UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

POSGRADO EN CIENCIA E INGENIERÍA DE LA COMPUTACIÓN

"A PARALLEL BIOINSPIRED WATERMARKING ALGORITHM ON A GPU"

T H E S I S

AS A FULFILMENT OF THE REQUIREMENT FOR THE DEGREE OF:

MASTER IN SCIENCES
(COMPUTER)

B Y:

EDGAR EDUARDO GARCÍA CANO CASTILLO

ADVISOR:

DRA. KATYA RODRIGUEZ VAZQUEZ

México, D.F. 2012.

UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

POSGRADO EN CIENCIA E INGENIERÍA DE LA COMPUTACIÓN

"ALGORITMO DE MARCA DE AGUA BIOINSPIRADA EN PARALELO EN UNA GPU"

T E S I S

QUE PARA OBTENER EL GRADO DE:

MAESTRO EN CIENCIAS
(COMPUTACIÓN)

P R E S E N T A:

EDGAR EDUARDO GARCÍA CANO CASTILLO

DIRECTOR DE TESIS:

DRA. KATYA RODRÍGUEZ VÁZQUEZ

México, D.F. 2012.

Este trabajo fue realizado gracias a los apoyos recibidos por parte del Consejo Nacional de Ciencia y Tecnolog´ıa (CONACYT), con la beca de posgrado nacional númer

Abstract

In this thesis, I'm presenting a research about the usability, advantages and disadvantages of using CUDA architecture to implement algorithms based on populations, specifically Parti- cle Swarm Optimization (PSO). Nowadays it is not necessary to invest in clusters, since it is enough to have a video card -as the ones from NVIDIA- that has a lot of cores in just one GPU, and takes advantage of this parallelism.

In order to test the performance of the algorithm, a hide watermark image application is implemented, and the PSO is used to optimize the positions where the watermark has to be inserted. This application uses the insertion/extraction algorithm proposed by Shieh et al. The whole algorithm was implemented for both sequential and CUDA architectures. The CUDA version of the watermarking-PSO algorithm takes advantage of the parallelism, where the fitness function is the union of two objectives: fidelity and robustness. The measurement of fidelity and robustness is computed by using Mean Squared Error (MSE) and Normalized Correlation (NC) respectively; these functions are evaluated using Pareto dominance.

The first chapter introduces watermarks, what they are and explains the two types of wa- termarks: visible and invisible. It also includes a perspective about what CUDA architecture is, how it was born and what it is used for nowadays. Later it gives an introduction about what Evolutionary and Bioinspired Algorithms are.

The second chapter gives an overview of Discrete Cosine Transform (DCT) applied to insert the watermark images. In addition to this method, are explained two watermarking metrics: watermarking fidelity and watermarking robustness. The fidelity represents the sim- ilarity of the watermarked image with the original image and the robustness represents the resistance of the watermark against manipulations applied on the watermarked image. The third chapter -related with the second one- explains the different types or watermark attacks. The attacks are applied to test the robustness of the watermarked image.

The fourth chapter explains the main CUDA features such as the architecture, how to or- ganize the data in the GPU, how to do the thread assignment to take advantage of parallelism, beside the different memory types such as: global, constant, registers and shared.

The fifth chapter gives in detail the steps that are involved in the Shieh algorithm, which is used to insert and extract the watermark image. In few words, this algorithm makes use of the DCT domain by splitting the original image in blocks of 8x8, then a ratio matrix between DC and AC coefficients is calculated. The next step is to compute the relation between the image content and the frequency bands where the watermark will be inserted; finally Inverse Discrete Cosine Transform (IDCT) is performed to get the watermarked image.

The sixth chapter introduces the theory about Particle Swarm Optimization (PSO), which is based on particles that fly through the problem space trying to find a solution each time step. To do this the particle moves are based in velocity and position vectors that change with time. To know if a particle is near to a solution, a fitness value must be calculated. In the case of this work, the fitness value is composed of two objectives: fidelity and robustness. These aims are evaluated using Pareto dominance whose theory is explained in chapter seven.

The chapter eighth finally links the whole theory seen in previous chapters to give life to the optimization algorithm applied in the watermark insertion. The algorithm is based on the Shieh and the PSO algorithms.

Finally, test, results and conclusions are exposed in chapters nine and ten.

Derived works

- A research paper submitted to the 3erd International Supercomputing Conference in Mexico (ISUM) to be hold in March 2012.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The goal of the present work is to research and analyze bioinspired algorithms applied to a watermarking insertion algorithm, using the parallel paradigm on Graphics Processing Units (GPUs), specifically based on Compute Unified Device Architecture (CUDA). The first part concerns with the implementation of the watermarking algorithm; this was carried out in a research stay at the Ecole de Technologie Sup´ erieure (ETS), Universit´ du Qu´ ebec in Canada, under Professor Robert Sabourin's and PhD student Bassem Guendy's supervision. The sec- ond part considers the bioinspired algorithm implementation and the integration with the wa- termarking algorithm. The second part was under the supervision of Dra. Katya Rodr´ıguez V´ azquez.

Digital watermarking came to be in great demand when sharing information on the Internet became a usual practice. When sharing files online, you never know if someone uses them without your consent.

A digital watermark is a pattern of bits inserted into a digital file such as an image, an audio or a video. Such patterns usually carry copyright information of the file. Digital water- marking takes its name from the faintly visible watermarks imprinted on paper to identify a manufacturer, an enterprise, a school, etc. In digital watermarking the objective is to provide copyright protection in digital files.

When speaking of digital image watermarking, we can divide watermarks into two main groups: visible and invisible watermarks.

A *visible* watermark is a visible semi-transparent text or image overlaid on the original image. It allows the original image to be viewed, but it still provides copyright protection by marking the image as its property. Visible watermarks are more robust against image transformation (especially if you use a semi-transparent watermark placed over the whole image). Thus they are preferable for strong copyright protection of intellectual property in digital for- mat.

An *invisible* watermark is an embedded image which cannot be perceived with human eyes. Only electronic devices (or specialized software) can extract the hidden information to identify the copyright owner. Invisible watermarks are used to mark a specialized digital content (text, images or even audio content) to prove its authenticity [2].

A GPU is a processor dedicated to graphics processing, to lighten the workload of the central processor in applications such as video games and interactive 3D applications. On this way, while much of the load related to the graphics processing is executed on the GPU, the CPU can focus on other calculations.

Using GPUs is possible to perform tasks more efficiently, which are optimized for floating point calculations. Therefore, a good strategy is to use brute force on the GPUs to complete more calculations at the same time.

In order to program the GPU, several languages can be used, among them C using CUDA extension, OpenCL, Fortran, Java, etc. CUDA is a parallel computing architecture of NVIDIA that allows a significant increase in performance of the calculations thanks to the power of the GPU.

With thousands of GPUs, software developers, scientists and researchers are finding opportunities to use CUDA, for example in image and video processing, biology and computational chemistry, simulation of fluid dynamics , the reconstruction of tomographic images, seismic analysis, evolutionary computation and more.

Currently, evolutionary computation makes use of models based on the natural evolution process, designing and implementing algorithms for solving problems. There are a large variety of proposals and studies on these models, which are called with the generic name of Evolutionary Algorithms. These have common features such as the inspiration in the simulation of the evolution of populations of individuals through processes of selection and reproduction.

Another set of proposals inspired by biological models, such as optimization algorithms based on Ant Colony and Swarm-based algorithms are classified into what has been called bioinspired algorithms, a new way to solve problems based on the behavior of animals or systems that take centuries to evolution.

1.1 Motivation

Due to the impossibility to control the information that goes through Internet, there is a need to protect our information from unauthorized copying or to legitimate our ownership over it, and the invisible watermarking comes out as an option that -combined with an optimization mechanism such as the bioinspired algorithm PSO-, provides a highly suitable tool for this purpose.

In recent years, new and cheaper technologies such as CUDA architecture have emerged with the concept of massive parallelism. Due to this new paradigm, it is not necessary to in- vest in expensive clusters, since it is enough to have a video card -like the ones from Nvidia- that have a lot of cores in just one GPU, and take advantage of its massive parallelism.

The combination of the bioinspired and the watermarking algorithms using the new mas- sive parallelism paradigm on GPUs to accelerate the process came out as a curiosity for me and became the motivation of the research in this work.

1.2 Contributions

The contributions of this thesis are:

- A proposal on how to implement a watermark optimization using Particle Swarm Optimization (PSO) on GPUs. In this proposal each block generated in Discrete Cosine Transform (DCT) is taken as a swarm. For each swarm, N particles are created, and these particles have part of the total solution. The particles fitness is measured by using mean squared error (MSE) and normalized correlation (NC); these are the two objetives that are evaluated using Pareto dominance.

- Two implementations of the optimization algorithm, one sequential and other that uses CUDA architecture. Those implementations help to compare the efficiency and speed up of the two different architectures, and to know which of them is more convenient to be used in algorithms based in populations.

1.3 Outline

The main theory for watermarking using the Shieh algorithm combined with PSO and the CUDA architecture is presented in the following chapters of this thesis.

- Chapter 2 presents the DCT theory as one of the main elements to embed a watermark, besides the metrics used to evaluate it.

- Chapter 3 decribes the watermarking attacks and the one used in the optimization algorithm.
- Chapter 4 explains the main features of the CUDA architecture, thread assigment, thread scheduling, device memory and some of the best practices to develop software with CUDA.
- Chapter 5 presents the details for the implementation of the algorithm proposed by Shieh *et al* [6].
- Chapter 6 describes the foundations of the PSO algorithm.
- Chapter 7 explains the foundations of the multiobjetive optimization.
- Chapter 8 explains how the whole algorithm (watermarking + PSO) was implemented.
- Chapter 9 presents the tests and results of the thesis.
- Chapter 10 draws the conclusions of this research work.

Chapter 2

Transform Methods for Watermarking

There are different types of transformations used in image watermarking such as Discrete Cosine Transform (DCT), Discrete Wavelet Transform (DWT), and Discrete Multiwavelet Transform (DMT).

DCT is commonly used in MPEG and JPEG as an orthogonal transform. In the DCT domain, the energy could be concentrated in the low frequency regions around the upper-left corner (see figure 2.1), but depending of the convention the energy could be concentrated in the center or in the other corners.

Figure 2.1: Original Lena image (left) and transform coefficients of Lena image obtained by DCT.

DWT decomposes the image into different frequency bands and still retains its spatial information. In wavelet watermarking techniques, since the DWT of an image gives multiresolution sampling, the watermark ends up being robust to downsampling operations.

DMT is relatively a new type of signal transform that is commonly used in image compression. The main motivation of using multiwavelet is that it is possible to construct multiwavelets that simultaneously possess desirable properties such as orthogonality, symmetry and compact support with a given approximation order [16].

At the EST -where I made a research stay-, Professor Robert Sabourin and his collabora- tors were working on a project to apply the watermark process in financial banking document

like checks, invoices and bills. The process to digitized the physical document is made using as equipment a scanner. The digital files are acquired by the scanner in grey scale, that is why the work focuses in the use of grey scale images. The client, Banctec needs to have digitized and watermarked tens of millions of documents per day, and that is why they need a rapid method to watermark a huge quantity of documents.

In addition, Professor Sabourin's team decided to apply DCT because small changes in some frequency bands are visually imperceptible. Moreover, JPEG and MPEG compression are based on DCT and with such method the watermark ends up being resistant against com- pression.

2.1 DCT Theory

The Discrete Cosine Transform is a Fourier-like transform, which was first proposed by Ahmed *et al.* (1974). While the Fourier Transform represents a signal as the mixture of sines and cosines, the Cosine Transform performs only the cosine-series expansion. The pur- pose of DCT is to perform the decorrelation of the input signal and to present the output in the frequency domain. The DCT is known for its high "energy compaction" property, meaning
that the transformed signal can be easily analyzed using few low-frequency components.

This fact made it widely used in digital signal processing.The most popular DCT is the two-dimensional symmetric variation of the transform that operates on 8x8 blocks and its in- verse. The two-dimensional input signal is divided into the set of nonoverlapping 8x8 blocks and each block is independently processed. This makes it possible to perform the block-wise transform in parallel.

The formal definition for DCT of two-dimensional for a sample of size $N \cdot N$ is defined as follows:

$$C(u, v) = \alpha(u)\alpha(v) \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} f(x, y) \cos \frac{\pi(2x+1)u}{2N} \cos \frac{\pi(2y+1)v}{2N} \quad (2.1)$$

The inverse of two-dimensional DCT for a sample of size $N \cdot N$ is:

$$f(u, v) = \sum_{u=0}^{N-1} \sum_{v=0}^{N-1} \alpha(u)\alpha(v)C(u, v) \cos \frac{\pi(2x+1)u}{2N} \cos \frac{\pi(2y+1)v}{2N} \quad (2.2)$$

where $u, v = 0, 1, ..., N-1$, also $x, y = 0, 1, ..., N-1$, and

$$\alpha(u) = \begin{cases} \sqrt{\frac{1}{N}} & \text{if } u = 0; \\ \sqrt{\frac{2}{N}} & \text{if } u = 0. \end{cases} \tag{2.3}$$

As it can be seen from 2.3, if $u = 0$ then $C(0) = \frac{1}{N}\sum_{x=0}^{N-1} f(x)$. By convention, this value is called the DC coefficient of the transform and the other are referred to as AC coefficients[11].

2.2 Watermarking Metrics

In the digital framework, watermarking algorithms that make use of information hiding techniques have been developed and hiding capacity has naturally been used as a metric in evaluating their power to hide information (the maximal amount of information that a certain algorithm can "hide" keeping the data within allowable distortion bounds).

2.2.1 Watermark Fidelity

The fidelity represents the similarity of the watermarked image with the original image. Peak Signal to Noise Ratio ($PSNR$) is commonly used to evaluate image degradation or reconstruction fidelity. It is defined for two images I and K of size $M \cdot N$ as:

$$PSNR(I, K) = 10 \log_{10} \frac{255^2}{MSE(I, K)} \tag{2.4}$$

Where I is the original image, K is a reconstructed or noisy approximation, 255^2 is the maximum pixel value in image I and MSE is a mean square error between I and K.

$$MSE(I, K) = \frac{1}{MN} \sum_{i=0}^{M-1} \sum_{j=0}^{N-1} \left[I(i, j) - K(i, j) \right]^2 \tag{2.5}$$

PSNR is expressed in decibel scale. In image reconstruction typical values for PSNR vary within the range [30, 50]. A PSNR value of 50 and higher calculated from two images that were processed on diverse devices with the same algorithm indicates that the results are practically identical.

2.2.2 Watermark Robustness

The robustness represents the resistance of the watermark against attacks -compression, rotation, scaling, etc. (detailed attacks are described in chapter 3)- done on the watermarked

image. The Normalized Correlation (NC) is used to measure the robustness between the original watermark and the extracted watermark. When different attacks have been applied to a watermarked image, the NC is calculated between the embedded watermark $W(i,j)$ and the extracted watermark from the attacked image $W(i,j)$, where both watermarks have the same dimensions $M_w \cdot N_w$.

$$NC = \frac{\sum_{i=1}^{M_W} \sum_{j=1}^{N_W} [W(i,j)W(i,j)]}{\sum_{i=1}^{M_W} \sum_{j=1}^{N_W} [W(i,j)]^2} \tag{2.6}$$

2.2.3 Watermark Capacity

Determining the capacity of a watermark in an image is to find how much information can be hidden in a digital image without perceptible distortion while maintaining its robustness [20].

Image watermarking capacity is a complex problem that may be influenced by many factors. The content of the image has as much influence in the capacity as the watermark strength. But higher strength in a watermark not always means higher watermark capacity. For example if we add ten units instead of one unit to the gray level value for each pixel in order to insert one bit of watermark, the strength becomes much higher, but the capacity remains the same [17].

2.3 Functions used in this thesis

Generally, the watermark is measured and characterized using three aspects, i.e. fidelity, robustness and capacity. There is a need to fix the capacity and to maximize both fidelity and robustness to reach a better watermarking characteristics system. Professor Sabourin's team decided to start working with fidelity and robustness as a first version of the application.

Chapter 3

Watermarking Attacks

Digital image watermarking has become a popular technique for authentication and copyright protection. In order to verify the security and robustness of watermarking algorithms, spe- cific attacks have to be applied to test them. A list of most common attacks is given as follows.

I. JPEG Compression - JPEG is currently one of the most widely used compression algorithms for images.

II. Geometric transformations

1) Flip - The image looks, as if it has been reflected along the central horizontal or vertical axis of the layer.
2) Rotation - It is used to move in some angle the image, it is used to straighten an image once it was scanned.
3) Cropping - It refers to an unwanted part of the image that is removed, to focus in a particular object.
4) Scaling -When a image is resized, sometimes the image is enlarged or reduced to fit in an specific place. The scaling could be applied in horizontal, vertical or both directions.

III. Enhancement techniques

1) Low pass filtering - The simplest operation to calculate it, is the average of a pixel and all of its eight immediate neighbors. The result replaces the original value of the pixel. Every pixel repeat the same process. This effect is also called blurring or smoothing.
2) Histogram modifications - This includes histogram stretching or equalisation which are sometimes used to compensate poor lightening conditions.[15]
3) Sharpening - It is used to increase the contrast between each pixel and its neighbors. The image must be blurring as first step, then the original and the blurred

version image are compare pixel by pixel. If a pixel is brighter than the blurred version it is lightened further; if a pixel is darker than the blurred version, it is darkened.

4) Gamma correction - Gamma correction is used to control the overall brightness of an image. This effect is used when the image is too dark.

5) Restoration - Sometimes it is necessary to reduce an specific degradation process (blur, noise, camera misfocus, etc.) in the image, this technique is used to reduce ("compensate for" or "undo") the effects of that degradation.

3.1 Examples of Attacks

This section explains some attacks considered by the Shieh algorithm for robustness [6].

3.1.1 JPEG Compression

The name "JPEG" stands for Joint Photographic Experts Group, the name of the committee that created the JPEG standard and also other standards. The JPEG compression algorithm is used with photographs and paintings of realistic scenes with smooth variations of tone and color. For web usage, where the amount of data used for an image is important, JPEG is very popular.

JPEG is based on a lossy compression method, which somewhat reduces the image fi- delity. This method discards (loses) some data in order to achieve its goal, with the result that decompressing the data yields content that is different from the original, though similar enough to be useful in some way.

3.1.2 Low Pass Filtering

Applying a low pass filter on 2D image in the frequency domain means zeroing all frequency components above a cutoff frequency. The result is transformed back into the spatial domain.

3.1.3 Median Filtering

The median filter is a nonlinear digital filtering technique, often used to remove noise. The main idea of the median filter is to run through the signal entry by entry, replacing each entry with the median of neighboring entries. The pattern of neighbors is called the "window", which slides, one entry at a time, over the entire signal.

3.2 Attack used in this thesis

In the present work, "quantization" is used as a watermarking attack. This attack was applied because, since it is already part of the CUDA libraries, it was not necessary to program it, and also because of its ease of use. Quantization is a method that can be added to the insertion/extraction algorithm although it is not intrinsic to it.

Just one attack was used to test the optimization algorithm (see chapter 8) considering that only one type of attack was sufficient to determine its performance. Nevertheless, other attacks might be implemented to test the algorithm further, which is a proposal for future updates of this application.

Quantization is applied to reduce the number of colors utilized in images; this technique is implemented on devices that support a limited number of colors and for efficient compres- sion, it makes possible to reduce the file size.

In quantization, the compression rate depends on the number of coefficients that are non- zero after quantization has been performed. If a compression rate of 75 percent (of the initial size) is required, 25 percent of least valuable coefficients should be zero after the quantization step.

Chapter 4

CUDA Architecture

In november 2006, NVIDIA introduced CUDA, a new general purpose parallel computing architecture with a new programming model and an instruction set architecture, a tool to develop scientific programs oriented to massively parallel computation. It is actually sufficient to install a compatible GPU and the CUDA SDK, even in a low end computer to develop a parallel program using a high level language as C.

CUDA's programming model requires that the programmer splits the problem under consideration into many independent subtasks which can be solved in parallel. Each subproblem may be further divided into many tasks, which can be solved cooperatively in parallel too. In CUDA terms, each subproblem becomes a thread block, each thread block being com- posed of a certain number of threads which cooperate to solve the subproblems in parallel. The software element that describes the instructions to be executed by each thread is called kernel. When a program running on the CPU invokes a kernel, the number of corresponding thread blocks and the number of threads per thread block must be specified. The abstraction on which CUDA is based allows a programmer to define a two dimensional grid of thread blocks; each block is assigned a unique pair of indixes that act as its coordinates within the grid. The same mechanism is available within each block: the threads that compose a block can be organized as a two or three dimensional grid. Again, a unique set of indixes is pro- vided to assign each thread a 'position' within the block. This indexing mechanism allows each thread to personalize its access to data structures and, in the end, achieve effective prob- lem decomposition [7].

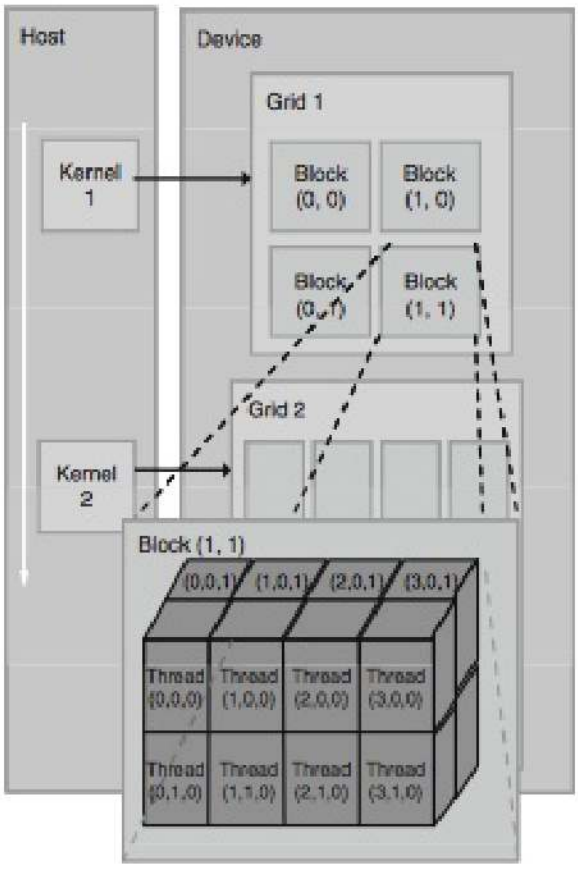

Figure 4.1: A multidimensional example of CUDA grid organizations [13].

A Graphics Processing Unit (GPU) is a processor dedicated to graphics processing in or- der to lighten the workload of the central processor in applications such as video games and interactive 3D applications. On this way, while much of the related to the graphics processing is executed on the GPU, the CPU can focus on other calculations.

The expertise of GPUs can perform tasks more efficiently, which are optimized for floa- ing point calculations. Therefore, a good strategy is to use brute force on the GPUs to com- plete more calculations at the same time. To program the GPU we can use several languages, such as C using CUDA extension, OpenCL, Fortran, Java, etc.

With thousands of GPUs, software developers, scientists and researchers are finding op- portunities to use CUDA. For example in image and video processing, biology and compu- tational chemistry, simulation of fluid dynamics, the reconstruction of tomographic images, sismic analysis, evolutionary computation and more.

4.1 Thread Assigment

The GPU is made up of a scalable array of multithreaded Streaming Multiprocessors (SMs), each of which is able to execute several thread blocks at the same time. When the CPU orders the GPU to run a kernel, thread blocks are distributed to free SMs and all the threads of a scheduled block are executed concurrently.

One key aspect about SMs is their ability to manage hundreds of threads running different code segments: in order to do so they employ an architecture called SIMT (Single Instruc- tion, Multiple Thread) which creates, manages, schedules, and executes groups (warps) of 32 parallel threads [7].

The runtime system maintains a list of blocks that needs to be executed and assigns new blocks to SMs as they complete the execution of blocks previously assigned to them.

Figure 4.2 shows an example in which three thread blocks are assigned to each SM. One of the SM resource limitations is the number of threads that can be simultaneously tracked and scheduled. Hardware resources are required for SMs to maintain the thread, block IDs, and track their execution status [13].

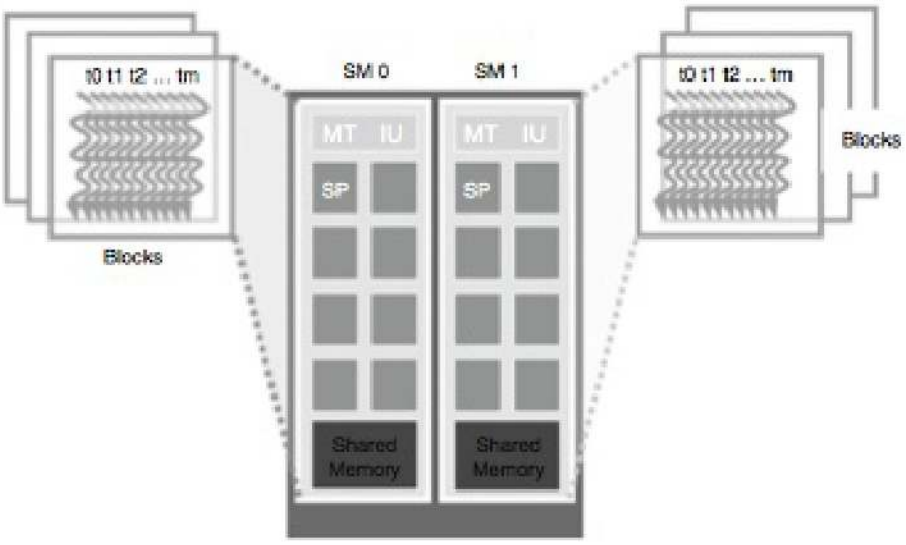

Figure 4.2: Thread block assigment to streaming multiprocessors (SMs) [13].

4.2 Thread Scheduling and Latency Tolerance

Once a block is assigned to a streaming multiprocessor, it is further divided into 32-thread units called warps. The size of the warps is implementation specific. In fact, warps are not part of the CUDA specification; however, knowledge of warps can be helpful in under- standing and optimizing the performance of CUDA applications on particular generations of CUDA devices. The warp is the unit of thread scheduling in SMs.

Each warp consists of 32 threads of consecutive threadIdx values: Threads 0 through 31 form the first warp, threads 32 through 63 the second warp, and so on. When an instruction executed by the threads in a warp must wait for the result of a previously initiated long- latency operation, the warp is not selected for execution. Another resident warp that is no longer waiting for results is selected for execution. If more than one warp is ready for execu- tion, a priority mechanism is used to select one for execution. This mechanism of filling the latency of expensive operations with work from other threads is often referred to as latency hiding.

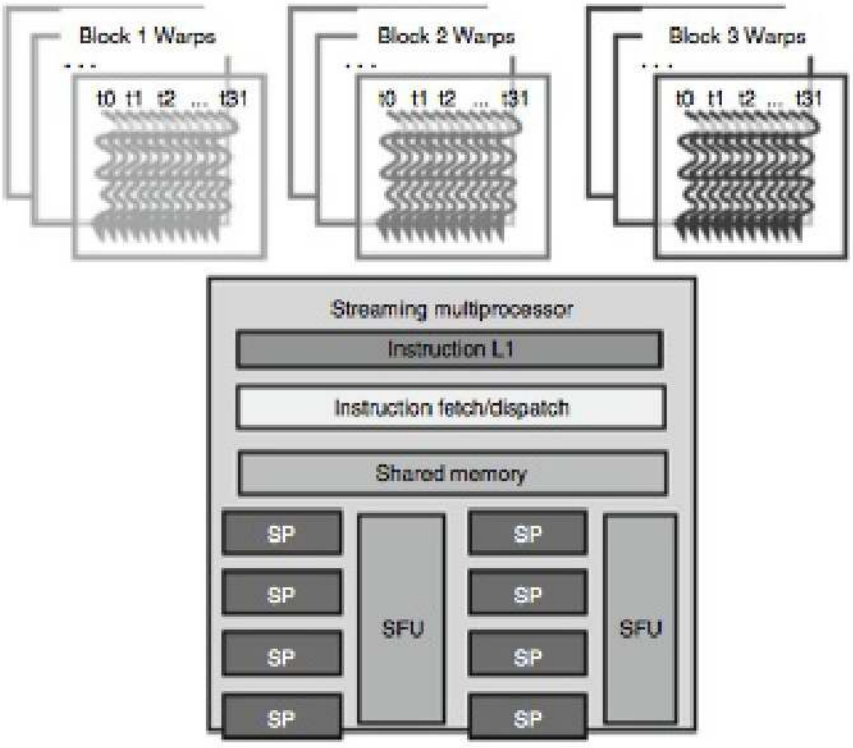

Figure 4.3: Blocks partitioned into warps for threading scheduling.

With enough warps around, the hardware will likely find a warp to execute at any point in

time, thus making full use of the execution hardware in spite of these long-latency operations [13]. The figure 4.3 shows the division of blocks into warps.

4.3 CUDA Device Memories

CUDA supports several types of memory that can be used by programmers. These types of memories can be written (W) and read (R) by the host by calling application programming interface (API) functions. In figure 4.4 we can see the memory model used by CUDA.

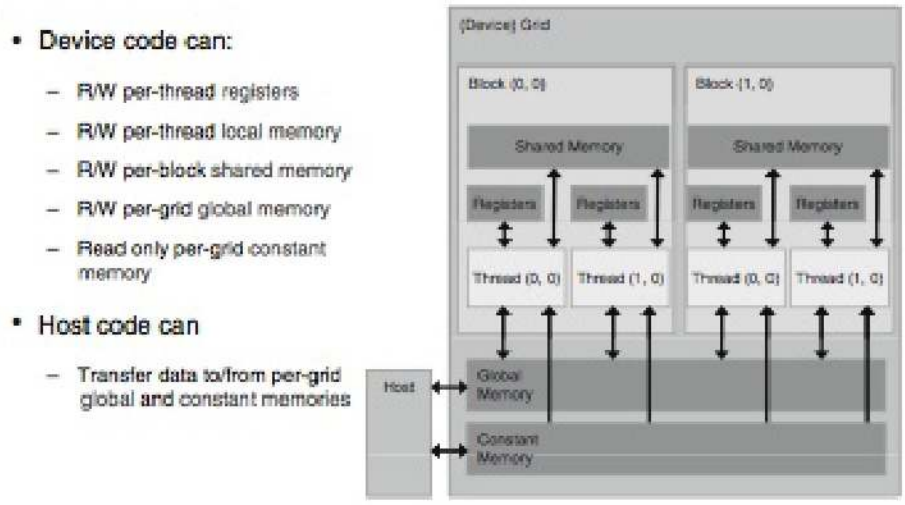

Figure 4.4: CUDA device memory model [13].

4.3.1 Global Memory

The global memory is implemented with dynamic access memory (DRAM), it has long ac- cess latencies and finite access bandwidth.

4.3.2 Constant Memory

The constant memory supports short latency, high bandwidth, and read only access -by the device- when all threads simultaneosly access the same location.

4.3.3 Registers

These are located on the chip memories. Variables that resides these type of memory can be accessed at very high speed in a highly parallel manner. Registers are allocated to individual threads; each thread can only access its own registers.

A kernel function uses registers to hold frequently accessed variables that are private to each thread.

4.3.4 Shared Memory

It is allocated to threads blocks; all the threads in blocks can access variables in the shared memory locations allocated by the block.

Shared memory is an efficient means for threads to cooperate by sharing their input data and the intermediate results of their work.

4.4 CUDA Events

An event in CUDA is essentially a GPU time stamp that is recorded at a user specified point in time. Since the GPU itself is recording the time stamp, it eliminates a lot of problems we might encounter when trying to time GPU executions with CPU timers. A time stamp con- sists of just two steps: creating an event and subsequentialy recording an event. The trickiest part of using events arises as a consequence of the fact that some of the calls we make in CUDA are actually asynchronous [9].

4.5 CUDA Best Practices

In order to obtain the best performance from this architecture, a number of specific programming guidelines should be followed, the most important of which are:

I. Minimize data transfers between the host and the graphics card

II. Minimize the use of global memory: shared memory should be preferred

III. Avoid different execution paths within the same warp

Moreover, each kernel should reflect the following structure:

I. Load data from global/texture memory

II. Process data

III. Store results back to global memory

All the recomendations about best practices are in [4] and [3].

Chapter 5

Shieh algorithm

Shieh *et al* [6] have proposed a DCT based watermarking embedding algorithm, where an image is transformed to the DCT domain after splitting to 8x8 blocks, and then a ratio ma- trix is calculated between the DC and AC coefficients. In the next step a Polarities matrix is computed. It represents the relation between image content and the embedding frequency bands, to embed the permuted watermark into the DCT domain using the Polarities matrix and, finally, IDCT is performed to get the watermarked image.

Shieh proposed the use of an evolutionary algorithm to optimize the position (frequency bands) where the watermark bits ought to be inserted within the original image. Through the different iterations, the algorithm tries to find out which are the best outcomes using PSNR and NC to evaluate the watermarked image.

I decided to use this watermarking algorithm because Professor Sabourin's team has an implementation of it in Matlab and it was easy for me to see how they implemented the func- tions involved in the algorithm, particularly the ones related with images, since I had not worked with images before.

They decided to use Shieh method because it is a blind method which means that it does not need the original cover image to extract the watermark. For the applications dealing with huge number of images, it would be very expensive to store all cover images for watermark extraction.

The steps of the algorithm are described below and shown in figure 5.4. Part of the implementation of the algorithm is described in the Appendix A. This appendix shows the configurations used for the GPU to implement the equations described in the next steps.

I. Initially, the image X of size $M \cdot N$ to be watermarked is splitted into $8 \cdot 8$ blocks to perform Discrete Cosine Transform on these blocks.

II. The individual $8 \cdot 8$ blocks are DCT transformed using the equation 2.1. The resultant matrix $Y_{(m,n)}(k)$ has the upper left corner as DC coefficient and the rest of the matrix are the AC coefficients, where the DCT coefficients are zigzag ordered as in figure 5.1.

III. The watermark image to be embedded W is assumed to be a binary image, of size $M_W \cdot N_W$. This binary image is permuted using a pre-determined key k_0 resulting W_p, see equation 5.1.

$$W_p = permute(W, k_0) \tag{5.1}$$

W_p is used for embedding the watermark bits into the selected DCT frecuency bands.

$Y_{(m,n)}(0)$	$Y_{(m,n)}(1)$	$Y_{(m,n)}(5)$	$Y_{(m,n)}(6)$	$Y_{(m,n)}(14)$	$Y_{(m,n)}(15)$	$Y_{(m,n)}(27)$	$Y_{(m,n)}(28)$
$Y_{(m,n)}(2)$	$Y_{(m,n)}(4)$	$Y_{(m,n)}(7)$	$Y_{(m,n)}(13)$	$Y_{(m,n)}(16)$	$Y_{(m,n)}(26)$	$Y_{(m,n)}(29)$	$Y_{(m,n)}(42)$
$Y_{(m,n)}(3)$	$Y_{(m,n)}(8)$	$Y_{(m,n)}(12)$	$Y_{(m,n)}(17)$	$Y_{(m,n)}(25)$	$Y_{(m,n)}(30)$	$Y_{(m,n)}(41)$	$Y_{(m,n)}(43)$
$Y_{(m,n)}(9)$	$Y_{(m,n)}(11)$	$Y_{(m,n)}(18)$	$Y_{(m,n)}(24)$	$Y_{(m,n)}(31)$	$Y_{(m,n)}(40)$	$Y_{(m,n)}(44)$	$Y_{(m,n)}(53)$
$Y_{(m,n)}(10)$	$Y_{(m,n)}(19)$	$Y_{(m,n)}(23)$	$Y_{(m,n)}(32)$	$Y_{(m,n)}(39)$	$Y_{(m,n)}(45)$	$Y_{(m,n)}(52)$	$Y_{(m,n)}(54)$
$Y_{(m,n)}(20)$	$Y_{(m,n)}(22)$	$Y_{(m,n)}(33)$	$Y_{(m,n)}(38)$	$Y_{(m,n)}(46)$	$Y_{(m,n)}(51)$	$Y_{(m,n)}(55)$	$Y_{(m,n)}(60)$
$Y_{(m,n)}(21)$	$Y_{(m,n)}(34)$	$Y_{(m,n)}(37)$	$Y_{(m,n)}(47)$	$Y_{(m,n)}(50)$	$Y_{(m,n)}(56)$	$Y_{(m,n)}(59)$	$Y_{(m,n)}(61)$
$Y_{(m,n)}(35)$	$Y_{(m,n)}(36)$	$Y_{(m,n)}(48)$	$Y_{(m,n)}(49)$	$Y_{(m,n)}(57)$	$Y_{(m,n)}(58)$	$Y_{(m,n)}(62)$	$Y_{(m,n)}(63)$

Figure 5.1: The matrix of the zigzag ordered DCT coefficients. Each $Y_{(m,n)}(k)$ is a frequency band where the watermark bits could be inserted.

IV. Initially frequency bands to embed the watermark are selected from 1^{th} iteration of the optimization problem using Evolutionary Computation (EC), e.g. choose $Y_{(m,n)}(6)$, $Y_{(m,n)}(9)$, $Y_{(m,n)}(12)$ and $Y_{(m,n)}(29)$. Along the iterations for optimization, these frequency bands are chosen for optimal embedding until the optimal frequency bands are reached using the EC algorithm. The transformed matrix $Y_{(m,n)}(k)$ is then used to get the ratio matrix between the DC and the AC coefficients $R(i)$ using the equation 5.2.

$$R(i) = \sum_{m=1}^{M/8} \sum_{n=1}^{N/8} \frac{Y_{m,n}(0)}{Y_{m,n}(i)}, \; i \in [1, 63] \tag{5.2}$$

In figure 5.2 just four blocks of the total grid of the whole image are shown. To get the value of $R(1)$ it is necessary to divide the element (0) and element (1) of each block, and then to add up all of them.

$$R(1) = \frac{B_{(0,0)}(0) + B_{(1,0)}(0) + B_{(0,1)}(0) + B_{(1,1)}(0) + \dots}{B_{(0,0)}(1)\; B_{(1,0)}(1)\; B_{(0,1)}(1)\; B_{(1,1)}(1)} \tag{5.3}$$

Same for $R(2)$:

$$R(2) = \frac{B_{(0,0)}(0) + B_{(1,0)}(0) + B_{(0,1)}(0) + B_{(1,1)}(0) + \dots}{B_{(0,0)}(2)\; B_{(1,0)}(2)\; B_{(0,1)}(2)\; B_{(1,1)}(2)} \tag{5.4}$$

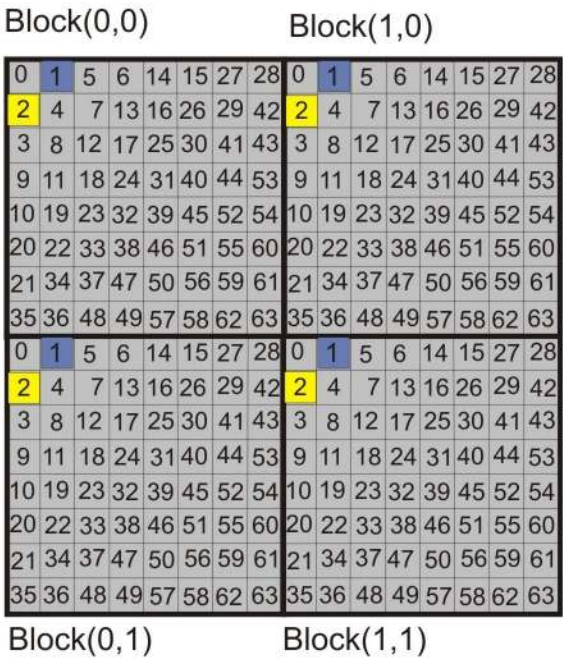

Figure 5.2: The image shows the zig-zag order of four 8x8 blocks of the original image. The R(1) value is the sum of the division of the element (0) between element (1) on each block of the whole image.

V. Then the polarities matrix P is calculated using the equation 5.5.

$$P_{(m,n)}(i) = \begin{array}{l} 1 \ if \ (Y_{(m,n)}(i) \bullet \mathrm{R}(i)) \geq Y_{(m,n)}(0), i \in F; \\ 0 \ otherwise. \end{array} \tag{5.5}$$

VI. Next, the watermarked DCT coefficient Y is obtained using the equation 5.6.

$$Y_{(m,n)}(i) = \begin{cases} Y_{(m,n)}(i) & \text{if } P_{(m,n)}(i) = W_{p(m,n)}(i) = 0,\ i \in F; \\ (Y_{(m,n)}(0)/R(i)) + 1 & \text{if } P_{(m,n)}(i) = 0,\ W_{p(m,n)}(i) = 1,\ i \in F; \\ Y_{(m,n)}(i) & \text{if } P_{(m,n)}(i) = W_{p(m,n)}(i) = 1,\ i \in F; \\ (Y_{(m,n)}(0)/R(i)) - 1 & \text{otherwise.} \end{cases} \tag{5.6}$$

The next figure shows an example of how to embed the watermark within the image. If the image size is $512 \cdot 512$ there are 4096 blocks $(512/8 * 512/8)$, and if the watermark size is $128 \cdot 128$ there are 16384 bits. Then, to embbed the watermark bits whitin the image, it is necessary to divide the number of watermark bits and the number of blocks of the image $16384/4096 = 4$. Number 4 represents the watermark bits that will be inserted in each block of the image.

Now, there will be chosen four frequency bands for each block where the watermark bits will be inserted applying equation 5.6 (where the polarities and ratio matrices are involved in the process), the frequency bands could be different from one block to another. Figure 5.3 shows an example.

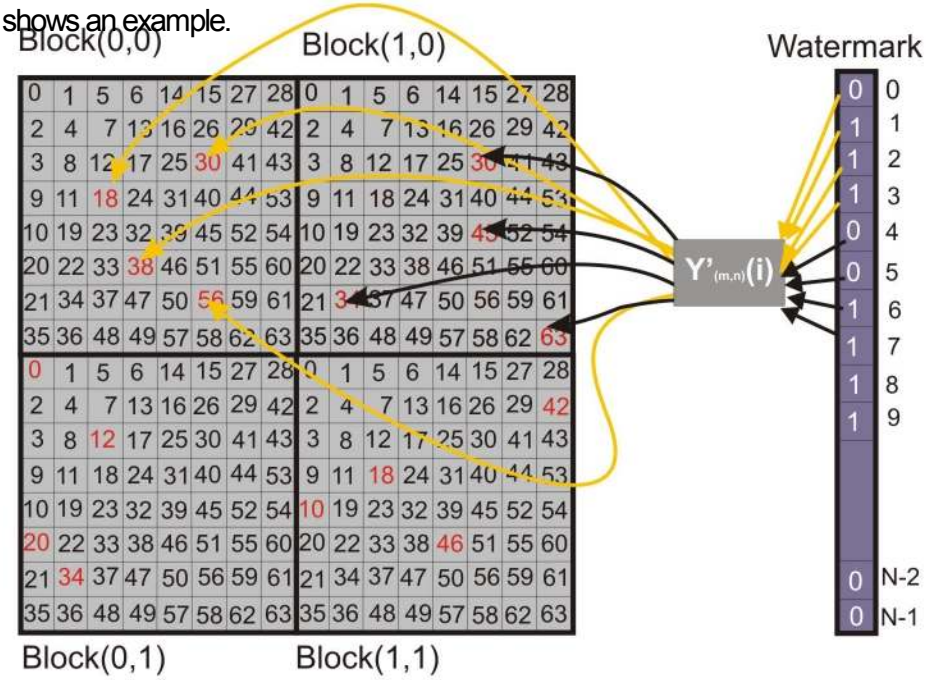

Figure 5.3: Embedding the watermark bits within the image. Each bit is inserted using the equation 5.6.

VII. After that, the watermarked image X_c is obtained by using the inverse DCT equation 2.2 for Y.

VIII. Now the PSNR is calculated as shown in equation 2.4 between the original image X and the watermarked image X_c using the MSE as seen in equation 2.5.

IX. Next, different attacks are applied to X_c and the attacked images are denoted by X_{cp}, where p is the number of attacking schemes. Then the NC is calculated between embedded watermark $W_{(ij)}$ and the extracted watermark from the attacked image $W_{(ij)}$ using equation 2.6

X. Finally the fitness function for the optimization problem is formalized using the aggregation of quality objective PSNR and the robustness objective NC, this can be formu- lated for the c^{th} iteration in the EC algorithm as 5.7.

$$f_c = PSNR_c + \sum_{b=1}^{p} (NC_{c,b} \bullet \lambda_{c,b}) \tag{5.7}$$

where $\lambda_{c,b}$ is the magnifying factor for the NC because the PSNR is dozen times larger. The process starts again in the step IV until obtaining the required optimization in the watermarked image.

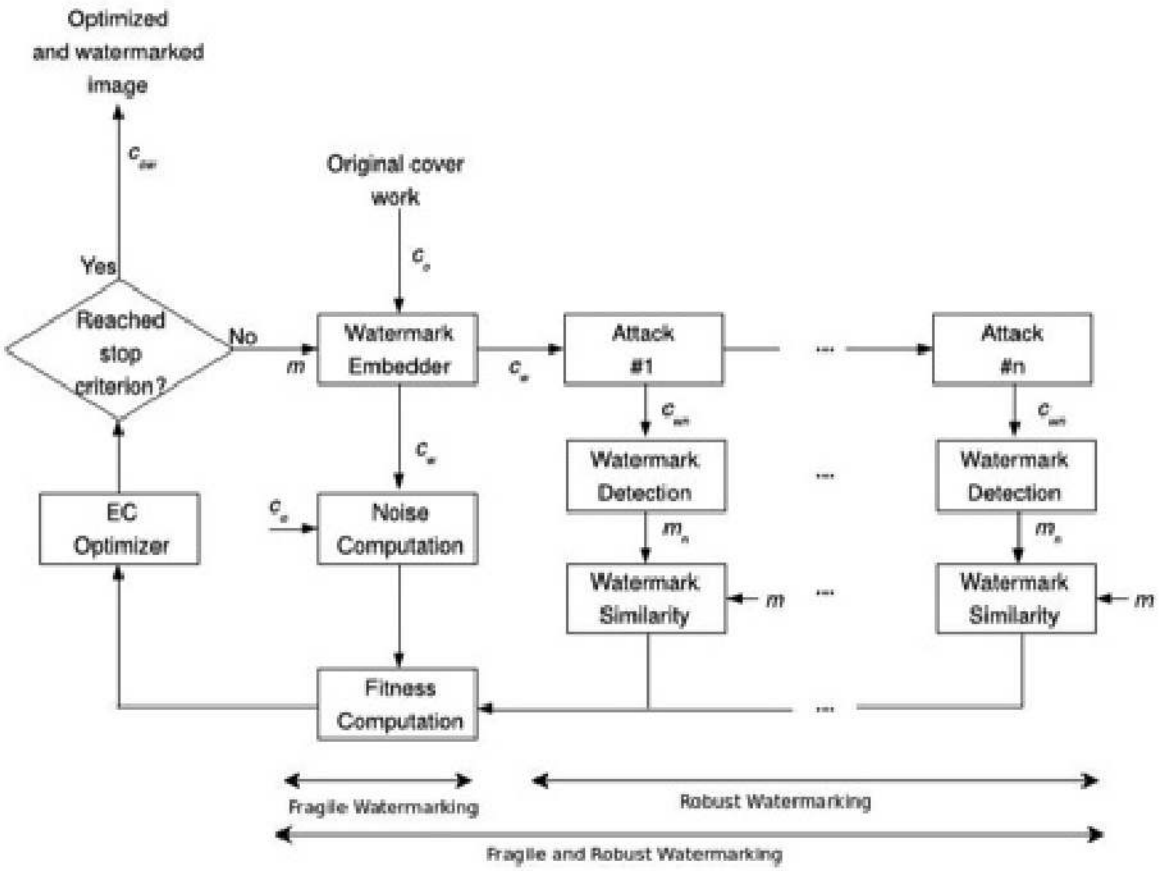

Figure 5.4: Generic Block Diagram for Watermarking.

5.1 The extraction algorithm

When extracting the watermarks, the original image X is not required in our algorithm. How- ever, the optimized watermarked image might be subjected to some intentional or uninten- tional attack, and the resulting image after the attack is represented by X. We calculate the DCT of the watermarked image after attacking Y, in the attacked X, with the secret key corresponding to the frequency set F, k_1. We then reproduce the estimated reference table R from the attacked X by following the operations in Eq. 5.8, and we are able to extract the permuted watermark,

$$W_{P,(m,n)}(i) = \begin{matrix} 1 \ if\ (Y_{(m,n)}(i) \bullet R(i)) \geq Y_{(m,n)}(0), \forall i, \\ 0 \ otherwise. \end{matrix} \tag{5.8}$$

$$W_p = \cup_{m=0}^{M/M_w-1} \cup_{n}^{N=1_w-1/N} \qquad \frac{\qquad}{Y_{m,n}(0)\ ,\ i \in F \\ Y_{m,n}(i)} \tag{5.9}$$

Finally, we use k_0 in Eq. 5.10 to acquire the extracted watermark W from W_p,

$$W_p = permute(W\ ,\ k_0) \tag{5.10}$$

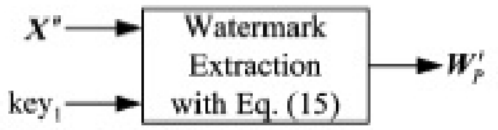

Figure 5.5: The block diagram for watermark extraction.

Chapter 6

Particle Swarm Optimization (PSO)

Nowadays, evolutionary computation makes use of models based on the natural evolution process, designing and implementing algorithms for solving problems.

There is a large variety of proposals and studies on these models, which are called with the generic name of Evolutionary Algorithms. These have as a common feature the inspi- ration on the simulation of the evolution of populations through processes of selection and reproduction.

Another set of proposals inspired by biological models, such as Ant Colony and Swarm-optimization algorithms are classified into what has been called bioinspired algorithms; a new way to solve problems based on the behavior of animals or systems that took centuries to evolve. These systems, Artificial Intelligence (AI) paradigms, are able to minimize the computation time of certain complex mathematical problems such as the traveling salesman problem.

Particle Swarm Optimization (PSO) is a population based stochastic optimization technique developed by Dr. Eberhart and Dr. Kennedy in 1995, inspired by the social behavior of bird flocking or fish schooling [10].

PSO shares many similarities with evolutionary computation techniques such as Ge- netic Algorithms (GA). The system is initialized with a population of random solutions and searches for the optimal using an iterative algorithm. However, unlike GA, PSO has no evolution operators such as crossover and mutation. In PSO, the potential solutions, called particles, fly through the problem space by following the current optimum particles. It has been successfully applied to many problems in several fields such as Biomedicine (S. Selvan 2006 [18] and Energy Conversion (J. Heo 2006 [8]), image analysis being one of the most frequent applications, like Biomedical images (Mark P. Wachowiak 2004 [19]), Microwave imaging (M. Donelli 2005 [12] and T. Huang 2007 [14]).

The proposal of using CUDA to implement these optimization algorithms is derived from the need of Banctec to have a tool to satisfy robustness and fidelity requirements for water- marking in huge quantities of gray scale images, which is why minimizing the time of the procedure was of great importance.

PSO is an algorithm based in populations, meaning that it has a lot of possible solutions that need to be evaluated, and finding the best one -depending on the problem- and the eval- uation itself consume a lot of processing time.

This is the main reason why CUDA comes as a viable option to accelerate the process due to the fact that operations involved in the algorithms could be parallelized (see appendix B), resulting on a minimization of the runtime of the operations.

The idea of using PSO as the optimization algorithm comes owing to the fact that it has few parameters to adjust. Since I was novice in programming with CUDA, it seemed like a suitable option to start working with.

6.1 Basic PSO

Each particle keeps track of its coordinates in the problem space which is associated with the best solution (fitness) achieved so far (this fitness value is stored). This value is called *pbest*. Another"best" value that is tracked by the particle swarm optimizer is the best value, obtained so far by any particle among the neighbors of the particle. This location is called *lbest*. When a particle takes all the population as its topological neighbors, the best value is a global best and is called *gbest*.

The PSO concept consists of, at each time step, changing the velocity (accelerating) of each particle toward its *lbest* and *gbest* locations. Acceleration is weighted by a random term with separate random numbers being generated for acceleration toward *lbest* and *gbest* locations.

After finding the two best values (*lbest* and *gbest*), the particle i updates its velocity and position with next equations 6.1 and 6.2, where $i = 1, 2, 3...N_S$.

$$V_i(t+1) = V_i(t) + \phi_1 r_1(B_i(t) - X_i(t)) + \phi_2 r_2(B_i(t) - X_i(t)) \quad (6.1)$$

$$X_i(t+1) = X_i(t) + V_i(t+1) \quad (6.2)$$

ϕ_1 and ϕ_2 are positive constants called acceleration coefficients, N_s is the total number of particles in the "swarm", r_1 and r_2 are random values, each component is generated between

$[0, 1]$, and g represents the index of the best particle in the neighborhood. The other vectors $X_i = [x_1, x_2, ..., x_{iD}] \equiv$ position of the i^{th} particle; $V_i = [v_1, v_2, ...v_{iD}] \equiv$ velocity of the i^{th} particle; $B_i \equiv$ best historical value of the i^{th} particle found, $B^g \equiv$ best value found of the i^{th} particle in the neighborhood [1].

Algorithm 1 Basic PSO

```
1: Initialize particles population
2: while do not get the max number of iterations or the optimal do
3:     Calculate the fitness for each particle i
4:     Update B_i if pbest is better than last one
5:     Calculate B^g of the neighbors i
6:     for each particle i do
7:         Calculate V_i (eq.6.1)
8:         Update X_i (eq. 6.2)
9:         Update best global solution (gbest)
10:    end for
11: end while
```

Another important feature that affects the search performance of the PSO is the strategy according to which B^g is updated. In *synchronous* PSO, positions and velocities of all i particles are updated one after another. The value of B^g is only updated at the end of each i generation, when the fitness values of all particles in the swarm are known.

The *asynchronous* PSO, instead, allows B^g to be updated immediately after the evaluation of each particle fitness. In *asynchronous* PSO, the iterative sequential structure of the update is lost, and the velocity and position update equations can be applied to any particle at any time, in no specific order [7].

6.2 Parallel PSO

The PSO was implemented in CUDA architecture to take advantage of the power offered by the massively parallel architectures available nowadays. The parallel programming model of CUDA allows programers to partition the main problem in many subproblems that can be solved independiently in parallel.

To exploit this feature of the CUDA architecture, in this thesis, the following implementation of the PSO algorithm was proposed. Figure 6.1 shows the UML diagram class of the PSO algorithm; it has been modeled with structs. Each particle has its position and velocity, besides the current fitness, best local fitness and best local position through the different iter- ations. The swarm has all the particles, the best global particle included.

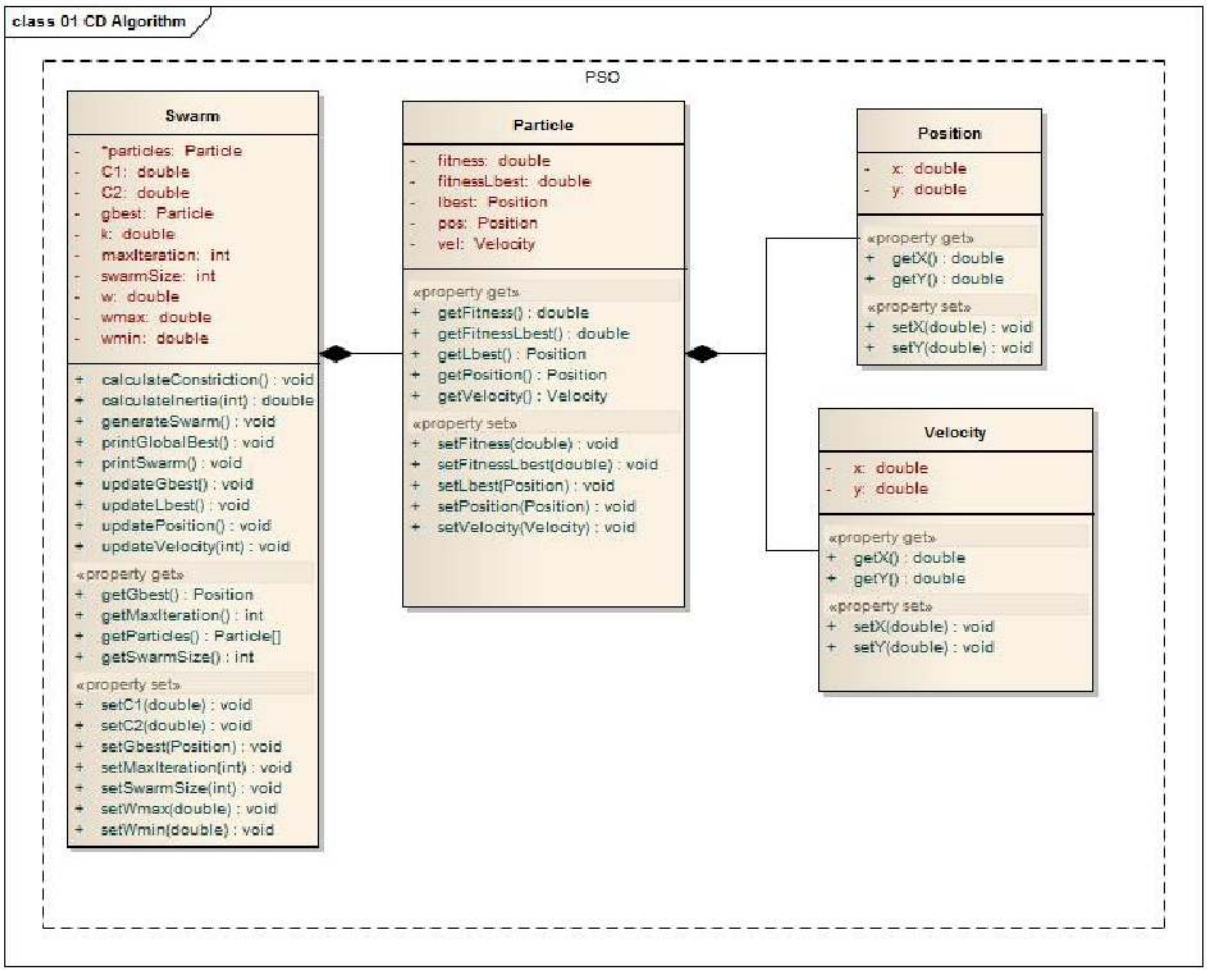

Figure 6.1: PSO UML Class diagram.

In the PSO algorithm, there will be as many swarms as the number of 8x8 blocks gener- ated after the DCT. If the image size is 512x512, then the number of blocks -as result of the DCT- will be 64x64 (4096 blocks). With the data separated into different blocks it is possible to compute them apart from each other, which means that they can be processed in parallel (the swarm 0 corresponds with the block 0 of the image). The implementation of the PSO in CUDA is described in the appendix B.

Each particle in the swarm has a possible solution where the watermark image could be inserted. The form to evaluate if the particle is a satisfactory solution is through the fitness value. In this work, Pareto dominance is used to evaluate the fitness function (see chapter 7).

Chapter 7

Multiobjetive optimization

When k objetive functions are simultaneously optimized in a problem, it is called multiobjective problem (MOP). In these problems maximization and/or minimization of k functions are required. In MOP, it is necessary to seek for the vector $x^* = [x_1^*, x_2^*, ..., x_n^*]^T$ to satisfy the inequality constraint set $g_i(x) \geq 0 \; \forall \; i = 1, 2, ..., n$ and the equality constraint set $h_i(x) = 0 \; \forall \; i = 1, 2, ..., m$ to optimize the functions vector $f(x) = [f_1(x), f_2(x), ..., f_k(x)]$ that represents the objetive function; where $x = [x_1, x_2, ..., x_n]^T$ is the decision variables vector. The solution ought to have acceptable values in the whole objetive set.

7.1 Pareto Theory

The notion of "optimum" was originally proposed by Francis Ysidro Edgeworth in 1881. This notion was later generalized by Vilfredo Pareto (in 1896). Although some authors call Edgeworth-Pareto optimum to this notion, we will use the most commonly accepted term: Pareto optimum.

7.1.1 Pareto dominance

A vector $u = (u_1, u_2..., u_k)$ dominates $v = (v_1, v_2, ..., v_k)$ if and only if u is partially less than v $(u \; v)$.

7.1.2 Pareto optimal

A solution $x^* \in \Omega$ is Pareto optimal if and only if there is no $x \in \Omega$ and $I = 1, 2, ..., k$ where $\forall i \in If(x) = f(x^*)$ and there is at least one $i \in If(x) > f(x^*)$.

The Pareto curve is the set of x^* where there are no other solutions for which simultaneous improvement in all objectives can occur. Generally a solution set known as non-dominated solutions is produced.

7.1.3 Pareto optimal set

For a MOP denoted by $f(x)$, the Pareto optimal set (P^*) is defined as:

$$P^* = x \in \Omega \mid \leftarrow \exists x \in \Omega f(x) \qquad f(x).$$

7.1.4 Pareto frontier

For a MOP denoted by $f(x)$ and a Pareto optimal set (P^*); the Pareto frontier (PF^*) is defined as:

$$PF^* = u = f = (f_1(x), f_2(x), ..., f_k(x)) \mid x \in P^*.$$

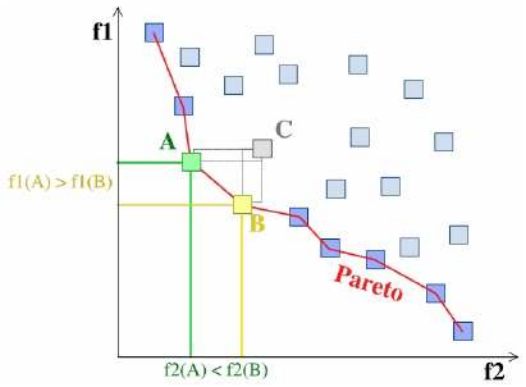

Figure 7.1: The boxed points represent feasible choices, and smaller values are preferred to larger ones. Point C is not on the Pareto Frontier because it is dominated by both point A and point B. Points A and B are not strictly dominated by any other, and hence do lie on the frontier.

7.1.5 Pareto Dominance used in this thesis

Choosing a good representation and constructing a good fitness function depend on the essence of the problem and it might be difficult. For this work, fidelity and robustness are considered as two objectives in conflict. By applying Pareto dominance it is relatively easy to evaluate the fitness function (consisting on the addition of fidelity and robustness) and moreover, add more objectives to the optimization process. In this process the objective is to minimize the disturbance of the original image after the insertion and the attacks.

In order to propose a simpler way to measure the fitness and the robustness spending the shortest time possible in the fitness calculation, the MSE was taken from the PSNR and the NC was changed. When measuring the MSE in each block just 64 comparisons are needed and they are executed at the "same time" in the other blocks. In the sequential process there

are needed 512x512 evaluations one after another for a 512x512 image size. The same case was applied for the NC, instead of being calculated for the whole image -as in the sequential form- it was just computed for each block.

The NC and the MSE are computed for each 8x8 block as showed in the figure 7.2. This was done with the purpose of dividing -as much as possible- the data in the GPU. In order to calculate the fidelity, it is necessary just to compare block by block how much the original image changes in contrast with the watermarked one. If the MSE value is zero, then it means that the block has not changed at all. As you can see, it is not necessary to calculate the PSNR if it is possible to obtain the same calculation -image fidelity- by just using MSE.

In the case of NC (for robustness), a variation of it was calculated. The bitwise operations are faster than a multiplication, which is why applying one of it reduces the runtime. In order to reduce the runtime in the evaluation of the NC, the logical operation "exclusive disjunc- tion", also called "exclusive or" (see formula 7.1) was used. The NC value must be close to zero between the original watermark (W) and the extracted watermark (W), to prevent the loss of the watermark image information.

$$NC = \frac{\sum_{i=1}^{M_W} \sum_{j=1}^{N_W} [W(i,j) \oplus W(i,j)]}{Bands\ per\ block} \tag{7.1}$$

The exclusive or calculation is shown in table 7.1.

W	W	Output
0	0	0
0	1	1
1	0	1
1	1	0

Table 7.1: Exclusive or.

The next image 7.2 shows -in big scale- how the blocks of the image -after the DCT- are organized. For each 8x8 block, the MSE and the NC are calculated. If the MSE and the NC values are close to zero, it is an indication that there is a good frequency bands set (see chapter 5) to insert the watermark image into the corresponding 8x8 blocks.

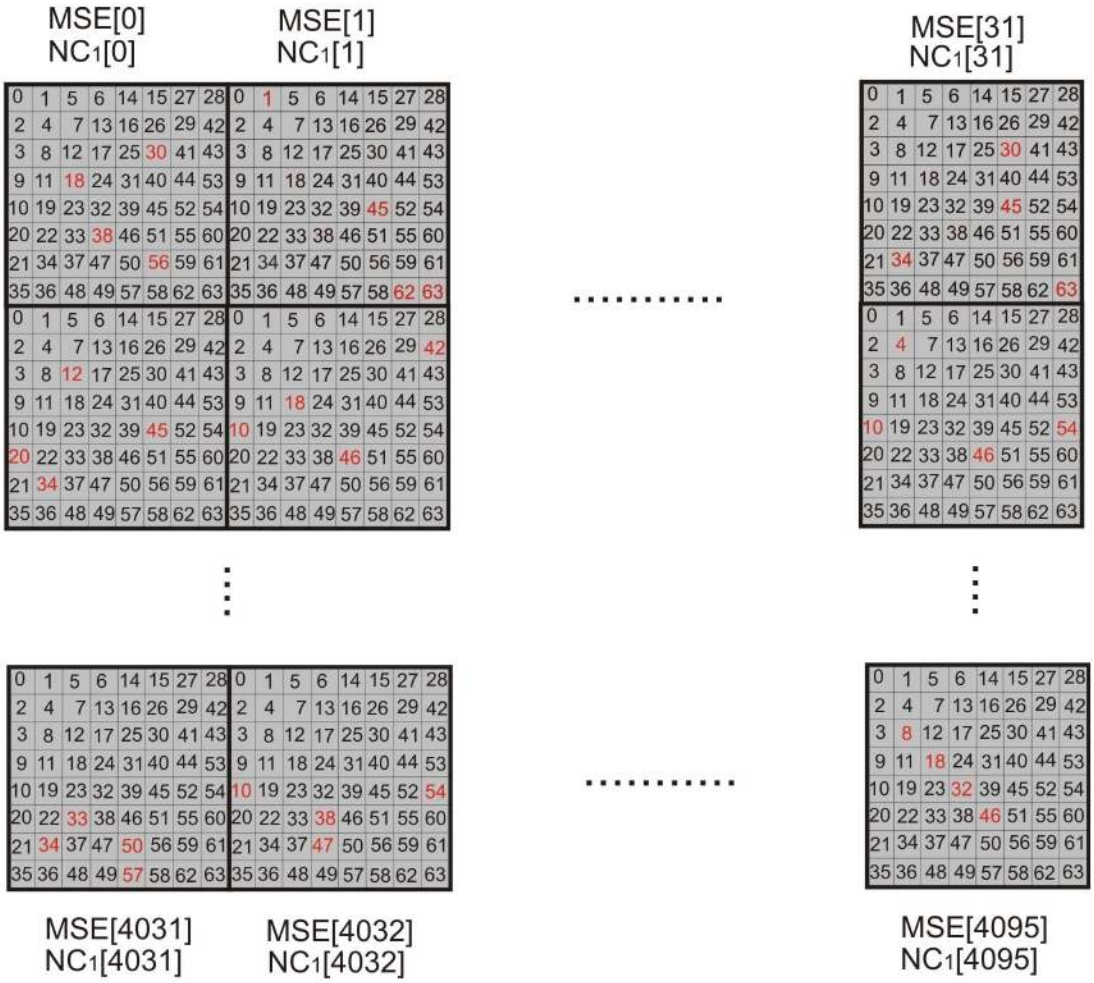

Figure 7.2: Image blocks organization.

The PSO algorithm spends a lot of time in the evaluation of the fitness function and in the calculation of the position and velocity vectors used for the particles to move, looking for other possible solution. Simplifying the functions -as much as possible- to calculate the fitness function helps to reduce the PSO's runtime.

Table 7.2 shows an example of the fitness (dominance) calculus (consider minimization in both objetives). The MSE and the NC must be close to cero; in the swarm, the particle with both values closest to zero is chosen to be the global best. In the example, there are six particles, particles 1, 3 and 4 are nondominated solutions, whereas 2, 5 and 6 are dominated by 3, 4 and 1 (see figure 7.3).

#	MSE	NC	Fitness
1	0.5	0	0
2	0.8	0.1	2
3	0.3	0.2	0
4	0.2	0.7	0
5	0.9	0.3	4
6	0.7	0.1	1

Table 7.2: Pareto dominance.

To calculate the fitness, all the particles are compared. Using particle 5 as example, the MSE of particle 5 always is higher for all the other particles, and the NC of particle 5 is always higher for all, except for particle 4, from the 5 comparisons made, in 4 of them parti- cle 5 is always higher in both values -MSE and NC-, that is why its fitness is 4. In the case of particle 6, it is just higher -in both values- to particle 1, that is why it has a fitness value of 1.

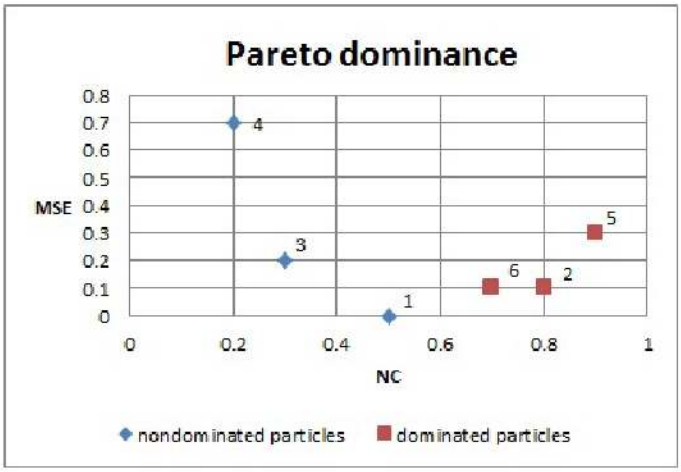

Figure 7.3: Pareto dominance chart.

Therefore, in the swarm there are three particles that could be used to insert the water- mark, but just one of them is taken as the best global particle in the swarm. To make this choice, the particle with the MSE closest to zero is chosen, if there is a tie -from the parti- cles with the same MSE-, the one with the NC closest to zero is chosen. If there are only dominated particles to choose, the particle with the MSE and the NC closest to zero is taken -under the same procedure already explained for nondominated particles-.

Chapter 8

The optimization algorithm

This chapter is dedicated to explain the complete procedure implemented to make the watermarking optimization algorithm combining the Shieh and the PSO algoritms. The objective of the optimization is to find the best frequency bands set to insert the watermark within the image. Different frequency bands are tested through the iterations of the algorithm findig out the best solution. At the end of the execution the application has as results the watermarked image and a matrix with the whole best positions (frequency bands) to insert the complete watermark.

The implementations in CUDA for the Shieh algorithm functions is detailed in the appendix A. The implementation of the PSO algorithm functions is detailed in appendix B. These appendices show the configuration of the threads for the functions involved in both algorithms.

This process is detailed as follows.

I. Using the DCT idea to split the image in 8x8 blocks, each block is used as a swarm.
An image of 512x512 has 4096 blocks; hence each block will be a swarm. At the same time, each swarm is mapped in the GPU as a block where the configuration of the threads depends of the function to be executed. The number of particles per swarm is specified as a configuration parameter of the algorithm. It is necessary to take into account that each particle in a swarm is a possible solution (frequency bands set).

II. Each particle has a position vector. The vector size depends on the number of watermark bits used to be inserted in each block of the image. If the watermark size is 128x128 and if it is divided uniformly in the 4096 blocks of the image, then 4 bits are inserted in each block. Each position corresponds to a frequency band in the 8x8 block, where the watermark bits are inserted.

At the beginning, all the swarms are initialized randomly (each swarm must have the same particles number). If 4 bits will be inserted, 4 bands are required, then 4 random numbers must be created between 1 and 63. This means that each particle will consist of 4 frequency bands (positions).

If each swarm has 5 particles, every particle has a set of 4 frequency bands used to originate 5 different solutions. To generate solution 1, all the particles with index 1 are taken and joined from every swarm; to generate solution 2, all the particles with index 2 are taken and joined from every swarm and so on. This procedure is shown in figure 8.1.

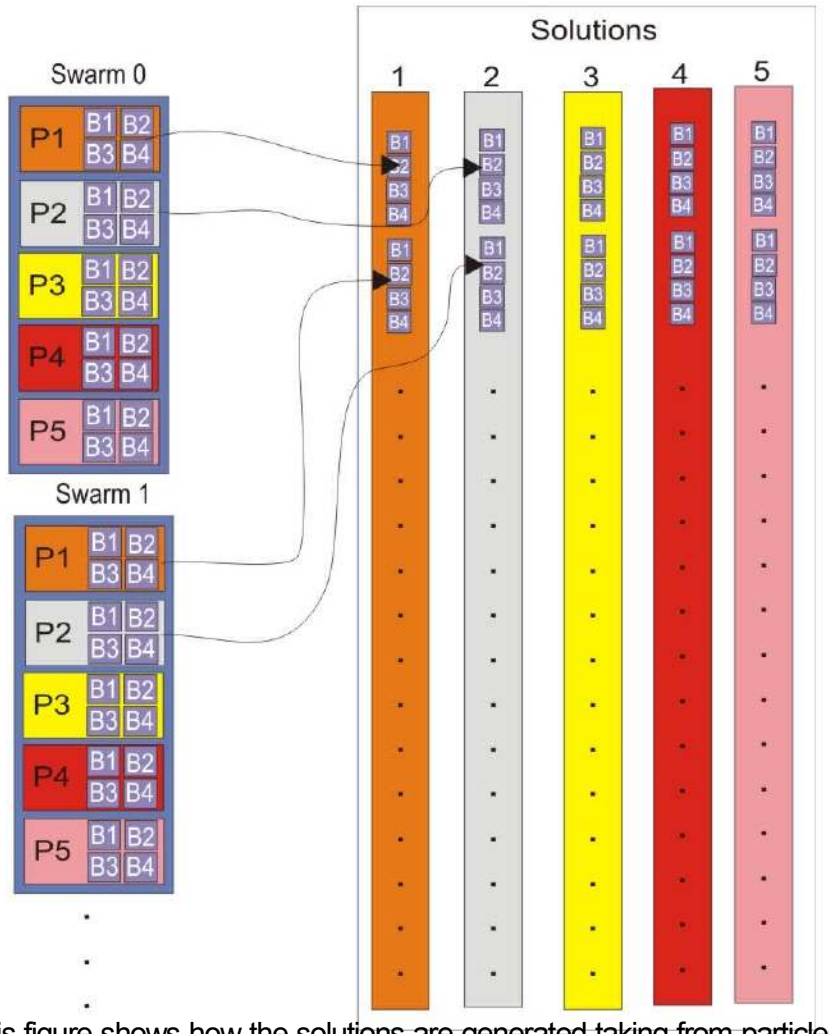

Figure 8.1: This figure shows how the solutions are generated taking from particles P1 and P2 - from the different swarms- the frequency bands B1, B2, B3 and B4, generating the corresponding solution.

III. After the insertion and the extraction operations (see chapter 5), the MSE (equation 2.5) and the NC (equation 7.1) are calculated. The addition of the MSE and the NC values is used as fitness function and its value is estimated -according with the theory in chapter 7- using Pareto dominance.

IV. One of the particles must be selected as the best global. Among the best options generated, one of them is chosen to be the best global. To choose the local best particle is considered to add up the MSE and the NC. If the new value is closest to zero than the old one, the new particle replaces the old one; otherwise the old one continues in the process (see chapter 7).

V. In the last step, the velocity and the new position of the particles are calculated, according to the formulas 6.1 and 6.2 . This generates the new bands and new the iteration begins. The next figure 8.2 shows the whole algorithm.

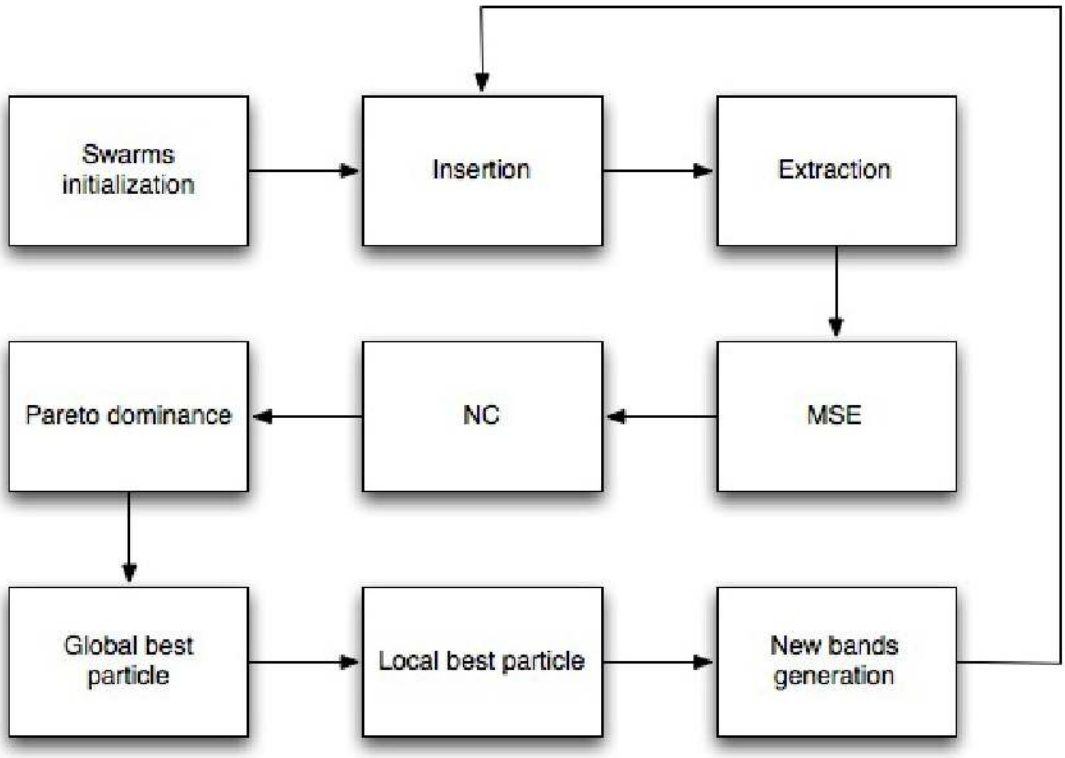

Figure 8.2: The optimization algorithm.

Chapter 9

Tests and Results

This chapter explains the servers features where the algorithm writen in C++ and CUDA C runs, the input necessary to execute the code, and the results of different tests.

9.1 Server Features

All tests were executed on two different servers with the following features. As you can see in the tables 9.1 and 9.2 the servers have the same GPU version, the same number of cores, but with different velocity.

Server name	Cores	CPU type
Uxdea	8	Intel Xeon E5620 @ 2.4GHz
Geogpus	8	Intel Xeon E5677 @ 3.47GHz

Table 9.1: CPUs Server features.

Server name	GPU	Cores
Uxdea	Tesla C1060	240
Geogpus	Tesla C1060	240

Table 9.2: GPUs Server features.

9.2 Input data

In order to test the implementations, figure 9.1 shows the original image (a) used in the algo- rithm to insert the watermark (b). The size of the original image is $512 \cdot 512$ in 24-bits BMP format.

(a) Original image

(b) Watermark

Figure 9.1: Input data

9.3 Outcomes

9.3.1 Shieh implementation

The figures 9.2 and 9.3 show the outcomes of executing sequential and CUDA implementa- tions in both servers Geogpus and Uxdea. The first tables show the results of executing five experiments, and taking the runtime for each function involved in the insertion and extraction algorithm.

These experiments were executed with the aim of comparing the runtimes between the implementation in C++ and the one in CUDA C based on the idea that the operations exe- cuted in the GPU must be faster than the ones computed in the CPU. The experiments shown in the tables were executed in both servers Geogpus and Uxdea.

The results obtained from the GPUs in both servers are faster than the ones collected from the CPU. At this point, the results seem to fit in the idea that the GPU is faster than the CPU. It should be noted that the functions are not considering the load and download of the data to and from the GPU.

GEOGPUS (sequential)

	DCT (ms)	RATIO (ms)	POLARITIES (ms)	IDCT (ms)	INSERTION (ms)	EXTRACTION (ms)
1	0.18646	0.00388	0.00032997	0.018899	0.001003	0.000041962
2	0.18629	0.003881	0.00033188	0.01913	0.0010269	0.000041008
3	0.18733	0.0038848	0.00033116	0.018871	0.0010052	0.000041008
4	0.18609	0.003866	0.00032997	0.018895	0.001003	0.0000422
5	0.1882	0.00385	0.00032997	0.018888	0.0010071	0.000040054
	0.18687400	**0.00387236**	**0.00033059**	**0.01893660**	**0.00100904**	**0.00004125**

GEOGPUS (CUDA)

	DCT (ms)	RATIO (ms)	POLARITIES (ms)	IDCT (ms)	INSERTION (ms)	EXTRACTION (ms)
1	0.0002141	0.0019349	0.000000051212	0.000051975	0.00014806	0.000082493
2	0.00021601	0.001907	0.000000050902	0.000051975	0.00010586	0.000085115
3	0.00021505	0.0016931	0.000000050592	0.000051022	0.0001049	0.000081062
4	0.00021482	0.001937	0.000000049901	0.000051022	0.00014687	0.000082803
5	0.00021505	0.0016671	0.000000049806	0.000051975	0.00010395	0.000082016
	0.00021501	**0.00182782**	**0.00000005**	**0.00005159**	**0.00012193**	**0.00008270**

Figure 9.2: Runtime for functions involved in the insertion/extraction algorithm running on the Geogpus server.

UXDEA (sequential)

	DCT (ms)	RATIO (ms)	POLARITIES (ms)	IDCT (ms)	INSERTION (ms)	EXTRACTION (ms)
1	0.28871	0.0060811	0.00057411	0.029216	0.0014939	0.000061989
2	0.28841	0.0059891	0.00054598	0.029207	0.0015059	0.000061989
3	0.28844	0.0059879	0.00056505	0.029239	0.0015011	0.000062227
4	0.28843	0.006079	0.00056291	0.029185	0.001503	0.000062943
5	0.28854	0.005985	0.00056815	0.029178	0.001508	0.000057936
	0.28850600	**0.00602442**	**0.00056324**	**0.02920500**	**0.00150238**	**0.00006142**

UXDEA (CUDA)

	DCT (ms)	RATIO (ms)	POLARITIES (ms)	IDCT (ms)	INSERTION (ms)	EXTRACTION (ms)
1	0.000211	0.0026978	0.000008579	0.000035012	0.00010896	0.000081062
2	0.00020599	0.002233	0.0000088007	0.000036001	0.00010085	0.000080824
3	0.00020599	0.002193	0.0000074502	0.00003602	0.00010204	0.000080109
4	0.00020695	0.002126	0.0000083215	0.000036955	0.00010109	0.000082016
5	0.00020695	0.0020871	0.0000073191	0.000036001	0.00010109	0.000077009
	0.00020738	**0.00226738**	**0.00000809**	**0.00003600**	**0.00010281**	**0.00008020**

Figure 9.3: Runtime for functions involved in the insertion/extraction algorithm running on the Uxdea server.

The figures 9.4 and 9.5 show the runtime of the complete procedure to insert and extract a watermark involved in Shieh algorithm. In these experiments —where the upload and download of the date are considered— the GPU does not seem such superior considering the results of the last figures. The MSE and the NC functions (see MSE and NC in the figures) executed on the GPU without considering the data transfer seems to be fast, but considering the data transfer are more expensive than the ones executed in the CPU (see MSE Total and NC Total in the figures).

GEOGPUS (sequential)

	INSERTION OP.	MSE	EXTRACTION OP.	NC
1	0.23152	0.002527	0.22152	0.0061359
2	0.22963	0.0025148	0.22486	0.0055099
3	0.23099	0.0025282	0.22162	0.0056429
4	0.22983	0.0025229	0.22303	0.0060408
5	0.22824	0.002522	0.22373	0.0061831
	0.23004200	**0.00252298**	**0.22295200**	**0.00590252**

GEOGPUS (CUDA)

	INSERTION OP.	MSE Total	MSE	EXTRACTION OP.	NC Total	NC
1	0.019511	0.012955	0.0000088215	0.038799	0.00579	0.0000081062
2	0.019452	0.012958	0.0000097752	0.037034	0.0064609	0.0000078678
3	0.019466	0.014669	0.0000088215	0.038534	0.0058	0.0000078678
4	0.019507	0.012967	0.000010014	0.037186	0.005774	0.0000081062
5	0.019937	0.013699	0.000010967	0.037502	0.0058062	0.0000081062
	0.01957460	**0.01344960**	**0.00000968**	**0.03781100**	**0.00592622**	**0.00000801**

Figure 9.4: Runtime of the insertion and the MSE, and the extraction and the NC operations on Geogpus.

UXDEA (sequential)

	INSERTION OP.	MSE	EXTRACTION OP.	NC
1	0.35498	0.0039091	0.34248	0.00634
2	0.35387	0.0039029	0.3457	0.006284
3	0.35552	0.003906	0.34104	0.007237
4	0.35404	0.0039041	0.3415	0.0062912
5	0.3539	0.0039091	0.34266	0.006207
	0.35446200	**0.00390624**	**0.34267600**	**0.00647184**

UXDEA (CUDA)

	INSERTION OP.	MSE Total	MSE	EXTRACTION OP.	NC Total	NC
1	0.035745	0.028732	0.0000078678	0.063088	0.0069678	0.0000081062
2	0.035785	0.030604	0.0000081062	0.063564	0.0069449	0.0000078678
3	0.035772	0.030602	0.0000081062	0.063319	0.0070071	0.0000081062
4	0.035756	0.030622	0.0000081062	0.06242	0.007031	0.0000081062
5	0.035592	0.030598	0.0000090599	0.062749	0.006952	0.0000078678
	0.03573000	**0.03023160**	**0.00000825**	**0.06302800**	**0.00698056**	**0.00000801**

Figure 9.5: Runtime of the insertion and the MSE, and the extraction and the NC operations on Uxdea.

In accordance with the features of the server, the GPU of Geogpus is faster than the one of Uxdea. Seeing the results in the figures 9.2 and 9.3, practically there is no difference in the runtimes, but seeing the results in figures 9.4 and 9.5 it could be established that the GPU of Geogpus has a better transfer velocity that helps it to be almost two times faster than Uxdea.

9.3.2 PSO implementation

The figures 9.6 and 9.7 show tables with the runtimes of the implementation of the optimization algorithm —PSO—. These figures present five experiments with a different number of iterations for the execution in the CPU and in the GPU. The outcomes are compared in experiments with the same iteration number. These experiments were made to compare the amount of time used for the algorithm and the quality of the results.

As in the experiments made for the Shieh algorithm, the operations on the GPU must be faster. The first point to evaluate in the PSO algorithm is the random number generation. Using the random numbers in the sequential version it is remarkable the difference in time. The use of those numbers consumes a big quantity of time due to its necessity to spend time in the CPU to generate different numbers. For the sequential version, the random numbers are generated using the C function "drand48" that returns a pseudo-random number in the range [0.0,1.0). On the GPU, the random numbers are generated using a library called curand [5].

GEOGPUS

10 Iterations

	Sequential (min)	Initial fitness	Final fitness	CUDA (s)	Initial fitness	Final fitness
1	53.7133	3.0079	0.24625	6.8422	5.3136	4.9884
2	53.6950	5.7397	0.27473	6.6205	0.66101	0.33551
3	53.6600	2.9303	0.25068	6.6103	0.82102	0.50254
4	53.6517	3.5922	0.27855	6.6197	15.315	14.716
5	53.6433	4.2929	0.2387	6.6786	1.9163	1.5937
	53.6727			**6.6743**		

30 Iterations

	Sequential (min)	Initial fitness	Final fitness	CUDA (s)	Initial fitness	Final fitness
1	161.35	2.4226	0.21096	18.126	1.7282	1.324
2	160.6917	3.4672	0.2026	18.516	0.78431	0.39118
3	160.635	5.8846	0.23375	18.081	0.81376	0.40794
4	161.075	3.1447	0.19965	18.397	1.6347	1.2302
5	161.35	3.3854	0.19588	18.113	1.4422	1.0504
	161.0203			**18.2466**		

Sequential no random numbers

	10 Iterations (s)	Initial fitness	Final fitness	30 Iterations (min)	Initial fitness	Final fitness
1	26.228	4.7199	0.22448	72.8140	26.167	0.223
2	26.108	5.1654	0.29945	72.6400	6.8238	0.23352
3	26.208	2.4494	0.2621	72.8590	7.7055	0.21568
4	26.279	10.571	0.26432	72.7310	1.9816	0.20543
5	26.167	6.073	0.20027	73.0250	2.9215	2.9215
	26.1980			**72.8138**		

CUDA no random numbers

	10 Iterations (s)	Initial fitness	Final fitness	30 Iterations (s)	Initial fitness	Final fitness
1	6.5794	2.8593	2.5458	17.7430	1.1509	0.75917
2	6.5145	1.2957	0.97485	17.9710	7.8518	7.2715
3	6.6214	0.82386	0.49934	17.8740	0.85486	0.47516
4	6.6271	5.2253	4.8637	17.6850	5.6334	5.2352
5	9.8096	2.321	1.996	17.0650	0.49091	0.1239
	7.2304			**17.6676**		

Figure 9.6: Runtime for PSO on Geogpus.

UXDEA

10 Iterations

	Sequential (min)	Initial fitness	Final fitness	CUDA (s)	Initial fitness	Final fitness
1	75.0933	4.7145	0.28886	17.696	2.429	2.1085
2	75.1300	11.316	0.3058	17.754	0.9868	0.64933
3	75.1467	5.4008	0.254	17.744	0.85646	0.53621
4	75.1433	3.1818	0.2784	17.626	2.1421	1.8232
5	75.0767	3.1201	0.2568	17.696	3.9049	3.6015
	75.1180			**17.7032**		

30 Iterations

	Sequential (min)	Initial fitness	Final fitness	CUDA (s)	Initial fitness	Final fitness
1	225.3833	3.8861	0.22232	48.535	0.70031	0.30285
2	225.1000	4.0846	0.20906	48.635	1.4996	1.1144
3	225.2667	2.5326	0.20198	48.794	2.0107	1.6072
4	224.8833	3.3661	0.2154	48.597	6.3867	5.4775
5	225.3833	4.4556	0.21926	48.757	0.67997	0.28269
	225.2033			**48.6636**		

Sequential no random numbers

	10 Iterations (s)	Initial fitness	Final fitness	30 Iterations (min)	Initial fitness	Final fitness
1	36.363	6.3162	0.24992	100.5400	3.3363	0.21798
2	36.664	3.0594	0.2729	100.6400	2.3531	0.22367
3	36.289	3.2031	0.26768	100.5000	8.4095	0.20501
4	36.302	2.9081	0.25941	100.5900	3.1055	0.20916
5	36.404	3.902	0.32894	100.5100	5.426	0.22008
	36.4044			**100.5560**		

CUDA no random numbers

	10 Iterations (s)	Initial fitness	Final fitness	30 Iterations (s)	Initial fitness	Final fitness
1	17.347	1.041	0.72842	47.8740	0.66338	0.26873
2	17.395	1.2237	0.90721	47.9710	7.8518	7.2715
3	17.59	5.0348	4.1984	47.9920	2.8143	2.4339
4	17.462	5.149	4.8263	47.8480	4.5469	4.1704
5	17.577	1.1466	0.8309	48.1320	0.8529	0.46197
	17.4742			**47.9634**		

Figure 9.7: Runtime for PSO on Uxdea.

Reviewing the values (figures 9.6 and 9.7) of the initial fitness and the final fitness it is noteworthy that the sequential version gives better results than the ones obtained from the GPU. For all the cases, the runtimes indicate that GPU is faster than CPU, even when all data have been loaded or when using static numbers in the CPU version. With this, it is possible to set up that —at least for this version of the application—, if the user wants a good opti- mization for the watermarking, the sequential version must be used. By contrast, if the user needs a quick approximation, the GPU version ought to be applied.

Chapter 10

Conclusions and future work

10.1 Conclusions

With the vast volume of information flowing on the Internet, watermarking is widely used to protect this information authenticity. The need for copyright a huge quantity of digital files, spending the less possible amount of time and avoiding the loss information were the reasons to propose the use of an algorithm for watermarking —Shieh algorithm—, Particle Swarm Optimization as an optimizer, and finally a GPU -based in CUDA architecture- to accelerate the process.

The use of a GPU for accelerating the operations involved in the algorithms of insertion and extraction of the watermark and in the optimization algorithm was a challenge, since it is a parallelism paradigm. There is not a standard configuration for the blocks, threads or the memory treatment in the GPU. That is why the analysis and design of the procedures are a requirement to take advantage of the parallelism. In order to use parallel programing in a GPU, it is necessary to shift from a sequential to a parallel thinking, strictly to learn how to divide a huge problem into small ones —divide and conquer—, attempting to have the best performance.

Using an image of size 512x512 as an input, it is possible to divide it in 64x64 blocks —such as in the DCT—. The 64x64 matrix is easily mapped to the same number of blocks in the GPU, and the configuration of the threads will depend on the type of operation to be executed. For example, in the calculation of the NC there were required just 4 threads to do the comparisons, but in the case of the MSE 64 threads working at the "same time" were required (see appendix B). Therefore, the configuration of the blocks and threads for an ap- plication on a GPU must be carefully analyzed.

Other point of consideration in the use of the GPUs is the memory treatment. In this application the global memory was used to put up the image and the watermark data, the ratio and polarities matrices, without forgetting the random numbers. This memory is used to carry the data from the host (RAM memory) to the device (GPU memory) and vice versa.

The problem of using it is the long time it spends in the transfer —that depends of the amount of data—. As you can see in the experiments, the runtime of the functions are quickest in the GPU without considering the data transfer. Considering the data transfer, sometimes the function spends more time than the sequential execution (see chapter 9). Other type of mem- ory used in this application was the shared memory. This memory is used just inside the blocks and it is not visible between others —unlike the global memory that is visible for all the blocks—. The shared memory is faster than the global memory, the problem with it is the handling and the overall synchronization with the threads (the MSE operation uses shared memory to execute a reduction operation, this is shown in the appendix B).

The design of the PSO algorithm was made applying object oriented analysis (see chap- ter 6) and it was implemented using C++ in order to have two implementations —C++ and CUDA C— to compare outcomes. At the moment of trying to map the classes from C++ to CUDA C there was a big problem: in the classes I used dynamic memory to store the results from the operations. At the moment when I tried to map it to the GPU memory it was not possible to keep the references, so it was necessary to make some changes for the CUDA C version. The use of structs instead of classes was the first change due to fact that the classes used in C++ are not equivalent in CUDA C. The second change consisted on the use of static memory instead of dynamic memory.

I analyzed the different options to implement the PSO, but I decided to use as much swarms as number of blocks used to divide the image in the DCT (see chapter 6). This was in profit of dividing a big problem in small ones, which suited with this parallel paradigm. As it was established, there is not a standard configuration in CUDA architecture, so I made the configuration in accordance with the need of the function. The PSO needs to evaluate two vectors: velocity and position. Position depends of the velocity that is why velocity needs to be computed first. If there are 4096 swarms —4096 blocks— and each swarm has five particles, then each of them need to update the velocity vector. The number of operation to be calculated in a CPU is: 4096 (swarms) * 5 (particles) * 1 (operation) = 20480 operations one after another. In the case of the same operation on the GPU, there are executed the same 20480 operations, but the difference is that there are 4096 swarms with 5 threads working in parallel computing one operation, hence there are 20480 threads working at the same time. If one thread in the CPU spends 1 second by operation the runtime will be 20480 s, but in the case of the GPU there are 20480 threads working at the same time, and they spend 1 second to finish the calculus. In the last example I am not considering the speed of the processor —neither CPU nor GPU— nor the upload/download of the data to/from the GPU.

The velocity vector needs random numbers to be calculated (see equation 6.1). In order to generate random numbers I used a library called curand (see [5]). This library is useful because it is easy to generate a lot of numbers in a short time; the problem comes with the memory. If there is a big quantity of this numbers generated and held in global memory, there might be a shortage of space to store other data. For one iteration of the PSO there are used two random numbers to calculate the velocity value. If there are 4096 blocks with 5 particles

each, 40960 random numbers for iteration are needed. There is another type of memory on the GPU, the constant memory. This memory is loaded in the GPU but it cannot be changed. This memory was considered to store the random numbers because they do not modify its value on the execution of the calculation of the velocity value.

There are a lot of GPUs on the market to be used, some of them for servers, others for PCs or laptops. I decided to use the ones from Nvidia since I already have a laptop with one of its cards. I started to program on it, but there was a problem, when I tried to execute the same code in a server with a better GPU, I realized that the float and the double numbers representation changed. This is not represents a big obstacle because in small GPUs the dou- ble number is changed to float automatically. Another feature that needs to be considered is —from GPU to GPU— the velocity of the processor. This is evident in the experiments because the Geogpus server is faster than the Uxdea server (see chapter 9).

In the case of the Shieh algorithm, the equations required to be parallelized were analyzed to get the best performance on the GPU. For the calculation of the MSE and NC there was not an improvement of the performance compared with the sequential version. The execu- tion of the functions is fast, but the transfer of the data to the GPU and back slows down the performance. For this reason it is necessary to seek for another solution for the transaction of the data.

To program an application oriented to be executed on a GPU it is necessary to have knowl- edge of how the CUDA architecture works. At the beginning it is not easy to start thinking in parallel and change a big problem in small ones. The important thing to make a good design of an application for a GPU is to consider the management of the different sorts of memories and their capacity to store data, as well as to bear in mind that the velocity of the processor changes with the versions of the GPU, such as the number precision representation, thus take out some portability.

To program on a GPU there is another language called OpenCL (Open Computing Lan- guage). It is made for running in any GPU independent manufacturer. At the moment of starting this work there was more information about CUDA than OpenCL, besides the option of program in the GPU of my own laptop. These were the reasons to start working with CUDA.

After this analysis of the present work, I can say that the use of CUDA helps to improve the performance of the application and that an algorithm based in population could be im- plemented on it, as long as the developer is aware of the features of this technology. This application is the cornerstone and it provides the opportunity to keep working on it to make it more robust.

10.2 Future work

In this thesis, quantization was used as an attack to the watermarked image, but it was applied before the IDC Transform in the insertion routine, and it could be implemented as an extern routine to apply after the insertion.

Other attacks can be implemented in order to be applied to the watermarked image; each new attack should be added as a new objective in the PSO evaluation. If there are more attacks, the application could have a switch used to adjust the attacks to be optimized in ac- cordance with the user requirements.

There is a library called Thrust that provides a flexible high-level interface for GPU programming and offers the possibility of doing operations without the requirement of configure the blocks and threads on the GPU —removing weight off the programmer's shoulders—. This library could help to improve the performance of the application.

A different implementation to calculate random numbers in the CPU could be done in order to improve the time needed to be generated.

Additionally, the PSO could be adjusted in order to look for best outcomes.

Appendices

Appendix A

Analysis, Design and Implementation of Shieh Algorithm

This appendix shows part of how the Shieh algorithm was implemented using CUDA C.

Figure A.1 shows the flow diagram of Shieh algorithm, and the operations already implemented in C++ and CUDA.

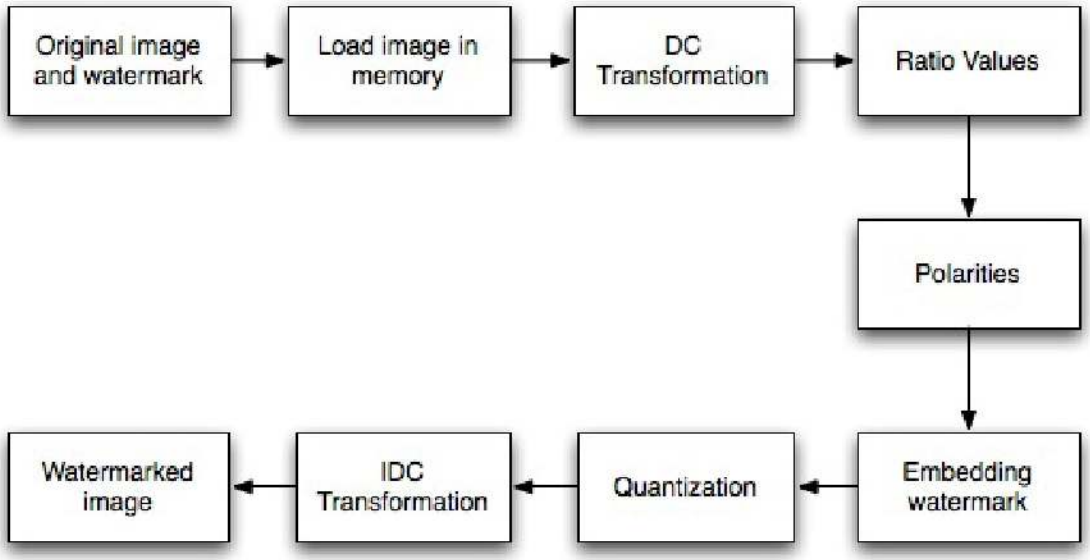

Figure A.1: Flow Diagram of Shieh Algorithm.

Figure A.2 shows the flow diagram for watermarking extraction, and the operations already implemented in C++ and CUDA.

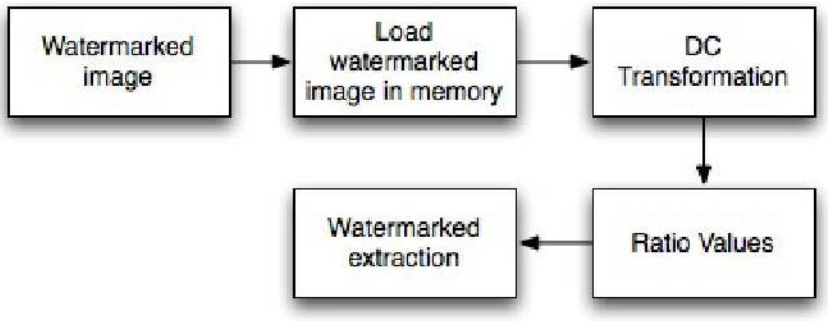

Figure A.2: Flow Diagram for Watermarking Extraction.

A.1 Shieh Operations

As an example -for all the set of operations on the Shieh algorithm- a 128x128 binary watermark is considered to be inserted into a 512x512 gray scale image. In agreement with the steps of the algorithm described in Chapter 6, it is necessary to load the image into the GPU memory and to apply the DCT. In order to take advantage of the parallelism, a library with this function provided by CUDA was used. After applying the DCT to the 512x512 image, a matrix of 64x64 blocks -that represents the image- is obtained. Each block is divided at the same time into 8x8 frequency bands where the watermark will be inserted. The configuration of 32x32 blocks is maintained in the GPU for all the operations, each block in the GPU rep- resents one block of the image after the DCT; what differs in the GPU is the configuration of the threads that depends on the need of the operation to be executed.

A.1.1 Ratio Operation

Once the image in DCT is already loaded in GPU memory, the next step is to get the ratio between the DC and the AC coefficients $R(i)$ using the equation 5.2. This operation was divided in two parts. First division between $Y_{mn}^{(0)}$ is performed and runs on the GPU. The $Y_{m,n}(i)$ second part is the sum, that runs on the CPU.

The block configuration on the GPU is:

```
dim3 ThreadsRatioBlocks(BLOCK_SIZE, BLOCK_SIZE);
```

```
dim3 GridRatioBlocks(Size.width/BLOCK_SIZE,
                                        Size.width/BLOCK_SIZE);
```

With this block configuration, a 64x64 grid of blocks is generated and each block has 8x8 threads. Each thread makes just one operation between the DC and the AC values. The AC corresponds with the thread position on the block (current coefficient). The results are stored in the vector *raux*, this vector is used to do the sum.

```
__global__ void CUDAKernelRatio(float *src, float *raux,
                                        int stride, int blockSize){

    // Block index
    int bx = blockIdx.x; int by
    = blockIdx.y;

    // Thread index (current coefficient)
    int tx = threadIdx.x; int ty =
    threadIdx.y;

    //copy current coefficient to the local variable
    float dividend = src[ (by * blockSize + 0) * stride +
                          (bx * blockSize + 0) ];          //DC value
    float divisor = src[ (by * blockSize + ty) * stride +
                          (bx * blockSize + tx) ];         //AC value

    //operation
    if( divisor != 0 ){
raux[ (by * blockSize + ty) *          stride +
            (bx * blockSize +          tx) ] = dividend / divisor;
    }else{
        raux[ (by * blockSize          + ty) * stride +
            (bx * blockSize +          tx) ] = 0;          //Default value
    }

    __syncthreads();
}
```

A.1.2 Polarities Operation

To make this operation based in equation 5.5, it is necessary to load from the host memory to the GPU global memory the ratio and the bands vector. Bands vector keeps the places where the watermark will be embedded in each block. Next example shows how to load the bands vector from host memory to the GPU global memory. The size of the bands vector must be equal to the size of the watermark image.

```
int *dev_bands;
HANDLE_ERROR( cudaMalloc( (void**)&dev_bands,
                    bandSize * sizeof(int) ) ); //allocate memory
                                                          //on GPU

HANDLE_ERROR( cudaMemcpy(dev_bands, bands,
                    bandSize * sizeof(int),
                    cudaMemcpyHostToDevice ) ); //copy memory from
                                                          //host to GPU
```

The block configuration on the GPU for this operation depends on the number of bands by block. The block number in the grid is 64x64, and the total thread number is equal to the bands by block. This is due to it is just necessary to compute the frequency bands where the watermark will be inserted.

```
dim3 ThreadsPolaritiesBlocks(bandsByBlock);
dim3 GridPolaritiesBlocks(Size.width/BLOCK_SIZE,
                                        Size.height/BLOCK_SIZE);
```

The results are stored in the vector p, this vector has the same size as the number of bands.

```
__global__      void CUDAKernelPolarities( float *image, float *p,
                float *r, int *bands, int height,
                int width, int stride, int blockSize,
                int bandsByBlock ){

    // Block index
    int bx = blockIdx.x;
   int by = blockIdx.y;

    // Thread index (current coefficient)
```

```
        int tx = threadIdx.x;

int ib = bands[ ( by * ((width/8)*bandsByBlock) ) +
                                    (bx * bandsByBlock + tx) ];

float c = image[ (by * blockSize + 0) * stride +
                                     (bx * blockSize + 0) ];
float a = image[ (by * blockSize + iY[ib]) * stride +
                                     (bx * blockSize + iX[ib]) ];
float b = r[ib];

if ( a * b >= c){
p[ ( by * ((width/8)*bandsByBlock) ) +
                            (bx * bandsByBlock + tx) ] = 1;
}else{
p[ ( by * ((width/8)*bandsByBlock) ) +
                            (bx * bandsByBlock + tx) ] = 0;
}

__syncthreads();

}
```

A.1.3 Watermark Embedding Operation

This operation is bassed on equation 5.6, and it requires to load the watermark from the host memory to the GPU global memory. The watermark size is the same as the number of bands.

```
dim3 ThreadsExtractBlocks(bandsByBlock);
dim3 GridExtractBlocks(Size.width/BLOCK_SIZE,
                                        Size.height/BLOCK_SIZE);
```

As it was seen in the code above, the grid configuration is the same as the one used in po- larities operation. The next code shows the watermark embedding operation into the image. The watermarked image quality is evaluated with the MSE as seen in equation 2.5.

```
__global__ void CUDAKernelWatermarkInsertion( int *bands,
                    float *image, float *newImage, int *water, float *p,
                    float *r, int height, int width, int stride,
                    int blockSize, int bandsByBlock ){

        // Block index
        int bx = blockIdx.x;
       int by = blockIdx.y;

        // Thread index (current coefficient)
        int tx = threadIdx.x;

int ib = bands[ ( by * ((width/8)*bandsByBlock) ) +
                                        (bx * bandsByBlock + tx) ];
int a = p[ ( by * ((width/8)*bandsByBlock) ) +
                                        (bx * bandsByBlock + tx) ];
float b = r[ib];

int idx = (by * blockSize + iY[ib]) * stride +
                                (bx * blockSize + iX[ib]);

if( a == 0 && b == 0){
newImage[idx] = ( image[ idx ] / b) + 1;
}else if( a == 1 && b == 1){
newImage[idx] = ( image[ idx ] / b) - 1;
}
__syncthreads();
}
```

A.1.4 Quantization

This function was applied using a library of CUDA. Due to the facility of use of this library, it was not necessary to program it.

A.1.5 Watermark Extraction Operation

This operation is based on equation 5.8, and it needs to load the watermarked image from the host memory to the GPU global memory to extract the watermark. As a result, it generates the watermark that was embedded in the last steps.

```
dim3 ThreadsWaterBlocks2(bandsByBlock);
dim3 GridWaterBlocks2(Size.width/BLOCK_SIZE,
                                        Size.height/BLOCK_SIZE);
```

The result is stored in *wm* and it will be compared with the original watermark using the Normalized Correlation (NC) shown in equation 7.1.

```
__global__ void CUDAKernelWaterExtraction( float *image, int *wm,
                float *r, int *bands, int height, int width, int stride,
                int blockSize, int bandsByBlock ){


    // Block index
    int bx = blockIdx.x;
   int by = blockIdx.y;

    // Thread index (current coefficient)
    int tx = threadIdx.x;
    //int ty = threadIdx.y;


int ib = bands[ ( by * ((width/8)*bandsByBlock) ) +
                                    (bx * bandsByBlock + tx) ];

float c = image[ (by * blockSize + 0) * stride +
                                     (bx * blockSize + 0) ];

float a = image[ (by * blockSize + iY[ib]) * stride +
                                     (bx * blockSize + iX[ib]) ];

float b = r[ib];

if ( a * b >= c){
wm[ ( by * ((width/8)*bandsByBlock) ) +
                                (bx * bandsByBlock + tx) ] = 1;
```

```
}else{
wm[ ( by * ((width/8)*bandsByBlock) ) +
                                (bx * bandsByBlock + tx) ] = 0;
}

__syncthreads();

}
```

Appendix B

Analysis, Design and Implementation of PSO Algorithm

This appendix shows how the PSO algorithm was implemented using CUDA C.

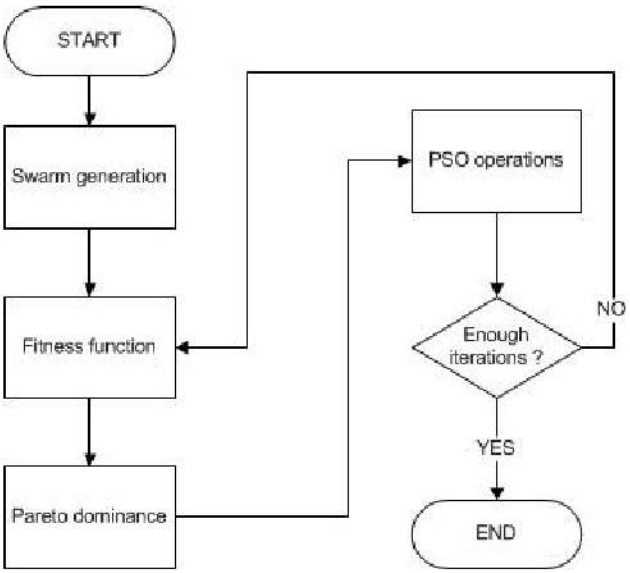

Figure B.1: Flow Diagram of watermarking algorithm (Shieh + PSO).

The PSO algorithm has the next set of steps:

I. The swarm initialization generates one swarm for each block of 8x 8 and it has N particles, each particle has the position or bands to insert the watermark image.

II. The insertion and extraction operations are executed to calculate the MSE and NC used to estimate the Pareto dominance (objetive function).

III. Pareto dominance is applied to get the best particles in each swarm.

IV. Before executing PSO operations, a random numbers array is calculated in the GPU and stored there, it is necessary at the moment of the particle velocity calculus. The PSO operations are executed to generate the next positions or bands to insert the watermark image.

V. Steps II, III and IV are in a loop of M iterations.

B.0.6 Random number generation

To generate the random numbers, the CURAND library was used. It provides facilities that focus on the simple and efficient generation of high-quality pseudorandom numbers on the GPU.

```
size_t n = 20 ;
curandGenerator_t gen;
float *devData;

/* Allocate n floats on device */
HANDLE_ERROR( cudaMalloc( (void **)&devData,
                          n * sizeof(float) ) );

/* Create pseudo-random number generator */
CURAND_CALL( curandCreateGenerator( &gen,
                          CURAND_RNG_PSEUDO_DEFAULT ) );

/* Set seed */
srand48( time(NULL) );
CURAND_CALL( curandSetPseudoRandomGeneratorSeed( gen,
                          lrand48() ) );

/* Generate n floats on device */
CURAND_CALL( curandGenerateUniform(gen, devData, n) );
```

B.0.7 PSO operations

These operations are based on 6.1 and 6.2 equations. The operations need as parameter the particles of each swarm. Each particle is loaded in shared memory, and at the end of the operations the outcomes are returned to the global memory. The results are used to generate new positions to insert the watermark image.

To take advantage of the parallelism in CUDA, each block executes its own evaluations of the functions. If the image size is 512x512, there are generated 4096 blocks (see chapter 8). For example, in the case of the evaluation of the velocity value, if there are five particles in each block, then five operation are executed in parallel in the 4096 blocks, for each particle it is assigned one thread. 20480 threads are working in parallel -4096 (blocks) * 5 (threads)- compared with the 20480 operations that would have been in the sequential mode.

In the case of velocity and position vectors, are assigned threads as number of particles by block.

B.0.7.1 Velocity

```
__device__ void updateVelocitiesGPU( Particle * particles,
Particle * gBest, float *radomNum, int swarmSize, float C1,
float C2 ){

int tid = blockIdx.x;
int tx = threadIdx.x;
int tid2 = ( blockIdx.x * 4 )+threadIdx.x;

__shared__ float a[4], b[4], c[4];

while( tid < swarmSize ){

a[tx] = particles[tid].vel.vel[tx];
b[tx] = C1 * radomNum[tid2] * ( particles[tid].lbest.pos[tx]
                        - particles[tid].pos.pos[tx] ) ;
c[tx] = C2 * radomNum[tid2] * ( gBest->pos.pos[tx]
                        - particles[tid].pos.pos[tx] ) ;

particles[tid].vel.vel[tx] =                a[tx]+b[tx]+c[tx];

tid += blockDim.x * gridDim.x;
tid2 += blockDim.x * gridDim.x;
}
}
```

B.0.7.2 Position

```
__device__ void updatePositionGPU( Particle * particles,
                                   int swarmSize ){

int tid = blockIdx.x; int tx =
threadIdx.x;
__shared__ float a[4];
__shared__ int c[4];

while( tid < swarmSize ){

a[tx] = (particles[tid].vel.vel[tx]*100) +
                (particles[tid].pos.pos[tx]*100);
c[tx] = fabs( a[tx] );
c[tx] = (c[tx] % 63) + 1;

particles[tid].pos.pos[tx] = c[tx];
tid += blockDim.x * gridDim.x;
}
}
```

The calculus of the MSE and the NC are based in the equations 2.5 and 2.6 respectively. To calculate the MSE value there are needed 64 threads for each block, where each thread executes a comparison (if there are 4096 blocks, then 262164 operations -4096 (blocks) * 64 (threads)- would be executed in parallel). The reduction is done by using in every iteration the half of the threads. If there are 64 threads, then the iterations start with 32 threads. The threads with indices lesser than this value do the job.

The figure B.2 shows the assignation threads for the reduction operation. Fer each itera- tion, the threads are divided by the half. At the end of the operation just one thread stores the result.

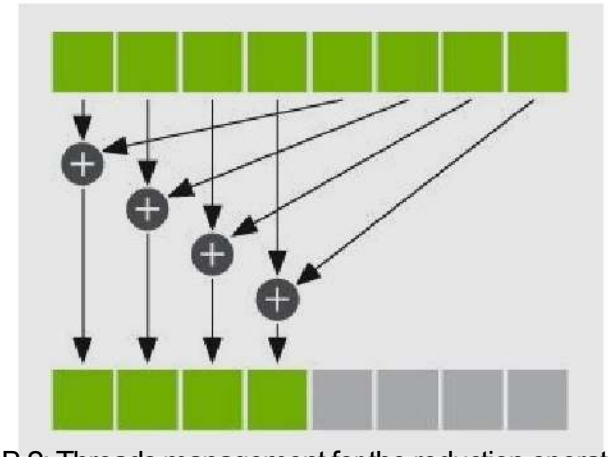

Figure B.2: Threads management for the reduction operation.

To calculate NC just 4 threads are required, each thread executes one bitwise operation. If 4 bits of the watermark were inserted by block, just 4 threads would be needed to make the comparisons (if there are 4096 blocks, then 16384 operations -4096 (blocks) * 4 (threads)- would be executed in parallel).

B.0.7.3 MSE

```
__global__ void MSEKernel( byte *Img1, byte *Img2,
                           float * answer, int Stride, ROI Size ){

__shared__ float cache[64];

// Block index
int bx = blockIdx.x;
```

```
int by = blockIdx.y;

// Thread index (current coefficient)
int tx = threadIdx.x; int ty =
threadIdx.y;

// Indices
int idx = (by * 8 + ty) * Stride + (bx * 8 + tx);
int ith = ty * 8 +tx;

cache[ ith ] = POW( (Img1[ idx ] - Img2[ idx ]) );

__syncthreads();

int i = 32;              // total block / 2

while (i != 0) {
if (ith < i)
cache[ith] += cache[ith + i];
__syncthreads();
i /= 2;
}

int bidx = by * Stride + bx;
if (ith == 0)
answer[bidx] = cache[0]/64;

}
```

B.0.7.4 NC

```
__global__ void ncKernel( int *waterO, int *waterE,
                          float *answer, int Stride ){

  __shared__ float cache[4];

// Block index
int bx = blockIdx.x; int by
= blockIdx.y;
```

```
int tx = threadIdx.x;

// Indices

int idx = (by * 4 )* 64                 + (bx * 4 + tx);

cache[ tx ] = waterO[ idx ] ^ waterE[ idx ];
__syncthreads();

int i = 2;
while (i != 0) {
            if (tx < i)
                        cache[tx] += cache[tx + i];
                          __syncthreads();
                          i /= 2;
             }

int bidx = by * 64 + bx;
             if (tx == 0)
                 answer[bidx] = cache[0]/4;
}
```

Appendix C

Utilities

Utilities are the stuctures that help in the algorithm, but they are not involved in the algorithm.

C.1 Timer.h

This structure is used to meassure the time when a code is running on the CPU or the GPU. The structure has two methods: *startTimer*(), to initialize the timer, and *stopTimer*() to stop the timer.

Figure C.1: Timer struct.

To meassure the time, the *sys/time.h* library is used.

```
struct timeval start, stop;

void startTimer(){
        gettimeofday(&start, 0);
}
```

```
float stopTimer(){

    gettimeofday(&stop, 0);

    float elapsedTime = (stop.tv_sec+stop.tv_usec*1e-6)-
                                (start.tv_sec+start.tv_usec*1e-6);

    return elapsedTime;
}
```

C.2 ShiehUtilities.h

ShiehUtilities is used to load one image and one watermark in memory, it has two methods: *loadImage*() and *loadW atermark*(). The path, and the image and the watermark names are stored in a file. Due to this, it is not necessary to re-compile the code to use a new image or watermark.

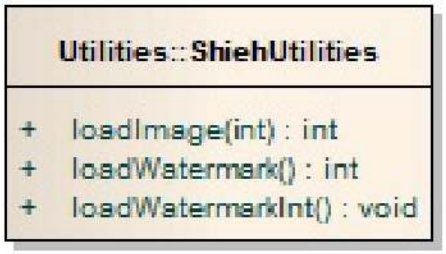

Figure C.2: ShiehUtilities struct.

```
struct      ShiehUtilities{

//IMAGE
char *SampleImageFname;
char *SampleImageFnameResCUDA2;
char *pSampleImageFpath;
ROI ImgSize;
int ImgStride; //Step between two sequential rows
byte *ImgSrc;
byte *ImgDstCUDA2;

//WATERMARK
char *WatermarkPath;
ROI WaterMImgSize;
int WaterMStride;
byte *WaterMSrc;
ROI WaterMSize;
int *WaterMSrcInt;

//BANDS
int *bands;
int bandsstart;
int bandsend;
```

```
int bandSize;
        int bandsbyblock;

ShiehUtilities(){

ImageParamLoader    ipl;
ipl.loadProperties();

SampleImageFname = ipl.getOrigin();
SampleImageFnameResCUDA2 = ipl.getDestination();
pSampleImageFpath = ipl.getHome();
WatermarkPath = ipl.getWatermark();
bandsstart = ipl.getBandsStart();
bandsend = ipl.getBandsEnd();

}


/**
************************************************************
* This function generates the initial bands in a row
*
* \param totalElements                    [IN] - Is equivalen to the
***                                            number or blocks
                                               times number of bands
* \return Array with all the bands for whole image
*/

void makeDiagonalBands(int totalElements){...}



/**
****************************************************************
* This function load the initial bands from a file
*
* \param totalElements [IN] - Is equivalen to the
*                                               number ob blocks times
*                                               number of bands
*
* \return Array with all the bands for whole image
*/
```

```
void loadDiagonalBands(int totalElements){...}


/**
************************************************************
* This function generates the initial bands in a row
*
* \param totalElements [IN] - Is equivalen to the
*                                                    number ob blocks times
*                                                    number of bands
* \param baux [IN] - The bans choosen by the user
* \param bandsByBlock [IN] - Number of band for each block
*
* \return Array with all the bands for whole image
*/

void makeDiagonalBands2(int totalElements,
                                int *baux, int bandsByBlock ){...}




/**
**********************************************************
* Load the original image to be used for watermarking
**
* \param op [IN] - Is the option to load de
*                                          original image
*                                          or the watermarked image
*
* \return integer, 0 = Successful, 1 = Error
*/

int loadImage(int op){...}


/**
************************************************************
* Load the watermark image
**
*
* \return integer, 0 = Successful, 1 = Error
```

```
*/

int loadWatermarkInt( ){...}
```

C.3 ImageParamLoader.h

ImageParamLoader is used to load the path where the auxiliar files are stored. Image and watermark names are extracted from a file called *image.properties* which contains the following information:

```
ORIGIN          = barbara.bmp
DESTINATION = data/barbara_cuda.bmp
IMAGE_HOME = data/barbara.bmp
WATERMARK     = data/logo128x128.bmp
BANDSSTART = 28
BANDSEND       = 31
BANDSBYBLOCK = 4
```

In the class, the method *loadProperties*() loads the parameters and they can be called by their getter function.

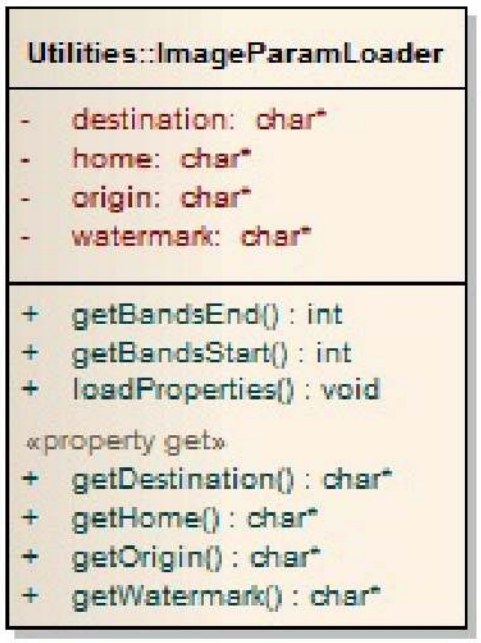

Figure C.3: ImageParamLoader class.

```
class     ImageParamLoader{
```

```
private:
char *origin;
char *destination;
char *home;
char *watermark;
int bandsstart;
int bandsend;

public:


ImageParamLoader();
~ImageParamLoader();

char* getOrigin();
char* getDestination();
char* getHome();
char* getWatermark();
int getBandsEnd();
int getBandsStart();

void loadProperties();

protected:




};
```

C.4 BmpUtil.h

BmpUtil is a library provided by NVIDIA. It contains basic image operations which are used by ShiehUtilities.h to load the image and the watermark into the memory.

Bibliography

[1] Mark Johnston & Mengjie Zhang Ammar Mohemmed. Particle swarm optimization based multi-prototype ensembles. GECCO, pages 57-63, July 2009.

[2] ByteScout. Digital watermark types, 2011. http://bytescout.com/.

[3] NVIDIA Corporation. Nvidia cuda c programming, 2009.

[4] NVIDIA Corporation. Nvidia cuda c programming - best practices guide, 2009.

[5] NVIDIA Corporation. Cuda curand library, 2010.

[6] Chin-Shiuh Shieh et al. Genetic watermarking based on transfrom-domain techniques. Pattern Recognition, 2004.

[7] Luca Mussi et al. Evaluation of parallel pso algorithms within the cuda architecture. Elsevier, 2010.

[8] & R. Garduno-Ramirez J. Heo, K. Lee. Multiobjective control of power plants using particle swarm optimization techniques. IEEE Transactions on Energy Conversion, vol. 21, no. 2, 2006.

[9] Jason Sanders & Eduard Kandrot. Cuda by Example: An introduction to general purpose GPU programming. Addison-Wesley, USA, first edition, 2010.

[10] R. C. Kennedy, J. & Eberhart. Particle swarm optimization. Proceedings of IEEE International Conference on Neural Networks, pages 1942-1948, 1995.

[11] Anton Obukhov & Alexander Kharlamov. Discrete cosine transform for 8x8 blocks with cuda, 2008. by NVIDIA.

[12] M. Donelli & A. Massa. Computational approach based on a particle swarm optimizer for microwave imaging of twodimensional dielectric scatterers. IEEE Transactions on Microwave Theory and Techniques, vol. 53, no. 5, 2005.

[13] David B. Kirk & Wen mei W. Hwu. Programming Massively Parallel Processors: A Hands-on Approach. Morgan Kaufmann, USA, first edition, 2010.

[14] T. Huang & A. S. Mohan. A microparticle swarm optimizer for the reconstruction of microwave images. IEEE Transactions on Antennas and Propagation, vol. 55, no. 3, 2007.

[15] M. Kutter & F.A.P. Petitcolas. A fair benchmark for image watermarking systems. Security and Watermarking of Multimedia Contents, 1999.

[16] Kitti Attakitmongcol & Arthit Srikaew Prayoth Kumsawat. The effects of transformation methods in image watermarking. -, 2010.

[17] F. Zhang & H. Zhang Radu Sion. Digital watermarking capacity and reliability. International Conference on E- commerce Technology, 2004.

[18] C. Xavier & N. Karssemeijer S. Selvan. Parameter estimation in stochastic mammogram model by heuristic optimization technique. IEEE Transactions on Information Technology in Biomedicine, vol. 10, no. 4, 2006.

[19] Mark P. Wachowiak & Renata Smolikova. An approach to multimodal biomedical image registration utilizing particle swarm optimization. IEEE Transactions on Evolution- ary Computation vol. 8 no. 3, 2004.

[20] F. Zhang & H. Zhang. Digital watermarking capacity research. International Conference on Communica- tions, Circuits and Systems, ICCCAS, 2004.