Treball fi de carrera

ENGINYERIA TÈCNICA EN INFORMÀTICA DE SISTEMES

Facultat de Matemàtiques
Universitat de Barcelona

OpenCL based machine learning labeling of biomedical datasets

---

Oscar Amorós Huguet

Directors: Sergio Escalera, Anna Puig
Developed at: Departament de Matemàtica Aplicada i Anàlisi. UB

Barcelona, 20th september 2010

Índice de contenido

1 Introduction

Once we analyzed OpenCL as seen in the TAD Appendix, we decided to use it for one project that could take advantage of GPU-OpenCL, and has not yet been implemented.

Anna and Sergio were developing a GPGPU adapted innovative algorithm for MRI reconstruction in GSGL, but needed more performance so I implemented an OpenCL version of the algorithm.

I started with a first optimization step, focusing on the slowest part that is PCIe, to later focus on typical optimization techniques and obtain the best performance I can.

1.1 Project scope

Due to GPU-OpenCL nature, this project is related to some different areas of computer engineering.

First of all, programming languages. In this case ANSI C and C++, for the original source code to be implemented in OpenCL, and OpenCL C.

Second, concepts like API, or Framework appear when talking about OpenCL Host code.

Another two areas are computer architecture and operating systems. To deeply understand OpenCL, is convenient to know some OS basic concepts, because OpenCL interacts deeply with it.

Computer architecture is needed for understanding how to program OpenCL and mandatory to do performance-OpenCL. Also a few basic concepts of operating systems can be needed for performance OpenCL when handling concurrent computing and data transfers.

The last main area for OpenCL is parallel programming. It is convenient to know about common parallel programming paradigms, and then know which ones OpenCL uses. OpenCL is mainly about parallelism since this is the basis for High Performance Computing, and even commercial hardware, but also about parallelism between heterogeneous computing architectures.

1.2 Motivation

When we knew about GPU computing and Cell Broadband Engine, we found it very interesting since with GPU performance, there were some magical equations. Price/Performance, and Power/Performance were incredibly improved for lots of scientific workloads. It was as to say that you could have Supercomputer performance in a Workstation, or even a Desktop. Today it includes Laptops and even a few Netbooks.

There are several biomedical applications that need to extract different objects of interest, such as tissues and organs, contained into a volume dataset from MRI, CT, fMRI and PET input captions. Diagnostic methods, structures volume measurements, and visualization systems require to specify to which anatomical structure each sample/voxel belongs. In the bibliography, Transfer Functions has been used in order to directly associate optical properties or labels to the different data samples according their belonging to a particular structure in the underlying data. However, in general, the anatomical structures are complex, and relationships between them do not allow to separate sufficiently the different structures. In these cases, Transfer Functions alone do not suffice in order to separate different objects and it becomes necessary to use labeling, or segmentation methods. In this sense, several approaches to label biomedical datasets have been proposed to discriminate different anatomical structures in an output tagged dataset depending on the used imaging modality.

Among existing methods, supervised learning methods for segmentation have been devised to easily analyze biomedical datasets by a non-expert user. In a preprocess, an expert user, such a radiologist, should identify a subset of samples of each anatomical structure. Then, during the learning step, the supervised method defines a classifier to be automatically used in the testing or classification stage. Since learning phase is carried on a pre-process, before labeling, the classifier can be used in different classifications several times by an inexpert user. Thus, to optimize the classification stage becomes imperative. Thanks to the recent emerging technologies of multi-core CPUs and GPUs, as well as new software languages, such as NVIDIA's CUDA and OpenCL, we propose to parallelize the classification step of a well-know supervised method, called Adaboost, in the GPU.

This project then is an effort to improve performance for a modern classification algorithm, using new High Performance Computing technologies.

1.3 General goals

The first goal of this implementation is to improve the performance achieved for binary voxel classification in OpenMP C++ and GSGL.

1.4 Specific Goals

We propose an alternative representation of the Adaboost binary classifier in order to increase the performance of the classical implementations of the Adaboost.

We use this proposed representation to define and implement a new GPU-based parallelized Adaboost testing stage using the GPU-OpenCL architecture.

We provide numerical experiments based on large available data sets and we compare our results to CPU-based strategies in terms of time and labeling speeds.

1.5 Project memory organization

This memory is organized in 5 main parts. A previous work section, were all the information related to the project is explained, except for the information related to OpenCL that is already explained in the TAD appendix. It is followed by a proposal section where all the work done is explained. Then a section for results shows them followed by another one with conclusions.

The last main section is a Gantt Chart showing the time spend on the project. Then a section for references and all the appendixes.

2 Previous work

In this section we explain all the basic concepts related to the project that are already known and can be found in the literature. These concepts are basic to understand the project, and some of them are explained deeply in the TAD appendix. We recommend reading the TAD to completely understand this work.

2.1 Background

Now we discuss some related work already done in the area to understand the point from where this project starts.

2.1.1. Volumetric Pixel World

A Volumetric Pixel Model or Voxel Model is a coordinate defined volume, divided as a regular grid of equal spaced and sized points with a common origin, that usually is (0,0,0). Each sub-cube defined by the coordinates of the grid is a Voxel or Volumetric Pixel.

Voxels have a size defined by the volume spacing. A spacing parameter is defined for each of the three dimensions of the volume. A density value is defined for each of the Voxels. It is possible then to visualize only the voxels for a given density, so a Voxel model can contain as many objects as density levels defined (see Figure 1.1) or a specific object can contain several density values (see Figure 1.2). For this reason, to select a specific structure becomes, sometimes, a tidy and manual task.

This technique is used in areas where representing regularly-sampled spaces that are non-homogeneously filled is needed. One of the most common areas is medical imaging, obtaining the density values from sources like MRI (Magnetic Resonance Imaging) and CT (Computed Tomographies).

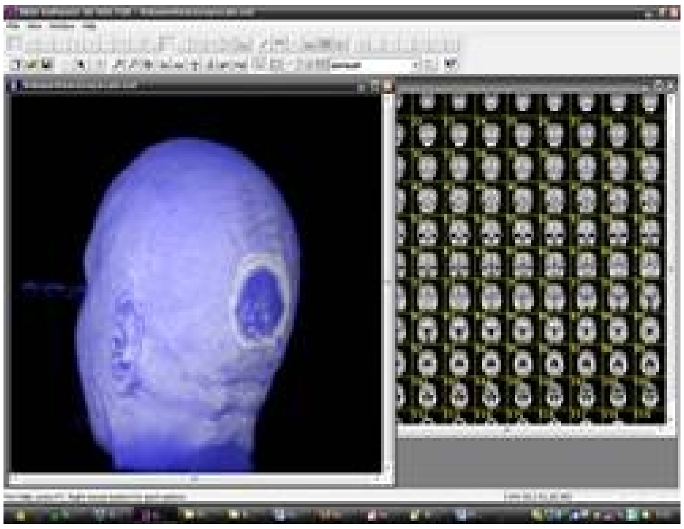


Figure 1.1: Reconstruction of a 3D voxel model from a head MRI data set.

In this work we use an Adaboost based algorithm adapted to massively parallel processors like GPU's to classify the density values of a Voxel model. This Voxel World is obtained from MRI and CT techniques.

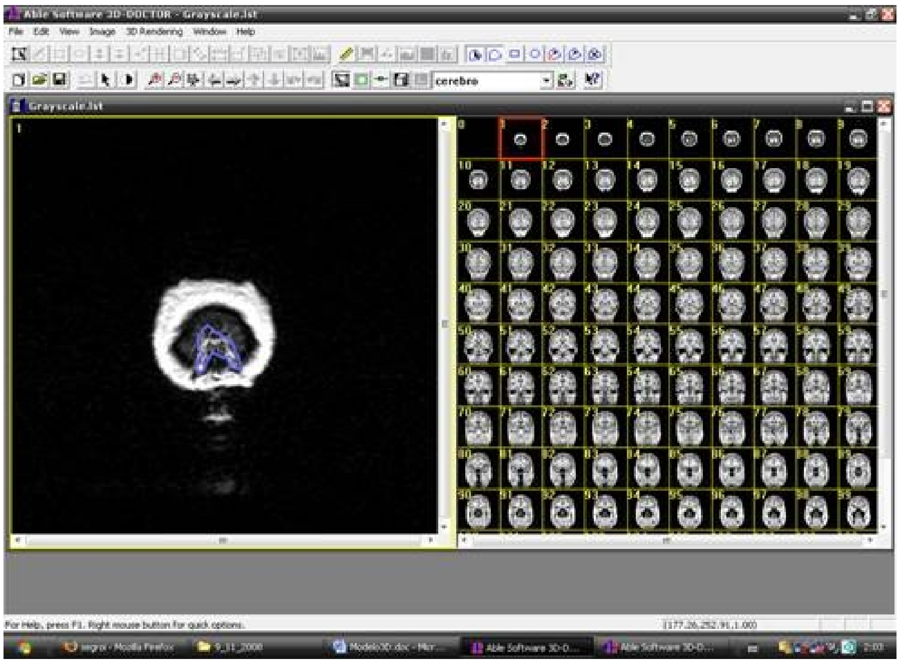

Able Software 3D-DOCTOR - Grayscale.lst
File Edit View Image 3D Rendering Window Help
cerebro
Grayscale.lst
For Help, press F1. Right mouse button for quick options.

Figure 1.2: selection of an anatomical structure from density values

2.1.2. Machine Learning methods to classify datasets

Data classification is a technique that groups elements of an array of data based in some criteria.

For our data set, that is a Voxel model, we wanted to use a Machine Learning Classification method. The idea is to provide users as doctors with an interface to a program that can learn from them. So the user can correct or introduce some new information into the classification results, and the program learns from that input to gain more precision and new classification capabilities without reprogramming the software.

Machine learning is a scientific discipline that is concerned with the design and development of algorithms that allow computers to evolve behaviors based on empirical data, such as from sensor data or databases. A learner can take advantage of examples (data) to capture characteristics of interest of their unknown underlying probability distribution. Data can be seen as examples that illustrate relations between observed variables. A major focus of machine learning research is to automatically learn to recognize complex patterns and make intelligent decisions based on data. The difficulty lies in the fact that the set of all possible behaviors given all possible inputs is too large to be covered by the set of observed examples (training data). Hence the learner must generalize from the given examples, so as to be able to produce a useful output in new cases.

AdaBoost, short for Adaptive Boosting, is a machine learning algorithm, formulated by Yoav Freund and Robert Schapire [1]. It is a meta-algorithm, and can be used in conjunction with many other learning algorithms to improve their performance. AdaBoost is adaptive in the sense that subsequent classifiers built are tweaked in favor of those instances misclassified by previous classifiers. AdaBoost is sensitive to noisy data and outliers. However in some problems it can be less susceptible to the over-fitting problem than most learning algorithms.

We focus on the Discrete version of Adaboost, which has shown robust results in real applications. Given a set of N training samples $(x_1,y_1),..,(x_N,y_N)$, being x_i a vector valued feature and $y_i = -1$ or 1. We define $F(x) = \sum^{M}_{1} c_f f_m(x)$, where each $f_m(x)$ is a classifier producing values ±1 and c_m are constants; the corresponding prediction is sign$(F(x))$. The Adaboost procedure trains the classifiers $f_m(x)$ on weighed versions of the training sample, giving higher weights to cases that are currently misclassified. This is done for a sequence of weighted samples, and then the final classifier is defined to be a linear combination of the classifiers from each stage. E_w represents expectation over the training data with weights $w = (w_1,w_2,..,w_N)$, and $1(S)$ is the indicator of the set S. For a good generalization of $F(x)$, each $f_m(x)$ is required to obtain a classification prediction just better than random. Thus, the most common "weak classifier" f_m is the "decision stump". For each $f_m(x)$ we just need to compute a threshold value and a polarity to take a binary decision, selecting that one that minimizes the error based on the assigned weights. This simple combination of classifiers has demonstrated to reduce the variance error term of the final classifier $F(x)$.

In Algorithm 1, we show the testing of the final decision function $F(x) = \sum^{M}_{1} c_f f_m(x)$ using the Discrete Adaboost algorithm with Decision Stump "weak classifier". Each Decision Stump f_m fits a threshold T_m and a polarity P_m over the selected m-th feature. In testing time, x^m corresponds to the value of the feature selected by $f_m(x)$ on a test sample x. Note that c_m value is subtracted from $F(x)$ if the hypothesis $f_m(x)$ is not satisfied on the test sample. Otherwise, positive values of c_m are accumulated. Finally decision on x is obtained by sign$(F(x))$.

```
1: Given a test sample $x$
2: $F(x) = 0$
3: Repeat for $m = 1,2,..,M$:
   (a) $F(x) = F(x) + c_m(P_m \cdot x^m < P_m \cdot T_m)$;
4: Output sign$(F(x))$
```

Algorithm 1: Discrete Adaboost testing algorithm.

This technique applied to each of the points of a 3D world requires a huge computational time, and it would be convenient to calculate operations as fast as possible to provide a responsive software for the user to interact with it in the way we described.

2.1.3. General-purpose computing on graphics processing units (GPGPU)

Most computer architectures are based in the Von Neumann computer model:

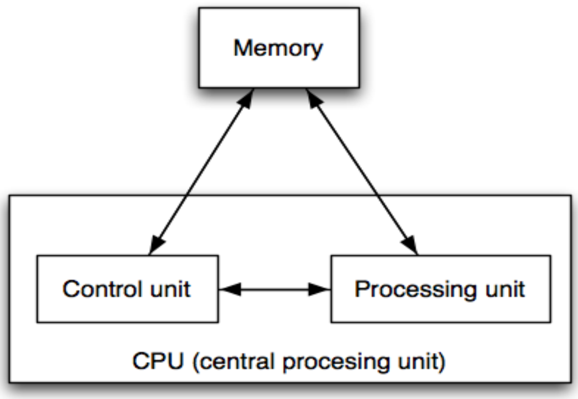


Figure 2.1: Von Neumann architecture representation.

This model was not designed for parallelism. Nevertheless, there are some ways to achieve parallelism based in Von Neumann design. These are mainly shared memory systems, where several Von Neumann processors share a common physical memory, with private and common virtual memory spaces, and message passing systems where each processor or node has its own memory, and communication is done through an interconnect system using an specific protocol.

A third architectural model called Data Parallel, was a variation of the Von Neumann model, in order to have a natively parallel architecture. It was used for very specific computing tasks and was the predecessor of vector units in CPU's (SIMD Single Instruction Multiple Data), as well as GPU's (SIMT Single Instruction Multiple Thread or SPMD Single Program Multiple Data). It was based in a single control unit for a set of processing elements, in order to execute concurrently the same instruction for arrays of independent data.

GPU's used to be instead a fixed function pipeline, until they started to implement some algorithms in hardware for 3D visual effects that required some computation.

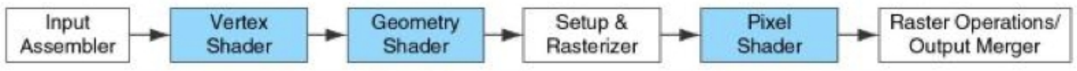


Figure 2.2: a tipical graphics card hardware pipeline [2]

This algorithms started to be more diverse and as a consequence, the designs of GPU's started to include more general-purpose capabilities in intermediate stages of the hardware pipeline.

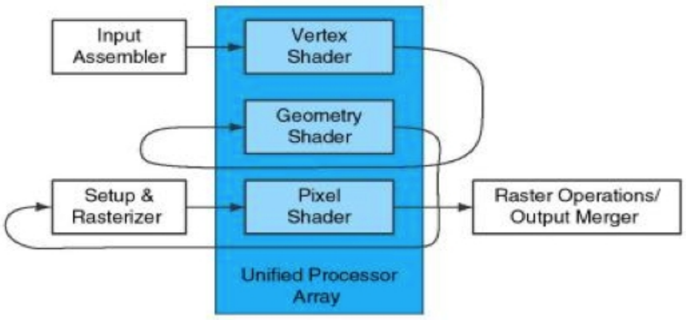


Figure 2.3: representation of the hardware stages computed by the Unified Processor Array [2]

Today, is possible to use GPU's only for it's more general-purpose capabilities, but these are not as general as the CPU.

Comparing to the Von Neumann model, a GPGPU hardware scheme would look like Figure 2.4.

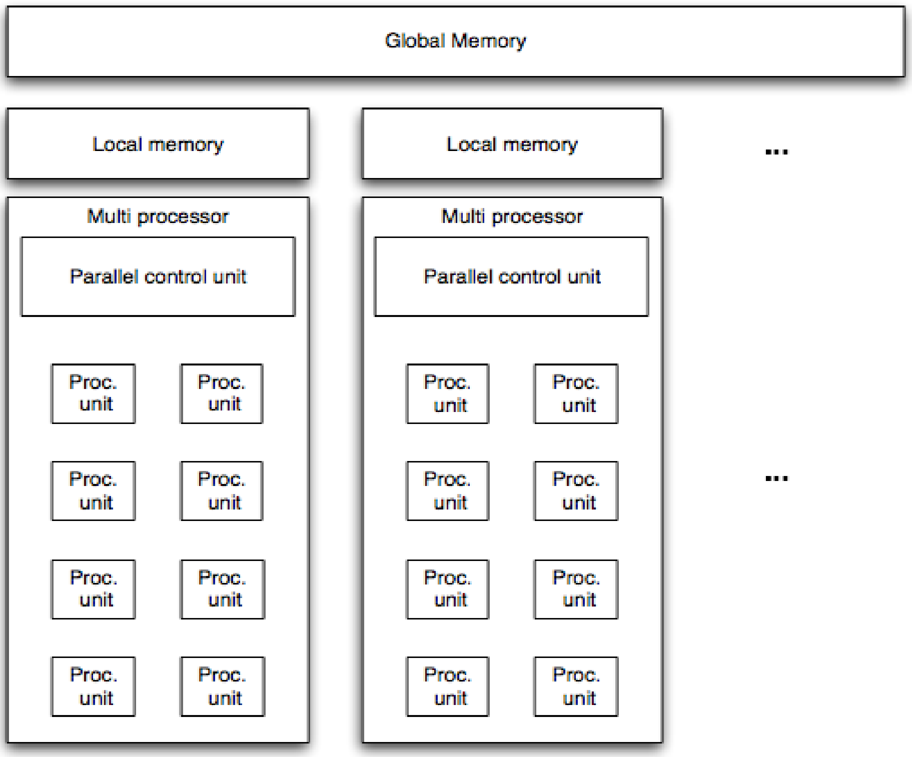


Figure 2.4: block diagram of the general purpose hardware of a GPU.

Although CPU's have a more complex memory hierarchy than the simple Von Neumann scheme by using several cache levels, these levels are transparent to the programmer since they have an automatic hardware driven actualization algorithm. At most, we can differentiate between values that reside on the main memory and values that probably reside on the register file.

Instead, GPU's are based in a specific memory hierarchy, where the programmer has almost perfect control over the location of the data. Also, there is a single control unit that manages several processing units for an in-hardware parallelism control. This block is called a multiprocessor and corresponds to the Unified Processor Array in Figure 2.3. The local memory allows for a multiprocessor to work with a private copy of data, that was on the global memory, so the transfers between a multiprocessor and global memory are controlled by the programmer in contrast with CPU's caches. That allows to pack several multiprocessors in communication with the same global memory and in a single chip without increasing exponentially the chip design complexity to scale in performance. Also, global memory is constructed in a way that each multiprocessor can have the responsibility to access a memory block through dedicated lanes. This way, the more multiprocessors, the more memory chips, and more memory lanes, so more memory bandwidth. This increases the hardware scalability, but also relies on software exploiting the aggregate bandwidth.

In general, a GPU used as a parallel computational unit is considered an accelerator since it has no context switching capabilities and others that help a CPU to handle efficiently for instance an operating system. In fact, today it does not exist an Operating System that can be run in a GPU.

There are several languages to program a GPU. The first one only used for computational purposes was OpenGL. It is not designed for general-purpose computation but for graphical programming. It was complex to program and gave no control over all the GPU computing hardware and memory hierarchy.

To solve that NVIDIA developed CUDA C (Compute Unified Device Architecture) and ATI CAL (Compute Abstraction Layer). Later Apple started a project to create an standard language and API to program not only any GPGPU but any hardware available on the system. For that reason it includes a low level abstraction to allow control over system communication and memory hierarchies for any kind of device. This makes programming OpenCL a bit complex.

Despite the fact that OpenCL can be more complex and slow to write than CUDA in the Host code part, and the performance shown in our own analysis is almost the same as CUDA, we chose OpenCL because of the portability.

OpenCL provides an abstraction to differentiate between the code that is executed on the processor that actually executes the operating system, and the code that is executed on any other hardware, including the first. The first code is referred as the Host code by the OpenCL specification, and is written in ANSI C with specific libraries and types. The second one is referred as the Device code or Kernel, and it is written in OpenCL C, that is a subset of ANSI C with some restrictions and native specific function calls and types.

Due to the fact that OpenCL wants a kernel to be able to target any hardware that implements any parallel paradigm or memory hierarchy etc, there are different types of kernels. Doing a kernel that suits every operation would limit programmer control over hardware. For this same reason, it is also possible to define a custom type of kernel or to use precompiled binaries. Some of those custom kernels can end being native OpenCL kernels in successive OpenCL versions.

For the GPU, OpenCL provides the NDRange kernel, which mimics the NVIDIA CUDA programming model, and works for NVIDIA GPU’s, ATI GPU’s and it is suitable for processors like the Cell BE. The NDRange kernel provides a mapping for the large amount

of threads executing on multiple cores of massively parallel processors like GPU's. This mapping starts with a two dimensional space called NDRange where all the threads reside. These threads are called Work-Items and further organized in groups called Work-Groups, that can be 1, 2 o 3 dimensional. So, at maximum, we can have "cube" groups of threads inside the NDRange "plane".
Due to hardware restrictions common or similar in NVIDIA and ATI cards, this organization has some restrictions or rules:

- To use 3D Work-Groups, declaring a 3D NDRange is needed, even the 3rd dimension of the NDRange is limited to size=1. So dimensions declared for the NDRange are propagated to the Work-Groups, but not the sizes for each dimension.
- Dimension size in an NDRange accounts for the number of Work-Items, not Work-Groups, in that dimension.
- Dimension size in a Work-Group accounts for the number of Work-Items in that dimension, inside the Work-Group.
- There is a proportionality to be kept between the dimension sizes of the NDRange and the dimension sizes of the Work-Groups. The size of an NDRange dimension has to be evenly divisible by the size of the Work-Groups in the same dimension.
- The maximum number of Work-Items in a Work-Group is 512 for NVIDIA pre Fermi devices. It is variable between vendors and models so it is possible to query this restriction to automatically adapt the NDRange and Work-Item sizes for each device.
- NVIDIA differentiates the different hardware characteristics and limitations in what they call compute capabilities.

2.2 Classification methods into GPU

A proposal for this year aims to parallelize Adaboost algorithm[3] using an european infrastructure. Others have studied an OpenCL implementation for packet classification for networking[4]. Also some machine learning work is done in GPU[5].

Other methods like neural networks have more and successful history in GPU computing, even mixing GPU and Multicore CPU computation[6].

2.3 Conclusions

In general, GPGPU involves an scalable parallel hardware design and a highly scalable parallel programming model.

In standard non parallel driven programming models like ANSI C, is very difficult to scale up to 80 CPU cores. That means that the software is not increasing performance by using more than that amount of cores. Using specifically parallel programming languages like OpenMP and MPI, scaling highly depends on the platform characteristics and the algorithm implementation, that also depends on the language and system used.

Instead GPGPU with languages like OpenCL only requires to adapt the algorithm to data parallelism, and it will automatically scale to new generations of GPU hardware without any change on the code. Data parallelism means executing the same instruction or program for multiple data, at the same time. We are talking about scaling to hundreds of processing units, generating hundreds of thousands or millions of threads.

In this project we have adapted the Adaboost algorithm to be highly parallel in first C++

implementation, and then substituted almost all for loops with GPU inherent indexing, increasing this way parallelism.

For 512x512x512 sized Voxel Models, we can generate 62.914.560 threads. So we have several years of GPU generations to scale up in performance without changing the code. For this reason, and the performance boost we can achieve now, we chose GPGPU to increase performance.

Because of the amount of documentation and technical articles based on NVIDIA, we decided to base this implementation on NVIDIA hardware. Because of our hardware availability we based the design for compute capabilities from 1.0 to 1.1. At the end of the project we had available a 2.0 compute capability device from NVIDIA. We will explore performance tweaks for that device as a future work.

3 Proposal

Now I will explain all the process, starting with adapting the Adaboost algorithm to a highly parallel form and then design an OpenCL implementation.

Before the classification algorithm there is a process that generates data from each point in the Voxel Model, a well known gradient calculation algorithm [7]. It has some implications for overall performance, and I will explain the global design of the OpenCL implementation based on that. Then we detail the design decisions and arguments and finally we list well known optimization techniques for NVIDIA GPU's and comment if we implemented them or not in our code and why.

3.1 Algorithm analysis for GPGPU:

They propose to define a new and equivalent representation of c_m and $|x|$ that facilitate the paralellization of the testing. We define the matrix $V_{fm}(x)$ of size $3 \times (|x| \cdot M)$, where $|x|$ corresponds to the dimensionality of the feature space. First row of $V_{fm}(x)$ codifies the values c_m for the corresponding features that have been considered during training. In this sense, each position i of the first row of $V_{fm}(x)$ contains the value c_m for the feature $\text{mod}(i,|x|)$ if $\text{mod}(i,|x|) \neq 0$ or $|x|$, otherwise. The next value of c_m for that feature is found in position $i+|x|$. The positions corresponding to features not considered during training are set to zero. The second and third rows of $V_{fm}(x)$ for column i contains the values of P_m and T_m for the corresponding Decision Stump.

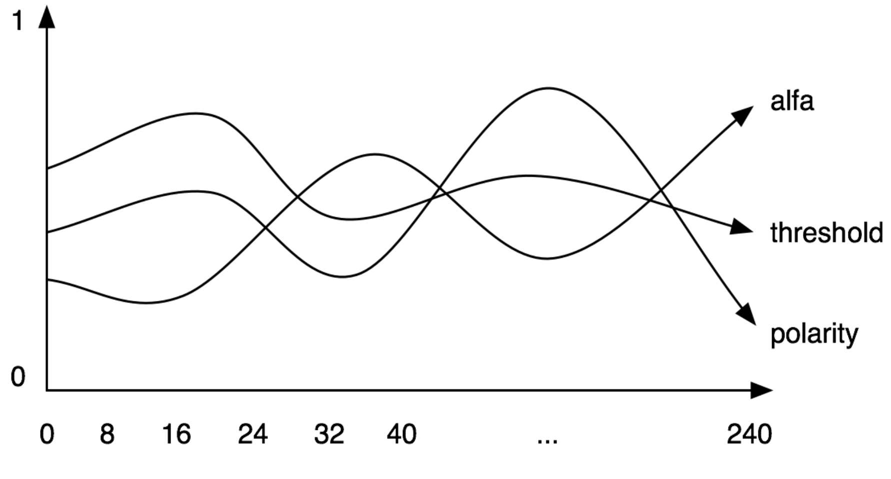


Figure 3.1: Representation of alfa, threshold and polarity for each one of the 240 values.

Note that in the representation of $V_{fm}(x)$ we loss the information of the order in which the Decision Stumps were fitted during the training step. However, though in different order, all trained "weak classifiers" are codified, and thus, the final additive decision model $F(x)$ is equivalent.

3.2 Adaptation to OpenCL:

My proposed OpenCL's implementation is based primarily on the GPU-PCIe accelerator concept, taking into account the PCIe bottleneck and the GPU's main memory or Global Memory. As we deal with large datasets of voxels, we use out-of-core techniques to subdivide the initial dataset into subsets which can fit into GPU main memory. Thus, PCIe bandwidth becomes an essential factor in the overall execution.

I decided to do a first implementation where optimizing PCIe usage implies more complexity in the kernel and even a previous kernel whose task will be explained in the next section. Also I decided to focus on one of the common recommendations in this first implementation, that is generating as much threads as possible in order to scale as mentioned in previous sections.

That means that this implementation has a very good usage of PCIe, a huge parallelism, few slow steps in order to have a good use of local memory, but a very bad use of global memory. This is because the difficult part was to improve the PCIe but once done and validated, it will be quite easy to improve global memory usage.

I overview my proposed OpenCL implementation in Figures 3.3 and 3.7. The eight features considered at each sample by the binary classifier are: the spatial location (x, y, z), the sampled value (v), and its associated gradient value $(gx, gy, gz, |g|)$. The binary classifier, for each feature, has $N=3 \star |\textit{number of weights}|$.

I create a matrix of Work-Groups that covers the x and y size of the dataset fitted into GPU global memory, whereas the component z is computed in a inner loop in the kernel. Each WorkGroup classifies one voxel. Inside each WorkGroup, we define $N \star 8$ threads, or WorkItems. Each thread computes a single operation with the 3 channels or weights of the weak classifier. The resulting $N \star 8$ values will be reduced at the end of the execution and compared to a reduced addition. The final label for each voxel is directly computed by this comparison.

3.3 Gradient calculation:

The computation of dataset gradients is an essential operation in many visualization techniques. Visualizing a given three dimensional dataset can be done by surface rendering algorithms, such as the Marching Cubes Algorithm [8], or by direct volume rendering algorithms, raycasting [9] or splatting [10]. For direct volume rendering methods the voxel intensity, gradient direction and magnitude are often used to shade and classify the dataset. For surface rendering techniques the gradient is used as an estimation of the surface normal which is used for shading.

Gradient operators are also often used during the classification of data as well. The classification procedure provides an optical density value for each voxel in the dataset, called opacity. Opacities are typically calculated using either voxel intensities or a combination of voxel intensities and gradient information. We use voxel intensities and gradient information in the Adaboost classification algorithm.

In the GSGL implementation, 8 float values for each Voxel where send to the graphics cards. With

OpenCL instead, is quite easy to do a gradient calculation, so we could send only the intensities, and launch a kernel that calculates the gradients.

This gradient calculation follows the shape of a 7 point stencil [11], but with a little modification in the front and behind values.

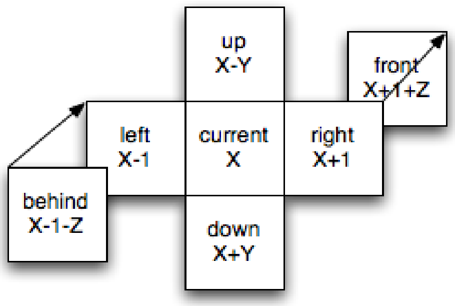


Figure 3.2: representation of the values used for gradient calculation for a current position

As shown in Figure 3.2, the front and behind values are displaced from a normal stencil configuration. For the current voxel we calculate a gradient value for each of the dimensions. It is *(left-right)/2* for x dimension, *(up-down)/2* for y dimension and *(behind-infront)/2* for the z dimension. This way we obtain what we note as gx,gy and gz in the code.

3.4 Global architecture:

As seen in every GPGPU implementation in the literature, there are always parts of a program more suitable for GPGPU computing than others. In our case we wanted to improve the voxel classification algorithm code performance of the program, but as mentioned in the previous sections introducing a previous step into the GPU allows for a huge performance gain in PCIe transfers. This previous step is the gradient calculation.
As we need 8 values for each voxel, density, 3 position values, one for each dimension, 3 gradient values one for each dimension again and a factor calculated fith the gradients, there should be 8 float values to be passed for each Voxel. But we only pass an unsigned char that correspond to the Voxel density. The positions are requested to OpenCL through API calls, the gradients and the factor are calculated with the gradient calculation kernel.

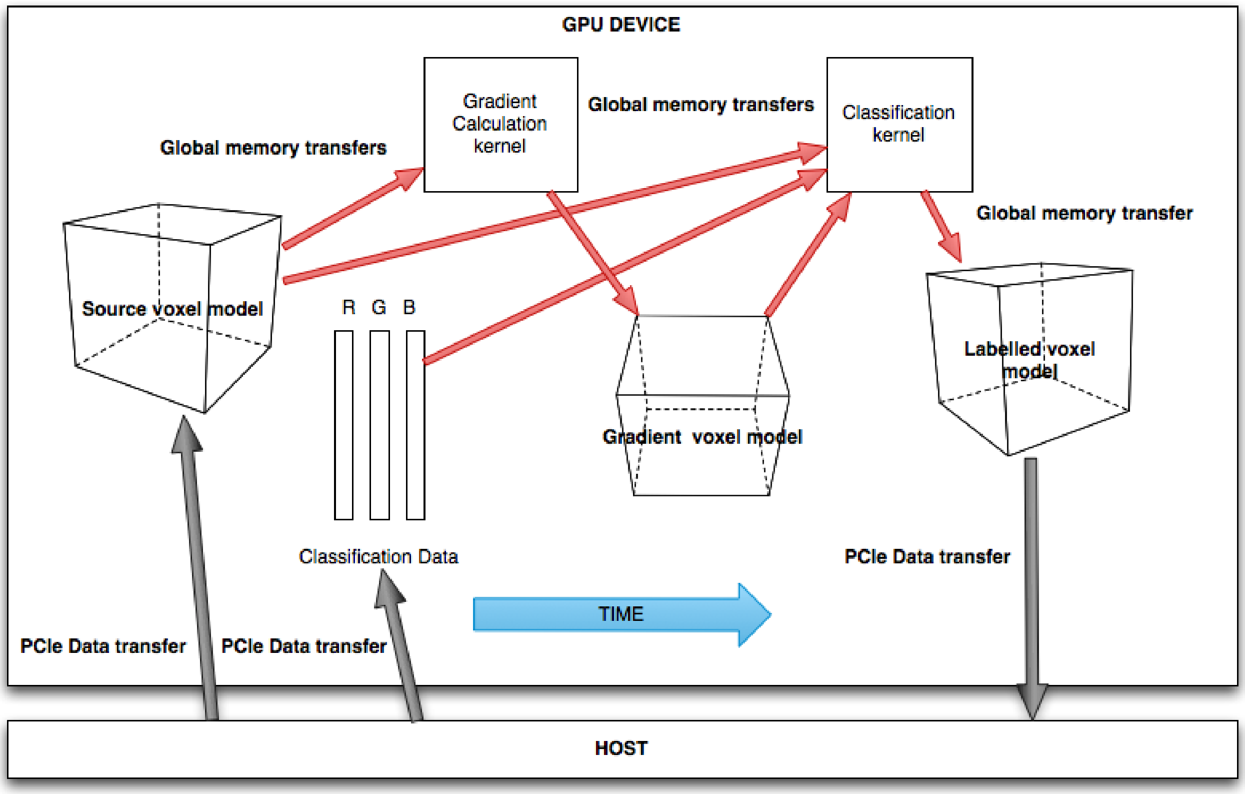


Figure 3.3: General program diagram

Figure 1 shows the main flow of the program and the data. A 3D unsigned char voxel data model is loaded into the graphics card main memory, as well as the data used to classify them.
A gradient calculation kernel is loaded into de device and executed to produce neighbor aware data for each voxel. This gradient data is not loaded back to the host.
Another kernel, the classification kernel, is loaded and executed then, with the gradient data produced by the previous kernel as a parameter.
This last kernel produces a labeled model of unsigned char values that are either 1 or 0.

The GPU's, a part from a graphic's processor, are PCIe based accelerators, with dedicated main memory, since now and attending to OpenCL specification, global memory. This mean s that we have to take in account the bandwidth of the PCIe connection, and the way that the GPU accesses it's dedicated global memory, as long as read/write bandwidth is concerned.
Looking at the GPU architecture, this is conditioning the inner workings of GPU and therefore the structure of the programs when extreme performance is the goal.

3.5 Design decisions:

Following the conditions mentioned in section 3.4, and based on the NVIDIA 1.0 and 1.1 compute capabilities for both gradient and classifier kernels, we used 32x8 Work-Groups that the scheduler group in 8 warps of 32 Work-Items for compute capabilities 1.x and 32x4 for compute capability 2.0 following what we found in the OpenCL analysis in the TAD. These warps are executed in two steps, executing in each step what is called a half-warp of 16 Work-Items. In compute capabilities 2.0 two half warps will be executed at the same time in each SM (Stream Multiprocessor or Compute Unit in OpenCL terms).

For the **gradient calculation kernel**, 16 Work-Items read 16 unsigned char values in one transaction. That means that only 16 bytes will be read, while 64byte segments could be read in one transaction. Moreover, as each unsigned char is 1 byte long, for compute capabilities 1.0 and 1.1 this operation will serialize into 16 memory transactions. For compute capabilities (CC) 1.2 and greater, the reads are coalesced in one memory transaction for 16 consecutive bytes.

That results in a 1/64th of the maximum global memory bandwidth, since maximum read bandwidth corresponds to 64byte reads and we are reading 1byte per memory transfer. Anyway, using integer or float types will mean copying 4 times the amount of data from host to the device, that would be the same as reducing the PCIe bandwidth by 1/4th. Comparing that numbers, and taking in account bandwidths for the same number of words to be transferred, but different word sizes, we found that for NVIDIA graphics cards from 8000M series up to some Geforce 9600M versions it would be faster to use float instead of unsigned char values. But for the rest, unsigned char is better, and more for compute capabilities 1.2 and above.

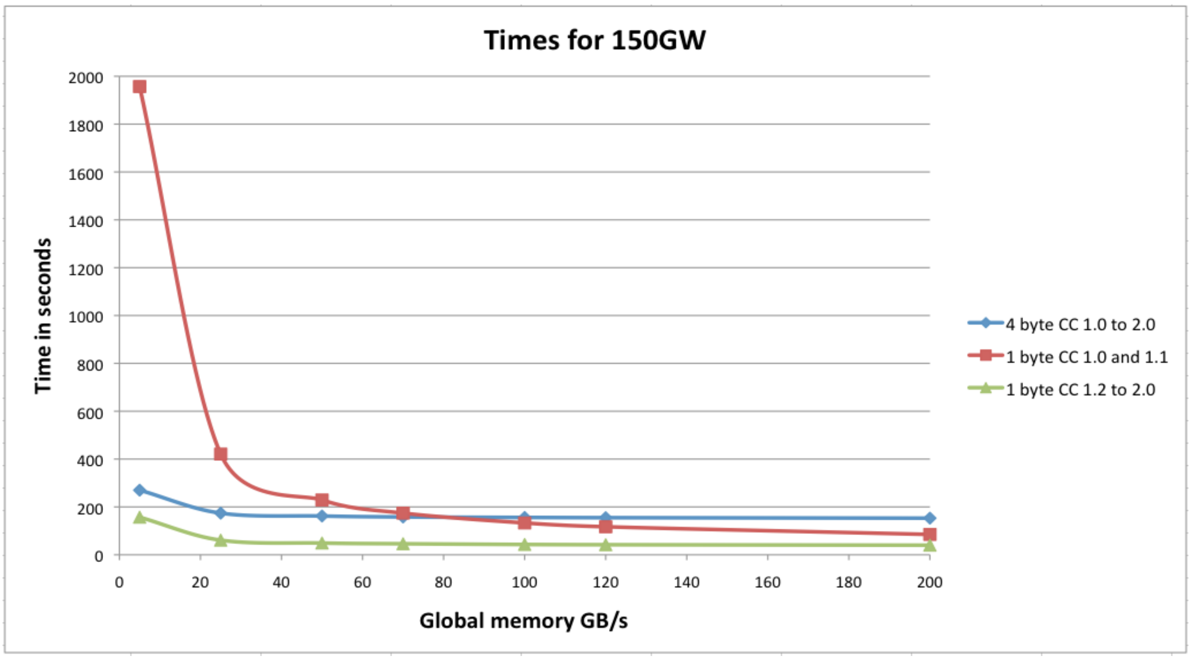


Figure 3.4: Comparing PCIe + global memory theoretical read times with uchar or float types. X axis represents global memory bandwidth in GB/s. Y axis represents time in seconds needed to copy 150GWords from Host to Device and from Global memory to the next memory hierarchy level in gradient kernel.

The difference decreases with better global memory bandwidths that vary a lot from card to card, while PCIe 2.0 bandwidth will never change. PCIe 3.0 will decrease the difference, but by then global memory speeds will provably be much better.

Figure 3.4 shows the theoretical read times for the same number of words to be copied. To calculate that I had to take in account that different word sizes implied different behaviors in global memory. To make the calculations more clear I used the number of words in GigaWord sizes instead of GigaBytes. Using float words, *1GW = 4GB*, and for unsigned char it is *1GW = 1GB*. This is a conversion factor so I note it as C_f.

In all cases, the PCIe bandwidth E_B is 4GB/s, and the global memory bandwidth G_B is variable.

The behavior change in global memory is expressed by B_f, where it is neutral for float words and 1/64 for unsigned char in CC 1.0 to 1.1 and 1/4 for unsigned char in CC 1.2 and above.

In the case of using floats I defined the time necessary to transfer 150GW from Host to Device as T_{hd} where $T_{hd} = 150GW / (E_B.(1GW/4GB))$. It is always 150 seconds. The time to read from global memory $150GW$ is $T_{gm} = 150GW / (G_B.(1GW/4GB))$. The total time is T_t.

Time formulas:

$T_{hd} = 150GW / (E_B.C_f)$
$T_{gm} = 150GW / ((G_B.B_f).C_f)$
$T_t = T_{hd} + T_{gm}$

Conversion factors:

$C_f = 1GW/4GB$ (float)
$C_f = 1GW/1GB$ (unsigned char)

Behavior change factor:

$B_f = 1$ (float)
$B_f = 1/4$ (unsigned char CC 1.2 to 2.0)
$B_f = 1/64$ (unsigned char CC 1.0 to 1.1)

Figure 3.5: time formulas for Figure 3.4.

To reduce global memory reads in **gradient kernel** I did a variation on the Micikevicius stencil for GPU's algorithm [12]. I used registers for behind and front values, and local memory for up, down, left and right values. Current value is not needed for gradient calculation. As seen in Figure 3.6 there is a displacement for behind and front values that translates in a change of shape for local memory plane comparing with Micikevicius algorithm.

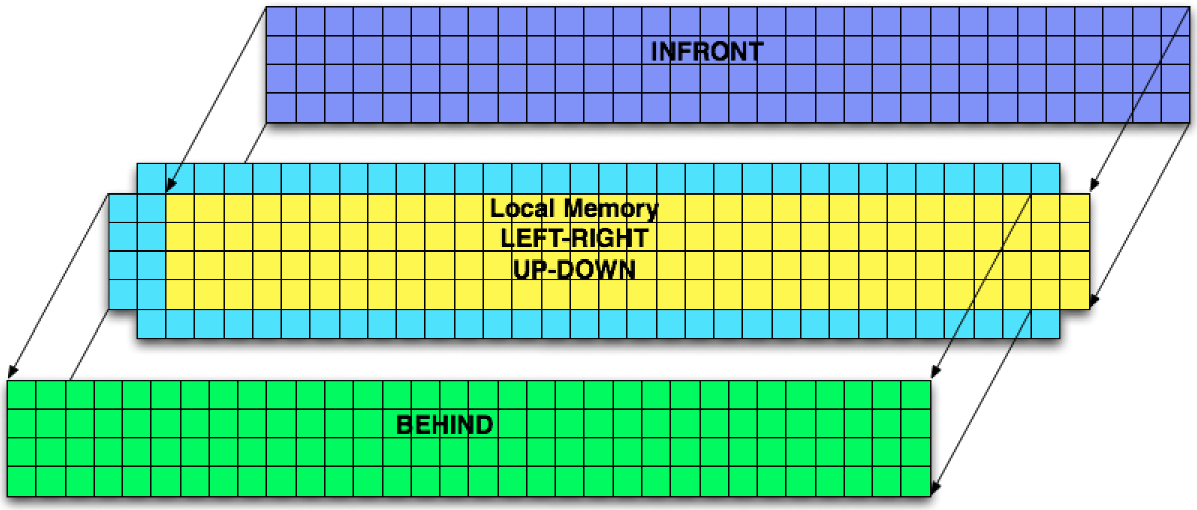
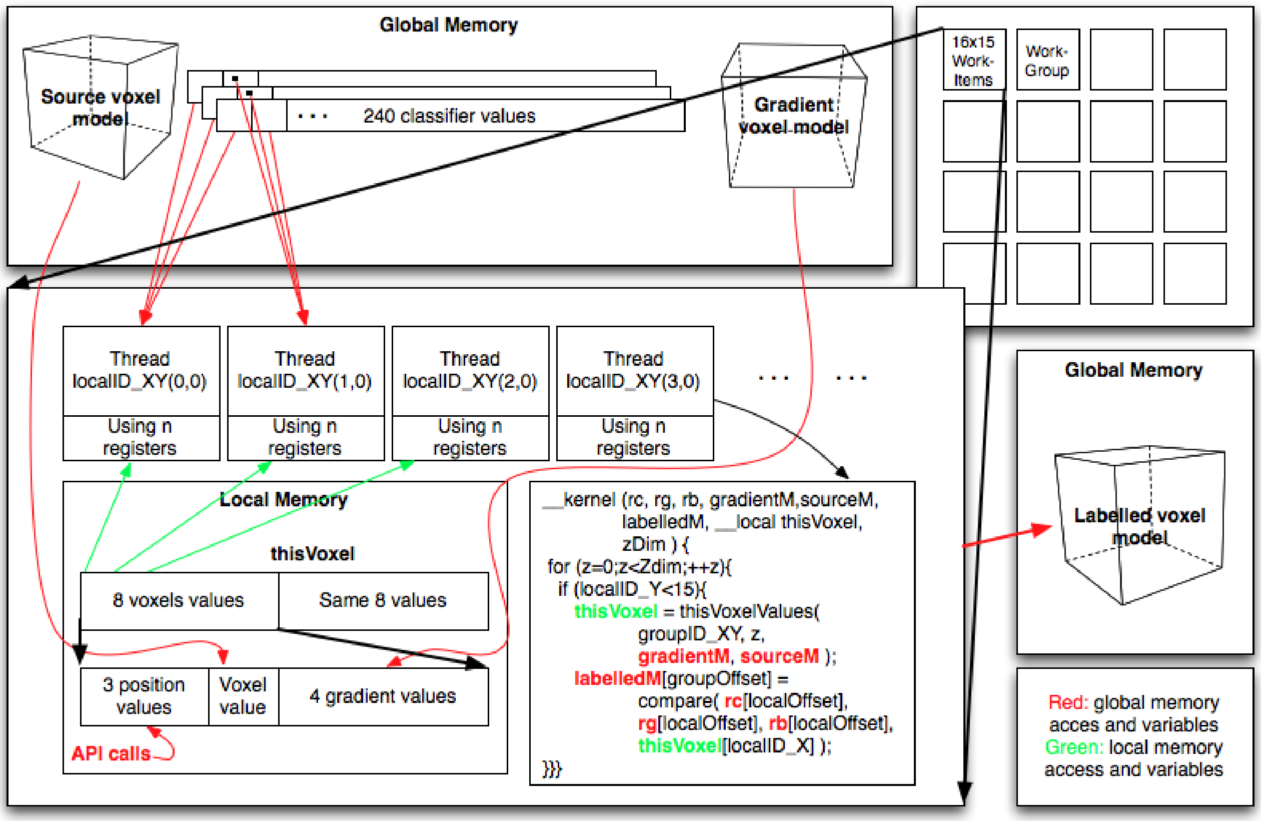


Figure 3.6: representation of the values in registers and local memory for gradient kernel. In yellow values read from infront registers. In blue values read from global memory to local and registers. In green values read from local memory to behind registers.

Finally, each Work-Item has 4 float elements to write into the global memory for the next kernel.

Once the gradient kernel ends, the **classifier kernel** is launched, reusing the same device pointer where the gradients reside to avoid copying to device, as shown in Figure 3.3.

This kernel tries to exploit parallelism as much as possible. The C version of this code has been tested with OpenMP only parallelizing the outer loop of two nested loops. Due to the massively parallel capabilities of graphics cards, we implemented a fully concurrent version that can scale perfectly with more cores, expected in each new GPU architecture version, so performance is expected to increase proportionately without code changes.



```
__kernel (rc, rg, rb, gradientM,sourceM,
          labelledM, __local thisVoxel,
          zDim ) {
 for (z=0;z<Zdim;++z){
  if (localID_Y<15){
    thisVoxel = thisVoxelValues(
          groupID_XY, z,
          gradientM, sourceM );
    labelledM[groupOffset] =
          compare( rc[localOffset],
          rg[localOffset], rb[localOffset],
          thisVoxel[localID_X] );
}}}
```

Figure 3.7: classification kernel diagram

Each voxel to be processed represents a Work-Group with enough threads to process concurrently all the tasks to classify this voxel. Specifically, we use 240 threads for each voxel that do the testing for each RGB channel. We have 240 data elements for each channel and 8 data elements from the voxel to compare with. 4 of that last ones are the gradient calculation kernel output.
This 240 element arrays are thirty groups of 8 elements whose proprieties correspond to each voxel 8-element array, from first to last.

The comparison reads the first element of the first 8-element group for each RGB channel, and the first element of the 8-element group of the voxel values. Then, the second element for the same 8-element groups is read, until the 8th element. Then, the first element of the voxel and the 9th of each RGB array that corresponds to the 1st element of the second 8-element subgroup are read, etc.

To be able to use the thread indexes to access the voxel data elements, we use a 16 float array, as shown in figure 3, to store the 8 voxel elements twice. So thread indexes (0,0), (8,0) and (0,5) will read the same value.

Once all is computed, we obtain two values for each of the 240 threads. Now we have to do vector reduction for each Work-Group and each data set. We use the widely described method for NVIDIA graphics cards, to avoid local memory bank conflicts. A final operation determines a single value using the two resulting from reduction and we end with a single uchar value for each work group.

For this reason we are not measuring the write times in Figure 3.7, since it wouldn't make a difference to write a float or a unsigned char value per Work-Group.

As we mentioned before, this kernel design is focused on exploiting parallelism as much as possible, but for compute capabilities 1.2 and above, it could be better to change this kernel to exploit CC 1.2 8-byte word coalescing global memory capabilities, by using each thread to classify one voxel or doing the same operation for several voxels, allowing then 16 unsigned char or 16 uchar4 coalesced reads inside each Work-Group, and increase global memory bandwidth up to 64x. Increasing bandwidth, at the end means increase the number of processor cycles with computation being done. It reduces the number of global memory transfers, and for each transaction removed we earn 600 computing cycles, so we expect huge speedups in the next iteration of the code.

In GPU programming there's always a balance between exploiting parallelism and execution throughput. What will be faster depends entirely on the characteristics of the hardware and on the application.

3.6 Optimizations for the NVIDIA architecture:

Now we describe most of the optimization techniques described for NVIDIA GPU's but can be also valid for ATI GPU's. I also comment if I applied or not the technique.

3.6.1 Global memory use:

Global memory in GPU's is like the main memory for a CPU. From the GPU code point of view it is the source of every single data element, so it is mandatory for a program to use global memory. It is also the biggest memory in terms of capacity, but also the slowest.

While reading any CUDA or OpenCL programming guide, the first recommendation in order to obtain performance is to read global memory in a coalesced way. Coalescence is produced when 16 4bit words are read in a single transaction. The conditions for that to happen vary from CC to CC decreasing the restrictions in each new CC, but overall it is required to read contiguous data from a single memory block. A memory block is a sub organization of global memory, where each element of a memory block can be accessed in the same memory transaction. Accessing data elements from different memory blocks, produces as many memory transactions as different memory blocks are accessed.

More information can be found in the NVIDIA Best practices guide [13] or the ATI Stream SDK OpenCL programming guide [14].

Gradient kernel uses global memory as few times as possible, but it does it moderately well. It uses 1/4th of the theoretical maximum bandwidth for compute capabilities 1.2 and above. But it only uses 1/64th in CC 1.0 and 1.1. As mentioned before, it will be easily solved in the next iteration of the program.

When writing to global memory, it uses local memory in order to have data in a way, that can be read and stored in consecutive chunks of 16 elements, so in a coalesced way, from local memory to global memory. This way I am saving a lot of computational cycles.

Classification kernel uses global memory also only one time per data, but when it does it, it does the worst use possible. It uses 1/64th of the maximum global memory bandwidth for any compute capability, by reading only one unsigned char per work group. Reading gradient values is also bad, it reads only 4 floats in each transaction. This will be also addressed in the next iteration of the software.

3.6.2 Local memory use:

The two most important GPU architectures share a common memory hierarchy. The properties, sizes and behaviors vary a little from ATI to NVIDA though.
In general, there are two main kinds of memories. Global-Memory that is big, slow and accessible to any thread in the kernel and Local-Memory that is small, fast and resides inside each multiprocessor or compute unit of the GPU, so each value stored by a thread in Local-Memory will only be accessible by the threads of the same Work-Group.
The difference in speed is really huge. According to NVIDIA documentation, accessing Global-Memory equals to wasting from 400 to 600 multiprocessor execution cycles. Accessing Local-Memory needs only 2 cycles. That explains too the importance of Global-Memory coalescing. Performing one coalesced transfer means loosing 600 cycles, dividing this transfer into 16 transfers means 600*16 lost cycles.
In our implementation we use Local-Memory in the following parts.

Gradient kernel:

- As each unsigned char value from source voxel model is read a minimum of 1 times and a maximum of 4 times in a non coalesced way, performing one coalesced read for each Work-Group to Local-Memory allows to compute reading from local memory and effectively use hundreds of processing cycles.
  Moreover using float types, we avoid Local-Memory bank conflicts. Reading 15 values from contiguous banks and one from another set of banks does not produce 16 transfers like in 1.0-1.1 Global-Memories, it produces two transfers and so 2x2 lost processing cycles instead of 600*16 cycles per half warp.
- We use Local-Memory again before writing to memory, as explained in section 3.6.1.

Classification kernel:

- In this case, we want to access 8 values using the local id's of each thread or Work Item. So we built a variable called thisVoxel, shown in Figure 3.7.

  As this values will be accessed 30 times, using Local-Memory we are increasing notably performance, and even more achieving Local-Memory coalescence by copying twice the values to achieve a 16 float long variable, that will be read by 16 threads in a single Local-Memory transaction.

  I have not used Local-Memory for the classification data arrays but in next iterations it will be useful to do it, because they are not only read once per plane in the z dimension but also each Work-Group will classify more than one Voxel, and then will read the classification arrays much more times.

3.6.3 Local memory bank conflict behavior:

As explained before, there are some differences on bank conflict behaviors and coalescence between compute capabilities in Global-Memory and between Global-Memory and Local-

Memory.

Local memory in NVIDIA cards is disposed in groups of 16 4byte words. This groups are called local memory banks. There are several groups of banks, packed in a way that only one bank can be accessed at the same time. So accesses produced by a single half warp in different banks are serialized in a way that there will be as many transactions as different groups accessed. When using words smaller than 4 bytes and two threads read two different words from the same bank, there is a bank conflict too, and there is a local memory transfer for each word.

In the case of **gradient kernel**, we store each Work-Item result (4 floats) in a Local-Memory variable, in a way that each Work-Item stores one of the values in each transaction 4 positions away from the left neighboring Work-Item. So a half warp performs 8 memory transactions. We have 16 half-warps in a Work-Group so 128 transactions will be needed for the first value of each thread. We have to store 4 values per WI, so that means 128*4 transactions per Work-Group. That is wasting 128*4*2 cycles, instead of 128*4*600 cycles if we store directly to Global-Memory. Having the data stored in local memory we can do coalesced writes in chunks of 16 floats into Global-Memory that means 16*4*600 lost cycles. So we achieve 128*4*2 + 16*4*600 instead of 128*4*600 lost cycles, 87,5% less.

In the case of **classification kernel**, the bank conflicts are a problem when performing matrix reduction.
Focusing only in Work-Items executing in chunks of 16, we could sum to the first 16 values the other 14 16-value groups in 14 steps, obtaining one vector for each 16x15 matrix. This first part is perfectly coalesced and bank-conflict free. Then, it could be possible to copy the two arrays in a single 32-value one in order to use a half-warp for all of them in a sum-pairs fashion, until we obtain a value in position 0, and another one in position 16.
But here comes the bank-conflict. When 16 threads try to sum 32 neighboring values in pairs in Local-Memory, 8 WI will be reading values of the same 64bit segment, and the next 8 WI will do the same in another Local-Memory segment. So we will need twice the number of Local-Memory transactions.
Instead of that, using the documented NVIDIA reduction version [15], we have a simple code resolving all the issues in a for loop.

3.6.4 Register count and execution scheduling:

As I commented before, the Work-Items in a Work-Group are scheduled in packs of 32 Work-Items. This structure is called a **warp**. When a warp is going to be executed, it is divided into two **half warps**, and each half warp is executed concurrently, if possible.
From this point we can take in account two concepts to tune performance:

- *Occupancy*: the number of active warps per multiprocessor or compute unit. An active warp means 32 Work-Items able to perform their execution. Inactive warps are those waiting for synchronization, memory reads etc.
- *The number of registers being used per Work-Group*.

Both concepts are tightly related, since altering the number of registers used, can alter occupancy in one or another way.

3.6.4.1 Occupancy tuning:

Achieving more occupancy means reducing the provabilities of waiting for data to be available while executing, by increasing the number of active warps available to the compute unit.

This is because each compute unit can have instantiated on registers more than one Work-Group. The compute unit then takes any set of instructions (warps) belonging to any of the Work-Groups in any order, although often sequentially for the warps of the same Work-Group. When an instruction needs to load data from Local or Global-Memory, the NVIDIA compute unit can hide the load latency by fetching another active warp from any of the Work-Groups with, while loading the data for the first one at the same time.

Sequentiality is more prone to happen within the same Work-Group, for instance, with barrier synchronization. That means that several warps will be inactive until some others finish execution inside the same Work-Group. Having only one Work-Group potentially decreases occupancy and thus decreases latency hiding. NVIDIA provides an excel sheet that helps calculating occupancy.

Using a lot of Local-Memory or a lot of registers can produce instantiating only one Work-Group in each compute unit. Sometimes it is impossible to avoid that without losing performance in Global-Memory reads, but there are techniques to reduce latency an improve performance in these cases.
That is the case for our implementation. For gradient kernel we could fit at most two Work-Groups so we focused on the techniques described in the next section. In contrast, classification kernel can have up to 7 Work-Groups in a single compute unit.

3.6.4.2 Register related techniques:

An NVIDIA compute unit, for 1.0 and 1.1 compute capabilities, has 8192 registers.
In gradient kernel we end up with a maximum of two Work-Groups per compute unit. We found that it is better to focus on prefetching and other techniques to reduce latency and take profit of the amount of registers available.
The techniques we considered are the following:

- *Prefetching*: we ensure register level data availability when an operation is performed reading Local or Global-Memories, by reading first the data into a single non pointer variable, that automatically resides on register.
- *Unrolling*: since there are thousands of registers free, we can transform a loop in to a sequence of registers without branching instructions hence reducing the number of instructions to execute, but increasing the number of instructions residing on registers.

Another approach that we will test is to spill the registers used for indexing into a Local-Memory variable or use directly the API calls, even reducing code clarity, to improve occupancy as we observed in the CUDA GPU Occupancy Calculator for **gradient** and **classification** kernels. Then, we will check if there's any performance difference for various compute capabilities.

4 Results

In this section I will show the time results for a data set of 128x128x128 voxels that contain a 3D image of a foot.

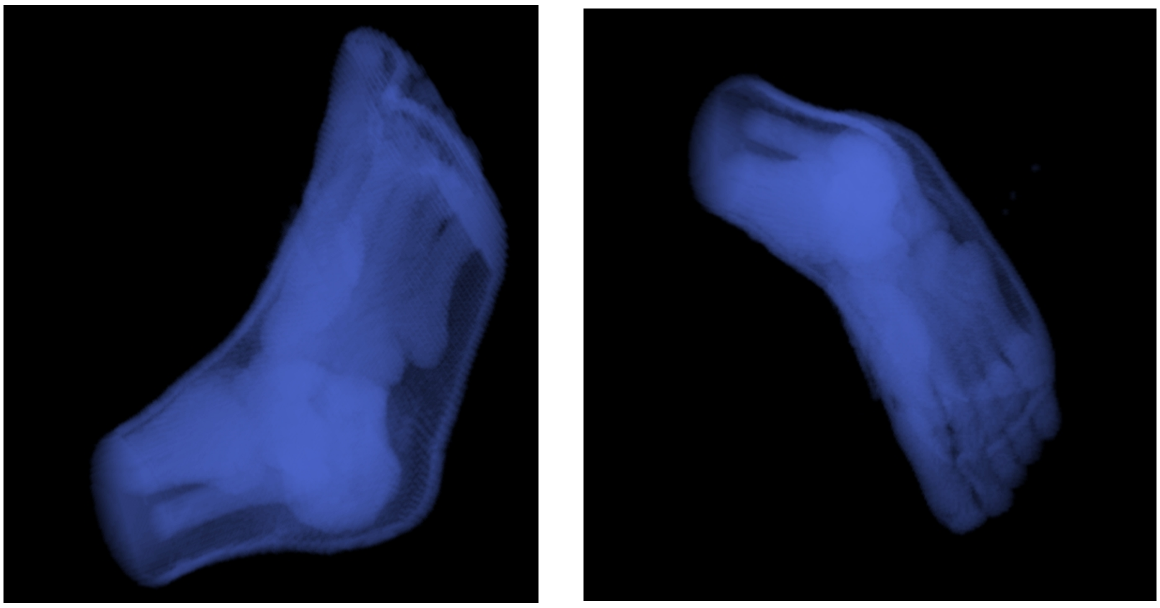


Figure 4.1: view of the classified data set representing a foot. Captured using Paraview [16].

The first results I will show compare only the classification times in CPU with C++, a dual core CPU with OpenMP, GSGL and my OpenCL implementation. For CPU tests we used the same Intel dual core processor and for GSGL and OpenCL the same Geforce 8800 GTX.

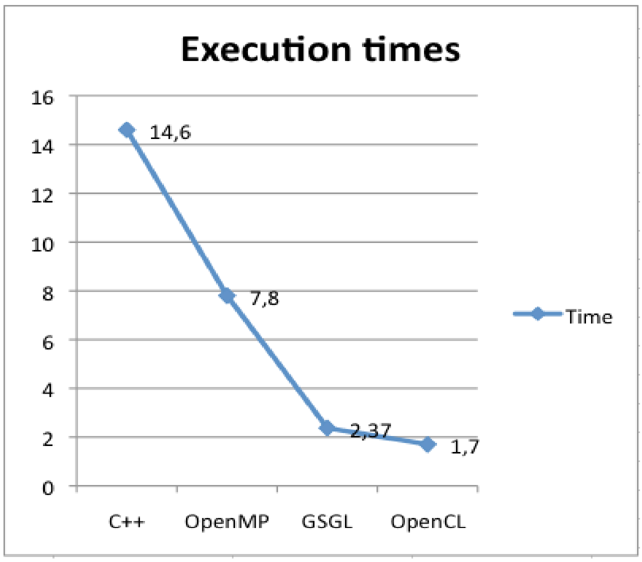


Figure 4.2: execution times for the different implementations of the classification algorithm.

Why this is a very good result? Remember that there is a pending performance improvement related to global memory that we expect to give a very big speedup over that version. Also, the 8800 GTX card is a CC 1.0 card, so there is a 1/64th factor of the maximum global memory bandwidth that is not applicable to cards of CC 1.2 to 2.0. We could expect that OpenCL version to be slower than the GSGL but the times are better. The exact speedups are the following:

8,5x of the C++ version
4,5x of the OpenMP version
1,4x of the GSGL version

This is without taking in account the Host-Device transfer times neither for the GSGL or the OpenCL. The OpenCL transfer times are 1/32th of the GSGL version, as it transfers 8 4byte values for each voxel.

Now I show a comparison of the execution times for the OpenCL version in different graphics cards. It shows the good scalability of the software depending on the number of cores of the GPU.

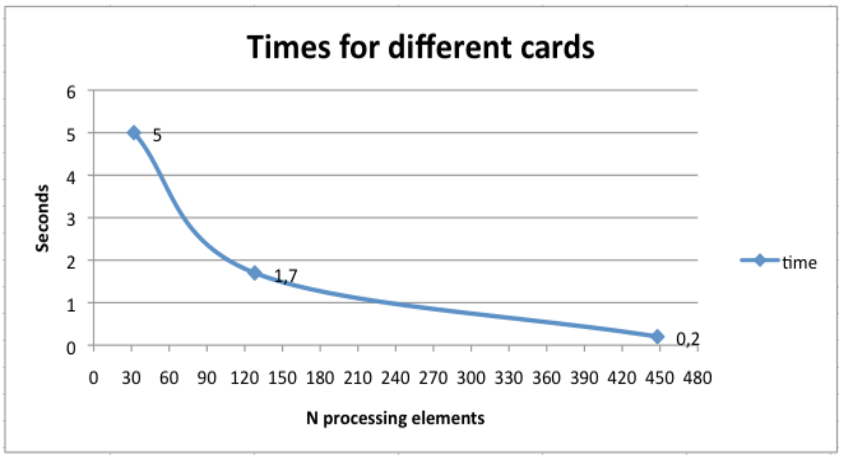


Figure 4.3: execution times for different GPU cards with different number of processing elements. Geforce 9600M GT with 32 PE's, Geforce 8800 GTX with 128 PE's and Geforce GTX 470 with 448 PE's.

The fastest card used to obtain the times in Figure 4.3 at the moment of this writing costs around 340€. This is not a huge price for the performance obtained, and the performance expected to obtain in next iterations.

5 Conclusions

The basic goal of the project was to achieve an execution time improvement over a GSGL version. Even though I decided to take a previous step with some slow parts, in order to later obtain greater speedups in the next step, the results are already better.

This can be explained by the increased parallelism used in the OpenCL version, and the greater control over the GPU hardware. We could control almost every data movement and operation, including all the memory hierarchy. This are the benefits of GPU-OpenCL.

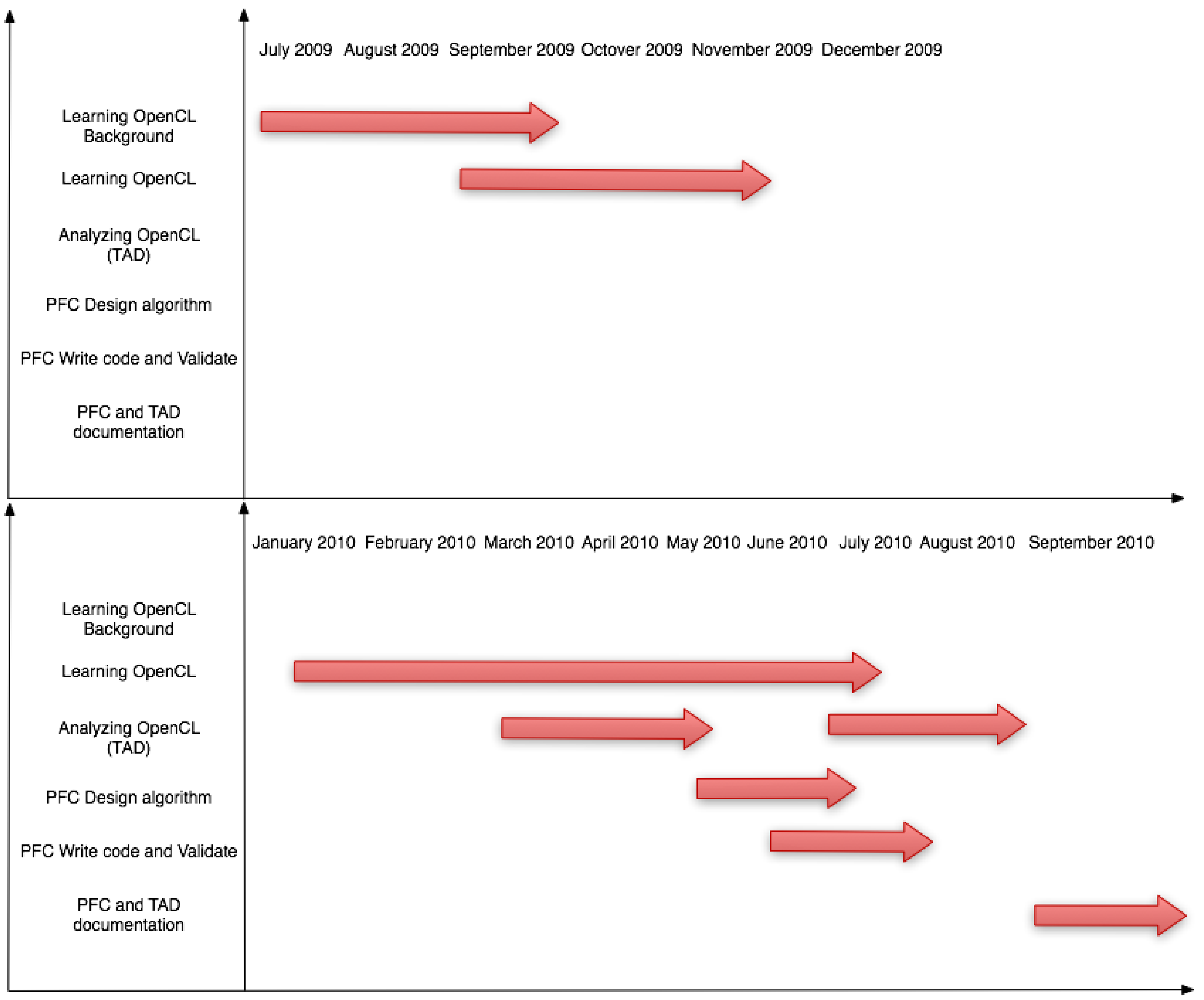
But GPU-OpenCL is not the only purpose of OpenCL. OpenCL is not only a GPU language. It is supposed to be a every thing language, including CPU's, FPGA's etc... A next iteration of the code will explore the CPU kernels in OpenCL, the LLVM compiler optimization capabilities, and the OpenCL task parallelism auto-fork.

The next steps are very promising. We expect to obtain better results in the GPU-OpenCL kernels, but also we can exploit the CPU's at the same time. As CPU's are slower than the GPU, a simple scheduling policy could help to use also CPU without making the GPU to wait doing nothing. As OpenCL API calls are non-blocking, we can wait to finish execution only for the GPU, give a piece of work to it, then check if CPU is working, if not give it some work and wait for the GPU to finish. If when we check the CPU it is working, we don't wait, we continue the loop and wait for the GPU to finish, as it is the fastest device.

6 Gantt chart

The time spent in this project is divided in three main parts. Learning, analyzing OpenCL and doing the PFC. The learning time starts with some basic concepts of computer architecture and parallelism, and then specifically OpenCL. Analyzing OpenCL was done in BSC (Barcelona Supercomputing Center) to test performance of OpenCL in comparison to CUDA, and to test portability to ATI cards. It was a crutial part in learning OpenCL, since I could program, test, and see how OpenCL works. Also I could see some OpenCL examples, learn how they work, and learn the optimization techniques.

Designing and developing the code for the PFC was the fastest part, since learning was really advanced at the moment of starting the project.



7 References

[1] Yoav Freund, Robert E. Schapire. "A Decision-Theoretic Generalization of on-Line Learning and an Application to Boosting", 1995.

[2] David A. Patterson, John L. Hennessy, Computer Organization and Design, Fourth Edition: The Hardware/Software Interface (The Morgan Kaufmann Series in Computer Architecture and Design), Appendix A.

[3] Balázs Kégl, Parallel AdaBoost, M.Sc. internship (stage) project proposal, Linear Accelerator Laboratory, University Paris-Sud/CNRS.

[4] GPU Packet Classification Using OpenCL: A Consideration Of Viable Classification Methods, http://saicsit.wits.ac.za/accepted30.

[5] Bryan Catanzaro, Narayanan Sundaram, Kurt Keutzer, Fast Support Vector Machine Training and Classification on Graphics Processors, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, CA, USA.

[6] Honghoon Jang, Anjin Park, Keechul Jung, Neural Network Implementation using CUDA and OpenMP, Department of Digital Media, College of Information Science, Soongsil University.

[7] Mark J. Bentum, Barthold B. A. Lichtenbelt and Thomas Malzbender, Frequency analysis of gradient estimators in volume rendering.

[8] http://en.wikipedia.org/wiki/Marching_cubes_algorithm

[9] http://en.wikipedia.org/wiki/Raycasting

[10] http://en.wikipedia.org/wiki/Texture_splatting

[11] http://en.wikipedia.org/wiki/Five-point_stencil

[12] Paulius Micikevicius, 3D Finite Difference Computation on GPUs using CUDA, NVIDIA

[13] NVIDIA Best practices programming guide, http://developer.nvidia.com/object/cuda_3_2_toolkit_rc.html

[14] ATI Stream SDK OpenCL™ Programming Guide (v1.05), http://developer.amd.com/gpu/ATIStreamSDK/assets/ATI_Stream_SDK_OpenCL_Programming_Guide.pdf

[15] David B. Kirk, Wen-mei W. Hwu, Programming massively parallel processors: a hands on approach, Morgan Kaufmann, chapter 6

[16] http://www.paraview.org/

8 Appendix TAD (OpenCL: an overview and case study)

8.1 Introduction:

In the past few years we have seen, as usual, an explosion of new technologies that increase the computing power of processors. In the past, this technologies have been introduced first as accelerators and rapidly integrated into general purpose processors as CPU's. This time, the process started as usual, but is having an historically unexpected duration.

The new technology that is revolutionizing the price/performance and power/performance equations in computing industry is called GPU (Graphics Processing Unit) computing or GPGPU. The first difference with other kinds of accelerators is that it was not designed from scratch but from an existent design that had a very specific use.

The graphical use for GPU's required every time more different and specific hardware acceleration for specific visual effects, so GPU designers started to search a way to abstract some parts of the hardware pipeline to create a shared resource model where one or more of the computing units can be used for different kinds of visual effects. That idea, derived to programmable computing units, as to allow programmers to implement new visual effects with the same GPU. At that point, this GPU processors had to manage millions of polygons, and map them to the corresponding pixels to obtain an image, and several times in a second to obtain a moving 3D image. That results in a processor able to modify with simple operations, millions of data points in a second.

Some people from computational science related fields imagined the great utility it could have for intensive Data-Parallel scientific applications and others. Some projects started to search the best way to access GPU's computation capabilities for general purpose, not only graphics, through the graphics API, typically OpenGL. They called that General Purpose GPU or GPGPU programming. Later, the GPU vendors started some initiatives to not only ease this kind of programming but to allow more control over the hardware to allow more different kinds of algorithms to fit on the GPU. So they got the GPU's even more general purpose.

At that point, we have a PCIe accelerator that surpasses the Data-Parallel computational capabilities of the CPU's, sometimes by factors of thousand's. But in contrast with a typical accelerator, GPU's are available to any one who has a computer, so any one has a development platform. Additionally, GPU's are more general purpose than typical accelerators and vendors are trying to increase the capabilities of GPU's in that direction. At architecture level, if we compare CPU's and GPU's, we'll find that trends are making GPU's to look more like a CPU, and CPU's to look more like a GPU. A GPU contains a group of multiprocessors, and each multiprocessor has several small and simple processors. A big scheduling unit in each multiprocessor allows to execute concurrently several threads in all processor of the multiprocessor. Some vendors now are adding caches outside the multiprocessor level, and double precision cores, and more single precision and integer cores in the multiprocessors. They even do, dual issue. So the trends in that case are adding more complexity an not much more multiprocessors. Also, the frequency is growing slowly, being half or less the frequency of CPU's. In contrast, CPU's are made of few big and complex cores, each with its own scheduling system, and sharing expensive coherent caches. Trends are to reduce vector instructions in order to reduce power consumption and die size, multi threading in a single core by making two cores to share scheduling (AMD bulldozer) or one single core to map two threads in the pipe line (Intel Hiperthreading). Also, there are some startups that build x86 processors (up to 100 cores in the same die) of simpler x86 cores, and slower in frequency.

To end with this architecture review, AMD and Intel are mixing CPU's and GPU's in a single die, but it seem's that it still won't be absorbed by the CPU, as there are a lot of possible paths to follow in the processor design both for the CPU and the GPU. It seems that by now they will still be programmed with different parallel paradigms with, different API's. Professionals in the field, predict two more years for the GPU programming to exist as an "out-of-CPU" processor. That will mean longer than expected survival time, and greater than usual changes to the CPU. This is not adding a component to the CPU, but redesigning the whole CPU.

It has been like a huge coincidence, as CPU's had to switch from increasing freqüency to increase performance, to increase complexity and now core count to increase performance. That GPU revolution wouldn't have happened without that factor, since parallel programing was a small market, only for big company's huge servers, or research supercomputers, and therefore most of the commercial development environments, operating systems and programming languages are prepared for one core, or at most 4 cores. To scale and use more than 2 or 4 cores, difficult development has to be done since not all parts of a program are parallelizable. So we end up talking about parallel fractions of a program. Even more problematic is, how well a parallel fraction scales? That is, how many cores is able to use this parallel fraction of the program?. So creating multicore processors means a huge change in the computing industry at hardware and software level, until an standard is reached that allows to develop portable code that scales without the problematic we find now.

The API's that GPU vendors have created, address the scalability problem since allow a deep data-parallelism to create millions of threads for a single program. That million-threaded program will scale perfectly in next generation GPU's accounting for thousands of cores. The hardware scheduling of the GPU's ensure a level os synchronization that allows easier programming.

Taking in account the availability of the hardware, the really cheap price points due to the consumer gaming market nature of the GPU devices, the parallelism problem already going on in the CPU for all markets and the need for new programming paradigms, and the effort of some GPU vendors to port basic libraries to GPU, and making GPU's more general purpose, all that is making the GPU computing not only a success but a revolution. Of course, every body expects the GPU to mix with CPU and at the end have a single programming paradigm, but it seem there's a lot of work in the way to go there.

For all that, and for the potential of GPU computing in scientific programming and hardware availability that allows supercomputing capabilities in a single PC, I've decided to analyze the youngest of the GPU computing languages and API's, OpenCL.

8.2 OpenCL 1.0 overview:

There are basically two major GPU vendors, ATI and NVIDIA. ATI created a GPU computing API called CAL, but they haven't pushed it to be adopted by developers as much as NVIDIA has with CUDA, it's own GPU computing API. Apple was using some of the capabilities of ATI CAL in its operating system but wanted to be able to create code that not only had the benefits of CUDA, but also be portable, so they started the OpenCL project where ATI, NVIDIA, IBM, ARM and several others joined to create an industry standard, managed by Khronos as well as many other standards like OpenGL.

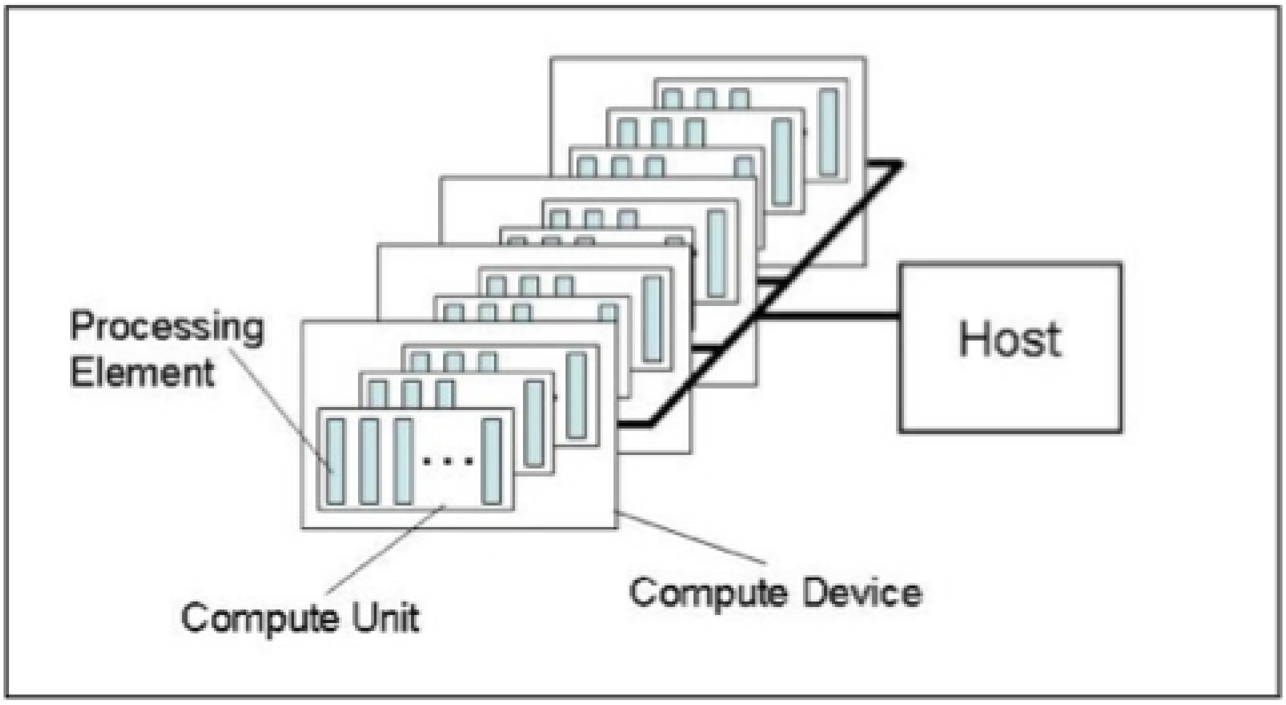
OpenCL is intended to be multi platform, multi vendor, multi parallel paradigm, for performance and commercial software. As to allow for developers to produce performance code without caring about hardware specifics, the compiling process is based on LLVM (Low Level Virtual Machine). LLVM is designed to take care of the best automatic-performance tweaks for the specific hardware being targeted. So OpenCL is trying to ease programming and porting for performance software.

In this work we will discuss first of all different possibilities for GPU OpenCL implementations, to see which kinds of programs are more suitable for GPU programming, and second we will check all that performance and portability attributed to OpenCL, comparing CUDA and OpenCL versions of the same code.

8.2.1 Platform Model

The **Platform Model** defines a virtual machine where usually compute intensive and parallel programs called "kernels" will be executed. This virtual machine is the one we will communicate with, using the OpenCL C language. It also defines the virtual machine components, execution and data transfer mechanisms and the equivalences in real hardware.

Next is defined the virtual machine's structure:



8.2.1.1 Host

The **host** can be understood as the computer itself. The **host** executes a controller program called **host program** that creates the virtual machine and instantiates and manages the kernels, memory objects etc. All that **host program** actions are controlled through C code with OpenCL libraries.

The **host program** can do some automatic resource management, but the programmer can do a more low level approach to do faster code and handle some resources and hardware capabilities manually. This can be also a more platform dependent approach. All that is actually being analyzed by some HPC centers.

8.2.1.2 Compute Device

A **compute device** represents the hardware devices as CPU's or graphics cards (GPU Graphics Processing Unit). If there are "cores" or similar structures inside the device, they are defined as specific components inside the **compute device**.

8.2.1.3 Compute unit

The compute unit is the CPU core or the GPU SM (Scalar Multiprocessor). That means a unit capable of decoding an instruction and executing it, and whose internal parts can only do a fraction of this process.

8.2.1.4 Processing elements

The processing elements in a GPU represent the ALU-like processors with a special external memory addressing system and execution coordination, inside the SM (compute unit). So in the platform model we have processing elements inside compute units, and that ones inside compute devices.
In the GPU's, the processing elements are able to do a few number of different operations, but they are also small. Therefore a big number of them fit in a silicon die. This allows to use them with parallel hardware memory addressing and execution control units to address a lot of parallel operations in each clock cycle, in a single silicon die.
Thats why vectorizing algorithms, and executing them in a GPU makes such a difference in performance. But that is not possible for all kinds of programs.

8.2.1.5 Some real hardware examples

At this point, is interesting to list some real hardware examples that any one can buy (commercial data).

- NVIDIA: GForce 9400M This is a small low-power graphics processor. It is sold integrated with mobile nvidia chipsets. It has 2 SM (Streaming Multiprocessor) or compute units, with 8 CUDA cores or processing elements each one for a total of 16. It develops 54 GigaFLOPS and has Computing capability version 1.1*

- NVIDIA: Tesla C1060 This is a supercomputing-adapted GPU. All the elements that are graphics-only have been removed. It is sold in a two slot size-one PCIExpress connection package. It has 30 compute units, 240 processing elements, is capable of near 1 TeraFLOPS calculations and has Computing capability version 1.3*

- ATI: HD 5870 Is a standard hig-end graphics card processor. Is the newest from AMD/ATI. It has 20 SIMD arrays or compute units each one with 80 SP or Stream Processors grouped in packs of 5 that conform 16 processing elements for a total of 320 in the device. It is capable of a bit more than a theoretical 2,7 TeraFLOPS calculations.

To compare with, the most powerful public supercomputer on the planet has a peak performance of 1456 TFLOPs using hundreds of thousands of processors as of june 2009. Using the ATI HD 5870 only hundreds would be necessary to achieve the same peak performance. The maximum performance is very different though. For the traditional supercomputer, it is between the 80 and 90% of the peak performance. In the graphics cards, it is usually a 20% of the peak performance in a very good implementation. This is because of the memory access managements and latencies. New graphics card technologies developed by NVIDIA are trying to address that with more advanced DMA's and thread scheduling.

The comparison for usable maximum performance is:

- Traditional supercomputer: 1105 TFLOPs with 129600 processors.
- GPU (ATI HD 5870): 1105 TFLOPs with 2050 GPU's. (2050 GPU's using only 20% of their peak capacity develop 1105 TFLOPS)

*Compute capability version is a set of computing capabilities described for NIVIDIA graphics cards that can vary from version to version. There are 1.0 to 1.3 computing capabilities, and the differences refer mostly to memory management, specifically for what NVIDIA describes as coalescing.

8.2.2 Execution Model

The execution model describes the processes in the virtual machine, detailed in the Platform Model, while executing an OpenCL program.
In this section I will explain those aspects of the Execution Model that are not explained through the "Interactions with the Execution Model" subsection in the previous section.
As described in the OpenCL Specification, the Execution Model occurs in two parts: the **kernels** that execute on the devices (that can include the CPU) and the **host program** that executes on the **host** (the CPU that executes the operating system).
Of this two parts, in a performance GPU OpenCL case, the most important to understand is the execution of the kernels, as this information provides keys for knowing how to dispose the information into **work-groups** for better performance. So I'll explain some concepts to understand better the connection between the Execution Model and the code, and the Execution Model itself.

8.2.2.1 Kernels and kernel instances

You can think of a **kernel** like a C function to be executed on a device. The differences with a C function are:

- The OpenCL **kernel** is written in **OpenCL C** code, that means using some specific types, modifiers and libraries and having to deal with a list of important restrictions (you should carefully read them before writing any **OpenCL C** code).
- This code can be precompiled or treated as a string, as source code, for run-time compilation.

We need such abstraction because sometimes is better to read the source code on runtime, to be able to compile this code for different architectures when the program is due to be executed in several different architectures like video games and this kind of commercial apps for the mainstream. This avoids having to include all possible binaries in the distribution software package.

The **kernel instance** is more similar to the object in object oriented programming languages. The **kernel instance** contains the behavior and pointers and variables defined by the **kernel**. There are several **kernel instances** of the same **kernel** running in a device, but using different data. Thats the behavior of the SIMD instructions and the data parallel model in GPU's.
So, what a kernel instance in fact is, is a copy of assembler code for a particular device architecture, executed in a compute unit.

8.2.2.2 Work-Item

What's the difference between a kernel instance and a work-item then?

Well, the work-item concept differs from the kernel instance in the use of the concept. The first is used to talk about the kernel instance with a particular position in an index space, and the second one to talk about the instance itself.

As we saw in the Platform Model section, a NDRange contains a group of work-groups, and each work-group contains a group of work-items.

8.2.2.3 The index space

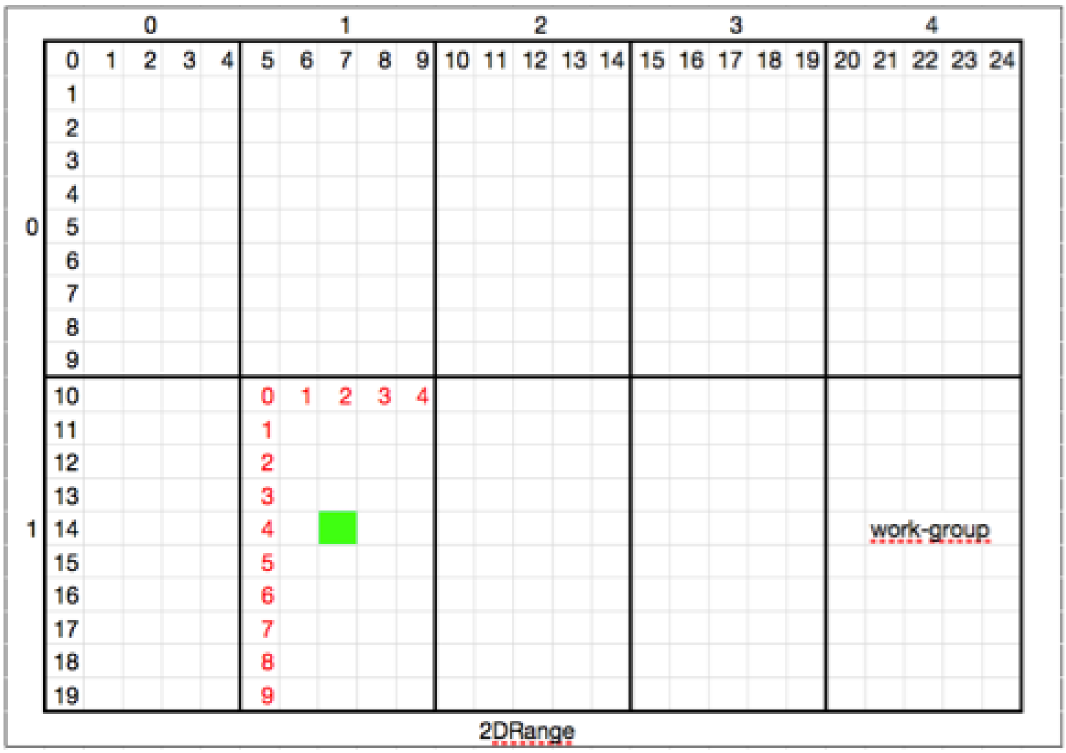
The index space also called NDRange, is indexed as it’s name indicates.

This is an OpenCL concept only for NDRange Kernels, that perform a data parallel model. Even though, this are the most powerful kernels due to the newest GPU’s capabilities.

The indexation is really simple:

- There is an index for each dimension in the index space. - Each index starts in position 0 and increases one unit for each work-item until reaching the number of work-items in this dimension minus one.
- To describe the position of a work-item in the N-dimensional index space, a tuple of N elements is used. (2,4,0) would be the index for a work-item in a 3DRange. This index is called the work-item global ID. It is unique for each work-item.
- Inside a work-group a local ID is defined for each work-item that follows the same rule as for global ID but inside the limits of the work-group. In a 2-dimensional work-group, (0,0) would be the index of the work-item in the top left corner of it.
- Each work-group has a work-group ID to identify it in the index space. It’s the same again. So we can identify a work-item by it’s global ID, or by a combination of the local ID and the work-group ID.

A visual example:



In this index space, you can see the global indexes for each dimension in black numbers, and the local indexes in red numbers.

So, for the green square that represents a work-item, we can define tree indexes:

- The global ID: (7,14) - The work-group ID for the work-group where it is in: (1,1)
- The local ID: (2,4)

To obtain the global ID from the combination of local ID and work-group ID you must perform the following operation:

(1*(work-group x size)+2 , 1*(work-group y size)+4) = (7,14)

Why this is not a good example?? Because in a NDRange the number of work-items per work-group should be a multiple of the number of processing elements in a compute unit, and also the number of work-groups should be a multiple of the number of compute units in a device. This provides optimal computing efficiency and facilitates **coalescing***.

In NVIDIA GPU's the work-groups are executed in portions of 16 work-items or half warp, serializing all the work group in the x dimension, so ideally, almost the x dimension of a work group should be multiple of 16 work-items.

This is because of the GPU's own internal structure. As seen in the Platform model "Some real hardware examples section", all brands use an even number of processing elements in the compute units. The optimal situation is having in each clock-cycle, the same number of work-groups to execute as compute units in the device, and for each w.g.-c.u. pair, the same number of work-items to execute as processing elements at every clock-cycle.

The reason is simple. Doing that, all processing elements will be working in each clock-cycle, and no computational capacity is wasted. For instance, what would be the execution sequence of the 2DRange on the image above, for a NVIDIA graphics card? First of all, a 9 series NVIDIA GPU has a minimum of two compute units with eight processing elements each one. That means that we will have warps with 5 work-items wasting almost half of the processing elements in each clock cycle or if it makes groups of 16 work-items anyway, then 2 processing elements will be wasted at the end of execution of each work-group.

Following the NVIDIA example, you could organize your kernels caring about one 9-series specific model, and think of multiples of its number of compute units, or better try to do multiples of the maximum number of compute units in the biggest model, because it will work for models with less compute units and still better, if it uses a big number of work-groups, it will be efficient with further GPU models that will have more compute units. Fortunately, ATI graphics cards use 16 processing elements per compute unit so it seems that is not that difficult to make efficient and portable code. Any way, doing profiling with NVIDIA cards could produce a code with some problems in an ATI card. That will be one of the analysis for the degree final project which this summary is involved.

So, if you are trying to do a platform independent implementation, maybe you are going to sacrifice some performance, but knowing "why" can help on trying to optimize as much as possible even this circumstances. For instance, using even numbers will always be better than odd numbers.

Also, each device has kernel addressing limits, so a limited amount of kernels can be addressed at a time. It varies not only for the brand but the model of the same brand. An option to face all that, is to query the device, and write host code that decides proportions based on that information.

*Coalescing is a hardware procedure described only for NVIDIA CUDA architecture, that takes groups of work-items aligned in 16 work-items as a “unit”, and move them from local memory to compute units in only one transaction.

8.2.2.4 Context

The context is the environment managed by the host program. The context includes a set of devices, the memory accessible to those devices, and one or more command queues used to schedule execution of one or more kernels. A context is needed to share memory objects between devices.

8.2.2.5 Command queue

As commented in the Platform model section, OpenCL command queues are used for submitting work and data to a device. They provide a channel to communicate with devices through commands. OpenCL executes the commands in the order that you enqueue them in the command queue if the command queue is set as in_order. An out_of_order can be created but it is defined to have implementation dependent behavior.

8.2.2.6 Program objects

An OpenCL program is a set of OpenCL kernels, auxiliary functions called by the kernels, and constants used by the kernels.

An OpenCL program object is a data type that represents your OpenCL program. It encapsulates the following data:

- A reference to a context for the program (which is needed to know on which devices the program can run).

- The program source code or binary. - The latest successfully built OpenCL program executable.

- The list of devices for which the program executable was built, the build options used, and a build log.

8.2.2.7 Memory objects

A memory object is a reference to a region of global memory. You can create memory objects to reserve memory on a device to store your application data. As commented in the Platform model section, there are two types of memory objects used in OpenCL: buffer objects and image objects. The host application can enqueue commands on the command queue to read from and write to memory objects.

8.2.3 Memory model

The memory model handles the device memory. It defines four memory regions with some or no access restrictions for the work-items. The memory objects placed in those regions will be affected by this restrictions.

- **Global Memory:** this memory region permits read/write access to all work-items in all work-groups. Work-items can read from or write to any element of a memory object placed in the global memory. Reads and writes to global memory may be cached depending on the capabilities of the device.

- **Constant Memory:** it defines a region of global memory that remains constant during the execution of a kernel. It means that no work-item can write in any memory object placed in constant memory.
- **Local Memory:** it defines a memory region that is “local to” a work-group. This memory region can be used to allocate variables that are shared by all work-items in that work-group. The local memory is on-chip memory and is the fastest memory available in the device, but also the smallest one.
- **Private Memory:** private memory defines memory only accessible to a single work-item. This memory is the same kind of on-chip memory as local memory. The kernel code has no access to that kind of memory.

If you care about performance, you care knowing when your program is using global or local memory, since local memory is faster and has no coalescing problems.

ATI and NVIDIA have slightly different naming for OpenCL memory hierarchy definitions.

This is the **ATI Memory model definition** (the most similar to OpenCL specification):

- **Private memory** is memory that can only be used by a single processing element. This is similar to registers in a single processing element or a single CPU core.

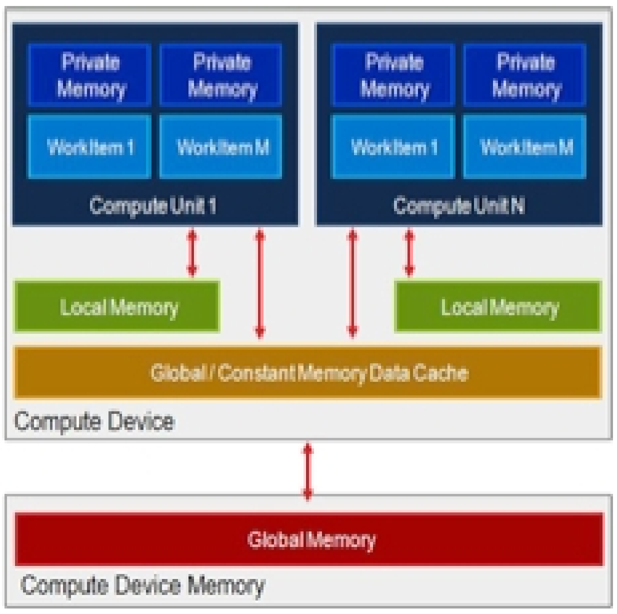
- **Local memory** is memory that can be used by the work-items in a work-group. This is similar to the local data share* that is available on the current generation of AMD GPUs.

- **Constant memory** is memory that can be used to store constant data for read-only access by all of the compute units in the device during the execution of a kernel. The host processor is responsible for allocating and initializing the memory objects that reside in this memory space. This is similar to the constant caches that are available on AMD GPUs.

- **Global memory** is memory that can be used by all the compute units on the device. This is similar to the off-chip GPU memory that is available on AMD GPUs.

So, in this case, the local memory is mapped into a special compute unit cache area accessible to the programmer. Also the constant memory is on-chip memory.

The next picture shows the ATI/Krhonos OpenCL memory definition:



Next is the **NVIDIA OpenCL Memory model definition**:

- **Global memory** is memory that can be used by all the compute units on the device. It is not cached, so the most important is to follow the right access pattern to get maximum memory bandwidth, especially given how costly accesses to device memory are.

- **Local memory** is memory that can be used by the work-items in a work-group. CUDA local memory is not cached, so accesses to local memory are as expensive as accesses to global memory.

- **Constant memory** is memory that can be used to store constant data for read-only access by all of the compute units in the device during the execution of a kernel.
It is cached so a read from constant memory costs one memory read from device memory only on a cache miss, otherwise it just costs one read from the constant cache.

- **Private memory** is memory that can only be used by a single processing element. This is register memory. Generally, accessing a register is zero extra clock cycles per instruction, but delays may occur due to register read-after-write dependencies and register memory bank conflicts.

- **Shared memory** is where OpenCL local memory resides. It is to say that shared memory differs from local memory only by the hardware place it is mapped. Shared memory is on chip memory so it's much faster than local and global memory. In fact, for all threads of a warp, accessing shared memory is as fast as accessing a register as long as there are no bank conflicts between the threads. In the code you may indicate shared memory with the "__local" flag instead of the "local" one used for local memory.

8.2.4 Programming Model

The OpenCL execution model supports data parallel and task parallel programming models, as well as supporting hybrids of these two models. The primary model driving the design of OpenCL is data parallel.

8.2.4.1 Data parallel programming model

A data parallel programming model defines a computation in terms of a sequence of instructions applied to multiple elements of a memory object. The index space associated with the OpenCL execution model defines the work-items and how the data maps onto the work-items. In a strictly data parallel model, there is a one-to-one mapping between the work-item and the element in a memory object over which a kernel can be executed in parallel. OpenCL implements a relaxed version of the data parallel programming model where a strict one-to-one mapping is not a requirement.

OpenCL provides a hierarchical data parallel programming model. There are two ways to specify the hierarchical subdivision.

- In the **explicit model** a programmer defines the total number of work-items to execute in parallel and also how the work-items are divided among work-groups.
- In the **implicit model**, a programmer specifies only the total number of work-items to execute in parallel, and the division into work-groups is managed by the OpenCL implementation.

8.2.4.2 Task parallel programming model

The OpenCL task parallel programming model defines a model in which a single instance of a kernel is executed independent of any index space. It is logically equivalent to executing a kernel on a compute unit with a work-group containing a single work-item. Under this model, users express parallelism by: Using vector data types implemented by the device Enqueuing multiple tasks, and/or Enqueuing native kernels developed using a programming model orthogonal to OpenCL

8.2.4.3 Synchronization

There are two domains of synchronization in OpenCL:

- Work-items in a single work-group. - Commands enqueued to command-queue(s) in a single context.
Synchronization between work-items in a single work-group is done using a work-group barrier. All the work-items of a work-group must execute the barrier before any are allowed to continue execution beyond the barrier. Note that the work-group barrier must be encountered by all work-items of a work-group executing the kernel or by none at all.

There is no mechanism for synchronization between work-groups. The synchronization points between commands in command-queues are:

- **Command-queue barrier:** the command-queue barrier ensures that all previously queued commands have finished execution and any resulting updates to memory objects are visible to subsequently enqueued commands before they begin execution. This barrier can only be used to synchronize between commands in a single command-queue.

- **Waiting on an event:** all OpenCL API functions that enqueue commands return an event that identifies the command and memory objects it updates. A subsequent command waiting on that event is guaranteed that updates to those memory objects are visible before the command begins execution.

8.2.5 Interactions of the platform model with the execution model

8.2.5.1 Device

A command-queue is used for sending commands to the device. This commands can be kernel instantiations, data copies or buffering, and event queries.

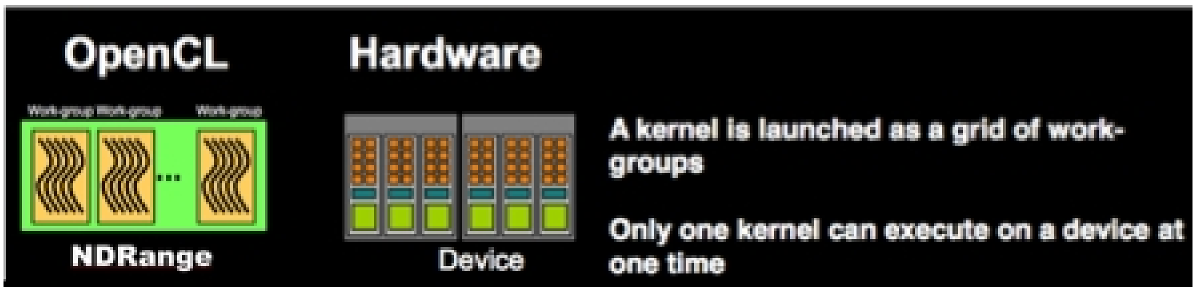
Events are objects representing the commands enqueued in the command queue and containing execution status information about them. This will be useful for debugging purposes and dynamic command management.

The structures used to store or buffer data to the device are the Buffer objects that are one dimensional memory object’s, and the Image objects that can be two or three dimensional memory objects.

We can: Create memory objects assigning to them host pointers to read from or write to. Enqueue this memory objects to the device to make a copy or create a buffer from host pointers to device global memory. Manipulate then the memory object contents with the kernel code, copying parts of it to faster local memories for processing and copying back the results to the data objects that reside in global memory. And finally reading the data from the buffer objects and copying it back to the host memory to make it available to the host program. The command-queue will also be used to create up to three different kinds of kernel instances inside the device. NDRange kernel (data parallel model), kernel (task parallel model) and Native kernel (hardware specific native binaries).

The same OpenCL program can execute different kernels in different devices controlled by the host program. When the host program submits a NDRangekernel to be executed in a device, the kernel is instantiated p times in a n-dimensional index space called NDRange (1d=1DRange | 2d=2DRange | 3d=3DRange). If n=2, x_size=32 and y_size=16 then the kernel will be instantiated p=32x16 times. Tis index space is divided into work-groups. All of them with the same dimension order as the NDRange, and all with the same sizes as each other work-group. For instance, dividing an index space dimension size for the number of work-groups in that dimension should return an integer, as to say that modulus should return zero. This work-groups give more flexibility and coarse grained execution, and make possible to manage bigger amounts of data.

Each device executes only one NDRange at the same time. This NDRange contains instances of only one kernel.

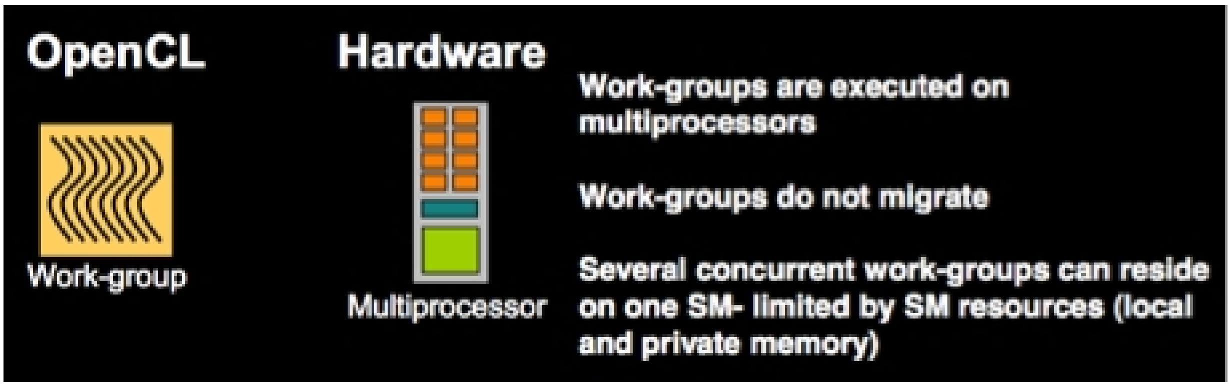

OpenCL
Hardware
Work-group Work-group Work-group
...
NDRange
Device
A kernel is launched as a grid of work-groups
Only one kernel can execute on a device at one time

8.2.5.2 Compute unit

In a compute device the kernels inside of a work-group are executed as concurrently as the compute device is capable of. For instance, in the NVIDIA GPU's with compute capability 1.x to 1.2, each work group is divided in warps of 32 elements and each warp in half warps of 16 elements. That 16 elements are sent to execution in to a CUDA multiprocessor or compute unit.

Executing several work-groups inside the same compute unit is also possible, depending on the sizes of the work-groups and the compute unit capability and memories.

In the picture below, the compute-unit is labelled as a Multiprocessor.


OpenCL
Hardware
Work-group
Multiprocessor
Work-groups are executed on multiprocessors
Work-groups do not migrate
Several concurrent work-groups can reside on one SM- limited by SM resources (local and private memory)

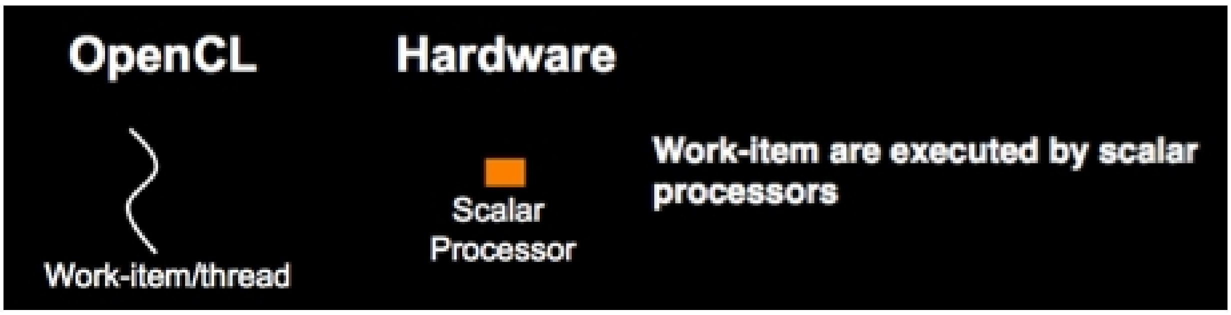
8.2.5.3 Processing element

The kernel instances, are called work-items in the execution model. This work-items are executed in the processing elements.
Each processing element executes one work-item, or kernel instance, with different data depending on the global_id or local_id of this work-item. This will be explained in the Execution Model section.

Optimizing OpenCL code for GPU's and similar architectures like CellBE will be then creating a code that ensures the maximum processing element use rate per clock cycle, for the total amount of processing elements in all the devices controlled by the host program.

The key factor in achieving this, is knowing how to manage the memory hierarchy model, since load and store operations waste lots of computational clock cycles with all the processing elements idle.

In the picture below the processing element is the NVIDIA labeled Scalar Processor.



8.3 Case study

Now we describe some possible OpenCL implementations, and finally the implementation we used to test OpenCL performance and portability.

8.3.1 Implementation of OpenCL in ATLAS for Matlab

Implementing OpenCL in some basic mathematical libraries seems the first natural step, since it makes possible to use OpenCL from any application that uses a non OpenCL version of the same library. In that case we thought that could be interesting to test it with Matlab that is very used for algorithms involving matrix and vector operations, that are very suitable for GPU computing. Matlab uses since version 7.0.1 specific libraries from CPU vendors. This libraries which implement BLAS (Basic Linear Algebra Subprograms) and LAPACK (Linear Algebra Package) are:

- MKL for Intel processors
- ACML for AMD processors.

Before version 7.0.1, Matlab used ATLAS libraries (Automatically Tuned Linear Algebra Software).

ATLAS is a BLAS Open Source implementation with some LAPACK components, that automatically generates a tuned binary version of that libraries for the system where is being installed. Nevertheless, the vendor implementations perform better than ATLAS. Although vendor libraries are not free, and expensive, Mathworks decided to use them for better performance. But Mathworks also allows to change the BLAS and LAPACK libraries an choose one of yourself. According to the online Matlab documentation, there are two environment variables to do so, and indicate the binary library files to use instead of the default ones. So, taking a look to the ATLAS project code, we can see that it is a complex package with C source code and several make files. The complexity of the package is due to the auto hardware detection, and several different versions of the same program depending on the kind of optimization depending on the hardware.

There are any way the basic versions. So we could take this basic codes and implement them in OpenCL. Now, the problem is that we don’t know how the Matlab behavior is. That is, we don’t know if Matlab calls the BLAS and LAPACK libraries with small data packs, or big data packs. In case a call is performed for a small data set, provably is better to use optimized CPU instead of GPU, because of the PCIe transfer times. So there should be a condition or something in the code to detect that and select between CPU or GPU version. We could rely on OpenCL CPU, but as we will see in the CPU OpenCL section, the LLVM optimizations for CPU are not as good as manual optimizations in code and compiler flags. So, because all that, and because there are groups of people on universities and businesses that use to do that kind of work, we searched for a simpler implementation to test the OpenCL characteristics.

At the moment of this work, there was no ATLAS-OpenCL project or similars, but a partial FFT-OpenCL implementation from Apple. Also, Mathworks said that they where not planning to implement any GPU support into their Matlab product. Today, there is an OpenCL Linear Algebra project and others:

- ViennaCL http://viennacl.sourceforge.net/index.php%3Fid=about.html
- Mathworks offers multi CPU, cluster and GPU support for Matlab with a separate product called Parallel Computing Toolbox 5.0.

8.3.2 Four point stencil based 3D finite differences

We implemented a very used numerical method for finite differences to test all the characteristics attributed to OpenCL. This method is a four point stencil, and we added some extras to validate the codes when simulating wave propagation in homogeneous materials. That allowed us to numerically and visually compare the results with already validated codes.

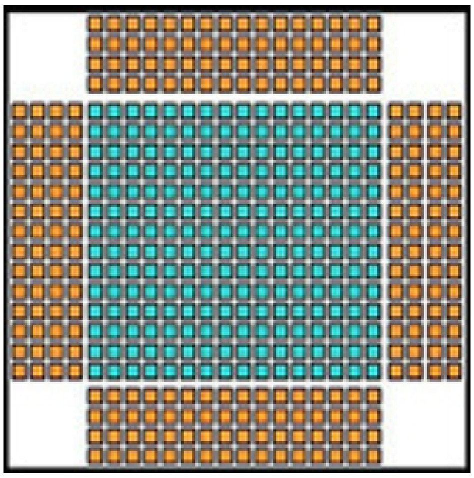
Stencil is a process that sums all the values around one point in a 2D or 3D numerical space, including the value of that point. It can be used for a lot of purposes, but in this case we used it to calculate an approximation of the finite differences of a wave propagation function for each of the points of a 3D numerical space. The number of neighboring points to be summed in each dimension is arbitrarily variable. The more, the better precision we can get on the finite difference, but computing complexity increases with more neighbors so, we chose a 4 point stencil (25 in total counting each side for each dimension). This method is used to calculate a different value for the central point according to the final value of the sum, and some criteria that use to be applying coefficients in one or another way. In this case, the coefficients give to each point a proportion in the influence of his value for the final value, depending on the distance of this point to the central one. In this case we use it to calculate the values of each point for the next time step in a propagation process.

We started with non vectorized implementations for both CPU and GPU to do a first test. We compared CUDA to OpenCL performance for an identical code and then the same OpenCL code in an ATI card, doing three versions of the kernel and testing performance on them to see the effects of each improvement in an ATI card. To deeply test the performance in ATI cards we will implement vectorized versions in a future iteration.

8.3.2.1 Implementation for CUDA and NVIDIA-OpenCL

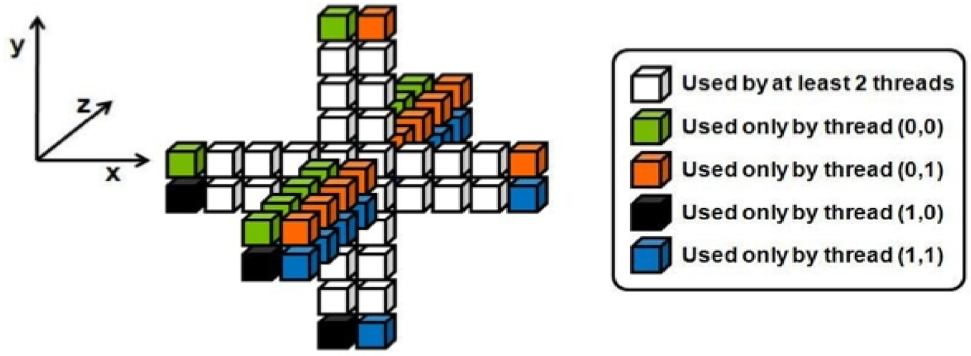
A 4-points stencil is implemented using well-known optimization methods for NVIDIA GPU’s, in CUDA and OpenCL languages. The kernel optimization is based on reducing global memory reads by using registers in the z dimension (”y” geophysical axis), and local memory for x and y dimensions.

Using the GPU’s memory hierarchy, the algorithm fills the registers and local memory to allow the stencil calculation for the inner points of the local memory x-y plane. Reading several times the same data from local memory and registers is far faster than reading from global memories in NVIDIA graphics cards. We used a 2D local memory shape to allow contiguous local memory reads in the x axis, so contiguous threads can read from local in a single transaction in chunks of 16 threads, as specified in the documentation.



Local-Memory data shape: blue is read from registers, orange is read from Global-Memory

The reasons to use registers for the rest of the points not only is that registers are faster, but also the amount of local memory available. Since 9 planes of the fastest dimensions for thread scheduling (32x4 threads in Fermi 32x8 non Fermi Tesla, and 64x2 ATI) doesn't fit in local memory, we use only the central plane with 4 point borders in local memory, and the rest in registers without the borders since we only calculate the points of the central plane in each step. So the 8 planes contained in registers are the ghost points of the inner local memory points, added to the 4-values external radius in local memory, that are also used as ghost points. The stencil calculation advances from plane to plane in the z axis, ensuring that calculations are finished for all the values of the actual plane before starting with the next plane. This is because we don't want a thread changing values in local memory until all calculations are finished for the actual data, to avoid invalid results.


y
z
x
Used by at least 2 threads
Used only by thread (0,0)
Used only by thread (0,1)
Used only by thread (1,0)
Used only by thread (1,1)

Data usage of 4 threads mapped to the 4 inner values

To update the data we only need to discard the values from the first front ghost sub-plane of the 9 planes, move all values one plane front, and fill the last back plane with the next unread plane. This is done by transferring data from registers to local, from local to registers, and reading from global memory only the last back plane and the local 4 point borders. The value of the point being calculated, as long as the ghost points needed in x and y axis, are in local memory, in one sub-plane with x and y sizes according to Work Group or Thread Block x and y sizes. Also, there is a memory padding done to increase coalescing on global memory reads and writes. We tested the performance of this implementation in OpenCL and CUDA against different cards, including one ATI (Juniper architecture), and comparing with a CPU version run on Intel and AMD. We found little to no performance difference between OpenCL and CUDA. For large data sets, CUDA performs a little bit better. This is provably due to a difference in the compiler register management between CUDA and OpenCL, given that high levels of unrolling increases performance in CUDA version and in the OpenCL is the LLVM compiler who decides the unrolling level. Unrolling increases the number of registers being used, and a bad compiler register optimization can produce more registers than needed being used, thus reducing the number of Work Groups or Thread Blocks being active in each Compute Unit or SM. As a consequence, Occupancy is reduced and throughput decreases, reducing the benefits of unrolling. We checked that 32x4 is the better Work Group dimension sizes for Fermi cards, and 32x8 for pre-Fermi NVIDIA cards, for both CUDA and OpenCL. Executing this implementation in the OpenCL version, on the ATI card, works by just changing Makefile parameters, because we focused on avoiding platform defined behavior API calls described on the Khronos OpenCL specification. But this direct execution results in a very low performance calculation.

8.3.2.2 Testing NVIDIA-OpenCL version in an ATI card

Another version of the kernel follows ATI OpenCL programming guide performance guidelines. According to this documentation, coalescing is not as problematic as in NVIDIA cards since ATI subdivides the warps or wavefronts in smaller consecutive chunks. So we avoided the memory padding. The performance difference between the NIVIDIA-OpenCL version with padding, and without padding, is huge. We also tested other padding sizes according to the ATI memory block sizes, but none improved performance. Te best performing NVIDIA version for the ATI is the one without padding.

8.3.2.3 Naïve version for ATI

In order to see the performance difference between using local memory and not using it in an ATI card, we implemented a naive version of the kernel that only uses global memory. We haven’t done it for NVIDIA cards because the difference is already well known and documented. The results are better than the NVIDIA version with padding in the ATI, and also worst than the NVIDIA version without padding on the ATI. To test that kernel, we had to change few things in the driver side, since we reduced the number of parameters to the kernel. We also packed the non-pointer parameters into a constant memory pointer.

8.3.2.4 Register only tuned version for ATI

The last kernel version for the ATI uses the same algorithm as the NVIDIA version, but without local memory. All values that on the other algorithm are in local memory now are read directly from global memory. We used registers for the front, back and center or current values, updating them in the same way as the NVIDIA version, but exploiting float4 data types to calculate and to move the data. We can calculate in chunks of 4 values, and move data in chunks of 3 for the update process, all in one cycle for each Work Item, due to the ATI VLIW capabilities. The left, right, up and down are read and updated all in every plane transition, reading from global memory to float4 registers. Here is where a future implementation will add local memory optimizations to lower the number of global memory reads. This can be done reusing the NVIDIA code. Applying only the first optimization, for the front, back and current values, the times improve noticeably. Adding the second, VLIW for the left-right and up and down, there is also an improvement but not so high as the first one. This shows that VLIW capabilities are important in ATI cards, but reducing global memory reads by using registers using VLIW capabilities in the process is much more effective. Also, we have to take in account that ATI registers are vectorial. There are 16k 128bit vectorial registers or 16k x 4 32bit registers. Twice the 32k of last Fermi NVIDIA cards. So global memory reads, is not so problematic as in NVIDIA cards, but is still a main performance factor in ATI. Also, we tested different Work Group dimension sizes and found that 64x2 is the best for the ATI card.

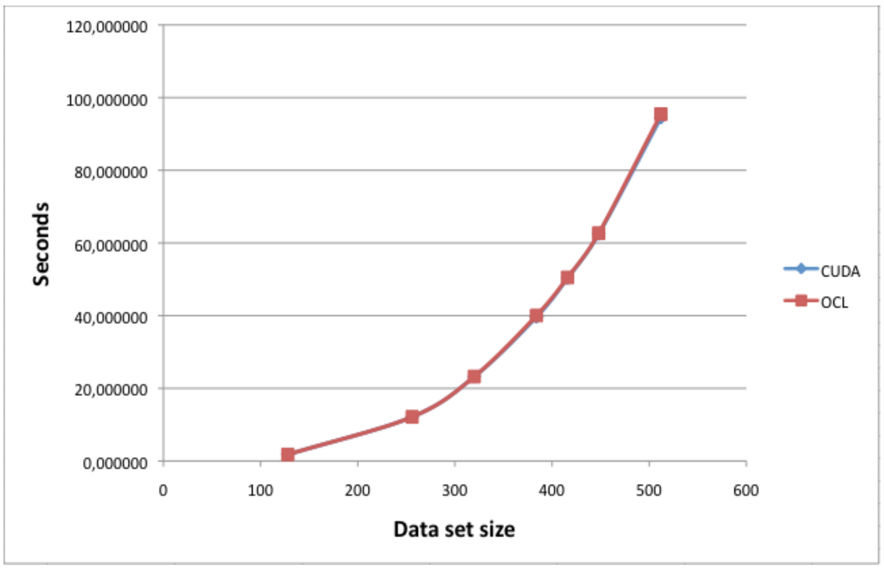
That makes sense since the warp or wavefront size in ATI is 64 Work Items. So this is an ATI sub-performance version, and we still have to implement a true VLIW version for the ATI, but testing it on NVIDIA (changing to 32x4) we found that performance is not so far away from that of the first NVIDIA-performance OpenCL implementation. So we could think of this ver- sion as an universal version, that works pretty fast on both ATI and NVIDIA graphics cards, and was very straight fast to implement it. With more time we could provably find better implementations that work faster than that one, on both architectures, since float4 works too in NVIDIA although without VLIW support.

8.3.3 Results

In this section we describe and discuss the results we had in the tests.

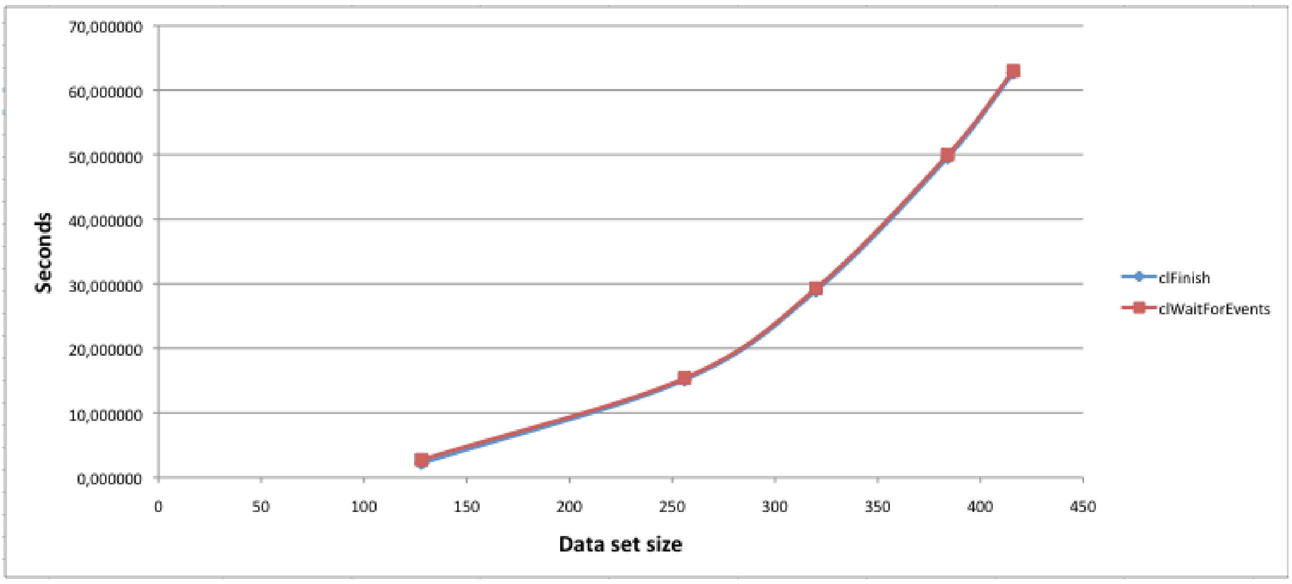
8.3.3.1 CUDA-OpenCL comparison

First of all we wanted to test the performance difference between OpenCL and CUDA for the same code.



As seen in the graphs, the average difference is almost 0% being CUDA the winner for a bery thin margin. So OpenCL for NVIDIA GPU's is still an option.

We also had the chance to compare two OpenCL API functions to synchronice queue execution commands, with different control level over events.



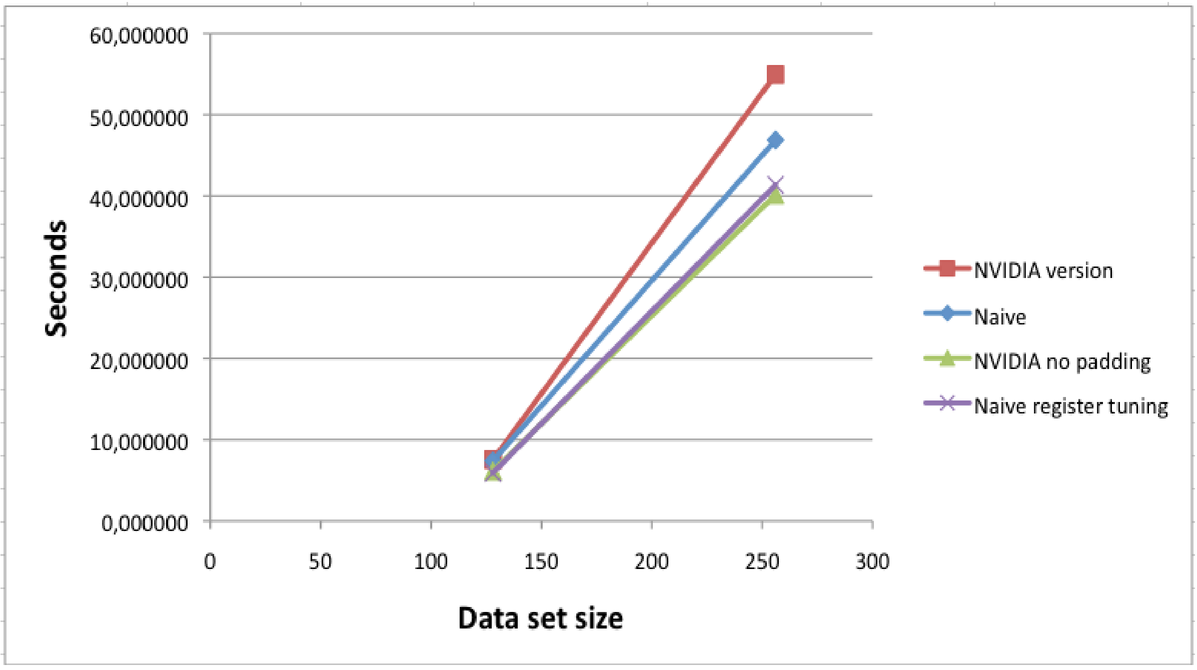
As seen on the above graph, using the OpenCL API call clFinish, is faster that clWaitForEvents. The functional differences are that clFinish acts only as a barrier for the Host program, and makes it wait until all commands are executed on the queue especified in clFinish call. Instead, clWaitForEvents, requires the creation of almost one event for a queued command, in order to make the Host program to wait until that specific event or events is or are finished. That means extra work that produces a measurable overhead.

8.3.3.2 ATI-OpenCL test

Now that we have tested OpenCL performance, we tested also OpenCL portability. We developed the code used in all tests with portability in mind, so we avoided all API calls that imply an implementation defined behaviour. That was a non difficult task.

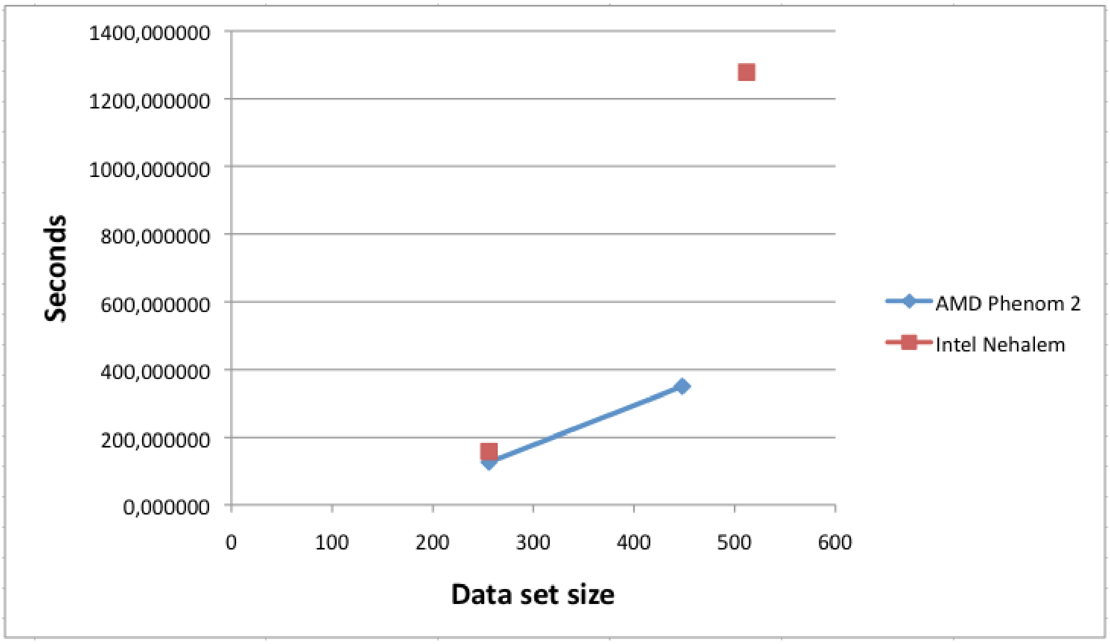
Executing the code used to obtain the results shown in the previous subsection on an ATI card, implied no change on the OpenCL code. Only Makefile had to be changed to target the OpenCL libraries provided by AMD.

We tested four version of the OpenCL kernel. One that is exactly the same as the one used for the NVIDIA tests, the same but without padding alignment, one naive version without using local memory or register optimization and an hybrid version that uses register optimization without local, using too VLIW registers, but not a true VLIW implementation form the ground up. A VLIW implementation will be done in a future iteration, given the VLIW capabilities of the ATI cards.



As a mention, we found that for a given kernel, in an ATI card, the best Work-Group sizes are 64x2.

8.3.3.3 CPU single core tests to compare



In this Figure we can see that CPU times are much more hig than GPU times. But it is important to note that the CPU version is a non multicore CPU version.

8.3.4 Hardware description

There are two main architectural differences that we found important when reading ATI and NVIDIA documentation. The first one is the warp size that ATI calls wavefront. For NVIDIA is 32, and for ATI is 64. We tested performance differences with Work Group sizes of 32x4 and 64x2 and the first performed better on NVIDIA Fermi, and the last one better on the ATI. The second one is the VLIW capabilities of ATI cards not present on NVIDIA, and used by ATI to measure the theoretical peak performance of their graphics cards. So at first, it seems more difficult for an ATI card to have a good percentage of use of it's peak performance, since not only we have to parallelize the program, but also use vectorized data types to achieve maximum performance. Apart from this ones, there are other differences like non coalesced global memory accesses behavior, and non 32bit data types, as well as global and local memory bank conflicts. Next we show hardware tables with the main characteristics for each processor used on the tests:

Card	NVIDIA Tesla C2050 (Fermi)	ATI HD5750 (Juniper)
Compute capability (only NVIDIA)	2.0	-
Processing elements	448	720
Stream Cores/CUDA cores	448	144
Compute units/Multiprocessors	14	9
Global Memory Size (GB)	3	1
GPU Frequency (MHz)	1.15	700
Memory Frequency (MHz)	1500	1150
Peak single precission throughput (GFlops/s)	1030	1008
Peak double precission throughput (GFlops/s)	515	-
Number of registers	32768	16384
Register width (32bit/unit)	1	4
Main memory standard	GDDR5	GDDR5
Local memory (KB)	48	32
Constant memory (KB)	64	48

Card	NVIDIA Tesla C1060	NVIDIA GTX 470 (Fermi)
Compute capability (only NVIDIA)	1.3	2.0
Processing elements	240	448
Stream Cores/CUDA cores	240	448
Compute units/Multiprocessors	30	14
Global Memory Size (GB)	4	1.28
GPU Frequency (MHz)	1296	1215
Memory Frequency (MHz)	800	1674
Peak single precission throughput (GFlops/s)	933	1088.64
Peak double precission throughput (GFlops/s)	78	136
Number of registers	16384	32768
Register width (32bit/unit)	1	1
Main memory standard	GDDR3	GDDR5
Local memory (KB)	16	48
Constant memory (KB)	64	64

Processor	CPU AMD Phenom2 x4 955 (Deneb)	CPU Intel Xeon x4 (Nehalem)
Sockets x cores	1 × 4	2 × 4
Memory per board (Gbytes)	4	?
Clock Frequency (GHz)	3.2	2.0?
Peak throughput (GFlops/s)	N/A	80.0
SIMD registers (per core)	N/A	N/A
SIMD width	N/A	128 bit
Main memory standard	DDR3	DDR3
Cache memory		
L1 (data + instr)	64KB + 64KB	32KB + 32KB
L2	512KB per core	N/A
L3	6MB shared	N/A

8.3.5 Future work

Future work can be centered in some different aspects. Tuning the OpenCL kernel: the kernel used in this experiment was developed by a famous programmer called Paulius Micikevicius, and explained in the paper 3D Finite Difference Computation on GPUs using CUDA. This algorithm is highly optimized for GPU's, but we think some small optimizations can still be performed, by completely removing the register movements, mixing it with an unrolling. That will increase de code size, but taking in account the Fermi caches that reduce latency for register instruction global memory spilling. Another possible way to optimize it, is by reducing local memory reads using the semi-stencil algorithm. Developing a true VLIW kernel version for ATI cards, and optimize it following the ATI guidelines and architecture. Then, compare price/performance between NVIDIA and ATI, and check which performance the ATI version has on NVIDIA cards.

8.3.6 Conclusions

Overall, the slowest part to code was the driver part. All the needed API calls, OpenCL objects and so on, makes the coding a long work. It also gives a lot of control over the process though. The kernels are almost the same as in CUDA. It was the fastest part. So there is a reasonable chance to successfully create interfaces that ease coding the Host part, as found in the literature. Portability in OpenCL is a valid feature, since taking care to avoid the implementation defined API calls, the code worked without any change. So having an OpenCL version of a program, allows to change hardware, and start from the tuning stage, instead of starting by learning a new API and language. Performance is comparable to that of CUDA, so if in the future, the majority of the hardware supports OpenCL, it will definitely be the way to go.

8.4 OpenCL 1.1 flash view

At the moment of writing this documentation, AMD has released few weeks ago the first public OpenCL 1.1 version. It is supposed to fix some issues found with ATI cards when using smaller than 32bit data types, and add support for SSE 2.x CPU's, plus adding the changes from 1.0 to 1.1 version. Major changes are related to integrating some extensions as default OpenCL functionalities, reduce OpenCL C restrictions and improving integration of OpenCL with different graphics API's like OpenGL and DirectX.

8.5 CPU OpenCL and other considerations

Apart from the device specific parallelism capabilities, there are several different kinds of parallelism that can be exploited on OpenCL. First of all, every call to a queue, is a non-blocking call, so the Host program can perform other tasks like performing other calls to other queues linked to other devices. That allows to control the scheduling of tasks between different devices of the same or different architecture. For instance we can program a work flow that is valanced by the OpenCL Host program between all CPU's and all GPU's on a system. We can also synchronize queues by blocking the Host program until all enqueued commands are performed in a given queue, or until one of the commands of a particular queue is finished. That last synchronization system uses what's called OpenCL events. NVIDIA OpenCL examples use this event synchronization approach, but for our code this is unnecessary, we can synchronize by waiting all commands to finish in the queue with a simpler API call and without generating OpenCL events, thus reducing the overhead. We measured this difference, and for small data sets it is very noticeable, but also observable for big ones. We tested a C version of the stencil algorithm implemented in an OpenCL CPU kernel to see the difference between compiling with gcc knowing the architecture of the CPU, so adding architecture specific compiler flags to improve performance, and LLVM automatic tuning compiler used by OpenCL. The results sow a big difference in performance, being the OpenCL implementation much slower. Fixstars has developed an OpenCL for CPU's implementation, and they say that their implementation performs much better than the AMD one, that is the OpenCL implementation we used to do the test. Until we don't find better results with OpenCL CPU implementations we won't put much effort into CPU OpenCL implementations. By now, is much more interesting for performance reasons, use OpenMP for CPU's and OpenCL for GPU's.

8.6 Conclusions

Learning OpenCL has some stages. The first one is conceptual. For those used to program parallel languages and CUDA, this is a no brainer. The new concepts maybe can be on the driver side if the programmer have not entered into much detail or multi-device programming with CUDA. The second one is to learn the OpenCL C language restrictions (this are only 15), and OpenCL API calls. All that can be done while coding having the OpenCL specification around, but reading carefully to avoid implementation defined behaviors and missing important details. Overall, OpenCL seems a good option for GPU computing today, for the support that NVIDIA gives and AMD focusing all the efforts in OpenCL. Also it can be a good choice for the future, in order to preserve development investments. Some new architectures are supporting OpenCL like the Zii processor, and OpenCL is designed to support any kind of hardware like accelerators, FPGA's etc...

9 Appendix A: how to run the software

For either ATI or NVIDIA version it is necessary to first install a specific driver into a PC with an ATI or NVIDIA graphics card supporting OpenCL. All that information can be found in their web pages.

Once installed all drivers, or havig an Apple computer with Mac OSX 10.6 with a compatible graphics card, you only need to compile the software with gcc and some specific flags depending on the OpenCL implementation.

For detailed instructions see the ATI or NVIDIA web pages or: The OpenCL Programming Book, Ryoji Tsuchiyama and others, Fixstars, http://www.fixstars.com/en/company/books/opencl/