or OpenCL directly instead of use this library. For example, if some user wants to create a new service to make use of KGPU, he will need to code some kind of device driver.

6.2 Future work

This project is only the first contact with GPU. The idea of KGPU was great in the sense of use GPU as a coprocessor, however its architecture is really hard to implement and it is not mature enough to be used. It is necessary to rethink the architecture and the way to use it. I propose three new steps for this work: change the KGPU architecture

for make easy for add new service to KGPU, improve the OpenCL code, and embedded the KGPU.

Next, improve the drive by adding GPU scheduler with prioritization. The scheduler is aimed to give some real time behavior to the drive. The idea that supports *Timegraph* will be put side by side with the GPU driver built in the first step. Finally, the idea of resource reservation will be attached to the drive. In conclusion, this step wants to create the base for future improvements in the drive by adding real-time scheduler.

The last step will be introduce the scheduling policies based on spatial scheduling ideas. After all of those ideas be put together in embedded context it is expected to gain a better performance in some tasks and a reduction of power consumption. For example, this device can be used to improve normal OS scheduling, making AES more faster, improve the character matches, and so forth.

References

ADRIAENS, J. et al. The case for gpgpu spatial multitasking. p. 12, 2011. Citado 2 vezes nas páginas 21 and 44.

ARROYO, G. E. M. Cu2cl: A cuda-to-opencl translator for multi- and many-core architectures. 2011. Citado 3 vezes nas páginas 35, 37, and 57.

BENEDICT, R. et al. *Heterogeneous Computing with OpenCL*. [S.l.: s.n.], 2013. Citado 2 vezes nas páginas 37 and 38.

GROUP, A. *ARM Mali*. 2014. <http://www.arm.com/products/multimedia/mali-cost-efficient-graphics/index.php>. Accessed: 2014-11-17. Citado na página 22.

GROUP, K. *Khronos OpenCL*. 2014. <https://www.khronos.org/>. Accessed: 2014-11-17. Citado na página 21.

HAN, .; JANG, K.; MOON, S. Packetshader: Gpu-accelerated software router. 2010. Citado na página 21.

HARRISON, O.; WALDRON, J. Practical symmetric key cryptography on modern graphics hardware. 2008. Citado na página 21.

KATO, P. et al. Timegraph: Gpu scheduling for real-time multi-tasking environments. p. 17, 2011. Citado 4 vezes nas páginas 13, 33, 42, and 43.

KIRK, D. *Programming Massively Parallel Processor - A hands-on approach*. [S.l.: s.n.], 2014. 250 p. Citado 5 vezes nas páginas 24, 26, 32, 33, and 35.

MARCONI, M.; LAKATOS, E. *Fundamentos de metodologia científica*. [S.l.: s.n.], 2003. Citado na página 47.

MCCLANAHAN, C. History and evolution of gpu architecture. p. 5, 2014. Citado 3 vezes nas páginas 23, 24, and 25.

OWENS, J.; LUEBKE, D.; GOVINDARAJU, N. A survey of gereal purpose computation on graphics hardware. 2007. Citado na página 21.

PACHECO. *An introduction to parallel programming*. [S.l.: s.n.], 2014. 370 p. Citado 2 vezes nas páginas 26 and 29.

PATTERSON, D. Computer organization and design. p. 500, 2014. Citado 9 vezes nas páginas 13, 23, 27, 28, 29, 30, 31, 32, and 33.

SCANNIELLO, G. et al. Greening and existing software system using the gpu. 2013. Citado na página 21.

SILVA, E.; MENEZES, E. *Metodologia da pesquisa e elaboração de dissertação*. [S.l.: s.n.], 2005. Citado 3 vezes nas páginas 15, 47, and 48.

SMITH, M. *What Is An APU? Technology Explained*. 2011. <http://www.makeuseof.com/tag/apu-technology-explained/>. Citado na página 34.

SUN, R. R. W. Augmenting operating system with the gpu. 2011. Citado 2 vezes nas páginas 41 and 54.

TOCCI, R.; WIDMER, N.; MOSS, G. *Sistemas digitais princípios e aplicações*. [S.l.: s.n.], 2007. 660 p. Citado na página 30.