Master in Artificial Intelligence (UPC-URV-UB)

Master of Science Thesis

PARALLEL ERROR-CORRECTING OUTPUT CODES CLASSIFICATION IN VOLUME VISUALIZATION: parallelism for IA and IA for parallelism

Oscar Amoros Huguet

Advisor/s: Sergio Escalera, Anna Puig
Dr./Drs. on behalf of the Advisor/s: Sergio Escalera, Anna Puig

UNIVERSITAT DE BARCELONA, UNIVERSITAT POLITECNICA DE CATALUNYA, UNIVERSITAT ROVIRA I VIRGILI

Abstract

Facultat de Matemtiques UB
Departament de Matemtica Aplicada i Anlisi

Master in Artificial Intelligence

by Oscar Amoros Huguet

In volume visualization, the definition of the regions of interest is inherently an iterative trial-and-error process finding out the best parameters to classify and render the final image. Generally, the user requires a lot of expertise to analyze and edit these parameters through multi-dimensional transfer functions. In this thesis, we present a framework of methods to label on-demand multiple regions of interest. The methods selected are a combination of 1vs1 Adaboost binary classifiers and an ECOC framework to combine binary results to generate a multi-class result. On a first step, Adaboost is used to train a set of 1vs1 binary classifiers, with a labeled subset of points on the target volume. On a second step, an ECOC framework is used to combine the Adaboost classifiers and classify the rest of the volume, assigning a label to each point among multiple possible labels. The labels have to be introduced by an expert on the target volume, and this labels have to be a small subset of all the points on the volume we want to classify. That way, we require a small effort to the expert. But this requires an interactive process where the classification results are obtained in real or near real-time. That why on this master thesis we implemented the classification step in OpenCL, to exploit the parallelism in modern GPU. We provide experimental results for both accuracy on classification and execution time speedup, comparing GPU to single and multi-core CPU. Along with this work we will present some work derived from the use of OpenCL for the experiments, that we shared in OpenSource through Google code, and some abstraction on the parallelization process for any algorithm. Also, we will comment on future work and present some conclusions as the final sections of this document.

Chapter 1

Introduction

In this chapter we present a summary of the goals and contributions of the thesis.

1.1 Goals

The main and primary goal of this thesis has been to explore the interaction of AI and parallelism. How can AI algorithms be parallelized, and how much do they benefit from that, as well as have a look on what can some AI technologies add to the design of parallel computing systems.

Our main goal in this thesis is to have a working proof of concept of a classification framework for medical volume dataset, that has to be interactive and semi automatic. To accomplish this, we want to implement a multi classification functionality, use standard visualization tools, and achieve accuracy and interactivity constraints. For matching the interactivity constraint we use parallelism.

Other goals have been appearing during the development of the thesis, like the creation of AIMparallel, a methodology for easing and guiding the parallelization process of an algorithm and SimpleOpenCL, a framework we created, devoted to reduce the amount of code needed in order to create a parallel OpenCL program.

As a second main goal, to explore AI techniques that could help on solve some problems on parallel computing designs, we found Multi Agent Systems is a source of a lot of features desirable for parallel computing systems of the future. So we propose a system based on specifically Self Organizing Multi Agent Systems.

This project requires knowledge from a wide variety of fields. Computer Vision, Machine learning, Computer Science (Parallelism), Visualization and Multi Agent Systems.

1.2 Contributions summary

In this thesis we show two proposals. The first and main one is a classification problem, and it's implementation using parallelism. The second one is based on the experience and learning process of the first proposal. On this second proposal we show a draft of a parallel computing system based on Self Organizing Multi Agent Systems, with some basic experimentation. On the first proposal we used parallelism to implement IA techniques, and on the second one we use IA technologies to define a parallel computing system.

For the classification problem we propose a general framework of supervised statistical classification methods to label on-demand multiple regions of interest (see Fig. 3.1). This provides an interactive classification/segmentation generic framework that could be used in several real applications, and it can be used with any existing classification/segmentation state-of-the-art method. The framework is composed by a pre-learning stage and an on-demand testing stage included in the renderer. The learning step gets a subset of pre-classified samples to train a set of Adaboost classifiers, which are codified as TFs, and combined in an ECOC design. Each classifier encodes a set of relevant properties in a 1D texture. We deal with several properties, such as density, gradient, space location, etc. of the original data without increasing the dimensionality of the classifier. Next, the testing stage multi-classifies and labels a subset of volume classes based on user interaction. The label mapping defines clusters of the selected classes, and then it assigns optical propeties and importance values to the final output classes to be visualizated. The labels computed in the testing step in the GPU memory can be used in several volume visualization approaches: to skip non-selected regions, to help computing importance values to different context structures, or to select the focus of interest for automatic view selections, just to mention a few. The label mapping reduces the edition of the TFs by only assigning the optical properties of a label. In addition, the labels, as a TF, can be applied directly to the voxels values, or alternatively to the sampling points derived from the interpolation of the nearby voxels. Up to now, the complexity of the learning process and the testing step of a large amount of data do not allow their integration into an interactive classification stage of the user-interface volume pipeline. In this sense, our proposal improves the TF specification process by combining a pre-processing learning step and a rendering integrated GPU-based testing method.

In summary, on the classification problem we bring four contributions: First, the definition of a general framework for multi-classification of volumes based on the ECOC approach; Second, the use and parallelization of Adaboost as a case study of this general framework; Third, the computation of an on-demand adaptive classification to a subset of features of interest; Finally, the proposal of a GPGPU OpenCL implementation of the testing stage of the multi-classifier integrated into the final rendering. This work

serves as a proof of concept for future embedding of the training step in the visualization pipeline. In this sense, online learning will be performed on-demand in the rendering. Results on diferent volume data sets of the current prototype show that this novel framework exploits GPU capabilities at the same time that achieves high classification accuracy.

The parallel computing system mainly proposes a design change on parallel computing systems, based on the evolution of the technology, that allows to have extra hardware executing Agent programs for the benefit of the whole system. This benefit comes from the optimization of the network use and from the decentralization of the scheduling. Many of this benefits are already pursued on other projects for parallel computing languages and frameworks, but they continue to use traditional programming paradigms for the implementation of the system.

The rest of the thesis is organized as follows: Next chapter overviews the parallelism state of the art and presents our AIMparallel methodology. Chapter 3 presents the classification problem, it's definition, parallel implementation, experiments and results. Chapter 4 explains the SOMAS based parallel computing system proposal with a basic first definition, a basic simulator and some small experiments with it. Finally chapter 5 gives some conclusions on all the work done.

Chapter 2

Parallelization

2.1 Introduction

On this chapter we are going to introduce the problem of parallelization and more specifically parallelism on the GPU. This chapter introduces concepts necessary for the main contribution, the classification problem, and que parallel system proposal. We begin with an state of the art and ends with a parallelization methodology and nomenclature that we propose as part of this thesis.

2.2 State of the art

2.2.1 Introduction

Parallelization technologies have been widely used for applications where the amount of data and/or calculations is very high. For having results in an acceptable time on this problems, several computers are required to be computing different parts of the same problem at the same time. Many languages and very specific networking and computer technologies have been used to tackle scientific or statistical problems for both public and private science and industry along last decades. Vector units first (instruction level parallelism) and Shared Memory Multiprocessors later introduced parallelism in certain degree into workstations, allowing for a wider audience to create parallel programs, or for increasing the processor densities of clusters.

This audience broadening turned into a tendency, and many languages have been adopted as standards like OpenMP (for shared memory multiprocessor systems) and MPI (for clusters). Nevertheless, many technical problems arise almost at the same time:

- An increased cost and difficulty of shrinking processors.

- An increased cost and difficulty of keeping power consumption levels with the increase of transistor number, added to an increased demand of power efficiency for mobile devices.
- An increased cost and difficulty of keeping the data interfaces (from networks to memory interfaces and off-chip wiring) to keep the data transfer performance on pair to the processors processing capabilities.

For reducing this problems at a hardware design level, the easiest solution is to increase parallelism in several architecture levels, plus increasing power modularity on the processors, leading to many parts of the processors being turned on and off depending on usage for saving power, and matching thermal requirements.

This increase on parallelism, at the processor and ALU level has many problems:

- Parallelism is not a need by it self in many applications, but an added complexity in order to fully use new systems. Programmers tend to skip it or even don't know how to use it.
- Vector units usually need proprietary compilers to be used, and is complex.
- At multi-CPU level, cache memories intended to mitigate the memory interface performance problem became a problem. The more CPU's the more complex and big becomes the caches and memory coherency ends up consuming most of the power and chip surface.
- Parallel programming models like OpenMP or MPI are known by a small community of programmers, and businesses try to avoid their need as much as possible. A proof of this is the success of specific frameworks or languages like those based on Map-reduce [JD04], Scala [dSOG10], etc... They are specific for certain tasks or needs, but they are easier to use since they are very high level.

GPU (graphics processing units) have been very specific processors that had much more freedom to evolve than general-purpose processors like CPU, which have to manage an operating system with all the complex functionalities it implies. Additionally, GPU's are added to systems with a piece of software (a driver) that makes the systems to know how to communicate with them. Also, some GPU programming standards appeared, like OpenGL, that abstracted architectures and offered a single graphics programming model for all GPU's. All this modularity and specific functionality allowed GPU designers to create almost any architecture, that better matched the graphics computing needs. These needs are mainly programmable data parallelism for allowing the programmer to create their own visual effects, a big memory bandwidth and some very common functionalities that can be tackled with super fast fixed function units. As a result,

GPU ended being a relatively cheap and very parallel and fast architecture, that once adapted with the help of some new languages (CUDA and OpenCL), evolved into a more general purpose parallel processors (GPGPU). Nevertheless, GPU's are still not true general purpose, they are very good for some tasks, good for some others, and bad for all serial codes or task parallelism (like operating systems). CPU's are the opposite, they can provide a very limited amount of parallelism compared to GPU's with a very less efficient power consumption per core, but they are much more efficient on computing serial tasks and operating systems. This makes CPU's and GPU's a pair of very complementary architectures, not to mention that they are present together in almost all computer systems, including mobile devices.

But CPU's and GPU's are not the only architectures around. In general, architectures can be classified by it's level of specificity or generality on the task they can perform. ASIC (application-specific integrated circuit) circuits designed to a single task are the most efficient solutions, but they are not adaptable, since they are not programmable, they will always perform the same task. FPGA's (Field Programmable Gate Array) are a bit more flexible, since they are intended to be programmable circuits, but programming a circuit is much more complex than programing on a compiled high level language, for a general purpose processor. GPU's include ASIC functionalities and semi general purpose highly parallel units. Some CPU's are starting to include some ASIC units for video processing too, like Intel Quick Sync. Nevertheless, mostly on mobile devices, there are more specific purpose units like video processors, and a lot of DSP that are mainly ASIC's.

Programming GPU's for general purpose can be done mainly with CUDA or OpenCL languages. They are mainly the same, and their performance too, but CUDA only works for NVIDIA GPU and OpenCL works in any GPU, CPU and is designed to work on any existent and future accelerator (floating point operations accelerator) technology like FPGA's and ASIC's. The advantages of CUDA are the availability of some NVIDIA GPU features that might not be supported in OpenCL (since NVIDIA is not creating OpenCL extensions for supporting them) and the more programmer friendly options that CUDA has on the CPU side of the code. As CUDA has been the first computing oriented GPU language to appear, it gained broad adoption on the scientific community and there are many papers on the literature implementing varied algorithms from many domains in CUDA [GLD*08, HGLS07, NL07]. Nevertheless, this algorithm implementations are almost identical when programmed in OpenCL, and additionally, having OpenCL code allows to test a wider range of hardware, with less coding effort than using proprietary languages for each hardware. In fact it is possible to use exactly the same code for different vendors.

Given the data parallelism of the algorithms and data we are using, the natural choice is GPU's. Additionally, the data being processed is already on the GPU for visualization

so we are not creating any unnecessary data transfer. For code portability and being able to work with different architectures we chose OpenCL instead of CUDA.

On the next sections we will review some important aspects of the GPU architecture and the GPU programming model.

2.2.2 GPU architecture review

GPU's are mainly a set of big vector processors, sharing a common memory space and each one with a big control unit. This vector processors or compute units (OpenCL nomenclature) have a very varying amount of ALU's for integer and floating point operations, depending on the model and the vendor (from 2 ALU's in a super scalar pipelined fashion on Intel GPU's to 192 ALU's on NVIDIA GPU's).

On this section we will divide the explanation of GPU architecture in to memory hierarchy, execution internals, and programming model.

2.2.2.1 Memory hierarchy

The main memory of this GPU's usually is a set of GDDR5 banks, in case of discrete PCIe GPU's. More recently, some GPU's share the same CPU DDR3 memory, where the operating system splits it into two different memory spaces, saving costs, but reducing performance. The newest technologies already available, allow the system to either simulate a single memory space for GPU and CPU, regardless of the real memory being used (NVIDIA CUDA 6), or to actually allow the GPU to coherently use the same memory space as the CPU through driver and hardware support (some AMD APU's). This memory is called global or device memory, according to OpenCL and CUDA nomenclatures respectively. From now on we will only use OpenCL nomenclature for clarity. Refer to Appendix A on this document for doubts about OpenCL nomenclature.

Additionally to this global memory, compute units usually have access to a coherent cache that allows to reduce the memory latencies generated when accessing global memory, that until today is always off-chip memory. Accessing off-chip memory implies signal amplification and sometimes translation. All this process is time and power consuming. Compute units might have to wait up to 600 computing cycles in order to have a single global memory transfer. For that reason, additionally to this cache memory, there is another memory space private to each compute unit, that is small but very fast, who's latency is usually 2 computing cycles. This memory is called local memory, and the programmer can explicitly create pointers to this memory space.

Local memory is more complicated than that though. When a programmer declares a variable that resides into local memory, there will exist as a different copy of the

same variable on each compute unit. The programming model, allows the programmer to identify each one of the copies using thread id's instead of variable indexing as in standard C. We will see more about this on the Programing model section.

Finally, each compute unit has a big register bank where instructions and data reside. On modern GPU's, we can declare a variable to be private, so it will more provably reside on registers, that is the fastest memory on the GPU. Nevertheless, the scope of private memory is reduced to a single thread, so there will be a copy of this variable for each thread on each compute unit. Again, the programmer can use thread id's to know which of the copies is he using, but each thread is responsible by default of one copy and can not access directly the other thread's copies. The data has to be moved to local or global memory in order to be accessible by other threads.

2.2.2.2 Execution internals

Compute units execute vector instructions named warps or wavefronts that can be usually split into smaller vector instructions if it is convenient on execution time. For instance, if only the first half of the warp has the data available to execute. Compute units have a memory management unit, so they can fetch data and execute instructions at the same time. This allows for memory latency hiding.

The programming model allows to treat each one of the data elements on a vector instruction as private memory for a different thread. If on a single warp, two or more threads have to execute a different instruction on the same data, this can only lead to have two or more separated warps, each executing different actions on different data elements. This clearly reduces parallelism. So divergence inside vector instructions is allowed at the programming level, but is translated into more instructions at the hardware level.

One of the most important concepts though, is coalescense. If all the data required by a warp, exists on consecutive positions in memory, and on the same memory bank, then all this elements can be read on a single memory access. In the worst case, if every data element required by the warp resides on different memory banks, then there will be a data transfer for each element instead of one for all. Cache memories mitigate the problem of non coalescing accesses but still, it is a problem when accesses are very sparse, and the proportion of calculations per memory access is low.

2.2.2.3 Programming model

First of all, it is important to remark that a program that uses GPU for computation has two separated codes. The Host code and the Device code (see Appendix A). When

we talk about the GPU programming model we are talking about the Device code, that is the code compiled for the GPU and executed on the GPU.

The GPU programming model stands on a basic principle. The code that the programmer writes, is executed by default by all existing threads. The number of existing threads is set up by the programmer on the Host code, so before the Device or GPU code is executed. Additionally, this threads or Work Items (as in OpenCL nomenclature) exist on a 1, 2 or 3 dimensional space. This space is called NDRange in OpenCL, where N can be 1,2 or 3, indicating the dimensionality. Then, each Work Item has id's with as many components as dimensions the space it exists in. Specifically, a Work Item is related to the following id's:

- Global id: This id is a unique id that identifies the Work Item on the NDRange space. It has 1, 2 or 3 numerical components. A Work Item in a 2DRange space has a global id of the shape (x, y).
- Local id: Work Items are grouped into Work Groups. The first dimension of the Work Group ideally should be multiple of the size of a warp. So, ideally, Work Groups are packs of several consecutive warps. They can be consecutive on any of the dimensions. See more about work groups on Appendix A. So a local id, identifies a Work Item inside the Work Group it belongs. That means that different Work Items can have the same local id, if they are in different work groups, but not if they are on the same work group. Local id's have the same shape as global id's. In the case of global id's, they are all unique.
- Group id: Work Groups also have an id for the programmer to be able to identify them. They have the same dimensionality as the NDRange.

By default, all Work Items are independent of each other, but they are executing the same code. In order to make them operate on different global or local data, we can use their id's to index pointers. That way, all Work Items will perform the same operation, but on different data. This is exactly data parallelism. But also, we can use the Work Item id's on conditional statements. That way, each Work Item can perform a different task. As we have seen previously, this leads to a reduction of overall performance.

The reason why it is possible to do Work Item divergence, despite it is bad for GPU performance, is because the alternative, moving data back to CPU (because the CPU does serial execution faster), is usually much slower than doing it directly into the GPU. This is because the time spent on moving the data can be greater than the time the GPU will spend executing serialized parts of the code. Nevertheless, this can change with the newest hardware, where GPU's have the capability to work with x86 virtual memory. In that case, there are not data transfers, only cache misses, if the GPU and CPU are not sharing the same cache.

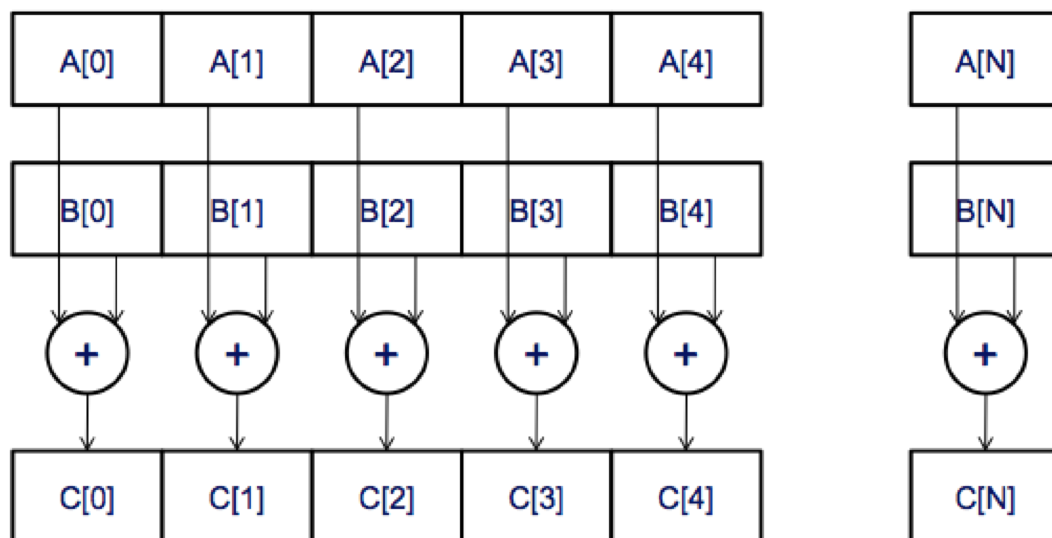


FIGURE 2.1: Vector addition in parrallel of variable arrays A and B storing the result in variable array C. On an 1DRange kernel, Work Item 0 reads A[0] and B[0], adds the two read values and stores the result in C[0].

2.2.3 GPU programming review

GPU programming is increasingly becoming the art of finding data structures and algorithms that allow some programs that are difficult to parallelize or coalesce, to run fast on the GPU.

Some of the most basic examples are dense matrix multiplication, stencil operation on volumes, vector addition, etc... As we can see on figure 2.1, some algorithms are perfectly suited for GPU's, because each Work Item is reading a memory position that is contiguous to the memory position that the next Work Item is reading, therefore, all warps will use all ALU available (except maybe for the last warp if N in figure 2.1 is not divisible by the warp size).

More difficult codes though present several programming difficulties to overcome, or different behaviors than in CPU. For instance, like particle simulations where scatter operations used to be more efficient on the CPU, but on GPU gather operations turn to be more efficient. We can see an illustration of this on figure 2.2. This turned to be very important to our implementation since we finally implemented the "owner computes all" rule mentioned on figure 2.2 (b). This means that conceptually we assign an output to a Work Item, and this Work Item is going to do all the work in order to obtain the output. It is not so obviously the best solution though, as we will see later.

Other algorithms are difficult to parallelize like sparse matrix. The CPU optimized sparse matrix algorithms that use data structures like CSR (Compressed Sparse Row format) [Rob] don't run very well on GPU's and there are some changes that can be

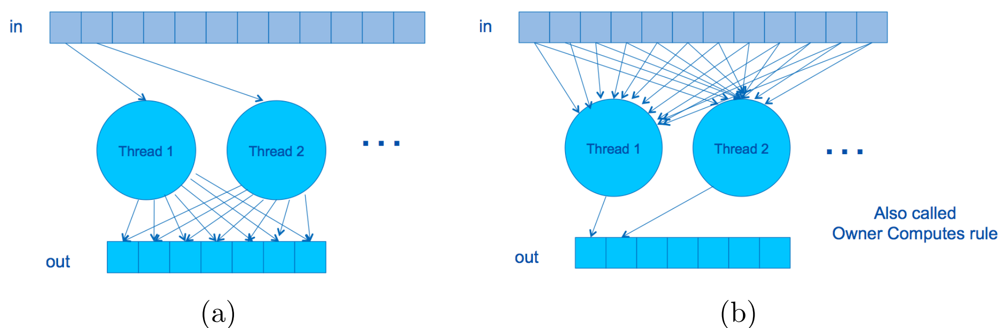


Figure 2.2: Scatter and gather operations. Both operations are equivalent in terms of results. (a) Scatter operation where each thread or Work Item reads a single input and writes partial results to multiple outputs. (b) Gather operation where all threads read all the input and each thread generates a single output.

done to improve performance. Other sparse matrix formats have been proven to be more efficient like a combination of ELL and COO [BG08] formats to make the algorithm to use the GPU resources as much as possible.

2.2.4 Conclusions

As GPU's are relatively cheap, and our algorithm is very data-parallel, we select them as the architecture with which to achieve the interactivity constraint. With the goal of having a portable code that can be deployed in any future architecture, plus being able to compare different present GPU implementations, we decided to use OpenCL in our experiments. Additionally we explored several possible OpenCL implementations to find the fastest and to better understand the GPU behavior.

2.3 AIMparallel Methodology

2.3.1 Introduction

In order for the reader to follow the implementation discussion, we first need to describe the nomenclature we will use. This nomenclature follows the definitions that the AIMparallel methodology (a methodology we created) presents. For that reason we will present AIMparallel in this section.

With AIMparallel we describe in general the problem of parallelization. Working with the algorithm implementation we have found the need for a generalization when describing the analysis of an algorithm and the relation of this analysis with the analysis of the features of the target parallel architecture. For that reason, we created a parallelization

analysis methodology, not only for easing the analysis of algorithms with many possible parallel implementations, but also to more easily document and communicate this analysis.

2.3.2 Definition

In this section we describe the AIMparallel methodology. This methodology is provably already mentally used by many experienced parallel programming users, but we wanted to give it a name and specify it in a generic way to ease the learning curve of all the new incomers on parallel programming, as well for experienced programmers that for the first time have to deal with the fact of having to tune parallel code on different parallel systems.

AIMparallel describes two levels of analysis, as seen in Fig.2.4. A high level that must be the first used, and a low level that needs the information extracted from the high level to be useful. On the high level AIMparallel performs an algorithm analysis on a theoretical system whit non defined resources. In this analysis we start generating parallelizaiton options for that algorithm with the goal of not to restrict too much the options but to see how the algorithm behaves when changing granularity, and memory layouts. This behavior is specified and labelled by three main overheads that we defined (TCO,WPO,TMO), that can be treated as a value that quantifies how much of each one of those overheads each parallelization option generates. On the low level, AIMparallel maps this overheads to the specific characteristics of the programming model and architecture. This mapping gives the information of how much a certain kind of overhead does affect to the overall performance in this parallel system. It is important to note that we use the term overhead as computing cycles lost doing things related to system management or doing nothing because of lack of work balancing or paralelization.

Some of the three proposed overheads are composed of two parts. The amount of overhead produced by the algorithm behavior, and the amount of overhead produced by the system behavior and design. The system overheads could be something that vendors previously benchmark and label with standardized numbers as a reference value. That way, the programmer could skip the tedious part of understanding the inner workings of the architecture needed to infer this overheads. Nevertheless, we propose that the programmer does not exhaustively quantify this overheads (unless it is extremely necessary). In many cases we can find on vendor documentation many information about which are the weaknesses and strengths of each architecture so we can use approximated values or inequalities (greater than, smaller than), that give valuable information to the programmer.

Next we describe the three generic overheads we defined, TCO, WPO and TMO.

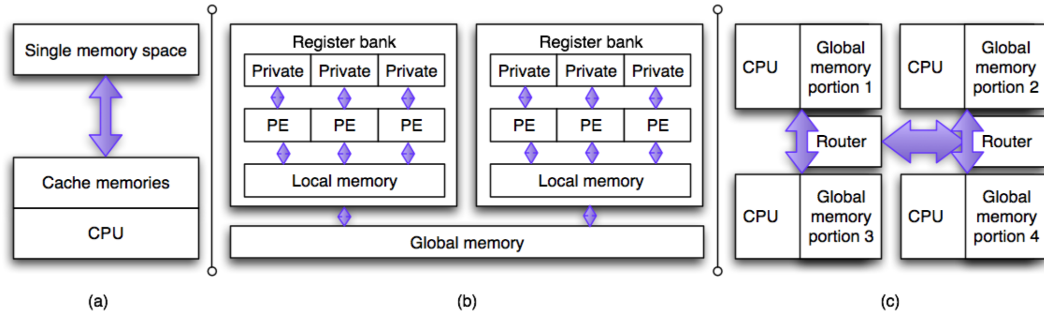


FIGURE 2.3: (a) Single memory space, (b) GPU memory hierarchy, (c) Adapteva's Epiphany network on chip memory hierarchy.

- The first overhead is called *Thread Communication Overhead* (TCO). It expresses the computing cycles lost when performing read/write operations from a thread to any of the memories of the memory hierarchy. From the algorithm side TCO it is affected by the number of memory accesses (algorithm-TCO or aTCO), and the temporal and spacial memory locality on that accesses. This is always affecting the performance on any system, so a measure of greater or lower locality on data accesses will help. On the system side it is affected by many system specificities (system-TCO or sTCO). Parallel and non parallel systems use to have a hierarchical memory system that tries to make the processors work with the low latencies of the small and fast memories, while having access to all the capacity of the higher latency, lower bandwidth big memories. As we can see on Figure 2.3 there are several memory distributions in different architectures or systems. We are also considering distributed systems, where each node has it's own memory space, and nodes communicate with messages through a network. In this case, messages have a big TCO because of both latency and bandwidth. In all the systems, processing elements are reading data from different memories, being they part o a single memory space or not. The programmer, can always manipulate the code to implicitly or explicitly modify the behavior of the accesses in order to reduce their number, so reduce TCO. sTCO then measures how bad is for a given system, that the algorithm has poor data locality, like a factor that increases more or less the aTCO depending on the system features. So, if we quantify this overheads, we could obtain a total value of TCO for a given algorithm parallelization option, and a given parallel system with a simple $TCO = aTCO \times sTCO$.

- The second overhead, named *Wasted Parallelism Overhead* (WPO), relates to the computing cycles lost when the number of working threads is smaller than the number of processors, due to the code or due to synchronization or other questions. If the number of processors of our theoretical system is unlimited, we can always say that, for a parallelization option A with less threads than another option B, option A has $aWPO$ over B. So aWPO is a measure of the granularity

of the algorithm partition. From the system side, sWPO is a measure of how much parallelism can the system afford, either proportionally to the number of cores, or through resources that allow the system to exploit having more active threads than cores. This time, the relationship between aWPO and sWPO is not that clear. Up to a certain point it is beneficial to increase the number of threads, but given a limit (that can depend on the amount of resources used by each thread), increasing thread count can penalize performance. This behavior could be drawn on a 2D graph where y is the performance, and x is the number of threads. This is a function, and that function depends mainly on the system resources. Finding this function and taking into account the resources used per thread is a tedious work for the programmer. A more intuitive way of handling this, that is broadly used, is to read some recommendation about the number of threads that use to be good for each system. Still, we can give some useful clues to the programmer by defining sWPO as a number that reduces aWPO according to the system resources. This time then $sWPO = \frac{1}{SR}$ where SR is a value that represents the system amount of computing resources (number of cores or vendor recommended number of threads) and $aWPO = NT$ where NT represents the number of threads used. Then WPO follows

$$WPO = \begin{cases} \frac{1}{aWPO \times sWPO} & \text{if } NT < SR \\ aWPO \times sWPO & \text{if } NT >= SR \end{cases}$$

This formulation can be used to decide which hardware to use or which algorithm partition or parallelization techniques to use. Then extra-tuning can be done finding the right value of SR for the algorithm by testing the code and different NT values.

- Finally, the *Thread Management Overhead* (TMO) is related to the cycles lost when creating, destroying and scheduling threads. Again, we have algorithm and system TMO (aTMO and sTMO). Using a big number of threads, and creating a destroying a lot of them both create aTMO. Depending on the system implementation of the thread, sTMO will be higher or lower. This time, the relationship of both aTMO and sTMO is direct $TMO = aTMO \times sTMO$. Nevertheless sTMO can vary on some languages since scheduling policies can be different and selected by the programmer.

With this overheads and the methodology to use them, we can select where to put our effort to improve performance when parallelizing a code for an specific system, but also, we can recycle the high level analysis for considering implementations in other architectures with some work already done. Once the programmer has chosen what to improve, then he has to focus on the specific architecture, and all those characteristics that affect to the most relevant overheads found.

Also, this three overheads could be used by vendors to provide very useful information about the specifics of the performance of their systems, helping customers to find the perfect fit for their needs, a not very trivial question.

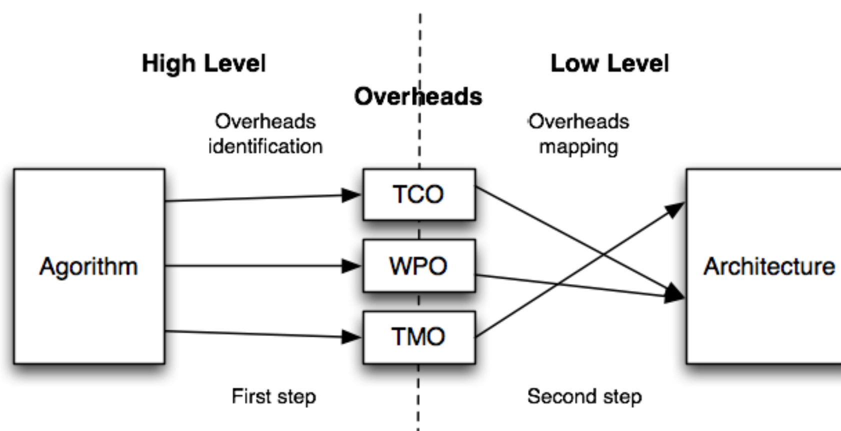


FIGURE 2.4: Two step methodology for analyzing parallelization options for a given algorithm and a given parallel architecture, using the three pre-defined parallelization overheads.

2.3.3 Low level overheads mapping

In this section we will describe the influence of each of the three overheads on specific architectures. This step is mandatory and implies the study of the architectures being used. This could be simplified if vendors could provide some standardized numerical values for each of the three overheads, that allow to more easily compare parallelism characteristics of different systems.

2.3.3.1 GPU overheads mapping

On this section we will comment on the three system overheads one by one, and add some discussion at the end.

- sTCO: The use of the GPU memory hierarchy is the most sensitive aspect for obtaining a good GPU performance [KH10]. The communication between work items (threads) is done through this memory hierarchy. Therefor, on GPU's, *sTCO* is caused by big memory access latencies, and it is the programmers duty to avoid it as much as possible by properly and explicitly using the memory hierarchy. Nevertheless, latencies and bandwidth are much higher than for instance distributed systems. Despite that, once the parallelization option is selected, if a programmer wants to optimice a code for GPU he can approximate the number of cycles lost,

N_{cl}, when introducing a memory access to a specific memory of the memory hierarchy. This value will be a worst case, since it doesn't take into account caches or latency hiding as in Equation 2.1.

$$N_{cl} = M_{ao} \times N_a \times C_f \tag{2.1}$$

M_{ao} is an average of the cycles lost per memory access; N_a is the number of work items that perform the memory access; and C_f is the coalescing factor defined as $\frac{1}{nWI}$, where nWI is the maximum number of work items that can perform a memory access operation at the same time (coalesced), for that memory access.

- sWPO: on GPU's sWPO is very big since they have a lot of cores to be used, and the total amount of computing power depends on having a lot of threads working. In fact, $SR > numberofcores$, since for exploiting all the cores for the maximum number of cycles on a program execution, it is necessary to have much more threads than GPU cores. It is important to note that GPU threads are very different to CPU threads, since there is no OS running on the GPU, no processes or threads as data structures that identify streams of code. The threads exposed on the programming model are translated into SIMD data positions.

- sTMO: on GPU's threads don't exists as data estructures, so there is no real thread management. The conseqences of uncoalesced reads and thread divergence could be considered sTMO, but they are conceptually related to both WPO and TCO. The real sTMO is the situation where the code for each thread is very big, and some of the instructions need to be on Global Memory (See Section 6.2.3). Of course, reading instructions from global memory instead of register banks is slower. Another consequence of big GPU threads is the reduction on the number of work items active on each GPU multiprocessor. This reduces memory latency hiding. Also, new technologies like NVIDIA dynamic parallelism allow to create and destroy more threads, action that has a penalty on performance. Another way to change the number of threads on a GPU program is to write differente kernels with different NDRange sizes. The only way to communicate from one kernel to the other is to store the data on global memory, something slower than keeping the data on faster memories. So we know now when sTMO increases on GPU's.

We have now mapped the GPU architecture main characteristics to three concepts that we can apply to any other parallel architecture, so the generic analysis becomes useful. We will show that it is useful to categorize the performance problems, and have abstract and common terms to refer to them across different architectures to ease the parallelization analysis and discussion.

2.3.3.2 Overheads mapping for CPU's with OpenMP

As standard CPUs have a limited amount of cores (from 2 to 16 per chip), one of the main overheads is the WPO. For this reason, we have to reduce the number of threads to close to the number of cores, sometimes more and sometimes a bit less, depending on the runtime management and DMA's. Also, even there are fewer cores there can be WPO because of work unbalance. So we have to analyze the code behavior to determine which OpenMP for loop parallelization scheduling is better in order to avoid WPO and don't generate too much TMO. In the case of TCO, we have to think of data locality. There are no local memories on the CPU, but we can increase the cache performance by increasing the spacial and temporal data locality of the operations in the code, so we will use more times cache blocks before replacing them with different data. Instead of explicit TCO optimization like in GPU's we have an implicit optimization.

2.3.3.3 Overheads mapping for CPU's with GCD

GCD [App09] is an API developed by Apple based on C Blocks, that allows the programmer to forget about the computer resources, and focus only on the sequentiality or asynchronicity of each part of the code, regarding the main thread of the program.

GCD at the programmer level substitutes the threads with queues, in where we can enqueue Blocks of code that can run asynchronously with the main thread. This queues are data structures much lighter than threads, that are accessed by few system controlled threads. The threads are responsible of searching for Blocks on these queues. This minimizes TMO and WPO by parallelizing the work scheduling for balancing. As the active threads are controlled by the system, the TMO is minimized automatically without the intervention of the programmer. The programmer does not need to analyze which is the perfect number of threads for a system, because the system is managing it. In fact, GCD can be aware of the active threads belonging to other applications, in order to reduce the number of active threads for the application using blocks, in order to reduce the number of context switches, and improving overall system performance. Nevertheless, this last feature is not affecting our tests, since we tested the execution times in systems running the minimum set of idle system processes and daemons and our code.

GCD can further reduce the WPO by using two types of queues, a global concurrent queue and an undefined number of FIFO queues. FIFO queues can be used in order to ensure the execution order of Blocks, when there are dependencies between them. That way, a thread can take for execution the first Block in the concurrent queue that still has been taken, and the first block introduced in any of the FIFO queues as long there is no thread executing the previous Block before it. This increases the probability for a thread to find work to do, and the source of work can be any part of the code. Again,

we are not taking advantage of this feature in our code, but we can if we integrate it with a visualization interface in an Apple machine.

Finally, GCD allows to reduce TMO (consider block enqueueing as TMO), by allowing blocks to enqueue other blocks. That way it is possible to parallelize block enqueueing, so to balance this task across threads, instead of giving the task only to the main thread. The more cores used, the more interesting this option becomes.

As a summary we can see, following the GCD specification and our discussion, that the codes (with unbalanced work) that use it do better scale with CPU core increments than free OpenMP implementations.

2.3.4 Conclussions

With all this information we can intuitively see which algorithm overheads we should try to minimize, for a specific hardware. An experienced parallel programmer may already mentally do that procedure, but we didn't find any literature explaining a generic methodology for avoiding redundant and useless work, when programming parallel systems.

Chapter 3

Classification problem

3.1 Introduction

In volume visualization, the definition of the regions of interest is inherently an iterative trial-and-error process finding out the best parameters to classify and render the final image. Generally, the user requires a lot of expertise to analyze and edit these parameters through multi-dimensional transfer functions. There are paid manual classification services that give good results, but are not available to any one due to the cost of the service. Additionally, the time required for having the volumes classified usually is several weeks. Additionally, classification techniques applied to large volumes are usually computational intensive tasks, that require overnight computation and make the whole user-system interaction slow. So there is not a tool where the users of CT or MRI machines can easily and accurately visualize the results in real time.

We want to create a framework that defines the steps to be performed, and on each step or functionality, the methods implemented can be varied and changed. But there are fixed constraints, as the interactivity feature and a minimum of classification accuracy that must be satisfied.

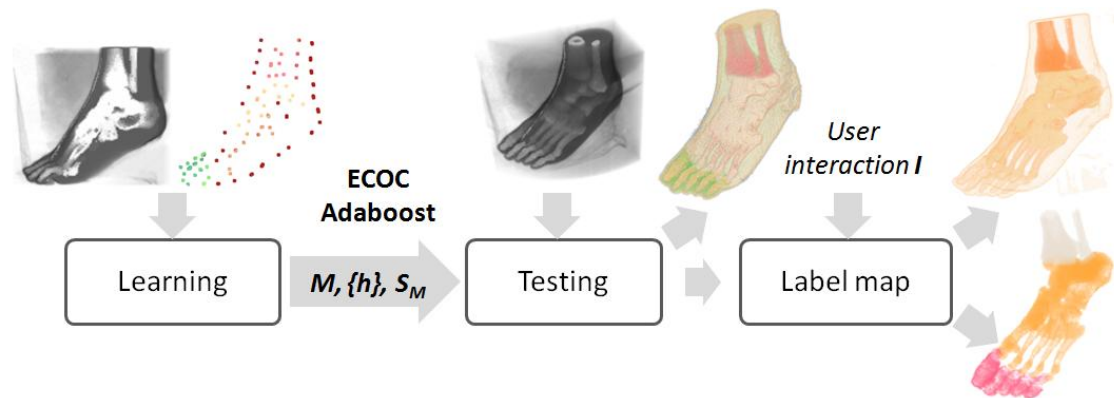


Figure 3.1: Overview of the ECOC-based visualization framework.

In this work we want to implement supervised multi-classification techniques, that can generate classifiers using a small labeled subset of the volume to be classified. The idea is to create a software that allows to create a tool for the CT/MRI user (a doctor) to be able to interact with the volume generated, by labeling few points that the doctor will be sure about, and the rest will be classified automatically in real time.

For the interactivity constraint, we consider parallelism as the conceptual computing tool in which to rely for execution time performance. Given the data-parallelism nature of the data to be processed, we will explore several parallel computing systems (hardware+language) and select classification techniques that can offer different work division granularities to better exploit any possible parallel architecture available to us.

3.2 Visualization

Knowledge expressiveness of scientific data is one of the most important visualization goals. The abstraction process the final user should carry out in order to convey relevant information in the underlying data is normally a difficult task. User has to mentally process a large amount of data contained in several hundreds of slices to find features, boundaries between different structures, regions surrounding pathological areas, semantic structures, and so on. During the last few decades new means of outlining significant features are being developed to gather visual information contained in the data, such as ghost views, Focus+Context approaches [APT08], importance-driven visualizations and automatic viewpoint selections. Some of these methods [VKG05] [KSW06] are useful for the exploration of pre-classified data sets as well as non-classified ones. However, most of them [BHW*07, KBKG08] require to previously define the structures of interest.

In volume rendering literature, many papers addressed classification by directly associating optical and importance properties to the different data values considering their belonging to a particular structure [PLB*01]. Most of them are based on the edition of transfer functions (TF) [BPS97, KD98]. One-dimensional TFs only take into account scalar voxel values, but in some cases they could fail at accurately detecting complex combinations of material boundaries. Multi-dimensional TFs consider vectorial values or combinations of local measures of scalar values (e.g. position, statistical signatures, or derivatives [KD98]). However, the design complexity and the memory requirements increase with the TFs dimensionality. In general, the user coherently assigns similar optical properties to data values corresponding to the same region. Selection of regions or structures is indirectly defined by assigning to zero the opacity since totally transparent samples do not contribute to the final image. Then, the manual TF's definition even by skilled users becomes complicated. In this sense, many works have focused on developing user friendly interfaces that make this definition more intuitive. Special emphasis has been done in the design of interfaces that deal with the definition of a

TF or partially automatize it [TM04, KKH01, MAB*97]. Nevertheless, to recognize semantic structures that apply to identify additional semantic information requires more sophisticated techniques.

3.3 Classification techniques

Automatic and user-guided segmentation strategies based on image processing are used to obtain classified data sets. Recently, some preliminary works using learning methods have been published based on data driven and on image-driven classification. These classification methods provide users with a high level of information about data distribution and about the final visualization. Supervised methods such as bayesian networks, neural networks [TLM03], decision trees [FPT06] and non-supervised methods [TM04] have been applied in different user interfaces of volume applications. For instance, in [GMK*92], clustering-based supervised and non-supervised learning methods are compared for classifying magnetic resonance data. An integration of interactive visual analysis and machine learning is used in an intelligent interface to explore huge data sets in [FWG09]. Still, the data driven classification problem as a pattern recognition process is an open issue that has been treated from different points of view: template matching, statistical, syntactic or structural, and neural [JDM00]. For instance, supervised statistical learning deals with the classification task by modeling the conditional probability distribution of the different pre-labelled data sample features.

Different Machine Learning (ML) approaches have been recently implemented using GPGPU for binary classifications in image processing applications. Clustering strategies and the computation of a k-nearest neighbor similarity classifier is presented in [GDB08]. A Geometrical Support Vector Machine classifier has also been implemented using GPGPU [HWS10]. It extends different GPGPU implementations for Neural Networks [YSMR10]. Adaboost is also a widely applied classifier. Based on a *weak classifier*, Adaboost defines an additive model combining simple weak classifiers to define a strong binary classifier with high generalization capability. Given its inherent parallel structure, its high performance, and its simplicity in order to train –Adaboost does not require tuning classifier parameters, Adaboost has a high potential for GPU applications.

Most of the previous approaches are binary by definition –they only learn to split from two possible labels[1]. In order to deal with multi-class labelling, they need to be combined in some way, for example, by means of a voting or a committee process. In this scope, the Error-Correcting Output Codes (ECOC) is widely applied as a general framework in the ML community in order to deal with multi-class categorization problems. The ECOC framework allows to combine any kind of classifiers, improving classification accuracy by correcting errors caused by the bias and the variance of the learning algorithm [DK95].

[1]Note that we use the terms classification and labelling indistinctly to refer the assignment of labels to data samples.

3.4 Proposal

Our automatic volume labeling system bases on the combination of a set of trained binary classifiers. We consider the general ECOC framework to deal with multi-class labeling. As a case study, we use Adaboost to train the sets of binary classifiers. Next, we briefly review these two methodologies.

3.4.1 Error-Correcting Output Codes

Given a set of N classes (volume structures or regions with certain properties) to be learnt in an ECOC framework, n different bi-partitions (groups of classes) are formed, and n binary problems over the partitions are trained. As a result, a codeword of length n is obtained for each class, where each position (bit) of the code corresponds to a response of a given classifier h (coded by +1 or -1 according to their class set membership, or 0 if a particular class is not considered for a given classifier). Arranging the codewords as rows of a matrix, we define a *coding matrix* M, where $M \in \{-1, 0, +1\}^{N \times n}$. Fig. 3.2(a) and (b) show a volume data set example and a coding matrix M, respectively. The matrix is coded using 15 classifiers $\{h_1, ..., h_{15}\}$ trained using a few voxel samples for each class of a 6-class problem $\{c_1, ..., c_6\}$ of respective codewords $\{y_1, ..., y_6\}$. The classifiers h are trained by considering the pre-labelled training data samples $\{(\rho_1, l(\rho_1)), ..., (\rho_k, l(\rho_k))\}$, for a set of k data samples (voxels in our case), where ρ is a data sample and $l(\rho_k)$ its label. For example, the first classifier h_1 is trained to discriminate c_1 against c_2, without taking into account the rest of classes. Some standard coding designs are one-versus-all, one-versus-one, and random [ASS02]. Mainly, they differ on the definition of the subgroups of classes in the partitions of each binary problem. Due to the huge number of bits involved in the traditional coding strategies, new problem-dependent designs have been proposed [ETP*08]. These strategies take into account the distribution of the data in order to define the partitions of classes of the coding matrix M.

During the decoding or testing process, applying the n binary classifiers, a code X is obtained for each data sample ρ in the test set. This code is compared to the base codewords ($y_i, i \in [1, .., N]$) of each class defined in the matrix M, and the data sample is assigned to the class with the *closest* codeword (e.g. in terms of distance). In fig. 3.2(b), the new code X is compared to the class codewords $\{y_1, ..., y_6\}$ using the Hamming Decoding [ASS02], $HD(X, y_i) = \sum_{j=1}^{n}(1 - \mathsf{sign}(x^j \cdot y_i^j))/2$, where X^j corresponds to the j-th value of codeword X, and the test sample is classified by class c_1 with a measure of 0.5. The decoding strategies most widely applied are Hamming and Euclidean [ASS02], though other decoding designs have been proposed [EPR10].

3.4.2 Adaboost classifier

In this work, we train the ECOC binary classifiers h using Adaboost classifier. Adaboost is one of the main preferred binary classifiers in the ML community based on the concept of Boosting [FHT98]. Given a set of k training samples, we define $h_i \sim F_i(\rho) = \sum_{m=1}^{\mathcal{M}} c_m f_m(\rho)$, where each $f_m(\rho)$ is a classifier producing values ± 1 and c_m are constants; the corresponding classifier is $\text{sign}(F(\rho))$. The Adaboost procedure trains the classifiers $f_m(\rho)$ on weighed versions of the training sample, giving higher weights to cases that are currently misclassified [FHT98]. Then, the classifier is defined to be a linear combination of the classifiers from each stage. The binary Discrete Adaboost algorithm used in this work is shown in Algorithm 3. E_w represents expectation over the training data with weights $w = (w_1, w_2, .., w_k)$, and $1_{(S)}$ is the indicator of the set S (1 or 0 if S is or not satisfied). Finally, Algorithm 2 shows the testing of the final decision function $F(\rho) = \sum_{m=1}^{\mathcal{M}} c_m f_m(\rho)$ using Adaboost with Decision Stump "weak classifier". A Decision Stump is a simple directional threshold over a particular feature value. Each Decision Stump f_m fits a threshold value T_m and a polarity (directionaly over the threshold) P_m over the selected m-th feature. In testing time, ρ^m corresponds to the value of the feature selected by $f_m(\rho)$ on a test sample ρ. Note that c_m value is subtracted from $F(\rho)$ if the classifier $f_m(\rho)$ is not satisfied on the test sample. Otherwise, positive values of c_m are accumulated. Finally decision on ρ is obtained by $\text{sign}(F(\rho))$.

ALGORITHM 1: Discrete Adaboost training algorithm.

1: Start with weights $w_i = 1/k, i = 1, .., k$.
2: Repeat for $m = 1, 2, .., (M)$:
  (a) Fit the classifier $f_m(\rho) \in -1, 1$ using weights w_i on the training data.
  (b) Compute $\text{err}_m = E_w[1_{(l(\rho) \neq f_{m(\rho)})}], c_m = \log((1 - \text{err}_m)/\text{err}_m)$.
  (c) Set $w_i \leftarrow w_i \exp[c_m \cdot 1_{(l(\rho_i) \neq f_{m(\rho_i)})}], i = 1, 2, .., k$, and normalize so that $\sum_i w_i = 1$.
3: Output the classifier $F(\rho) = \text{sign}[\sum_{m=1}^{\mathcal{M}} c_m f_m(\rho)]$.

ALGORITHM 2: Discrete Adaboost testing algorithm.

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

3.5 Proposal: General framework for multi-class volume labeling

Here, we present our automatic system for multi-class volume labeling. The system performs the following stages: a) All-pairs ECOC multi-class learning, b) ECOC submatrix definition, c) Adaptive decoding, and d) Label mapping.

3.5.1 All-pairs multi-class learning

Given a set of pre-labelled samples for each volume structure, we choose the one-versus-one ECOC design of $N(N-1)/2$ classifiers to train the set of all possible pairs of labels. An example of a one-versus-one ECOC coding matrix for a 6-class foot problem is shown in Fig. 3.2(a) and (b). The positions of the coding matrix M coded by +1 are considered as one class for its respective classifier h_j, and the positions coded by -1 are considered as the other one. For example, the first classifier is trained to discriminate c_1 against c_2; the second one classifies c_1 against c_3, etc., as follows:

$$h_1(x) = \begin{cases} 1 & \text{if } X \in \{c_1\} \\ -1 & \text{if } X \in \{c_2\} \end{cases}, .., \quad h_{15}(x) = \begin{cases} 1 & \text{if } X \in \{c_5\} \\ -1 & \text{if } X \in \{c_6\} \end{cases} \tag{3.1}$$

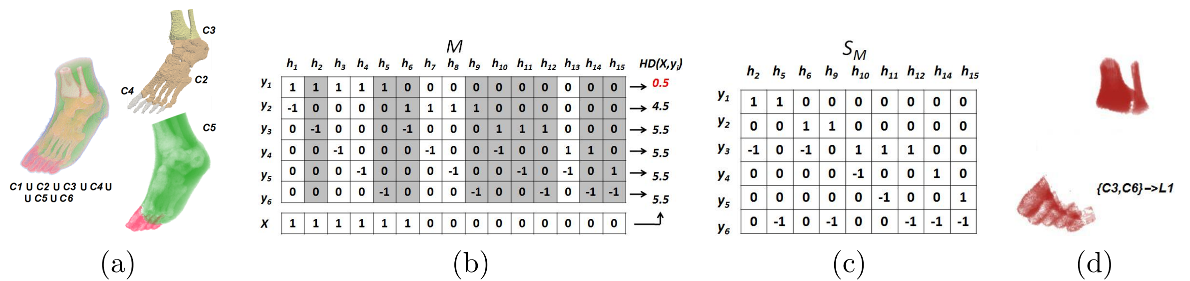


FIGURE 3.2: (a) True labels for a foot volume of six classes; (b) One-versus-one ECOC coding matrix M for the 6-class problem. An input test codeword X is classified by class c_1 using the Hamming Decoding; (c) Submatrix S_M defined for an interaction set $I = \{\{c_1, c_2, c_4, c_5\}, \{c_3, c_6\}\}$; (d) Visualization in the new label space.

The selection of the one-versus-one design has two main benefits for our purpose. First, though all pairs of labels have to be split in a one-versus-one ECOC design, the individual problems that we need to train are significantly smaller compared to other classical ECOC classifiers (such as one-versus-all). As a consequence, the problems to be learnt are usually easier since the classes have less overlap. Second, as shown in Fig. 3.2, considering binary problems that split individual labels allow us for combining groups of labels on-demand just by the selection of a subgroup of previouly trained classifiers, as we show next.

3.5.2 ECOC submatrix definition

Given a volume that can be decomposed into N different possible labels, we want to visualize in rendering time the set of labels requested by the user. For this purpose, we use a small set of ground truth voxels described using spatial, value, and derivative features to train the set of $N(N-1)/2$ Adaboost binary problems that defines the one-versus-one ECOC coding matrix M of size $N \times n$. Then, let us define the interaction

of the user as the set $I = \{I_0, .., I_z\} = \{\{c_i, .., c_j\}, ..\{c_k, .., c_l\}\}$, where $I, |I| \in \{1, .., N\}$ is the set of groups of labels selected by the user, and I_0 contains the background (always referred as c_1) plus the rest of classes not selected for rendering, $I_0 = \{c_i\}, \forall c_i \notin \{I_1, .., I_z\}, \bigcup_{c_i \in I} = \{c_1, .., c_N\}, \bigcap_{c_i \in I} = \emptyset$. Then, the submatrix $S_M \in \{-1, 0, +1\}^{N \times Z}$ is defined, where $Z \leq n$ is the number of classifiers selected from M that satisfies the following constraint,

$$h_i | \exists j, M_{ji} \in \{-1, 1\}, c_j \in I \setminus I_0 \tag{3.2}$$

For instance, in a 6-class problem of 15 one-versus-one ECOC classifiers (see Fig. 3.2(b)), the user defines the interaction $\{c_3, c_6\}$, resulting in the interaction model $I = \{\{c_1, c_2, c_4, c_5\}, \{c_3, c_6\}\}$ in order to visualize two different labels, one for background and other one for those voxels with label c_3 or c_6. Then, from the original matrix $M \in \{-1, 0, +1\}^{6 \times 15}$, the submatrix $S_M \in \{-1, 0, +1\}^{6 \times 9}$ is defined, as shown in Fig. 3.2(c). Note that the identifier i of each classifier h_i in S_M refers to its original location in M.

3.5.3 Adaptive decoding

The proposed ECOC submatrix S_M encodes the minimum required number of binary classifiers from a one-versus-one coding matrix M to label the sets of structures defined by the user in I. However, this novel definition of ECOC submatrices requires a readjustment of the decoding function δ applied. For instance, if we look at the matrix M of Fig. 3.2(b), one can see that each row (codeword) of M contains the same number of positions coded by zero. If one applies a classical Hamming decoding, obviously a bias in the comparison with a test and a matrix codeword is introduced to those positions comparing membership $\{-1, 1\}$ to 0. Note that the zero value means that the class has not been considered, and thus, it makes no sense to include a decoding value for those positions. However, since in the one-versus-one design all the codewords contain the same number of zeros, the bias error introduced for each class is the same, and thus, the decoding measurements using classical decoding functions are comparable. In our case, this constraint does not hold. Look at the submatrix S_M of Fig. 3.2(c). The selection of submatrices often defines coding matrices with different number of positions coded to zero for different codewords. In order to be able to successfully decode a submatrix S_M we take benefit from the recent Loss-Weighted proposal [EPR10], which allows the decoding of ternary ECOC matrices avoiding the error bias introduced by the different number of codeword positions coded by zero. The Loss-Weighted decoding is defined as a combination of normalized probabilities that weights the decoding process. We define a weight matrix M_W by assigning to each position of the codeword codified by $\{-1, +1\}$ a weight of $\frac{1}{n-z}$, being z the number of positions of the codeword coded by zero. Moreover, we assign a weight of zero to those positions of the weight matrix M_W that contain a zero in the coding matrix M. In this way, $\sum_{j=1}^{n} M_W(i, j) = 1, \forall i = 1, ..., N$. We assign to each position (i, j) of a performance matrix H a continuous value that corresponds

to the performance of the classifier h_j classifying the samples of class c_i as shown in eq. 3.3. Note that this equation makes H to have zero probability at those positions corresponding to unconsidered classes. Then, we normalize each row of the matrix H so that M_W can be considered as a discrete probability density function (eq. 3.5). In fig. 3.3, a weight matrix M_W for a 3-class toy problem of four classifiers is estimated. Fig. 3.3(a) shows the coding matrix M. The matrix H of Fig. 3.3(b) represents the accuracy of the classifiers testing the instances of the training set. The normalization of H results in a weight matrix M_W shown in Fig. 3.3(c). Once we compute the weight matrix M_W, we include this matrix in a loss-function decoding formulation, $L(\theta) = \mathbf{e}^{-\theta}$, where θ corresponds to $y_i^j \cdot f(\rho, j)$, weighted using M_W, as shown in eq. 3.6. The summarized algorithm is shown in table 3.1.

Loss-Weighted strategy: Given a coding matrix M,

1) Calculate the performance matrix H,

$$H(i,j) = \frac{1}{m_i}\sum_{k=1}^{m_i} \varphi(h^j(\rho_k^i), i, j) \tag{3.3}$$

$$\text{based on} \quad \varphi(x^j, i, j) = \begin{cases} 1, & \text{if} \quad X^j = y_i^j, \\ 0, & \text{otherwise.} \end{cases} \tag{3.4}$$

2) Normalize H: $\sum_{j=1}^{n} M_W(i,j) = 1, \quad \forall i = 1, ..., N$:

$$M_W(i,j) = \frac{H(i,j)}{\sum_{j=1}^{n} H(i,j)}, \quad \forall i \in [1, ..., N], \quad \forall j \in [1, ..., n] \tag{3.5}$$

3) Given a test data sample ρ, decode based on,

$$\delta(\rho, i) = \sum_{j=1}^{n} M_W(i,j) L(y_i^j \cdot f(\rho, j)) \tag{3.6}$$

TABLE 3.1: Loss-Weighted algorithm.

3.5.4 Adaboost Look up table (LUT) representation

We propose to define a new and equivalent representation of c_m and $|\rho|$ that facilitate the parallelization of the testing. We define the matrix $V_{f_m(\rho)}$ of size $3 \times (|\rho| \cdot M)$, where $|\rho|$ corresponds to the dimensionality of the feature space. First row of $V_{f_m(\rho)}$ 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 $V_{f_m(\rho)}$ contains the value c_m for the feature $mod(i, |\rho|)$ if $mod(i, |\rho|) \neq 0$ or $|\rho|$, otherwise. The next value of c_m for that feature is found in position $i + |\rho|$. The positions corresponding to features not considered during training are set to zero. The second and third rows of $V_{f_m(\rho)}$ for column i contains the values of P_m and T_m for the corresponding Decision Stump. Note that in the representation of $V_{f_m(\rho)}$ we lose 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 represented, and thus, the final additive decision

model $F(\rho)$ is equivalent. In our proposal, each "weak classifier" is codified in a channel of a 1D-Texture.

3.5.5 Label mapping

Given the submatrix S_M and the user interaction model I, after classification of a voxel ρ applying the Loss-Weighted decoding function, the obtained classification label $c_i, i \in \{1, ..N\}$ is relabelled applying the mapping,

$$L_M(I, c_i) = \begin{cases} l_1 & \text{if } c_i \in I_1 \\ ... & \\ l_z & \text{if } c_i \in I_z \end{cases}$$

where $l_i, i \in \{1, .., z\}$ allows to assign RGBα or importance values to all voxels that belong to the corresponding selected classes in I_i. These functions are useful to provide flexibility to our framework in order to be applied in several visualization tasks, such as importance-driven visualizations or automatic viewpoint selections. As an example, applying the interaction model $I = \{\{c_1, c_2, c_4, c_5\}, \{c_3, c_6\}\}$ for the 6-class problem of Fig. 3.2(a), we obtain the submatrix S_M of Fig. 3.2(c). Applying the Loss-Weighted decoding over S_W, and the mapping function $\{\{c_1, c_2, c_4, c_5\}, \{c_3, c_6\}\} \longrightarrow \{0, 1\}$, the new volume representation is shown n Fig. 3.2(d).

$$M = \begin{bmatrix} 1 & 1 & -1 & 0 \\ 1 & -1 & 0 & 0 \\ -1 & 1 & 1 & -1 \end{bmatrix} \quad H = \begin{bmatrix} 0.955 & 0.955 & 1.000 & 0.000 \\ 0.900 & 0.800 & 0.000 & 0.000 \\ 1.000 & 0.905 & 0.805 & 0.805 \end{bmatrix} \quad M_W = \begin{bmatrix} 0.328 & 0.328 & 0.344 & 0.000 \\ 0.529 & 0.471 & 0.000 & 0.000 \\ 0.285 & 0.257 & 0.229 & 0.229 \end{bmatrix}$$

(a) (b) (c)

FIGURE 3.3: (a) Coding matrix of four classifiers for a 3-class toy problem, (b) Performance matrix, and (c) Weight matrix.

3.6 Classification parallelization implementation

3.6.1 Introduction

In this section, we will explain in detail which part of the framework we have parallelized and how we have done it. We will describe some possible parallelizations and select one following the AIMparallel methodology. Additionally we will implement one of the discarded implementations to compare the results and verify.

The slowest part on the framework is the classification, where depending on the number of classes and voxels on the volume, can last about 5 hours on Matlab. For that reason, on this thesis we parallelized the classification step of the framework, where many binary

Adaboost classifiers are combined using an ECOC matrix, for obtaining labels from more than two clases for each boxel.

3.6.2 Algorithm overview

In this section we describe the algorithm being parallelized, and the tasks it can be decomposed into.

As shown in Algorithm 3, the critical section in our application is a triple iteration loop to go through all voxels of the voxel model, VM, in an ordered traversal. Each iteration performs three tasks: the gradient computation (Task 1), the X code word estimation (Task 2), and the final labeling calculation (Task 3). These tasks are called $T1$, $T2$ and $T3$, respectively in Algorithm 3, and are performed for each voxel v_{xyz}.

ALGORITHM 3: Critical section serial pseudocode for the testing stage.

inputVolume : Original 3D voxel model with density values (d)
outputVolume : Multiclass labeled voxel model with single value samples
voxelFeatures : pointer to the 8 sample features for the voxel sample being processed
$\mathcal{L}$: $\mathcal{L}$ matrix pointer containing the LUTs
X : code word of binary decission values for a single voxel sample
M : coding matrix
background_value: density value threshold for the actual voxel sample to be processed

```
for z ← 0 to dim_z do
    for y ← 0 to dim_y do
        for x ← 0 to dim_x do
            d ← input[z][y][x] // d refers to the density value in the voxel model
            if d > background_value then
                computeGradient(z, y, x, inputVolume[z][y][x], voxelFeatures); // T1
                XCodeEstimation(voxelFeatures, ℒ, X); // T2
                FinalLabeling(X, M, outputVolume[z][y][x]); // T3
            end
        end
    end
end
```

Specifically, the features of each v_{xyz} are computed and stored in memory (pointed by the voxelFeatures variable), at each iteration in the *computeGradient* function ($T1$). Then, at the same iteration, a X code word of binary decision values is generated by *XCodeEstimation* function ($T2$) for the actual v_{xyz} voxel. In this function, the $voxelFeatures$ and the $\mathcal{L}$ variables are read to generate the contents of X. In order to improve serial execution, we modified the Discrete Adaboost testing algorithm (see Algorithm 2). The features are computed depending on the P_m values (lines 3(a) - 3(b)), but the Adaboost Look Up Table allows to compute all the features without branching depending on the weight. Then we are substituting conditional statements by more efficient arithmetical and logical operations. So, in our algorithm, lines 3(a) and 3(b) are substituted by Equation 3.7.

$$F(\rho) = F(\rho) + c_m \cdot ((2 \cdot (P_m \cdot \rho^m < P_m \cdot T_m)) - 1) \tag{3.7}$$

Finally, before changing to the next voxel, the $FinalLabeling$ function ($T3$) reads X and the coding matrix, M, to generate a single class value for v_{xyz}.

3.6.3 Generic parallelization proposals

In this section, we follow the methodology described in the theoretical environment we defined, for each one of the tasks involved in the Discrete Adaboost testing algorithm (see Section 3.6.2). At the end of the section, we comment the overheads for the interactions between tasks (T1, T2 and T3).

3.6.3.1 Task 1 and high level overheads mapping

First we analyze the parallelization possibilities for Task 1 ($T1$), which is devoted to the stencil gradient calculation, as we can see on Figure 3.4. In this task, the gradients g_x, g_y and g_z of a sampled voxel v_{xyz}, as well as its magnitude $|g|$, are computed using central differences. $T1$ is a 7 point stencil operation. Thus, for each dimension, we need the two neighbor voxel values of v_{xyz}. This operation is data independent among voxel samples, because the source data is not modified, and the results are 8 output-only values per voxel.

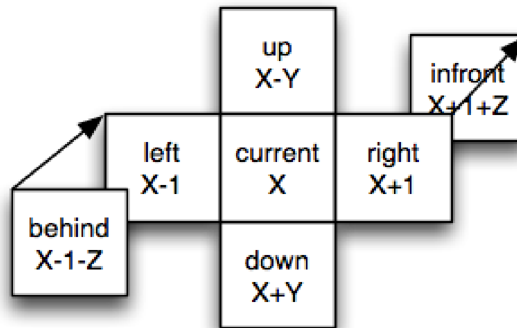


Figure 3.4: Stencil operation used on T1. The neighbors indicated are used for generating extra feature values of X.

As a first parallelization option $T1_1$ we could use a thread for each voxel and calculate the gradients for all of them concurrently. This generates TMO but less than having three threads per voxel, that in addition to more TMO, they would generate TCO and WPO on the reduction operations.

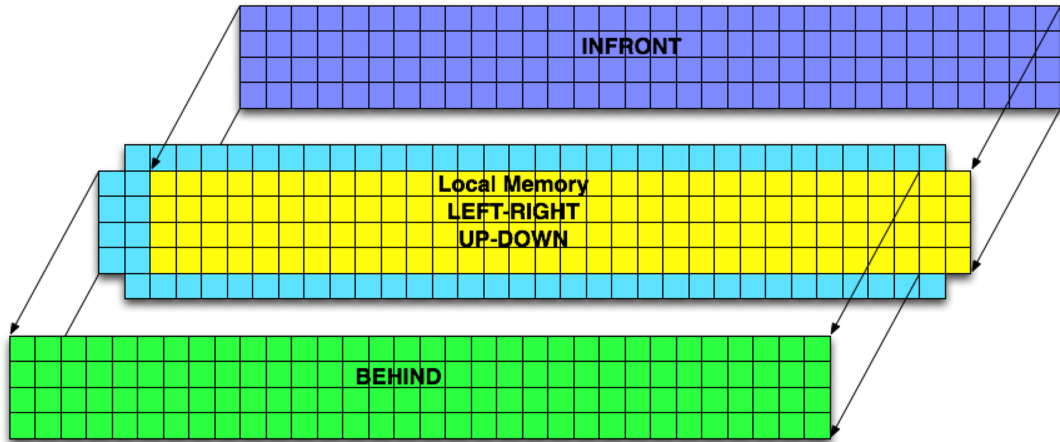


FIGURE 3.5: $T1_1$ implementation. On yellow the voxels for which we are calculating the gradients. Infront and behind are stored on private memory, and yellow and blue on local memory. The image corresponds to the data used by a workgroup of 32×4 work items.

As $T1_2$ option we can then focus on data locality to reduce TCO. The only way to do so is by making threads to process more than one voxel, specifically voxels that share data for calculating the gradients. This reduces the number of threads, therefore it produces WPO but also reduces TMO and TCO. We can find in the literature several methods for reducing TCO in this way like the Semi-stencil algorithm [dlCAPC10] that exploits data locality. In our case, as we are using only one value per neighboring side, and we are not using the central value, we can only do operations for the voxel corresponding to the central value. Thus this technique is not useful. Depending on the architecture we will calibrate the number of threads taking into account this analysis.

3.6.3.2 Task 2 and high level overheads mapping

Next, we propose several parallelization options for Task 2 ($T2$), which performs the binary-classification step. $T2$ refers to the testing or decoding process, where the corresponding LUTs of the n binary classifiers are applied to each voxel v_{xyz} (see Section 3.4.1).

As we defined in Section 3.5.4, each $\mathcal{L}$ has three $(|\rho| \times W_c)$-sized rows. Let $dim(VM)$ be the total amount of voxels of the model. Thus, the $T2$ cost is $O(dim(VM) \times n \times (|\rho| \times W_c))$. We propose multiple options for parallelizing $T2$:

- First one is called $T2_1$ and it consists of creating a thread for each voxel v_{xyz}, as shown in Fig. 3.6. Ideally, in a machine with $dim(VM)$ processors, the final cost of this option is $O(n \times (|\rho| \times W_c))$. Among the three values $dim(VM)$ will always

be greater than n or $(|\rho| \times W_c)$. Also, there is no thread to thread communication so we have a balance between parallelism (WPO) and communication (TCO), and also thread management (TMO).

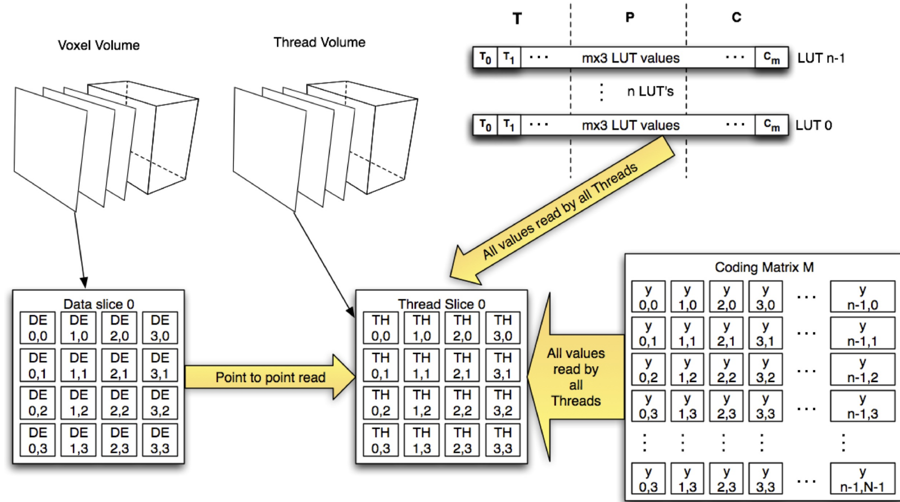


FIGURE 3.6: Parallelization option $T2_1$: a thread is created for each voxel v_{xyz}.

- $T2_2$ is a second parallelization option of Task 2. In this option we increase parallelism, as shown in Fig. 3.7. We can create $dim(VM) \times |\rho| \times W_c)$ threads. One thread computes a value for each LUT-entry for each voxel (see Fig. 3.7). The theoretical cost of this approach is $O(n)$. However, threads that classify the same voxel v_{xyz} should communicate in order to compute the final sign $sign(F(\rho))$ for v_{xyz}. This behavior increases the thread communication overhead TCO defined in Section 2.3.2. Moreover, this option also has WPO and TMO because there are different numbers of threads being used on each step of $T2$. So the increased parallelism is only effective in some steps of $T2_2$.

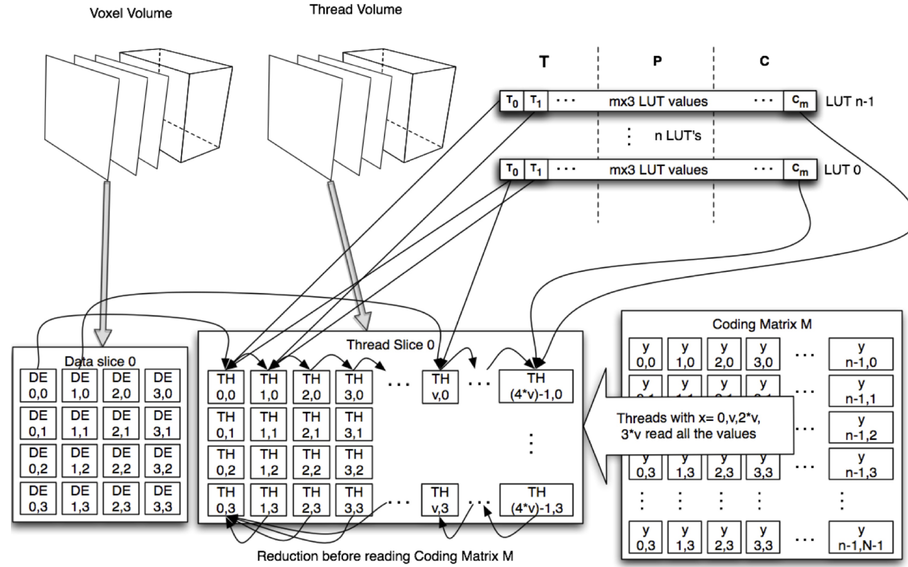


FIGURE 3.7: Parallelization option $T2_2$: one thread computes a single LUT value for each LUT, for a single voxel. Some extra communication among threads should be done in order to compute the final $sign(F(\rho))$ for each v_{xyz}.

- Finally, $T2_3$ is the last parallelization option we propose for Task 2. It is based on creating $n \times \dim(VM)$ threads. Each thread processes one voxel v_{xyz} with one LUT. Then, the final cost is $O(|\rho| \times W_c)$. In this solution, we deal with $dim(VM)$ groups of n threads with a value related to the same voxel v_{xyz}. These n threads can use this own generated values (ideally stored on local memories) to process later $T3$ in parallel. This option still presents a big TCO because all the threads need the data generated on $T1$ but it is possible to skip the final reduction step, depending on the configuration of step $T3$. It has more WPO but less TCO than $T2_2$.

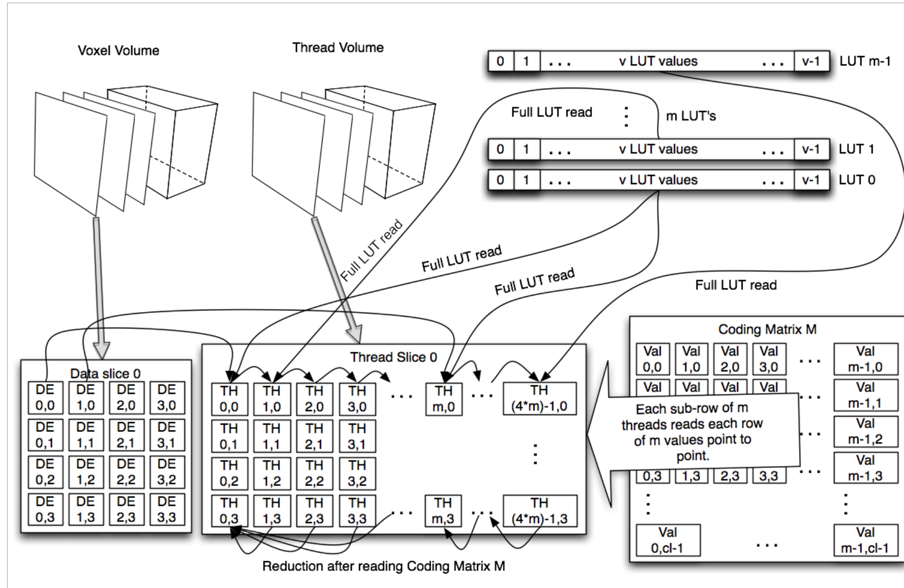


FIGURE 3.8: Parallelization option $T2_3$: one thread reads a full LUT, and a row of m threads share the same value from the input data. Reading of the coding matrix M is done concurrently.

Following the reasoning of the analysis on $T1_2$ we could try to recycle data, by making each thread to process more than one voxel and storing the LUT values in local memories. This reduces TCO and increases WPO. This can be applied in all three options $T2_1$, $T2_2$ and $T2_3$ if the architecture offers enough resources.

3.6.3.3 Task 3 and high level overheads mapping

Finally, the last analysis is for Task 3 ($T3$), which performs the Multi-class labeling.

$T3$ computes for each v_{xyz} voxel the lowest distance from its X-code word to each of the N rows of the coding matrix M. The row corresponding to the minimum distance identifies the final class label. Thus, this process has a total cost of $O(dim(VM) \times dim(X) \times N)$ where $dim(X) = n$. As in $T2$, we describe different options to parallelize it, $T3_1$ and $T3_2$. $T3_1$ maintains a cost of $O(dim(X) \times N)$ by mapping one work item per voxel as in $T2_1$. On the other hand, $T3_2$ uses one work item per voxel, X-code word value. Thus it would have $O(N)$. This second option is ideal to be implemented along $T2_3$, since the number of threads through $T2_3$ and $T3_2$ is the same, avoiding TMO and reduction operations between options TCO. The combination has a lower WPO as well, but it still has a broadcast operation at the beginning of $T2_3$ and a reduction step at the end of $T3_2$ that generate TCO.

3.6.4 GPU implementation (high and low level merging)

In $T1$ we have to read repeated data from global memory. To avoid this problem, we implemented $T1_2$ following the Micikevicius GPU stencil algorithm [Mic09] that moves the repeated readings from global memory to local and private memory. This algorithm uses $dim_x \times dim_y$ work items and a for loop for dim_z. With this approximation, we work with from 16384 work items (128^3-sized voxel dataset) to 160000 work items (400^3-sized voxel dataset), that are good amounts of work items for a GPU. In real environments this algorithm will scale to up to 4194304 work items with 2048^3 voxel datasets. If needed we can increase the number of threads used by executing two or more equally distant $dim_x \times dim_y$ planes in dim_z. This planes will use the same algorithm for a smaller volume. We could find the balance between increased TCO and reduced WPO that way as a future work.

Considering $T2$, as we have seen on section 2.3.3.1, on GPU's $sTCO$ is the most relevant overhead, and in $T2_2$ and $T2_3$ aTCO is bigger than in $T2_1$. We can approximate TCO by calculating the total amount of Ncl in the work item communication operations. This value is huge, thus we selected $T2_1$ as the best parallelization option. We also implemented $T2_2$ in order to empirically check our assumptions. In both $T1_2$ and $T2_1$ we used $dim_x \times dim_y$ work items in order to take advantage of having the results in private memories from the first task T1. Nevertheless both $T2_2$ and $T2_3$ can be good candidates for the newly announced Dynamic Parallelism in the Tesla K20 NVIDIA card and Maxwell cards [NVI12].

$T2$ generates a testing X code word, for each voxel. The size of each X code word is n (the number of binary classifiers used). In our experiments n varies from 1 to 36. In the worst case then we will have to store $dim(VM) \times 36$ 4byte elements that easily might not fit into the GPU global memory. We solve this by following the same philosophy as in $T1_2$, by processing one $dim_x \times dim_y$ plane in $T1$, reading the results on $T2$ and storing the code words for only that plane. Once used by $T3$, the pointer used for the code words is reused for the next plane. That way we only need a pointer of size $dim_x \times dim_y \times n \times 4bytes$, to process all the voxel world.

When considering $T3$, we also have to consider how $T2$ stored the data on memory, and in which memory. This will condition how the data can be read from $T3$ in an efficient manner. If we choose $T2_1$ or $T2_2$, then it is not worth to use $T3_2$ option since it would mean storing in $T2$ and reading on $T3$ from global memory in a non coalesced manner. Instead, $T3_1$ would maintain coalesced reads to global memory. But if we where to use $T2_3$, then $T3_2$ would be the best. Again, the reduction steps in $T3_2$ make it the option with more TCO, comparing to $T3_1$, but a good choice for testing Dynamic Parallelism [NVI12]. We implemented $T3_1$ and integrated the three tasks in a single kernel, using $dim_x \times dim_y$ work items.

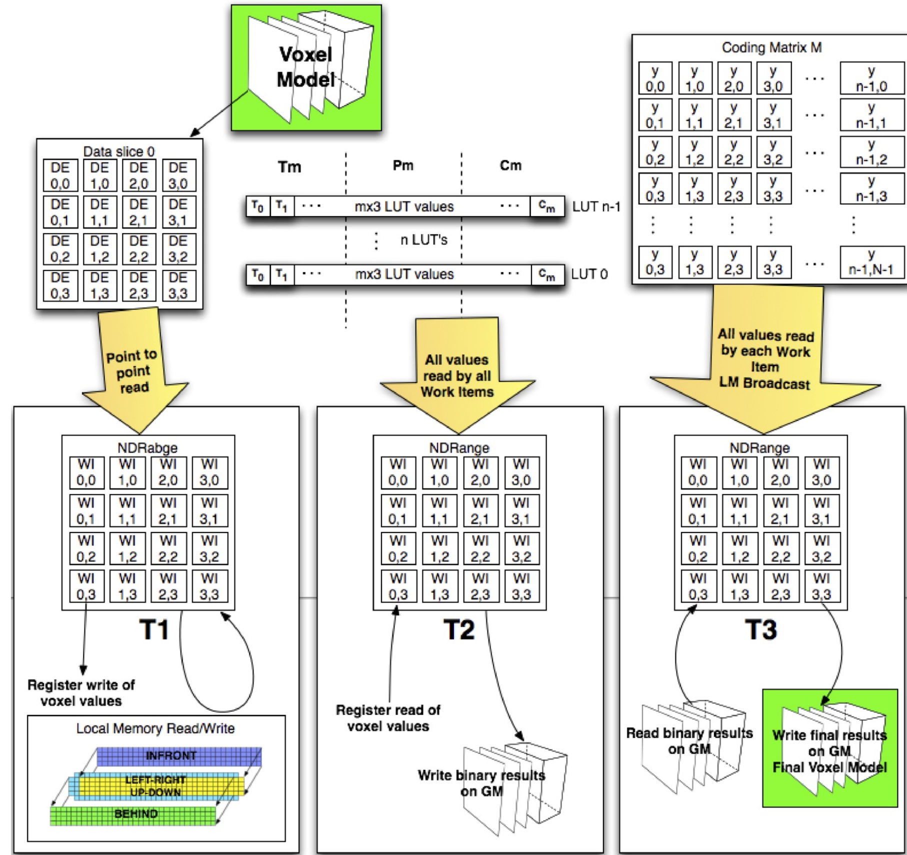


FIGURE 3.9: Data flow for all the three steps: 1) The results of $T1$ are kept on registers to be read from $T2$; 2) The results of T2 are stored in global memory; 3) $T3$ reads the results of $T2$ and reads the coding matrix from local memory since it will be read throughout all z axis. Results are finally stored in global memory.

As it can be seen in Fig. 3.9, the data-flow between the three tasks using $T2_1$ and $T3_1$ is as follows:

1. The results of $T1$ are kept on registers to be read from $T2$.
2. The results of $T2$ are too big to be stored on registers or local memory so we use global memory. In this case, GPUs with L2 cache reduce the overhead of using this global memory. The LUT values are also stored in global memory because for some values of n they don't fit on local memory.
3. $T3$ reads the results of $T2$ and reads the coding matrix from local memory since it will be read throughout all z axis. Results are finally stored in global memory.

As a final remark, considering the Host to Device memory transfer overhead, we decided to transfer only the density value of each voxel to the GPU, and on the GPU calculate the rest of the values in $T1$, like the voxel coordinates, gradients and magnitude. Also, we are only reading from Host the final GPU results and not any intermediate value.

3.6.5 OpenMP and GCD CPU implementations

A fair CPU-GPU comparison should exploit the parallelism both options offer and it should compare similar price point CPUs and GPUs. For this purpose, we have used both OpenMP and GCD for implementing two CPU versions. OpenMP is a well-known API for shared-memory parallel programming in C, C++ and Fortran. On the other hand, we explore GCD since it presents some novel concepts that we think can make it technologically a better option than OpenMP.

3.6.5.1 Implementation decisions for OpenMP

For $T1$, $T2$ and $T3$, with OpenMP the limitation is the same, TMO. We have taken the idea of one thread per voxel in a $dim_x \times dim_y$ plane in $T1_2 + T2_1 + T3_1$ and reduced it to several voxels per thread. So we have parallelized the external for loop (see Algorithm 3) by scheduling some z iterations to each thread. The scheduling we have chosen is the OpenMP dynamic scheduling, because the work done for each x iteration depends on an initial comparison. If the density value of the voxel is below a threshold, we will not process this voxel. So we need to dynamically feed the threads. This dynamic scheduling introduces TMO because there is a thread on runtime devoted to work scheduling, but reduces WPO by balancing the work load.

3.6.5.2 Implementation decisions for CPU's with GCD

For GCD we followed the same philosophy as with OpenMP. The difference is that, instead of having to analyze the work balancing and choose one of three scheduling policies provided by OpenMP, we just had to convert the same external for loop, into a GCD function that enqueues each iteration as a block, with an incremental variable corresponding to the variable that represents dim_z position and which the for loop increments. Knowing the GCD specification, work balancing will be automatically done, and we wanted to compare the execution times to the same parallelization strategy with OpenMP on the same system.

3.6.5.3 Other architectures not implemented

As we mentioned in Section 3.6.4 the theoretical option $T1_2 + T2_3 + T3_3$ has a high potential to be the best option for NVIDIA Dynamic Parallelism [NVI12]. With Dynamic Parallelism, a parent grid (the same as an NDRange space of work items) would follow Task 1 as in the GPU implementation (one thread per voxel for one plane). Then, each thread would call a child grid with the proper dimensions to have one thread per LUT ($T2_3$). The child grid would produce a test code word value per thread that will be

compared to one of the corresponding coding matrix code word values. The child grids will repeat the operation for each row on the coding matrix. The final step would be a reduce-compare operation.

Following recent release of Adapteva's Epiphany documentation [Ada13, Ada], we see that this architecture follows a global memory and local memory space model, similar to the GPU. It is also connected to a CPU with a system memory like in the case of GPU's. There are two differences with GPU's though: first each core is completely independent of the others at execution level, and second the global memory space is distributed among cores and accessed through a fast on-chip network. In this case we would feed the data while reading results due to the limited amount of global memory. Each core would be responsible for processing several voxels like on CPU implementations. Thus we could store the coding matrix in local memory and the LUT's in global memory. Then process all the assigned voxels and finally send the results while receiving the next voxels.

3.7 Simulations and results

This section describes the experimental setup and shows the performance evaluation in terms of classification accuracy and execution time.

3.8 Setup

Data: We used three data sets, *Thorax* data set[2] of size 400×400×400 represents a MRI phantom human body; *Foot* and *Brain*[3] of sizes 128×128×128 and 256×256×159 are CT scans of a human foot and a human brain, respectively.

Methods: We use the one-versus-one ECOC design, Discrete Adaboost as the base classifier, and we test it with different number of decision stumps. For each voxel sample ρ, we considered eight features: x, y, z coordinates, the respective gradients, g_x, g_y, g_z, the gradient magnitude, $|g|$ and the density value, v. The system is compared in C++, OpenMP, GCD, and OpenCL codes.

Hardware/Software: The details of the different CPU and GPU hardware configurations used in our simulations are shown in Tables 3.2 and 3.3, respectively. The viewport size is 700×650. We used the MoViBio software developed by the GIE Research Group at the UPC university [mov].

Measurements: We compute the mean execution time from 500 code runs. For the accuracy analysis, we performed 50 runs of cross-validation with a 5% stratified samplings.

[2] http://www.voreen.org
[3] http://www.slicer.org/archives

CPU's	AMD Phenom 2 955	Intel Core 2 Duo P8800	Intel Core i5 750
Frequency	3.2GHz	2.66GHz	2.66GHz to 3.2GHz
Cores	4	2	4
Threads per core	1	1	1
L3 cache	6MB	0MB	8MB
SSE level	4A	4.1	4.2

TABLE 3.2: Different CPU configurations used for evaluation.

GPU's	ATI	NVIDIA
Processing Elements	720	448
Stream or CUDA cores	144	448
Compute Units	9	14
Max PE per WI	5f / 0d	1f / 1d
PE available per CU	80f / 0d	32f / 4d
Warp size	64 Work Items	32 Work Items
Memory type	GDDR 5	GDDR 5
Global Memory size	1GB	1.28GB
Local Memory size	32KB	16KB or 48KB

TABLE 3.3: Different GPUs architectures used for evaluation, where 'd' and 'f' stands for double and float, respectively.

3.9 Classification accuracy analysis

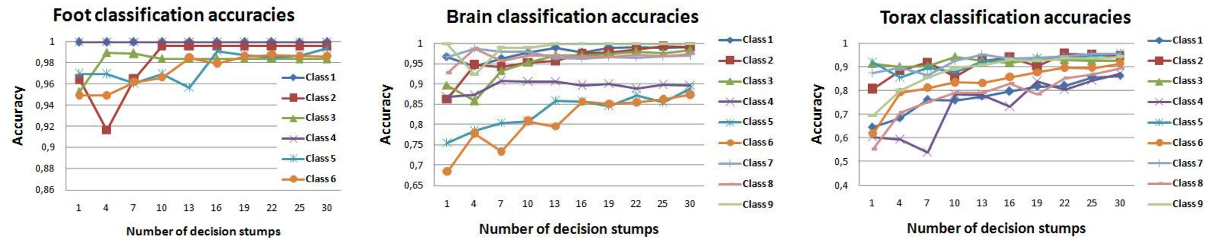


FIGURE 3.10: Classification accuracies for the different data set structures and number of decision stumps in the Adaboost-ECOC framework.

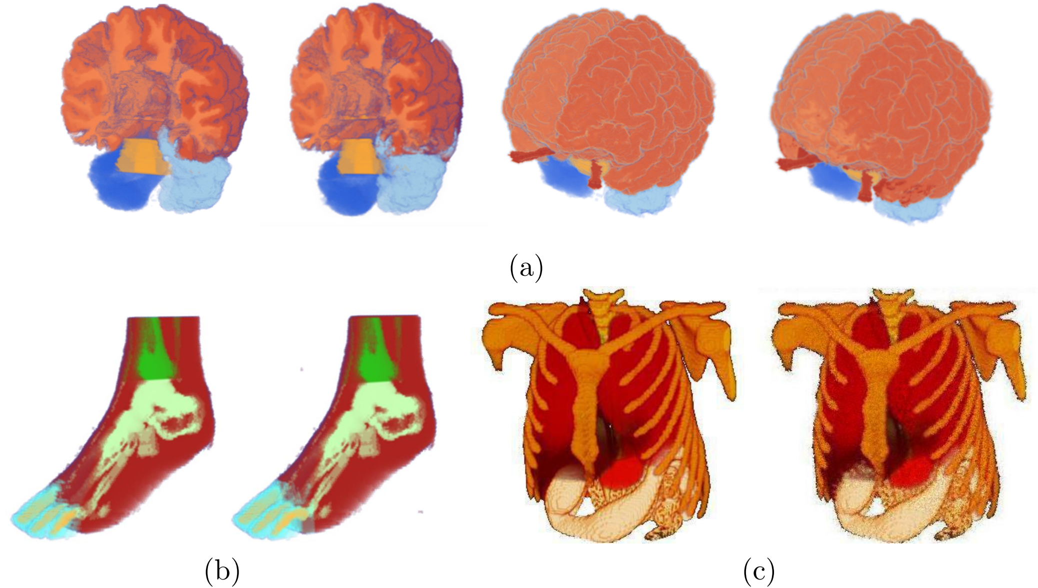


FIGURE 3.11: Comparison of the pre-labelled (left) and the classified data set (right) for (a) the *Brain*, (b) Foot, and (c) *Thorax* data sets, respectively.

Fig. 3.10 shows the classification accuracy of the framework for the data sets considering different number of decision stumps $\mathcal{M}$. The accuracies are shown individually for each

volume structure. From Fig 3.10 one can see that even for different complexity of volume structures, most of the categories obtain upon 90% of accuracy.

In the case of the *Foot* data set, accuracy achieves near 100% for all the categories. In Fig. 3.11(b), we show the same section of the full pre-labelled *Foot* data set for six classes and the obtained classification with our proposal. In Fig. 3.12, we show selections and different shadings of the visualizations to demonstrate the flexibility of submatrix testing and the label mapping for different interaction models I.

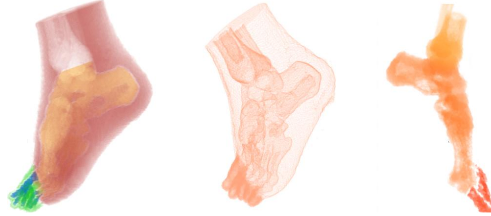


FIGURE 3.12: Case study of the 6-class foot volume: c_1 is the background, c_2, c_3, c_4 are the bone structures of the palm, ankle and tooth, respectively, and c_5 and c_6 are the soft tissue of the palm/ankle and tooth, respectively. Color Plates show the user interaction sets (from left to right) $I = \{\{c_1\},\{c_2\},\{c_4\},\{c_5\},\{c_6\}\}$, $I = \{\{c_1\},\{c_2,c_3\},\{c_5\},\{c_6\}\}$, and $I = \{\{c_1\},\{c_2\},\{c_3\},\{c_4\}\}$, respectively.

In case of the *Brain* data set, we have trained nine classes of different structures of the brain that shares the same density and gradient values, achieving accuracies between 90% and 100%, where the lowest value is for the class c_6, with values near 90%. The highest accuracy corresponds to the class c_9. In Fig. 3.11(a), we show two sections of the pre-labelled brain structures together with the results obtained by our multi-classifiers. Note that classification inaccuracies do not present significant artifacts in the rendering. In Fig. 3.13 we also present some illustrative visualizations of this multi-classified data set applying different user interactions I. Finally, Fig. 3.11(c) shows an example of a full Thorax volume classification for different structures[4].

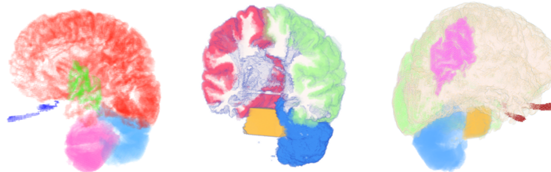


FIGURE 3.13: Case study of the 9-class *Brain* volume: the classes corresponds to different structures of the brain such as hemispheres, optical nerves and cerebellum with different shading techniques.

[4]Note that the proposed multi-class GPU-ECOC framework is independent of the classification/segmentation strategy applied. Thus, different feature sets as well as labeling strategies could be considered, obtaining differences in classification accuracy.

3.10 Execution time analysis

We compared the time performance of our GPU parallelized testing step in relation to the CPU-based implementations based on sequential C-code, OpenMP and GCD approaches. In table 3.4, for each implementation we show an averaged time of the 500 executions. We employ different sized data sets with several number of classes to be labelled and distinct user-selections. Our proposed OpenCL-based optimization has an average of speed up of 109x, 31.11x, and 31.14x over the sequential C-coded algorithm, the OpenMP and the GCD based algorithms, respectively. The results are remarkably promising considering that the whole tested data sets fit in the GPU memory card. We are planning to expand our implementation using well-known bricking techniques. In table 3.4, we can observe that time values are proportional to three features: the data set sizes, the number of classes,N, and the number of classifiers, Z, used for the selected classes. Specifically, the most influential feature in the time value is the size of the data set (see rows 6, 10 and 16 of table 3.4). Thus, including more classifiers hardly affect the time performance of the system. In addition, we obtain real-time in the *Foot* data set.

Data set	N	Sel. classes	Z	CPU	OpenMP	GCD	OpenCL
Foot	3	2	2	0.387	0.111	0.111	0.008
	3	3	3	0.577	0.165	0.165	0.002
	4	3	5	0.948	0.271	0.271	0.020
	4	4	6	1.139	0.325	0.325	0.038
	6	6	15	4.986	0.760	0.769	0.062
	9	9	36	8.319	1.787	1.777	0.091
Brain	9	2	15	39.396	11.190	11.177	0.358
	9	4	26	68.485	19.475	19.615	0.649
	9	6	33	87.558	24.947	24.875	0.848
	9	9	36	96.859	27.642	27.557	1.263
Thorax	2	2	1	26.849	7.604	7.600	2.694
	8	2	13	321.768	94.589	94.579	2.011
	8	4	22	564.577	160.801	160.784	3.532
	8	8	28	754.203	220.430	220.388	6.007
	9	7	35	923.225	260.751	259.489	5.955
	9	9	36	971.915	270.751	269.751	7.763

Table 3.4: Testing step times in seconds of the different datasets using a submatrix S_M depending on the subset of selected classes. The whole matrix M is used when all classes are selected. The CPU, OpenMP and GCD times are tested on a Intel Core i5 750 (see Table 3.2). OpenCL times are tested on a GTX 470 (see Table 3.3).

In Table 3.5 we observe time values of our OpenCL implementation for different parallel platforms: the ATI Radeon, the Nvidia Geforce GTX470, and the AMD Quad Core. We conclude that the best performance is achieved on the Nvidia graphic card whose time ranges from a few seconds in the worst case to some miliseconds for the best one. In addition, we executed the OpenCL code in the AMD CPU and we obtained speed ups of 1.02x, 1.06x, and 1.2x over the corresponding OpenMP implementation for 9 classes and 36 classifiers of the three data sets, *Foot*, *Brain* and *Thorax*, respectively. We observe that the greater dataset, the better OpenCL performance.

3.11 Conclusions

In volume visualization, users usually face with the problem of manually defining regions of interest. To cope with this problem, we proposed an automatic framework for general multi-class volume labelling on-demand. The system is decomposed into a two-level GPU-based labelling algorithm that computes in time of rendering voxel labels using the ECOC framework with the Adaboost classifier. After a training step using few volume voxel features from different structures, the user is able to ask for different volume visualizations and optical properties. Additionally, to exploit the inherent parallelism of the proposal, we implemented the testing stage in C++, OpenMP, GCD, and GPU-OpenCL. Our empirical results indicate that the proposal have the potential to deliver worthwile accuracy and speeds up execution time. Overall, the proposal of this paper presents a novel, automatic, and general-purpose multi-decision framework that performs real-time computation.

Data set	N	Selected classes	Z	ATI	NVIDIA	AMD Quad
Foot	3	2	2	0.028	0.008	0.236
	4	3	5	0.066	0.020	0.495
Brain	9	2	15	1.498	0.358	7.114
	9	4	26	2.636	0.649	12.393
	9	6	33	3.403	0.848	16.047
	9	9	36	3.818	0.959	18.446
Thorax	8	2	13	8.750	2.011	56.374
	8	4	22	14.860	3.532	95.657
	8	6	27	18.340	4.467	118.978
	8	8	28	19.150	4.752	125.454

TABLE 3.5: Different timings for different parallel architectures obtained by the ATI-Radeon, Nvidia GTX470 and AMD Phenom 2 955.

There exists several points in this novel framework that deserve future analysis. We plan to analyze the effect of classifier generalization reducing the initial number of voxels in the ground truth training set, look for different feature space representations including contextual information, study the use of different base strategies in the ECOC framework from both learning and segmentation points of view, as well as new parallelization optimizations. In order to avoid possible memory usage limitations for larger volume data sets, we will also analyze bricking strategies. Finally, we plan to embed the training step in the visualization pipeline so that online learning and forbidden steps can be performed on-demand in the rendering.

Chapter 4

Parallel system proposal

Additionally to all the work presented, we show a proposal of a parallel system, that tries to be scalable and simplify parallel programming.

4.1 Introduction

Having a parallel computing system with all of its processing units busy with program code most of the time for the execution of a parallel program, is the main problem to solve in Parallel Computing Science. There are plenty of improvements in one area or another (hardware, algorithms, compiler, languages, operating systems), in order to make parallel systems more efficient, scalable, and easy to program. Sometimes, for improving scalability, sacrificing some programability may turn to be a success, as is the case for GPGPU's. Other times, making it easier to parallelize a code, may be successful too, like with OpenMP. Some other works have tried to make any parallel code, portable across architectures, like OpenCL. Others try to ease the programming burden for supercomputer programmers, like OmpSs [BSC], but don't make it easier for the average programmer.

In this proposal we describe a Self Organizing Multi Agent System that tries to exploit Agents natural parallel and decentralized working. We start by considering that a computer parallel program, is programed in any sequential language like C, C++, Java, etc... but adding the concepts already found in Apple's GCD [App09]. Mainly, the concepts of code Block and FIFO or concurrent queues that allow to define concurrency or serialization across Blocks.

Starting from this point, we already have a lot of instruction blocks that point to data they need to read or write, in a single memory space. We consider this conceptual environment like a market in where Self Organizing Agents have the only goal to manage to be as much time as possible consuming Blocks. They can have code Blocks, but they

may lack the data necessary to execute them. So they trade with both code Blocks and Data, and set their expected selling or buying prices, according to this code-data matching. The more matchings a resource (either Block or Data) is giving to an agent, the more valuable it is for the same agent, since it will allow him to execute more instructions.

We don't specifically define how a Data resource transaction is done (from a computer architecture perspective), since depending on the architecture it can be more efficient to consider buying a byte, or a set of bytes (a page for instance). So when we refer to data, we talk about a conceptual unit of data with a unique global identifier. Also, each agent has it's own global unique identifier.

The rest of the section continues as follows. The next subsection, Definitions, presents more specific definitions regarding the Agent behavior, how do they set prices (agent local view of the value of a resource), and how do they negotiate (agree on resource transactions). On the Experiments section we describe the experiment we set to start testing this computing paradigm and be able to observe how the blocks and the data spread across agents, as well as analyzing how much time are the agents waiting for resources. On section Results we sow and discuss the experiment results, and some conclusions, and finally on the Conclusions section we discuss what are the conclusions we extract from our own proposal, experiments and results, as well as possible future work.

4.2 Previous work

Some works on schedulers, and programming languages like OmpSs [BSC], are already implementing many of the features we pursue with this proposal.

- Data affinity: in some systems it is possible to know in which memory the data of a given piece of code is, and therefore send the corresponding thread to the device or node containing this data [Bol01, TTG95, BMD*11]
- Hierarchical scheduling: sometimes there are many tasks to schedule and having a single core or node to do all the scheduling work in run time can prove to be very inefficient. If the language allows to generate new tasks in execution time, it is possible for the master node to delegate further scheduling of the tasks generated in a particular piece of code [HSSY00]
- Add asynchronicity to pieces of code with annotations over serial code [App09, BSC]

All this features are achieved with very varied implementations, but none is implementing all of them by concept. We try to define a system where all this features and more

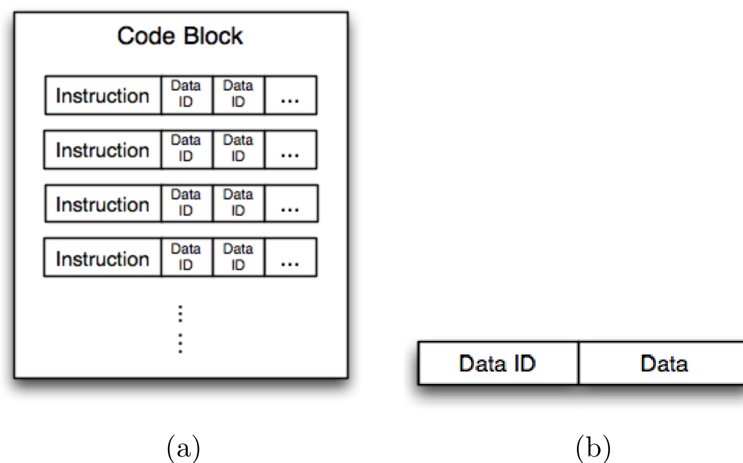
are built in to the design of the Agents, and the Multi Agent System, in a natively decentralized manner.

4.3 Definitions

On this section, we describe in detail how the system works. We define the resources, pricing mechanism, agent behavior and all the necessary definitions in order to be able to simulate the system.

4.3.1 Blocks and Data

We now show some figures representing a Block and a Data element.



(a) (b)

FIGURE 4.1: (a) Instruction Block and (b) data element

As seen on Figure 4.1 (a), a Block is made of instructions that use Data elements. We only describe the need for an instruction to use a Data element with a certain unique global ID. Blocks also have a global and unique ID that on normal systems is the memory position in where the code of the Block starts.

As seen on Figure 4.1 (b), a Data element is formed by a global unique ID, and data. We will only define the ID for our experiments, since we focus on scheduling. The size of the data (like using pages of data) is something we will consider in further work, towards a real implementation of the system.

4.3.2 Negociation

We define negotiation as an comparison on expected prices (each agent has it's own expected price) for a given resource. A selling action is performed when the two agents

involved agree on the price. A price agreement is a condition where the buyer agent sets an expected price equal or higher than the price set by the seller agent.

4.3.3 System initialization and global market

The system starts by creating FIFO and concurrent queues (as defined on GCD). There is a global market, responsible for doing that, and enqueueing Blocks that will be shown to the agents so they can buy them, along with the necessary data. Te same way as in GCD a Block can enqueue another Block in a queue, but we restrict FIFO queues to be owned only by the queue creator agent. For the sake of simplicity, on this paper we skip the possibility of a Block to have parameters defined. With this scheme, the problem of data dependencies is solved, the same way as in GCD.

The global market is generating only the blocks that are called from the main program. The global market does not execute any block. So even Block creation is distributed as in hierarchical scheduling.

The global market sets a Market Value (MV) for all the resources, either Data or Blocks. It also adds Transmission Costs (TC) that may vary depending on the real hardware implementation. MV may be equal to all Blocks, or the global market may analyze the amount of work present in a Block, and set a price depending on that.

4.3.4 Agents and trading

We define a negotiation SOMAS where the following conditions are assumed:

- An agent can perform seller and buyer roles.
- The resources an agent can trade with are either asynchronous instruction blocks, or data elements used by the instruction blocks in a single memory space.
- There is a global source of resources (the market) with a fixed or variable price, that has already defined the instruction blocks in a way pretty similar as in GCD [App09]. The programmer should be then aware of the dependencies and generate blocks that are not accessing other blocks variables on the wrong order (like in GCD). A way to further improve this could be to add dependency control as in OmpSs [BSC]. But we start defining a more simple system focusing on the system collaborative behavior.
- When we say that an Agent buys, we mean that there has been a price agreement in favor of the buyer, and he gets the resource. The opposite for the seller. This are not true roles, but a way of explaining the Agent behavior. There is no currency,

the only thing exchanged between Agents are Blocks or Data elements, that we call resources.

- A resource has a value V composed of a primary value PV and Rr. Each agent has a private view of V so he is responsible to calculate V for all resources he sees or is aware of. PV is calculated by the amount of computation that the resource allows the agent to do. The instruction blocks have enabled and disabled instructions (to simplify we could enable or disable the full block). An instruction is enabled when the agent has the data used on this instruction. So a data resource may enable a number of instructions, or an instruction block may have more or less instructions activated according to the data that the agent has. The PV for a resource grows with the number of enabled instructions available on the agent due to this specific resource.

- An agent has a limit in the number of resources it can own. This limit is affecting V. If the agent is almost full of resources, V will decrease, so he will sell cheaper (more) and buy cheaper (less). On the contrary, if the agent has few resources the V will be increased, so he will sell more expensive (less) and be able buy more expensive (more). The result is $V = PV + Rr$ where $Rr = Resourceratio$ is a value that quantifies the number of resources available for the agent to execute blocks (memory, cores etc...).

- When an agent is performing the buyer role, it sets it's expected price for the resource by calculating the V for that resource according to the previous rules.

- When an agent is performing the seller role, it sets it's expected selling price by calculating the V for that resource, plus adding the "transport fares". So it is adding the costs (based on transfer time) of sending the resource to the other agent, taking into account the performance costs of sending data or blocks through the network. Setting this extra cost allows to regulate how prioritary it is for the system to do reduce communication, depending on the hardware used for connecting agents.

When an agent evaluates a resource, it compares the resource V he calculated, with the V that a limited amount of neighbors have calculated. The Agent that gives the greatest value is the buyer and takes the resource.

4.3.5 Expected behaviors

With this definition, we expect the system to automatically use hierarchical scheduling as we previously mentioned. Also it will perform data affinity scheduling, since it is controlling how much a data element is matching one or another agent. The Blocks are

created with a syntax similar to annotations. Additionally, the system can adapt to different network latencies and bandwidths.

We can expect data-flow behaviors, where some agents keep the same blocks of code, but the data moves as an stream through spontaneously pipelined agents.

We can also expect blocks of code flowing and moving from agent to agent, if the data owned is used by many blocks in each agent, and only few blocks need the data on other agents at a certain point.

As we can see, the intention is to take advantage of the Agent programming spirit to allow many sorts of scheduling that will spontaneously happen as the most efficient scheduling, depending on data and block affinities, agent available resources, and communication costs.

4.4 A hardware proposal for the system

Some may argue that all the Agent processing is a huge overhead, and that a system like this can not be efficient. The first clue towards thinking otherwise is the actual existence of multi agent systems for grid computing [SHL*06, MB05]. In this systems, the network has a huge cost, so the overhead of the Agent program, is compensated by the efficient communication policies.

Additionally to this, the tendency on parallel systems is to increase the number of cores more and more. Some projects even use low power mobile processors, and lots of them, to create supercomputers [mon11]. But the cost of the network increases. The networks have difficulties keeping performance on pair with the amount of data the processors can consume. Moreover, the cores are increasingly cheap. That means, that from a market point of view, it makes sense to devote some hardware to only the Agent program, and some other hardware to do the calculations.

An ideal platform for experimenting would be the Adapteva's parallella board [par13]. This board is a complete systems with CPU, ram, hard disc, and all peripheral connectors expected on a PC. It uses two ARM CortexA9 cores traditionally used on mobile phones, paired with an Epiphany 3 or 4 chip with 16 or 64 very low power cores, specialized on floating point operations. The two ARM cores can be devoted to execute the OS and the Agent program. The blocks then would be Epiphany kernels, and a virtual distributed unified memory system should be also implemented in software using pagination. The Agent should have some tables with information of owned kernels and memory pages, along with a copy of the pages of few neighboring boards.

4.5 Experiments and results

In this section we present the simulator we created, the experiments performed with it and the results, along with some conclusions.

4.5.1 The simulator

We implemented a simple simulator following a simplified version of the definitions. The simulator is written in Python, for faster programming. With this version we are able to create Agents that have the purpose of executing instructions that need data to be executed.

Each Agent is performing two tasks, executing Block instructions (including raising data requests), and managing the data requests he has stored. This two actions happen serially on the simulator, but on real systems that could be simultaneous.

For simplicity, these two actions are performed in a single time step. The agent will execute the next time step when all the rest of the agents have executed it own time step.

The variables we can modify to see experimental results are the following:

- Number of agents on the grid.
- Number of Blocks.
- Number of data elements.
- Number of instructions for each Block.
- Number of data requests the agent will raise to the grid on a single time step.
- Number of instructions the agent will fetch on each time step.

Mainly, the question being analyzed on this work, is the effects of all this variables on the number of data requests on the grid. This can be taken as a performance metric, since every data request will have a cost in any system, and will reduce the overall execution performance.

We will now explain a bit deeper each class on the simulator, on the next subsections.

- Block class: This class represents a simplified GCD Block. We defined a Block as a sequence of instructions without branches or function calls. All the instructions read one data element. For simplicity we haven added write instructions and all the instructions have the same execution cost.

We represented the sequence of instructions in a Python List. Each element of the list contains a number that represents the dataID that the instruction would read. So there are no real instructions, but only the information of the data that the instruction would read.

The class has a method that simulates the fetch operation of an instruction. It searches for the data element on a data table (another Python List "dataTable" owned by the agent). If the data is on that table, the instruction is considered to be executed, and is erased from the list "dataSequence". If the data element was not on the "dataTable", the method returns the dataID so the agent knows that the instruction needs data that is not available at the moment. RequestsManager class This class is a small one, that allows to control both the data ID requested by any block on the Agent, and the number of times it has been requested. The purpose of storing the number of times a data ID has been requested, is implemented on a method called "popPriorityRequest". This methond searches for the data ID with the greatest number of requests, erases it from the lists, and returns the data ID, since the agent will put the request on the grid.

- Agent class: This class represents an Agent. The Agent class contains a Python List of Block objects, and another List of integers that represent data ID.

  It has two main methods. The first, "senseResoruces", has that name to refer to the concept of an agent process executing in a different peace of hardware than the one that is executing Block instructions. The Blocks might be executing and at a certain time interval, the Agent senses the state of the blocks and stores the information. What the method really does, is to do what in theory was already done. It executes a number of instructions and stores the data requests if any.

  The second method is "manageRequests". This method uses the requestManager object to get the priority data ID and raise a data request on the grid.

  The number of instructions executed on the "senseResources" method and the number of data requests raised to the grid on the "manageResources" method are set on every call. This is on the purpose of analyzing the effects in the number of data requests by changing these values. There is a this method called by the grid, that informs the agent that someone in the grid is asking for a given data ID. The functionality of the method is simplified to directly return the data ID and erase the data ID from the Agent "dataTable".

- Grid class: This class represents a grid. This grid is very simplified. It does not represent space, so the agents have no distance between them. All the cost of communication is the same for all the agents. This is one of the first things to improve on next versions, although grid access is analyzed, and can be used to compute an access cost using the Agent ID.

This class has several methods. The constructor takes all the data and all the Blocks, and generates two Python Lists with as many Lists as Agents each. Each sublist contains a list of data elements for the first list and a list of Blocks for the second list. Each pair of sublists will be assigned to a different Agent. The data is spread almost randomly, as well as the Blocks. Future work will include distributing blocks and data, as well as instructions in a block, according to parallelization strategies and algorithms to enrich the analysis.

Two methods are made for initialization purposes. They create the agents with a pair of Block and dataTable sublists as arguments, as well as the grid itself and an agent ID. "finished" method increments a counter to know when all the agents have finished. It also includes methods for reading the number or data requests and for sending the data requests to the proper agent.

4.5.2 The experiments

The simulator was written in Python, an interpreted language that compared to other compiled languages like C, is very slow. For this reason, the number of experiments we have don is quiet limited, but show that the simulator works and shows the expected behavior. Also, the information being analyzed is quite collective. The interactions between Agents are limited to data queries, and so the possible situations are quite predictable.

Experiment 1 The configuration for the experiment is the following:

- Agents = 100
- Blocks = 200
- Data elements = 100000
- Block Size = 100
- Requests = 1
- Instructions = 100

The number of total data requests was 19743

This is the smallest experiment. We did it just to confirm that using less Blocks and Agents will translate into less data requests.

Experiment 2 The configuration for the experiment is the following:

- Agents = 1000

- Blocks = 3000
- Data elements = 100000
- Block Size = 100
- Requests = 1000
- Instructions = 100

The number of total data requests was 299567

We can see as the number of data requests on the grid has increased due to an increased number of both Agents and Blocks. This experiment has two values that should be suboptimal comparing to the other experiments. When the number of requests sent to the grid on each time step, is bigger than the number of instructions executed, it means that at each step, all the requests generated on the agent will be sent to the grid, erasing the possibility of doing a single grid data request for several internal agent data requests that can be coincidental.

Experiment 3 The configuration for the experiment is the following:

- Agents = 1000
- Blocks = 3000
- Data elements = 100000
- Block Size = 100
- Requests = 10
- Instructions = 1000

The number of total data requests was 299451

The difference is almost not noticeable on the total number of requests comparing to the previous experiment, but in this one the results are better. We expect a greater difference when the number of blocks is bigger. Now, each Agent has only 3 blocks, so the request grouping can be very big.

Experiment 4 The configuration for the experiment is the following:

- Agents = 1000
- Blocks = 3000
- Data elements = 100000

- Block Size = 100
- Requests = 100
- Instructions = 1000

The number of total data requests was 299486

The number of requests for this experiment is quite smaller than the experiment 2. This is normal taking into account the way agents manage requests internally.

Chapter 5

Conclusions and future work

We have seen that Parallelism, as a Computer Science research topic, and as a tool, is deeply connected to AI. First, many AI algorithms share common purposes and applications, but some are more parallelizable than others. This can be an advantage since this parallelization allows to add more complexity for increased accuracy on execution time constrained applications. Second, AI techniques can be applied for decision making when scheduling tasks on a system or when deciding which resources to use for a given task. This could lead to a software substitution of the parallel programmers that does a combination of preprocessing and runtime work, so any code can run parallel.

Adaboost and the ECOC framework showed a wide range of granularity, that can be exploited in different ways. The results we obtained where satisfactory, and there is room for improvement on accuracy and execution time with bigger volumes. On the accuracy side we could use more neighbor data, like generating gradients reading more than two neighbors on each dimension, and reading classification decisions from the neighbors to do a second classification round with this extra data to automatically correct errors.

Self Organizing Multi Agent Systems seem promising for grid computing and smaller systems that every time have more cores. The Agent programming is a perfect paradigm for local and distributed scheduling sytems, autonomy, and resilience, a concept that is a growing concern on the design of exa-scale supercomputers, where the provability of a node failure is growing, due to the huge amount of nodes.

As a future work we could explore further algorithm improvements for the classification problem, both for accuracy and execution times, and on the parallel system, we could create a full simulator either with standar languages or with Multi Agent System languages like 2APL that can use platforms like JADEX for distributed computing and it's own java environment interface to interact with other languages like OpenCL, C, etc...

Chapter 6

Appendix A (OpenCL summary)

6.1 Introduction

High performance computing (HPC) has traditionally been limited to scientific or military codes, since in contrast personal computing was evolving much more affordably with general purpose and serial processors. Parallelism and heterogeneity was very limited to big budget projects. Also, the programming languages, tools and paradigms where limited since there was not a bit number of projects using parallelism.

In the last years there has been an expansion of HPC thanks to the adoption of commodity general purpose CPU's. This CPU's have included vector processing units but overall they are much cheaper to improve at architecture level, since the architecture used is sold in high volumes. This is what is traditionally called economies of scale, architectures that are affordable to improve because they are sold in volume.

Until now, there was just the x86 and ARM architectures fitting on the idea of volume production architectures. ARM was not enough computationally powerful to be used on HPC, although recently research is being done to use it as the base of a Supercomputing system [1]. So most of the HPC systems where (and still are) based on x86_64. Nevertheless, some years ago, another very different architecture, that was and still is being sold in volume to the mass market, acquired some general purpose capabilities. They are called Graphics Processing Units (GPU), and their purpose was specific for games and visualization.

The market for GPU's is almost as big as CPU's, since nowadays, the need to visualize data or user interfaces is a given for mass market products. The GPU, has been traditionally a fixed function pipeline processor, but with the introduction of programable shaders, it became possible to create visual effects that run fast on the GPU without fixed function circuitry. Scientists saw that this had potential for matrix computations

and started what was called General Purpose GPU. The GPU companies started to offer some programing models to harness the potential of the GPU's, ideally for any code (NVIDIA CUDA, ATI CAL) but there was no standard. Apple was very interested on it and so created along with the GPU companies OpenCL, that pretends to allow easier GPGPU programming than doing so with shaders. OpenCL is an Apple registered trademark used with Apple's permit by the Krhonos group.

The situation was that for the first time, a mainly specific purpose architecture was available at consumer prices for data intensive and matrix computations, perfect for a lot of scientifical codes. Also, some tasks as simulations for industrial component design and film special effects, or simply scientifical experiments related to particle simulation and others, could be done on a PC with one or more high end GPU's. That marked the return of the workstation. So, it was then possible that the historical trend of a general purpose CPU absorbing or surpassing all specific technologies could be changed, and instead, have more heterogeneous systems, with two or more different architectures, each one with it's strengths and weaknesses.

6.2 OpenCL description

OpenCL (Open Computing Language) is an open royalty-free standard for general purpose parallel programming across CPUs, GPUs and other processors, giving software developers portable and efficient access to the power of these heterogeneous processing platforms.

OpenCL consists of an API for coordinating parallel computation across heterogeneous processors (OpenCL Host Code), and a cross-platform programming language with a well-specified computation environment (OpenCL Device Code).

The concepts behind OpenCL can be grouped in the following list:

- Platform Model
- Execution Model
- Memory Model
- Programming Model

Next we will explain one by one the concepts behind this items.

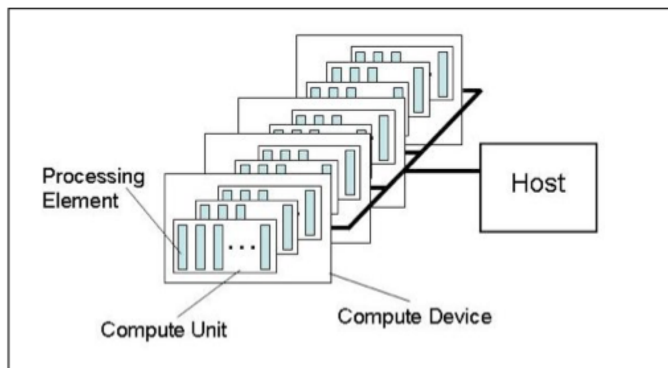


Figure 6.1: OpenCL Platform Model

6.2.1 Platform Model

The platform model defines a virtual architecture, where the system hardware is described in an abstract way. The typical hardware, conformed by one or more CPU's sharing the same physical memory, and an Operating System executing the main program and managing communication and scheduling is called the Host. A GPU, FPGA or any other hardware set as a plugin device, that communicates with the Host through any kind of I/O and/or driver, is called a Device. Nevertheless, a CPU is also considered a Device. So a CPU is both part of the Host and a Device.

Next we list some facts about the OpenCL Platform organization:

- A Host system can have one or more OpenCL devices, and use them concurrently.
- A device can contain several Compute Units, that are responsible of one or more execution flows.
- Each Compute Unit (CU) contains one or several Processing Elements (PE), that are those who execute the code. In most cases a PE is an Arithmetical Logical Unit (ALU).

6.2.2 Execution Model

The execution of an OpenCL program occurs in two parts: a Host program (Host code) that is written in C or C++ with OpenCL calls and data types, and one or more device Kernels (Device code) that execute on OpenCL Devices, and is written in OpenCL C language. The OpenCL C language is based on ANSI C, with some restrictions and built-in data types and functions.

6.2.2.1 Host code

The Host code manages the execution of the kernels and the data transfers between the Host and the Devices. It is programmed explicitly through OpenCL calls and objects. The OpenCL objects involved on the Execution Model are the following:

- Device objects. This objects represent an OpenCL device.
- Kernel objects. A kernel object represents a compiled kernel device program, for an specific device, ready to be executed on that device.
- Program objects. This objects represent the source code, and are used to compile it on run time, for a given device.
- Memory objects. This objects represent pointers. They are divided into buffer and image memory objects. The host and any device in the same context as the memory object can access the contents of a memory object through OpenCL API calls.
- Platform objects. This object allows to use different OpenCL implementations within the same program. If a manufacturer offers an OpenCL implementation for it's hardware, and another manufacturer a different implementation for different hardware, the way to access all of them from a single program is to use a platform object for each hardware manufacturer. As seen on figure 2, any OpenCL object can only belong to a single platform.
- Context objects. A context object is defined for a given platform, and a given device or set of devices from the same platform.
- Queue objects. This objects are used to send data to or read from memory objects and to send kernels to a device for execution. A queue is attached exclusively to a single device.

6.2.2.2 Device code

The programming model of the OpenCL kernels for GPU's is based on the NDRange space. An NDRange space is defined by N dimensions from 1 to 3, where each element is a Work Item (WI), exactly like a CUDA thread. By default, the code of a NDRange kernel is executed by all the WI's. It is possible to assign different data or even different tasks to each WI, in a way similar to MPI. The index of the WI can be used as a pointer index (do the same for different data), or as a conditional expression argument (assign different tasks to each WI).

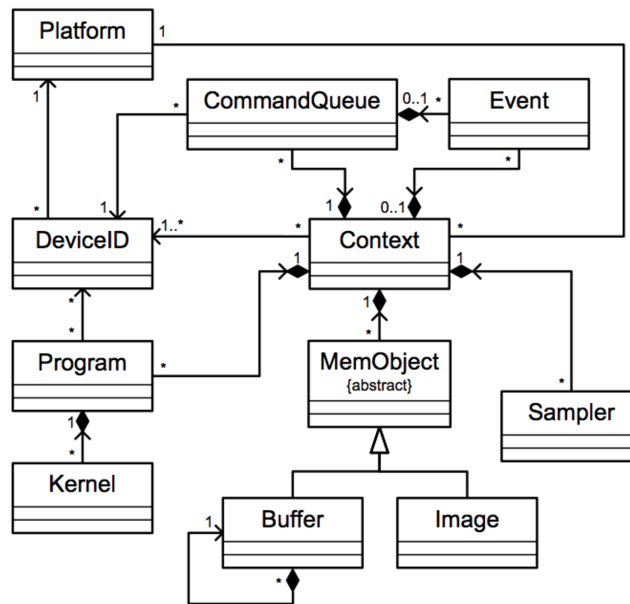


FIGURE 6.2: OpenCL Execution Model diagram

To better map the WI to GPU hardware, they are grouped in Work Groups (WG), the same as the CUDA thread blocks. There is a hardware defined limit for the number of WI in a WG, and a hardware defined limit for the number of WI in a given dimension of the NDRange kernel.

Regarding synchronization, the programmer must take into account that the OpenCL specification doesn't ensure synchronization between WI's. So WI's can execute the same instruction in any order. It is true that on GPU's, due to hardware scheduling, there is a synchronization between blocks of 32 or 64 WI's (depending on the model). But to make portable code, the programmer must not take that into account, and try to solve any possible synchronization problem between WI's.

The only way to synchronize WI's is through barriers and memory fences. Barriers produce that all WI's stop on the barrier call until all the WI have executed the barrier. That way the programmer ensures that no WI will do something before something else is done. The memory fences, produce that all or specific memory operations stop in the memory fence call, until all WI's have executed the memory fence. The barrier implicitly calls a memory fence.

6.2.3 Memory Model

The memory model is again based on the GPU memory model. It applies to kernels only (device code). This memory model allows to specify which kind of memory on the hierarchy will be used for each pointer, in order to improve the execution times.

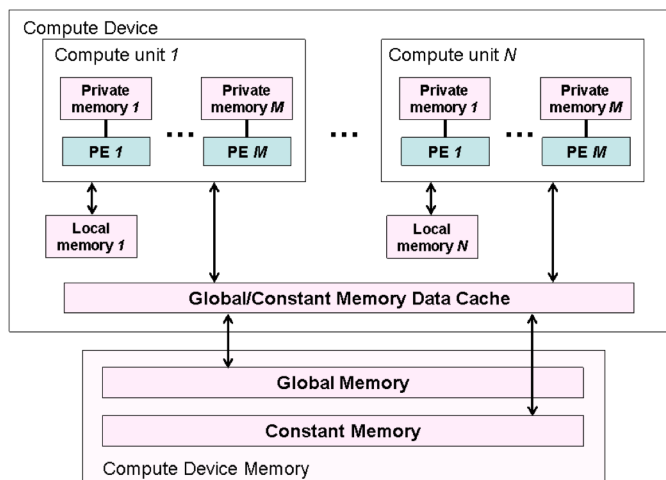


Figure 6.3: OpenCL memory model

The memory model components are the following:

- __global: this is the slowest memory on the GPU, and also the biggest. It can be read or written by any WI on the kernel.
- __local: this is like a programable cache. Is fast, not as registers but very close, but is also small. It can only be accessed by the WI's in the same WG. WI's from different WG's can not share data through local memory.
- __private: this are the registers. The tag __private is not necessary since any variable declared inside the kernel without any memory tag, is automatically considered private. The variables declared as __private can only be accessed by the WI that has declared it.
- __constant: this is a portion of the global memory with size n, that is read-only and has the ability to broadcast one value to m WI's. n and m depend on the device model, but n uses to be small.

Figure 3 shows the relationship between memory layers. Optimizing the memory hierarchy use, is one of the most important matters in order to optimize GPU code.

cm

Chapter 7

Appendix B (SimpleOpenCL)

7.1 Introduction

OpenCL tries to provide an abstraction that can be used to access all possible different processors, from a single program, and with the same code. Nevertheless, the center of the first release was GPGPU, since it is the most successful coprocessor or accelerator of today. Actually, there are several businesses that are developing implementations of OpenCL for FPGA's [Alt14, zii10], or their own parallel CPU architecture [Ada13]. Also, on version 1.2, OpenCL is providing access to fixed function circuits.

Given that massive compatibility, OpenCL has a hard tradeoff. It is very slow to write, since it requires to know a lot of concepts and to write a lot of code lines to manage all the OpenCL objects, their relationship, and their behavior. In this sense, SimpleOpenCL provides a way to write OpenCL code, without having to know anything about OpenCL objects or their functionality. This saves programming time. It is implemented using the functionality that OpenCL provides, plus basic ANSI C libraries. That way, SimpleOpenCL is as compatible as OpenCL, and so there is no need to modify compilers, operating systems or drivers in order to develop SimpleOpenCL.

7.2 SimpleOpenCL description

OpenCL host code is very complex and demands a lot of coding to do simple things. For that reason, we have developed SimpleOpenCL. It is a library that allows to forget about the OpenCL objects on the host code side of the program. With SimpleOpenCL the user only interacts with normal C pointers, and maps them to the memory hierarchy of the GPU in a single function call.

SimpleOpenCL provides the main functionality of automatic OpenCL object management. We provide this functionality through the following:

- SimpleOpenCL functions: this are functions that implement some typical steps and offload the programmer from this tasks.
- SimpleOpenCL data types: this are mainly two data types that encapsulate all the OpenCL objects in two main concepts, *hardware* and *software*.

Next we describe the SimpleOpenCL functions and data types.

7.3 SimpleOpenCL functions

We classify the SimpleOpenCL functions in different abstraction levels. We classified the functions in three levels, depending on their proximity to OpenCL or to the goals of SimpleOpenCL.

We defined this abstraction levels to help the project contributors to better understand our goals, and to better show that some functions serve some others inside the library, and should not be used directly by all users. In addition, there could be some third party libraries that offer advanced functionalities that increase execution performance and can be accessed from SimpleOpenCL from the adequate abstraction level (typically the second level). The levels are the following:

- Third level: at this level we use native OpenCL functions and what we call SimpleOpenCL third level functions. This third level functions perform basic tasks such as reading .cl source files and putting them on a char pointer, printing the error code names, etc...
- Second level: in this level we provide SimpleOpenCL second level functions that help with most common Host code tasks, reducing the code needed, but also gives most of the control to the programmer. Also this level is the place to use other libraries that give some advanced low level automatism or functionality. That is the case of GMAC [5] and AMD virtual memory support [7]. In SimpleOpenCL, GMAC would be considered a second level or maybe second and third level library, and would be used to improve first level function's performance. This changes don't affect the first level functions, so the user will get the benefits without changing it's code.
- First level: in this level we offer the functions that are the primary goal for this library. This functions make using OpenCL much more easy and short to write as shown on SimpleOpenCL web page [6].

With this abstraction layers we obtain a modular library. Furthermore, a program using exclusively first level functions would automatically get all the benefits of improving

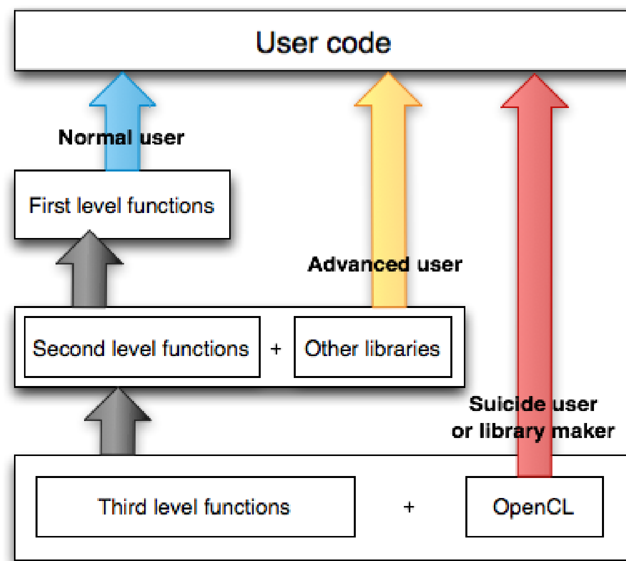


FIGURE 7.1: SimpleOpenCL programming levels.

```
cl_event sclManageArgsLaunchKernel( sclHard hardware,
                                    sclSoft software,
                                    size_t *global_work_size,
                                    size_t *local_work_size,
                                    const char* sizesValues,
                                    ... );
```

FIGURE 7.2: SimpleOpenCL first level function example.

SimpleOpenCL second level functions with or without third party libraries, without changing the user code. That's also the goal of first level functions, to give a reduced and stable set of functions with a stable set of parameters, that will automatically use whichever available low level improvements. The only requirement we put is that all features provided by SimpleOpenCL must be based on what the OpenCL standard provides and standard C, but must not be OS dependent. That means that we want SimpleOpenCL to only require an standard OpenCL installation in order to work. We are not hiding Device code complexity, so we expose the Device memory hierarchy to the programmer in the Host code side.

As shown in figure 5, the function sclManageArgsLaunchKernel takes as arguments an sclHard variable (a single device), a sclSoft variable (a single kernel function), the sizes for the NDRange space, a constant string and a variable number of undefined type arguments.

The last two arguments, are an string that contains the information of what are the arguments coming next. This is similar to the printf function. We arbitrarily defined which letters mean a type or a number of arguments expected as see on table 7.3. On SimpleOpenCL web page there is a detailed explanation of what is the syntax to specify

any possible argument. For some arguments OpenCL needs to know the size of the pointer, so when specifying an argument that is a pointer, the function is expecting two arguments, first the size, and then the pointer itself.

Character	Number of parameters	Action to be performed by SimpleOpenCL	Memory layer flags on the kernel code
%a	2 (size_t varSize,void* var)	When SimpleOpenCL finds %a, it reads a size_t argument, and a void* argument in this order. It is intended for non pointer arguments like int, float etc. size_t size is the variable size in bytes and void* var is the pointer to the variable. **Only for experienced OpenCL programmers** SimpleOpenCL only sets this variable as a kernel parameter without creating any buffer. For this reason, a% can be also used for any cl_mem variable manually controlled by the programmer.	__global, __constant
%N	1 (size_t varSize)	When SimpleOpenCL finds %N, it sets a local memory pointer with size "varSize"	__local
%w	2 (size_t varSize, void* var)	Wirte only. When SimpleOpenCL finds %w, it sets a global memory pointer with size "varSize". When the execution of the kernel is done, it automatically copies the results from the hardware to the variable "var".	__global
%r	2 (size_t varSize, void* var)	Read only. When SimpleOpenCL finds %r it sets a global memory pointer with size "varSize". Before the execution of the kernel, the contents of "var" are copied to the hardware device selected.	__global, __constant
%R	2 (size_t varSize, void* var)	Read write. When SimpleOpenCL finds %R it sets a global memory pointer with size "varSize". Before the execution of the kernel, the contents of "var" are copied to the hardware device selected. When the execution of the kernel is done, it automatically copies the results from the hardware to the variable "var".	__global
%g	1 (size_t varSize)	When SimpleOpenCL finds %g it sets a global memory pointer with size "varSize" in read write mode. Tis pointer will not be read or written from the main program.	__global

FIGURE 7.3: Table showing what characters to use on the string sent as parameter to the sclManageArgsLaunchKernel function.

This function takes as arguments standard C pointers, creates the necessary buffers, transfers the necessary data, orders kernel execution, and reads the needed results. This behavior is specified on the string argument. This string argument becomes then a sort of host code programming script, that can be enhanced to be able to offer more advanced programming options.

7.4 SimpleOpenCL types

One of the things we wanted to simplify is the management of the devices. We wanted to use a single data type to manage all needed to execute any kernel on one device. Also we wanted a function that can return an array of devices available on the system each one encapsulated in this type. The user must only need to initialize a variable of this data type and pass it to a function as a parameter, along with other parameters, and this should be everything needed to execute a kernel. No need to know about platforms, contexts, queues, device types etc etc etc.

For that purpose we created the following data types:

- sclHard: this is an struct that contains pointers to all the necessary OpenCL objects in order to access a device. The idea is to use this type as the only one required by any first or second level function. The initialization of this data type

takes into account other devices present on the system. Any of the hardware initialization functions uses (or is going to use) the sclGetAllHardware function that checks if two or more devices are from the same vendor and have the same characteristics in order to create a single context for all of them. This allows the devices to share memory objects. By now, this feature can be useful only from second level functions.

- sclSoft: this is also an struct that contains both the program and kernel objects for an OpenCL C source file, and an string with the name of the kernel function being used. The idea is to have an sclSoft variable for each kernel function.

Bibliography

[Ada] Adapteva: *Epiphany SDK Reference Manual: http://adapteva.com/docs/epiphany*$_s dk_r ef.pdf.Tech.rep., Adapateva.$

[Ada13] Adapteva: *Epiphany Architecture Reference: http://adapteva.com/docs/epiphany*$_a rch_r ef.pdf.Tech.rep., Adpateva, 2008-$ 2013.

[Alt14] Altera: *Altera OpenCL webpage: http://newsroom.altera.com/press-releases/nr-opencl-v14-0-altera.htm?GSA*$_p os = 1WT.oss_r = 1WT.oss = opencl.Tech.rep., Altera, 2014.$

[App09] Apple: *http://developer.apple.com/library/mac/#documentation/Performance/Referen* Tech. rep., Apple, 2009.

[APT08] Abellan P., Puig A., Tost D.: Focus+context rendering of structured biomedical data. In *EG VCBM 2008 : Eurographics workshop on visual computing for biomedicine* (2008), pp. 109–116.

[ASS02] Allwein E., Schapire R., Singer Y.: Reducing multiclass to binary: A unifying approach for margin classifiers. *JMLR 1* (2002), 113–141.

[BG08] Bell N., Garland M.: *Efficient Sparse Matrix-Vector Multiplication on CUDA*. NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation, Dec. 2008.

[BHW*07] Burns M., Haidacher M., Wein W., Viola I., Groeller E.: Feature emphasis and contextual cutaways for multimodal medical visualization. In *Eurographics / IEEE VGTC Symposium on Visualization 2007* (May 2007), pp. 275–282.

[BMD*11] Bueno J., Martinell L., Duran A., Farreras M., Martorell X., Badia R., Ayguade E., Labarta J.: Productive cluster programming with ompss. In *Euro-Par 2011 Parallel Processing*, Jeannot E., Namyst R., Roman J., (Eds.), vol. 6852 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2011, pp. 555–566.

[Bol01] Boland V.: Affinity scheduling of data within multi-processor computer systems, July 31 2001. US Patent 6,269,390.

[BPS97] Bajaj C. L., Pascucci V., Schikore D.: The contour spectrum. In *IEEE Visualization'97* (1997), pp. 167–174.

[BSC] BSC:. //http://pm.bsc.es/ompss-docs/specs/OmpSsSpecification.pdf.

[DK95] Dietterich T., Kong E.: Error-correcting output codes corrects bias and variance. In *S. Prieditis and S. Russell* (1995), of the 21th International Conference on Machine Learning P., (Ed.), pp. 313–321.

[dlCAPC10] de la Cruz R., Araya-Polo M., Cela J.: Introducing the semi-stencil algorithm. In *Parallel Processing and Applied Mathematics* (2010), vol. 6067, pp. 496–506.

[dSOG10] d. S. Oliveira B. C., Gibbons J.: Scala for generic programmers. *Journal of Functional Programming 20*, 3,4 (2010), 303–352. Revised version of the WGP2008 paper.

[EPR10] Escalera S., Pujol O., Radeva P.: On the decoding process in ternary error-correcting output codes. *IEEE Transactions on Pattern Analysis and Machine Intelligence 32* (2010), 120–134.

[ETP*08] Escalera S., Tax D., Pujol O., Radeva P., Duin R.: Subclass problem-dependent design of error-correcting output codes. In *IEEE Transactions in Pattern Analysis and Machine Intelligence* (2008), vol. 30, pp. 1–14.

[FHT98] Friedman J., Hastie T., Tibshirani R.: Additive logistic regression: a statistical view of boosting. *Annals of Statistics 28* (1998).

[FPT06] Ferré M., Puig A., Tost D.: Decision trees for accelerating unimodal, hybrid and multimodal rendering models. *The Visual Computer 3* (2006), 158–167.

[FWG09] Fuchs R., Waser J., Gröller M. E.: Visual human-machine learning. *IEEE TVCG 15*, 6 (Oct. 2009), 1327–1334.

[GDB08] Garcia V., Debreuve E., Barlaud M.: Fast k nearest neighbor search using gpu.

[GLD*08] Govindaraju N. K., Lloyd B., Dotsenko Y., Smith B., Manferdelli J.: High performance discrete fourier transforms on graphics processors. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing* (Piscataway, NJ, USA, 2008), SC '08, IEEE Press, pp. 2:1–2:12.

[GMK*92] Gerig G., Martin J., Kikinis R., Kubler O., Shenton M., Jolesz F.: Unsupervised tissue type segmentation of 3-d dual-echo mr head data. *Image and Vision Computing 10*, 6 (1992), 349–36.

[HGLS07] He B., Govindaraju N. K., Luo Q., Smith B.: Efficient gather and scatter operations on graphics processors. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing* (New York, NY, USA, 2007), SC '07, ACM, pp. 46:1–46:12.

[HSSY00] Hamscher V., Schwiegelshohn U., Streit A., Yahyapour R.: Evaluation of job-scheduling strategies for grid computing. In *Grid Computing — GRID 2000*, Buyya R., Baker M., (Eds.), vol. 1971 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2000, pp. 191–202.

[HWS10] Herrero S., Williams J., Sanchez A.: Parallel multiclass classification using svms on gpus. In *ACM International Conference Proceeding Series, Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units* (2010), vol. 425, pp. 2–11.

[JD04] Jeffrey Dean S. G.: Mapreduce: Simplified data processing on large clusters. *OSDI'04: Sixth Symposium on Operating System Design and Implementation, San Francisco, CA, December, 2004.* (2004).

[JDM00] Jain A. K., Duin R. P., Mao J.: Statistical pattern recognition: A review. *IEEE Transactions on Pattern Analysis and Machine Intelligence 22* (2000), 4–37.

[KBKG08] Kohlmann P., Bruckner S., Kanitsar A., Gröller M.: *The LiveSync Interaction Metaphor for Smart User-Intended Visualization.* Tech. Rep. TR-186-2-08-01, Institute of Computer Graphics and Algorithms, Vienna University of Technology, Favoritenstrasse 9-11/186, A-1040 Vienna, Austria, Jan. 2008.

[KD98] Kindlmann G., Durkin J.: Semi-automatic generation of transfer functions for direct volume rendering. In *IEEE Symposium on Volume Visualization* (October 1998), IEEE Press, pp. 79–86.

[KH10] Kirk D. B., Hwu W.-m. W.: *Programming Massively Parallel Processors: A Hands-on Approach*, 1st ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2010.

[KKH01] Kniss J., Kindlmann G., Hansen C.: Interactive volume rendering using multi-dimensional transfer functions and direct manipulation widgets. In *Proc. of the conference on Visualization 2001* (2001), IEEE Press., pp. 255–262.

[KSW06] Krüger J., Schneider J., Westermann R.: Clearview: An interactive context preserving hotspot visualization technique. *IEEE Trans. on Visualization and Computer Graphics (Proc. Visualization / Information Visualization 2006) 12*, 5 (sep- oct 2006).

[MAB*97] Marks J., Andalman B., Beardsley P. A., Freeman W., Gibson S., Hodgins J., Kang T., Mirtich B., Pfister H., Ruml W., Ryall K., Seims J., Shieber S.: Design galleries: a general approach to setting parameters for computer graphics and animation. *Computer Graphics 31*, Annual Conference Series (1997), 389–400.

[MB05] Manvi S. S., Birje M. N.: An agent-based resource allocation model for grid computing. In *Services Computing, 2005 IEEE International Conference on* (July 2005), vol. 1, pp. 311–314 vol.1.

[Mic09] Micikevicius P.: 3d finite difference computation on gpus using cuda. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units* (2009), pp. 79–84.

[mon11] 2011. http://montblanc-project.eu/home.

[mov] //movibio.lsi.upc.edu/website.

[NL07] Nyland L. Harris M. P. J.: *Fast n-body simulation with CUDA*. 2007.

[NVI12] NVIDIA: Cuda dynamic parallelism programming guide.

[par13] 2013. http://www.parallella.org.

[PLB*01] Pfister H., Lorensen B., Baja C., Kindlmann G., Shroeder W., Avila L., Raghu K., Machiraju R., Lee J.: The transfer function bake-off. *IEEE Computer Graphics & Applications 21*, 3 (2001), 16–22.

[Rob] Roberts M.:. //www.cs.colostate.edu/$\sim mroberts/toolbox/c++/$ $sparseMatrix/sparse_matrix_compression.html$.

[SHL*06] Shi Z., Huang H., Luo J., Lin F., Zhang H.: Agent-based grid computing. *Applied Mathematical Modelling 30*, 7 (2006), 629 – 640. Parallel and Vector Processing in Science and Engineering Parallel and Vector Processing in Science and Engineering.

[TLM03] Tzeng F. Y., Lum E., Ma K. L.: A novel interface for higher dimensional classification of volume data. In *Visualization 2003* (2003), IEEE Computer Society Press, pp. 16–23.

[TM04] Tzeng F.-Y., Ma K.-L.: A cluster-space visual interface for arbitrary dimensional classification of volume data. In *Proceedings of the Joint Eurographics-IEEE TVCG Symposium on Visualization 2004* (May 2004).

[TTG95] Torrellas J., Tucker A., Gupta A.: Evaluating the performance of cache-affinity scheduling in shared-memory multiprocessors. *Journal of Parallel and Distributed Computing 24*, 2 (1995), 139 – 151.

[VKG05] Viola I., Kanitsar A., Gröller M. E.: Importance-driven feature enhancement in volume visualization. *IEEE Trans. on Visualization and Computer Graphics 11*, 4 (2005), 408–418.

[YSMR10] Yudanov D., Shaaban M., Melton R., Reznik L.: Gpu-based simulation of spiking neural networks with real-time performance and high accuracy. In *WCCI-2010, Special Session Computational Intelligence on Consumer Games and Graphics Hardware CIGPU-2010* (2010).

[zii10] *Ziilabs official OpenCL webpage: http://www.ziilabs.com/products/software/opencl.php.* Tech. rep., Ziilabs, 2010.