TAMPERE UNIVERSITY OF TECHNOLOGY

PAU VILA FERNÁNDEZ

PARALLEL LANGUAGE PROGRAMMING IN DIFFERENT PLATFORMS

Masters of Science Thesis

Examiner: Professor Jarmo Takala and
Vladimír Guzma
Examiner and topic approved in the
Computing and Electrical
Engineering Council, meeting on
05/05/2013

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
VILA FERNÁNDEZ, PAU : Parallel Language Programming In Different Platforms
Master of Science Thesis, 44 pages
June 2013
Major: Digital and Computer Electronics
Examiner: Prof. Jarmo Takala
Examiner: M.Sc. Vladimír Guzma
Keywords: CUDA, GPU, Halide, OpenACC, OpenCL, Parallel Programming

The need to speed-up computing has introduced the interest to explore parallelism in algorithms and parallel programming. Technology is evolving fast but computing power in sequential execution is not increasing as much as earlier but CPUs contain more and more parallel computing resources. However, parallel algorithms may not be able to exploit all the parallelism in computers. The key issue is that algorithms need to be divided in independent parts to be executed at the same time. By using suitable parallel processors, such a GPU, we can address this problem and explore possibilities for higher speed-ups in computation. High performance calls for efficient parallel architecture but also the tools used to convert high level program description to parallel machine instructions, like languages, compilers. are equally important.

This thesis remarks on the importance of using suitable languages to describes the algorithm to exploit the parallelism. This thesis discusses the following parallel programming languages: CUDA, OpenCL, OpenACC and Halide. The programming model and how they run on parallel processors. Each language has its different properties and we discuss portability, scalability, architectures and programming models. In the thesis these languages are compared and advantages and drawbacks are considered.

PREFACE

This Master of Science Thesis has been undertaken at Tampere University of Technology (TUT) at the Faculty of Computing and Electrical Engineering from December 2012 to May 2013.

I would like to thank Jarmo Takala assigning me this thesis and give me the opportunity to come to Tampere and enjoy this year. Also, I would like to thank Vladimír Guzma for having spent most of his time helping me.

CONTENTS

NOMENCLATURE AND ABBREVIATIONS

ALU: Arithmetic Logic Unit

AMD: Advanced Micro Devices

API: Application Programming Interface

CPU: Central Processing Unit

CUDA: Compute Unified Device Architecture

DRAM: Dynamic Random Access Memory

GPU: Graphic Processing Unit

OpenACC: Directives for accelerators

OpenCL: Open Computing Language

SIMD: Single Instruction Multiple Data

SRAM: Static Random Access Memory

1. INTRODUCTION

This thesis proposes an analysis of different parallel programming languages, CUDA, OpenCL, OpenACC and Halide. It takes into account functionalities and portability among other characteristics. Nowadays, the algorithms that can be executed in a parallel way are commonly used in many devices such as cell phones and cameras. But also in Internet, where web pages are becoming every day more visual. Blogs and social networks are examples to demonstrate the necessity of images processing. These technologies allow the user to upload big amount of data in internet, of which most are pictures or animation sequences. Hence, it could be interesting to manage these data using accelerators devices, such as GPUs.

At the same time, these technologies are increasing their quality and size. However, this means that the processing becomes harder, so we have to optimize algorithms looking for efficiency, readability, portability among others features. These algorithms must have a condition, they need to be divided in pieces of code and executed in parallel to make the algorithm faster.

We have many parallel algorithms that can be programmed for one specific device, such as GPU, in order to improve the speed-up. However, many of them are difficult to read, difficult to update and hardly portable. Hence, they hardly can be improved or modified in a future by other programmers. In this thesis we discuss different parallel programming models [1] and in a second time we have focused the attention on the main different characteristics among them. After that we have tried to find out which is the best one or which one would be the best depending on what is going to be done. At last, we have described what in the future could be done, since this technology is not fully explored yet and there is still a long way to go about parallel computation.

1.1 Motivation

Technology has become one of the most important parts of our lives. We are surrounded by electronic devices, like cell phones, tablets and computers. That influence our everyday life and from which we depend. The field of computer systems (processors) is daily increasing; hardware and software are evolving faster, both of them should evolve in the same level to take advantage of all the properties.

Hardware provides us a lot of resources to improve the speed-up of many algorithms. For example, the multiple GPUs that we can find in the market provide us a good way to run operations at the same time. This is very useful to take advantage of the resources, but it is also important to look for a way to teach the programmers to know how to use these accelerator devices. Also, they should know about the parallel programming languages such as CUDA, OpenCL, OpenACC, Halide among others. There are more languages that work over accelerator devices, but the most commons are the ones mentioned.

By using these technologies properly we would be able to increase the efficiency of many applications that are commonly used, like image processing [2], DSP [3], etc. A good knowledge of them would be a big step to compute faster algorithms whose need a high runtime. On the other hand it would be also interesting to look for readability, portability, etc.

1.2 Objective

The main goal is to understand how some specific parallel programming languages work. If the algorithms we can write are easy to understand, modify and if they are portable to different platforms. The parallel programming languages that we have studied in this thesis are four: CUDA and OpenCL that are the most used and widely known. OpenACC that even if is not as common as the previous it could represent a valid alternative. As last one, a fairly recent language: Halide, which could generate outputs for the other languages after the compilation.

We have described the advantages and disadvantages of the different parallel programming languages that have helped us to explore parallelism in different platforms. We have evaluated the different characteristics of each parallel programming language separately, we have described the differences between them and finally we have presented our conclusions showing the best languages depending of the goal we want to achieve.

Through some program examples we have demonstrated how useful can be these technologies and how we can improve the speed-up of many algorithms that can run over accelerator devices. Depending on the device, we have seen if the programming model can be used in different platforms and if it can be programmed easily.

1.3 Structure of Thesis

This thesis is organized in eight different chapters. The Chapter 2 gives a brief review about the history of computer devices and their evolution. The next four chapters will describe and define the main characteristics of the different parallel languages that we will treat, from how to take advantage of the hardware resources to the explanation of some implementation of the code. The order of the these chapters will be as follows: CUDA (Chapter 3), OpenCL (Chapter 4), OpenACC (Chapter 5), where we will talk about platform models, memory model, architecture, etc for each language; and Halide (Chapter 6). In this chapter we will describe more accurately how it works because the point of view is different than from the others. Chapter 7 will present the differences between the program models explained before. Finally, the Chapter 8 will describe the conclusions that we can draw from the previous chapters.

2. BACKGROUND

2.1 Brief description of parallel computing

Parallel computation [4] is a way to execute many instructions at the same time instead to do it sequentially. Using this characteristic properly, we can divide some programs that require a large time of computation in small parts to execute them at the same time. This technique can not be used in every application. This is because the parts of a program executed in a parallel way have to be independent from the others. This means that each part can not modify external data during the execution.

A good example to explain this clearly is done in a bank. We suppose that we have a bank where there are many customers waiting in a queue to be served. We have four offices (4 processors). It has not sense serve all the customers in one office. The best idea is distribute the different customers in the available offices, making easier and faster the process.

Another example is how to make a poll of 10000 persons for a concrete case. The best way to manage this is look for different interviewers. Each one of them should interview more than one person. We have divided the problem in small parts and we have obtained better time results instead to do it in a sequentially way. This means, interview all the persons in different times. Finally, the recount could be also parallel. We could organize groups to manage de different polls for then put the results in common.

During last years, parallel computation has been increasing and being more used in computation due to the advantages it provides. One of the main problems in sequentially programming is that hardware can not afford high frequencies and it generates high temperatures. The solution for this is the parallel computation. We will try not to overload the hardware. This technique can be also useful for cell phones or other mobile devices to save energy in their batteries. Parallel computation requires a control of the hardware and software. The hardware must provide multiple processors to execute different instructions or a same instruction with different data at the same time. On the other hand, software must provide different techniques to manage these processors and the execution of the instructions through synchronization. This fact will be described in the next sections.

Thus, parallel computation is taking an important roll in the evolution of computer science allowing us to explore other ways of efficiency. Although it is a popular topic, this techniques needs to be studied deeper to get better results. The most important issue is the implementation is not easy as a parallel program can be written in many different ways. So it is difficult to find out the best solution. In this way, taking a bad decision could be worse than execute the same program in a sequentially way. There are also different programming languages and many architectures what is also a big inconvenience.

A good point to start studying this techniques is identifying which problems we can already manage in a parallel way. Some application would be: images processing, mathematical models [5], artificial intelligence [6].

Image processing consist on the execution of different functions over pixels. These functions are not usually depending on others. For this reason, the algorithms in this field can be improved significantly. We will use this application to demonstrate through examples how useful can parallelism be.

Mathematical problems are used to treat with equations which have many variables and constants. The solutions can need a high time of computation which can be improved with parallelism. Physics and concretely quantum mechanics are some fields that can be also managed in a parallel way.

Artificial intelligence tries to mimic human behavior. Usually this field looks for decisions that have to be compared to figure out the best one. An example would be how to solve a Rubik's cube in an efficient way. There are many decisions that can be taken, so they can be checked in parallel and compared.

2.2 Hardware

We have a standard architecture for sequential computation. Thus, it is easy to program applications for this model. In parallel computation we find several different architectures which are daily changing looking for optimizations. In fact, we have not enough information about the first parallel machines and only a few companies has survived working in this field, like IBM. However, it seems clear that the tendency is try to make platforms to get the best relation cost and performance. If any platform provides the correct software and solve some of the main problems of parallel programming it will have future. The issues that have to be fixed to evolve would be: improve the memory access, use standard processors or reduce the cost to make it accessible to buyers.

Parallel computers can be classified in different levels: depending on the tools it provides, how many processors, multiprocessors, etc. We can find multicore computing, distributed computing and symmetric multiprocessing and specialized parallel computers. Processors with multicore computing has several execution units in a chip. This characteristic allows to execute multiple instructions per cycle.

Distributed computing joins many individual computers, creating a large system with massive computational power. As the work is divided into small parts that can be processed simultaneously, research time is reduced from years to months.

Symmetric multiprocessing allows to some processing units to share the memory access. Therefore, any processor can work in some task, regardless of the memory location. With a good operating system support, these systems can move easily tasks between the processors in an efficient way.

Specialized parallel computers are devices that focus in a determinate parallel problem. So, the domain is limited and the future of this kind of processors are poor.

2.3 Software

The tendency of software is evolve scientific applications where we are looking for reach high speeds. As it has been said, we could use parallel applications for fields like artificial intelligence, robotics or graphs among other examples. One of the main goals for this section is giving enough tools to practice and familiarize with parallel computing. Hence, it has to provide powerful compilers, languages, etc and enough informations, such tutorials, to learn about these software.

The main problems we have in parallel applications are: the software is not portable to other platforms. There is not any standard platform as the CPUs. Therefore, each application should be modified depending of the platform it would be run. There are many accelerator devices in the market that differs each other, this is the reason why the portability is hard.

Also, there are not enough tools to develop software easily. Some frameworks used to program on accelerator devices are continually changing. Therefore, a change can lead that a development of a project has to be changed. Hence, it has a big impact to the development of applications.

2.4 How GPU works

In this thesis GPU is an important concept. In this section we will explain more accurately the operation of a standard GPU to show an architecture.

A GPU has a fast memory. Another characteristic is that the structure of a GPU is highly divided. This means that we can find different functional units. This units are divided in two types: the functional units that processes vertexes and the others processing pixels. Vertex are programs that manage the vertex of 3D images, such the movement of a person. Pixel is used to decide the color, illumination, where a pixel will be drawn, etc.

With the basic concepts mentioned before, we can explain how a GPU works. First of all, the CPU (host) will send vertexes to the GPU. Then we define the pixels and we store them in a cache memory to draw them in a screen forming the desired image.

The architecture of a GPU is thought to do massive parallel operations. As in the common CPUs where we can find different kernels. But many more in GPUs. A GPU will manage a huge quantity of data. So it is clear that the bandwidth of data memory have to be big enough to keep all the data demanded by the process units. If it is not like this, we would lose performance and we would not take advantage of parallelism.

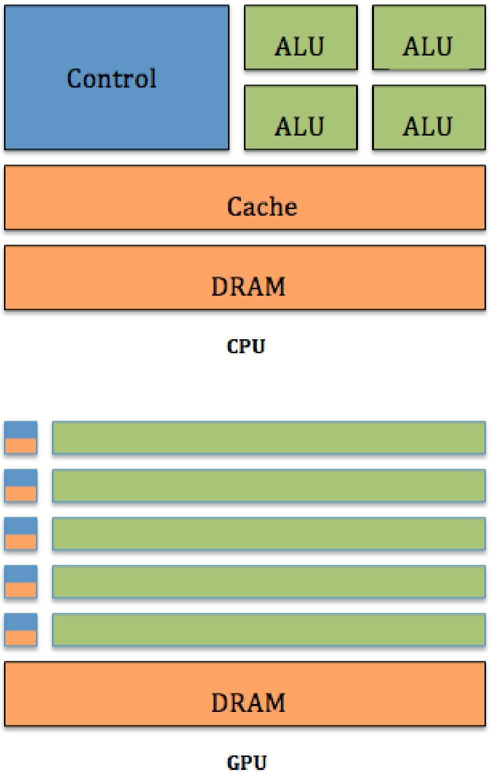


Figure 2.1: CPU vs GPU

In the Figure 2.1 we can observe the differences between the number of ALUs that CPU has and the number of ALUs that a GPU has. We have to mention that this computing units of a GPU are slower than the one of a CPU. This means that the way to parallelize a program is very important. If it is done in an incorrect way the execution in a CPU would be faster than in a GPU. But if we parallelize a program in a proper way we would obtain a good results, It means, parallel algorithms running on CPUs would be much faster on GPUs. We can also see that a GPU is formed by multiple multiprocessors which each of them has its own registers, shared memory (SRAM), cache of constants and textures, and the global memory (DRAM) that is faster than the one of a CPU.

3. CUDA

CUDA [7] [8] [9] was created by NVIDIA [10] as an extension of C. It was thought to explore parallelism and take advantage of GPUs in order to improve speed-up of parallel algorithms. Most of them common in image processing field or 3D.

In the consumer market, many applications, applications that need a parallel computation, are executed through CUDA. CUDA is being used by many researches that are working on dynamic molecular field, financial market, etc. These fields need a high level of computation. It is necessary to note that this language is not the only in the market.

The performance measurement of a CPU is defined by the number of operations we can compute in one second. The current processors can compute 50 GFLOPS (billion floating point operations per second) and GPUs can compute about 500 GFLOPS. As CUDA was created by NVIDIA [10] it is just available for NVIDIA GPUs. Concretely from G8x series until the GPUs of nowadays. Thus, the portability is very poor.

Basically we must be clear that CUDA is useful for parallel algorithms. Those that work with many threads and can compute at the same time. Otherwise it would be useless the usage of the GPU.

A simple example of a useful algorithm executed in a GPU would be the product of two matrices. The operations that we have to do are always the same but in different positions. These functions will be called kernels. Now we want to execute these kernels in parallel and each execution will be called thread. Therefore, in this individual case each thread is responsible to compute one element of the result matrix. So we will have as many threads as elements of the result matrix.

The graphic cards that supports CUDA are formed by several multiprocessors and each one is formed by many more processors. These processors are responsible to execute the threads.

3.1 CUDA architecture

CUDA architecture allows the programmer to use the graphic card like a processor of general purpose. The image below, Figure 3.1, shows a standard model of hardware architecture of a common graphic card of NVIDIA. As we can see is formed by different multiprocessors, each one of them with an architecture SIMD. Where we can find more processors (SP) inside them. There is no synchronization between multiprocessors. Each processor inside a multiprocessor would compute the same function with different data.

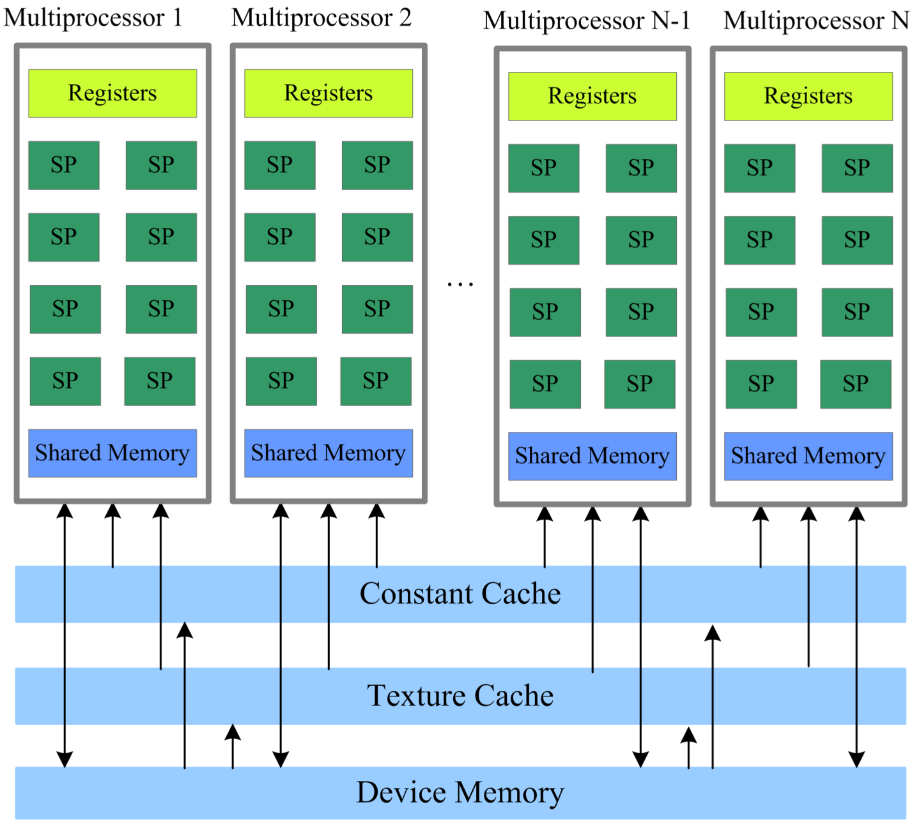


Figure 3.1: Hardware model. Figure from [10].

3.2 Programming model

A GPU (device) offers to CPU a co-processor highly divided in threads. This co-processor has own memory where each thread can be executed in parallel through the different kernels. Threads in CUDA are light. This means they are created in a very short time and the switching is instantaneous. The programmer of CUDA has to declare all the threads that a GPU needs to achieve the scalability and performance desired.

Threads can be organized in blocks cooperating among them, sharing information through fast memories and synchronizing. Each thread is identified with its thread identifier. This is the number of the thread within a block. It helps to define it in more than one dimension and make the identification easier.

In CUDA, the numbers of threads are limited in each block, nonetheless, CUDA threads can be grouped by blocks of threads. So, in the running we could execute more threads than the limit without synchronization.

3.3 Memory model

The next image below, Figure 3.2, shows a typical architecture of CUDA memory. There is a limitation of number of threads running inside a block. However, we could distribute threads to multiple blocks, this relation is called "grid". Each one of them executing the same kernel but taking into account that there would not be communication between blocks as said before.

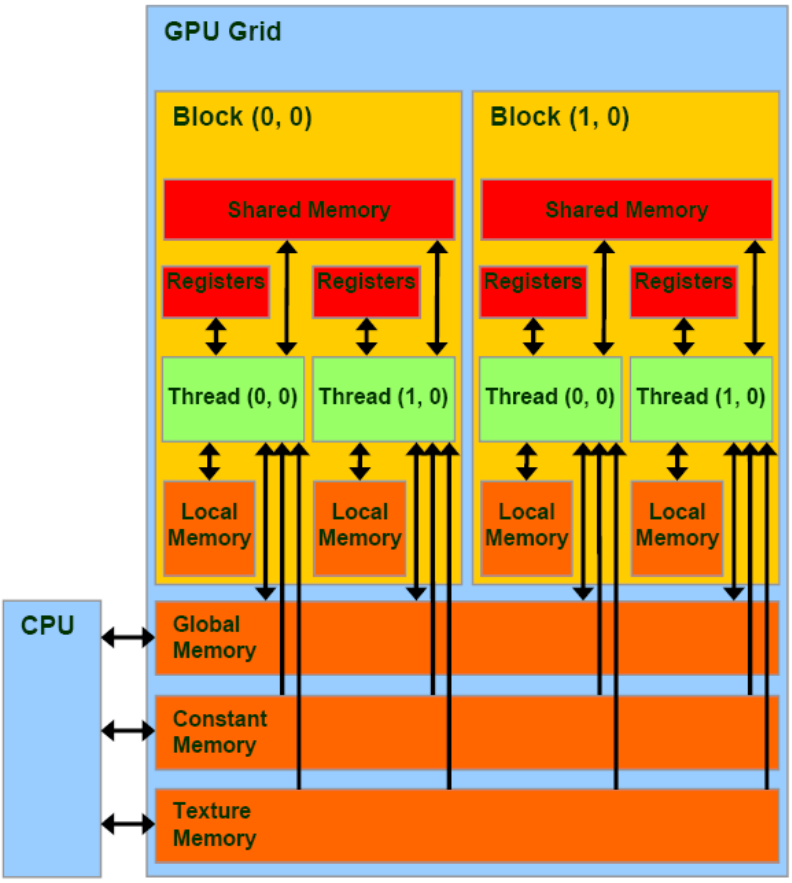


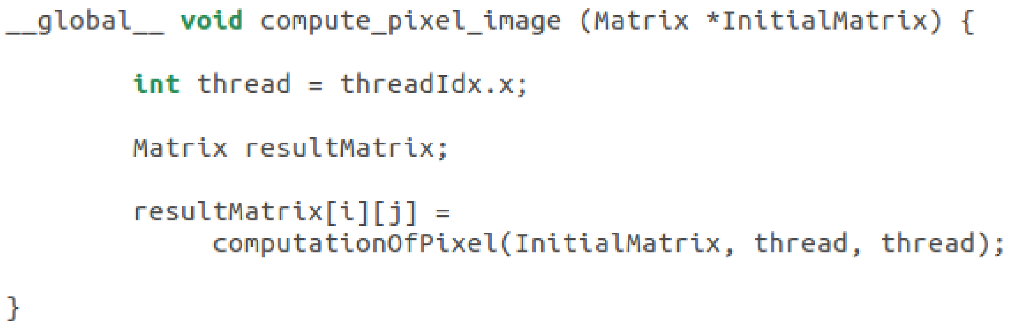
Figure 3.2: Memory model. Figure from [10]

About memory hierarchy we can differentiate four different types of memory: local memory, global memory, shared memory and texture memory. Local memory is the memory available just for each thread. Global memory can be read and written by threads of GPU and CPU. Shared memory is shared for each thread in a block. Therefore it exist communication between threads and we could synchronize points inserted in the kernel. Shared memory stands out for being very fast. Texture memory is generally read only and cached.

When a thread is executed in the device it can only access DRAM (global memory) shared memory and local memory. Taking into account that the access to the memory scopes are as follows: local memory of each thread, shared memory and global memory can be accessed in a read/write way. Texture memory can only be accessed in a read way. The application can write and read global and texture memory.

3.4 Programming in CUDA

As it is said before, CUDA is an extension of C. In this way, a programmer familiarized with this language would be able to manage a NVDIA graphic card through the NVIDA APIs. An example of generated kernel with CUDA code would be as follows below, Figure 3.3.



```
__global__ void compute_pixel_image (Matrix *InitialMatrix) {

        int thread = threadIdx.x;

        Matrix resultMatrix;

        resultMatrix[i][j] =
             computationOfPixel(InitialMatrix, thread, thread);

}
```

Figure 3.3: Code example

The program above declares a variable called "thread" to decide which threads we will use and then makes operations over each pixel on an image. The function "computationOfAPixel" makes a determination operation over a given pixel, it could be smooth the pixel. We will follow this example during the thesis. This function will be executed once per thread. This means that the execution time will be faster than the same program executed in a sequential way. CUDA allows us to choose which dimension we want to define, this can be one, two or three. In this example we would us two because we are working with matrices.

3.5 Conclusion

CUDA adds benefits to past, present and future generations of NVIDIA. Nowadays it is interesting to check this platform beacuse it gives us a wide repertoire of applications. However, its future is not as clear as the present. CUDA is just for NVIDIA platforms, the APIs cannot compete with open APIs, that are already in the market, like OpenCL.

The specialization in determinate platforms, the ones make by NVIDIA, makes CUDA a powerful language that take all the advantage of each part of the architecture. The customers are assured of the optimization of NVIDIA.

4. OPENCL

OpenCL [11] [12] [13] is an open standard for programmers of CPUs, GPUs or other devices that could be even cell phones. OpenCL allows us to decide which element of a device, for example a determinate processor in a platform, will manage a specific operation. This is done without changing the language used in the code. As CUDA, OpenCL was thought to improve the speed-up in parallel algorithms. Thus, it would not be very useful for algorithms that has to be executed in a sequential way.

One of the aims of OpenCL is look for efficiency, but also portability. So, if we have the proper drivers, OpenCL can be used in many different platforms.

We will divide this chapter in different parts. We will talk about the platform model, the execution model, the properties of the kernel, the properties of the host, the memory model, the programming model and, at the end, we will show an example that demonstrates that OpenCL can be really useful.

4.1 Platform model

Platform model defines a connected host to one or more devices that supports OpenCL. The host can be a CPU running over an operating system that supports OpenCL. Host executes an application that will manage commands and instructions. These operations will run through the OpenCL APIs in a determinate device. This runtime system will manage the device following the instructions given by the host. An OpenCL device is a collection of one or more computing units, which are, subdivided into processing elements. Figure 4.1 shows a platform model.

4.2 Execution model

Execution model is formed by kernels and the application that is controlling the different devices. As CUDA, kernels are functions of code, that can be executed in parallel for explore parallelism. These kernels have to be executed in a device, which will compile it in execution time. Finally, the application controls the devices connected to the host and it also controls the flow of data between the host and devices.

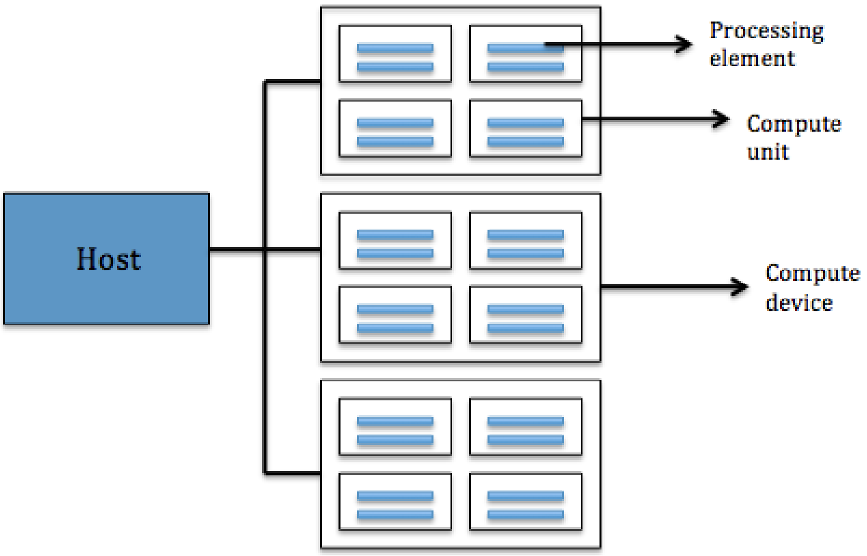


Figure 4.1: Platform model

4.3 Properties of kernel

The application put the kernels in the queue before being executed. But before this we have to define a dimension. OpenCL allows us to choose which dimension we want to define. There are three dimensions. If we were working with vectors the dimension could be one. But, if we were working with matrices the dimension should be two or more. An example of this can be found in the field of image processing, where we have to compute pixels in a matrix. Figure 4.2 shows us a dimension = 2.

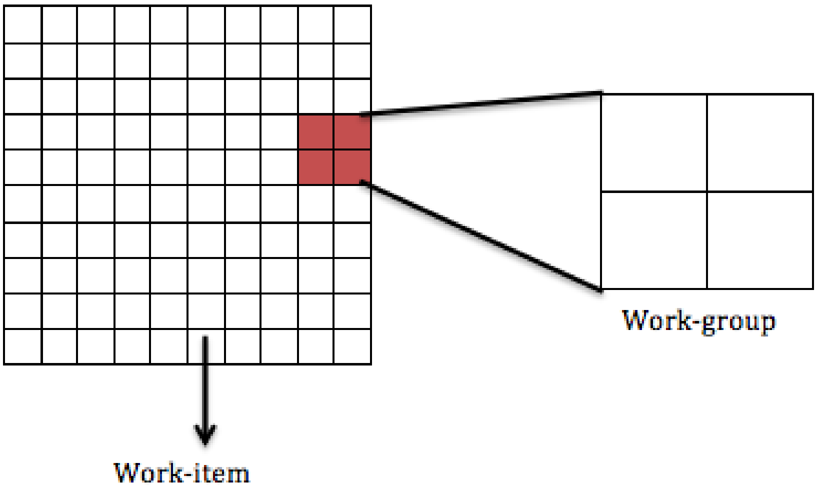


Figure 4.2: Dimension of an image

Each element of the dimension, called work-item, will execute the same code. In this way, the elements will have their own ID. OpenCL also allows us to create groups of elements, called work-groups. This groups will be useful to decide which elements of the dimension will share memory. This is important if some elements needs to be synchronized with another element. OpenCL provides us the necessary tools to allow communication between them. If we do not define any group, the APIs of OpenCL will do that for us.

4.4 Properties of the host

The host manages the queues and also the flow of the memory between devices. It will decide which kernels are going to be executed in a specific device and also it will manage the synchronization between kernels. All of this will be done through the APIs of OpenCL.

4.5 Memory model

The host need to send and receive information. This is done through the different memories that its model provide us. A common model of memory has a host memory that is the memory available for the host application responsible to control the different devices. Following the example of computing image's pixels we will describe the other memories. It is important to remind that each pixel was an element and elements could form groups to share information. According to this example we can describe global memory and constant memory (Figure 4.3).

Global memory allows read/write for all the elements of all the groups (elements/groups mentioned in section 4.3). To read and to write in global memory can be stored in cache depending on the device. Constant memory is a part of a global memory that holds constant during the execution of one kernel. Host will manage the data in this memory.

This model also has a deeper memory. Only shared for the groups of elements or just for the elements. These memories are called local memory and private memory (Figure 4.3).

Local memory is the memory allocated in each group. This memory can be used to assign variables that will be shared for all the elements of the group. Finally, the private memory is just available for each element. One element cannot access to others private memories.

We have described the memory in the order of access. The order is:

$HostMemory \rightarrow GlobalAndConstantMemory \rightarrow LocalMemory \rightarrow Work-item \rightarrow PrivateMemory$

Host cannot access to local memory or private memory.

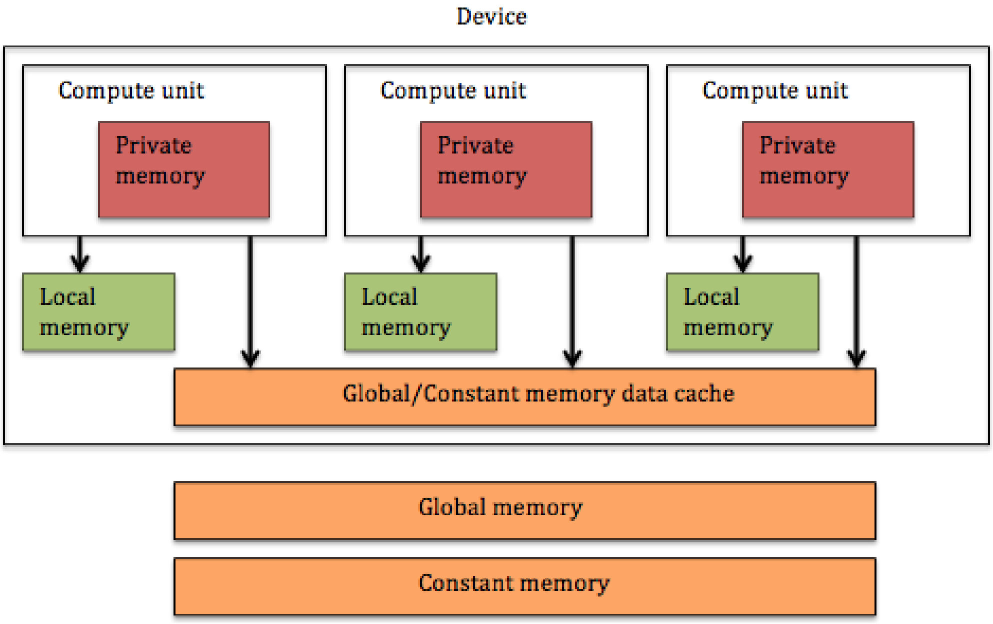


Figure 4.3: Memory model OpenCL

The main problem in this memory model is the bottleneck between host and devices. A device, like a GPU, could be much faster than a host processing data.

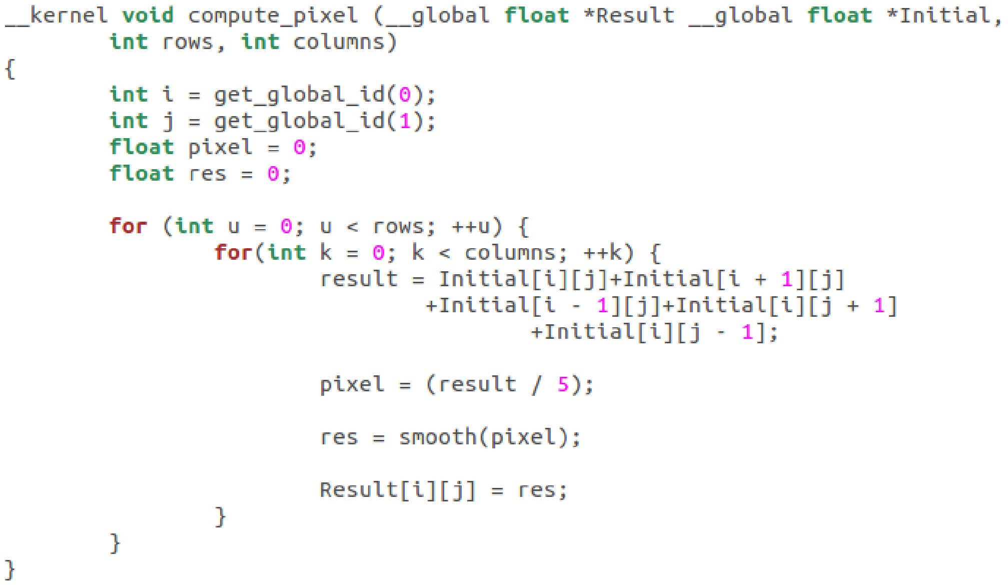
4.6 Programming model

OpenCL as a parallel model of programming provides us enough tools to explore parallelism. As we said before OpenCL allow us to use synchronization. Synchronization can be used in devices (GPUs) or in a host. We can not assure the execution of the kernels in a specific order. Therefore, we can put control points inside devices to control the execution of the different kernels. Synchronization in a host can be very useful. It allows us to do a proper communication between host and devices. In this way, while host is waiting for any information from the devices, it can work in others processes.

4.7 Example of application

We will put into practice the example of the image formed by different pixels. Imagine that we have to smooth an image from an initial picture. Each pixel has to compute an average of the surrounding pixels of the initial image and make specific operations to achieve the desired result. As we can see the result of one pixel (element) does not depend on the others.

As said before, we will define a dimension of two where each element will be a pixel. This means that we will design a kernel for each pixel. Then, each kernel will have an identifier formed by two numbers that will be (i, j), got from the functions of OpenCL called get_global_id(). We have as many kernels as the value of the dimension. The image below, Figure 4.4, shows how to generate a kernel over an element.



```
__kernel void compute_pixel (__global float *Result __global float *Initial,
    int rows, int columns)
{
    int i = get_global_id(0);
    int j = get_global_id(1);
    float pixel = 0;
    float res = 0;

    for (int u = 0; u < rows; ++u) {
        for(int k = 0; k < columns; ++k) {
            result = Initial[i][j]+Initial[i + 1][j]
                +Initial[i - 1][j]+Initial[i][j + 1]
                    +Initial[i][j - 1];

            pixel = (result / 5);

            res = smooth(pixel);

            Result[i][j] = res;
        }
    }
}
```

Figure 4.4: Algorithm OpenCL

This kernel (function) receive an initial image (matrix) and a result matrix. Each pixel of the result is the average between the surrounding pixels and the application of the function "smooth" that make the specific changes to achieve the desired result. This algorithm will be executed as many times as elements in the matrix, and we can execute it at the same time in a GPU. This feature give us the ability to improve greatly the speed-up of this algorithm. In a sequential way it would be much slower. We also could apply this parallelism in other fields, like video, DSP, etc.

4.8 Conclusion

OpenCL is an emergent technology that can be supported for different platforms. One of the main aims is improve the portability and the performance. OpenCL is thought for parallel programs that can compute multiples operations at the same time. A sequential program programmed in OpenCL would be useless. In a future, OpenCL probably will be present in lots of platforms thanks to its flexibility and its scalability. Besides, it can be used to improve image processing, video, digital signal processing, etc.

5. OPENACC

OpenACC [14] could be an alternative language for CUDA and OpenCL. It has been developed by CAPS, Cray and PGI. OpenACC is looking for the improvment the speed-up in parallel algorithms using other devices, like GPUs, through one host that could be a CPU. OpenACC APIs are based in C, C++ and Fortran what makes greatly portable for different platforms.

The main goal of this technology is to forbidden the programmer to manage the GPU or the transfer of data between the host and the device. The compiler has to be capable to generate an efficient way to use the device. The idea was to make the programming easier for those programmers who are not experts in OpenCL or CUDA.

This chapter is formed by three sections. In the first one we will explain the execution model. Then, the memory model and, at the end, we will see an example about image processing to clarify the improvement of speed-up of algorithms with this technology.

5.1 Execution model

Execution model consist of a host related with a device, such a GPU. The host will manage one or more devices through an application made by an user. The device will be responsible to explore the parallelism. It will compute functions than can be executed at the same time without affecting the final result of an algorithm. As it has been said in previous chapters, this functions will be called kernels. The host will decide which kernel has to be in a specific queue, which data that has to be sent to a device, when the host has to receive the data from the device, etc.

OpenACC has two ways of work: using kernels and parallelism. The architecture is formed by different processors. Each processor is provided with many threads. Knowing this we could distinguish three levels using parallelism: gang, worker and vector. The following Figure 5.1 shows the different levels.

Gangs would be mapped in the processors where they would not communicate between them because there is no possible synchronization between gangs. Worker would be related with threads. Finally, at the deepest level, vector, as the word says, would be a vector within a thread. The last levels could communicate among

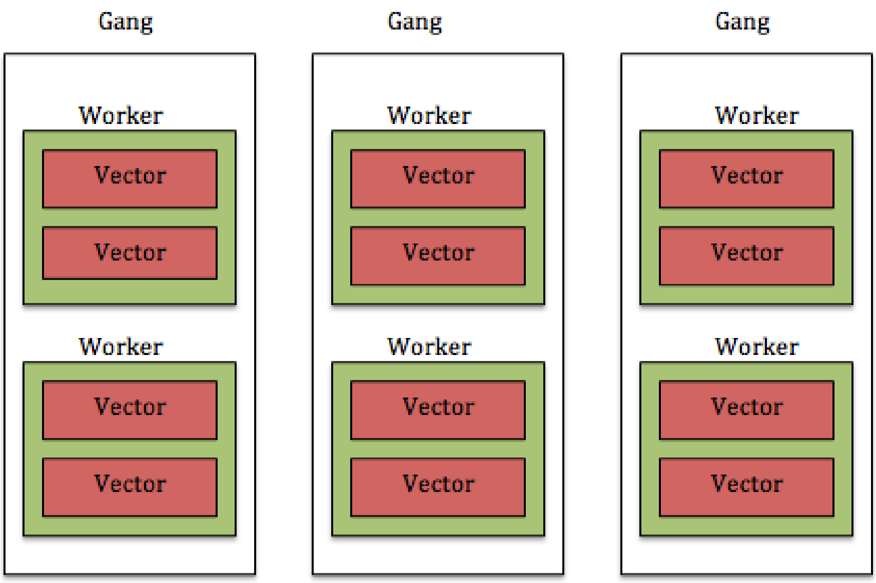


Figure 5.1: Levels OpenACC

them through the common memories that we will explain in next section.

How to allocate kernels depends directly of the device we are using. In this way, the compiler will be able to manage the kernels properly for a specific device. First of all, the compiler will check the code and will select the operations than can be executed in parallel. Then, it allocates them in a proper part of the hardware. Finally, the compiler manages the generation and optimisation of the code.

5.2 Memory model

As current GPUs, the memory of the host and the memory of a device are separated. All the transfer of data has to be managed by the compiler through some directives given by the programmer.

The programmer has to take care about the architecture of the devices in order to do not overload the memory and not to cause a bottleneck.

5.3 Programming model and examples

OpenACC is thought to reusing code, in other words, to be portable. The main goal is reuse code that have been already created adding some directives. This means that we would do just modest changes in the code. This language is supported by C, C++ and Fortran compilers. What we will do is just describe how to compile the parts of the code that can be split to be paralleled. Then the compiler will generate the gangs, workers and threads following the instructions given by the programmer. Next Figure 5.2, shows the responsibility of generate kernels by the compiler through the directives.

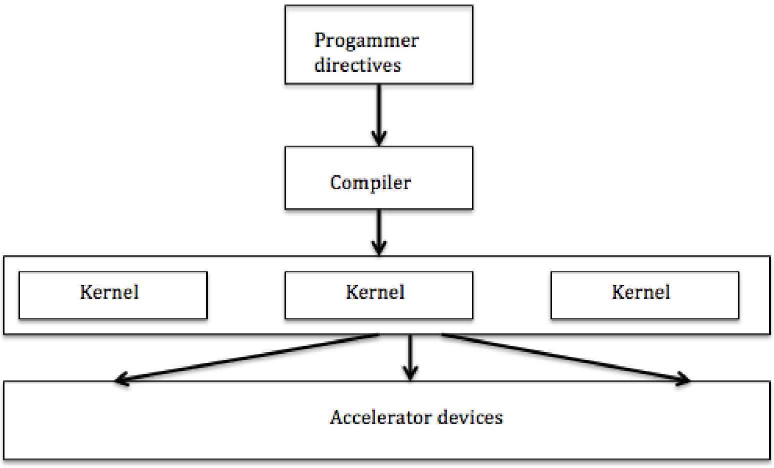


Figure 5.2: Architecture OpenACC

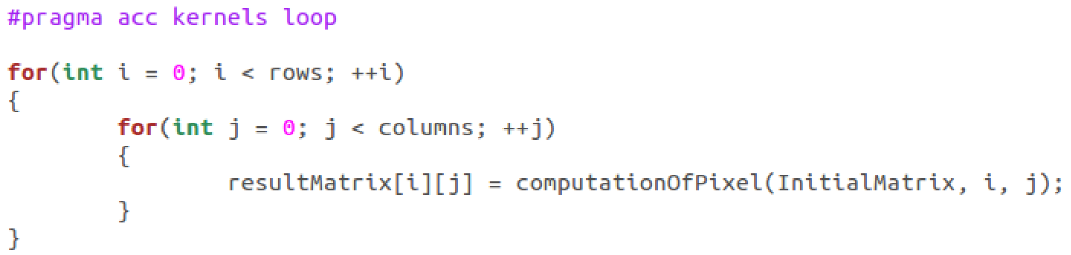
We will show some examples that will lead us to identify the differences between kernels and parallelization. We will take the example that we have been following until now, the processing of one image. In this case, the computation of each pixel of an image to achieve a specific result. Result pixels does not depends on the other.

The code below, Figure 5.3, shows us how to make notice the compiler if we want to use a parallel or a kernel construction.

In the case of using a kernel construct we have to add the line "program acc kernel loop" at the beginning of the program. The compiler will split the program in kernels, the construct says to the compiler to generate a kernel from the single loop. The compiler will convert these kernels to parallel kernels that will run in a GPU. Each kernel will be launched separately. In this example each computing of a pixel will be a different kernel.

If we want to use a parallel construct, we must change the first sentence of the program for "program acc parallel loop". In this case, the compiler will create a group of parallel threads that will execute the code of the construct. This construct generates a group of parallel gangs that will execute the code. The main difference in front of kernel construction is that parallel construction is a single CUDA kernel.

We can add clauses after each construction. This will help the compiler to identify how the different parts of the code should be accelerated. In the case of a kernel and parallel construct we could find some clauses like the next ones:



```
#pragma acc kernels loop

for(int i = 0; i < rows; ++i)
{
        for(int j = 0; j < columns; ++j)
        {
                resultMatrix[i][j] = computationOfPixel(InitialMatrix, i, j);
        }
}
```

Figure 5.3: Defining a kernel

- num-gangs clause is just available for parallel constructs, this clause allows us to define the number of gangs.

- if clause. This clause evaluates a condition, if this condition is true, the construct, kernel or parallel, will be executed on the accelerator, if not, it will be executed a thread will execute the construct.

We can find more clauses in the official document of OpenACC [15], and also the constructs to use in the case of programming in Fortran.

OpenACC provides us a run time library with many routines. The use of this library could be a disadvantage for the platforms that cannot support this API. In this way, we could lose some portability. Some examples of routines are:

- acc-get-num-devices: This routine let us know the number of accelerator devices related with the host.

- acc-set-device-type: It shows to the programmer which devices he/she could use.

- acc-get-device-type: It allows to the programmer to select the accelerator devices that he/she wants to use.

We can find more routines in the official documentation of openACC [15].

5.4 Conclusion

OpenACC is an alternative language thought for people that are not familiarized on languages like CUDA or OpenCL, which require a high knowledge. This system makes easier for the programmers to work on this kind of devices. In order to accelerate algorithms which can be executed in a parallel way. The programmer will not need to focus on the memory model. The person just will have to know some basic concepts, such the bandwidth to make a proper connection between host and devices. The compiler will manage the allocation of the data provided by the application from the host. In this way, the programmer will be able to create kernels and define the parts of the algorithm which can be executed in parallel using the different three levels mentioned: gang, worker and vector. Thus, OpenACC is simple and portable where the compiler compile code programmed in C/C++/Fortran to make it run in different devices.

6. HALIDE

Halide [16] is an incoming programming language implemented by researchers of MIT (Computer Science and Artificial Intelligence Laboratory (CSAIL)). Concretely is an extension of C++. MIT released the Halide's libraries.
This language programming emphasizes two aspects:

- Simplifies the process of writing image processing software. It makes the code more readable and also easier to modify.
- As other parallel languages the main goal is economize the use of energy through running programs in a more efficient way.

The process of taking a picture with a cell phone device is simple: press a button and in less than a second it will appear on the screen. However, the image processing requires a very complex sequence of operations. These processes are related with the capture of light read to deduce each pixel color and contrast settings. Besides, a constant correction process that change the image to get something closer to what captures the human eye. As more mega pixels a camera has, more time needs the device to process them.

By rewriting the algorithms in this language, researchers have been able to increase the speed of processing an image by three times and significantly reducing the length of the code.

There is not doubt this idea can benefit not only those who work with images. Also it is an interesting approach to parallelism which could be extended to other areas. But we would like to emphasis, in this thesis, the image-processing field because is its principal use.

What Halide suggest is separate the algorithm from the schedule through pipelines. In this way, we will be able to modify the algorithm without changing the schedule or, in the case contrary, change the schedule without modify the algorithm.

So Halide has two different parts, algorithm that specify what the program is doing, which mathematical functions we will use (what is computed), and schedule that define where and when we will execute or allocate what we have done in the algorithm (how is mapped in a machine). It means that programmer will have to care about the schedule, not letting to the compiler do all the schedule work.

For each machine the processing schedule could be different. Thus, we could say schedule is never portable. However, the algorithm is completely portable. Almost in all the cases a code written in Halide the algorithm is the biggest part of the code. If this is the 90% of the code, then the 90% of the code is portable. Besides, from Halide's web page we can download all the resources necessaries to use it. With many examples of Halide applications and demonstrate that the algorithm is the most predominant part in the algorithm. So, what we should do is just change the schedule and adapt it for a determinate machine what is easy work for an architecture expert that is looking for efficiency.

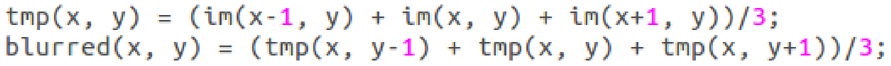
In the next section there is explained step by step what algorithm and schedule means more accurately.

6.1 Algorithm

The part of the algorithm will be defined as a functional language. This means that we will emphasise in the application functions instead of the imperative program style. In this way is easier to manage the algorithm by the schedule, but it will be explained in the next section. The characteristics would be:

- Arithmetic and logical operations.
- The control flow operations if, then and else.
- The capacity to call other function, even external.
- Focussing in image-processing, we will load external files, in this case images.

A code example of and algorithm would be like the image below, Figure 6.1:



```
tmp(x, y) = (im(x-1, y) + im(x, y) + im(x+1, y))/3;
blurred(x, y) = (tmp(x, y-1) + tmp(x, y) + tmp(x, y+1))/3;
```

Figure 6.1: Example of algorithm. Extracted from [16].

What has been done in the image above is processing the pixels of an image and making the average of each one. We can deduce that we could calculate different pixels at the same time depending on the processor or GPU we are working on. We must notice that in this example the second function definition depends on the first one. And this will be important to know how to manage the schedule.

6.2 Schedule

As it has been explained in the last section, the algorithm is a chain of functions which defines the algorithm. Thus, we have to take care in the schedule what is stored, what is recomputed and the order of the execution. The programmer will handle the schedule. Hence, explode the sources through GPU, SIMD, etc. Also it should be noticed that we will not need to get a very complicated compiler.

Following the last example of algorithm, we will explain the different ways to manage the schedule. Through the functions defined in the algorithm of example we could realize that there is a dependence. The second function declared need the results from the first. So, we can define the first as callee and the second as caller. Halide allows four types of realations between caller and calle to manage the schedule:

- Inline: Compute as needed.
- Root: Recompute all the region.
- Chunk: Compute only subregions.
- Reuse: Load regions computed.

6.2.1 Inline

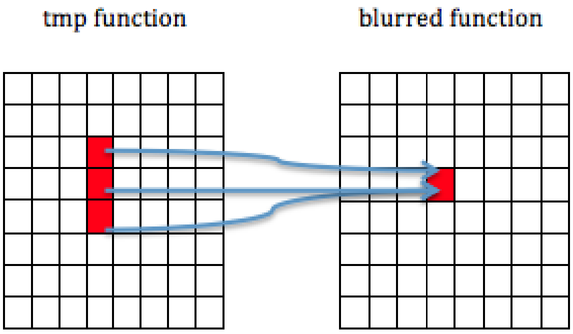


Figure 6.2: Inline

The image above, Figure 6.2, defines the functions of the Figure 6.1 as two squares. In each square is defined an image that we will call region. In this way, the first region is the callee and the second one is the caller.

Inline is the easiest way to schedule. When the caller needs to calculate a pixel it will call the callee to compute the values needed. So, for each pixel of the caller we will compute three of the callee. This means that we will generate redundant computation. Usually, it does not require too much storage and we are winning in temporal locality. The problem could be the redundant computation.

6.2.2 Root

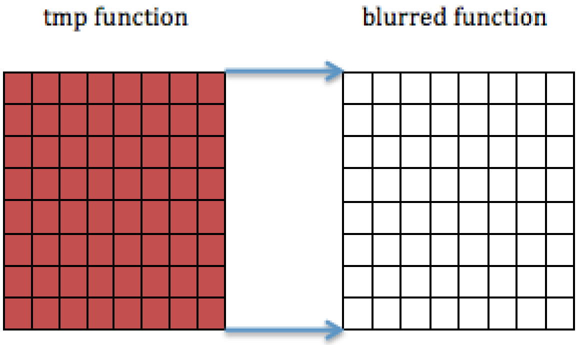


Figure 6.3: Root

In this case, Figure 6.3, what have been done is computing all the region before executing from the second function (blurred). We are storing all the region in order to avoid redundant information. This time each pixel of the callee is computed only one time.

On the other hand, Inline requires to store the entire region. Thus, the temporal locality would be poor, but we would avoid redundant computation.

6.2.3 Chunk

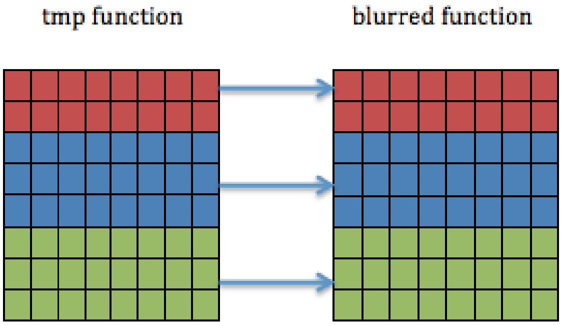


Figure 6.4: Chunk

This type of relation, Figure 6.4, would be like an intermediate type between Inline and Root. As we can see in the region below, the region of the callee is split in sub-regions. In each sub-region will be stored if it is needed. Contrary Root, we are reducing storage, but we could find some redundant information between the limits of sub-regions.

6.2.4 Reuse

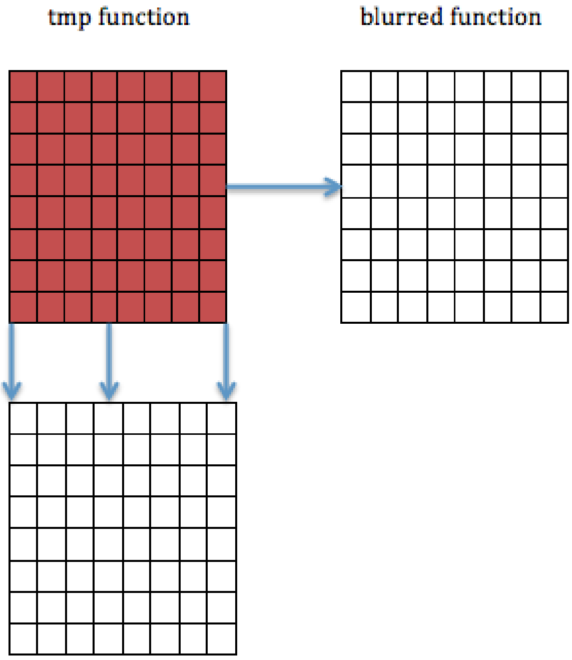


Figure 6.5: Reuse

Reuse can be used if previously we have chunked or rooted some function that could be used in the future for other functions. In the Figure 6.5 we can see another region that means a new function depending on the first. What we are doing using Reuse is avoiding recompute again values computed before. We should use reuse always we can.

We could apply, once explained the four types of relations, the functions as function.root(), function.chunk(), etc.

6.2.5 Modelling dimensions

What remains to be explained yet is how to manage each region or sub-region. Halide provides us some functions to define the order of the regions, how to vectorize them among others.

Following the same example where each region is supposed to be an image. We have to decide how to manage them. Here we have some examples that define how to use some of the functions provided by Halide.

In the example of the Figure 6.6 we are using the function transpose, used as image.transpose(x, y). We have defined the direction of how we will read the image. In this case, we have first moved through the rows and then through the columns. We may get the opposite effect changing the values 'x' and 'y'; image.transpose(y, x).

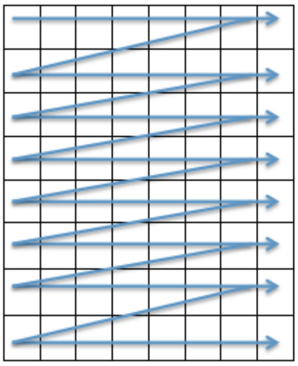


Figure 6.6: Transpose

Another example would be the one at the Figure 6.7. Here we are reading the matrix through the rows and then through the columns, but now we have vectorized the rows in parts of four pixels. The function that we may use would be image.transpose(x,y).vectorize(x,4).

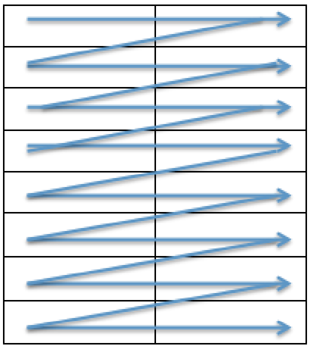


Figure 6.7: Transpose and vectorize

Another important function among others, would be "parallel". The following

Figure 6.8 shows us the result in a matrix which combines the function image parallel(y).vectorize(x, 4). What is done here is vectorize the rows in parts of some pixels and parallelize the two columns resulting of the function vectorize.

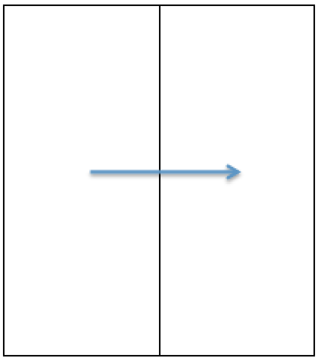


Figure 6.8: Parallelize and vectorize

A complete example, Figure 6.9, combine functions which define the regions and then they models them. It would be; image.root().vectorize(x,4).transpose(x,y).parallel(x). In this case we are defining the image as a root. This means that we will store the whole region. Then we will vectorize the rows in a group of pixels and we will define the direction with the function "transpose". Finally, it would parallelize the rows.

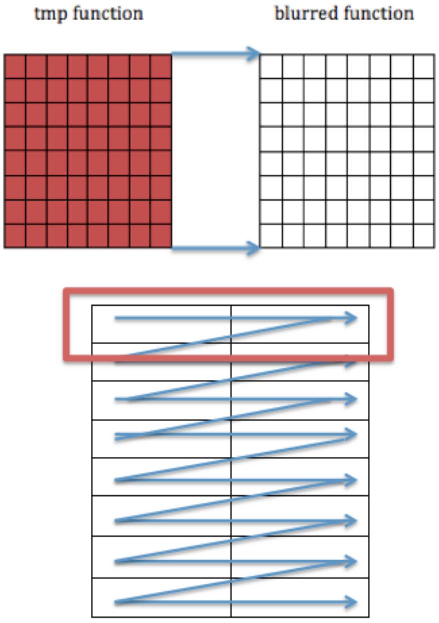
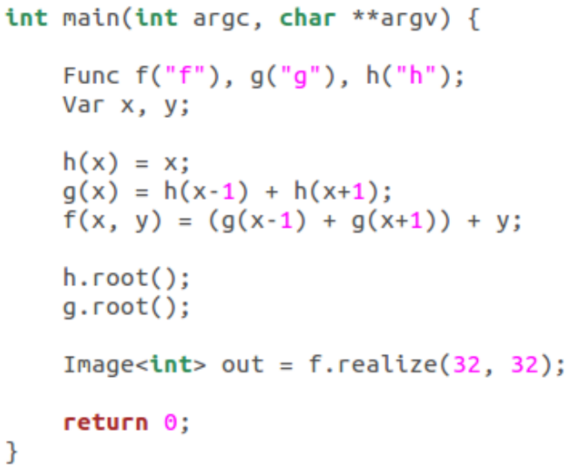


Figure 6.9: Complete example

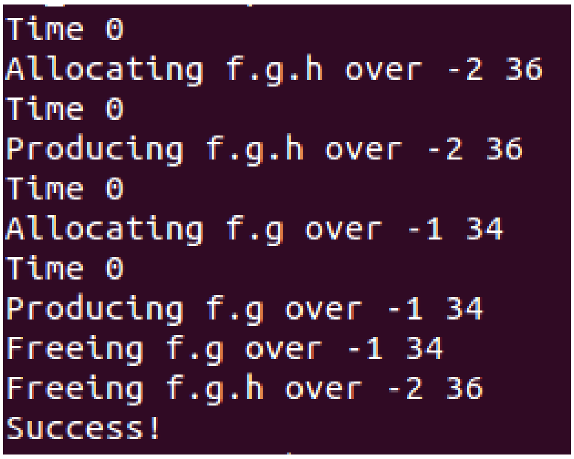
In order to explain better the functionalities of Halide we have a short program, Figure 6.10. This code makes the same operation on each pixel. It defines the schedule and finally define a domain 32x32. We can see easily the two parts: the algorithm and the schedule. The algorithm is formed by three functions declared. We can see how "g" depends on "h" and "f" depends on "h". The schedule will be defined as root for "h" and "g".



```
int main(int argc, char **argv) {

    Func f("f"), g("g"), h("h");
    Var x, y;

    h(x) = x;
    g(x) = h(x-1) + h(x+1);
    f(x, y) = (g(x-1) + g(x+1)) + y;

    h.root();
    g.root();

    Image<int> out = f.realize(32, 32);

    return 0;
}
```

Figure 6.10: Code example

If we execute this code in our computer we will see how Halide works. First we will compute all "f.g.h". Then "f.g" as a root. Finally we will compute the last function which is "f". The following Figure 6.11 is an output of this program.



```
Time 0
Allocating f.g.h over -2 36
Time 0
Producing f.g.h over -2 36
Time 0
Allocating f.g over -1 34
Time 0
Producing f.g over -1 34
Freeing f.g over -1 34
Freeing f.g.h over -2 36
Success!
```

Figure 6.11: Output

6.3 Compilation

The compiler used to compile Halide will be LLVM. As we have been saying before, Halide programs are composed by the functional algorithm and schedule. In this way, the compiler has to transform the algorithm as an imperative language. Then, the compiler manages to create an architecture-specific LLVM bitcode in order to run it in a machine.

6.4 Applications

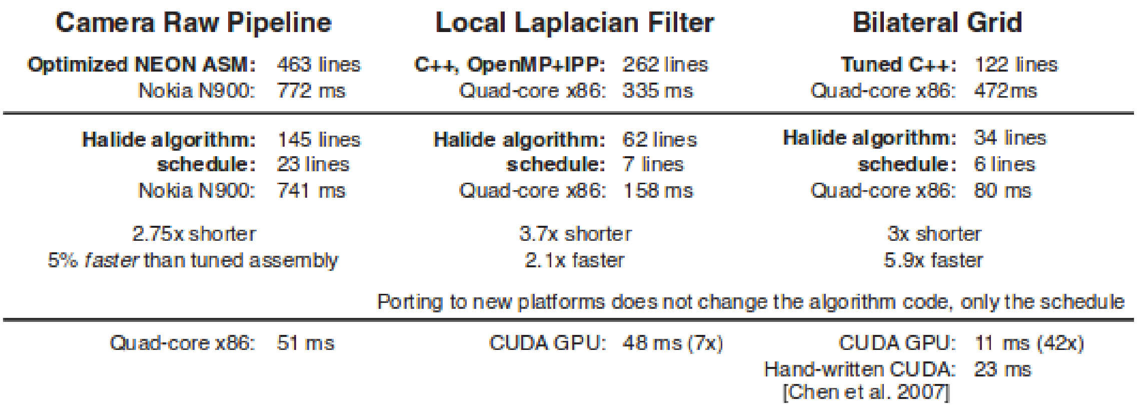
The article of Halide [16] shows some applications done to demonstrate how we can significantly reduce the code and the performance. The code of these applications are released, so we can download them and check how they are implemented

The target is compare the applications made by halide in front of the same application implemented in other languages. This programs are executed in the same machine. The applications implemented are: camera raw pipeline, local laplacian filter, and bilateral grid.

Camera raw pipeline converts a picture from an image sensor, that contains little processed data, to a color image. There are several processes to achieve this, like hot-pixel suppression, demosaicking, etc.

Local laplacian improve the contrast of the images. Finally, bilateral grid smoothes an image without loosing quality.

The image below, Figure 6.12, shows the results obtained by the researchers of MIT.



Camera Raw Pipeline	Local Laplacian Filter	Bilateral Grid
Optimized NEON ASM: 463 lines Nokia N900: 772 ms	**C++, OpenMP+IPP:** 262 lines Quad-core x86: 335 ms	**Tuned C++:** 122 lines Quad-core x86: 472ms
Halide algorithm: 145 lines **schedule:** 23 lines Nokia N900: 741 ms	**Halide algorithm:** 62 lines **schedule:** 7 lines Quad-core x86: 158 ms	**Halide algorithm:** 34 lines **schedule:** 6 lines Quad-core x86: 80 ms
2.75x shorter 5% *faster* than tuned assembly	3.7x shorter 2.1x faster	3x shorter 5.9x faster
	Porting to new platforms does not change the algorithm code, only the schedule	
Quad-core x86: 51 ms	CUDA GPU: 48 ms (7x)	CUDA GPU: 11 ms (42x) Hand-written CUDA: 23 ms [Chen et al. 2007]

Figure 6.12: Results. Figure from [16]

We can see that the code is more reduced than the others languages, separated by the algorithm and the schedule. Besides, we can observe that the applications implemented by Halide are faster.

6.5 Conclusion

As other language, Halide needs to be improved and generate outputs for others languages like OpenCL or OpenACC. Halide developed in a proper way can be very useful in the future. It gives many facilities to expert and non-expert programmers who are working in GPUs. Besides, the code is easily readable. This means that it does not take lot of time to understand code written by other people. Also, it is easily portable because, as we have seen in the previous sections, algorithm and schedule are totally decoupled. Thus, we just need to know the structure of our GPU.

7. DIFFERENCES

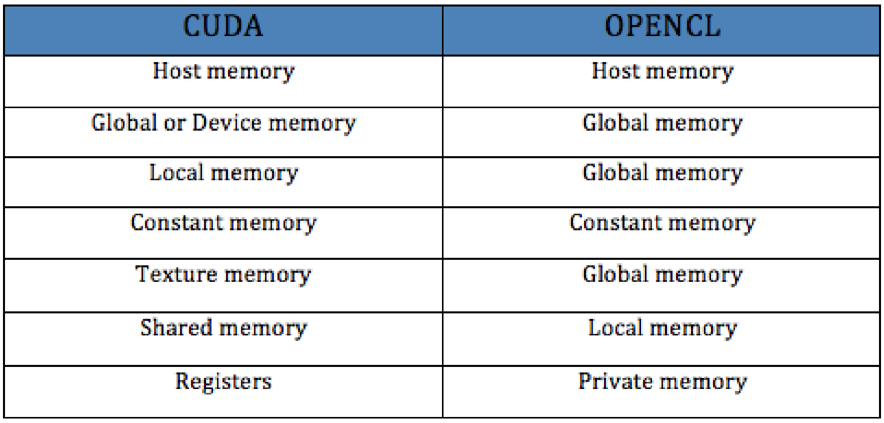
In this chapter we will show a comparative between the previous parallel computing languages explained: OpenCL, CUDA, OpenACC and Halide. The aim of the section is to show the reader the similarities and differences that we can find in the languages mentioned. Also, it would be useful for the reader to know which language a certain programmer could use in a given situation. This chapter is structured in three sections. CUDA vs OpenCL because they are platforms that share common properties and are the most common languages nowadays in the market. Next section will be OpenACC as an alternative for CUDA and OpenCL. OpenACC has some properties that CUDA and OpenCL don't have. Finally, Halide as an emergent language that could work on the previous platforms mentioned. Halide is an interesting language that can generate outputs for the other languages, it give us new ideas to address parallel computation that differs of the originals of CUDA, OpenCL and OpenACC.

7.1 CUDA vs OpenCL

CUDA is a language dedicated for NVIDA platforms and OpenCL is an open solution for parallel computing. NVIDIA devices can support OpenCL. However, CUDA can run only in NVIDIA GPUs. Knowing this, we could deduce that CUDA is not a parallel language which we should dedicate time studying. But we would be wrong. NVIDIA is taking good care of its language and this means that CUDA can be more powerful than OpenCL. So NVIDIA and OpenCL are not equal.

The first difference between these platforms is the terminology. While CUDA uses threads as a terminology, OpenCL uses words as groups (work-groups) or elements (work-items). This could lead to a confusion. CUDA threads correspond to work-items on OpenCL and CUDA blocks correspond to work-groups on OpenCL.

Despite the great nomenclature of both languages we can also find some differences in the memory model. The names of the memories in each language are different. This could also lead to errors when we are programming and managing the different memories. Specially when we are dealing with local memory which is not the same in CUDA and OpenCL. Next Figure 7.1, shows the main differences of nomenclatures about memory.



CUDA	OPENCL
Host memory	Host memory
Global or Device memory	Global memory
Local memory	Global memory
Constant memory	Constant memory
Texture memory	Global memory
Shared memory	Local memory
Registers	Private memory

Figure 7.1: Memory differences

OpenCL seems more difficult than CUDA in some cases. In OpenCL we have to create the execution context, load de disk sources and hand compile OpenCL kernels, while CUDA does not need to take responsibility for this task.

On the other hand, you can immediately determine the advantage of creating the context manually, one can determine at runtime if the parallel portions of the program should run on the CPU, GPU or both at the same time. In this characteristic OpenCL is better because CUDA can only run kernels on the GPU.

A similarity between them is the portability of kernel. The portability of a OpenCL kernel to CUDA and CUDA kernel to OpenCL is easy because the languages architectures are similar. The problem we can find is the portability between hosts.

In the case of the host we can find the most significantly difference. While in CUDA the management of the host executing a kernel is easy, the same operation in OpenCL using the same kernel is totally different and very much complicated. OpenCL has available some interesting APIs in C and C++ to make it easier. But programming is still complicated. We can deduce that for the portability the part that corresponds to the host is complex.

Is harder to know which program done in CUDA and OpenCL is the faster. CUDA has its own platform, so it can take advantage and just be optimized for one platform. Also OpenCL can take advantage of the platforms of CUDA, CUDA has its implementation of OpenCL running over its platforms. Besides, OpenCL has to deal with other platforms (not NVIDIA) and this makes harder the implementation of a parallel language and can not take all the advantage of a platform. It could lead to do not get the fullest optimization.

OpenCL offers more portability for different platforms. Not only in NVIDIA and AMD devices, it also can control a CPU with different processors. This means that

OpenCL is able to decide if some parts of a parallel program can be executed in a CPU or GPU giving us more flexibility.

7.2 OpenACC as an alternative

The most common frameworks in the market nowadays are CUDA and OpenCL. But there are alternative technologies like OpenACC that looks for parallelism computation and try to make easier the management of accelerators devices. One of the most important difference between OpenACC and CUDA/OpenCL is that OpenACC does not leave responsibilities to the programmer. On the other hand, CUDA/OpenCL allows to programmer take decisions of how to manage the memory. The OpenACC compiler has to manage in an efficient way the program through some directives given by the programmer. This means that CUDA and OpenCL are available just for expert programmers. This is not the case of OpenACC which was thought for non-expert programmers who do not need a high knowledge of the low-level architecture.

The terminology of this language is also different, like CUDA/OpenCL. In this case we have to define kernels or parallel that are the regions we will parallelize. Also, we have to define the gangs, workers, vector-lenght, etc. Gangs and wokers would be related with work-groups (groups) and work-items (elements) in case of OpenCL.

As a consequence, the compiler has the responsibility to allocate the program in the device. OpenACC makes easier the rewrite of some programs and increase highly the readability of them. So, the APIs provided by OpenACC are easier to understand for programmers. Hence, to develop an application in OpenACC is easier than do it in CUDA/OpenCL.

Ruse some programs programmed in C, C++ or Fortran made for other platforms is one of the best goals of OpenACC. Contrary to CUDA or OpenCL, with OpenACC we can just to change some lines of code, deleting the schedule, if there is, and add some directives that we will give to the compiler.

OpenACC provides a compiler guidelines for C, C++ and Fortran. It is not complicated to know the characteristics of the accelerator device attached to a CPU. This makes this language portable to different platforms, more than CUDA.

The memory model has different nomenclatures as CUDA and OpenCL. But the structure is similar, the memory of the host and the accelerator device are separated.

7.3 Halide vs CUDA, OpenCL and OpenACC

In this section we compare Halide with the other platforms as a language that could generate outputs for parallel languages. This language does not provide us a framework as OpenCL or CUDA. Halide is a compilation of libraries based in C++ and made by researchers at MIT. As the emergent language it is, it has some limitations comapred with the other parallel languages. But it has some interesting points to take into account.

The portability of Halide nowadays is very limited because it can generate only kernels for CUDA. And, how it has been, CUDA just runs in NVIDA platforms. However, researches are working in the generation of kernels for OpenCL. A task that will not take lot of time due the structure of Halide where the code is written in an standard language (C++) and the algorithm and schedule are totally separated. Thus, they have to adapt just the part of the schedule that has a few functions. So, now the portability is limited compared with OpenCL, but it is the same than CUDA.

As we are using a standard language it can be improved easily. It can be updated to achieve new targets as CUDA and OpenCL which are in constant development. All of them competing for the best places in the market.

A characteristic that differs this language from the others is the terminology. We will use specific functions like parallalize, vectorize, etc (explained in Halide chapter) to define threads, the schedule of the program, etc. But, also the algorithm will be programmed as in a functional language. Then it will be compiled in an imperative language. Contrasted to CUDA/OpenCL/OpenACC this way is totally different from the programs that are written always in an imperative way.

In this case Halide was thought to know accurately how the architecture of different platforms works, as CUDA or OpenCL. We have the responsibility to look for the best implementation in the schedule through the given functions. Halide does not requires a complex compiler as OpenACC to do the memory work. In this way, Halide requires a programmer with a good knowledge about the architecture.

One of the best points of Halide is readability. The decoupling of the algorithm and schedule cannot be found in CUDA, OpenCL or OpenACC. A program written in Halide can be much easier to read and understand than other languages. We can clearly distinguish the parts of the algorithm and update or improve a code made by another programmer can be easy. This point is very important because most of parallel programs code are very difficult to understand and hard to modify or improve for different people than the creator.

As an emergent language, this language is not consolidated as CUDA/OpenCL. But it can have future because it is a new idea and very much different than the other languages. It just has to be more developed and be available for other platforms like OpenCL, but it will be soon.

One of the problems of Halide compared to other languages is we have just a few applications. To check, the only place to look for some is the sources that can be download in the official web page of Halide. In contrast with CUDA/OpenCL/OpenACC we can find lots of sources and applications through internet and learn about this.

The recently studies of Halide are focused in image processing, just one field. In the other hand, CUDA/OpenCL/OpenACC are researching in many more fields as artificial intelligence, scientific researches, etc. So, nowadays Halide is much less developed than others parallel platforms, but with an interesting future.

8. CONCLUSIONS AND FUTURE WORK

8.1 Conclusions

Nowadays and in the future, parallel computation will be very useful for many applications. Technology is evolving quickly and it needs more power of computation. In this way, this kind of accelerator devices are very useful and important. The different languages mentioned in this thesis are very important to manage this technology. They give us the possibility to use these accelerators devices, such GPUs. We can not choose the best language. We can only mention the best goals for each in contrary of the others. We have mentioned here the best points for each language in the different fields.

OpenCL, CUDA, OpenACC and Halide has different terminologies and we cannot distinguish the best one. With a good user's manual we should be able to write a program without many difficulties. The most complicated language to program could be with Halide. Normal programmers are used to develop in imperative languages instead of functional languages. Because of this, it could be difficult for users that are not related with the way of programming.

If we are looking for portability OpenCL would be the best choice. It is the most supported language for different platforms. On the other site we find CUDA which runs only over NVIDIA platforms. For the moment Halide just can generate kernels for CUDA. But, if we are looking for efficiency CUDA would the best choice. This is because they are focusing on determined platforms and they take the fullest advantage of the device.

The complexity to develop code with these languages is not the same. In CUDA, OpenCL and Halide it is harder because we have to know how we will allocate the parts of the parallel program in the memory. Contrary to this, the OpenACC compiler take care of all this process. So, with OpenACC it is not necessary to have the same responsibility as in the others languages. The compiler manages it for us what make it appropriated for those non-expert programmers in low level architectures.

As it is well known in programming, readability is an important point to take into account. It is very important for a programmer to understand code made by others. In this case, Halide could be the best option. The decoupling of the algorithm from the schedule helps the programmer to develop programs easily and understandable.

The future of these languages is uncertain. At the moment CUDA and OpenCL are more powerful in the market. However, every day there are appearing new ways to manage the parallel computation like Halide and OpenACC. This competition is good because it provides other points of view and new totally different ideas. As we have seen until now, each language offers different features for every field and there is not a concrete language that we could say is the best one.

8.2 Future work

To finish we expose a possible future works we could do after this thesis:

- Those parallel languages are continually evolving trying to look for improvements. So, it requires us to continue study of these technologies as they can be changed significantly in the future.

- It would be interesting to program different applications and then test them one by one in different platforms to make a more practical analyse of these languages. Then check which applications could be better used depending of the language applied.

- Focusing in Halide. It would be interesting to study other fields, not only image processing. For example intelligence artificial or signal processing among others parallel applications.

- CUDA/OpenCL/OpenACC/Halide are not all the languages that supports parallel computing. It would be interesting to study others like Spiral (program generation system for DSP, the aim is to explore the parallelism and look for optimizations) [17] and compare with the already known.

- Try to make more accessible to the programmer these technologies and offer the possibility to play with codes without having to know very much about architecture.

- Make easier the way of synchronization. One of the biggest problems in parallel computing is how to synchronize the parts of the program. Either through semaphores, critical sections, shared memory, etc.

- Make more accessible the access to accelerator devices with high performance for common users, reducing the cost of the devices and the way of how to learn about use them.

- Focusing in CUDA it would be interesting to extend the power of managing kernels. It would be interesting to be able to generate kernels also for CPUs. Hence, we would have more control over the different parallel algorithms.

BIBLIOGRAPHY

[1] David B. Skillicorn, Domenico Talia. Models and Languages for Parallel Computation. Journal ACM Computing Surveys (CSUR), 1998. New York, USA.

[2] Jason Ng, Jeffrey J. Goldberger. Practical Signal and Image Processing in Clinical Cardiology. Published by Springer, 2010.

[3] John D. Lovell, David C. Nagel. Digital filtering and signal processing in Behaviour Research Methods and Instrumentation. Perceptual Systems Laboratory University of California, Los Angeles, 1973.

[4] Ananth Grama, George Karypis. Introduction to Parallel Computing. Published by Pearson Addison Wesley, 1994.

[5] Diederich Hinrichsen, Anthony J. Pritchard. Mathematical Models. Mathematical Systems Theory I. Published by Springer, 2005.

[6] Jens Kubacki. Artificial Intelligence. Technology Guide. Published by Springer, 2009.

[7] Kanupriya Gulati, Sunil P. Khatri. GPU Architecture and the CUDA Programming Model, Hardware Acceleration of EDA Algorithms. Department of Electrical and Computer Engineering, Texas A and M University. Published by Springer, 2010.

[8] Max Grossman, Alina Simion Sbirlea. CnC-CUDA: Declarative Programming for GPUs. Languages and Compilers for Parallel Computing, 2011. Department of Computer Science, Rice University.

[9] David B. Kirk, Chief Scientist. NVIDIA CUDA Software and GPU Parallel Computing Architecture. NVIDIA Corportation 2006-2008. Available online in the web page of NVIDIA (http://www.nvidia.com).

[10] Lindholm, E., Nickolls, J., Oberman, S., Montrym, J. NVIDIA Tesla: A Unified Graphics and Computing Architecture. Published by the IEEE Computer Society, 2008.

[11] Jungwon Kim, Sangmin Seo, Jun Lee. OpenCL as a Programming Model for GPU Clusters. Languages and Compilers for Parallel Computing. Center of Manycore Programming School of Computer Science and Engineering, Seoul National University. Published by Springer, 2013.

[12] Peter Collingbourne, Cristian Cadar Symbolic Testing of OpenCL Code. Hardware and Software: Verification and Testing. Department of Computing, Imperial College London. Published by Springer 2012.

[13] Ralf Karrenberg, Sebastian Hack. Improving Performance of OpenCL on CPUs. Compiler construction. Published by Springer, 2012.

[14] Sandra Wienke, Paul Springer, Christian Terboven. OpenACC First Experiences with Real-World Applications. JARA, RWTH Aachen University, Germany, Center for Computing and Communication. Published by Springer, 2012.

[15] The OpenACC application programming interface, 2013. Available in (http://www.openacc-standard.org).

[16] Jonathan Ragan Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman Amarasinghe, Saman Amarasinghe. Decoupling Algorithms from Schedules for Easy Optimization of Image Processing Pipelines. MIT Computer Science and Artificial Intelligence Laboratory, Massachusetts, 2012.

[17] Markus Puschel, Franz Franchetti, Yevgen Voronenko. Spiral. Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA, USA, 2011.