**Signal Processing with CUDA and OpenCL**

*Benoît Arnal - Bochra Ben Abda - Soukaina Bennani – Sylvain Premel*

*January 21st 2011*

***Summary :*** *This paper tackles the issue of two parallel computing architectures that are CUDA and OpenCL and that enables dramatic increases in computing performance by harnessing the power of the GPU (Graphics Processing Unit) also called GPGPU(General-Purpose computation on Graphics Processing Units ). This paper is composed in five parts. The first two parts are dedicated to the presentation of the GPGPU in the context of the pursuit of high performance signal processing algorithms. The following part deals with the specificities of such architectures and highlights them through the description of a kernel execution. The third part is the put into practice of these architectures on a complex optical flow algorithm called FOLKI. Finally it presents some results in order to conclude on the interest and the efficiency of GPGPU implementation face to the usual CPU one.*

## I. Introduction

For the last twenty years, and particularly since the advent of complex processing such as 3D ones, the demands for efficient computing techniques have always been more numerous. Processor manufacturers have first tried to create always faster architectures by increasing CPUs' frequency in order to speed the sequential execution of the instructions. However, and because of physical limitations such as heat dissipation or power consumption, a continuous increase is not possible. A solution has been proposed to by-pass this problem: the GPU computing through technologies such as CUDA and OpenCL.

Since the beginning of the century, the computing power of GPUs has widely increased due to the explosion of the needs in signal processing and in video games applications, to name but a few, which require intensive computing in order to execute their purposes efficiently. This alternative has the advantage of liberating the computers of important computation times relative to the 3D scenes. Nowadays, the GPU computing is more than just a means to compute 3D scenes. Indeed, thanks to the rising of new hardware and new software technologies, GPUs are part of cheap means to achieve parallel and distributed computing. This paper tackles the issue of two rival technologies that are CUDA and OpenCL, which are used for few years in GPU computing, also called GPGPU.

*Figure 1 : A graphic card*

## II. Context

Computing is evolving from "central processing" on the CPU to "co-processing" on the CPU and GPU. A GPU is a specialized microprocessor attached to a graphics card and dedicated to calculating floating point operations. It incorporates custom microchips which contain special mathematical operations commonly used in graphics rendering which can compute, independently the ones from the others, operations on different elements. It is this aspect that makes it very attractive for high efficiency computing. Indeed, since a GPU is designed to transform an amount of pictures, from a frame sequence,(called textures) and several vertexes in a 3D scene and project it on screen several tens of time per second, the graphic card needs to process as many data as possible in a minimum of time. Modern GPUs are very efficient at manipulating computer graphics, and their highly parallel structure makes them more effective than general-purpose CPUs for a range of complex algorithms.

General-purpose computing on graphics processing units (GPGPU) is the technique of using a GPU, which typically

handles computation only for computer graphics, to perform computation in applications traditionally handled by the CPU. The GPGPU, as it exists today is due to the combined efforts of several companies such as Nvidia, ATI or even Khronos Group.

As it is said before, this paper tackels only two GPU computing technologies : CUDA and OpenCL but of course there are many others.

CUDA (Compute Unified Device Architecture), is a parallel computing architecture developed by NVIDIA. It is the computing engine in NVIDIA GPUs that is accessible to software developers through variants of industry standard programming languages.

OpenCL (Open Computing Language) is an open royalty-free framework for writing programs that execute across heterogeneous platforms consisting of CPUs, GPUs, and other processors. It gives any application access to the Graphics Processing Unit for non-graphical computing.

## III. GPU Architecture

Processors hierarchy

GPUs have been originally design for processing multiple data concurrently as it is what graphic rendering is about. Consequently all their architecture revolves around this same constraint: parallelism.

Indeed; while recent computers possess only 2 to 4 processors capable to read and execute sequentially a various range of instructions -, new GPUs contains up to 800 computing-oriented processors enabling to execute instruction concurrently on a large data flow. This architecture is called SIMD (Same instruction, Multiple Data)

When looking on the inside of a GPU, what could be seen would be a set of entities called Streaming Multiprocessors (SM). Each of them contains several computing unit able execute instructions called Streaming Processors (SP), they are basically the core of the parallelism aspect of GPU : One GPU can compute concurrently as many instruction as is possesses of Streaming Processors.

Memory hierarchy

In order to compute data, it is first needed to be stored into memories. GPU contains many kinds of memories. The first one and the main one is called DRAM (Dynamic Random Access Memory). This memory also called Device memory is physically placed outside of the GPU chipset. This is the reason why:

- It is available in big quantity : new GPUs contains up to 4GB of DRAM
- It possesses a big latency: 400 to 800 cycles are needed to access a data stored on the DRAM.

The DRAM is itself divided into several compartments:

- The Global memory which is used to store most of the variables
- The Local memory which is used to store variables too big to be placed in registers.
- The Texture memory: this memory enables filtering on 2D or 3D data frequently use in graphic processing.
- The Constant memory.

All of those memories are physically bound to the DRAM and consequently, have a big latency. However constant and texture memory being cached, their access is actually much quicker than the global and local memory, we will see details further.

Inside the GPU chipset are other kind of memories which are internal to each Streaming Multiprocessors (two data stored in different Multiprocessor memories can't have access to each other.)

The registers: registers have basically the same usage than in computer processor, they are very small memories (usually 32bits) which are extremely fast to access (less than 1cycle). That is generally where static local variables are stored.

The Shared Memory : to sum up, the shared memory has two major advantage:

- Like the registers, this memory has a very low latency.
- Like DRAM, it enables dynamic allocation.

However due to its small size (16 to 32kB per multiprocessors), shared memory needs to be managed very carefully.

In addition to those memories, each multiprocessor is also composed of a Constant cache and a Texture cache of 8kB each. Those are read-only memories which enable to store recently used data from Texture Memory and

Constant Memory (and therefore gain a quicker access for the next usage).

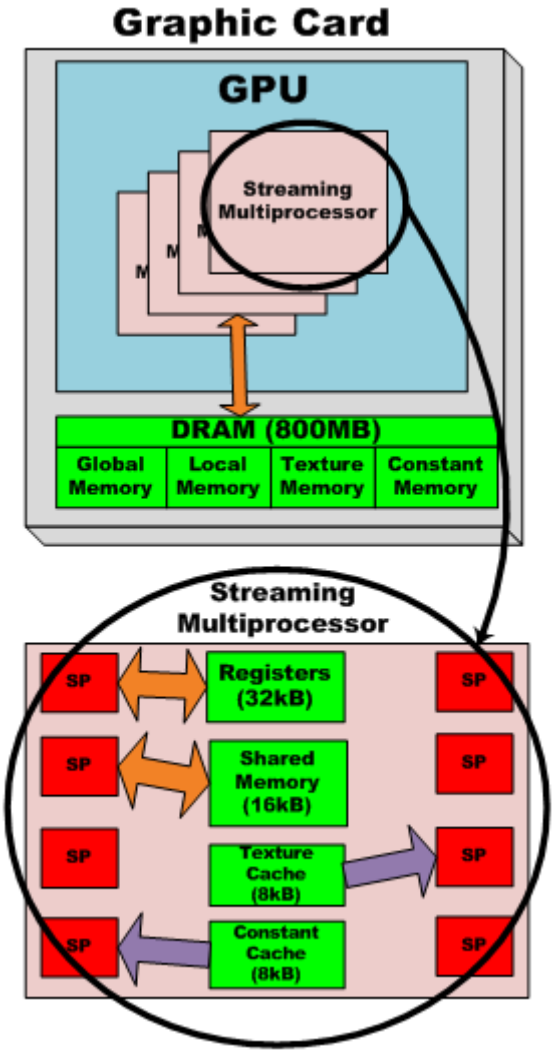


*Figure 2: Simplified architecture of a GPU*

## IV. Programming Model of CUDA and OpenCL

Both programming language CUDA and OpenCL have settled a model which enables the use of parallel architectures through a system of multiple divisions and subdivisions of the data that need to be processed.
Kernel (CUDA/OpenCL) are functions which are executed on a grid of computing units called threads (CUDA)/work-items (OpenCL). Each thread executed is given a unique ID accessible through kernel functions. To facilitate their handling, threads/work-item are grouped into blocks (CUDA)/work-group (OpenCL) which are themselves gathered in one Grid (CUDA).

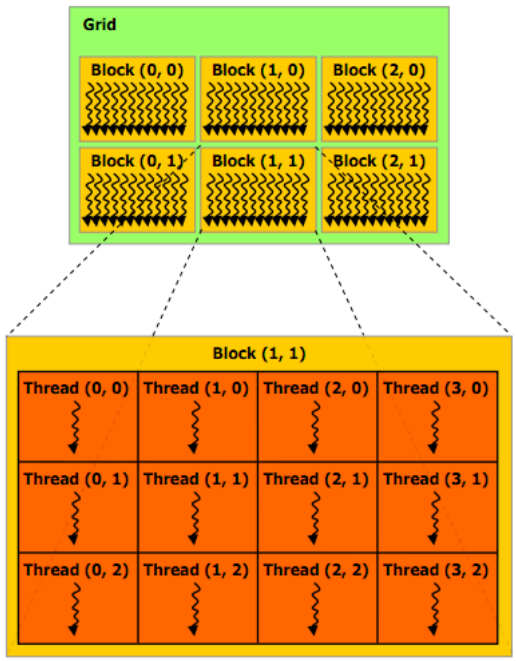


*Figure 3: Programming Model with CUDA*

Implementation through CUDA and OpenCL basically consists in the following steps:

- Dividing your data into as many sub-data as needed
- Associating each one of them to a unique thread/work-item ID.
- Executing the kernel function on the grid of threads/work-item.

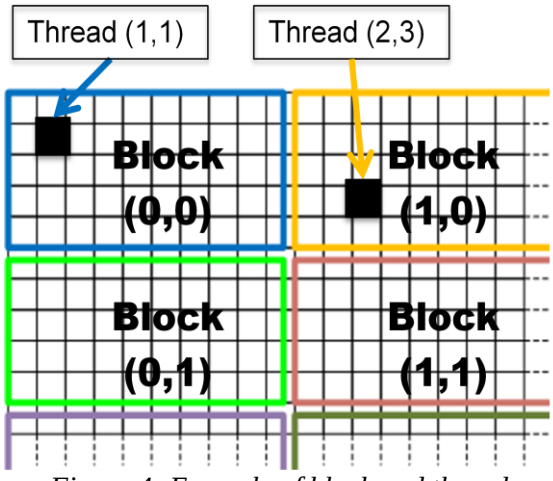


*Figure 4: Example of block and thread division.*

Flow processing

During execution, Blocks/Work Group will be split between the available Streaming Multiprocessors while each Thread/Work-item

within a Block/Work-group will be associated toward different Streaming Processors.
Considering thread/block repartition is managed during execution, one cannot know a priori in what orders they will be executed, neither if they will be executed sequentially or concurrently.
Here is an exemple of a data flow processing using a 3x2 blocks division and 3x2 threads division.

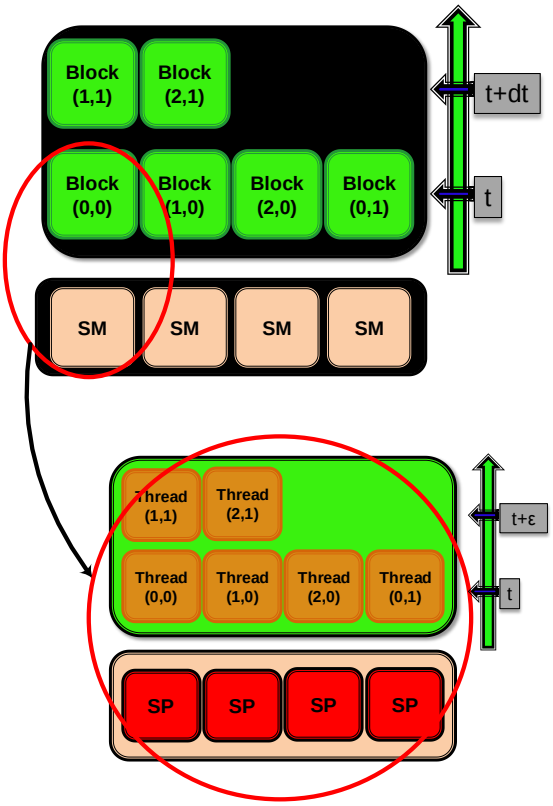


*Figure 5: Example of a data flow processing.*

## IV. FOLKI algorithm

At first, it is important to remind what an optical flow is. An optical flow is the observable displacement field measured in the image coordinate frame in a video sequence. It results from independent motions of objects within the scene as well as from sensor ego-motion, but can also be impacted by temporally varying illumination.
The concept explained above is illustrated by the following picture

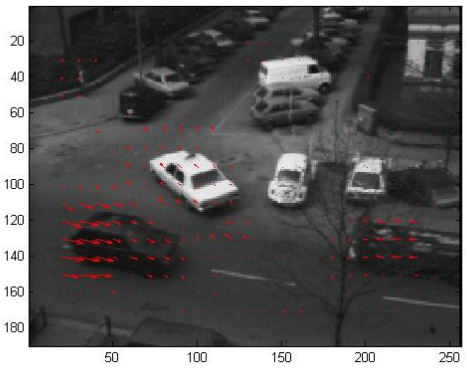


*Figure 6: Optical flow between two pictures*

FOLKI is an algorithm developed by G. Le Besnerais and F. Champagnat in 2005 in order to provide a fast dense optical flow estimates using a window-based iterative and multiresolution registration. FOLKI relies on a specific Taylor expansion of the window registration term which leads to a consistent iterative scheme while saving many image interpolation operations. This specificity enables FOLKI to be much faster than other iterative algorithms, as, for instance, the Pyramidal Lucas-Kanade algorithm, and also more accurate.

FOLKI is an optical flow algorithm, efficient for large scale movements, and based on a multiresolution approach. It is used in many signal processing algorithms such as motion estimation, stereo vision or even image registration.

Considering this, why use the FOLKI algorithm to prove the interest of GPU implementation face to the CPU one? The reasons are simple. FOLKI is both simple and complex. On the one hand, FOLKI is finally rather easy to implement in CUDA and OpenCL for a novice, but on the other hand, it uses most of the elementary signal processing operations that are term by term additions, subtractions, divisions and multiplications but also convolution or even bilinear interpolation. This enables the comparison between the efficiency of CPU and combined CPU-GPU implementations.

Before showing the CUDA and OpenCL implementations, the last thing to highlight is the algorithm itself. The figure

below is a block diagram of the FOLKI algorithm.

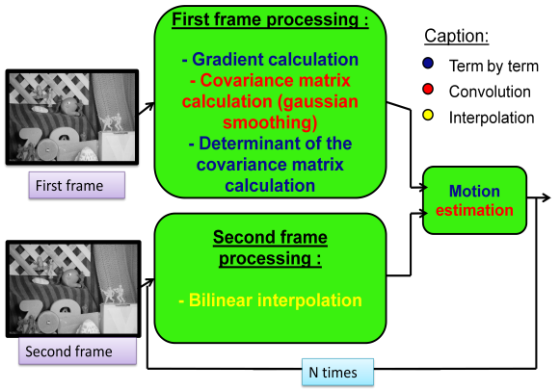


*Figure 7, FOLKI algorithm*

## V- CUDA and OpenCL implementation of the FOLKI algorithm:

As seen above, FOLKI algorithm is mainly composed of three types of operations:

Term by term operation

Those are the simplest and fastest to implement trough a parallel based architecture: images first need to be divided into blocks, each blocks being divided into threads. Then each pixel's coordinate is associated to a unique thread in charge to compute the term by term operation and store the result. All of the data are stored in the global memory since only one reading per computation is performed during the operation.

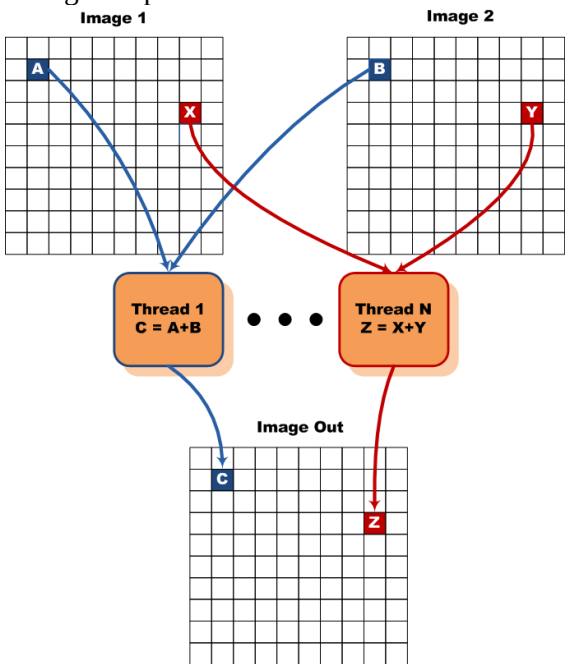


*Figure 8: Thread processing for a term by term operation*

Convolution algorithm

Three implementation of convolution have been planned.

The first one consisted of doing the exact same cut out of the data than term by term operations: each pixel's coordinate is associated to a thread which computes the convolution of its associated pixel and then, store the result in global memory. In this implementation, all images are stored and managed through global memory.
The main advantage of this implantation is the ease of the conception and the possibility to use large kernel filter without any troubles. However performances of this algorithm are - as expected - quite low. Indeed, if you consider a NxN kernel filter, doing one convolution on the global memory is equivalent to do NxN reading on global memory per convolution. Considering the extremely big latency in accessing global memory, one could sum up this implementation as follow: all the beneficial effects of parallelism are wasted out in time-transferring data.

The second implantation had for aim to make some use of the shared memory which is much quicker to access than global memory. However one block of shared memory cannot store one image entirely. (To compare, a 512x512 gray image on 256 levels has a size of 262kB). Therefore, the image needs to be divided into two-dimensional blocks.
First the size of one block is set, and then the image is divided into as many blocks as necessary depending of image size.
Each block is stored into a potentially different Shared Memory, meaning blocks cannot communicate directly.
The execution of a thread is almost identical to the previous implementation, but this time, before performing the convolution, the thread first loads its pixel into shared memory. Only problem is when the thread's pixel is stored on the edge of a block, it cannot work:

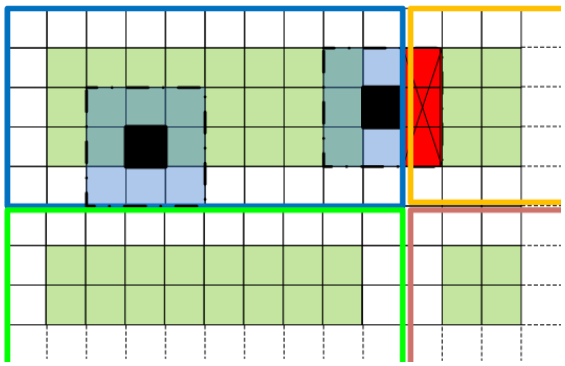

*Figure 9: Illustration of edge problems encountered with a classic block and thread division, the filter size is 3x3.*

Indeed, a convolution requires an access to its neighborhood, and yet when it comes to a pixel located in the edge of a block, some of its neighbor pixels are located in a different shared memory, and therefore are not available. Since edge computation cannot be done, the storing of data into shared memory, (and therefore the block division of image) has to be in some way redundant. Here is a figure of how the block division has been made in order to prevent those edge problems.

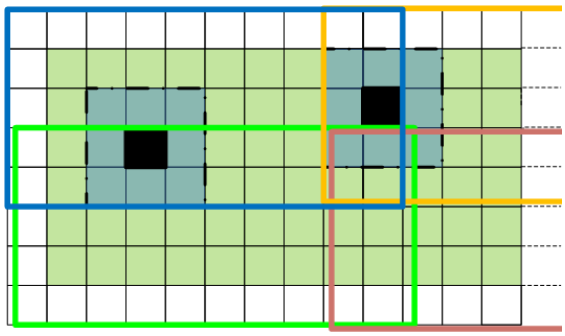

*Figure 10: Illustration of blocks division with block lapping.*

The technique used consisted of performing a block division with block lapping to ensure every data will be processed once and only once.
As shown in the figure, each block laps the previous one by a certain surface function of the filter size, by doing so, edges data from one block become centered data from its neighbor block. Every data (except the usual image edges where convolution is not define) can be processed.
This implementation is faster than the first one, but has one major drawback: since one block of threads cannot contain more than 512 threads (CUDA restriction), this implies two things
(1)A maximum of 512 pixels can be stored in shared memory which is equivalent to 2k. This is somewhat an under usage of the 16kB offered by the shared memory
(2)The size of the kernel filter cannot exceed 512.(22x22 for a square filter).

Those downsides led to the third and final implementation: the block lapping division of the image remains exactly the same. However, in this implantation, each block is divided into Nx1 threads. Then, each thread loads into shared memory –not one - but a row of P pixels and performs convolutions on the entire row (except the row edges which presents the same problems as above).

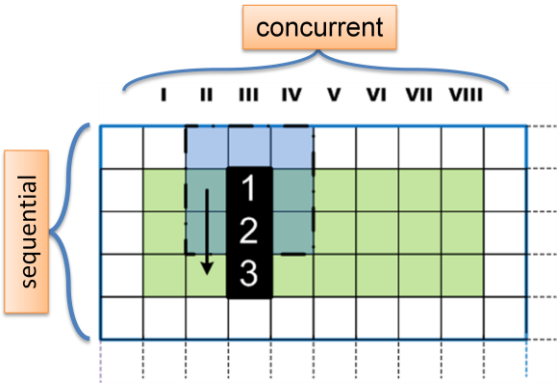


*Figure 11: Illustration of the third implementation of convolution with N = 10 and P = 5.*

Among the three implementations, this one provides the best performances since it makes the best usage of the shared memory.
Let's add that in order to apply convolution with filter of non-static size, filter is dynamically loaded into the shared memory.
When executing the convolution with this algorithm, the user has to ensure two things:

- The size of the block-image plus the size of filter stored into shared memory must not exceed shared memory size.
- The row of pixels on which a thread work must be longer than the filter height.

**Interpolation.**
Since interpolation is an operation widely used in many graphic rendering algorithms, GPU's provides hardware structures that are specifically designed to accelerate and

facilitate execution of those kinds of algorithms.
Indeed, as seen above, the texture memory enables to work efficiently with two or three dimensional data and automatically perform filtering algorithms like interpolations.
In CUDA, the steps to benefit from texture possibilities are the following:

i. Allocating a two/three dimensional CUDA array.
ii. Loading the image into the CUDA array.
iii. Creating a Texture
iv. Binding the texture to the CUDA array.
v. Choosing texture options.
vi. Choosing the type of interpolation.

Once those steps have been validated, the function tex2D()/tex3D() enables to directly work with non integer coordinates, the chosen interpolation is automatically performed.
As expected, interpolation through Texture memory possibilities showed much better performances than global memory based interpolation.

## VI- Results

We tested CUDA, C and OpenCL versions of our application on an NVIDIA GeForce GT 9500. Both CUDA and OpenCL development tools were at version 3.2.

For the experiments in this paper, we aimed for maximum performance. So, no interaction with the computer was attempted during the actual data-gathering runs to make sure that the GPU's computing power remained dedicated to the FOLKI application.

The application goes through the following steps during its run:
(1) Setup the GPU (includes selecting the device, compiling the kernel for OpenCL..)
(2) Allocating vectors in device memory,
(3) Copy data from CPU to the GPU,
(4) Run the kernel on the GPU,
(5) Copy data back to the host,
(6) Process the returned data using the CPU and output the results, and free device memory.

Images considered in tests are the following:


*Figure 12: Test images: the Wooden data set from University of Middlebury*

To get time execution, we have used the clock function and we have run the kernel 10 times with both CUDA and OpenCL to get repeatable times.

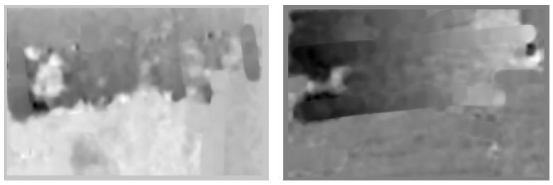

*Figure 13: Resulting images of the FOLKI algorithm – Flow motion according to X axis and Y axis.*

The following table reports the total amount of time needed for different operations from allocation and transfer to transfer and free device memory through the FOLKI algorithm.

| Version | CUDA | Open CL |
|---|---|---|
| Allocation and Transfer | 171.43 ms | **11.53 ms** |
| Term by term operation | **47.54 ms** | 298.44 ms |
| Convolution | **25.20 s** | 27.81 s |
| Operation on CPU | 22.28 ms | **10.57 ms** |
| Interpolation | **0.22 ms** | 39.80 ms |
| Transfer and desallocation | **2.43 ms** | 7.71 ms |

*Table 1- Detailed time execution*

Table 1 shows that CUDA's kernels execution (Element by element operation, convolution and Interpolation) are faster than OpenCL's, despite the fact that the two implementations are running nearly identical code. This is due to the fact that CUDA model is directly based on the NVIDIA architecture, therefore, binaries optimization performed through the

CUDA compiler (nvcc) are more efficient than openCL. However let's note that OpenCL strangely manage to obtain better result during the allocation and transfer phase.

This test led us to measure the whole execution time which is reported in table 2.
The End-To-End time shows the amount of time needed to run the whole application from the beginning to the end, corresponding to time spent for steps 1 through 6.

| **Version** | **CUDA** | **C** | **Open CL** |
|---|---|---|---|
| **End-To-End Time** | 25.44 s | 40.98 s | 28.19 s |

*Table 2- End-To-End Time*

The aim of this test was to compare sequential version and parallel implementation.
This table shows that computation on GPU is two times as fast as simple CPU computing.

In order to look at the impact of memory handling on the performances, we compared two methods in implementing the CUDA version: The first method using the convolution with Global memory and the second using the shared memory convolution.

| | **CPU** | **GLOBAL MEMORY** | **SHARED MEMORY** |
|---|---|---|---|
| End-To-End Time | 41.2s | 25.44 s | 6.95 s |
| Percentage of gained time | - | 38.25 % | 83.13% |

*Table 3 – Memory Benchmark*

These results reveal that using shared memory increases the evaluated gain by 83.13%. Shared Memory maximizes greatly time performances.
This can be explained by reducing access to the device memory by staging data into shared memory.
The last test aimed to benchmark two GPU available at ENSEIRB-MATMECA.

| | **NIVIDIA GT 9500** | **NVIDIA GT 240** |
|---|---|---|
| End-To-End time | 28.19 s | 2.28 s |

*Table 4- GPU benchmark*

Table 4 shows that not only memory can raise the time gain but is also extremely related to the GPU architecture used. This test was only performed with the OpenCL implementation.

## VII- Conclusion:

From our report we were able to draw the conclusion that the power of GPU computing is further proof whether we choose to use OpenCL or CUDA. Nevertheless, CUDA version had shown a best efficiency and performance but unlike the portability of OpenCL, CUDA-enabled GPUs are only available from NVIDIA.
The chipmaker of NVIDIA announced last month that they are working on the "Denver Project" where They will couples the power of parallel computing GPU to the qualities of hearts serial "ARM"; As with its rival AMD Fusion APU (which meet CPU and GPU multicore x86 with DirectX 11 on a single chip).
Maybe the future is in convergence of CPU and GPU to overcome the currently instable GPGPU market.

## References


[1] Guy Le Besnerais and Frédéric Champagnat, *« Flot optique rapide sur grandes images par GPU et applications »*
[2] CUVELIER Thibaut -*"CUDA approfondi"*
[3] NVIDIA - *"CUDA:Compute Unified Device Architecture"*
[4] NVIDIA - *"OpenCL Programming Guide for the CUDA Architecture"*
[5] Guy Le Besnerais and Frédéric Champagnat, "*Dense optical flow estimation by iterative local window registration," in Proceedings of IEEE ICIP'05, vol. 1, pp. I–137–40, Genova, Italy, Sept. 2005.*
[6] Guy Le Besnerais, Frédéric Champagnat, Aurélien Plyer, Riadh Fezzani ONERA/DTIM *Une approche itérative rapide pour le calcul de flot optique par correlation*
[7] Dr. Dobb's, "*CUDA, Supercomputing for the Masses"*