ARISTOTLE UNIVERSITY OF THESSALONIKI

FACULTY OF ENGINEERING
DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING
DIVISION OF ELECTRONICS AND COMPUTER ENGINEERING

FLCC: A LIBRARY FOR FAST COMPUTATION OF CONVOLUTION AND LOCAL CORRELATION COEFFICIENTS

DIPLOMA DISSERTATION

Georgios Papamakarios

Georgios Rizos

Supervisor: **Nikos P. Pitsianis**
Assistant Professor

Thessaloniki, December 2011

Abstract

Convolution and correlation between digital images constitute two of the most basic and significant operations in the field of digital image processing. Their considerably high computational complexity, though, combined with current demands concerning time performance and arithmetic precision, has been a constant challenge in scientific research. The above problem becomes exacerbated in cases where there is also the need for local normalization of the coefficients and/or the number of the image dimensions increases. In this dissertation we describe a set of algorithmic methods to efficiently deal with the problem, without sacrificing the arithmetic precision of the computations. Furthermore, we present and analyze the FLCC library, a powerful yet handy computational tool, which implements the aforementioned methods whilst utilizing the strengths of modern efficient parallel architectures (multi-core systems, GPUs) in order to achieve fast computation of convolutions and correlation coefficients between 2D and 3D images. We conclude with indicative experimental results, which demonstrate the usefulness and efficiency of the FLCC library.

Acknowledgments

In June 2009 we started with the following:

✓ ***An interesting subject***	we owe it to Mr. Nikos Pitsianis
✓ ***The right attitude***	we owe it to each other
✓ ***Two reliable computers***	we owe it to our parents

The rest was just a matter of time.

Thessaloniki, December 2011

The authors

Table of Contents

List of Figures

List of Tables

0 Introduction

The central subject of this diploma dissertation regards two of the most basic mathematical operations in the field of digital signal processing. The first among them is convolution. Convolution is a process during which a system, mathematically expressed as a signal called filter, modifies an input signal into an output signal. The significance of the aforementioned operation is fundamental in digital signal processing, as much in theory as in practice. It is a fact that convolution is found to be a basic stage of processing in a multitude of modern image processing applications, such as the design of finite impulse response filters, pattern recognition, feature extraction and object detection in computer vision applications and image compression/decompression to name only a few. It is noteworthy that convolution is computed in a very similar manner as another cardinal mathematical operation, cross-correlation. Their difference lies on their physical significance, cross-correlation being a means of computing a similarity degree between two signals. As in the case of convolution, cross-correlation has numerous applications in fields such as pattern recognition, signal detection and time-series analysis.

The second mathematical operation with which we occupy ourselves is the table of correlation coefficients with local normalization or local correlation coefficients or simply LCCs. From a statistical point of view, the correlation coefficient is a measure of the linear similarity/dependence between two random variables. The table of local correlation coefficients between images, on the other hand, is actually a variation of cross-correlation in which we also proceed to the local normalization of the signals. The tables of correlation coefficients are often used in image processing applications such as template matching, image registration, change or motion detection and more.

Despite their usefulness and the frequency with which they appear in practice, the above three operations and especially the table of local correlation coefficients unfortunately present a rather high arithmetic complexity that leads to increased computation times. This becomes displeasingly evident in a multitude of applications where there is also the matter of deadlines for the execution of the computations, a most indicative example being real-time applications (such as real-time video processing, computer vision and lots more). Furthermore, modern applications require the processing of images of a very high resolution and definition (as in HDTV) and even of three-dimensional images (as in axial tomography). The discovery of solutions that lead to a practically efficient execution of the computations in question is therefore necessary.

To efficiently compute convolution (and cross-correlation) a number of algorithms have been developed which, using the convolution/correlation theorem, transfer the major bulk of the computations from the spatial domain to the frequency domain. Thanks to this, the use of fast Fourier transforms becomes possible, leading to a significant decrease of computational complexity. It is also observed that by using the frequency domain, in the case the computation concerns multiple convolutions while using the same filter, the pre-computation of a part of the whole operation is permitted, thus further decreasing the execution time per single convolution. However, in the case of a table of local correlation coefficients the development of frequency domain methods is not directly evident. Instead, in the case of a table of correlation coefficients, algorithms have turned up that try to reduce the execution time by computing arithmetic approximations of the result, using devices such as the application of general normalization instead of local. However, these approximations do not always pose a desired solution, as often applications are encountered where the arithmetic precision of the calculations is of critical importance, like applications related to the processing of information of medical context. Especially interesting, finally, is an effort to confront the problem by using powerful hardware architectures. Increasingly promising are parallel computing architectures like graphics processing units.

Our goal in this diploma dissertation is to develop a complete approach for the fast computation of convolution and correlation coefficients with local normalization, without making any compromises concerning the precision of the results. We approach the problem on three levels. The first is the development of efficient algorithms aiming to reduce the arithmetic complexity of the calculations. Specifically we examine the basic two kinds of methods for the computation of each of the operations, the computation in the spatial domain and the computation in the frequency domain using fast Fourier transforms, and we develop efficient algorithms that exploit the potential of each method. The second level consists of the use of two modern parallel computer architectures that have been chosen based on their high performance and also their wide availability. These are the multi-core processors and the graphics processing units. The third and final level is the introduction and use of a mechanism that performs the computations according to the plan-execute model. This mechanism combines the advantages of all the individual approaches in a unified scheme and allows for the addressing of the general computational problem in its entirety.

As a result of the above, we have created the C (or C++) language library named FLCC, a complete software package that provides its user the ability of executing fast computations of convolutions and correlation coefficients with local normalization. The name of the library is an acronym and it means Fast Local Correlation Coefficients. FLCC provides a simple and concise interface to its user that is able to carry out the computations under examination in a versatile and effective way. FLCC comprises the crystallization of our total effort for an efficient approach to the problem: it implements the various algorithms developed and it executes them on the parallel architectures we have chosen. A vital part in the use of the library is the plan-execute mechanism, the central idea of which is the automatic search for the fastest possible means of execution on the user's computer system for each distinct case of a computation. The FLCC library is freely available, fully functional, is at version 1.3 at the time of syntax of this diploma dissertation and is accompanied by full documentation in the form of a usage manual.

We will proceed with a small synopsis of what will follow next in the document.

In chapter 1 we will do a quick recap to all the basic theory concepts as they will be used as a basis for the next chapters. We will provide the definitions for convolution, correlation and tables of correlation coefficients with local normalization. We will furthermore talk about the fact that correlation is actually a convolution between the image and the inverted filter, the representation of discrete signals in the Fourier domain and also about how a convolution or correlation can be expressed in the Fourier domain through the convolution/correlation theorem. In the end of the chapter all the theoretical groundwork will have been laid and thus the inherent similarities in the calculation of the three operations under examination will be evident.

In chapter 2 we will see that each of the operations is amenable for solution using two different methods, those being the direct method, i.e. the computation in the domain of space, and the Fourier domain method, i.e. the computation in the domain of frequency. By capitalizing on the ideas of these two methods, we develop efficient algorithms for each one of the operations examined. Furthermore, we find the computational complexities for each one of the algorithms as a function of the sizes of the image and the template/filter and we locate the theoretic “preference bounds” between the individual algorithms. In this chapter we also discuss another matter, that of carrying out the computations under examination between one template/filter and a series of images of the same size and dimensions, which we call a stream of images. We examine the algorithmic improvements resulting from the above case, we find the new computational complexities and we locate the new “preference bounds”.

In chapter 3 we will discuss the modern parallel computer architectures which we use to achieve higher performance, i.e. multi-core processors and graphics processing units. We will describe the organization of their hardware, their memory hierarchy and the means by which they are programmed. We will make references to the tools that we use to program them (pthreads and CUDA) and the libraries we use to execute fast Fourier transforms on each one (FFTW and CUFFT).

In chapter 4 we will discuss the FLCC library itself. We will begin with a synopsis of the features provided by the version that is valid at the time of syntax of this document. We will describe FLCC’s interface and how a user may find use for it in a program of their own making. Finally, we will talk about its internal operation, i.e. how the algorithms were implemented on each of the architectures, issues regarding the implementation of streaming and of course about the plan-execute model.

In chapter 5 we submit the FLCC library to a series of experiments and we present their results. Initially we will present the execution times for the different algorithms and we will examine the extent to which the theoretic “preference bounds” between the algorithms are validated. We will also study the library’s behavior on each individual architecture and we will make an effort to compare the latters’ performance. Finally, we will study the performance of FLCC’s planning stage, i.e. its decision on a most suitable computation method for various convolution and correlation coefficients problem sizes (image and template/filter).

Chapter 6 consists of a review of the subjects undertaken in this dissertation and a synopsis of the results produced.

In chapter 7 we ponder on how FLCC library may be improved, expanded on and become more versatile in its use and also on other matters that resulted during the time we were engaged on this intricate yet interesting subject.

Finally, chapter 8 is an appendix in which we provide, without comments, results produced by executing chapter 5’s experiments on a different computer system.

1 Theory Elements

This chapter serves as a quick and easy reference to preparative notions regarding the scientific field of this diploma thesis. We begin with a small number of basic definitions and we describe our field of interest (1.1). We move on to provide the mathematical definitions of convolution (1.2) and correlation (1.3) and we explain their physical meaning and their properties in the field of digital signal processing. We then give the definition for the normalized correlation coefficient or else Pearson coefficient and we discuss the table of correlation coefficients and the idea of local normalization between an image frame and a template (1.4). Furthermore, we make a reference to one of the most important tools of digital signal processing, namely the Fourier domain, and we provide the definitions for its basic transforms (1.5). We conclude this theory summary with a review of the well-known convolution/correlation theorem (1.6).

We would like to add here that the experienced reader may ignore sections, or even the entire chapter, and proceed directly to the next one, where our work actually begins.

1.1 Images as Signals

In this thesis we deal with the processing of two-dimensional and three-dimensional **digital images**. A digital image is a rectangular arrangement of pixels (picture elements), where each one corresponds to a real or vector value. This value expresses the color and the luminance of the corresponding pixel. From now on we will deal exclusively with real-valued images, that is, single-color grayscale images. This does not sacrifice the generality of our approach, since for multi-color images (such as RGB) each color component may be separated and be analyzed on its own.

On the above perspective, from now on we will view an image as a two-dimensional or three-dimensional signal which is discrete (it constitutes an integer-indexed sequence) and it receives exclusively real values. Such a signal can be compared to a function, in the sense that it represents the relation between two parameters – variables, an independent and a dependent one. The independent variable can be alternatively named sample index (and will correspond to the according pixel) and the dependent one may be called magnitude.

A further characteristic images have when viewed as discrete signals is that they constitute signals of finite size. We call impulse or sample a delta function which has been shifted and scaled with a random constant. For instance $a\delta[n-n_0]$ is an impulse. We can therefore view each discrete signal as a summation of single impulses. A finite signal is equal to a sum of a finite number of impulses, assuming at the same time that all the remaining sample values are equal to zero.

Summing up, for the rest of this document we will refer exclusively to signals which are:

- discrete
- real-valued
- finite in size
- two or three-dimensional

and we will imply that these signals represent images. In Figure 1.1 we show an example of mathematical representation of an RGB image in the form of three real-valued signals.

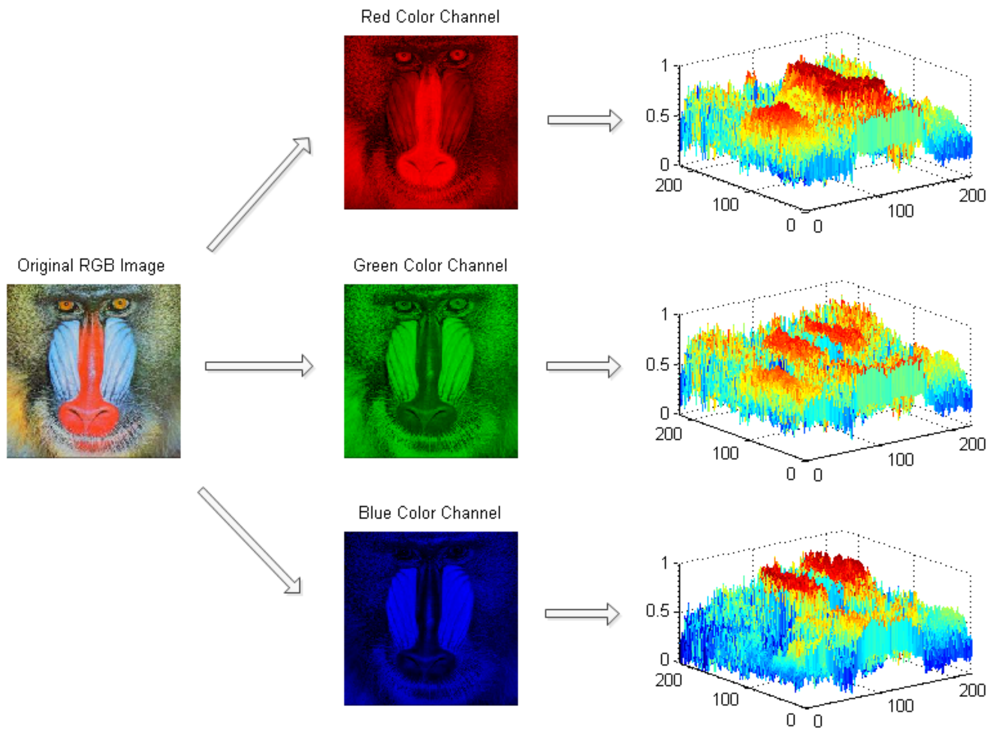

Figure 1.1: Representation of an RGB Image as a Signal

1.2 Convolution

Image processing devices can mathematically be described through the idea of the system. A way to define a system is the following: a description of how a signal is differentiated in relation to a second signal. When the system receives a discrete signal as input (input signal), it produces a second signal as output, which is therefore called output signal and represents the result of the processing. A generic manner to symbolize this process is with a mathematical operator – transform, such as $\mathrm{T}\{\cdot\}$. That way, with $x[n]$ and $y[n]$ being the input and output signals respectively, we may write:

$$y[n] = T\{x[n]\} \tag{1.1}$$

The systems which we are interested in are called Linear and Shift Invariant, or simply LSI. The above two properties can be described as follows:

- A system is linear when the following holds

$$T\{\mathrm{a}_1 x_1[n] + \mathrm{a}_2 x_2[n]\} = \mathrm{a}_1 \mathrm{T}\{x_1[n]\} + \mathrm{a}_2 \mathrm{T}\{x_2[n]\} \tag{1.2}$$

  where a_1, a_2 are constants.
- A system is shift invariant when the following holds

$$y[n - n_0] = T\{x[n - n_0]\} \tag{1.3}$$

  where n_0 is an integer that represents the shift and $y[n] = T\{x[n]\}$.

Further analysis of the consequences of the above two properties is considered to be beyond the scope of this theory review.

Suppose now we have an LSI system which we may call S. The signal which is produced at the output of S when it receives the impulse function $\delta[n]$ as its input is called impulse response and it is denoted by $h[n]$. It is interesting to see that the impulse response alone contains the full set of information needed to fully determine the system. Indeed, if one knows the impulse response, one is capable of determining the output of S for any input. It can be shown that for an arbitrary input signal $x[n]$, the corresponding output signal $y[n]$ is always given by the convolution of $x[n]$ with the impulse response $h[n]$.

Let's talk about **convolution**. As we saw above, the convolution is the mathematical relation that interconnects the three signals that are found in the description of an LSI system (input signal, output signal and impulse response) and this is where convolution owes its significance in the field of signal processing. For discrete signals, the sum of convolution is defined as:

$$y[n] = \sum_{k=-\infty}^{+\infty} x[k]h[n-k] \tag{1.4}$$

and in a shorter form is symbolized as:

$$y[n] = x[n] * h[n] \tag{1.5}$$

An important thing we shall note is that in case we refer exclusively to signals of finite size, the size of the output signal depends on the sizes of the other two signals in the equation, from which it results in a straightforward way. To be precise, in one dimension the length of the output signal is equal to the sum of the other two minus one:

$$Y_1 = X_1 + H_1 - 1 \tag{1.6}$$

where Y_1 is the length of signal $y[n]$ along the first (and only) dimension and likewise for $x[n]$ and $h[n]$.

Generalizing to more than one dimensions is a rather simple task. Here follows the definition of convolution for the two-dimensional case:

$$y[n_1, n_2] = x[n_1, n_2] * h[n_1, n_2] = \sum_{k_1=-\infty}^{+\infty} \sum_{k_2=-\infty}^{+\infty} x[k_1, k_2] h[n_1 - k_1, n_2 - k_2] \quad \textbf{(1.7)}$$

For the lengths along each dimension the following holds:

$$\begin{aligned} Y_1 &= X_1 + H_1 - 1 \\ Y_2 &= X_2 + H_2 - 1 \end{aligned} \quad \textbf{(1.8)}$$

while the size of the two-dimensional convolution is equal to:

$$Y_{tot} = Y_1 Y_2 \quad \textbf{(1.9)}$$

Likewise, the definition for the three-dimensional case is:

$$\begin{aligned} y[n_1, n_2, n_3] &= x[n_1, n_2, n_3] * h[n_1, n_2, n_3] = \\ &= \sum_{k_1=-\infty}^{+\infty} \sum_{k_2=-\infty}^{+\infty} \sum_{k_3=-\infty}^{+\infty} x[k_1, k_2, k_3] h[n_1 - k_1, n_2 - k_2, n_3 - k_3] \end{aligned} \quad \textbf{(1.10)}$$

The lengths are:

$$\begin{aligned} Y_1 &= X_1 + H_1 - 1 \\ Y_2 &= X_2 + H_2 - 1 \\ Y_3 &= X_3 + H_3 - 1 \end{aligned} \quad \textbf{(1.11)}$$

and the size of the output signal is:

$$Y_{tot} = Y_1 Y_2 Y_3 \quad \textbf{(1.12)}$$

Finally, we shall generalize the above and talk for an arbitrary number of ν dimensions. The definition of convolution is:

$$y[\boldsymbol{n}] = x[\boldsymbol{n}] * h[\boldsymbol{n}] = \sum_{\boldsymbol{k} \in \mathbb{R}^\nu} x[\boldsymbol{k}] h[\boldsymbol{n} - \boldsymbol{k}] \quad \textbf{(1.13)}$$

where $\boldsymbol{n} = [n_1 \ \ n_2 \ \ \dots \ \ n_\nu]^T$ and $\boldsymbol{k} = [k_1 \ \ k_2 \ \ \dots \ \ k_\nu]^T$.

The lengths are:

$$Y_i = X_i + H_i - 1, \quad i = 1, 2, \dots, \nu \quad \textbf{(1.14)}$$

Finally, the size is:

$$Y_{tot} = \prod_{i=1}^{\nu} Y_i \quad \textbf{(1.15)}$$

We shall note that, in the field of digital image processing, the signal $h[\boldsymbol{n}]$ which we referred to as "impulse response" may alternatively be called "filter" or "template", depending on the implied context. From now on, whenever we refer to convolution we will use interchangeably any of the above terms.

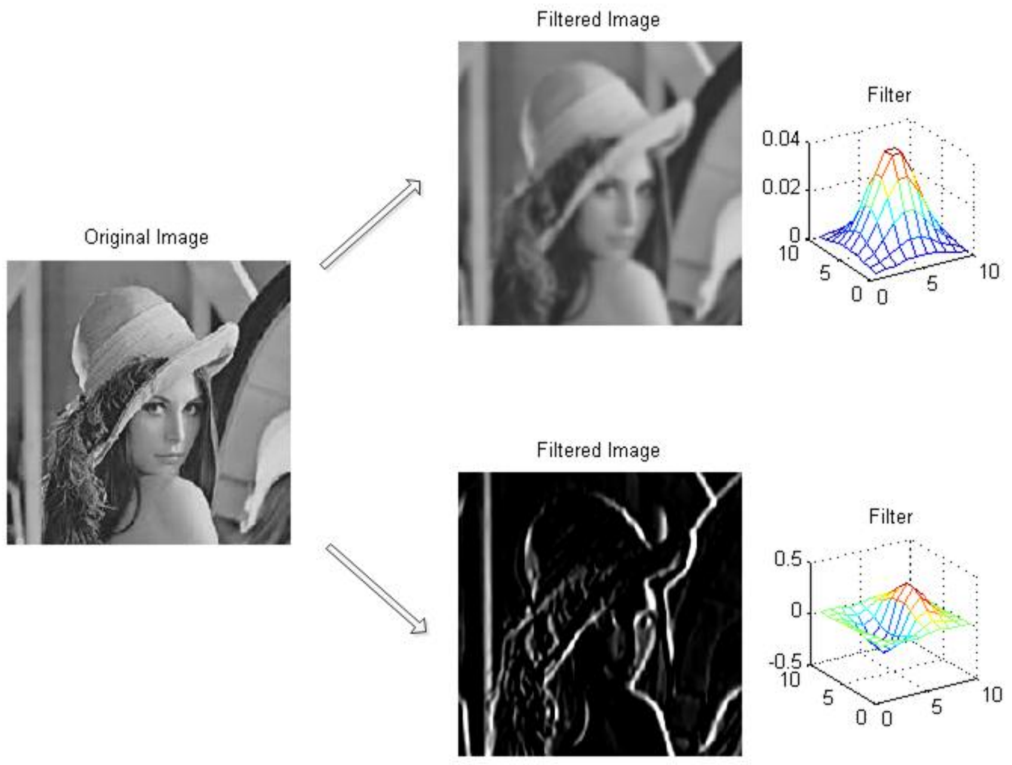

Figure 1.2: Convolution Applications

Two convolution applications on digital images are presented in Figure 1.2. In the first case the original image is convolved with a Gaussian filter in order to produce a "blurred" copy. In the second case the same original image is convolved with a Gabor filter in order to extract its contour.

1.3 Correlation

In this section we study **correlation,** an equally significant mathematical process in the field of digital signal processing. Correlation, from a mathematical point of view, is similar to convolution, in the sense that it uses two signals and produces a third. Nevertheless, these two processes demonstrate a totally different physical behavior. We no longer refer to an LSI system which accepts an input signal and produces some kind of response.

In the heart of correlation lies the following problem: given an input signal, what can be a way to determine where in this signal a second signal – template is located. Using correlation, the answer is a third signal which is called cross-correlation of the two input signals, whose values determine the "similarity degree" between the input signal and the template for every relative positioning. In case the input signal is correlated with itself, the signal we get as the result of the process is called auto-correlation. In Figure 1.3 we can see an example of cross-correlation between a two-dimensional image and a segment of it.

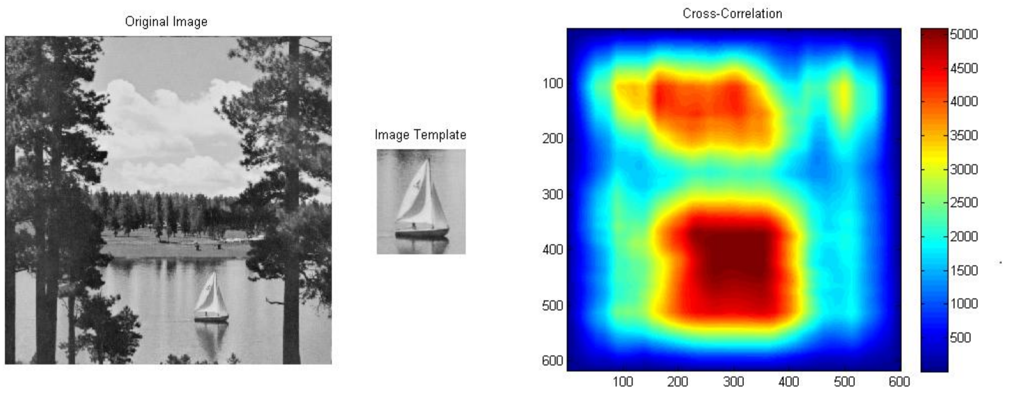

Figure 1.3: Cross-Correlation between an Image and a Template

We proceed to mathematically defining correlation. Denote by $t[n]$ a discrete template or target. The sum of correlation for one-dimensional real-valued signals is the following:

$$y[n] = \sum_{k=-\infty}^{+\infty} x[k-n]t[k] \tag{1.16}$$

and it can be written in a shorter form as:

$$y[n] = x[n] \star t[n] \tag{1.17}$$

The generalization of the above definition for multi-dimensional signals follows. In the two-dimensional case we get:

$$y[n_1, n_2] = x[n_1, n_2] \star t[n_1, n_2] = \sum_{k_1=-\infty}^{+\infty} \sum_{k_2=-\infty}^{+\infty} x[k_1 - n_1, k_2 - n_2]t[k_1, k_2] \tag{1.18}$$

For the tree-dimensional case we get:

$$\begin{aligned} y[n_1, n_2, n_3] &= x[n_1, n_2, n_3] \star t[n_1, n_2, n_3] = \\ &= \sum_{k_1=-\infty}^{+\infty} \sum_{k_2=-\infty}^{+\infty} \sum_{k_3=-\infty}^{+\infty} x[k_1 - n_1, k_2 - n_2, k_3 - n_3]t[k_1, k_2, k_3] \end{aligned} \tag{1.19}$$

Finally, generalizing for the v-dimensional case gives:

$$y[\boldsymbol{n}] = x[\boldsymbol{n}] \star t[\boldsymbol{n}] = \sum_{\boldsymbol{k}\in\mathbb{R}^{\nu}} x[\boldsymbol{k} - \boldsymbol{n}]t[\boldsymbol{k}] \tag{1.20}$$

where $\boldsymbol{n} = [n_1 \;\; n_2 \;\; \dots \;\; n_\nu]^T$ and $\boldsymbol{k} = [k_1 \;\; k_2 \;\; \dots \;\; k_\nu]^T$.

Referring to the most general case, by changing variables in the sum of correlation we may observe that:

$$y[\boldsymbol{n}] = \sum_{\boldsymbol{k}\in\mathbb{R}^{\nu}} x[\boldsymbol{k}-\boldsymbol{n}]t[\boldsymbol{k}] = \sum_{\boldsymbol{k}\in\mathbb{R}^{\nu}} x[\boldsymbol{k}]t[\boldsymbol{n}+\boldsymbol{k}] \tag{1.21}$$

The above formula is of remarkable significance and it means that we can calculate the cross-correlation between two signals by simply reversing the template in the discrete time domain along every dimension and then calculating the convolution as it is already defined. In other words:

$$x[\boldsymbol{n}] \star t[\boldsymbol{n}] = x[\boldsymbol{n}] * t[-\boldsymbol{n}] \tag{1.22}$$

The aforementioned result reveals that, computationally, correlation is equivalent to convolution, provided the template has been already reversed. A similar result can be extracted if we try to reverse the image itself instead of the template.

As for the length of the output signal along each dimension and its total size, the same results as in convolution hold, so we shall not repeat ourselves.

1.4 Correlation Coefficients

In section 1.3 we saw that the process of correlation reveals a “similarity degree” for each segment of a signal with a certain template. The values of correlation, though, are subject to the “magnitude” of the signal and the template values and thus the resulting “similarity degrees” among different positions and/or signals cannot be comparable. This problem can be solved with prior normalization of the correlation values, which leads us to defining the so called correlation coefficients, a thorough description of which can be found in the following subsections.

1.4.1 Correlation Coefficient Definition

Here we describe a widespread measure for the comparison of continuous variables – quantities, namely the **linear correlation coefficient**. It is also known as the Pearson coefficient. For each pair of quantities (x_i, y_i), $i = 0, 1, \dots, N-1$, the linear correlation coefficient r is given by the formula:

$$r(x,y) = \frac{\sum_{i=0}^{N-1}\big(x_i - \mu(x)\big)\big(y_i - \mu(y)\big)}{\sqrt{\sum_{i=0}^{N-1}\big(x_i - \mu(x)\big)^2}\sqrt{\sum_{i=0}^{N-1}\big(y_i - \mu(y)\big)^2}} \tag{1.23}$$

where $\mu(x)$ is the mean value of x_i and $\mu(y)$ the mean value of y_i. N is the number of data points of each quantity.

The above formula can be otherwise written as:

$$r(x,y) = \frac{1}{N}\frac{\sum_{i=0}^{N-1}(x_i - \mu(x))(y_i - \mu(y))}{\sigma(x)\sigma(y)} = \frac{cov(x,y)}{\sigma(x)\sigma(y)} \qquad (1.24)$$

where $cov(x,y)$ is the covariance of x, y and $\sigma(x)$ is the standard deviation of x (same for $\sigma(y)$). **Note**: contrary to some authors, we assume that the variance is calculated by the formula:

$$\sigma(x)^2 = \frac{1}{N}\sum_{i=0}^{N-1}(x_i - \mu(x))^2 \qquad (1.25)$$

dividing by N **and not** by $N-1$.

Each quantity may also be of more than one dimensions, that is, a variable x may be described by a set of data points $x_{i_1,i_2,\dots,i_n}$.

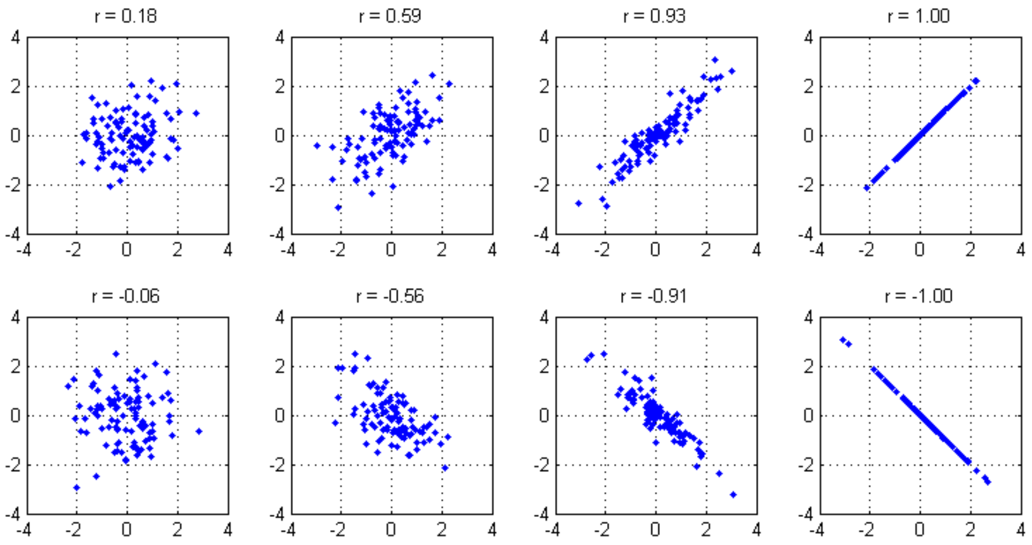

Figure 1.4: Correlation Coefficient Examples

Due to the Cauchy – Schwarz inequality, the value of coefficient r lies always within the closed interval $[-1,1]$. When it is of value 1 we say that there is an absolute positive linear correlation between the two quantities. That means that in a diagram where each axis represents a variable, the points M_i which are determined by the corresponding pairs (x_i, y_i) all lie on a straight line of positive slope and thus x and y increase proportionally to each other. Value 1 is independent though of the specific value of that slope. Likewise, when the coefficient is of value -1 we say that there is an absolute negative linear correlation between the two variables and thus the same as above will hold with the only difference that the straight line now will be of negative slope. A value approaching 0 signifies that the two variables x and y are uncorrelated – orthogonal, with a value of exactly 0 meaning total lack of any linear correlation whatsoever. In Figure 1.4, we can see examples of (x_i, y_i) sets of data points with different correlation coefficients.

1.4.2 Correlation Coefficients with Local Normalization

In this subsection we will study the calculation of **Local Correlation Coefficients** (or LCCs) between an image – frame and a template. The image is due to be analyzed against the template, the latter possibly representing a different image or a subsection of the former.

We view the images and the templates as two-dimensional or three-dimensional variables. Since we deal only with images, we consider that the value of each data point (which in this case may be called pixel) is real.

Denote by T a template and by S an image having N_T and N_S data points respectively. We may assume that in principle the image is larger than the template and that therefore $N_T < N_S$. Similarly $X_T < X_S$ and $Y_T < Y_S$ shall hold, where by D_T we denote the size of dimension D of the template and likewise D_S for the image. Finally, we assume P to be a sub-image or panel of S having the same size as T.

As we have already mentioned, the correlation coefficient between P and T is:

$$r(P,T) = \frac{1}{N_T}\frac{\sum_{x,y}\left(P_{x,y} - \mu(P)\right)\left(T_{x,y} - \mu(T)\right)}{\sigma(P)\sigma(T)} = \frac{cov(P,T)}{\sigma(P)\sigma(T)} \tag{1.26}$$

Let us now consider the degenerate case where $\sigma(P)\sigma(T) = 0$ and which means that either the template or the panel are constant, that is all their points are equal to one another and to their mean value.

If the template is constant, then the problem is reduced to whether the panel is constant as well. This kind of examination can be carried out in a very simple way. We may therefore assume that the template is never constant and that $\mu(T) = 0$ and $\sigma(T) = 1/\sqrt{N_T}$, having normalized it accordingly. A rather more common and interesting case is when the panel is constant. We express the correlation coefficient in the following non-symmetrical formula:

$$\bar{\sigma}(P)r(P,T) = \sum_{x,y}\left(P_{x,y} - \mu(P)\right)T_{x,y} \tag{1.27}$$

where $\bar{\sigma}(P) = \sqrt{N_T}\sigma(P)$.

When the left-hand side vanishes, either the panel is constant with $\sigma(P) = 0$, or the correlation coefficient is zero. Following the assumptions we have made regarding the template, both cases show that the panel is uncorrelated – orthogonal to the template, thus the coefficient shall be set to zero.

We are now in the position to fully describe the field with correlation coefficients between the template T and each and every panel P of the image S. It is going to be a two-dimensional or three-dimensional table, depending on the dimensionality of the template and the image. From now on we will refer to the two-dimensional case, considering the three-dimensional case a simple generalization. We describe the translation of T relatively to the image S as $T(x-u, y-v)$, where (u, v) is the position of a predetermined index point of T in image S. In other words, $(x-u, y-v)$ denotes the position of the template points relatively to the index point. The latter can be arbitrarily chosen and may for instance be the first template point at the north-west corner. According to the above, each panel of image S, which depends on (u, v), can be expressed as:

$$P(x, y; u, v) = S(x, y)B_T(x-u, y-v) \tag{1.28}$$

where B_T is the binary characteristic function of spatial support of the template. After substituting, we get the expression of all the correlation coefficients between T and each and every panel in the image:

$$\bar{\sigma}(u, v)r(u, v) = \sum_{x,y}\big(P(x, y; u, v) - \mu(u, v)\big)T(x-u, y-v) \tag{1.29}$$

where (u, v) is a position in S, which traverses all S, $\mu(u, v)$ and $\sigma(u, v) = \bar{\sigma}(u, v)/\sqrt{N_T}$ are the mean value and the standard deviation of $P(x, y; u, v)$ and $r(u, v)$ is the correlation coefficient between the template and the corresponding panel. The field with the correlation coefficients with local normalization (LCCs) $\{r(u, v)\}$ contains the collective information about the position of the greatest correlation between the template and the corresponding panel. The above is better shown in Figure 1.5, where we illustrate the LCC table between a two-dimensional image and a subsection of it. It is interesting to compare the above to Figure 1.3, which shows the cross-correlation between the same images, albeit without local normalization.

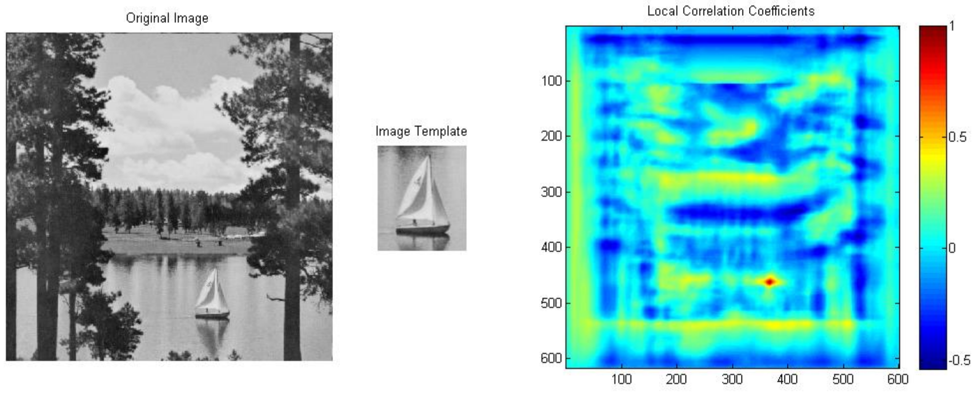

Figure 1.5: LCC Table between an Image and a Template

1.5 Fourier Transform

In this section we study the representation of discrete signals in the frequency domain. We begin with the definition and the description of the Discrete Time Fourier Transform (1.5.1). We move on to analyze the Discrete Fourier Transform (1.5.2). Finally, we talk about the well-known and most effective algorithm for the calculation of the Discrete Fourier Transform, the Fast Fourier Transform (1.5.3).

1.5.1 Discrete Time Fourier Transform

It is well-known that, in the frequency domain, a signal is expressed via the Fourier Transform. In the case where the signal is of discrete time (or space), such as the kind of signals we study in this thesis, the Fourier Transform takes the form of the **Discrete Time Fourier Transform** (or DTFT). For a signal $x[n]$, the DTFT is defined by the following formula:

$$X\left(e^{j\omega}\right) = \sum_{n=-\infty}^{+\infty} x[n]e^{-j\omega n} \tag{1.30}$$

The original signal can be recovered via the inverse DTFT, as described by the following formula:

$$x[n] = \frac{1}{2\pi}\int_{-\pi}^{\pi} X\left(e^{j\omega}\right)e^{j\omega n}d\omega \tag{1.31}$$

We observe that the DTFT of a signal is a function of the continuous variable ω, which expresses the angular frequency in $rad/sample$. It is interesting to notice that, unlike the Fourier Transform of a continuous signal, the DTFT is a periodic function of period 2π.

We are going to generalize the definition of the DTFT for multi-dimensional signals. In the two-dimensional case, the forward and inverse DTFT are respectively defined as follows:

$$X\left(e^{j\omega_1}, e^{j\omega_2}\right) = \sum_{n_1=-\infty}^{+\infty}\sum_{n_2=-\infty}^{+\infty} x[n_1, n_2]e^{-j\omega_1 n_1}e^{-j\omega_2 n_2} \tag{1.32}$$

$$x[n_1, n_2] = \frac{1}{(2\pi)^2}\int_{-\pi}^{\pi}\int_{-\pi}^{\pi} X\left(e^{j\omega_1}, e^{j\omega_2}\right)e^{j\omega_1 n_1}e^{j\omega_2 n_2}d\omega_1 d\omega_2 \tag{1.33}$$

Finally, in the three-dimensional case we get:

$$X\left(e^{j\omega_1}, e^{j\omega_2}, e^{j\omega_3}\right) = \sum_{n_1=-\infty}^{+\infty}\sum_{n_2=-\infty}^{+\infty}\sum_{n_3=-\infty}^{+\infty} x[n_1, n_2, n_3]e^{-j\omega_1 n_1}e^{-j\omega_2 n_2}e^{-j\omega_3 n_3} \tag{1.34}$$

$$x[n_1,n_2,n_3] = \\ = \frac{1}{(2\pi)^3}\int_{-\pi}^{\pi}\int_{-\pi}^{\pi}\int_{-\pi}^{\pi} X\left(e^{j\omega_1},e^{j\omega_2},e^{j\omega_3}\right)e^{j\omega_1 n_1}e^{j\omega_2 n_2}e^{j\omega_3 n_3}d\omega_1 d\omega_2 d\omega_3 \qquad \textbf{(1.35)}$$

1.5.2 Discrete Fourier Transform

The **Discrete Fourier Transform** (or DFT) is a mathematical analysis which applies to finite signals in the discrete time (or space) domain. The result is a second discrete signal, of the same size as the original, which expresses the original signal in the frequency domain. These two signals are equivalent in terms of their informational content and correspond uniquely to one another. Denote by $x[n]$ the original signal and by $X[k]$ its transform, we may symbolize the process of transforming as:

$$x[n] \overset{DFT}{\Longleftrightarrow} X[k] \qquad \textbf{(1.36)}$$

For an original signal $x[n]$ of length N, the one-dimensional DFT is defined by the following formula:

$$X[k] = \sum_{n=0}^{N-1} x[n]e^{-\frac{j2\pi nk}{N}} \qquad 0 \leq k \leq N-1 \qquad \textbf{(1.37)}$$

The $x[n]$ signal can be fully recovered from its transform via the reverse DFT, as described below:

$$x[n] = \frac{1}{N}\sum_{k=0}^{N-1} X[\mathrm{k}]e^{\frac{j2\pi nk}{N}} \qquad 0 \leq n \leq N-1 \qquad \textbf{(1.38)}$$

The aforementioned formulas are a pair of transforms and their usage reveals the equivalence between $x[n]$ and $X[k]$, as it has already been mentioned. From the above formulas we can see that the DFT is essentially an N-point sampling of the DTFT in the interval $[0, 2\pi)$. Note that the DTFT is a continuous function in the frequency domain with a period of 2π.

We are ready to generalize the definition of the above pair of transforms for multi-dimensional signals. In the two-dimensional case, the forward and inverse DFT are respectively defined as:

$$X[k_1,k_2] = \sum_{n_1=0}^{N_1-1}\sum_{n_2=0}^{N_2-1} x[n_1,n_2]e^{-\frac{j2\pi n_1 k_1}{N_1}}e^{-\frac{j2\pi n_2 k_2}{N_2}} \qquad \begin{matrix} 0 \leq k_1 \leq N_1-1 \\ 0 \leq k_2 \leq N_2-1 \end{matrix} \qquad \textbf{(1.39)}$$

$$x[n_1,n_2] = \frac{1}{N_1 N_2}\sum_{k_1=0}^{N_1-1}\sum_{k_2=0}^{N_2-1} X[k_1,k_2]e^{\frac{j2\pi n_1 k_1}{N_1}} e^{\frac{j2\pi n_2 k_2}{N_2}} \qquad \begin{matrix} 0 \le n_1 \le N_1-1 \\ 0 \le n_2 \le N_2-1 \end{matrix} \tag{1.40}$$

While for the three-dimensional case we get:

$$X[k_1,k_2,k_3] = \sum_{n_1=0}^{N_1-1}\sum_{n_2=0}^{N_2-1}\sum_{n_3=0}^{N_3-1} x[n_1,n_2,n_3]e^{-\frac{j2\pi n_1 k_1}{N_1}} e^{-\frac{j2\pi n_2 k_2}{N_2}} e^{-\frac{j2\pi n_3 k_3}{N_3}} \\ \begin{matrix} 0 \le k_1 \le N_1-1 \\ 0 \le k_2 \le N_2-1 \\ 0 \le k_3 \le N_3-1 \end{matrix} \tag{1.41}$$

$$x[n_1,n_2,n_3] = \frac{1}{N_1 N_2 N_3}\sum_{k_1=0}^{N_1-1}\sum_{k_2=0}^{N_2-1}\sum_{k_3=0}^{N_3-1} X[k_1,k_2,k_3]e^{\frac{j2\pi n_1 k_1}{N_1}} e^{\frac{j2\pi n_2 k_2}{N_2}} e^{\frac{j2\pi n_3 k_3}{N_3}} \\ \begin{matrix} 0 \le n_1 \le N_1-1 \\ 0 \le n_2 \le N_2-1 \\ 0 \le n_3 \le N_3-1 \end{matrix} \tag{1.42}$$

The DFT can be equally applied to either real or complex signals. In case the original signal is exclusively real-valued, the result of the transform exhibits the following useful property:

$$X[N-k] = X^*[k] \qquad 0 \le k \le N-1 \tag{1.43}$$

where $X^*[k]$ denotes the complex conjugate of $X[k]$. Furthermore we consider that $X[N] \equiv X[0]$. The above property shows that, for a real signal, the DFT coefficients exhibit conjugate symmetry in relation to their “center” and thus half of them can be easily derived from the other half. As a result, half of the DFT coefficients suffice for fully determining the transform. This property is found quite useful when calculating transforms of real signals. Note that the conjugate symmetry property holds in a similar way in the multi-dimensional case too.

In Figure 1.6 we see the DFT of a two-dimensional image. The DFT magnitude (in a logarithmic scale of luminance) and the DFT phase are shown in the form of two grayscale images.

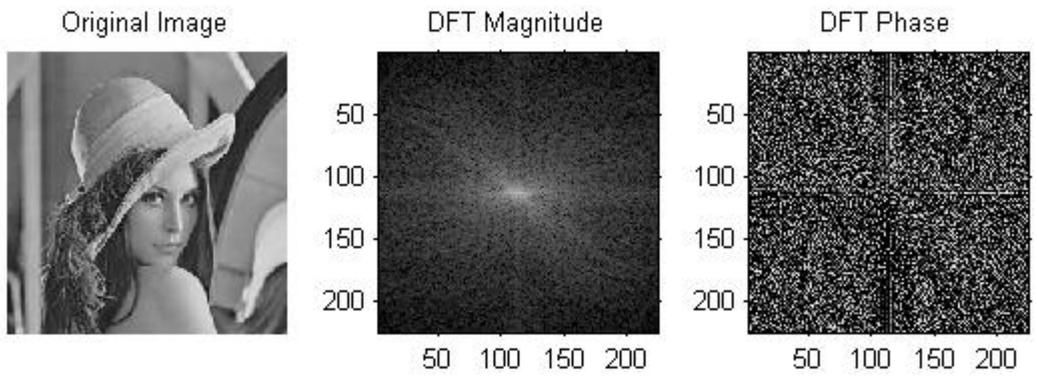

Figure 1.6: Example of Discrete Fourier Transform

1.5.3 Fast Fourier Transform

The direct computation of a Discrete Fourier Transform of a signal of size N, as suggested by its definition, exhibits a complexity of order $O(N^2)$, since for the calculation of each of the $X[k]$ points precisely N multiplications and $N-1$ additions are required. The same is true for the inverse transform as well. Hence, this method is computationally expensive and more efficient alternatives ought to be searched for.

In order to solve the aforementioned problem, a series of fast algorithms has been developed, which are collectively referred to by the title **Fast Fourier Transform** (or FFT). By the name FFT we mean any algorithm which calculates the N-point DFT (of any number of dimensions) in a $O(N \log N)$ time. Up to this date a large number of such algorithms has been proposed, which cover a wide range of characteristics and/or methodologies. As an example we may refer to the Cooley – Tukey algorithm, whose basic idea is the "divide and conquer" methodology, that is the decomposition of an N-point DFT into a series of smaller DFTs. There exist though plenty of other FFT algorithms based on totally different mathematical principles, but a more detailed description of them is considered to lie beyond the scope of this review.

The important element which we shall emphasize is that no N-point FFT performs more than $5N \log N$ constant-time operations. The above holds in the general case of an FFT of a complex signal. We have already mentioned that if the signal to be transformed is real, the result of the transform exhibits conjugate symmetry ($X[N-k] = X^*[k]$). There exist techniques which allow for the exploitation of the above property and the reduction of a real N-point FFT to a complex $N/2$-point FFT. Thus, using such techniques, a real FFT may not perform more than

$$5\frac{N}{2}\log\frac{N}{2} = 2.5N(\log N - \log 2) \cong 2.5N \log N \tag{1.44}$$

constant-time operations. We will refer to this implementation of a real FFT as "half" FFT.

1.6 Convolution – Correlation Theorem

The **convolution – correlation theorem** is one of the most fundamental in systems theory. Its significance lies on the fact that it expresses the processes of convolution and correlation between two signals in the frequency – Fourier domain. Below we examine the form of the theorem in the cases of convolution and correlation separately.

In the case of convolution, the theorem in a few words states that the DTFT of a signal which is equal to the result of the convolution between two other signals, $x[n]$ and $h[n]$, is in fact the product of the DTFTs of $x[n]$ and $h[n]$. To be precise:

$$x[n] * h[n] \overset{DTFT}{\Longleftrightarrow} X(e^{j\omega})H(e^{j\omega}) \tag{1.45}$$

On the other hand, in the case of correlation between signals $x[n]$ and $t[n]$, the expression in the frequency domain is:

$$x[n] \star t[n] \overset{DTFT}{\Longleftrightarrow} X(e^{j\omega})T^*(e^{j\omega}) \tag{1.46}$$

where $T^*(e^{j\omega})$ is the complex conjugate of $T(e^{j\omega})$.

The above theorem holds in a similar way in the case of convolution – correlation of multi-dimensional signals.

2 Algorithms

In chapter 1 the fundamental image processing operations with which we will occupy ourselves were defined and described, i.e. convolution, correlation and correlation coefficients with local normalization. A common characteristic of these operations is the fact that their calculations express high computational complexity, often rendering their usage in computational environments problematic. The above further becomes evident in cases where the size of the images increases significantly (e.g. High Definition) or when the real-time processing of the images is required.

In this chapter we develop and analyze efficient algorithms for the fast computation of the image processing operations under examination. We focus our attention on the computation of convolution (2.1) and correlation coefficients with local normalization (2.2).

2.1 Convolution

2.1.1 Direct Method

The first method for computing convolution that we will present is the direct method, as it directly results from the definition. As we have seen from the equation **(1.13)**, the operation of convolving v-dimensional signals is defined as such:

$$y[\boldsymbol{n}] = x[\boldsymbol{n}] * h[\boldsymbol{n}] = \sum_{\boldsymbol{k}\in\mathbb{R}^{\nu}} x[\boldsymbol{k}]h[\boldsymbol{n}-\boldsymbol{k}] \tag{2.1}$$

where $\boldsymbol{n} = [n_1\ n_2\ \dots\ n_\nu]^T$ and $\boldsymbol{k} = [k_1\ k_2\ \dots\ k_\nu]^T$. Denoting by X_i and H_i with $i = 1, 2, \dots, \nu$ being the lengths along every dimension of the signals $x[\boldsymbol{n}]$ and $h[\boldsymbol{n}]$ respectively, the total sizes of the input and output signals are:

- $N_x = \prod_{i=1}^{\nu} X_i$ for $x[\boldsymbol{n}]$
- $N_h = \prod_{i=1}^{\nu} H_i$ for $h[\boldsymbol{n}]$
- $N_y = \prod_{i=1}^{\nu}(X_i + H_i - 1)$ for $y[\boldsymbol{n}]$

Following the definition equation, the direct method for computing the convolution consists of the following steps:

ALGORITHM: Direct Method for Convolution

1. We pad the image $x[\boldsymbol{n}]$ with $H_i - 1$ zeros on each "side" along every dimension $i = 1, 2, \dots, \nu$, so that the length of each dimension results in $X_i + 2H_i - 2$.
2. We invert the template $h[\boldsymbol{n}]$ along every dimension.
3. We perform the inner product between the inverted template ($h[-\boldsymbol{n}]$) and the panel of the image that gets overlapped by the template and is of equal size to it.
4. We repeat step **3** for every possible position of the template against the image.

In Figure 2.1 the above process is depicted graphically.

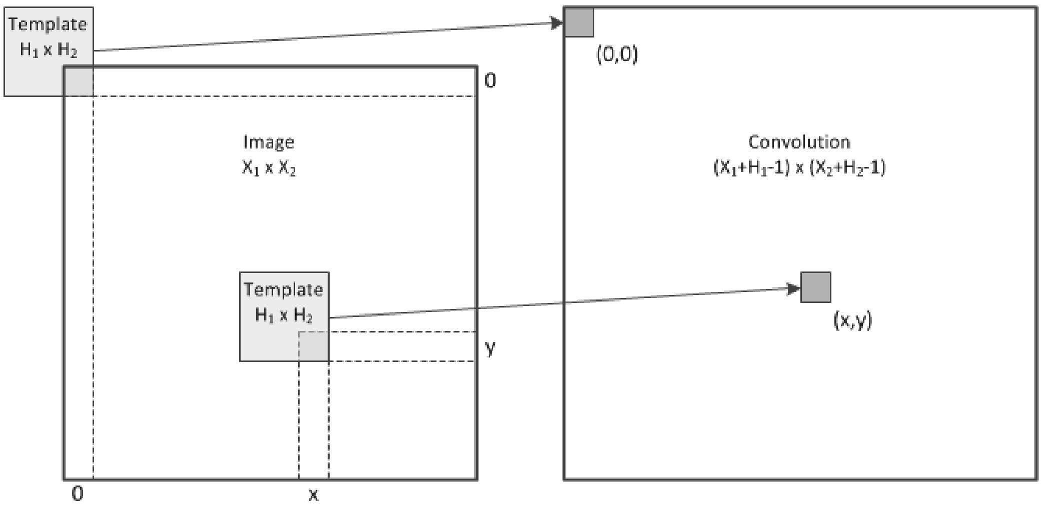

Figure 2.1: Depiction of the Direct Method for Convolution

We will proceed with analyzing the computational complexity of the direct method. We saw that for each of the elements of the output signal, any of which corresponds to the total set of different "overlapping" positions between the two input signals, the computation of the inner product for N_h elements is needed, an operation that requires N_h multiplications and $N_h - 1$ additions. Consequently, the total number of operations for the computation of convolutions equals to:

- $N_y N_h$ multiplications
- $N_y(N_h - 1)$ additions

We observe that the direct method presents complexity of order $O(N_y N_h)$.

It is evident that the complexity of the above method, as it has been presented analytically, is high and thus is not recommended as a general method for computing convolution. However, as a method it does express some points that, if exploited, may confer a rather satisfactory efficiency for some cases of computation. Those points are:

- The calculation of each one of the output elements is independent on the calculation of the others. This shows that there is the possibility for high parallelization of the total computation, rendering the implementation of this method notably suitable on array processors such as many graphics cards.
- This method presents high spatial locality, something that can lead to the efficient use of the in-between cache memory. Indeed, neighboring elements of the output signal have as an input highly overlapped memory segments.
- If the template $h[\boldsymbol{n}]$ is sufficiently small, which something that occurs rather often in practice, the factor $N_y N_h$ of the complexity is also small and is mainly dependent on the size of the image $x[\boldsymbol{n}]$. So, it is possible for the method to prove to be efficient for small sizes N_h. Furthermore, we may note that the whole template is used to compute any of the output elements, something that presents us with the opportunity, if its size is sufficiently small, to have it loaded from the beginning onto a high speed memory (like L1 cache), significantly accelerating the total computation.

The analysis of the above subsection may easily be adapted for the computation of correlation instead of convolution. Indeed, from the function **(1.22)** stems the fact that the application of the above algorithm without the inversion of the template $t[\boldsymbol{n}]$ (step **2**) results to the correlation of the two signals. This relationship may be used likewise for every algorithm for the computation of convolution. This confirms our theoretical statement that convolution and correlation are computationally equivalent, and thus we will not elaborate on the latter.

2.1.2 Fourier Domain Method

The next method we will present is based on the convolution theorem. From equation **(1.45)** we know that for convolution:

$$x[n] * h[n] \overset{DTFT}{\Longleftrightarrow} X(e^{j\omega})H(e^{j\omega}) \quad \textbf{(2.2)}$$

Equation **(2.2)** indicates an alternative way for computing convolution, which we will be referring to as the “Fourier domain method”. This method consists of the transform of signals $x[n]$ and $h[n]$ to the frequency domain, the calculation of their inner product and finally the inverse transform to the temporal (or spatial) domain.

In order to render this method computationally feasible we have to use DFT instead of DTFT. We remind the reader that a DFT of size N constitutes a sample of N points in a period of DTFT. Taking into account the above, we describe the steps of the algorithm for computing convolution of two one-dimensional discrete signals $x[n]$ and $h[n]$ of lengths X_1 and H_1 respectively by using DFT:

ALGORITHM: Fourier Domain Method for Convolution

1. We pad the two signals with zeros on their end, such that both of them are of length $N_1 = X_1 + H_1 - 1$ (as is the result).
2. We calculate the N_1-point DFTs of the two signals, $X[k]$ and $H[k]$.
3. We multiply point-by-point the results of the DFTs so as to create the product $Y[k] = X[k]H[k]$.
4. We calculate the inverse DFT of $Y[k]$. This result will be equal to $y[n] = x[n] * h[n]$.

In Figure 2.2 the above algorithm is depicted in the form of a flow chart.

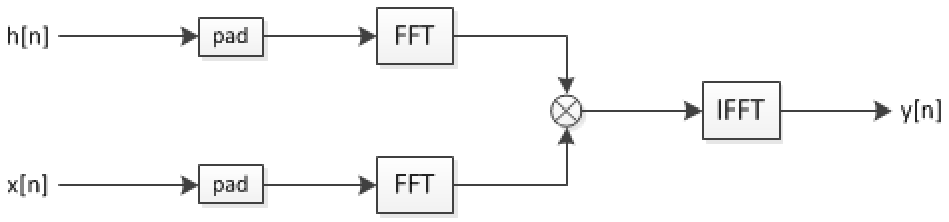

Figure 2.2: Fourier Domain Method for the Computation of Convolution

The description of the above algorithm regards the computation of convolution of one-dimensional signals. In the case we want to use the same process to compute convolution of signals of more dimensions, the process is similar. Specifically, let $x[\boldsymbol{n}]$ be a v-dimensional discrete signal with dimensions X_1, X_2, …, X_v and $h[\boldsymbol{n}]$ be a v-dimensional discrete signal with dimensions H_1, H_2, …, H_v. The only difference is that we pad the two signals with zeros on their ends, such that their every dimension is of length $N_1 = X_1 + H_1 - 1$, $N_2 = X_2 + H_2 - 1$, …, $N_v = X_v + H_v - 1$, and we use forward and inverse DFTs of size $N_1 \times N_2 \times \ldots \times N_v$.

Let us proceed to the analysis of the computational complexity of the above method. The total size of the convolution will be $N_y = \prod_{i=1}^{v} N_i$. We saw that the computation of three DFTs (two forward and an inverse) and one point-by-point multiplication are needed, all of size N_y. The transforms may be calculated via FFT, with each one demanding $5N_y \log N_y$ operations. If we take into account the fact that the signals $x[\boldsymbol{n}]$, $h[\boldsymbol{n}]$ and $y[\boldsymbol{n}]$ are all real, we observe that we can use "half FFTs instead of regular ones, so that each FFT requires only $2.5N_y \log N_y$ operations. For the multiplication N_y operations are needed, one per element. Thus, the total number of operations equals to:

$$3(2.5N_y \log N_y) + N_y = 7.5N_y \log N_y + N_y \cong 7.5N_y \log N_y \qquad \textbf{(2.3)}$$

We assume that the linear factor that is due to the multiplication is negligible relative to the FFT factor. We notice that this method is of complexity of order $O(N_y \log N_y)$.

Regarding the implementation of the above algorithm in a parallel environment, it is dependent on the parallelizing potential of the FFT. Generally, the FFT is considered to be a rather “difficult” algorithm for parallelization. However, we assume that the analysis of the parallelization of FFT is out of the bounds of this dissertation. As for the multiplication, it is an operation with a high potential for parallelization on element level, rendering it an ideal operation for implementation on vector processors.

Finally, regarding the computation of correlation instead of convolution, it can be done likewise by inverting the template before the application of the above (or any) algorithm. Especially in the case of the algorithm under examination, however, the computation may alternatively be performed based on the correlation theorem and the equation **(1.46)**:

$$x[n] \star t[n] \overset{DTFT}{\Longleftrightarrow} X(e^{j\omega})T^*(e^{j\omega}) \tag{2.4}$$

From this equation and the definition of DFT stems the fact that for the computation of correlation instead of convolution the two following differentiations must be made on the basic algorithm:

- The point-by-point multiplication is performed between the DFT of $x[\boldsymbol{n}]$ ($X[\boldsymbol{k}]$) and the complex conjugate of the DFT of $t[\boldsymbol{n}]$ ($T^*[\boldsymbol{k}]$).
- The final result of the algorithm must be cyclically shifted by $X_i - 1$ elements towards the positive direction in every dimension i, where $i = 1, 2, \dots, \nu$. Cyclical shift means that the values that “exit” out of bounds return from the opposite end.

We should note that the conversion to complex conjugate may be incorporated in the process of the conjugate multiplication so that it does not have any impact on the total number of operations of the basic algorithm. On the other hand, a cyclical shift has infinitesimal cost as an operation as it just rearranges elements in memory.

2.1.3 Image Streaming

In practice it is very often the case in which a template is applied via the operation of convolution (or correlation) on a multitude of images of the same size. A typical example would be the process of video data, where the convolution of each frame of the video with the same template is very frequently required. A series of images of the same size to be processed will be called an “image stream” and their processing with the same template will be called “streaming”.

By taking into account the frequency of streaming in practice, we gain new possibilities for improving the methods for image processing via convolution/correlation. Next we examine how and how much the algorithms for computing convolution that have been presented above may achieve better results when applied to a stream of images instead of individual images.

We begin by examining the direct method. When processing a stream, convolution must be computed using the method for each image separately, as if they were individual images. We therefore notice that that the direct method does not appear to improve in the case of streaming.

Concerning the Fourier domain method, the case is different. We notice that the FFT of the template ($H[\boldsymbol{k}]$) has to be calculated only once for the whole stream. Thence, for any new image:

1. the FFT of the image must be computed
2. the point-by-point multiplication between the two FFTs must be performed
3. the inverse FFT of the product must be calculated

In Figure 2.3 the flow chart of the Fourier domain method is depicted anew, this time having colored the part that needs to be calculated only once for the whole stream.

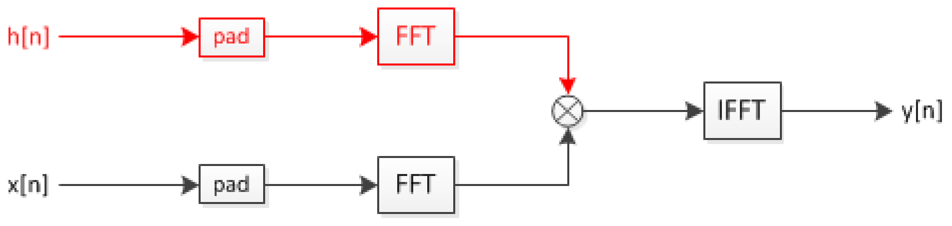

Figure 2.3: Fourier Domain Method for the Computation of Convolution with Streaming

We therefore notice that for each new image there is a need for the computation of two "half" FFTs and one multiplication. Assuming the number n of the images is $n \gg 1$, the number of operations of the algorithm per image is reformed to:

$$2(2.5N_y \log N_y) + N_y = 5N_y \log N_y + N_y \cong 5N_y \log N_y \qquad \textbf{(2.5)}$$

The above constitutes a significant improvement relative to the $7.5N_y \log N_y$ operations that would be needed were the algorithm be applied individually per image. With this method we expect an improvement of the total processing time of the stream by

$$\frac{7.5nN_y \log N_y - 5nN_y \log N_y}{7.5nN_y \log N_y} = \frac{2.5}{7.5} \cong 33.3\% \qquad \textbf{(2.6)}$$

2.1.4 Method Comparison

We will proceed to the comparative analysis of the two methods for computing convolution that have been presented in the above subsections. We remind the reader that the total number of operations that are required for each one is:

• Direct method:	$N_y N_h$
• Fourier domain method:	$7.5 N_y \log N_y$
• Fourier method with streaming:	$5 N_y \log N_y$

In the direct method we take into account the number of multiplications, assuming the total execution is determined mainly by them. For our analysis we will assume that $N_h \ll N_x$, something that is frequently true in practice. Therefore we have $N_y \cong N_x$.

From the above we notice that the execution time of the direct method presents a significant dependency on the size of the template N_h. For small templates we expect a small execution time which will increase significantly though as the template size becomes larger.

On the contrary, the Fourier domain method presents very small dependence on the size of the template. Its execution time is determined mainly by the size of the image N_x. We expect that the execution time will be increasing infinitesimally to none at all relative to the increase of template size.

The above analysis indicates that there is a critical template size N_h^{crit} that marks the "preference bound" of one method against the other. For a given image size N_x, this size theoretically results as follows:

$$N_y N_h^{crit} = 7.5 N_y \log N_y \Rightarrow N_h^{crit} = 7.5 \log N_y \xRightarrow{N_y \cong N_x} N_h^{crit} = 7.5 \log N_x \tag{2.7}$$

Respectively, if on the above analysis we assume image streaming, the critical size results in $N_h^{crit} = 5 \log N_x$. As a final conclusion, we may state that when $N_h < N_h^{crit}$ the direct method is deemed preferable whilst if $N_h > N_h^{crit}$ the Fourier domain method is chosen.

2.2 Correlation Coefficients with Local Normalization

2.2.1 Direct Method

In the present subsection we will present the direct method for the computation of the table of correlation coefficients with local normalization between two-dimensional or three-dimensional images. We remind the reader that each element of this table is described by the equation **(1.26)**:

$$r(P,T) = \frac{1}{N_T} \frac{\sum_{x,y} \left(P_{x,y} - \mu(P)\right)\left(T_{x,y} - \mu(T)\right)}{\sigma(P)\sigma(T)} \tag{2.8}$$

where T corresponds to the template and P is a panel of the image of the same size as T. We additionally have:

- $\mu(P) = \frac{1}{N_T} \sum_{x,y} P_{x,y}$ the mean value of panel P
- $\sigma(P) = \sqrt{\frac{1}{N_T} \sum_{x,y} \left(P_{x,y} - \mu(P)\right)^2}$ the standard deviation of panel P

Likewise the mean value and standard deviation of the template T are defined, $\mu(T)$ and $\sigma(T)$ respectively.

We rewrite equation **(2.8)** in the form:

$$r(P,T) = \sum_{x,y} \left(\frac{P_{x,y} - \mu(P)}{\sqrt{N_T}\sigma(P)}\right)\left(\frac{T_{x,y} - \mu(T)}{\sqrt{N_T}\sigma(T)}\right) = \sum_{x,y} \hat{P}_{x,y}\hat{T}_{x,y} \qquad \textbf{(2.9)}$$

We notice that the individual correlation coefficient equals to the inner product between the signals $\hat{P}$ and $\hat{T}$, which result by normalizing P and T respectively to a mean value of 0 and a standard deviation of $1/\sqrt{N_T}$, where N_T is the number of their elements. This formula proves that the computation of the table of correlation coefficients may be performed by variating the direct method for computing correlation, i.e. the one presented in subsection 2.1.1. The variation consists of the normalization each time of P and T before the extraction of their inner product. From a computational viewpoint, the normalization of the template T does not pose a problem, as it may be performed only once in the beginning and thus the normalized template $\hat{T}$ can be used for the whole image. However the same does not occur regarding the normalization of the panels P, as this must be performed separately for each individual panel of the image. Given the large number of these panels, one may understand that the immediate use of the direct method as is presents prohibitively high computational complexity. Specifically, with this method there would be a need for three “passes” of each panel P: the first to calculate the mean value $\mu(P)$, the second to calculate the standard deviation $\sigma(P)$ and the third to normalize it and calculate its inner product with the template $\hat{T}$.

We will attempt to improve the direct method for the computation of correlation coefficients with local normalization via an algebraic reformulation of the definition equation. By assuming that the template T is normalized once in the beginning, we rewrite the equation **(2.9)** as it follows:

$$r(P,T) = \sum_{x,y} \left(\frac{P_{x,y} - \mu(P)}{\sqrt{N_T}\sigma(P)}\right)\hat{T}_{x,y} = \sum_{x,y} \frac{\left(P_{x,y} - \mu(P)\right)\hat{T}_{x,y}}{\sqrt{N_T} \cdot \sqrt{\frac{1}{N_T}\sum_{x,y}\left(P_{x,y} - \mu(P)\right)^2}} =$$
$$= \frac{\sum_{x,y} P_{x,y}\hat{T}_{x,y} - \mu(P)\sum_{x,y}\hat{T}_{x,y}}{\sqrt{\sum_{x,y}\left(P_{x,y} - \mu(P)\right)^2}} = \frac{\sum_{x,y} P_{x,y}\hat{T}_{x,y} - N_T\mu(P)\mu(\hat{T})}{\sqrt{\sum_{x,y}\left(P_{x,y} - \mu(P)\right)^2}} \qquad \textbf{(2.10)}$$

We already know that $\mu(\hat{T}) = 0$. Therefore, the correlation coefficient is given by the following formula:

$$r(P,T) = \frac{\sum_{x,y} P_{x,y} \hat{T}_{x,y}}{\sqrt{\sum_{x,y} \left(P_{x,y} - \mu(P)\right)^2}} \tag{2.11}$$

We notice that the nominator of equation **(2.11)** corresponds to the inner product between P and $\hat{T}$, without previous normalization of P. Until now we have managed to "transpose" the order of the operations of correlation and normalization, by having the latter performed after the former. As of now two "passes" of the panel P are required: the first calculates the mean value $\mu(P)$ and the inner product with the template while the second calculates the standard deviation $\sigma(P)$. Continuing our effort for optimization, we get the radicand of the denominator of the right part of the equation **(2.11)** and we rewrite it in the following way:

$$\begin{aligned}\sum_{x,y} \left(P_{x,y} - \mu(P)\right)^2 &= \sum_{x,y} \left(P_{x,y}^2 - 2\mu(P)P_{x,y} + \mu^2(P)\right) = \\ &= \sum_{x,y} P_{x,y}^2 - 2\mu(P) \sum_{x,y} P_{x,y} + \sum_{x,y} \mu^2(P) = \\ &= \sum_{x,y} P_{x,y}^2 - 2N_T\mu^2(P) + N_T\mu^2(P) = \sum_{x,y} P_{x,y}^2 - N_T\mu^2(P) = \\ &= \sum_{x,y} P_{x,y}^2 - N_T \left(\frac{1}{N_T} \sum_{x,y} P_{x,y}\right)^2 = \sum_{x,y} P_{x,y}^2 - \frac{1}{N_T} \left(\sum_{x,y} P_{x,y}\right)^2\end{aligned} \tag{2.12}$$

Therefore, the initial definition formula of the correlation coefficient with local normalization results in the following equal:

$$r(P,T) = \frac{\sum_{x,y} P_{x,y} \hat{T}_{x,y}}{\sqrt{\sum_{x,y} P_{x,y}^2 - \frac{1}{N_T} \left(\sum_{x,y} P_{x,y}\right)^2}} \tag{2.13}$$

As of now, all the elements that are required for the computation of $r(P,T)$ may be computed using the equation **(2.13)** and by performing a single pass of the panel P. Consequently, we are in a position to be able to present in its completeness the optimized direct method for the computation of the table of correlation coefficients with local normalization:

ALGORITHM: Direct Method for LCCs

1. We normalize the template T once to mean value of 0 and standard deviation of $1/\sqrt{N_T}$.
2. We pad the image with zeros on each "side" along every dimension, in the way described in step **1** of the direct method for convolution.
3. For each individual panel P of the image we calculate in one and only pass:
    - the inner product with the normalized template $\hat{T}$
    - the sum of all the elements of P
    - the sum of the squares of all the elements of P
4. We compute the correlation coefficient $r(P,T)$ from the equation **(2.13)**. If the denominator equals to 0, we set $r(P,T)=0$ (see subsection 1.4.2).
5. We repeat steps **3**, **4** for all the panels of the image.

We proceed to the analysis of the computational complexity of the above algorithm. Regarding the size of the table of correlation coefficients the same apply as with convolution/correlation: if X_S, Y_S are the dimensions of image S and X_T, Y_T are the dimensions of the template T, the size of the table $\{r\}$ is:

$$N_r = (X_S + X_T - 1)(Y_S + Y_T - 1) \tag{2.14}$$

For each element of the table $\{r\}$ the following are required:

- N_T multiplications and $N_T - 1$ additions for the calculation of the inner product between P and $\hat{T}$
- $N_T - 1$ additions for the calculation of the sum of the elements of P
- N_T multiplications and $N_T - 1$ additions for the calculation of the sum of the squares of the elements of P

We assume that the implementation of the formula for the computation itself of the coefficient r in the end of each "pass" is negligible related to the operations required during the "pass" itself. We furthermore assume that the template T is small (something that often occurs in practice) and so its normalization is negligible related to the main computation. So, according to the above, for the complete computation of the table of correlation coefficients with local normalization the following are required:

• $2N_rN_T$	multiplications
• $3N_r(N_T - 1)$	additions

Therefore, the total computational complexity of the algorithm is of order $O(N_rN_T)$. We remind the reader that the direct method for computing convolution/correlation is of the same rank but with smaller constant factors.

Something noteworthy is that the computation of each element of the table $\{r\}$ is independent of the rest, something that means that the algorithm may be parallelized on element level, rendering its implementation on a vector processor suitable. Additionally, the algorithm presents further advantages, similar to those of the direct method for computing convolution/correlation. Those advantages are the high spatial locality (neighboring output elements use overlapping input data) and the good behavior for small templates T. The reader may refer to the subsection 2.1.1 for the further analysis of these advantages.

2.2.2 Fourier Domain Method

In subsection 2.1.2 we described an alternative method for computing convolution/correlation, the better part of which computations is performed in the Fourier domain. In this subsection we will attempt to develop a similar method for the computation of correlation coefficients with local normalization.

As has been thoroughly described in subsection 1.4.2, each individual panel P of the image that corresponds to the position (u, v) may be mathematically represented in the following way:

$$P(x, y; u, v) = S(x, y)B_T(x - u, y - v) \tag{2.15}$$

where S is the full image and B_T is the binary-valued characteristic function of the spatial support of the template. By substituting the above relation in the equation **(2.13)**, we get the description of the complete table of correlation coefficients $\{r(u, v)\}$ between the template T and the image S as follows:

$$r(u, v) = \frac{\sum_{x,y} S(x, y)\hat{T}(x - u, y - v)}{\sqrt{\sum_{x,y} S^2(x, y)B_T(x - u, y - v) - \frac{1}{N_T}\left(\sum_{x,y} S(x, y)B_T(x - u, y - v)\right)^2}} \tag{2.16}$$

By meticulously observing the equation **(2.16)**, we ascertain that:

$$\sum_{x,y} S(x, y)\hat{T}(x - u, y - v) = S \star \hat{T} \tag{2.17}$$

$$\sum_{x,y} S^2(x, y)B_T(x - u, y - v) = S^2 \star B_T \tag{2.18}$$

$$\sum_{x,y} S(x, y)B_T(x - u, y - v) = S \star B_T \tag{2.19}$$

This means that the three individual sums, when expanded to the full image, may be described in the form of correlations. So, by substituting the equations **(2.17)**, **(2.18)** and **(2.19)** in equation **(2.16)**, the table of correlation coefficients may alternatively be written as follows:

$$r(u, v) = \frac{S \star \hat{T}}{\sqrt{(S^2 \star B_T) - \frac{1}{N_T}(S \star B_T)^2}} \tag{2.20}$$

The meaning of equation (2.20) lies in the fact that it lends us the opportunity to "transfer" the computation in the Fourier domain. Indeed, we have seen that correlation is equal to convolution where one of the two signals is inverted along every dimension. So, by applying the convolution theorem we have:

$$S \star \hat{T} = \mathcal{F}^{-1}\left\{\mathcal{F}\{S\} \cdot \mathcal{F}\{\hat{T}\}\right\} \tag{2.21}$$

$$S^2 \star B_T = \mathcal{F}^{-1}\{\mathcal{F}\{S^2\} \cdot \mathcal{F}\{B_T\}\} \tag{2.22}$$

$$S \star B_T = \mathcal{F}^{-1}\{\mathcal{F}\{S\} \cdot \mathcal{F}\{B_T\}\} \tag{2.23}$$

Assuming we have inverted the signals $\hat{T}$ and B_T before their transform. Of course, as the signal B_T is filled with units, its inversion would equal to the signal itself. So the only signal that really needs to be inverted is $\hat{T}$. We notice therefore that by applying equations (2.21), (2.22) and (2.23) the performance of the better part of the computations described by equation (2.20) in the Fourier domain and not in the spatial domain is rendered possible.

At this point we have achieved in developing an algorithmic method for the computation of the table of correlation coefficients with local normalization that has similar features as the Fourier domain method for the computation of convolution that has been described and analyzed in subsection 2.1.2. Therefore, based on the analysis until now, we are at a position to fully describe the Fourier domain method for the computation of the table of correlation coefficients with local normalization. The individual steps of this algorithm are the following:

ALGORITHM: Fourier Domain Method for LCCs

1. We normalize the template T to mean value of 0 and standard deviation of $1/\sqrt{N_T}$ and then we invert it along every dimension.
2. We form the table B_T, which will be of the same size as the template and have unit value in every position.
3. We pad the tables T and B_T with zeros on their end so as their total size is equal to the result's.
4. We perform the FFTs of T and B_T, those being $\mathcal{F}\{T\}$ and $\mathcal{F}\{B_T\}$ respectively.
5. From the table S (image) we form the table S^2. We pad both of them on their ends with zeros so that their sizes are equal to the result's.
6. We perform the FFTs of S and S^2, those being $\mathcal{F}\{S\}$ and $\mathcal{F}\{S^2\}$ respectively.
7. We calculate the point-by-point products $\mathcal{F}\{S\} \cdot \mathcal{F}\{\hat{T}\}$, $\mathcal{F}\{S^2\} \cdot \mathcal{F}\{B_T\}$ and $\mathcal{F}\{S\} \cdot \mathcal{F}\{B_T\}$.
8. We perform the inverse FFTs of the three above products.
9. We compute the final result using the equation (2.20). If the denominator equals to zero, the correlation coefficient is also set to zero value (see subsection 1.4.2).

This algorithm is also presented in the form of a flow chart in Figure 2.4.

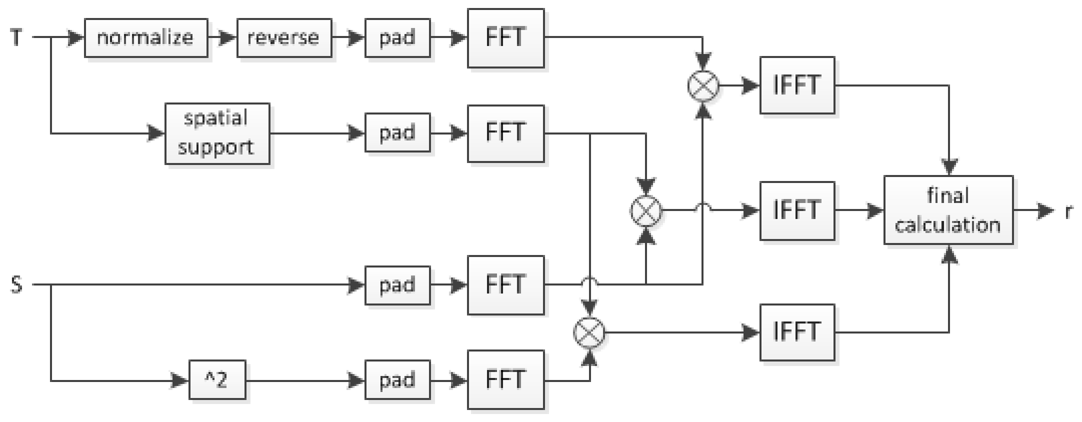

Figure 2.4: Fourier Domain Method for Computing LCCs

We will proceed to the analysis of the computational complexity of the above algorithm. We denote by N_r the size of the table of correlation coefficients, which at the same time equals to the size of all the individual tables that take part in these computations. We notice that, totally, 7 FFTs are needed, 4 direct and 3 inverse. By taking into account the fact that the direct FFTs are applied to real-valued signals and that the inverse FFTs result in also real-valued signals, we notice that all of the FFTs can be implemented as "half", as has been stated in subsection $1.5.3$, with each one requiring not more than $2.5N_r \log N_r$ operations. Furthermore, the algorithm incorporates the calculation of three point-by-point complex multiplications (step **7**), squaring (step **5**) and the final computation of the output table (step **9**). The final operations that have been referred to all have linear complexity regarding size N_r. Finally, assuming the template T is sufficiently small in size (as it typically occurs so in practice), we will assume that its normalization and inversion (step **1**) is of negligible time. Based on the above, the total number of operations of the algorithm results in:

$$7(2.5N_r \log N_r) + \alpha N_r = 17.5N_r \log N_r + \alpha N_r \cong 17.5N_r \log N_r \qquad \textbf{(2.24)}$$

which means that the total complexity is of order $O(N_r \log N_r)$. The constant factor α of the above relation is due to the linear complexity operations and its value may be considered sufficiently small such that the linear factor may be omitted.

As for the parallelization of the algorithm, the same applies as with the Fourier domain method for the computation of convolution/correlation. The linear complexity operations (complex multiplication, power of two, division, square root extraction, etc.) can all be parallelized on element level. The FFTs on the other hand are more "difficult" in their parallelization, but as we have already stated this matter is out of the bounds of this dissertation.

2.2.3 Image Streaming

Next we will analyze the performance of the two algorithms for computing correlation coefficients with local normalization that have been described in the above subsections in the case of image streaming. We remind the reader that image streaming is the case of a series of images of the same size (for example video) being processed by the same template. This case is rather frequent in practice; therefore it is particularly interesting to examine how much the aforementioned algorithms can be rightly adapted so as to achieve better overall performance. In the following we examine the two algorithms separately.

We begin with the direct method. We notice that the normalization of the template (step **1**) can be performed only once for the whole stream. Thence, the algorithm must be fully applied on each image. However, as the template typically is sufficiently small, its normalization comprises a process that costs very little in time. Therefore we notice that the direct method is not gaining any significant advantage from streaming in contrast to the case where it would be applied as is on every image of the stream separately.

Regarding the Fourier domain method, things are more interesting. We notice that the normalization and the inversion of the template, the formation of the table B_T and the computation of the FFTs of the template and the B_T can be performed only once for the whole stream. The above operations correspond to the steps **1** to **4** of the algorithm. Thence, for each image there is the need to perform anew only the steps **5** to **9**. This is presented schematically in Figure 2.5, where the flow chart of the Fourier domain method is presented anew, this time having colored the part of the algorithm that only needs to be computed only once for the whole stream.

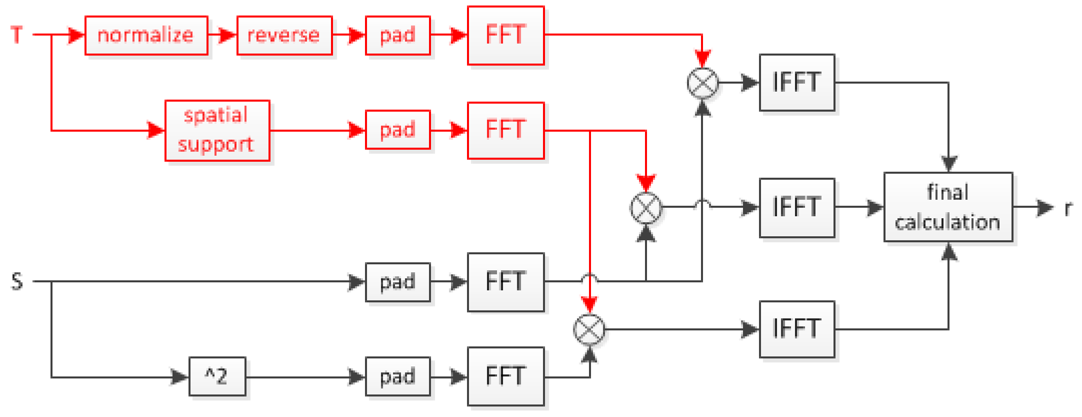

Figure 2.5: Fourier Domain Method for Computing LCCs with Streaming

Although the operations of normalizing and inverting the template are negligible, the saving of computing 2 of the 7 FFTs totally is particularly significant, specifically of $\mathcal{F}\{T\}$ and $\mathcal{F}\{B_T\}$. So, assuming the number n of the images is $n \gg 1$, the number of operations of the algorithm per image is reformulated to:

$$(7-2)(2.5N_r \log N_r) + \alpha N_r = 12.5N_r \log N_r + \alpha N_r \cong 12.5N_r \log N_r \qquad \textbf{(2.25)}$$

Where the constant factor α corresponds to the linear complexity operations regarding N_r and is sufficiently small such that the linear factor may be omitted. The above consists a significant improvement against the $17.5N_r \log N_r$ operations that would be required were the algorithm be applied separately on each image. With this method we expect an improvement of the total processing time by

$$\frac{17.5nN_r \log N_r - 12.5nN_r \log N_r}{17.5nN_r \log N_r} = \frac{5}{17.5} \cong 28.6\% \qquad \textbf{(2.26)}$$

2.2.4 Method Comparison

In this subsection we will compare the methods that have been presented until now for the computation of correlation coefficients with local normalization regarding their total number of required operations. We remind the reader that the number of operations required from each method equals to the following:

• Direct method:	$2N_rN_T$
• Fourier domain method:	$17.5N_r \log N_r$
• Fourier method with streaming:	$12.5N_r \log N_r$

In the direct method we only take into account the number of multiplications, assuming the total execution time is defined mainly by them. We will further assume that $N_T \ll N_r$ (as it typically occurs in practice) and thus $N_r \cong N_S$, where N_S is the size of the image.

It is noteworthy that the above computational complexities are, regarding their form, similar to the complexities of the respective algorithms for computing convolution/correlation. Their differentiations lie specifically on their constant factors, which are presently larger (due to the need, of course, for local normalization). So, the further analysis and comparison of the methods adheres to the same concept as with the analysis in subsection 2.1.4 regarding convolution/correlation.

We begin with the analysis of the direct method. We notice that its complexity is dependent in a proportionate way on the size of the template N_T. This means that for small templates the total number of operations will be small and the method will be temporally efficient. Instead, as the size of the template increases, the method will all the more be rendered unsuitable.

The complexity of the Fourier domain method on the other hand presents small dependency on the size of the template. Instead, the total number of operations is almost inclusively dependent on the size of the image N_S. We expect therefore the execution time of this method to be increasing least to none as the size of the template increases.

The conclusion of the above reasoning is similar to the conclusion we had stated regarding the computation of convolution/correlation. The existence of a critical size of template N_T^{crit} that signifies the "preference bound" between the two methods is become evident. For a given image size of N_S, this critical size theoretically results as following:

$$2N_r N_T^{crit} = 17.5N_r \log N_r \Rightarrow N_T^{crit} = 8.75 \log N_r \overset{N_r \cong N_S}{\Longrightarrow} N_T^{crit} = 8.75 \log N_S \qquad \textbf{(2.27)}$$

Likewise, if in the above analysis we assume image streaming, the critical size results in $N_T^{crit} = 6.25 \log N_S$. As a final conclusion, we can state that when $N_T < N_T^{crit}$ the direct method is deemed suitable while if $N_T > N_T^{crit}$ the Fourier domain method is chosen.

3 Parallel Architectures

The continuous demand for increasingly powerful computational systems gave birth, among others, to the idea of using many Central Processing Units (or CPUs) that operate in parallel. Of course, a lot of suggestions have been made over the years on how this quite generic idea could be realized. There exist issues such as whether we consider these CPUs as separate computers accompanied by separate input/output units each, as several CPU chips inside a single computer that share its various resources, or even as a single control unit and a program counter that direct many Arithmetic and Logic Units (ALUs) in such a way that they perform the same operation on large sets of data simultaneously. After all this, basic issues are the nature of the CPUs and their privileges over various resources (memory, input/output units, etc.), their number and their mode of intercommunication. There exist quite a few different parallel architectures, that is implementations of each of the aforementioned perceptions, with a variety of capabilities, which nevertheless does not make one better than the rest but rather more appropriate for certain kinds of problems.

Given their substantially increased computational power and their continuous expansion, parallel architectures are the most suitable for a high-performance library. Therefore, among the wide spectrum of today's available modern parallel architectures, we attempt to select those that will best support the development of the FLCC library and the implementation of its algorithms. The two basic computational platforms that we have concluded on are the **multi-core processors** and the **graphics processing units**. The criteria that led to this selection are their power, their suitability for our calculations and their widespread usage.

In this chapter we thoroughly describe the two architectures of parallel computation that we make use of. We will talk about their characteristics, their capabilities and their programming model. In particular, in section 3.1 we study multi-core processors while in section 3.2 we move our attention to graphics processing units.

3.1 Multi-core Processors

3.1.1 Hardware Structure

Multiprocessors are parallel systems which are composed of two or more CPUs which share a common physical memory of RAM type. The communication among the CPUs is mainly done through this memory, which is hence called shared memory. We say "mainly" because, even though there exists a common physical memory, communication among CPUs may also be done and/or with message passing. Communication through shared memory is done as follows: all CPUs, or to be more precise the processes which run at a given moment on the CPUs, have equal access to the shared memory and may read or write data to it. In a few words, all CPUs share a common address space. It is noteworthy that multiprocessors are quite popular and widespread exactly thanks to the simplicity of the CPUs intercommunication.

In the above there exist various issues of data consistency among CPUs. For instance, two CPUs may try to write at the same time in the same memory location. Also, each CPU could have a cache memory of its own. That means that the set of data which is stored in the cache has to be updated according to the changes in the data in the main memory. The above may lead to problems if the mode in which the multiprocessor deals with such processes is not well predicted. We will not proceed though in such analysis since it is out of the scope of this thesis.

A multiprocessor has access to many modules and levels of memory, as well as to input/output devices. In case where all CPUs are equivalent, they equally have access to each available memory level (with the possible exception of each one having a dedicated cache memory) and input/output device and are also viewed as equal by the operating system, the system is called a symmetric multiprocessor or SMP. In Figure 3.1 the structure diagram of a simple SMP with 8 similar processing units is presented. In the same figure we can also observe the internal architecture of a single CPU, which is composed by: the control unit, responsible for the control flow of a program, the arithmetic and logic unit, responsible for the execution of arithmetic and logic operations, and the registers which are small and fast memory modules in which the data to be processed are fetched.

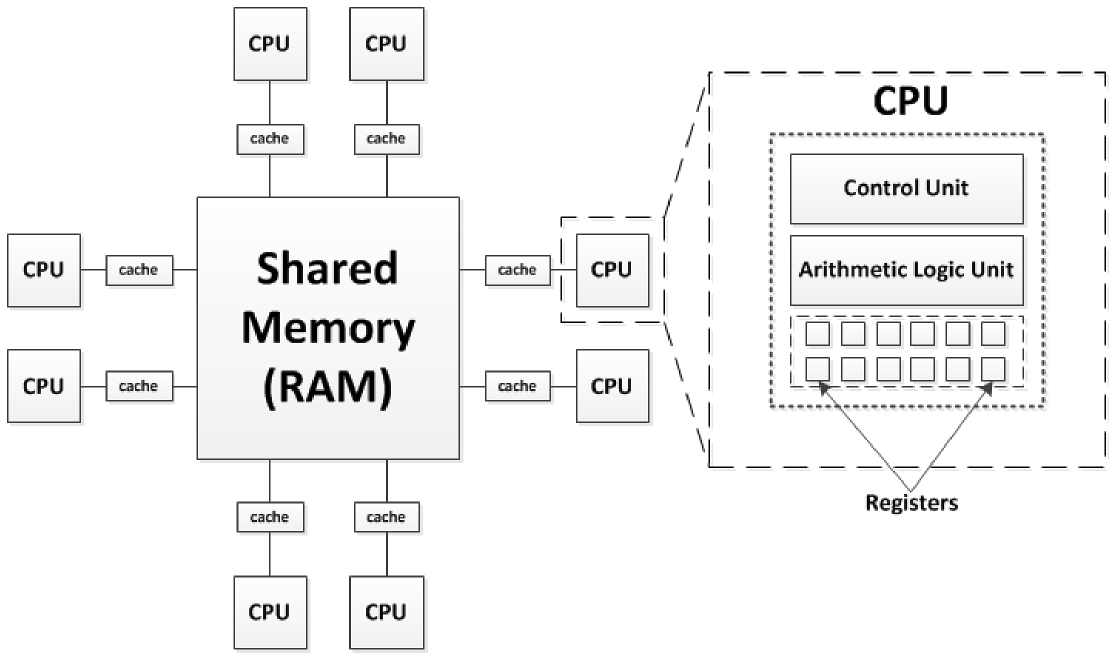

Figure 3.1: Structure Diagram of a Simple SMP

According to Flynn's taxonomy for parallel computers, multiprocessors are described as Multiple Instruction Multiple Data or MIMD systems. Flynn's taxonomy is derived by the examination of two notions: the instruction streams (program counters) and the data streams. A MIMD system has more than one instruction streams and more than one data streams. This means that each CPU is capable of simultaneously executing a totally different process on a totally different data set.

A **multi-core processor** is a single chip in which more than one full CPU has been incorporated. In this case the CPUs are called cores. Very often the cores do not share cache memories but they do share the same main memory. We may hence characterize them as multiprocessors. To be precise, multi-core processors are often called Chip-Level Multiprocessors or CMPs. Also, the majority of today's marketed multi-core chips are SMPs. It is important to note that multi-core processors are today notably widespread even in Personal Computer (PC) environments and their growth continues at a high pace.

We have talked about the various memory levels of multiprocessors, which do not differ than those of a classic non-parallel processor, since the notions of RAM, cache memory and registers are present in both architectures. What differs is the need for effective modes of communication among the CPUs and for data consistency. As it has been mentioned, communication is mainly preformed through RAM, which is common for all the CPUs. On the contrary, each CPU has its own cache memory and its own registers. In Figure 3.2 we see the memory hierarchy for multi-core processors and a qualitative presentation of the differences in speed and size among different memory levels.

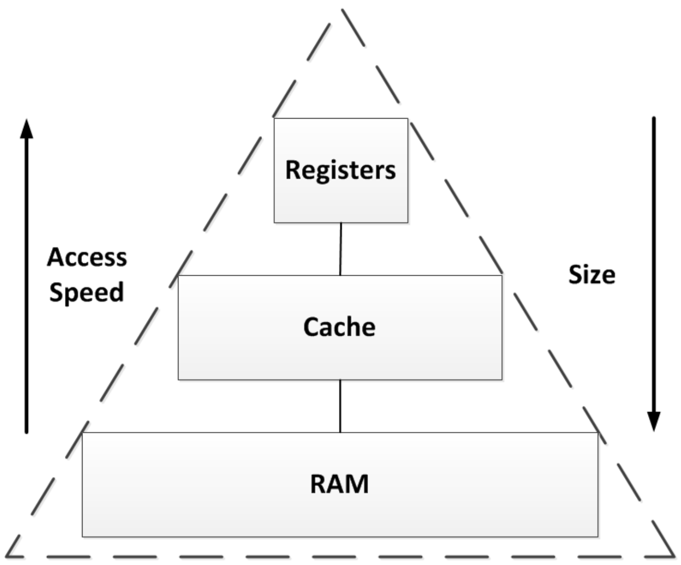

Figure 3.2: Memory Hierarchy of a Multi-core Processor

We have already discussed in chapter 2 the theoretical reduction in the execution time of the algorithms if they are meant to run in parallel. Thus, according to the above, multi-core processors are a powerful choice as implementation architecture for these algorithms, since they offer the computational power of an efficient parallel architecture, they demonstrate easy communication among their cores and they are suitable to be used for the solution of many different problems. Also, they are quite widespread, which makes them available to a vast user group and of course it makes a variety of other tools that have been developed for them available too. Of course the sole existence of the hardware without the appropriate software taking advantage of its capabilities is not enough, as we will see in the following section.

3.1.2 Multi-threading

3.1.2.1 Processes

We call a **sequential process** or just process an instance of a running program, which contains the current values of the program counter and the program variables. In other words, a process is the combination of a control flow or thread and a private address space which is not visible to other processes.

The address space is the entire set of addresses which can be used by a process to perform a certain transaction with memory. As a result, the address space of a process may correspond to main memory size much larger than the existing one. This is overcome with memory paging. The address space is further divided into certain main sections that each contains addresses for diverse data types. For a UNIX/Linux system these are as follows:

- The *stack section*, which contains the stack being used by the main control flow of the process. The local variables of the program functions are located in the stack, with those belonging to the current function being on top of the stack.
- The *text section*, which contains the addresses for the code of the program in execution.
- The *data section*, which contains the already initialized global variables of the program in execution.
- The *heap section*, which contains the heap being used by the main control flow of the process and in which memory mappings of files or variables of dynamic length take place.

Furthermore, the process also maintains a private program counter and a stack pointer. In the upper part of Figure 3.4 we can see a schematic diagram of an address space of a process with one control flow.

Every time the user requests that a program be executed, the operating system creates a process which contains in its address space all the necessary information for the program's successful execution. We referred to the process as an instance of some program and this is evident by the fact that if the same program is called multiple times, a distinct process will be created for every call.

A process, at any time, is being served by a virtual CPU. We say virtual because usually in a computer there is a far greater number of processes being executed at any time than the number of actual existing CPUs. The various processes are being served in turns by one CPU according to a certain scheduling algorithm and thus it seems that all are being executed in parallel while in reality each CPU serves only one process at a given moment (this is most evident when there is not but one CPU). Of course if there exist in fact more than one CPUs, an actual hardware parallelism takes place in the processes execution, simultaneously with the pseudo-parallelism we previously described.

This rotation of the processes being served by the CPUs is called multiprogramming. In each CPU there exists a single actual program counter, whilst at the same time each process maintains its own program counter and therefore knows at what point of its execution it is. When the time comes for a certain process to be served by some CPU, its program counter is loaded in the actual program counter. When, later on, another process's turn comes to keep that CPU busy, the old process is responsible for the storing of its own program counter.

The reason why multiprogramming is useful is that it maximizes the time percentage that the CPU is being used. For instance, if a process needs to wait for the completion of some input/output operation it would not be particularly effective to keep all the others waiting too. So, some other process takes the place of the waiting one and makes use of the now available CPU. This technique offers a lot to the maximization of CPU usage because usually input/output operations take much longer than a single instruction cycle.

3.1.2.2 Threads

As it has been mentioned, processes are defined as instances of a certain program and have one **control thread** and their own address space. Sometimes it is useful or even necessary to have more than one threads which are able to be executed in parallel in the address space of the same process. A basic reason, analogous to the reason for the pseudo-parallel execution of the processes, is maximizing the CPU usage. Furthermore, a thread can be created and destroyed much faster than a process. This happens because the necessary overhead for a thread's creation has already taken place when its mother-process was created. So, threads are considered lightweight as long as execution is concerned. The presence of multiple threads within a process is called multithreading. Figure 3.3 makes clear the possibility of having one or more independent processes simultaneously and the idea of multithreading.

The usefulness of threads becomes truly evident if we consider the possibility of their actual parallel execution by parallel hardware, namely a multi-core processor. This possibility can often be properly exploited in order to achieve high computational performance. Indeed, if we deal with a problem which can be broken down to smaller pieces – subproblems which in turn can be solved independently, the parallel solution by distinct threads running in parallel offers the total solution in only a fraction of the total execution time. It is true that the above idea could also be implemented with parallel processes instead. Nonetheless, given their substantially faster creation and destruction and their easier management, threads are in principle proven to be a more effective choice for high performance applications.

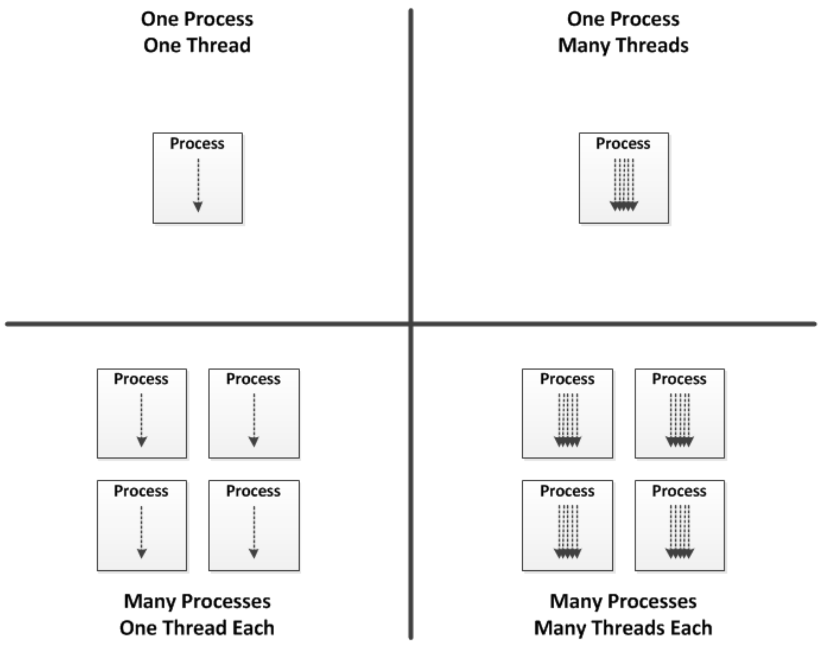

Figure 3.3: Combination of Processes and Threads of Different Numbers

Something that deserves to be mentioned here is that the various threads of a process are not independent since they share a common address space and hence the same global variables. It is the programmer's responsibility to create threads which cooperate with one another and of course manage their common data in a consistent and reliable way.

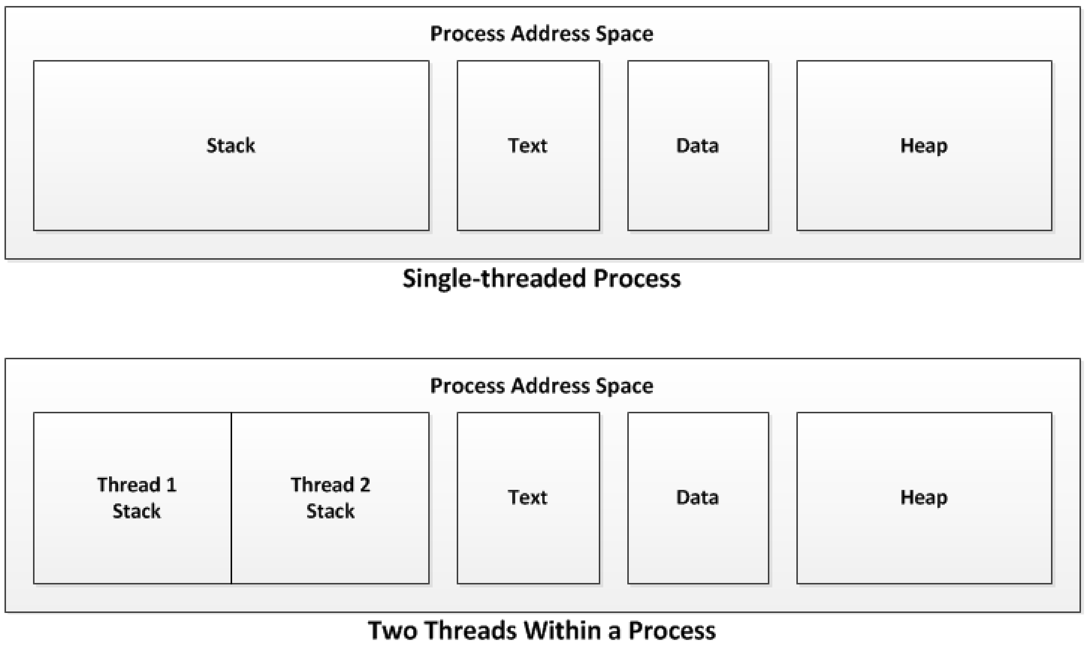

Figure 3.4: Address Space of a Process with One or More Threads

The difference in the address space of a multi-threaded process in comparison with a single-threaded one is that the stack section of the former is divided in as many subsections as the number of threads. Thus, each thread maintains its own private stack. Figure 3.4 makes the stack section's division clearer. Needless to mention is that each thread within the process also retains its private program counter and stack pointer.

3.1.2.3 POSIX Threads

The need to write portable programs with threads is a rather obvious one. For that reason the standard IEEE POSIX 1003.1c has been developed that enables the usage of threads in the C language and is supported by UNIX/Linux systems. Threads that follow this standard are known as **POSIX Threads** or pthreads; they are notably popular and are those that are used in the FLCC library.

Let's discuss some practical issues on pthreads usage. A program in C when executed is considered to be a process with one thread or control flow. Pthreads library provides the programmer with a function for the creation of a new thread. This function is **`pthread_create(…)`** and can be called several times in order to create as many threads as needed, one for every call. The threads are able to have a completely different control flow (or simply to execute different programs) and different access to data and finally they are scheduled according to the operating system. The specific function which every new thread will execute is determined by a pointer to it, which is passed as an argument to **`pthread_create(…)`**.

As for the data, each thread has access to all the global variables of the process to which it belongs, which in turn are declared in the initial control thread. Furthermore **`pthread_create(…)`** accepts a pointer as an argument which points to a unique data type to which the thread has access. With that in mind, the way to pass more than one pieces of data to a thread is to create a data structure containing them and set the aforementioned pointer to it. It is up to the programmer to pass as an argument to every thread an arithmetic identity. Here we see the MIMD type implementation, with the threads having the ability to run different programs on different data. Note that in case more than one threads share some common data, issues of competition arise among them. The programmer may avoid them, using mutual exclusion and condition flags which are already provided by the pthreads library.

Finally, there exist several ways of terminating and synchronizing the already created threads. A typical one would be the following: the main thread, after having created some other thread using **`pthread_create(…)`**, may continue with its own tasks and then wait for the new thread to finish its work. This waiting is achieved by calling the function **`pthread_join(…)`**, whose arguments indicate which thread to wait for. The new thread on the other side, after having finished its own work, may declare that it has finished and terminate by calling the function **`pthread_exit(…)`**. This way the two threads synchronize and the main thread can continue with its own control flow. Figure 3.5 shows an example of the creation of two new threads by a main one and later on their joining with it.

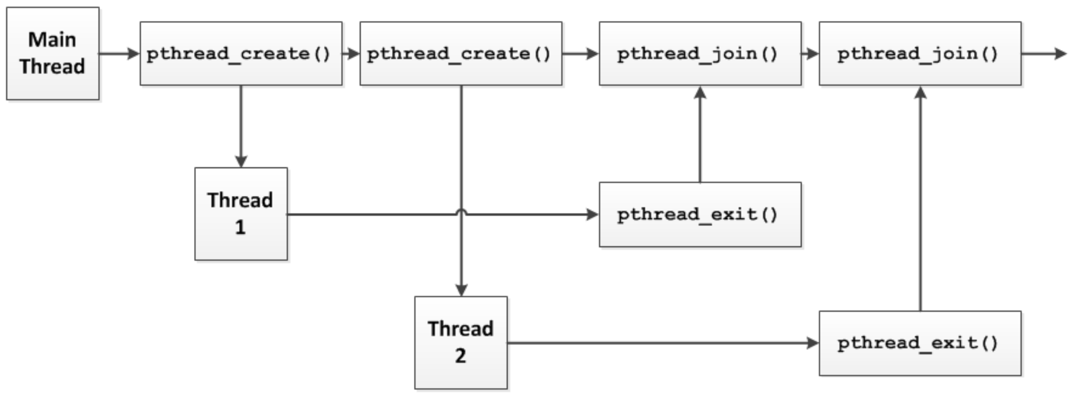

Figure 3.5: Creation and Synchronization of Pthreads

There exist more ways of thread termination, such as their canceling by the main thread, but we will not proceed to further analysis.

3.1.3 FFTW

FFTW (Fastest Fourier Transform in the West) is a library for the C programming language which offers fast computation of one-dimensional and multi-dimensional DFTs, was developed by Matteo Frigo and Steven Johnson of MIT and was first published in 1997. It uses a variety of different FFT algorithms in order to carry out computations, a basic one among them being Cooley – Tukey's FFT that has been already mentioned.

FFTW has a remarkable performance independently of the machine it runs on and there are three fundamental ideas behind this success. First, the transformation is computed by an *executor* composed of highly optimized code segments in C (which the creators call codelets). Secondly, a *planner* calculates a fairly efficient way (the so called plan) to combine the executor's codelets, adapting to the architecture on which it runs (the specific machine). Last but not least, the codelets are generated by a *codelet generator* which has been written in the language Objective Caml (an ML dialect) and takes into account the special characteristics of the machine it runs on to generate them. This way, FFTW is able to run effectively on a wide spectrum of diverse machines.

We will analyze FFTW's plan-execute model a bit more, in the form it appears from the user's perspective. The central idea is that the user interacts externally with FFTW only through planners and executors. FFTW provides the user with functions for the planning and execution of transforms of one or more dimensions. It should be noted that, internally, FFTW does not use only one specific algorithm to compute the DFTs. Instead, through the planner it adapts to the capabilities of the hardware in order to achieve maximum optimization and selects every time the most suitable algorithms for the calculation. Thus, the total process of transforming is divided in two sections. First, FFTW's planner finds the fastest way for the calculation of a transform of a certain size on the user's machine. The planner then creates a data structure, the so called plan, which stores the above information. Later on, the plan can be executed by the executor and the transform is calculated for specific data according to the information from the plan. The same plan can be used for the same size, but not necessarily for the same data, as many times as needed. It is evident that a great performance advantage is given to problems which require many different transforms of the same size, a case found quite often in practice, since the planning process can be executed only once and hence its time cost can become easily acceptable.

We use FFTW in FLCC internally in order to calculate the DFTs which participate in the Fourier domain methods. We selected FFTW because it is a highly optimized library, notably effective and it also offers the possibility of parallel execution via pthreads. Also, we adopted the plan-execute model in FLCC so the planning cost of FFTW is fully contained in the planning step of FLCC. More about the functionality of FLCC can be found in the following chapter.

3.2 Graphics Processing Units

3.2.1 Hardware Structure

Graphics Processing Units or GPUs are highly parallel, massively multithreaded, multi-core (also referred to as many-core) processors with a substantial computational power and large memory bandwidth. A GPU, due to its highly parallel character, is often much more efficient in comparison with a CPU regarding computations of high parallelizability. A usual example of such computations is graphics rendering, which historically has been GPU's main target application, hence the name "GPU". Nevertheless, in principle any application that can be expressed as highly parallel data calculations has the ability to be efficiently executed on a GPU (in other words, any program which is executed in parallel on a large number of data points simultaneously, with each data point being assigned to a parallel thread). Thus, given its relatively recent technologic development, GPU has moved past graphics rendering to become a general-purpose parallel processor, often being referred to as General Purpose GPU (GPGPU).

In subsection 3.1.1 we saw that multiprocessors, as viewed by Flynn's taxonomy of parallel architectures, belong to the MIMD category. GPUs on the other hand are classified in the second practically used parallel architecture, the so called SIMD or Single Instruction Multiple Data. This architecture may be described in a few words as follows: there exists a single control unit (a program counter) which sequentially executes the program's instructions, albeit the instructions operate on a multitude of elements at the same time. It is considered that computers following SIMD architecture are ideal for dealing with scientific problems of high computational load in which operations on data structures such as vectors or arrays are expected to appear often.

We focus on the GPU architecture that has been used in this thesis, the one that has recently been introduced by the company NVIDIA. A GPU of this architecture is implemented as an array of similar multithreaded multiprocessors, each one in turn being composed by a number of processing units (cores). These multiprocessors follow an architecture which NVIDIA's experts refer to as Single Instruction Multiple Thread, or simply SIMT. SIMT architecture is relatively similar to the well-known SIMD. The difference is that each thread running on the GPU is able of having branches, albeit with a possible performance cost. The overall result of this structure is a closed processing system with a substantially high number of cores and powerful capabilities for multithreading.

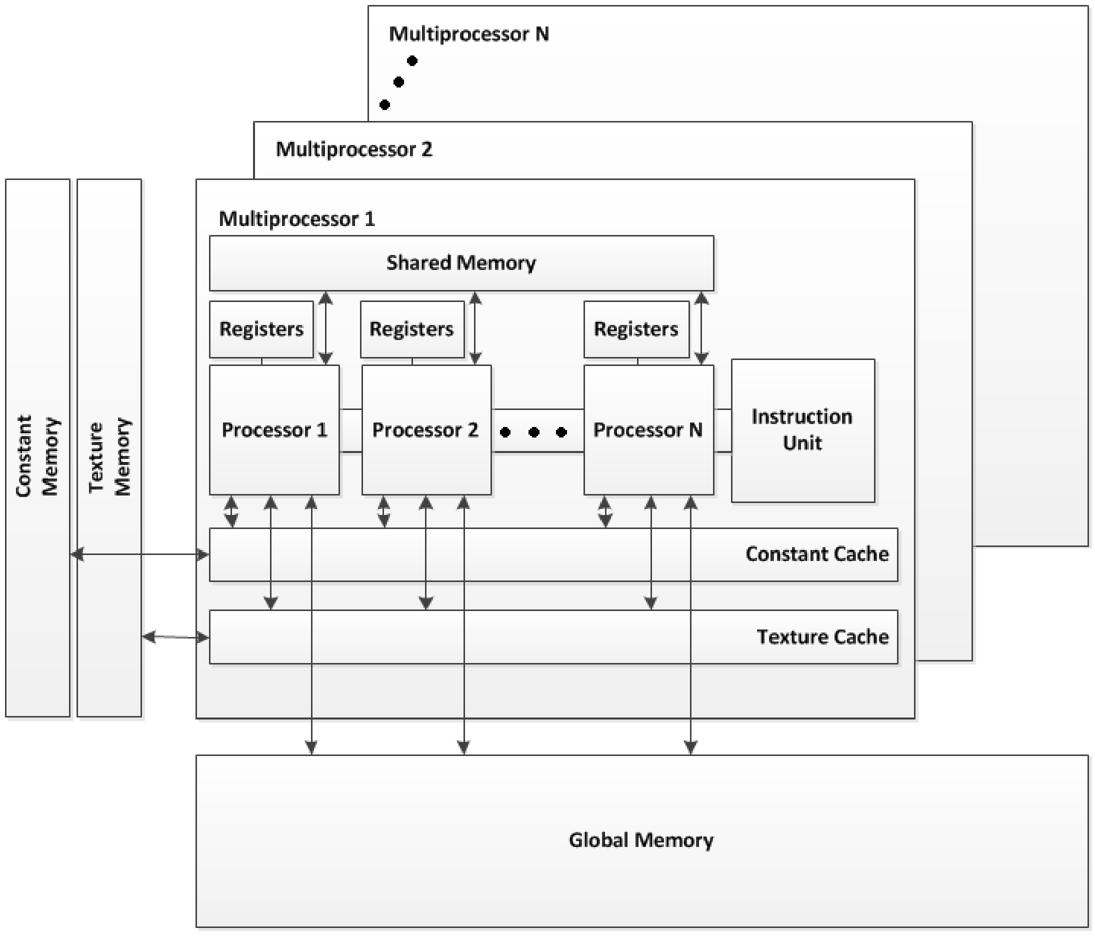

Figure 3.6: GPU Architecture of NVIDIA

We will add a few more words regarding the various memory modules of an NVIDIA's architecture GPU. Every processor within each multiprocessor has a private local memory in the form of registers. Every multiprocessor has its own shared memory which is on-chip, is being shared by all threads which are being executed at some point on the multiprocessor and is also considered to be a high-speed cache memory. There also exists a global memory that belongs to the entire device and is being shared by all running threads with no exceptions. This memory is in principle larger in size but with a slower access. Finally there also exist a constant memory and a texture memory, which are both located within the device, all threads have access to them and are again considered to be cached. Figure 3.6 shows the entire organization of NVIDIA's architecture GPU, where we can see the multiprocessors as well as the various memory modules.

Fast computation of convolution and LCCs requires the fast execution of operations on arrays (images) possibly of fairly large sizes. These operations, as it has been discussed in the algorithms' chapter, demonstrate high data parallelism up to pixel level. Based on what we have said, GPU offers the possibility to vastly parallelize these operations in order to maximize performance and therefore is the ideal hardware architecture to deal with the problems we are concerned with.

3.2.2 CUDA

CUDA (Compute Unified Device Architecture) is a general-purpose parallel programming architecture, product of NVIDIA, that makes use of the parallelism capabilities of the GPUs of the same company. CUDA provides a new parallel programming model for GPUs, accompanied by a new set of instructions, and attempts to solve complex computational problems with a much more efficient way compared to a single or multiple CPUs.

A CUDA's basic element is that it offers its user the possibility to create, with the usage of a new syntax, functions in C language, the so called kernels. A kernel, when called in a C program, is executed N times by N different CUDA threads (once for each thread) contrarily to the classic C functions that run just once when called by a control thread.

In the following, we will discuss four fundamental characteristics of the CUDA programming model, namely thread hierarchy, thread synchronization, memory hierarchy and heterogeneous programming.

Each CUDA thread has its own unique thread ID to which it has access through a built-in variable. This variable has the form of a three-dimensional vector which allows a set of threads to be logically arranged as a one-dimensional, two-dimensional or three-dimensional block of threads. The dimensionality of the block and the length of each dimension are defined by the user during each kernel's call. For instance, a thread that belongs to a three-dimensional block of, say, dimensions (D_x, D_y, D_z) may retrieve its coordinates (indices) (x, y, z) by the built-in variable and thus calculate its identity, which would here be $(x + yD_x + zD_xD_y)$. This method facilitates the execution of operations on data which are in turn arranged as a vector, an array or a field. Threads within a block have the ability to be synchronized by an intrinsic lightweight instruction.

Due to reasons that concern the GPU's inner structure, there exist limitations regarding the maximum number of threads within a block. For this reason, if it is desirable that a kernel run in more threads than this number, threads can be organized in several blocks which in turn are arranged in a one-dimensional, two-dimensional or three-dimensional grid, in a way that is again determined by the user during each call. When a C program, as we described above, calls a kernel to be executed in a grid of threads, each block within this grid is indexed and assigned to an available multiprocessor (remember that GPUs are implemented as arrays of multiprocessors). Each block's threads are executed simultaneously and when a certain block fully terminates a new one takes its place. These multiple blocks, if possible, are executed in parallel, but they may also be executed sequentially (depending on whether there exists an available multiprocessor in the GPU and the size of the blocks). All the above leads to the selection of the thread number according to the number of multiprocessors existent in the GPU, contrary to the usually followed practice in classic multithreaded programming.

Threads have access to the various memory modules within a GPU according to a well-defined hierarchy. To begin with, each thread has its own private local memory which corresponds to the local memory of the processor it runs on. Secondly, threads within the same block all share the common shared memory of their multiprocessor. The above two memory types maintain their values and hence their meaning for as long as the kernel is being executed, that is for the lifespan of the block. Furthermore, there is the global memory which may be accessed by all threads at any time. Access to global memory though is the slowest amongst the various memory modules. Finally, constant memory and texture memory may also be accessed by all threads. Access to them is faster compared to global memory but their size is smaller in principle. Values stored in global, constant and texture memories also remain after the kernel has terminated. Figure 3.7 shows a representative example of thread hierarchy in CUDA, as well as the corresponding memory hierarchy.

According to CUDA programming model, each kernel runs on the GPU and not the CPU. That means that, from the time a kernel is called, the flow of the C program continues normally without the CPU waiting for the kernel to terminate its execution. The C program may later call a special CUDA function which suspends its execution until the GPU is done with its work, hence synchronizing GPU with CPU. Note also that the host system (that is the CPU on which the program runs on) and the GPU device have each a distinct physical memory space. The CPU's memory corresponds to the system's RAM while the GPU's memory corresponds to its global memory, with these two distinct memories being connected via a PCI bus. The programmer is responsible for the proper transfer of data from RAM to the GPU's global memory and vice versa. CUDA offers a set of special-purpose functions that enable the above data transfer.

In conclusion, it is noteworthy that CUDA has been designed to support a variety of programming languages. One of its great advantages is the relative ease with which it can be learned by people who already have some experience in programming with certain common languages, such as C. This compatibility with C and its notable potential for parallelism, combined with its rapid development during recent years, are the basic reasons for its usage in this thesis.

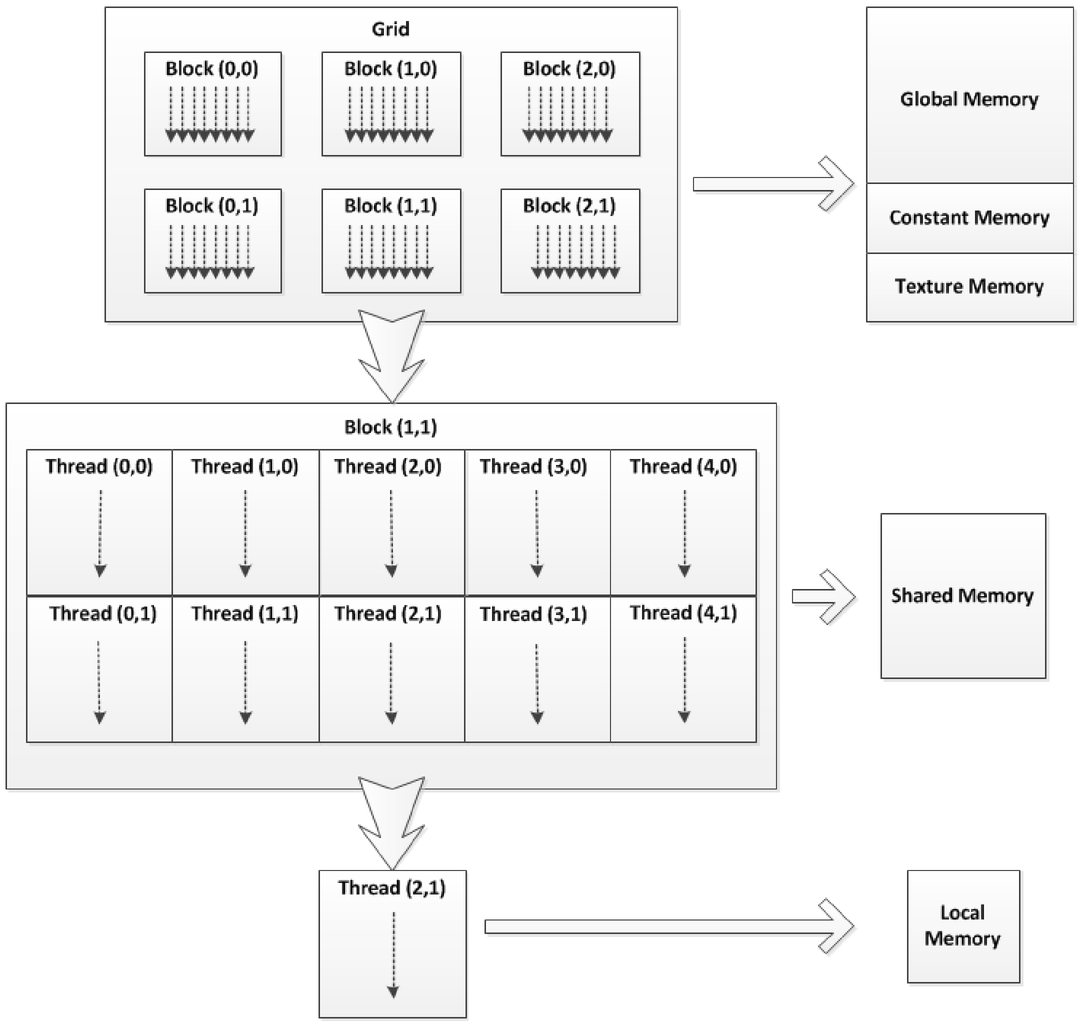

Figure 3.7: Thread and Memory Hierarchy in CUDA

3.2.3 CUFFT

Following FFTW's example, NVIDIA has developed a library within CUDA's programming environment which provides an easy-to-use interface for the efficient execution of parallel FFTs on GPUs. The library's name is **CUFFT** (CUDA Fast Fourier Transform). CUFFT provides, amongst others, the possibility of transforming arrays of any size or dimension, while it supports floating point arithmetic of both single and double precision.

In order to achieve its high performance, the CUFFT library implements a working model similar to FFTW's plan-execute mechanism. To be precise, CUFFT internally implements a variety of diverse FFT algorithms in the form of parallel kernels, which a *planner* composes in the optimal way for a certain transform size in order to maximize its performance. Next, an *executor*, according to the information produced by the planner, may execute multiple transformations of the same size, thus exhibiting an obvious performance benefit.

In this thesis, CUFFT library constitutes the “twin” library of FFTW for the FFTs execution in a GPU environment, which as it has been mentioned are necessary for the implementation of our Fourier domain methods.

4 FLCC Library

We have already described the different algorithms for the computation of convolution and correlation coefficients with local normalization and we have calculated theoretically the number of required operations for each one of them. Furthermore, we have presented the specific parallel architectures that we chose to support the execution of the computations. The next natural step in our research is the development of code based on the algorithms and the architectures, so as to make an essential and realistic result comparison between those. However we will not stop there. The multiformity of the algorithms and the effectiveness of the architectures regarding the confrontation of the problem lend themselves for a more complete handling of the programs' implementation. We desire this implementation to acquire the form of a realistic, practical schema that is able to successfully confront the problem of the computation of convolution and correlation coefficients as it occurs in real contexts, creating this way a useful and practical computational tool that can be used by anyone.

To this direction, we proceeded to the development of the **FLCC library**, a powerful and versatile computational tool aimed to the holistic confrontation of the problem of convolution and correlation coefficients in practice. The FLCC library is comprised of a collection of functions that effectively implement each one of the aforementioned algorithms, using the chosen modern parallel computational architectures. At the same time it is provided for use in the form of a complete software package, easy-to-learn and, we hope, useful in practice.

At the time of syntax of this diploma dissertation the FLCC library has reached version 1.3 (FLCC v1.3), about which we will thoroughly talk in this chapter. We will initially refer to the features that the current version provides its users (4.1). Next, we will describe the library's interface, meaning the set of functions that are being defined by it and via which the user gains access to its usefulness (4.2). Finally, we will make an extended description of its internal operation, i.e. what exactly happens from the moment the user calls any one of the functions provided (4.3).

4.1 Features

FLCC (Fast Local Correlation Coefficients) is a library for the C (or C++) programming language that provides its user a set of instructions for the fast computation of two basic operations of the field of digital image processing that are under examination in this diploma dissertation. As we already know, they are the sum of convolution (or correlation, exploiting the duality of these two operations) and the correlation coefficients with local normalization (or LCC distribution or LCCs) between an image and a filter/template. Their computation, especially the case of LCCs, has all along been considered to be of high arithmetic complexity and time-consuming, especially in the case of real-time systems and thus their use was being rendered problematic.

The aim of this library is to surpass this problem and provide its users a simple yet powerful interface for the execution of the aforementioned computations. In the case of LCCs a lot of implementations until now have tried to reduce the computation time by sacrificing the local normalization feature or by computing approximations of the result in some lossy ways that introduce inherent inaccuracies. This library on the contrary manages to reduce the computation time to the least possible, without making any compromises regarding the quality of the result. The user may be assured that the result at the output is precisely the "real" distribution (table) of LCCs, exactly as it is formally defined (see subsection 1.4.2 for the precise definition of LCCs).

The efficiency gain of FLCC is achieved in two ways. On the one hand, the library implements and incorporates a set of considerably optimized fast algorithms for the computation of convolutions and LCCs. These algorithms are precisely those that have been presented and analyzed within chapter 2. On the other hand, it fully exploits modern parallel hardware architectures with lots of capabilities and even already established libraries as has been described in chapter 3. Specifically, the libraries that are employed by FLCC for the execution of the computations are, as we have already seen, multi-core processors and graphics processing units (GPUs), the two of them comprising examples of high performance architectures for parallel processing. Therefore, the library maximally exploits the computational resources of the system it runs on, as it executes the computations concurrently on a number of threads on the central processing unit (CPU) and/or it transfers the computational load so as to have it executed by the admittedly powerful GPU of the system.

It is a fact that the different algorithms and architectures that are exploited within the FLCC library express different efficiency for each case of computation. In order to combine them among themselves towards an optimal result, the library applies a clever and flexible plan-execute mechanism. Via this mechanism, FLCC automatically chooses for each case of computation the optimal way of execution, i.e. the individual algorithm and architecture to be used, according naturally to the sizes of the images and the capabilities of the computer system. This way the optimal efficiency of FLCC is always guaranteed, without the need for any additional knowledge or action by the user, simultaneously guaranteeing the portability of this efficiency among different computer systems.

At the time of syntax of this diploma dissertation the FLCC library is at version 1.3 (FLCC v1.3). This version, albeit being fully operational and correct, still is in a stage of infancy and will have several more features added to it in the future (see chapter 7). What follows next is a concise enumeration of the features the FLCC version 1.3 library provides its prospective users:

- Fast computation of convolutions for 2D and 3D images and filters of any size
- Fast computation of distributions of correlation coefficients with local normalization (LCCs) for 2D and 3D images and templates of any size
- Single precision arithmetic
- Accelerated computation of LCCs and convolutions between a stream of images and the same template/filter
- Potential for execution of parallel computations either on a multi-core CPU or a (presently one) GPU

Finally, we would like to report that FLCC version 1.3 is supported for use on UNIX/Linux and MS Windows (via Cygwin) operating systems. At the time of syntax of this text it is freely available to install and use from the webpage http://flcc.cs.duke.edu/, distributed under the FreeBSD license, complete with full documentation.

4.2 Interface Description

Until now we have talked of the different algorithms and the different architectures that we employ in the FLCC library so as to compute convolution (or correlation) and correlation coefficients as efficiently as possible. In this section we will analyze the interface that the FLCC library provides its user.

We remind the reader the basic conclusion that was produced in chapter 2, i.e. the fact that one of the two types of methods (direct and Fourier domain) is more suitable than the other for any specific problem is dependent on the sizes of the images and mainly on that of the template. At the same time, it is a fact that each individual computer system has different features and that the comparative performance among each individual multi-core processor and GPU varies. We therefore are aware of the fact that the choice of a most suitable algorithm-architecture combination for the execution of any individual computation is not preordained but must be made in any case during run-time.

A basic design element of FLCC is that externally no piece of information appears regarding the means by which the computations are executed. In fact, this is the central concept of the library, to not relate the user to any practical issues regarding the computation of convolution/correlation or LCCs so that they be able, using a simple and concise API, of getting the results they desire the fastest and easiest way possible. In order for the user to remain disconnected from the type of algorithm that is executed and the architecture that is used in any case and for the simultaneous guarantee for a most efficient computation, the FLCC library applies on the interface level a plan-execute mechanism, similar to the one used by the libraries FFTW and CUFFT. This mechanism constitutes the "spine" of the library, as it manages to incorporate in a unified scheme all the different algorithms and architectures we have described until now.

The operational logic of the plan-execute mechanism is the following: initially the user calls a planner type function to which the desired type of computation is declared (number of dimensions, size of each dimension, computation for an image or a stream). The planner processes these data and searches for the most efficient algorithm-architecture combination that executes the desired computation. It therefore creates an object called plan that contains all the information for the optimal combination. Next, the user may call an executor type function that based on the specific plan executes the computation for specific images. The advantage of this approach is that the plan may be reused for an unrestricted amount of times for the execution of the same type of computations on the same or different images (which obey the specifications that have been given during the creation of the plan). Given the fact that in practice the execution of many computations of the same type is a recurring occurrence, one realizes that the total process of planning-execution may prove to be significantly efficient, as the plan is computed only once and is used for a multitude of different computations. This way, the FLCC library manages to efficiently and flexibly execute the computations, to combine the advantages of different algorithms and architectures and to adapt to the features of different computer systems, while this whole process is performed in a totally transparent towards the user way.

Before we proceed to the thorough description of the interface, we remind the reader that as part of the documentation of the FLCC library a detailed user manual is distributed, that is updated for every new version. We refer the user to the manual if the their aim is to use the functions and the data types of FLCC in their own program, so as to have under their disposal with detail all the new possibilities provided by the each time current version of the library. The manual is written in a way that the operations and features of the entire interface are explained simply and understandably, it describes issues concerning the installation and usage of the library and provides code examples.

Let us proceed then to the description of the interface. The version 1.3 of FLCC provides the capability for the computation of convolutions and LCCs using six main functions. Three of them have to do with the computation of convolution and three of them for the computation of LCCs. There are additionally two more functions for the allocation and deallocation of memory. In the subsections that follow we will thoroughly discuss the functions that perform the planning process (4.2.1), those that execute the main computations (4.2.2), those that deallocate the bound resources (4.2.3) and those that are charged with memory management (4.2.4). We will conclude with some information concerning the means of installation and usage of the FLCC library (4.2.5).

4.2.1 Planning Functions

We will begin the presentation of the individual functions with those that perform the planning process. They are the functions **`conv_plan(…)`** and **`lcorr_plan(…)`**, which are used for convolution and LCCs respectively. Their aim is to create the optimal plan for a type of computation. Their input is common among them and their declarations are the following:

```
flccResult  conv_plan(flccPlan *plan, int dim,
                      flccSize imSize, flccSize temSize,
                      flccType type, flccPlatform platform)

flccResult lcorr_plan(flccPlan *plan, int dim,
                      flccSize imSize, flccSize temSize,
                      flccType type, flccPlatform platform)
```

As is evident, we have defined new data types for some of the arguments. This is done in order to organize the argument input into the functions and to provide the user a structured way for setting the parameters for the execution of the functions. The general aim of the arguments is to describe the type of computation for which the planning is due. Next we explain one by one these arguments and the corresponding data types.

The object of type **flccPlan** is a data structure in which the plan is stored by the above functions, i.e. the necessary information with which FLCC will execute the computations for convolution of LCCs in as efficient as possible a way. The details regarding what kind this information is exist in subsection 4.3.5. The variable **plan** constitutes the output of the planner functions. The functions get as input a memory pointer to this variable, which must have already been allocated by the user.

With the integer number **dim** the user declares the number of dimensions of the problem. FLCC supports problems of two or three dimensions and by inputting another value the function returns an error.

The two next arguments are of type **flccSize**. This type is a data structure that contains the information regarding the size of an image. It is defined as follows:

```
typedef struct {
      int h;
      int w;
      int d;
} flccSize;
```

With the argument **imSize** the size of the image that is to participate in the computation is declared while with the argument **temSize** the size of the template is respectively declared.

The images are stored in memory in the form of arrays. The user must have in mind that the dimension **d** is the one that changes more often in their array, next is **w** and finally is **h**. The arrays are assumed to be arranged in row-major order (known also as C-order). In case the user has declared only two dimensions for their problem, the value **d** is ignored by FLCC and therefore the information must be contained in **h** and **w**. So, in the case of two dimensions, the size of each individual array equals to $h \times w$. In the case of three dimensions the size equals to $h \times w \times d$.

The data type **`flccType`** is an enumeration that is defined as follows:

```
typedef enum {
      FLCC_SINGLE,
      FLCC_STREAM
} flccType;
```

We therefore notice that the argument **`type`** can receive as values only the parameters **`FLCC_SINGLE`** and **`FLCC_STREAM`**. If the user intends to execute computations on a single image they must enter the value **`FLCC_SINGLE`**. If on the other hand they intend to execute computations on a stream of images, they must enter **`FLCC_STREAM`**. This choice exists so that the planning process be able to take into consideration the algorithmic benefits that occur from the use of streaming if the computation includes a stream of images. Let us note that the execution of the computation would operate correctly, i.e. would produce correct results, even if this variable had not received the appropriate value. In that case though, it would not be guaranteed that the computations were performed in the fastest possible way.

The data type **`flccPlatform`** is also an enumeration, which in turn is defined as follows:

```
typedef enum {
      FLCC_HOST,
      FLCC_DEVICE,
      FLCC_ANY
} flccPlatform;
```

It is evident therefore that the variable **`platform`** can receive three different values, those being **`FLCC_HOST`**, **`FLCC_DEVICE`** and **`FLCC_ANY`**. Depending on this value, the functions will respectively take into account during the planning stage only methods that are executed on a CPU, only methods that are executed on a GPU or finally all the available methods. So, the value **`FLCC_HOST`** forces the planner to choose the CPU, the value **`FLCC_DEVICE`** the GPU while the value **`FLCC_ANY`** lets the library free to choose the optimal architecture. This is the only way the user may intervene on the choice of architecture. We have chosen to add this capability to cover the case in which the user is either interested in the execution on one particular platform or certain that one of the two platforms is faster and thus does not wish for the planner functions to spend time in examining the methods of the other architecture. Then again if none of the above reasons is the case, we recommend the value **`FLCC_ANY`**, so that the full available potential of FLCC is exploited.

We notice that the above functions also have an output of type **`flccResult`**. This type is an enumeration of values that correspond to error messages of the FLCC library. The functions return the value **`FLCC_SUCCESS`** in the case of a successful execution or any other that signifies the occurrence of an error and its type. So, with a simple examination of the return value, the user can check whether the functions were executed correctly and if not they may immediately find out what exactly went wrong. Table 4.1 comprises the comprehensive list of the possible values of type **`flccResult`**, as well as the significance each one of these values has.

Table 4.1: Possible Error Values of the FLCC Library

Error Value	Significance
`FLCC_SUCCESS`	The function executed correctly
`FLCC_INVALID_PLAN`	The plan object is not valid
`FLCC_INVALID_DIMENSION`	The dimension is not valid
`FLCC_INVALID_SIZE`	Some image size is not valid
`FLCC_INVALID_TYPE`	The type object is not valid
`FLCC_INVALID_PLATFORM`	The platform is not valid
`FLCC_INVALID_VALUE`	Some pointer or array is not valid
`FLCC_ALLOC_FAILED`	FLCC failed to allocate memory
`FLCC_EXEC_FAILED`	FLCC failed due to an internal error

4.2.2 Execution Functions

Having executed a planner function, we receive at its output an object of type **`flccPlan`**. This will later be used for the performance of the computations. The two functions that execute the computations are **`conv_exec(…)`** and **`lcorr_exec(…)`**, for convolution and for LCCs respectively. The templates of these two declarations follow. Again, their inputs are the same.

```
flccResult  conv_exec(flccPlan plan,
                      float *image, float *templat,
                      float *conv, int imCount)

flccResult lcorr_exec(flccPlan plan,
                      float *image, float *templat,
                      float *lcc,  int imCount)
```

The first argument is the plan object that has been created by the call of one of the respective planner functions. It is the user's responsibility to call **`conv_exec(…)`** with a plan that was produced by **`conv_plan(…)`** and similarly to call **`lcorr_exec(…)`** with a plan that was produced by **`lcorr_plan(…)`**.

The three following arguments are pointers to **`float`** and point to the memory locations in which the images that participate in the computation are stored. These are in order the image (single or stream), the template/filter and the result (single or stream). We should note that FLCC assumes that the images with which it works are stored in memory as real-valued element arrays and, since presently it only supports single precision arithmetic, those elements are of type **`float`**. Each one element of an array corresponds to the value of a pixel of the respective image. The arrays **`image`** and **`templat`** comprise the input of the functions and are read by them without being modified. The array **`conv`**/**`lcc`** on the other hand is the output and in the end of the execution it will have been filled with the values of the result of the convolution/LCCs between **`image`** and **`templat`**. It is the user's responsibility to have appropriately allocated memory for the storage of these arrays (see subsection 4.2.4).

The library assumes that all the arrays are stored in row-major order (otherwise known as C-order). The order of the dimensions is **`h`**, **`w`**, **`d`** with **`d`** being the one that changes the fastest and **`h`** being the one that changes the slowest (the variables **`h`**, **`w`**, **`d`** refer to the data type **`flccSize`**, see subsection 4.2.1).

When the computation refers to a stream of images and not just a single image, the FLCC library assumes that these images are stored in RAM in successive memory locations and are all of the same size. So, the variable **`image`** is considered to point at the first element of the first image of the stream while the other images follow successively. In a similar form will occur the results, i.e. FLCC will store them in successive memory locations, starting from the location at which the variable **`conv`**/**`lcc`** points.

The integer number **`imCount`** informs the functions regarding the number of images to be processed, i.e. the number of images of the stream. If although during the call of the planner function the user had chosen as a parameter the value **`FLCC_SINGLE`** then the value **`imCount`** is ignored and it is assumed that there is only one image to be processed. In this case the variable **`imCount`** may receive any value but we nevertheless recommend the value 1 for reasons of consistency.

We will stand a little more on the issue of the sizes of the arrays, so that it becomes completely clear. The array **`templat`** is in any case of the size of the template/filter that has been declared during the call of the respective planner function. For the other two arrays there is a differentiation that is as follows: in the case of a single computation (i.e. if the plan has been created with the parameter choice **`FLCC_SINGLE`**) the size of the array **`image`** is simply equal to the size of the image as it has been declared during planning and the array **`conv`**/**`lcc`** will be of convolution/LCCs size, as it results from the sizes of both image and template. We have thoroughly explained how the convolution/size results in chapter 1. In the case of a computation of a stream of images (i.e. if the plan has been created with the parameter choice **`FLCC_STREAM`**) the size of the array **`image`** will be equal to the size of the image as it has been declared during planning multiplied with the number **`imCount`**. Similar to **`conv`**/**`lcc`**, its size will be the convolution/LCCs size multiplied with the number **`imCount`**. This stems from the fact that for a stream of N images, there will also be N output tables. Beware when choosing the value **`imCount`** as for a smaller value than the correct one, fewer computations will be performed than desired by the user while for a larger one garbage will be processed.

The value returned by the functions is of type **`flccResult`** and its aim is to inform the user concerning the success or not of their execution and the type of error that may have occurred. Table 4.1 comprises the comprehensive list of the possible values of type **`flccResult`**, as well as the significance each one of these values has.

The executor functions may be called multiple times for the same or different images using the same plan. A single prerequisite exists that each individual image be compatible to the dimension, size and the parameter **`flccType`** that have been used during the planning process for that plan.

4.2.3 Deallocation Functions

From the moment a plan is not further useful it should be destroyed, i.e. the memory that stores all the information retained by it should be deallocated. Towards this, two functions have been developed, the templates of which are the following:

```
flccResult  conv_destroy(flccPlan *plan)

flccResult lcorr_destroy(flccPlan *plan)
```

In both cases, the sole input argument is a pointer to the object of type **`flccPlan`** that we want to deallocate. The **`conv_destroy(…)`** function is intended for the destruction of the plans that have been created by **`conv_plan(…)`** whereas **`lcorr_destroy(…)`** is intended for those that have been created by **`lcorr_plan(…)`**. Following its destruction a plan is not reusable.

These functions return a value of type **`flccResult`** that informs the user regarding their successful or not execution. Table 4.1 contains the full list of possible values of this type and their explanations.

4.2.4 Memory Management Functions

All the arrays required as input arguments by the executor functions must have already been allocated on RAM by the user. In order to have the FLCC library successfully operate and fully exploit its potential, the memory allocated must obey to some restrictions. To accommodate the user in the resolution of this process without requiring of him knowledge of further internal details, FLCC provides a function for the allocation of memory that operates as the known **`malloc(…)`** and guarantees that the memory is allocated appropriately. Therefore, all the arrays to be used by FLCC must always be allocated by the following function:

```
void *flcc_malloc(size_t bytes)
```

The argument **`bytes`** signifies the number of bytes that are required to be allocated. The function returns a pointer to the first location of the allocated memory. This pointer is of type **`void`** and must be cast to the desired type (here **`float`**). If the allocation fails the pointer is set to zero.

Naturally there also exists the corresponding function for the deallocation of memory. This must be used instead of **`free(…)`**.

```
void flcc_free(void *array)
```

The argument **`array`** points to the memory that needs to be deallocated. This function does not return any value.

4.2.5 Installation and Usage of the Library

The FLCC library is free open source software. Its official webpage is http://flcc.cs.duke.edu/. On this webpage one can at any time find its freely available, most recent version. The library is distributed under the form of a complete software package, protected by the FreeBSD license, complete with full documentation and thorough instructions for the installation process. Presently it can be installed on UNIX/Linux or MS Windows (via Cygwin) environments. Following its installation, FLCC can be used in programs of the user's making by linking it to them during their compilation.

In order for the FLCC library to operate in its full form and potential, all the individual tools it uses must have already been installed on the system, namely pthreads, FFTW, CUDA and CUFFT. In case the user does not have either CUDA or CUFFT installed on their system, or does not own a GPU or simply does not wish to use one, the FLCC library provides the possibility for an installation without the use of a GPU. This way, all the methods that transfer the computational load to the GPU are excluded and FLCC restricts its computations to the system's CPU(s). The same applies to the case where the user does not have installed and does not wish to install the FFTW library. FLCC provides the possibility for an installation without the use of FFTW, excluding all the methods that are based on it (Fourier domain methods on the CPU). The two possibilities for partial installation may even be combined. In any case though, for the full exploitation of its potential, we recommend to the prospective user to have the FLCC library installed in its fullest version.

4.3 Internal Operation

A basic feature of the FLCC library is that while the interface provided is especially concise and simple, its structure and internal workings present proportionately greater complexity. Issues like the implementation of the various techniques and methods that we have discussed in previous chapters and the exploitation of the potential of the various architectures we have described are resolved transparently concerning the user, who is not required to know anything about them. Next in this section we proceed to the description of those worthy of analysis matters.

In subsection 4.3.1 we will talk of how the library is logically structured and of the type of functions that comprise it. Next, in the following two subsections (4.3.2 and 4.3.3) we will describe the exact means of the implementation of the direct method and the Fourier domain method respectively, on each one of the two architectures we used. We will therefore talk about how every stage of each algorithm is implemented, how the parallelism is achieved, what is computed by each thread, what each architecture offers, programming "tricks" like loop unrolling and the potential provided by ready-made software packages. In subsection 4.3.4 we will see the techniques with which the performance of the computations is improved in the case of a stream of images, as much algorithmically (explained in chapter 2), as by exploiting the asynchronous execution potentials provided by GPUs. Finally, in subsection 4.3.5 we will explain everything that happens concerning the plan-execute model, specifically for the FLCC library. We will describe everything that is not visible to its user, like what exactly happens by calling the interface functions, what information is contained in the plan structure, how the latter is used in the execution stage, various safety nets and more.

Let it be noted that for most of the issues we will next talk there are no noteworthy differences between the computation of two-dimensional and three-dimensional images. Thus, we will talk once per each issue, assuming that the extension of the concept for another number of dimensions is obvious.

4.3.1 General Structure of the Library

In Figure 4.1 the complete internal structure of the FLCC library is depicted in a schematic form. We notice that FLCC is structured as a three level logic, with each one of the levels using the services of the adjacently lower.

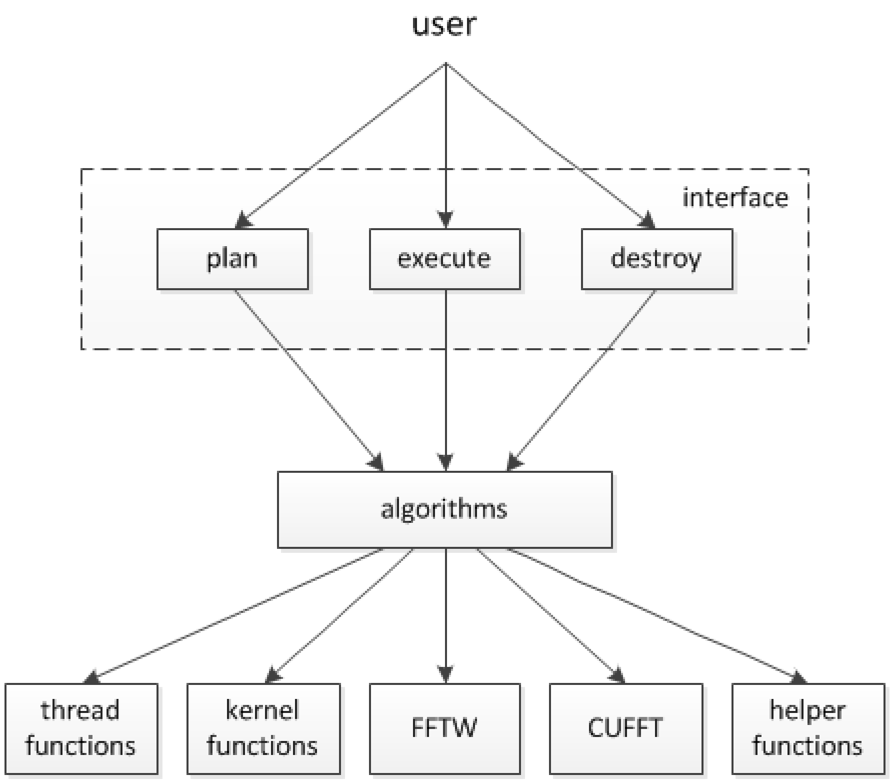

Figure 4.1: Schema of FLCC Library's Structure

The upper level is comprised of the functions that constitute the interface of the library and it is the only level accessible to the user. It implements the plan-execute mechanism by managing properly the functions of the lower levels. The interface level's functions have thoroughly been described in section 4.2.

The in-between level of the library includes all the functions that implement the individual algorithms that FLCC uses. This level's functions are managed by the interface level, which as we stated calls them in a proper way to execute the processes of planning and execution. Table 4.2 lists the main functions that comprise this level and explains all about their operation.

Finally, the lower level is comprised of all those elementary functions that are used by the in-between level of algorithms for the execution of low level computations. These are thread functions, CUDA kernels, the FFTW and CUFFT ready-made functions and some helper functions for a variety of uses. This level is responsible for the execution of all the basic computations of the library and for this reason it is comprised of low level, highly optimized functions.

In the subsections that follow we will proceed to a further explanation and analysis of the implementation of the individual functions that have been presented in this subsection.

Table 4.2: FLCC Library's In-between Level's Functions

Function	Operation
`conv2_host(…)`	Two-dimensional convolution using the direct method on a CPU
`conv2_dev(…)`	Two-dimensional convolution using the direct method on a GPU
`conv3_host(…)`	Three-dimensional convolution using the direct method on a CPU
`conv3_dev(…)`	Three-dimensional convolution using the direct method on a GPU
`fconv2_host(…)`	Two-dimensional convolution using the Fourier domain method on a CPU
`fconv2_dev(…)`	Two-dimensional convolution using the Fourier domain method on a GPU
`fconv3_host(…)`	Three-dimensional convolution using the Fourier domain method on a CPU
`fconv3_dev(…)`	Three-dimensional convolution using the Fourier domain method on a GPU
`lcorr2_host(…)`	Two-dimensional LCCs using the direct method on a CPU
`lcorr2_dev(…)`	Two-dimensional LCCs using the direct method on a GPU
`lcorr3_host(…)`	Three-dimensional LCCs using the direct method on a CPU
`lcorr3_dev(…)`	Three-dimensional LCCs using the direct method on a GPU
`flcorr2_host(…)`	Two-dimensional LCCs using the Fourier domain method on a CPU
`flcorr2_dev(…)`	Two-dimensional LCCs using the Fourier domain method on a GPU
`flcorr3_host(…)`	Three-dimensional LCCs using the Fourier domain method on a CPU
`flcorr3_dev(…)`	Three-dimensional LCCs using the Fourier domain method on a GPU

4.3.2 Direct Method Implementation

We have already discussed in chapter 2 about the direct algorithms with which convolution and correlation coefficients with local normalization are computed. These two operations are different however the direct methods for their computation are rather similar in that the template/filter passes over the image and a series of operations is being performed between certain elements of the image and the template itself in order to compute each individual output element. For given dimensions and for each relative position between image and template, the positions of the required elements of the image are the same for the two methods and only the series of operations differs. This is the reason we will examine the implementation of the two direct methods alongside, as the means by which their parallelization is carried out is very similar.

The main concept here is that we deal with two direct algorithms that are highly parallelizable. Every output element can be computed independently of the rest just by knowing the template and a part of the image. This provides the possibility to have every element be computed by a separate thread. Next we will talk about exactly how we exploit this property in the case the computations are performed on a CPU (as a rule a multi-core processor) but also in the case the computations are transferred for execution on a GPU.

4.3.2.1 Implementation on a CPU

We will begin with the case of **one or more CPU cores**. The inherent parallelism of the computations that we have described is implemented with the use of threads. As we have stated before, FLCC itself uses the pthreads library in order to create portable threads. As we said in subsection 3.1.2, the function `pthread_create(…)` is used to create new control threads among which the computational load is to be shared.

Let us discuss now about how the new set of threads is used in the execution of the direct methods. The number of threads the FLCC library creates is declared by the user during its installation and will henceforth be denoted by ν. As a rule, the library operates more efficiently when this number is equal to the number of cores on the processor chip (see subsection 5.2.3). Of course the user may select the number 1 in case he does not own a multi-core processor and thus one thread will perform all the computations.

As a rule there will be many more elements in the output table than there are cores. This means that each thread will be responsible for the computation of many more than one output elements. Each thread will have access to the following data: the whole image that has already been padded with zeros on each side (something that is also performed with the use of threads), the template, the table of convolution/LCCs (output table) in which the results of the computations will be stored and finally information relevant to the section of the output table that is under the thread's responsibility. Regardless of whether the image is of two or three dimensions, the memory that has been allocated for the output table is one-dimensional (linear). Let N_S be the total number of elements of the output table. Each thread is responsible for the computation of a continuous linear section of this table with a size of about $\frac{N_S}{\nu}$ elements. We say about because usually there is a remainder υ in this division. This means that an additional element corresponds to the first υ threads that are created, such that the division be as balanced as possible. The number of elements that correspond to each thread is called the offset of the thread. Beyond the number of elements, in order to assign correctly a section to a thread, each thread requires the knowledge of what the first element of the section is. This is easily calculated from the sum of all the offsets of the previous threads. It is clear that the first thread starts from the first element of the table. In Figure 4.2 an example of the apportionment of a two-dimensional output table of size 16×16 among 10 threads is depicted.

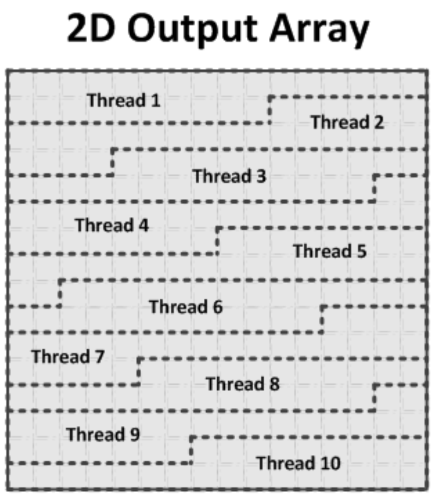

Figure 4.2: Example of the Apportionment of a 2D Output Table among threads on a CPU

As each thread begins the execution of its given thread function, it enters a loop that performs a number of iterations equal to the thread's offset, with each one computing one output element. So, in each iteration the proper computation between image and template/filter is performed, i.e. inner product in the case of convolution and the steps **3** and **4** of the direct method in the case of correlation coefficients (more in chapter 2). So in the end of the iterations each thread would have computed the entire section that corresponds to it.

It is evident from the algorithms of the direct methods that for the computation of each output element an iteration loop structure is required that passes the whole template and the image's panel that corresponds to a given relative positioning. The number of iterations of this loop will be equal to the size of the template. However it is a fact that in practice the overwhelming majority of the templates that require the direct method have equal dimension sizes (squares or cubes) and a rather small size (let us not forget that for large template sizes the Fourier domain method is more suitable). This provides a rather notable possibility for optimization of the functions being executed by the threads that lies in the concept of loop unrolling. Indeed, the FLCC library implements a general thread function for the computation of convolution/LCCs with a regular use of the iteration loop but it does not stop there. It retains optimized versions of this function that refer to specific template sizes (from 2×2 to 32×32 for the two-dimensional case and from $2 \times 2 \times 2$ to $10 \times 10 \times 10$ for the three dimensional). Since the size of the template is known for each one of these individual functions, the use of an iteration loop becomes unnecessary as the number of its iterations is consequently known. Therefore, these functions fully unroll the iteration loop, achieving notably high time performances.

The means by which FLCC performs the full unrolling of the loops in the thread functions is a clever use of macro-instructions. Using the pre-processor instruction **`#define`**, a macro-instruction is defined that implements the thread function and receives as arguments the dimensions of the template. During the installation of the FLCC library several calls of this macro-instruction that correspond to the specific template sizes we stated before are given for compilation. In the definition of the macro-instruction there is, of course, the iteration structure normally, so as of now its avoidance is not evident. What has happened though is that the number of the iterations has been set to constant values. So, during FLCC's installation, the compiler knows the precise number of iterations of the loop and thus unrolls it fully.

During the execution, right before the threads are created, a check is performed to see whether the template is square or cubic and also if its size corresponds to one of the optimized thread functions. If not, the general, unoptimized function is set to perform the computation. So, the optimized execution is achieved for the overwhelming majority of the common cases while simultaneously the correct performance of the computation is guaranteed even in the uncommon cases.

4.3.2.2 Implementation on a GPU

Let us proceed to the case of execution with the use of a **GPU**. Based on what we have already discussed regarding GPUs and CUDA, in this case it is possible to have as many CUDA threads be created as the number of elements of the output table, with each thread executing the kernel that implements the direct method for only one output element.

We will briefly talk now about the execution of kernels and the organization of threads into a grid of blocks. The blocks may have a maximum number of threads, which on current GPUs is of the scale of 1024. We used square blocks in the case of a two-dimensional image and cubic blocks in the case of three-dimensional images. The blocks' sizes are constant and are set by us. The size of the two-dimensional blocks is 16×16 and of the three-dimensional blocks is $8 \times 8 \times 8$. In the case of two-dimensional images, the first thread block is responsible for the output elements that are in a manner of speaking on the upper left corner of the output table (if we assume that is the origin of the axes for each dimension). The rest of the blocks are arranged to the right in order and similarly downwards, so as to form an apportionment of the output table in a rectangular way. In a similar way the blocks are also arranged in the case of three-dimensional images. In Figure 4.3 an example of the apportionment of a two-dimensional output table of size 16×16 in CUDA blocks and threads is depicted.

For a random table size it is evident that most of the times the blocks that are at the end of a dimension would be responsible for less output elements than their number of threads. Thus, in the kernel there should normally be a control structure **`if`** to verify whether each thread is within the bounds of the table, so as to prevent it from writing on memory locations that are out of bounds. In FLCC though, for performance reasons we desired to avoid this control structure and so we tried something different. We allocate on the global memory more space than is needed for the image and the output table, such that the size of each dimension of the output table is larger or equal to the convolution's size of the dimension and also perfectly divisible with the respective block dimension size. Thus we completely avoid the control structure inside the kernels. This means, of course, that some threads belonging to the extreme blocks will perform some computations that do not have a logical counterpart. This however is not a problem as these elements are removed from the final result and thus do not have any impact on it. Let us note that this process of expanding the tables is incorporated in the padding process of the image that takes place one way or the other before the execution of the main computation and therefore does not negatively affect the final time performance.

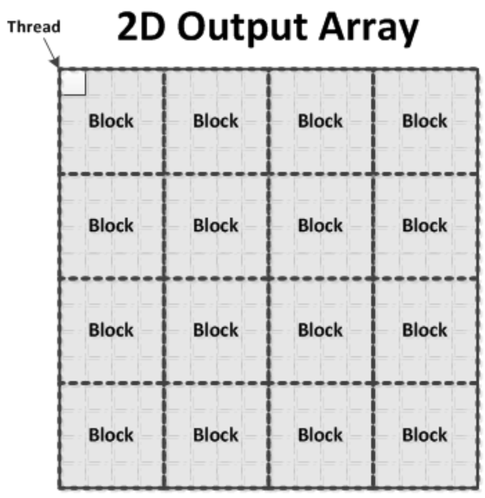

Figure 4.3: Example of the Apportionment of a 2D Output Table among Threads on a GPU

The means by which CUDA perceives the arrangement of blocks is called a grid. For a two-dimensional image the grid is also two-dimensional and its size is the following:

$$\frac{S_X}{B_X} \times \frac{S_Y}{B_Y} \tag{4.1}$$

where S_D is the size of dimension D of the output table S (which we allocate in the way we described before) and B_D is the size of dimension D of the block. This way the grid exactly corresponds to the apportionment of the output table in blocks we have described. Something similar could also take place in the case of a three-dimensional image, meaning that we could form the three-dimensional grid like:

$$\frac{S_X}{B_X} \times \frac{S_Y}{B_Y} \times \frac{S_Z}{B_Z} \tag{4.2}$$

which perfectly relates to the way the table is divided among blocks. It is a fact though that some GPUs do not support a three-dimensional grid, just two-dimensional, so for portability reasons we choose to do something different. We arrange thus all the three-dimensional blocks into a two-dimensional grid of size equal to:

$$\left(\frac{S_X}{B_X} \cdot \frac{S_Y}{B_Y}\right) \times \frac{S_Z}{B_Z} \tag{4.3}$$

In other words, the dimensions X and Y of this "virtual" three-dimensional grid (as we assume the output table is divided into blocks) are depicted along the first dimension of the "real" two-dimensional grid (as CUDA perceives it) while the dimension Z is depicted along the second one. The depiction is performed in a way such that, if (U, V) is the position of a block in the real two-dimensional grid, its position (x, y, z) in the virtual three-dimensional grid would result like so:

$$x = U \; mod \left(\frac{S_X}{B_X}\right) \tag{4.4}$$

$$y = \left\lfloor U / \left(\frac{S_X}{B_X}\right) \right\rfloor \tag{4.5}$$

$$z = V \tag{4.6}$$

Using the above coordinate transform, each thread will be able to determine the block in which exists the output element that it corresponds to. Next, receiving the coordinates of itself inside the block, the thread can easily determine the general position of the element on the output table. The above process is clearly depicted in Figure 4.4, where a characteristic example of the formation of a two-dimensional grid from three-dimensional blocks is presented.

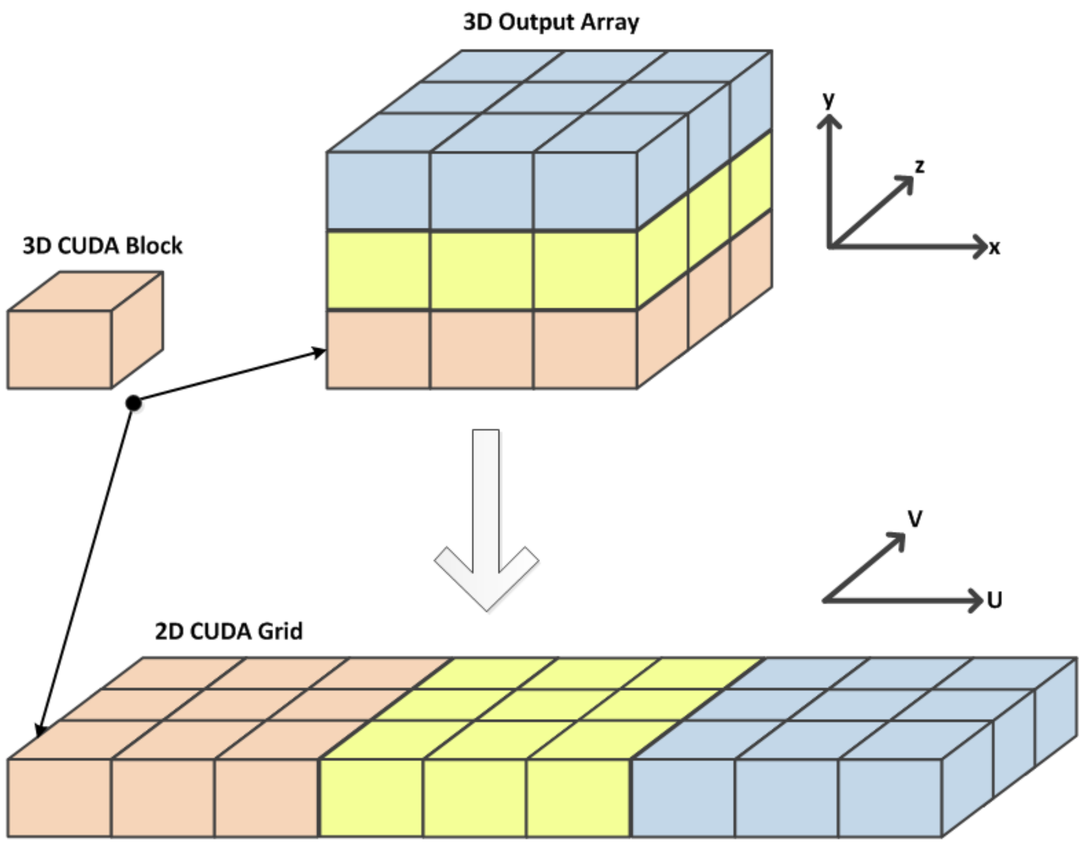

Figure 4.4: Example of the Transform of a 3D Virtual Grid to a 2D Real Grid

With what we have defined until now we have developed the naive kernel according to which all the tables that the threads require access to exist on the GPU's global memory. However there is a multitude of possibilities for further optimization, which we analyze in the following paragraphs.

Each thread block, as we have said before, is executed by a separate multi-processor of the GPU. Each multi-processor has its own shared memory that is considered to be cached and presents a significantly smaller access time in comparison to the global memory. The notable feature of this shared memory is that, in contrast to the cached memory of a CPU, it is completely managed by the programmer. This means that the correct exploitation of its potential is a critical factor in the achievement of high efficiency. The basic concept behind its use is the transfer to it of those data that are used again and again by multiple threads in the block. In our case, these data are on the one hand the template and on the other hand the section of the image that is required for the computation of the output elements that correspond to the whole block. Each dimension of this section is equal to the respective dimension of the block plus the respective dimension of the template minus one. In Figure 4.5 the means by which FLCC manages the shared memory in the case of a two-dimensional computation is depicted.

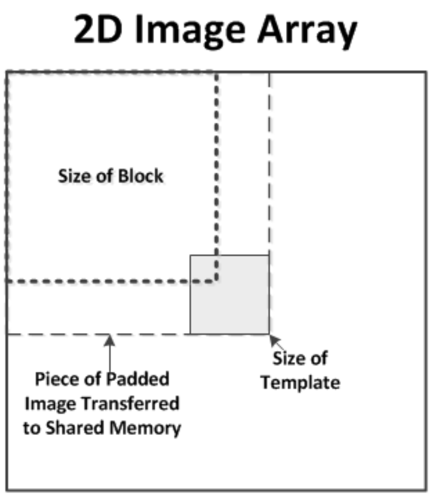

Figure 4.5: Schematic Representation of the Use of Shared Memory for a 2D Image

The transfer to the shared memory is implemented like so: before the execution of the main computations, each thread inside a block transfers concurrently with the others one element from the template and one element from the image to the shared memory. This is repeated as many times as it is needed (typically a few) until the transfer of all the necessary elements is complete. Once all the elements are on the shared memory, they will be visible to all the threads inside a block and thus the main computation will be in a position to commence. The key to the success of this strategy is that the transfer is performed once and lasts little, as it is being performed concurrently by all the threads. Still, the elements that result into the shared memory are used lots of times from more than one thread each, with a minimal access time every time.

Relative to the implementation of the direct method on a CPU, FLCC creates highly optimized kernels for specific cases of template sizes. The optimization also lies in this case on the full unrolling of the iteration loop that passes over the template and the section of the image for the computation of one output element. The kernels that are optimized this way correspond to template sizes from 2×2 to 32×32 for two dimensions and from $2 \times 2 \times 2$ to $8 \times 8 \times 8$ for three. This time though a completely different technique is used than the one in the CPU case, specifically, code generation. Specifically, we developed using MATLAB a code generator that generates the code of a kernel for a given template size. Next we used this generator to generate the kernels that correspond to the sizes we stated before. In these kernels there is usage of shared memory and of full unrolling of loops. The code of these kernels is included in the library and is compiled during its installation. Therefore, during the execution, before the kernel is called using CUDA's special syntax, a check is made to see whether the template corresponds to any optimized kernel. If so, that kernel is called. If not, the naïve kernel is called so that the correct computation of the result is guaranteed in any case.

We will conclude this subsection by making a comparison between these two methods for optimizing code using loop unrolling that we used in FLCC. We remind the reader that these methods are on the one hand the use of macro-instructions and on the other hand the explicit code generation by a generator. It is clear that the code generation method definitely leads to a maximum efficiency, as the code it generates can be fully optimized. This method though has one significant disadvantage: it drastically increases the size of the source code and thus the compilation time during FLCC's installation. The macro-instruction method on the other hand does not have the disadvantages of the use of generators but it does not exhaust the potential for code optimization. During FLCC's development the achievement of a maximum efficiency existed always as a design goal, thus we did not hesitate in using the code generation method for GPUs. In the case of CPUs though, it happens that the macro-instruction method produces equivalently good results as the code generation method, because of the high optimization that is achieved by the compiler. We therefore chose to use this method so as to restrain the size of the source code to lower levels as well as the compilation-installation time.

4.3.3 Fourier Domain Method Implementation

Having finished with the implementation of the direct method, in this subsection we will discuss the means of implementing the Fourier domain method. We have described in chapter 2 the relevant algorithms for the computation of convolution and LCCs and we have seen that they present a degree of similarity. This similarity lies on the fact that in the case of LCCs essentially a computation of three convolutions is performed in the Fourier domain, which in turn combined appropriately result in the LCC distribution. So, the description of the implementation of the Fourier domain methods will concentrate on the stage of convolution and the transforms that take place during which.

We remind the reader that we compute convolution based on the convolution/correlation theorem (section 1.6), i.e. on the transform of the image and the template into the Fourier domain, their point-by-point multiplication and the inverse transform of the resulting complex signal back to the spatial domain so as to receive the result of the convolution. Additionally, we said that since we have to occupy ourselves exclusively with image processing, the value that any element in the spatial domain has is real. This means that by properly using FFT the Fourier transform of a real-valued signal can be executed in $2.5N \log N$ operations (i.e. half than the regular case) where N the size of convolution, something that we will call a "half FFT". The reason for this reduction in time is that nearly half the elements of the complex signal in the Fourier domain are needed for the retrieval of the initial signal with an inverse transform. Specifically, in our case $S_X(\frac{S_Y}{2}+1)$ elements are needed for two dimensions and $S_X S_Y(\frac{S_Z}{2}+1)$ elements for three, where S_D is the D dimension of the convolution table S. This number for any number of dimensions is called the Hermitian size.

Since the above are common concerning computations as much of two as of three dimensions, the only noteworthy differentiation lies on their implementation on the two different architectures we use, i.e. the multi-core processors and the graphics processing units.

4.3.3.1 Implementation on a CPU

We begin with the case of the execution of the computations on **one or more CPU cores**. For the execution of the FFTs we use the already established FFTW library, which provides ready optimized functions for the computation of the half FFTs of two and three dimensions that we need. It also supports multi-threading, as it internally uses the pthreads library and therefore can be executed in parallel, exploiting all the available CPUs of the system. On the other hand, for the execution of the point-by-point multiplications, we share the computational load among different CPUs using the pthreads library. We will examine all of this analytically in the following paragraphs.

Let us discuss now the way FFTW is used for the execution of the FFTs. The moment the FFTs are supposed to be executed there already exist two FFTW plans that have been created during FLCC's planning stage (FFTW's planning stage is incorporated in the respective FLCC stage, see subsection 4.3.5). One of these is responsible for the transform from a real-valued signal to a complex-valued one (real-to-complex) and the other one for the inverse transform from a complex-valued signal to a real-valued one (complex-to-real). Both of these transforms are performed with a half FFT. Each created plan is given as an argument to FFTW's appropriate for the type of the transform execution function (real-to-complex or complex-to-real) along with two pointers to tables. These tables are the current input signal and output signal of the transform, the real-valued signal being represented by an array of type **`float`** and the complex signal of the frequency domain being represented by an array of type **`fftwf_complex`**. The type **`fftwf_complex`** is defined by FFTW as an array of two **`float`** values, which correspond to the real and the imaginary part of a complex value. It is noteworthy that the real-valued table is always of convolution size while the complex-valued table is of Hermitian size.

It is a fact that the speed with which an FFT is executed by FFTW is heavily dependent on the dimension sizes of the table to be transformed. The overwhelmingly faster cases are the ones where the dimension sizes of the input table may be written as products of small prime numbers. In order to exploit this feature of FFTW, we expand the tables to be transformed to a convolution size that fulfills this requirement (internally in FLCC). In other words, if the real convolution size as given is not satisfactory, FLCC seeks to find the next larger convolution size that it deems it will lead to a fast transform and expands the tables to be transformed to this size by padding with zeros at the end of each dimension. In the two-dimensional case, the size of each dimension that is chosen is equal to the next power of 2 for a dimension size up to 1024, while for larger ones it is equal to $1024 + k \cdot 512$ with k being the smallest possible. Respectively for three dimensions, the next power of 2 is chosen for a dimension size up to 128, while for larger the value $128 + k \cdot 64$ is chosen with k being the smallest possible. This process is incorporated into the otherwise necessary padding process and in the end the extra elements that have been added are removed from the final result.

We will now proceed to an explanation of how the point-by-point multiplication between two complex signals is being performed using the pthreads library. More on the means that we use pthreads to share the computational load have already been presented in the previous subsection (4.3.2) since their use there is more intensive. We will redo a small review though for the sake of clarity. This multiplication is performed between two signals that are in the Fourier domain and thus are being represented by tables of type **`fftwf_complex`**. The number of complex elements they comprise of equals to the Hermitian size. This means that their number of elements is equal to about half the number of elements of the respective signal in the spatial domain, as that many are needed since they have been formed by a half FFT from a real-valued signal and that many are needed to fully retrieve that initial signal. The table that contains the result of this complex multiplication will also be of Hermitian size. Each complex element of position i of this element is computed only by knowing the two respective complex elements in position i of these two tables to be multiplied. We therefore create using pthreads ν threads of which each one is responsible for the computation of a continuous section of the output table with complex elements numbering $\frac{H}{\nu}$, where H is the Hermitian size. Naturally this division always leaves a remainder υ. Therefore the first υ threads will have access to $\frac{H}{\nu} + 1$ elements. The number of elements that corresponds to any thread is called the offset of the thread. Beyond the number of elements, each thread needs to know the first element of the section of the output table that corresponds to it. This is easily calculated to be equal to the sum of all the offsets of the previous threads. Obviously the first thread starts from the first element of the table.

A detail that we would like to note is that by transforming a signal to the Fourier domain and next back to the spatial domain using FFTW's functions, the resulting signal is not the initial. The initial will result with the division of each element of the new signal with the total number of its elements, i.e. the convolution size. This necessary division is incorporated by FLCC in the above multiplication so as to have it executed in parallel, since because of the linearity of all the computations, it is not important at which stage it is performed.

4.3.3.2 Implementation on a GPU

We therefore proceed to the case of a **GPU**. For the computation of FFTs on a GPU we use the CUFFT library by NVIDIA and for the point-by-point computation we call a kernel that performs it.

The use of CUFFT for the case of execution on GPUs within FLCC mirrors the use of FFTW for the case of execution on CPUs. So, when during the stage of execution of the computations of convolution/LCCs a number of FFTs has to be performed, two CUFFT plans have already been created by the planning stage of FLCC. Again one of these plans contains all the necessary information for the execution of a Fourier transform from a real-valued convolution-sized signal to a complex-valued Hermitian-sized signal and the other for the inverse transform. Therefore the appropriate execution function of CUFFT is called (we use one for the real-to-complex execution and one for the complex-to-real execution for each individual convolution) and the respective plan and the two tables that correspond to the input and output signals are given as arguments. These two tables are allocated on the global memory of the device. Warning: the real-valued table that corresponds to the signal that exists on the spatial domain is of type **`cufftReal`** and of convolution size and the table that corresponds to the signal in the Fourier domain is of type **`cufftComplex`** and is of Hermitian size. These types are CUFFT's embedded data types for the representation of real and complex elements respectively.

CUFFT, as FFTW, performs much faster in transforms with dimension sizes that can be expressed as products of small prime factors. So, similar to the execution on a CPU, FLCC expands the dimension sizes of the tables to be transformed to the next it deems will lead to a fast transform. The sizes that are chosen are the same as those that are also chosen by FFTW (see previous paragraph) so we will not repeat ourselves.

Let us move on to the means by which the point-by-point multiplications between signals in the Fourier domain are performed. As we have stated before we have to do with two tables allocated on the device memory that are of Hermitian size. A complex multiplication is executed between each couple of values of position i of the two tables and the result is stored in the position i of a third table, also of Hermitian size. With the inverse Fourier transform of this third table we get the convolution of the two initial signals. So for the computation of each of these complex elements of the $i-th$ position of this third table only the complex elements of the $i-th$ position of the first two tables are needed. So, to each CUDA thread the computation of an output element is assigned. We have therefore developed a kernel that runs for a number of threads equal to or less than the Hermitian size. The computation of each element of the table is assigned to a thread of some block. Since the block size is constant (16×16 for two dimensions, $8 \times 8 \times 8$ for three), what changes is their number. For two dimensions, the upper left section of the table is assigned to the first block (assuming that the origin of the axes is located there) and each next is arranged to the left and down in a rectangular grid, receiving simultaneously the proper coordinates. Somewhat similar the apportionment of the jurisdiction of the blocks is performed for three dimensions. As for the arrangement of blocks in a grid, the same apply as stated in subsection 4.3.2 and thus we will not repeat ourselves.

We should finally note that, exactly as in the case of the usage of FFTW, CUFFT requires the division of the output elements of the inverse transform by the convolution size, a process that is incorporated in the above kernel and is performed during the stage of the multiplication.

4.3.4 Streaming Implementation

In this subsection we will talk about the means by which streaming is implemented on FLCC. Except for the means by which the general algorithmic improvement is achieved due to streaming and has been thoroughly explained in chapter 2, we will talk about the particularities that occur when using a GPU.

As we have already stated in the section about FLCC's interface (4.2), the option is provided to the user to obtain the result of convolution/LCCs between one template and a series of images with just one call of the execution function. Let there be N images to be processed. It is consequent that after the execution of one of the executor functions on streaming mode there will also exist N convolutions/LCCs. This series of images we call a stream of images. This stream is given to one of the executor functions in the form of an array equal to the size of each individual image multiplied by N, on which all the images to be processed are stored.

Internally in FLCC, for the execution of computations on a stream of images the same functions are used as in the case of an individual computation. Before the process of computation commences, a **`for`** loop structure starts that performs a total number of iterations equal to the number of images in a stream. Along one iteration the computation between one image from the stream and the template is performed, so that at the end of the loop the entire stream would have been processed. It is evident that in the case of the processing of a single image the loop will be executed only once (as a trivial case of streaming). In the case of streaming with the use of a Fourier domain method, the structure **`for`** contains within it only that part of the computations that must be performed for each new image and those parts that can be computed only once are computed before the **`for`**. The details concerning the algorithmic improvements in computations on streams of images have already been stated in chapter 2.

Let us discuss now some of the GPU's particularities that hamper the performance of computations on a stream and the means with which we confronted them. A disadvantage in the use of the GPU is the long time needed for the transfer of data from the computer's RAM to the global memory of the GPU and vice versa. For a single computation the transfer of the image and the template from RAM to global memory and the transfer of the convolution/LCCs from the global memory to RAM are required totally. For computations on a stream of images the transfers are the following: the template, all the images of the stream to the global memory and all the convolutions/LCCs back to RAM. This would be particularly time-consuming though CUDA provides an alternative that we exploit, not to avoid these transfers but to "hide" them under the computation time. CUDA lends the possibility for asynchronous concurrent data transfer and kernel execution for certain GPUs. The concept therefore is: during the time the convolution/LCCs between the i-th image and the template is computed, the $(i-1)$-th convolution/LCCs is concurrently transferred to RAM and the $(i+1)$-th image is transferred to the device. With this concurrency trick the only data transfers "visible" are the ones of the template and the first image to the global memory and of the last convolution/LCCs to RAM as the rest of the transfers are performed during the interval that the computations are concurrently performed.

However a problem arises. Naturally no computations may be performed on an array during the time its elements are being changed because of transfers. Therefore we use two buffers, one of image size to have the image be transferred to it from RAM and one of convolution size to have the result be transferred from it to RAM. So the process is being done in two phases like so:

1. During the first phase the following are being performed concurrently:
    - Transfer of next image from RAM to respective device buffer
    - Transfer of previous result to RAM from respective device buffer
    - Computation of current result
2. During the second phase the following are being performed concurrently:
    - Transfer of image from image buffer to the position of array to be processed
    - Transfer of newly-computed result to result buffer

In Figure 4.6 the above process is schematically depicted, where the operation of the aforementioned buffers in the individual phases of streaming is clearly presented.

Let us state though that all the above are valid for the GPUs that support such processes. In the case the GPU does not support the concurrent transfer and execution, the various operations are performed correctly though in a serial way, having a proportionate impact to the overall performance.

Having explained the logic with which streaming on a GPU is implemented, we will now talk about all the technical issues regarding this concurrent transfer and execution. The means by which CUDA performs this asynchronous execution is with the use of CUDA streams (beware not to confuse these with image streams). These streams are CUDA instruction sequences that can be executed concurrently. We use three streams, something that is preordained by the first phase we described in which there is the requirement of three different operations being performed concurrently. So each one of the three individual operations of the first phase is assigned to a stream and thus the three of them can be executed in parallel. For the second phase the two out of three streams are being used for the two individual operations that are also being executed in parallel. We should note though that between the two phases the appropriate CUDA synchronization function is being called which ascertains that the operations of one phase have been terminated before the initiation of the other one.

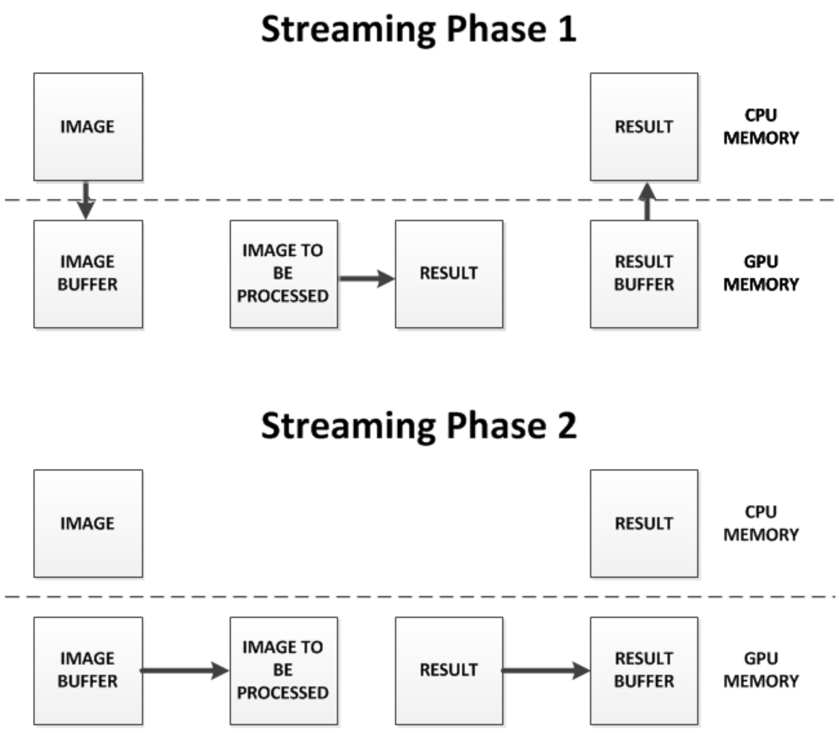

Figure 4.6: Streaming Process on a GPU

Finally, for asynchronous data transfer, CUDA requires that the space in RAM that the data are stored or are to be stored be page-locked. CUDA provides the necessary functions for the correct allocation and deallocation of page-locked memory. These functions are called by our own `flcc_malloc(…)` and `flcc_free(…)`, ascertaining that the memory is being allocated as page-locked and thus the overlap of transfer and execution is possible. We should state that an additional benefit of the use of page-locked memory is that the speed by which the data transfer between RAM and GPU is significantly increased (regardless of overlap or not with the execution). The mandatory disadvantage inherent in its use is that the maximum quantity of page-locked memory that can totally be allocated is limited in contrast to the respective quantity of regular allocated memory.

4.3.5 Plan-Execute Mechanism Implementation

We will now talk about what exactly happens by each one of the main interface functions of FLCC. We will see, namely, what constitutes the stages of planning, execution and deallocation that constitute the main usage of FLCC.

4.3.5.1 Planning Stage

We will now explain what is being performed inside the planning functions by initially laying bare the code defining the data structure **flccPlan**. We remind the reader that an object of such a type constitutes the output of the aforementioned functions. Next we will explain how these functions compute everything that is inside this structure.

```
typedef struct {
        flccSize        imageSize;
        flccSize        templateSize;
        int             dimension;
        flccMethod      method;
        flccType        type;
        flccPlatform    platform;
        int             numCpuCores;
        int             numGpuDevices;
        void *          c2r_plan_dev;
        void *          r2c_plan_dev;
        void *          c2r_plan_host_diff;
        void *          r2c_plan_host_diff;
} flccPlan;
```

The variables **imageSize**, **templateSize**, **dimension**, **type** and **platform** get the values of the respective arguments that are given by the user at the call of the planning function, as they are analyzed in subsection 4.2.1. In the case invalid elements are given the planning function returns a relevant error.

The variable **numCpuCores** gets the value that also is given by the user during the installation of the FLCC library and corresponds to the number of threads that will be created using the pthreads library.

The variable **numGpuDevices** constitutes the number of GPU devices that FLCC uses. Although FLCC still does not support sharing of the workload among multiple GPUs, this variable is always set to value 1 and presently is not used anywhere. We chose to implement it however because in some future version of FLCC this may change.

The user gives as the function's input argument the sizes of the dimensions of the image and the template. The function calculates the convolution size and four plans are created, two by FFTW and two by CUFFT. For each couple, one plan contains the information for the half Fourier transform (real-to-complex) and the other for the inverse Fourier transform (complex-to-real). These plans are used inside the planning process of FLCC, are stored in the structure **`flccPlan`** and are the same that will possibly be used in the execution stage. Therefore, the executor function will use FFTW's plans in case the Fourier domain method on a CPU is chosen and CUFFT's plans in the case the Fourier domain method on a GPU is chosen. The plans' types are declared as pointers to **`void`** so as to render that part able to be compiled in case the user does not have FFTW and/or CUFFT installed on their system. When some plan is being used later by the executor, the proper type casting is performed.

The most significant part of the planning stage is the choice of the most suitable method for the execution of the computations. As we have previously seen in subsection 4.3.1, different functions that implement each method, on each architecture and each number of dimensions have been developed. The means by which the information of which one of these functions must be called inside the execution function is stored in the plan is reflected on the variable **`method`** of **`flccPlan`**. This variable can get the values that are declared by the following enumeration:

```
typedef enum {
      FLCC_DIRECT2_DEV,
      FLCC_DIRECT3_DEV,
      FLCC_FAST2_DEV,
      FLCC_FAST3_DEV,
      FLCC_DIRECT2_HOST,
      FLCC_DIRECT3_HOST,
      FLCC_FAST2_HOST,
      FLCC_FAST3_HOST,
      FLCC_NULL_METHOD
} flccMethod;
```

From the above strings, the word DIRECT designates the direct method, FAST designates the Fourier domain method, the numbers 2 or 3 designate the dimension of the problem, DEV designates the implementation of the method on a GPU and HOST designates the implementation on a CPU. The last string **`FLCC_NULL_METHOD`** is a safety net in the case none of the existing methods could be chosen.

The logic of the process for choosing methods is the following: for random image and template tables of the given dimensions the corresponding computations are performed (convolution or LCCs) using each one of the available functions. Each one of the individual functions is timed with precision and the function that requires the least time to perform the computations on the random data is the one that gets chosen to be used in the execution stage and the respective tag will be stored in the variable **`method`** of the plan object.

There is a possibility that any of the functions may fail in its execution (for example due to a lack of space in RAM or a disconnected GPU), in which case some error is returned. So, before any decision is made regarding the choice of the most suitable method, the element **`method`** is set to the value **`FLCC_NULL_METHOD`**. For the first function that succeeds in its computations, the corresponding string is given to the variable **`method`**. For every next function, if it fails it is discarded without further analysis. If on the other hand it terminates successfully and its execution time is less that the execution time of the one that corresponds to the string that is at any time stored in the variable **`method`**, then the new relevant string is stored. This way, even if all the individual methods fail, the value of **`method`** will still be **`FLCC_NULL_METHOD`** and the planning function, taking that fact into account, will return a relevant error thus notifying the user.

In the case the user has chosen the value **`FLCC_STREAM`** for the variable **`type`** the timing is done in another way. Instead of timing the regular functions that execute the computations (as in the case of the value **`FLCC_SINGLE`**), some other functions are timed that, for each case, execute only those computations that are required for any new image of belonging in the stream and not those that are performed once (in the last are also included the data transfers among RAM and GPU). These functions do not compute the real result of convolution/LCCs, however they simulate in a much better way the time required per image in the stream and therefore produce more reliable results compared to the executions of the regular functions.

Let it be noted finally that if the user selects via the **`platform`** to contain the execution of the computations in only one platform, the planning function will respond accordingly by excluding from the timing process all those methods that are to be executed in the non included platform. Something similar happens in the case the user has chosen to install FLCC without the use of the libraries FFTW and/or CUDA. In that case the code sections that time the methods using the excluded libraries will not be compiled.

4.3.5.2 Execution Stage

When an execution function is called, an object of type **`flccPlan`** is given as an argument that has resulted from a previous execution of a planning function. The execution function knows what computation function it is supposed to use by reading the value of the variable **`method`**. The rest of the information that is contained in the plan, meaning the image and template sizes, the existence or not of a stream of images, (if needed) the pre-computed plans of FFTW or CUFFT and (if needed) the number of threads to be created, are used by the execution function so as to use as arguments for the proper call of the designated method. Finally, we remind the reader that the tables on which the computations will be performed are given as arguments to the execution function and are conveyed by it to the appropriate computation method.

4.3.5.3 Deallocation Stage

The objective of the functions that destroy the FLCC plan is essentially to destroy the plans of FFTW and CUFFT by calling their appropriate deallocation functions. It is a fact that the plans of those two libraries allocate system memory that is advisable to be deallocated when they are not further needed. The existence of the deallocation functions of FLCC preserves the transparency to the user regarding those two libraries and furthermore secures in any case the proper destruction of the plans.

5 Experiments

In chapter 4 we presented FLCC library and we described its particular characteristics. In this chapter we will thoroughly test the library through a series of experiments in order to show its computational capabilities. At the same time, we will explore the way in which its various algorithmic and architectural features, as they have been analyzed so far, affect its behavior and determine its performance.

The experiments we are about to present in the following sections were carried out on a computer system of a multi-core processor (CPU), equipped with a Graphics Processing Unit (GPU). Their technical features are described in Table 5.1 and Table 5.2 respectively.

Table 5.1: CPU's Technical Features

Model	Intel Xeon E5620
Clock speed	2.40 GHz
Number of cores	2×4 with hyper-threading
Cache memory size	12 MB
RAM size	32 GB
Operating system	GNU/Linux

Table 5.2: GPU's Technical Features

Model	NVIDIA Tesla C1060
Clock speed	1.30 GHz
Number of cores	240
Shared memory size	16 KB/block
Global memory size	4 GB
Driver program	CUDA Driver v4.0

The presentation of the experimental results is divided into three sections. The first one (5.1) is about the comparative performance of the algorithms the library implements. The second one (5.2) focuses on the comparative performance of the two architectures that are used (multi-core processor and GPU). The third and last one (5.3) examines the behavior of the plan-execute mechanism that governs the library's functionality.

We shall finally note that the FLCC library's version that was used in all experiments presented here is FLCC v1.3.

5.1 Comparative Performance of Algorithms

In chapter 2 we developed two different methods for the computation of both convolution and Local Correlation Coefficients (LCCs). These are the direct method and the Fourier domain method (with or without streaming), which, after having been adapted to the peculiarities of both convolution and LCCs, derive four different algorithms, two for convolution and two for LCCs. The library implements these four algorithms separately for the two-dimensional and the three-dimensional cases. Distinct implementations also exist for the multi-core processor (CPU) and the Graphics Processing Unit (GPU). In this section we measure the time performance of the entire set of the above 16 routines for a range of different image and template sizes and we present the results in diagram form.

5.1.1 Constant Image – Varying Template

To begin with, we will measure the time performance of each algorithm for a **constant image size** and a **varying template size**.

We start with the two-dimensional case, where the image was chosen to have a size of 2000×2000 pixels while the template size varies from 2×2 to 32×32 pixels with a step of 1 pixel in each dimension (always square). In Figure 5.1 we can see the execution times on CPU, using 16 threads. The diagrams show the execution times for both a single image and the streaming case of 10 images, with time being per image.

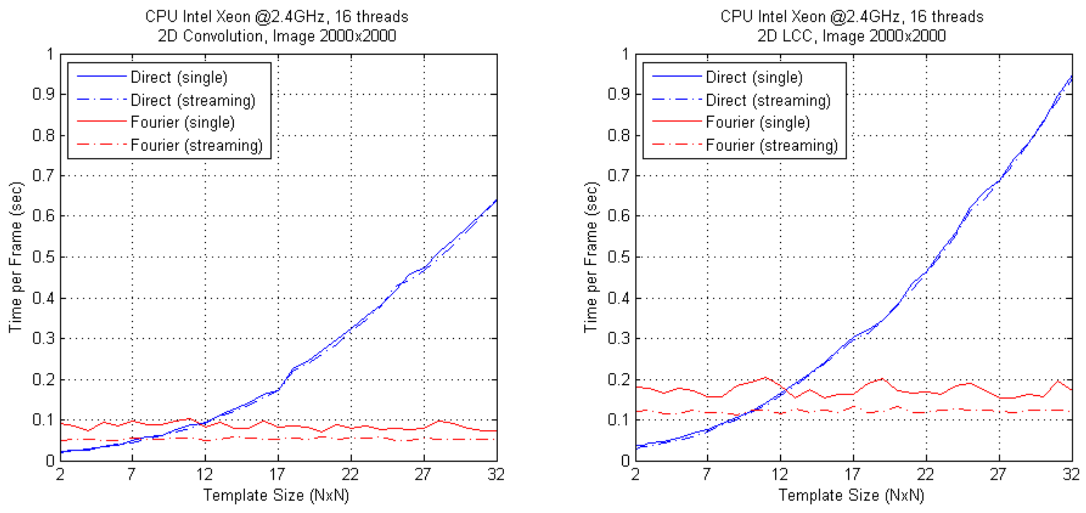

Figure 5.1: Execution Times on CPU for Constant 2D Image

Similarly, in Figure 5.2 we can see the corresponding execution times on GPU.

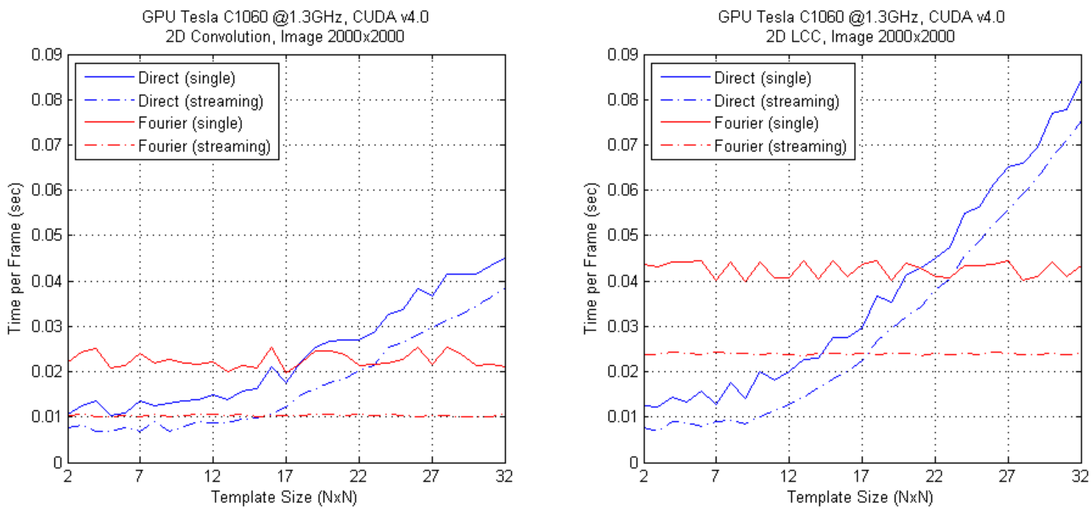

Figure 5.2: Execution Times on GPU for Constant 2D Image

The conclusions following from the above diagrams are both many and interesting. First of all, we observe that the execution time of the direct method increases at a quadratic rate while the execution time of the Fourier domain method remains steady. This fact fully confirms the theoretical analysis on the algorithms that was developed in chapter 2, where it had been predicted that the execution time of the direct method demonstrates substantial dependence on the size of the template, contrary to the Fourier domain method where the execution time (for relatively small templates) depends almost exclusively on the image size and remains practically unaffected by template size.

The second conclusion that is noteworthy has to do with the effect of streaming on execution time. We had theoretically predicted that the direct method does not benefit by streaming. Indeed, in the case of CPU the execution time with or without streaming appears to be the same. In GPU though there is everywhere a constant time improvement of about $5-10$ ms. This improvement is not algorithmic but rather technical. In order for the computation to be executed on the GPU the relevant data need be transferred from RAM to GPU's global memory, which in principle requires a certain non-negligible amount of time. Nonetheless, in the case of streaming, meanwhile an image in the stream is being processed, the next image to be processed is being transferred to GPU, so that it will be available when its turn should come. At the same time, the result is being transferred back from GPU's global memory to RAM. This way, data transfers are literally being "hidden" behind execution, thus inducing an improvement in the execution time per image that corresponds to exactly the time needed for data transfer. On the other hand, in the case of the Fourier domain method, the improvement achieved by streaming is a remarkable one in any case. Table 5.3 presents this time improvement in real percentages versus theoretically predicted ones.

Table 5.3: Fourier Method's Improvements in Streaming for Constant 2D Image

	Convolution	**LCC**
Improvement on CPU	$25 - 45\%$	$20 - 40\%$
Improvement on GPU	$50 - 60\%$	$40 - 47\%$
Theoretical Improvement	33.3%	28.6%

We notice that the Fourier domain method indeed confirms the theoretically predicted time improvements that were achieved by algorithmic means (FFT pre-computation, see chapter 2 for details). In addition to that, in the GPU case the improvement is even greater, thanks to the overlapping in transfer and execution which is realized in the same way as in the direct method and was described in detail above.

Our third major conclusion regards the "critical point", that is the template size for which the execution times of the direct method and the Fourier domain method become equal. The significance of this size is fundamental to the functionality of the library as it signifies the "preference border" between the two methods. Table 5.4 indicates the critical template sizes in each case in comparison with the theoretically predicted ones.

Table 5.4: Critical Sizes of 2D Template

	Convolution		**LCC**	
	Single	**Streaming**	**Single**	**Streaming**
CPU	11×11	8×8	12×12	10×10
GPU	18×18	15×15	21×21	17×17
Prediction	13×13	10×10	14×14	12×12

From the above values it becomes clear how the critical template size depends on the architecture implementing the algorithm. In the case of the multi-core processor, the critical size appears to be slightly smaller than the theoretically predicted one, a fact which is mainly due to the excellent performance of the FFTW library through which the Fourier domain method's FFTs are computed on the CPU. On the other hand, on the GPU the critical size seems to be somewhat greater than the theoretically predicted one, a fact which demonstrates the GPU's "preference" on algorithms with a high parallelism potential, such as the direct method, against harder to be parallelized ones, such as the FFT and hence the Fourier domain method.

We move on to repeat the same experiment as above, this time on the three-dimensional case instead. The image here was selected to have a constant size of $100 \times 100 \times 100$ pixels while the template varies from $2 \times 2 \times 2$ to $8 \times 8 \times 8$ pixels with a step of 1 pixel in each dimension (always cubic). Figure 5.3 shows the execution times on CPU (with 16 threads) for both a single image and streaming of 10 images, with time being per image.

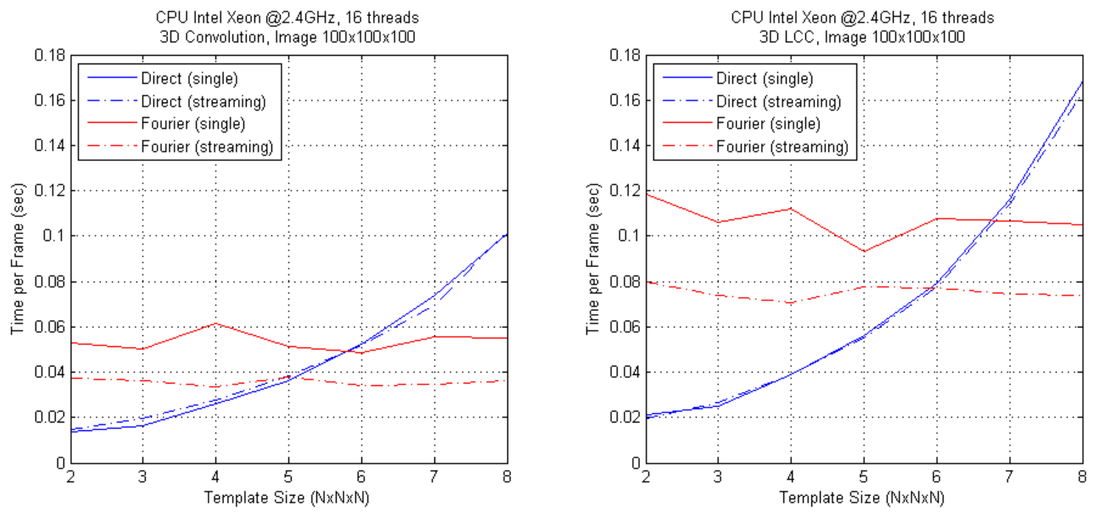

Figure 5.3: Execution Times on CPU for Constant 3D Image

Similarly, Figure 5.4 shows the corresponding execution times on GPU.

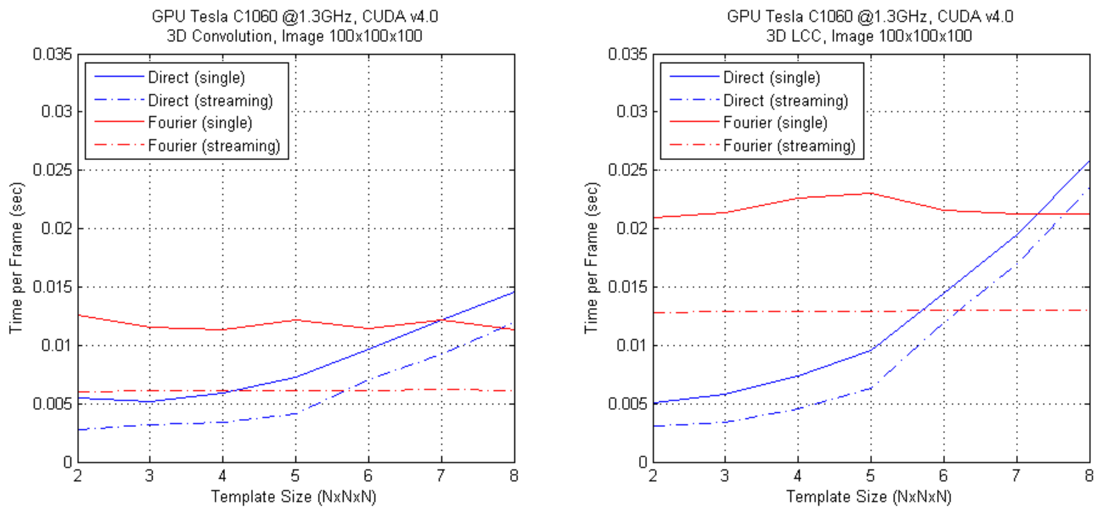

Figure 5.4: Execution Times on GPU for Constant 3D Image

The conclusions following the three-dimensional case are similar to those of the two-dimensional case. In fact, we observe that the execution time of the direct method increases substantially as the template size gets bigger whilst the execution time of the Fourier domain method remains at the same level. Streaming's effect is in the same way beneficial. Even if the direct method is not affected on CPU, we notice that on GPU it does benefit by streaming for a steady amount of time of about $2-3$ ms, due to the overlapping in data transfer and execution. The Fourier domain method, as expected, is further benefited. Table 5.5 indicates the percentage improvements on the execution time of the Fourier domain method thanks to streaming.

Table 5.5: Fourier Method's Improvements in Streaming for Constant 3D Image

	Convolution	LCC
Improvement in CPU	$26 - 46\%$	$20 - 35\%$
Improvement in GPU	$46 - 53\%$	$39 - 44\%$
Theoretical Improvement	33.3%	28.6%

We notice that on CPU the time improvements vary around the expected ones while on GPU they appear to be even greater, due to the further benefit of the overlapping in transfer and execution. Finally, Table 5.6 lists both real and theoretical critical sizes of three-dimensional template, as these follow from the diagrams and the theoretical analysis respectively.

Table 5.6: Critical Sizes of 3D Template

	Convolution		LCC	
	Single	**Streaming**	**Single**	**Streaming**
CPU	$6 \times 6 \times 6$	$5 \times 5 \times 5$	$7 \times 7 \times 7$	$6 \times 6 \times 6$
GPU	$7 \times 7 \times 7$	$6 \times 6 \times 6$	$7 \times 7 \times 7$	$6 \times 6 \times 6$
Prediction	$6 \times 6 \times 6$	$5 \times 5 \times 5$	$6 \times 6 \times 6$	$5 \times 5 \times 5$

We see on the above table that the real critical sizes agree with the theoretically predicted ones.

5.1.2 Constant Template – Varying Image

We now move on to the second phase of our algorithmic experiments, where we will measure the time performance of the various routines of FLCC library for a **constant template size** and a **varying image size**.

We begin with the two-dimensional case, where the template was selected to have a size of 16×16 pixels while the image varies from 1000×1000 to 4000×4000 pixels with a step of 100 pixels in every dimension (always square). In Figure 5.5 we can see the execution times on CPU (with 16 threads) for both a single image and the streaming of 10 images, with time being per image.

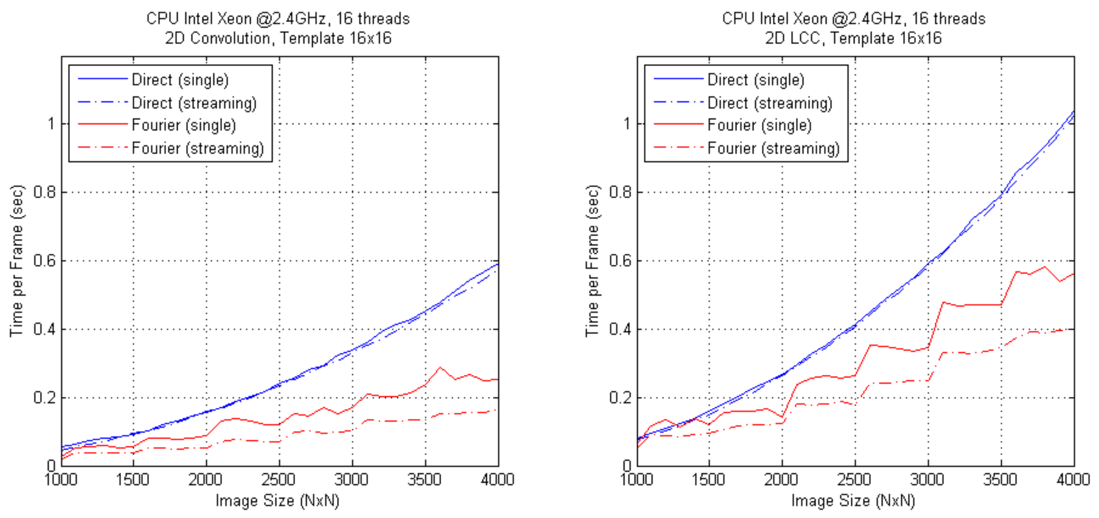

Figure 5.5: Execution Times on CPU for Constant 2D Template

Similarly, in Figure 5.6 we can see the corresponding execution times on GPU.

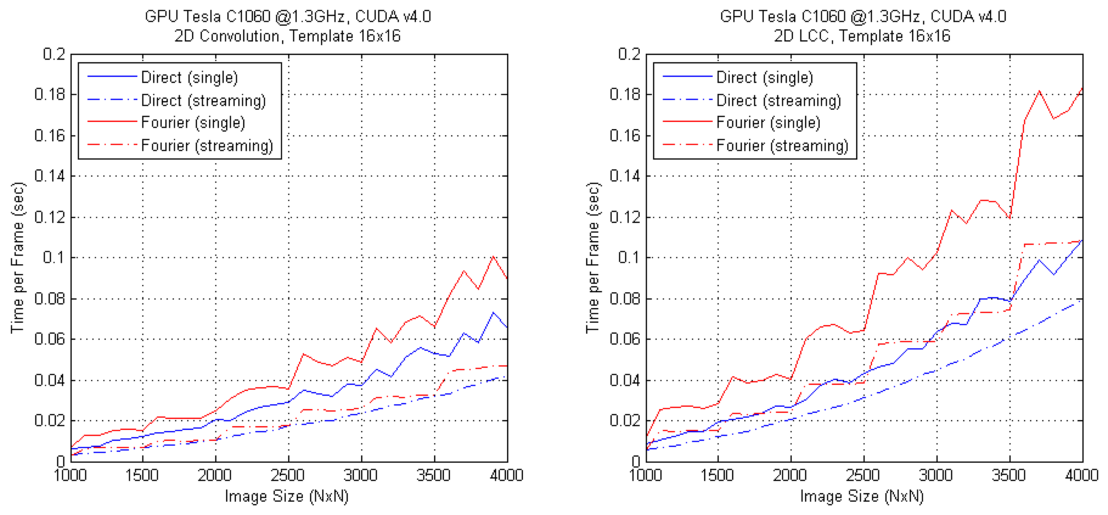

Figure 5.6: Execution Times on GPU for Constant 2D Template

From both the above diagrams it is apparent that the execution time of both methods increases at a quadratic rate as the image size gets bigger, as it is predicted by the theoretical complexities of the corresponding algorithms. The above is the basic difference that the variations in the image and template sizes have on the execution time of the two methods. Indeed, the execution time of the direct method increases at the same rate as both sizes increase, whilst the execution time of the Fourier domain method is only affected by the image size and not by the template size.

The effect of streaming is obvious in the above diagrams. We can see the improvement on the execution time of the direct method on the GPU due to the overlapping in transfer and execution. We can also see that this improvement increases in absolute numbers as the image size gets bigger, which is quite rational if we think that a larger image size implies a greater delay for the transfer of the image and the result to and from the GPU's global memory respectively. It is interesting to compare the above to the experimental results for a constant image size, where the time improvement of the direct method was a constant number. On the other hand, in the case of the Fourier domain method, Table 5.7 lists both the real and the theoretical improvement percentages on the execution time thanks to streaming.

Table 5.7: Fourier Method's Improvements in Streaming for Constant 2D Template

	Convolution	LCC
Improvement on CPU	$25-45\%$	$20-35\%$
Improvement on GPU	$46-58\%$	$36-46\%$
Theoretical Improvement	33.3%	28.6%

It appears from the above table that the time improvements on CPU vary around the predicted ones. On GPU they are even greater due to the overlapping in transfer and execution.

It is interesting to notice that the execution time of the Fourier domain method appears to increase in a scale form, or in other words in the form of successive "stairs". In fact, this is due to the technical implementation of this method by the FLCC library. It is true that the libraries that have been used in FLCC for FFT computation, namely the FFTW library on CPU and the CUFFT library on GPU, perform a lot faster when the signal to be transformed has "good" dimensions, that is dimensions that can be expressed as a product of small prime factors, with the best case being powers of two. The difference in performance is so big that FLCC chooses instead to "enlarge" the images to be processed by adding zeros, in order to achieve image dimensions that, according to FLCC, lead to a faster transform. Of course, these zeros do not distort the result in any way and are removed at the end of the processing. This way, all images in a certain range of dimensions are enlarged to the next "good" dimensions. This practice, which leads to efficient processing even of images of "difficult" dimensions (such as large prime numbers), is what causes the scale form in this method's execution time.

We proceed to the three-dimensional case. The template is selected to have a constant size of $8 \times 8 \times 8$ pixels and the image varies from $50 \times 50 \times 50$ to $250 \times 250 \times 250$ pixels with a step of 25 pixels in each dimension (always cubic). Figure 5.7 presents the execution times on CPU (with 16 threads) for both a single image and the streaming of 10 images, with time being per image.

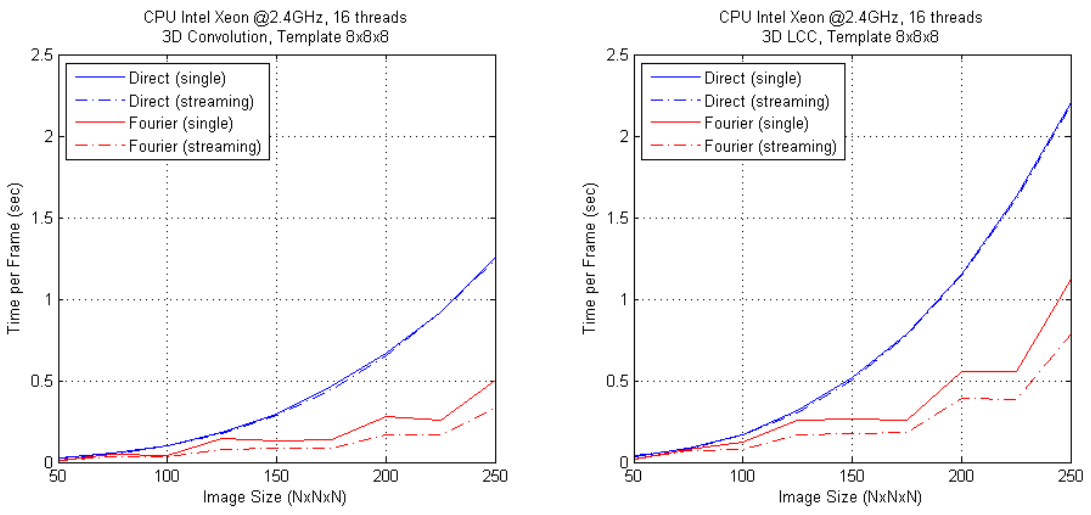

Figure 5.7: Execution Times on CPU for Constant 3D Template

Similarly, Figure 5.8 presents the corresponding execution times on GPU.

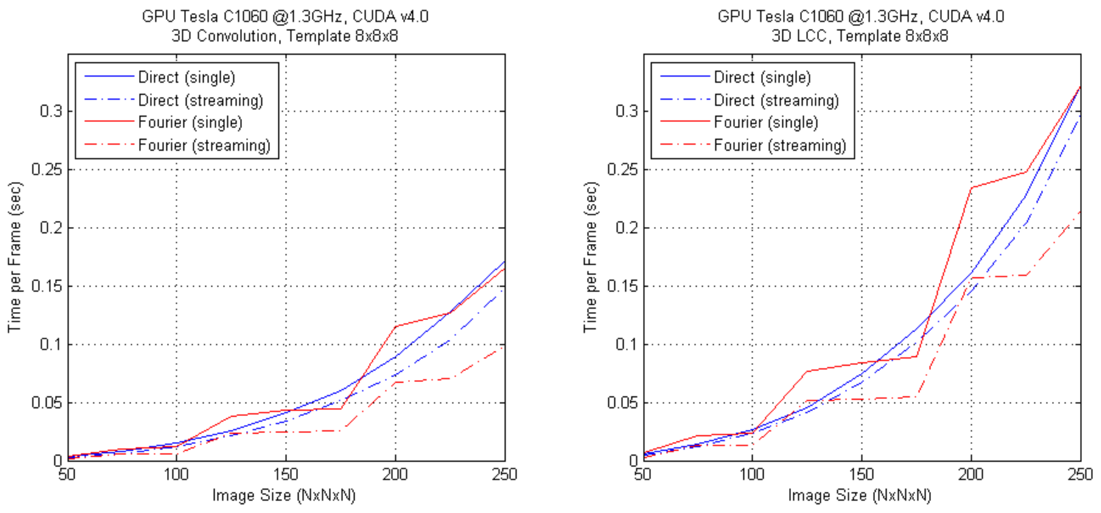

Figure 5.8: Execution Times on GPU for Constant 3D Template

The conclusions following from the above diagrams are similar to those of the two-dimensional case. We see that both methods exhibit a similar increase rate in their execution time as the image size increases. Streaming causes an improvement to the direct method on GPU, which corresponds to the transfer time of the image and the result and becomes larger, in absolute time units, as the image size increases. In the case of the Fourier domain method, Table 5.8 includes the real time improvements in comparison with the theoretical ones.

Table 5.8: Fourier Method's Improvements in Streaming for Constant 3D Template

	Convolution	LCC
Improvement on CPU	$20-45\%$	$15-40\%$
Improvement on GPU	$40-65\%$	$32-60\%$
Theoretical Improvement	33.3%	28.6%

We can see that the improvements on CPU vary around the expected ones while on GPU they are even greater due to the overlapping in transfer and execution. Finally, notice that also here the execution time of the Fourier domain method appears to increase in a scale form, which is due to the enlargement of images to the next dimensions that produce fast transforms, as it has already been explained.

5.2 Comparative Performance of Architectures

Having completed the comparative analysis among the various algorithms used in the FLCC library, in this section we focus on the time performance of the architectures themselves on which the algorithms are implemented, namely the multi-core processor and the Graphics Processing Unit. Our aim is to study the way each algorithm behaves on each architecture, how each architecture's special characteristics affect the time performance of the algorithms and finally what is the comparison in performance between the two architectures. Of course, this study will inevitably refer to the system we have and use, we hope though that the special results that will be obtained will have the ability to be generalized beyond the limits of a certain system and thus will describe effectively a large group of systems of the same kind, leading to conclusions for multi-core processors and Graphics Processing Units in general.

Before we start referring to experimental results, we will attempt to provide a somewhat naïve but indicative first answer to the question of what the relation between the performances of CPU and GPU on our system is. Our CPU contains 8 cores with hyper-threading technology (so virtually 16 cores) and has a speed of 2.40 GHz, while our GPU contains 240 cores and has a speed of 1.30 GHz. Taking into account the above parameters we get:

$$\frac{240}{16}\cdot\frac{1.30\,GHz}{2.40\,GHz}=8.125 \tag{5.1}$$

In other words we expect the GPU to be typically around 8.125 times faster than the CPU. Of course this value is simply indicative since we do not take other important factors into account at all, with the main one left out being memory access speed. Using it as a reference point, though, we should be able to say that if for some algorithm the relative performance between GPU and CPU is greater than 8.125 then the algorithm shows a clear preference to the GPU, likewise if it is much lower than 8.125 its preference seems to be the

CPU and finally if the two values are close to each other then the two platforms support equivalently the implementation of the algorithm.

5.2.1 Constant Image – Varying Template

We proceed to the presentation of the experimental results. First of all, we will use the results that were obtained from the experiments of section 5.1 and we will present them in such a way that reveals the relative performance between the two architectures. We begin with the two-dimensional case of a **constant image size** and a **varying template size**. Figure 5.9 presents the ratio of the execution time on CPU (with 16 threads) over the execution time on GPU. We remind that the image is of size 2000×2000 pixels while the template varies from 2×2 to 32×32 pixels with a step of 1 pixel in each dimension (always square). The diagram shows the time ratio for both a single image and the streaming of 10 images.

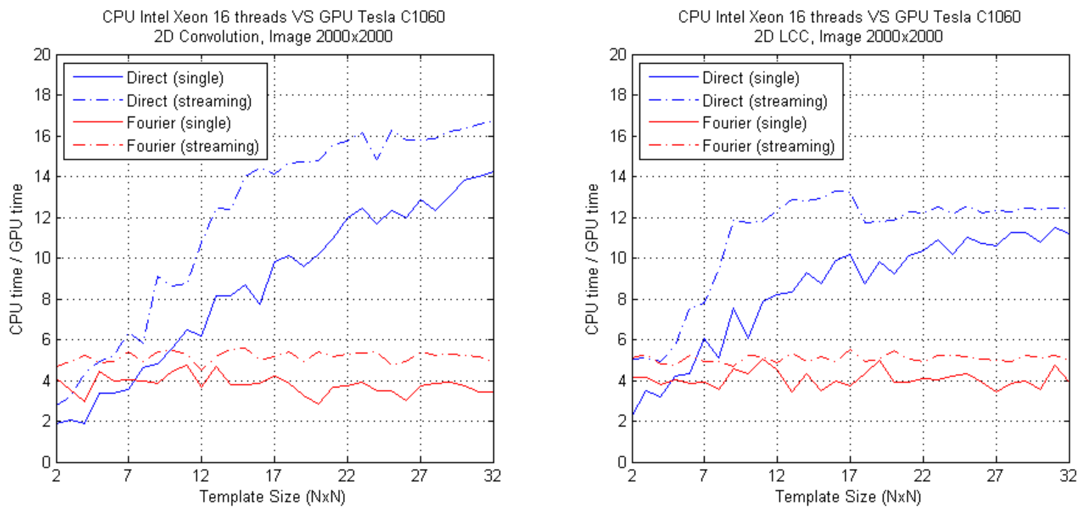

Figure 5.9: Comparison between CPU and GPU for Constant 2D Image

From the diagrams we can see, as it was expected, that the GPU is generally faster than the CPU. Based on the above results, we examine first the direct method in detail. We observe that for quite small templates the GPU is not significantly faster than the CPU but as template size increases, the direct method appears to prefer GPU to CPU. We may say that for large templates the GPU seems much more appropriate for the implementation of the direct method while for small ones its advantage vanishes. This result is a rather expected one since the GPU is designed to perform best for algorithms of high parallelizability and computational intensity. The direct method belongs certainly to this category, with its computational intensity being proportional to template size. For small templates, the ratio of computation over data transfer to/from memory is quite low, leading the GPU, whose memory is undoubtedly its weak point, to lower performance. On the other hand, the Fourier domain method appears to be steadily about 4 times faster on GPU. Since this value is half of 8.125 that we consider to be a logical value for performance comparison between

GPU and CPU, we arrive to the conclusion that the Fourier domain method performs better on CPU. This is a result that we have implied several times, stating that the FFT is an algorithm not ideally suitable for implementation in a GPU environment. Also, the fact that CPU appears to be more appropriate than GPU for the Fourier domain method is certainly assisted by the usage of FFTW for the execution of FFTs on CPU, whose high performance is definitely well-established.

Another weak point of GPU is the unavoidable transfer of necessary data between RAM and GPU's global memory, which is in principle non-negligible and, in several cases, causes a significant delay to the overall processing time. In the case of streaming, though, data transfer is able to be carried out simultaneously with computation and therefore does not affect negatively GPU's efficiency. The above is apparent in the diagrams, since GPU's speed compared to CPU increases significantly in the case of streaming, thus canceling its disadvantage and revealing its true computational power.

We continue with the comparison between CPU and GPU in the three-dimensional case. In the following diagrams, the image remains at a constant size of $100 \times 100 \times 100$ pixels while the template varies from $2 \times 2 \times 2$ to $8 \times 8 \times 8$ pixels with a step of 1 pixel in each dimension (always cubic). Figure 5.10 records the execution time on CPU (with 16 threads) divided by the execution time on GPU (or in other words the performance ratio). Results for a single image and the streaming of 10 images are both presented.

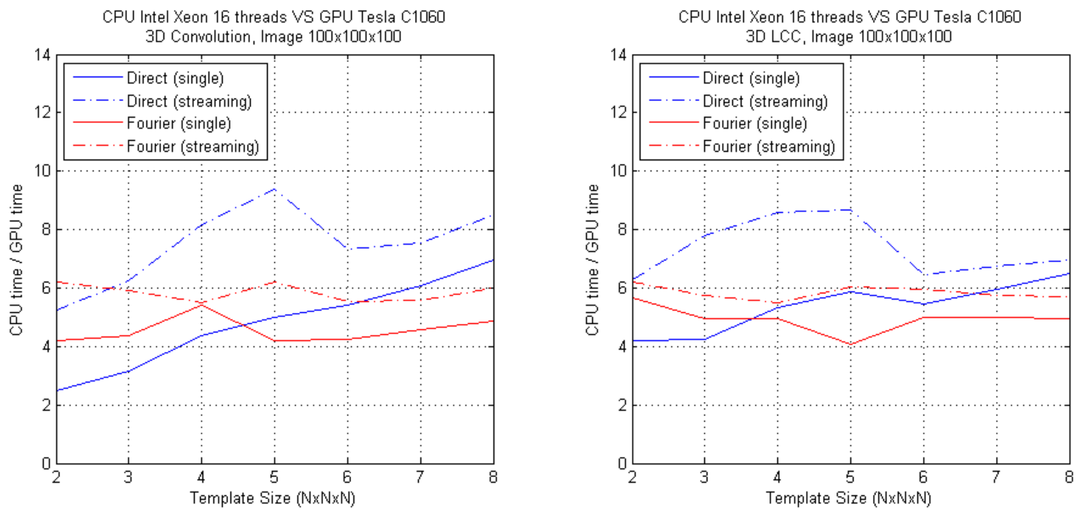

Figure 5.10: Comparison between CPU and GPU for Constant 3D Image

The above results agree with those of the two-dimensional case. As it has been stated, GPU performs better for cases where the ratio of computation over data transfer is relatively high, therefore, as it can be observed, GPU performance increases as the template gets larger. Nevertheless, while in the two-dimensional case the direct method appeared to be up to 14 times faster on GPU than on CPU (without streaming), here the ratio is no more than 7. The Fourier domain method on the other hand is steadily $4 - 6$ times faster on GPU, similarly to the two-dimensional case. Since this value is considered somewhat low for the relative capabilities of GPU, it follows that this method performs better on CPU. This is due on the one hand to the FFT being an algorithm with low ratio of computation over data transfer and on the other hand to the powerful FFTW library supporting its computation on CPU.

Also here it can be seen that streaming cancels GPU's disadvantage on data transfer to/from its global memory, thus increasing significantly its relative performance.

5.2.2 Constant Template – Varying Image

We continue with the two-dimensional case of a **constant template size** and a **varying image size**. We remind that the template is of a constant size of 16×16 pixels while the image varies from 1000×1000 to 4000×4000 pixels with a step of 100 pixels in each dimension (always square). Figure 5.11 presents the execution time on CPU (with 16 threads) divided by the execution time on GPU, for both a single image and the streaming of 10 images.

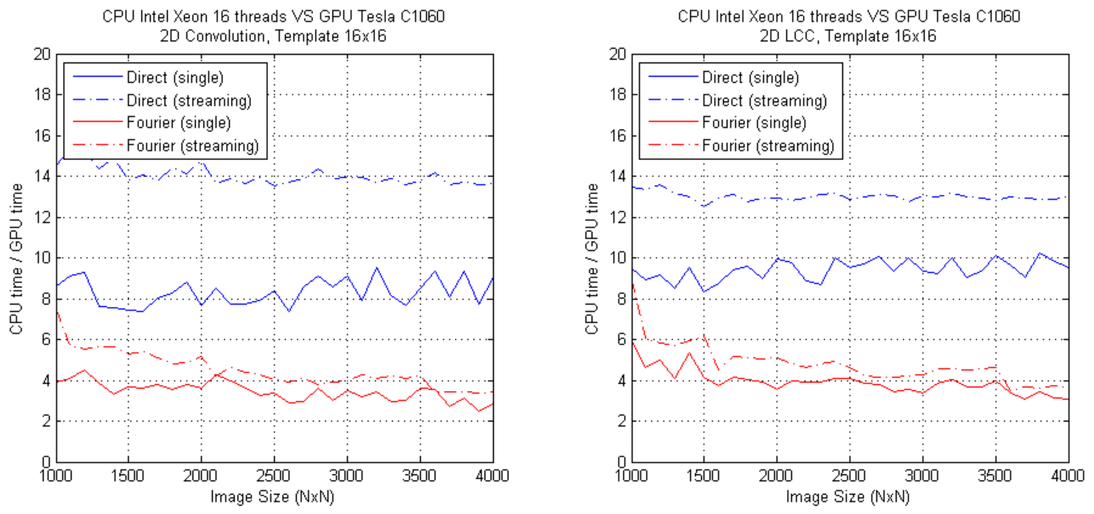

Figure 5.11: Comparison between CPU and GPU for Constant 2D Template

In the diagrams above it appears that the direct method is steadily about $8 - 10$ times faster on GPU than on CPU (for a single image). We see that the relative performance of GPU for the direct method remains practically independent of image size. That means the relative performance of GPU for that method is affected only by template size and not by image size and, as we have already seen, GPU's performance increases as the template becomes larger. On the other hand, the Fourier domain method performs on GPU not as successfully as the direct method, as we have already seen in our previous experiments. Also, we observe that its relative performance decreases at a slow but steady rate as the image becomes larger. This happens because for a larger image, and hence larger transform, the requirements on memory access of the FFTs increase and therefore GPU's work, whose memory as it has been mentioned is its weak point, becomes harder.

Again it appears that streaming, thanks to the elimination of the overhead for data transfer, increases substantially GPU's efficiency, especially in the case of the direct method where the performance almost doubles.

Lastly, we examine the three-dimensional case of a constant template size and a varying image size, where the template is of a size of $8 \times 8 \times 8$ pixels and the image varies from $50 \times 50 \times 50$ to $250 \times 250 \times 250$ pixels, with a step of 25 pixels in each dimension (always cubic). Figure 5.12 presents the execution time on CPU (with 16 threads) divided by the execution time on GPU, for both a single image and the streaming of 10 images.

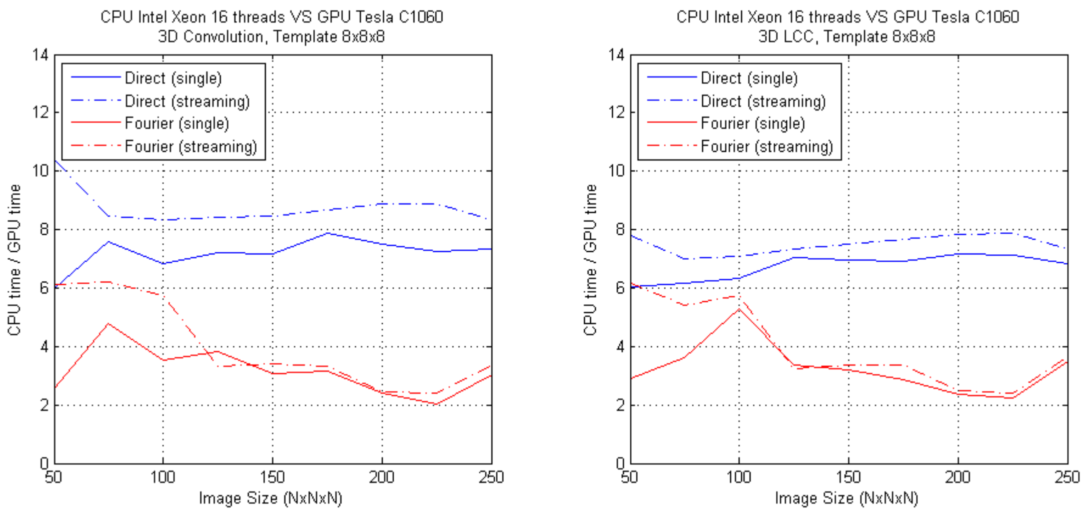

Figure 5.12: Comparison between CPU and GPU for Constant 3D Template

Also here the observations agree with the two-dimensional case. The direct method steadily appears about 7 times faster on GPU, for any image size, while the Fourier domain method is less efficient on GPU and its performance slowly decreases as image size increases. Streaming increases GPU's performance, especially for the direct method.

5.2.3 Number of Threads

Until now we have thoroughly studied the comparison between CPU and GPU, under various circumstances. In this subsection we focus exclusively on the CPU and particularly on the **number of threads** running on it. Up to this point, in all our experiments on CPU we always used 16 threads. The number of threads that the FLCC library uses when it runs on CPU, as it has been said, is determined by the user once and for all during FLCC's installation time and cannot be modified in runtime. It is therefore important to know for what number of threads the library performs best, in order to be in the position to make the most suitable choice.

In the next experiment we examine how the execution time of the various algorithms of FLCC library is affected when they run on CPU with different numbers of threads. For the two-dimensional case we will use an image of 2000×2000 pixels and a template of 16×16 pixels. Figure 5.13 shows the execution times per image, for a single image and the streaming of 10 images. The number of threads progresses geometrically from 1 to 1024, with the intermediate values being successive powers of two.

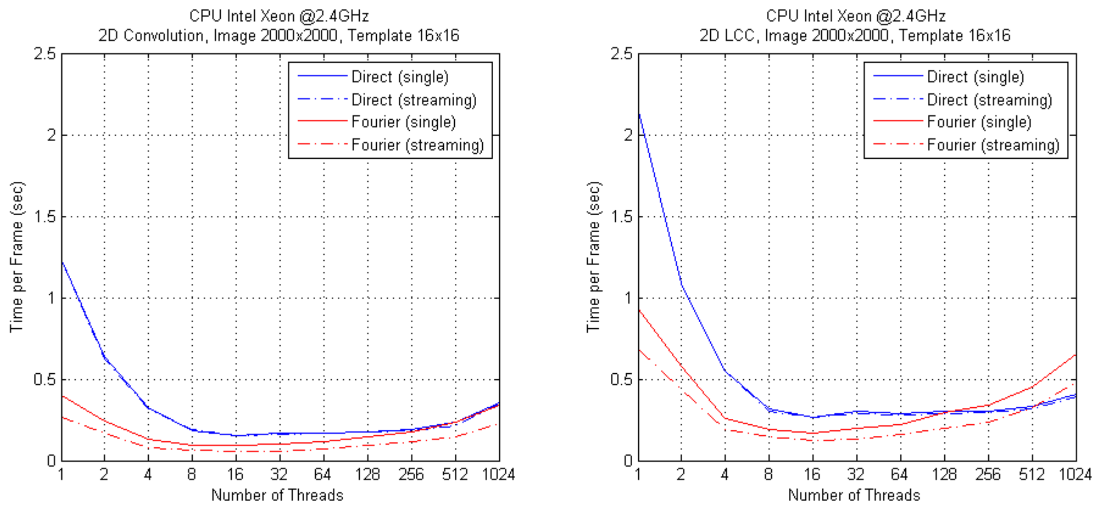

Figure 5.13: CPU Execution Times 2D for Varying Number of Threads

For the three-dimensional case we select an image of size $100 \times 100 \times 100$ and a template of size $8 \times 8 \times 8$. Figure 5.14 presents the execution times per image, for a single image and the streaming of 10 images. The number of threads progresses geometrically from 1 to 1024, with its values being successive powers of two.

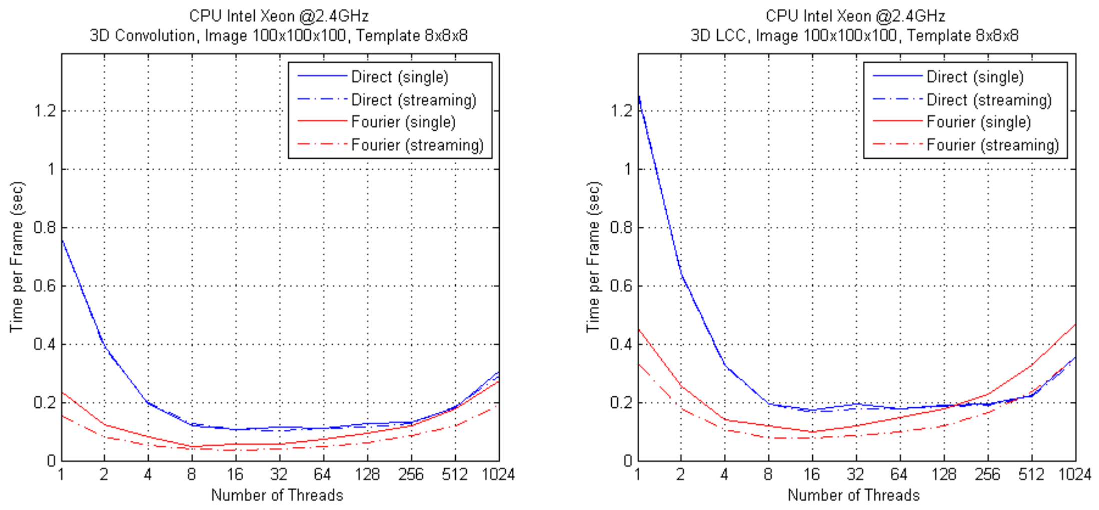

Figure 5.14: CPU Execution Times 3D for Varying Number of Threads

From the above diagrams it follows that both the direct method and the Fourier domain method behave in a similar way as the number of threads varies. In particular, we observe that the execution time, for a number of threads from 1 to 16, appears to be somewhat inversely proportional to the number of threads. For more than 16 threads, the execution time seems to stabilize at a low level before starting to rise again at a slow but steady rate as the number of threads increases.

In order to understand the reasons behind this behavior, we need to take into account that our system's processor contains 8 physical cores with a common memory, which have the ability to operate simultaneously at the same data pool (an architecture of SMP type). Furthermore, this particular processor supports hyper-threading technology and therefore each core is capable of executing two threads in parallel, leading to a number of 16 virtual cores. Thus, for as long as the number of threads remains smaller than the number of (virtual) cores, there is further computational potential not being exploited. Given that the algorithms demonstrate a high level of parallelizability regarding the data to be processed, the inverse proportional relation shown in the diagram is derived. In simpler words, it appears that the computational load is distributed evenly and independently among cores, as long as there are cores available. When the number of threads becomes equal to the number of (virtual) cores, that is when all cores are fully occupied, the execution time reaches its lowest point. From then on, the execution time cannot be further reduced, since there no longer exist computational resources to be utilized. On the contrary, further increase in the number of threads generates a small rise in execution time, due to the accumulated time delay induced by the creation and scheduling overhead of the extra threads.

Based on the experimental results and the above analysis, it follows that the optimal number of threads is essentially the number of independent (virtual) cores of the system or at most a slightly larger one. This way not only the utilization of the hardware resources of the system is maximized but also the overpopulation of threads that would slow the system down is avoided.

5.3 Behavior of Plan-Execute Mechanism

In sections 5.1 and 5.2 we studied the time performance of the FLCC's routines that implement its various algorithms for every possible case. The way the FLCC library works, though, is not as simple as that. Its interface provides the user with access to a level of functions which is implemented on top of the level of the routines implementing the various algorithms. We refer here to the level of functions that implement the library's plan-execute mechanism, which can be seen as the "entry point" of the user in the library. Instead of the user selecting the most suitable routine for the computation of each case, this task is automatically carried out by the library itself, without the user needing to have any kind of special knowledge about the implementation of the various algorithms and their characteristics. This is done via the functions of planner type, which receive the parameters of the computation as an input and return the optimal, in their opinion, methodology of executing the computation, which, among others, includes the selected algorithm and the suggested architecture. From then on, the user only needs to call the function of executor type with the data to be processed and the library executes the computation for them in the optimal way. In this section we will experimentally study the behavior of the library's planning stage, a stage which, as we have seen, constitutes the "connective tissue" among all the various features of FLCC library. In particular, we are interested to record, for a number of different cases, the planning stage's output, namely the selected computation routine, and to find out whether this choice is consistent with the experimental results of our analysis so far.

As we have seen in chapter 4, the planning functions receive as input the dimensions of the image and the template, a parameter that denotes whether we have a single image or a stream of images and another parameter that designates the desired platform for the computation to be executed on. The last one may be CPU, GPU, or either. In any case, the output of the planning function will be an object of plan type which, among others, will contain the selected routine. The aforementioned parameter for platform selection limits the search for an optimal solution to only one of the two available platforms, if its value specifies such, or allows the library to freely select by itself the most suitable platform, if its value is "any of the two". The selection of this parameter's value is up to the user, based on the system available to them and their specific needs.

In the experiments we are going to present in the following subsections, we record the selected computation routine for a multitude of different image and template dimensions, for both a single image and a stream of images. As for the execution platform, we chose to present results separately for each one of the two platforms, in other words to set the platform selection parameter in the planning functions first to "CPU" and then to "GPU" and avoid the value "any of the two". In the last case, given that in our system GPU is in principle more powerful than CPU (as the results of section 5.2 have shown), the GPU would dominate in most cases (if not all), with CPU becoming practically invisible. Of course, in a different system where CPU and GPU were comparable, studying the free selection option for planning would certainly be of greater interest.

5.3.1 CPU Results

In this subsection, we will begin by presenting the results on the **CPU** and in particular the two-dimensional case. Both the template and the image here are chosen to always be square. The template size varies linearly from 2×2 to 32×32 pixels with a step of 1 pixel in every dimension while the image size varies geometrically from 32×32 to 4096×4096 pixels, the intermediate values being successive powers of two. Figure 5.15 presents the selections for the computation routine on the CPU in the form of a color map, in the case of a single image.

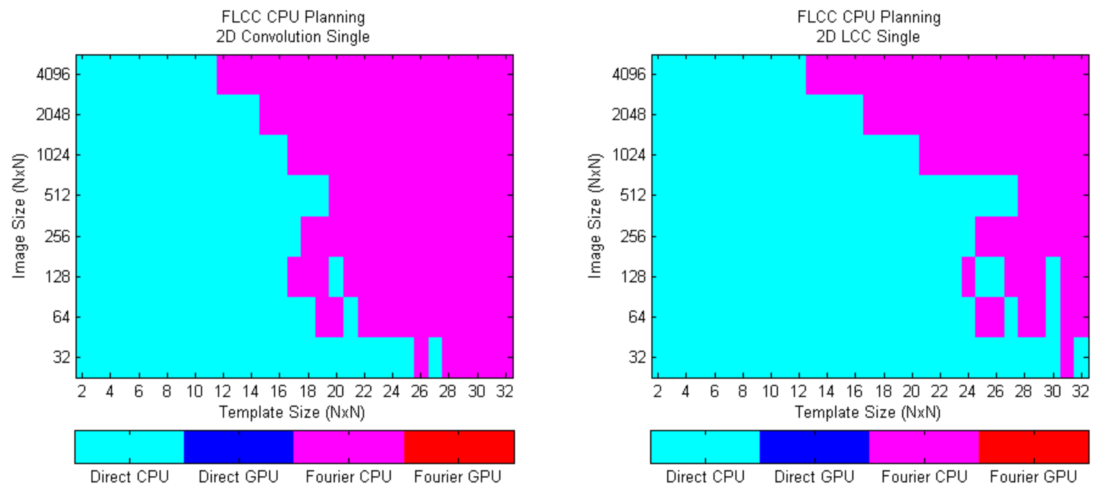

Figure 5.15: Method Selection on CPU for 2D Single Image

Similarly, Figure 5.16 presents the selection on the CPU for the two-dimensional streaming case.

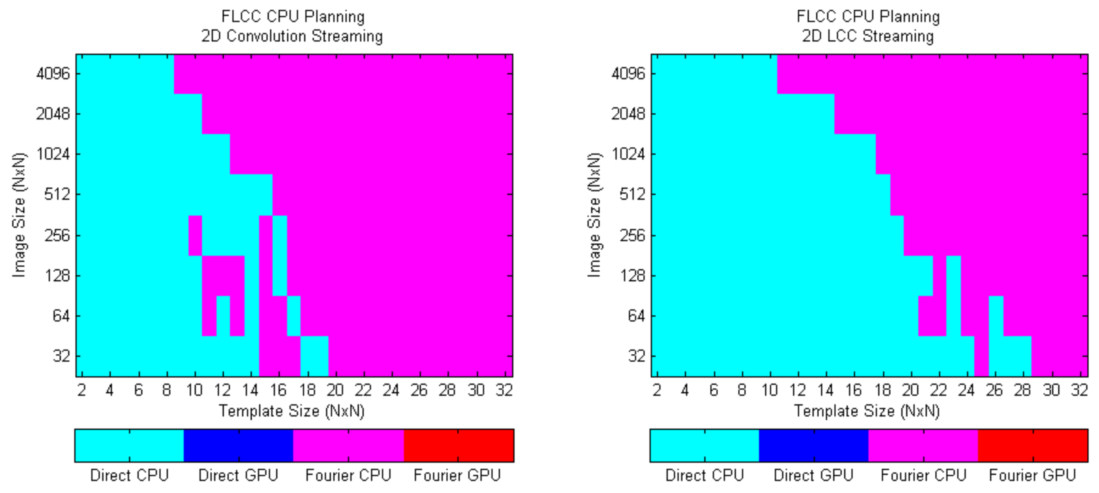

Figure 5.16: Method Selection on CPU for 2D Streaming

We observe that the selection between the direct method and the Fourier domain method agrees both with the theoretical analysis and the experimental results so far. Indeed, it appears that in every case there exists a certain template size (to which we have referred as critical size) which distinguishes the selection between the two methods. For template sizes smaller than the critical size it appears that the direct method is selected while for larger ones the Fourier domain method is selected. Furthermore, we observe that the critical size depends on the image size, as well as on whether we have streaming or not. Also, the critical size is different between convolution and LCC.

It is interesting to compare the results between the case of a single image and the case of streaming. We note that in streaming the Fourier domain method appears “stronger” than in the single image case. This is rather expected if we remember that the Fourier domain method algorithmically benefits from streaming, contrary to the direct method which does not. We shall note here that the planning function of FLCC library, in the case of a single image, selects the optimal method by measuring the time performance of the actual algorithms, leading to results of absolute precision. On the contrary, in the streaming case, the planning function times a modified version of the algorithms which simulates what happens per image. In spite of the latter algorithms not corresponding to an actually meaningful computation, this way the library makes a reliable estimation about the time that would be required for streaming, thus giving itself the possibility to select the optimal method, if not with absolute precision such as in the case of a single image, at least with a fair amount of certainty. This mechanism allows the library to modify its selections between the two cases, so that these selections correspond better to reality, as it is shown by the above experimental results.

We proceed to presenting the results for the three-dimensional case. Both the template and the image here are chosen to be cubic. The template size varies linearly from $2 \times 2 \times 2$ to $8 \times 8 \times 8$ pixels with a step of 1 pixel in every dimension, while the image size varies geometrically from $8 \times 8 \times 8$ to $512 \times 512 \times 512$ pixels, with the intermediate sizes being successive powers of two. In Figure 5.17 we can see the selections for the computation routine on the CPU in the form of a color map, when the selection regards a computation of a single image.

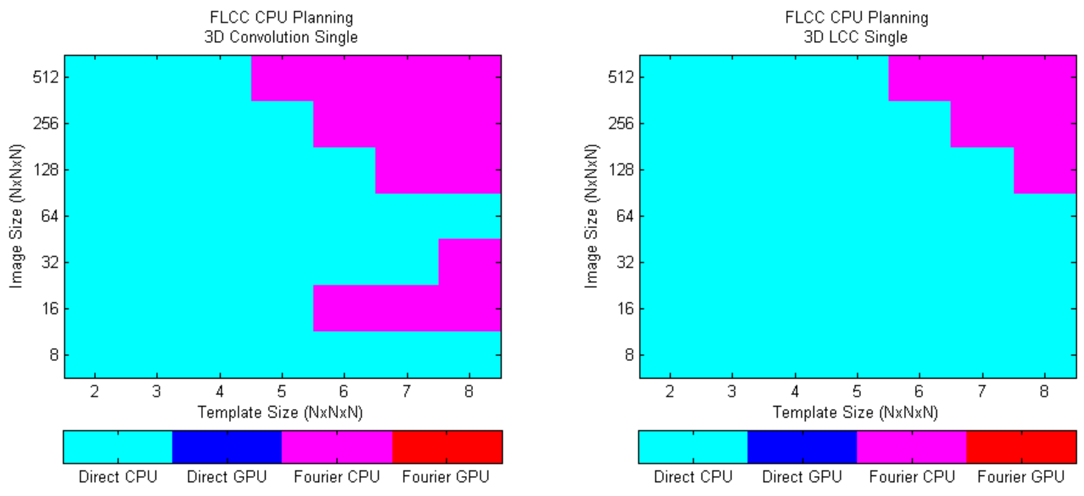

Figure 5.17: Method Selection on CPU for 3D Single Image

Similarly, Figure 5.18 presents the color map of CPU selections for the three-dimensional case of streaming.

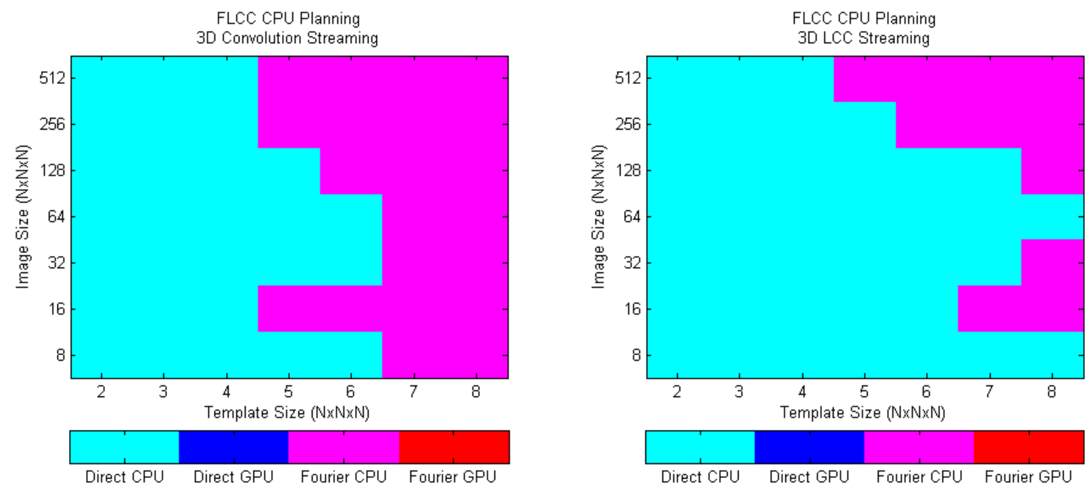

Figure 5.18: Method Selection on CPU for 3D Streaming

The results are quite as expected and agree with those of the two-dimensional case. We see that the direct method is mostly selected for the relatively smaller templates whilst the Fourier domain method is preferred for the larger ones. Furthermore, when the selection regards streaming, we notice that the Fourier domain method appears stronger in comparison with the selection for a single image.

5.3.2 GPU Results

We move on to presenting the FLCC library's selections when we use the **GPU** as the platform for execution. In Figure 5.19 we can see the color map of the planning results for the two-dimensional case of a single image.

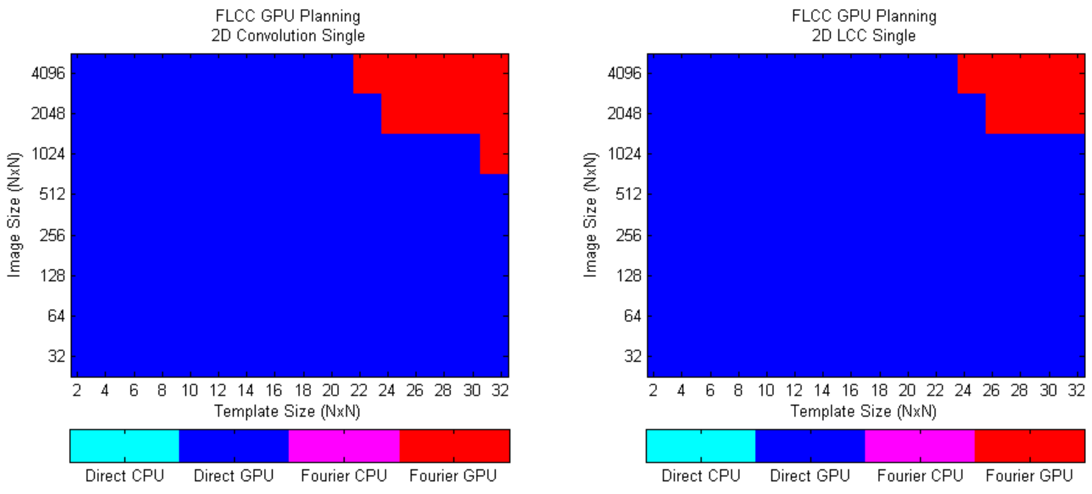

Figure 5.19: Method Selection on GPU for 2D Single Image

Similarly, in Figure 5.20 we can see the color map of the results for the two-dimensional streaming case on the GPU.

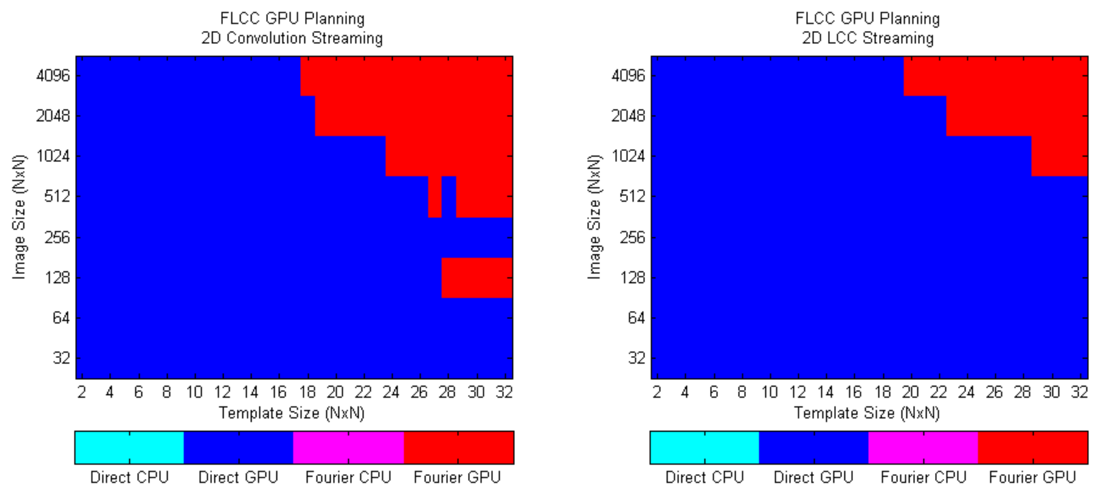

Figure 5.20: Method Selection on GPU for 2D Streaming

GPU's selections reveal a pattern similar to that of CPU. We can see that also here the direct method is preferred for smaller templates while the Fourier domain method dominates for larger ones. It is important to note that, compared to CPU results, the direct method prevails in a much larger part of the map, due to its excellent performance on GPU, as it has also been noted in section 5.2. In the case of streaming, we see that, as it is expected, the Fourier domain method becomes stronger compared to the single image case.

In conclusion, we will present the results for the three-dimensional case when the selection takes place on the GPU. In Figure 5.21 we can see the color map of the selections when they regard a single image.

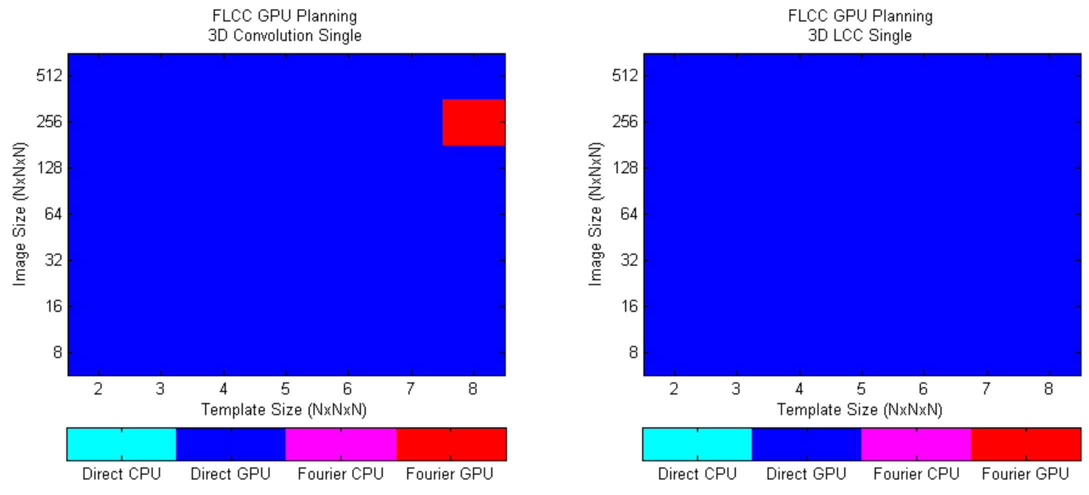

Figure 5.21: Method Selection on GPU for 3D Single Image

Similarly, Figure 5.22 presents the respective color map for the GPU when the selection regards three-dimensional streaming.

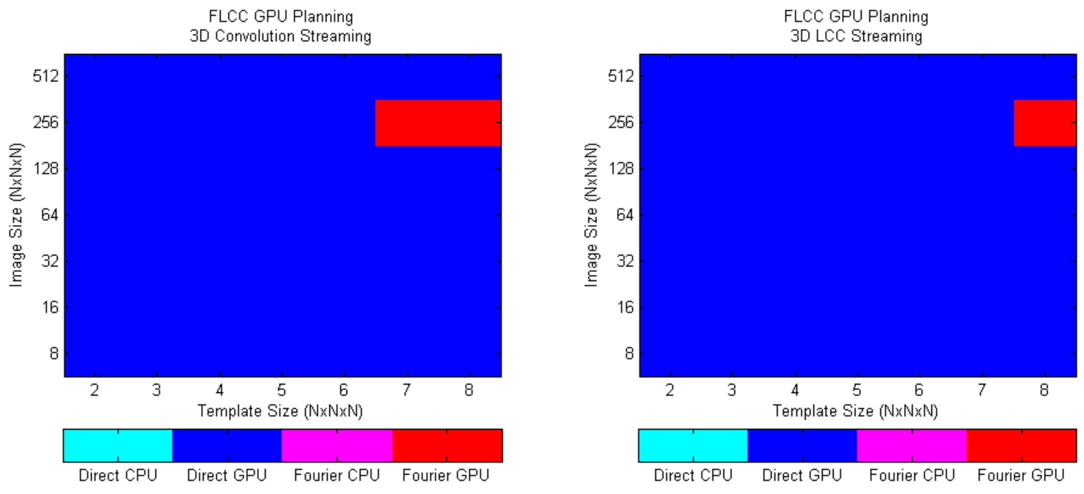

Figure 5.22: Method Selection on GPU for 3D Streaming

From the above results we notice that the direct method strongly overpowers the Fourier domain method on the GPU in comparison with the CPU, since the GPU appears to be more suitable for its implementation. In fact, it seems that in the case of a single image LCC the direct method is selected for the entire set of cases. In the case of streaming, the Fourier domain method is slightly more preferred than in the case of a single image.

A detail in the above diagrams that we would like to highlight is the case where the image is of size $512 \times 512 \times 512$, in which the direct method is always selected while we would expect the Fourier domain method to appear among the selections too. This is due to the fact that in our system the CUFFT library, which computes the FFTs on GPU, crashes for this particular image size, inevitably leading the algorithms using it to fail as well. This failure, though, provides us with the opportunity to demonstrate another interesting feature of the planning functions. In fact, we see that in case some algorithm of those being tested during planning fails, the planning function does not fail but instead it detects the particular algorithm's failure and excludes it from the selection process. Thus, we see that when the Fourier domain method happened to fail, the planning function returned the direct method as the suggested one, so that even in this case the computation be executed unobstructedly and transparently.

6 Conclusions

In this diploma dissertation we extensively occupied ourselves with the subject of computing two operations central in the field of image processing (and signal processing generally). These are: convolution and correlation coefficients, with or without local normalization, between an image and a filter/template. We saw that these two operations are of great importance, as much in theory as in practice. First of all, they are met in a great number of image processing operations and problems, many times comprising basic, integral stages thereof. Because of this, their physical and mathematical importance is indisputable. Indeed, we saw that convolution is at the heart of linear and space-invariant systems (LSI), systems that constitute a large part in signals' and systems' theory generally. Additionally, correlation coefficients are one of the most useful and widespread definitions of the concept of "linear dependence", which has widespread theoretical and practical application. Nevertheless, we ascertained since the beginning of our dissertation that, despite their frequency and usefulness, their computational complexity itself obstructs their application in practice. We noted that both these operations show high computational complexity that stems from their own mathematical definition. Especially when there is need for local normalization of the factors, any naïve means for confronting the issue becomes prohibitive. What is more, the requirements of modern applications stand to worsen the situation. Very often in practice there is the need of real-time data processing (like video processing), that set austere specifications on the time efficiency of each processing operation. Furthermore, a lot of scientific, medical, recreational and other applications require the processing of images with very high definition (like HDTV) or even of three dimensions (e.g. axial tomography), which raise the computational requirements to even further heights.

All the above render the need for solutions for the problem of the efficient computation of convolution and correlation coefficients, critical and imperative. To this direction, this diploma dissertation strives to confront this matter in a holistic way, by examining as much the theoretical methods as the hardware architectures that could efficiently support their computation. According to that line of thought, we developed and analyzed efficient algorithms for the theoretical calculation of the operations under examination and we chose suitable architectures for the practical execution of the algorithms. Furthermore, we proceeded in combining the above under a practical scheme and we produced a complete solution for the problem, the FLCC library.

Let us begin with an overview of the algorithms. We used two basic methodologies that we called "direct method" and "Fourier domain method". The former attempts the computation of convolution and correlation coefficients in the domain of space, efficiently interpreting the mathematical definition. The latter completely transfers the computation in the frequency domain, achieving in any case a significant decrease of the computational complexity. We managed to adapt our methodology as much to convolution as to correlation. It is noteworthy that, regarding the latter, the methods’ application is achieved even in the difficult case of local normalization. Indeed, even when performing normalization, the direct method may still calculate the result in only one pass, while the Fourier domain method is still able to execute the full computation in the frequency domain, exactly as it happens without the normalization. Thus, the increase in complexity that is induced by local normalization is noticeable only on the constant factors and not on its rank. We did not stop there though. By taking into account the special characteristics of signals representing images, we introduced two further improvements in the Fourier domain method. The first one refers to the FFTs’ computation, which can be done in half the time by taking into account the fact that signals representing images are always real-valued. The second one has to do with the introduction of the idea of image streaming, i.e. the processing of a series of images with the same template. By taking this into account, the total processing time may be reduced by nearly one third. Finally, by examining their arithmetic complexities, we saw that the efficiency of the direct method is significantly dependent on the template’s size, contrary to that of the Fourier domain method which is dependent respectively little or not at all. This has led us to the conclusion that the template’s size is the basic factor that should determine the suitability of each method, the direct method being more suitable for smaller templates while the Fourier domain method for larger. We have supported the above conclusion by calculating exactly those “critical sizes” of a template that determine the preference bound between the two methods.

The next logic step for confronting the problem was to search for those hardware architectures that could sufficiently support the previously analyzed algorithms. We saw that the general computation problem expresses grand parallelizability of data, i.e. of the images. Indeed, we ascertained that in the direct method the full computation may be divided even on pixel level while the Fourier domain method has some points where a similar division may be achieved. On the other hand, even though the basic computational stage of the Fourier domain method, i.e. the FFT, cannot be directly parallelized, it can be supported satisfactorily in a parallel implementation by lots of existing software packages. The above have led us naturally to a search for an architecture of SIMD type, i.e. an array processor. Due to its wide availability today and the relative easiness of its programming, the GPU was chosen as one of the two basic platforms we use, as it poses a modern, practical and rather powerful solution. The second basic platform is the multi-core processor, a classic choice, widely available today, even in personal computers. The use of the second choice was deemed necessary, as although the GPU’s availability is as of now wide, one is not available in every computer system, in contrast to a CPU of one or more cores.

Having developed efficient algorithms and chosen suitable architectures, we proceeded in the combination of all into a practical and yet clever scheme, that lead to the creation of the FLCC library. This library incorporates all the algorithms and architectures that have been examined, in a way transparent towards the user. A basic element in its design is the use of the plan-execute model, according to which the library chooses automatically the suitable methodology (algorithm-architecture combination) for the execution of the operation required by the user, without having the latter need to know how exactly this is being managed. The basic advantage of this design is that the choice (planning stage) has to be done only once per case, while its result may be used multiple times in different calculations (execution stage), something that leads to the noteworthy efficiency of the overall execution. Apart from this model, the library uses a multitude of other technical elements in order to improve efficiency. It uses internally the libraries FFTW and CUFFT for the execution of FFTs on CPU and GPU respectively. Those are libraries with a proven exquisite time performance. The planning stage required by those two libraries is incorporated in the planning stage of FLCC itself in a way that does not intrude in the least on the execution stage. The calculation of the FFTs is further accelerated by suitably expanding the images to dimensions that lead to fast transforms, so as to achieve efficient processing even of images with “difficult” dimensions (as in prime numbers). For the direct method a code generator was developed that generates highly optimized routines (full loop unrolling, computation minimization, optimized memory usage) leading to decisive optimizations in its execution time. Memory usage has a special part in the library’s efficiency. On GPU there has been an extensive use of shared memory that accelerates the direct method’s computations. Furthermore, special functions have been implemented in the interface of the library for the allocation and deallocation of memory on RAM, ensuring in a user-transparent way the efficiency of the data transfers. Finally, streaming was implemented in a way such that the time required to transfer the data to the GPU’s global memory be possibly nullified, by overlapping the image transfer with their processing, using a pipeline logic. All the above are combined under a small, simple and easy-to-use interface, providing the user with a software package ready for immediate use, without their need to know the least regarding the minutiae of its internal functions.

Our work could not have possibly been complete without the experimental confirmation of all that have been described previously. We designed and performed an extensive series of experiments, aiming to cover all the individual aspects of the FLCC library, i.e. the algorithms, the architectures, the design and the implementation. Our goal was to test the degree to which the FLCC library satisfies its specifications and comprises a satisfactory solution to the problem of computing convolution and correlation coefficients. We ascertained that the execution times confirm the theoretical estimations and justify the choice of architectures. Specifically, the time efficiency of the developed algorithms was highlighted, the acceleration attained by the powerful architectures was demonstrated and the functionality and purposefulness of the basic design idea of the library, the plan-execute model, that similar to a web combines all the individual elements into a unified functional entity was, indeed, proven.

Concluding our endeavor, the final conclusion we could state is that the FLCC library constitutes an efficient, competitive and easy-to-use catholic solution to the problem of computing convolution and correlation coefficients that achieves its goal not by sacrificing the integrity of the calculations but by combining efficient algorithms and powerful architectures under a clever and practical scheme.

7 Future Work

An endeavor as wide as this one could not easily be completed solely in a diploma dissertation. Until now we have thoroughly described FLCC library and have presented its features and capabilities. In this chapter, on the contrary, we will rather focus on its weaknesses and constraints. Our purpose is to search for those directions that our work from now on should follow, keeping in mind its constant improvement and expansion. We will discuss the three basic components of our work separately, i.e. the algorithms, the architectures and the FLCC implementation itself.

We begin with the algorithms. In this thesis two basic algorithmic methods were developed, namely the direct method and the Fourier domain method. We deliberately though left aside the examination of a third method, no less promising than the other two. The latter is the method of "separable convolution – correlation". This method can be applied in the case where the template is a separable function, that is, it can be expressed as the product of one-dimensional functions, their number being equal to the dimensionality of the template. After that, convolution – correlation may be fully calculated as a series of successive convolutions – correlations where the one-dimensional factors of the original template serve as the new templates. This method can lead to particularly low computational complexity. Its basic flaw, though, is that it is applicable only when the template is separable. Yet, even in this case, there is the possibility to expand its applicability to the general case if we take into account that every template, via a decomposition method such as the Singular Value Decomposition (SVD), can always be expressed as a finite sum of separable templates. Therefore, given that convolution and correlation are linear operations, it would be enough to perform "separable convolution – correlation" with each of the separable templates and then pointwisely add the results. We see now that, contrary to the methods already used in this thesis, the performance of "separable convolution – correlation" is highly dependent on the special characteristics of the template and in particular on the minimum number of separable templates the latter can be decomposed in. Nevertheless, the separable method could serve as a promising alternative in many cases and thus it would be of great interest to investigate its possibility of being incorporated in FLCC library.

A second algorithmic improvement that would be worth to explore and possibly incorporate in the library is the usage of modified FFTs in the implementation of the Fourier domain method. The reason why this could be algorithmically beneficial is that, as we have seen, FFT results are themselves of no value whatsoever, serving merely as intermediate and temporary results. This means that we are not interested in the correctness of the transform itself but we only use it as a mere computational tool. The above observation offers the opportunity of modifying the FFT in such a way that allows its faster calculation, even without producing the correct result, provided that this modification maintain the validity of convolution – correlation theorem. This improvement in the Fourier domain method could be quite important. Its disadvantage, which is the reason why this suggestion was left out of this thesis, is that it prohibits the usage of ready-to-use packages of optimized software that perform FFTs with notable efficiency, such as the FFTW and CUFFT libraries that we chose to utilize, and therefore makes it necessary to implement the modified transforms from scratch. It does not cease to be an interesting alterative to be explored though.

We will now discuss our second subject, architectures. It is a fact that new hardware platforms are designed and become available on a constant basis, while the old ones evolve and improve all the time. Therefore, a software package, in order to remain up-to-date, needs to follow hardware evolution. Moreover, it is always interesting to transfer the software to a different, already existing and successful platform, since it expands its application field. With that in mind, FLCC library could benefit by the exploration of new architectures to be transferred to or by the greater utilization of those that already supports. For instance, for now FLCC library does not support an environment for distributed computation, such as a network of computational units (e.g. of MPI type). It is also noteworthy that the library can utilize only one GPU at a time while it would be important (and rather possible) to be able to take advantage of the computational power of more than one simultaneously, provided they were available in the system. In any case, the platforms already supported, and especially GPU, are rapidly developing architectures and consequently the library ought to follow their evolution and exploit them accordingly.

Finally we will focus on the design and implementation of the FLCC library itself. It is a fact that the potential for improvement is something that follows every software package, even long after its birth. FLCC could not have been an exception to that. Its ambitious aim makes it vulnerable to weaknesses and constraints. We will here refer to version 1.3, which is the most recent up to the writing of this thesis. Even though the capabilities of this version are plenty, as then have been presented in detail, limitations are always present. For instance, we have already seen that the library offers the possibility for the processing of a stream of images, a feature of great significance, since it allows for a substantial increase in performance. Nonetheless, this possibility is limited to images of the same size, which are stored in successive memory locations and are all available in RAM before the start of the processing. Contrarily, several times in practice applications require streaming among images of various sizes, which might be stored in arbitrary memory locations or, even worse, become available meanwhile processing (such as in a real-time application). Another limitation regards the number of templates. FLCC library's functions for now support convolution/LCC with one template at a time. Nevertheless, there exist applications requiring convolution/LCC with multiple templates which might as well be very different to one another, a feature not yet supported in the library.

Having discussed the existing limitations, we now proceed to suggestions on expanding the functionality of FLCC library. Let's begin with the simplest, i.e. the image dimensionality. Although typical images are two-dimensional or three-dimensional, an expansion of the FLCC operations to other dimensionalities (such as the one-dimensional case) or even the support of an arbitrary number of dimensions could broaden its application field. The same is true for the numerical precision of its calculations. Until now our first priority has been the computation speed and thus only single-precision arithmetic has been used, which is well-known to lead to faster computations on the one hand but to greater numerical errors on the other hand. Expanding to double-precision arithmetic could serve to applications where minimization of numerical errors plays a crucial role (such as medical-related applications) and the resulting decrease on speed is tolerable. As for the padding process on images, we have seen that the library in every calculation assumes the image to expand to every direction in zeros. In various applications it is desirable that this expansion be realized with values other than zero, such as with continuous repetition of the marginal values or periodical repetition of the whole image. Thus, it would be useful that this functionality be offered by the library's functions and that the user be able to select a padding type according to their needs. Even more, it is possible that the user need only calculate a fraction of the whole convolution/LCC, such as the one where the overlapping between the image and the template is full (without the need for padding) or the one having the same size as the original image. Although the library calculates in each case the full result and therefore the user may always isolate the fraction of their interest, it would be useful if it were possible to calculate solely the fraction needed, so that no time be wasted on redundant calculations. A different and rather more ambitious expansion would be to incorporate in FLCC new operations, such as the calculation of local maxima – minima or local sums – averages. The common characteristic of the above operations with convolution/LCC is that all of them are based on the locality of calculations, which in fact governs the philosophy of the whole library. In a more long-term level, the library could be expanded to a generic library of local operations with a wide range of applications. Finally, regarding a more technical issue, the library could be transferred to other operating systems, such as MS Windows (without the need for Cygwin) or Mac OS. For now, the library only supports UNIX/Linux or MS Windows, via Cygwin, systems. Such an expansion could substantially widen its functionality spectrum.

In conclusion, we would like to highlight that any weaknesses, limitations or expansion possibilities of FLCC library do not cancel its functionality and its various capabilities. On the contrary, they constitute opportunities for evolution, exploration and improvement. After all, constant improvement and updating of a software package is the only way that ensures not only its efficiency but also its longevity.

8 Appendix

In this appendix we will repeat the entire set of experiments of chapter 5, this time on a different computer system with a multi-core processor (CPU) and a Graphics Processing Unit (GPU). Table 8.1 and Table 8.2 describe the features of the new CPU and GPU respectively.

Table 8.1: CPU's Technical Features

Model	AMD Opteron 6168
Clock speed	1.90 GHz
Number of cores	2×12
Cache memory size	512 KB
RAM size	64 GB
Operating system	GNU/Linux

Table 8.2: GPU's Technical Features

Model	NVIDIA GeForce GTX 480
Clock speed	1.40 GHz
Number of cores	480
Shared memory size	48 KB/block
Global memory size	1.5 GB
Driver program	CUDA Driver v4.0

In order to make a quick initial comparison between CPU and GPU, we will use the same index as in section 5.2. Our CPU has 24 cores and a clock speed of 1.90 GHz, while our GPU has 480 cores and a clock speed of 1.40 GHz. Therefore we get:

$$\frac{480}{24} \cdot \frac{1.40\, GHz}{1.90\, GHz} \cong 14.767 \qquad \textbf{(8.1)}$$

This value indicates that GPU should typically appear to be 14.767 times faster than CPU.

In the rest of this chapter we will present in the form of diagrams all the results obtained by the re-execution of the experiments of chapter 5. We will simply list the results here without commenting on them. The purpose of this chapter is not to try to understand the functionality of FLCC (this has already been done in chapter 5) but to reinforce our experiments with further data, to include a second computer system in them and to contrast FLCC's behavior between two different systems. The reader may refer to chapter 5 for a detailed description of the experiments and a thorough analysis of their results.

As in chapter 5, the presentation of the experimental results is divided in three sections: comparative performance of the various algorithms (8.1), comparative performance of the various architectures (8.2) and behavior of the plan-execute mechanism (8.3).

8.1 Comparative Performance of Algorithms

8.1.1 Constant Image – Varying Template

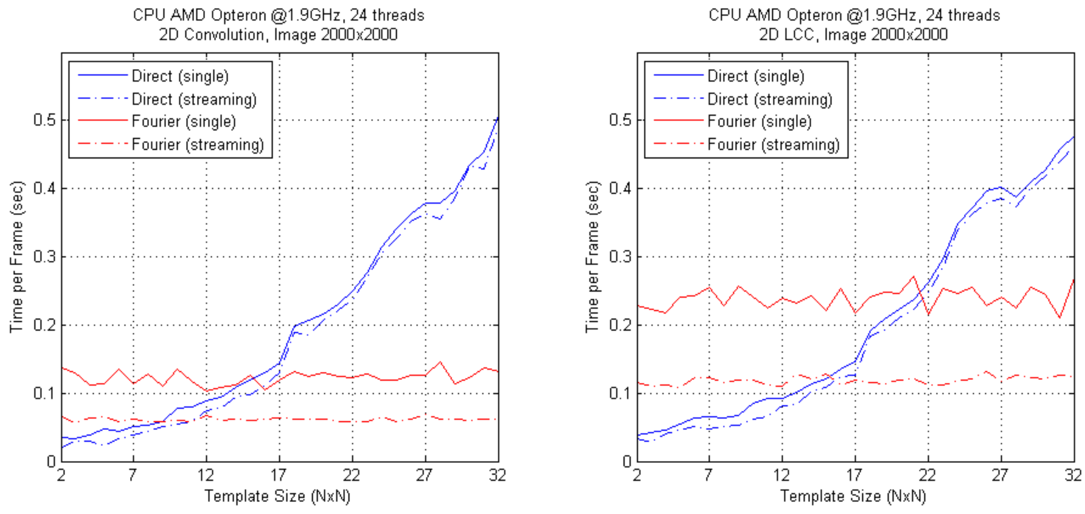

Figure 8.1: Execution Times on CPU for Constant 2D Image

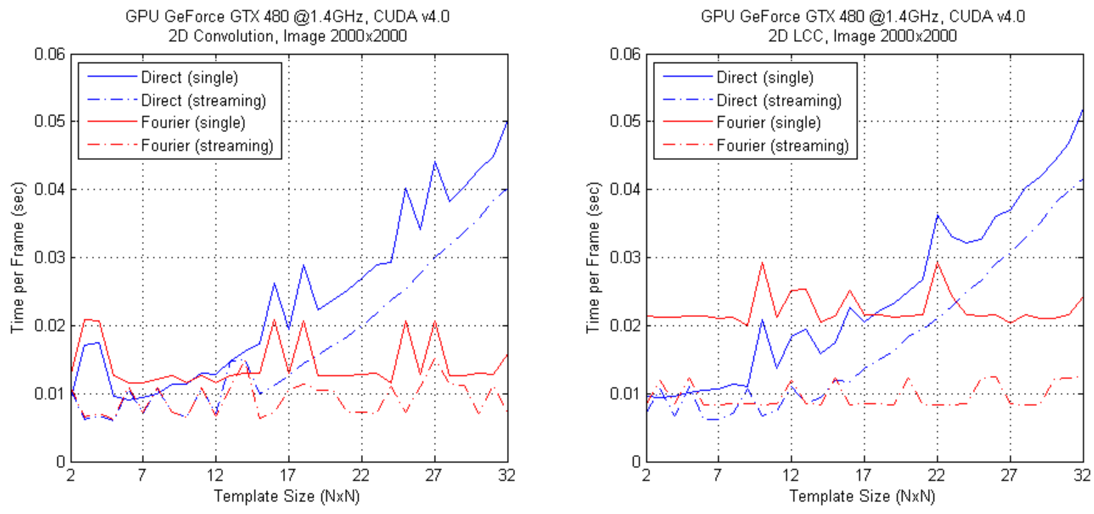

Figure 8.2: Execution Times on GPU for Constant 2D Image

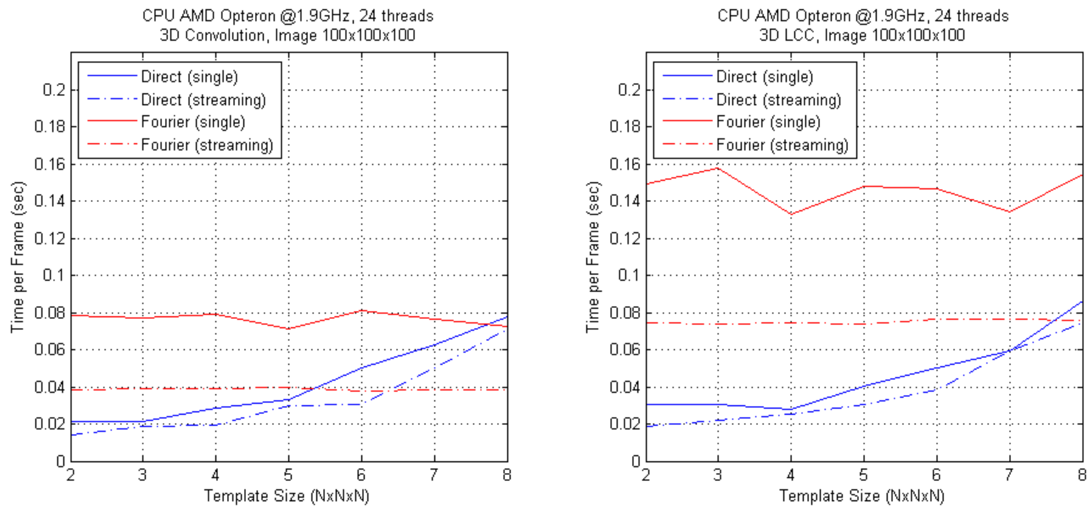

Figure 8.3: Execution Times on CPU for Constant 3D Image

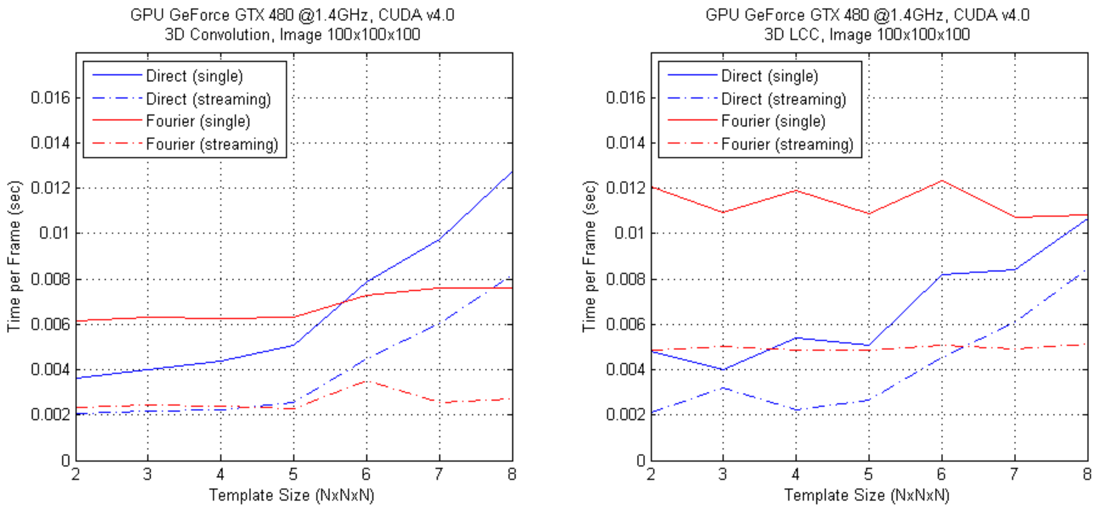

Figure 8.4: Execution Times on GPU for Constant 3D Image

8.1.2 Constant Template – Varying Image

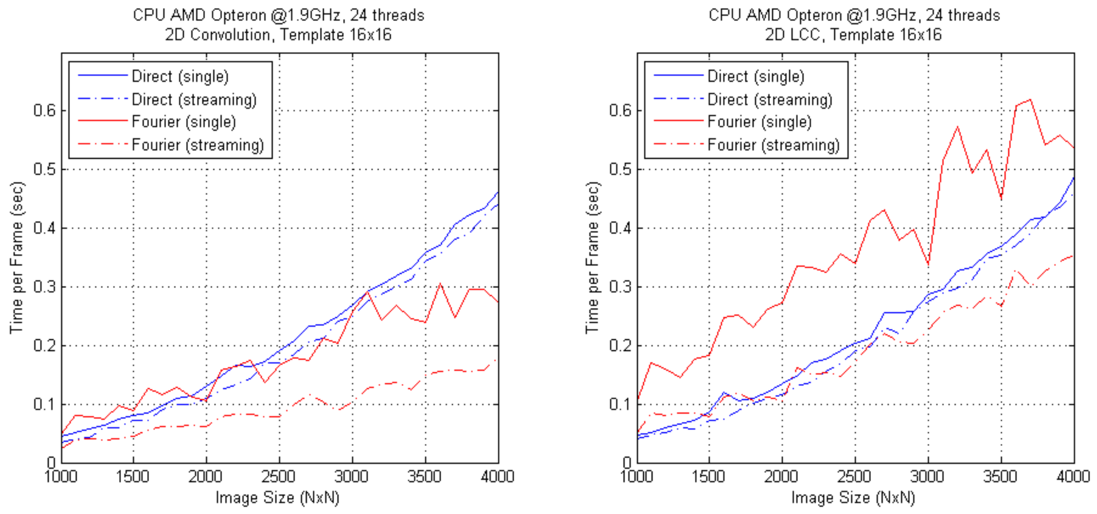

Figure 8.5: Execution Times on CPU for Constant 2D Template

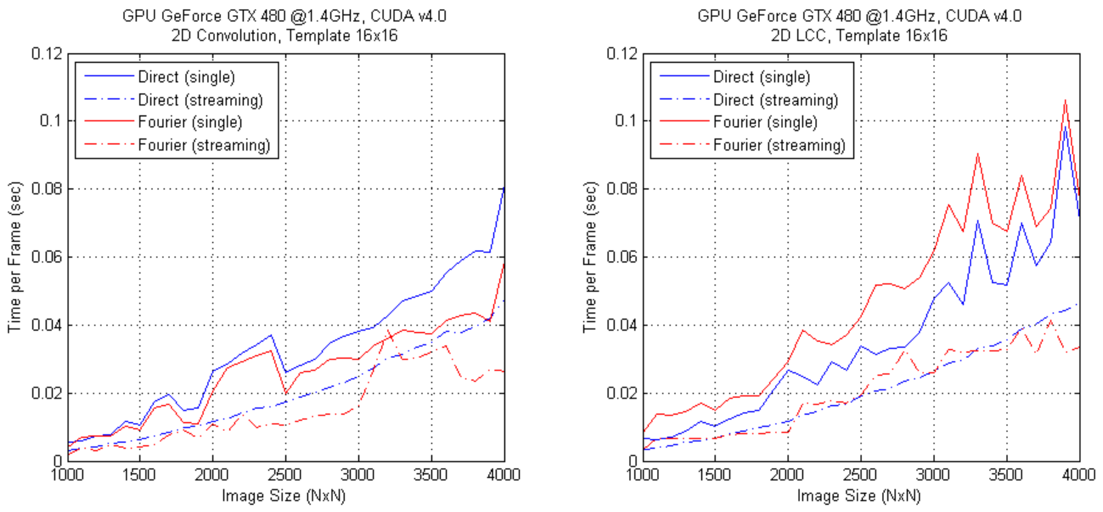

Figure 8.6: Execution Times on GPU for Constant 2D Template

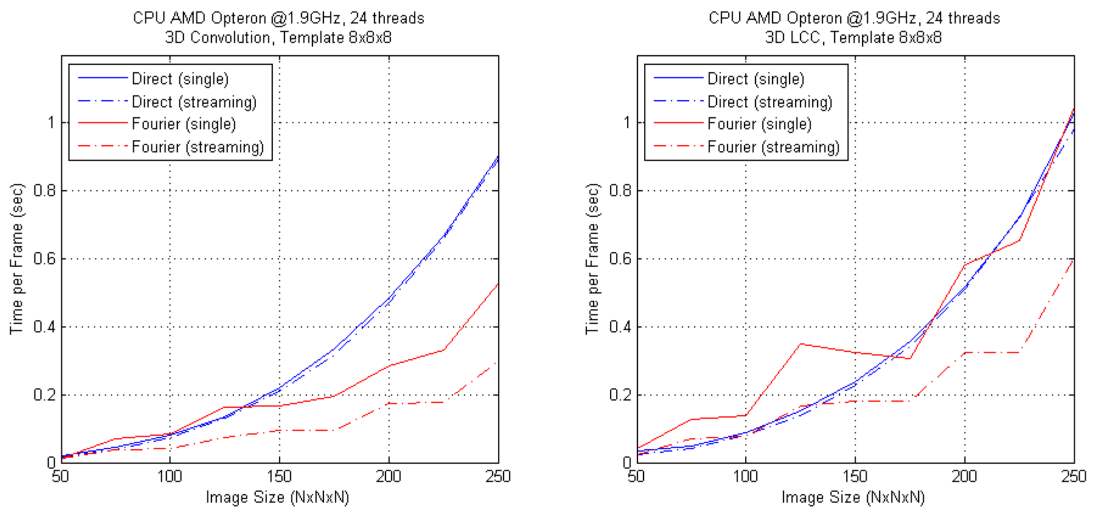

Figure 8.7: Execution Times on CPU for Constant 3D Template

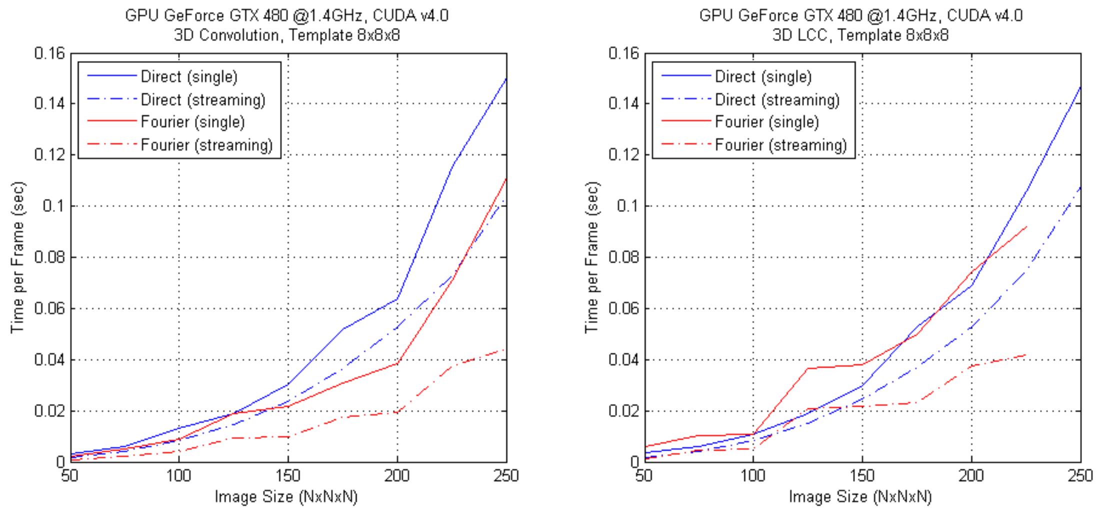

Figure 8.8: Execution Times on GPU for Constant 3D Template

We shall note here that in Figure 8.8 and in particular in the case of LCC, there does not seem to exist a result for the Fourier domain method (for both the single image and the streaming case) and for the image size of $250 \times 250 \times 250$. This happens because the memory requirements of this particular computation exceed the GPU capabilities of our system, hence the computation inevitably fails.

8.2 Comparative Performance of Architectures

8.2.1 Constant Image – Varying Template

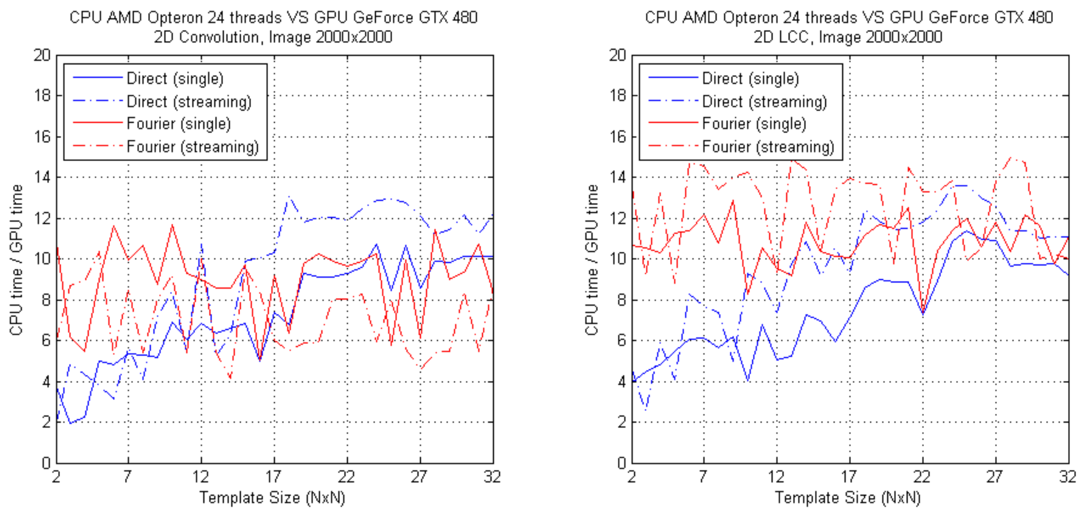

Figure 8.9: Comparison between CPU and GPU for Constant 2D Image

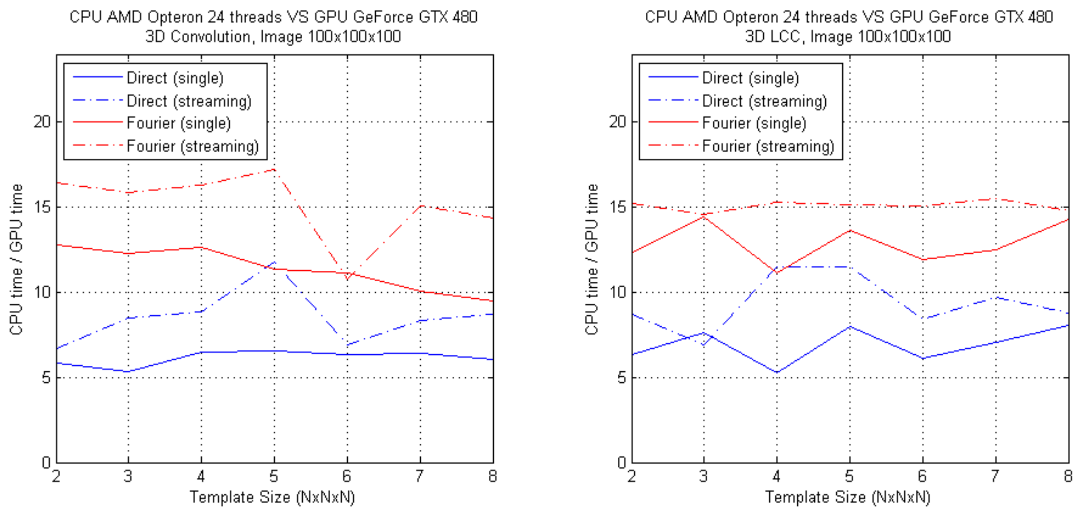

Figure 8.10: Comparison between CPU and GPU for Constant 3D Image

8.2.2 Constant Template – Varying Image

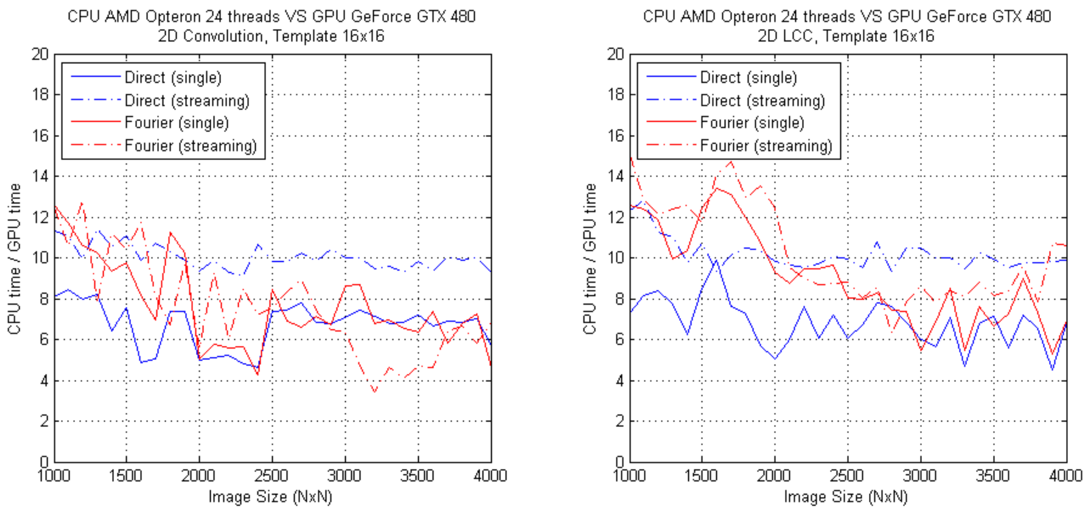

Figure 8.11: Comparison between CPU and GPU for Constant 2D Template

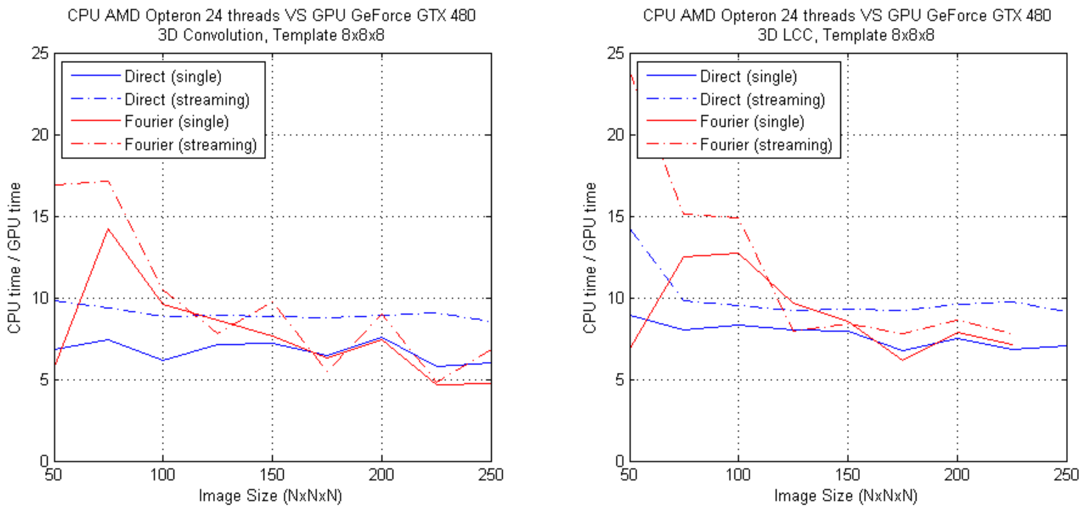

Figure 8.12: Comparison between CPU and GPU for Constant 3D Template

In Figure 8.12, for the case of LCC via the Fourier domain method, there does not exist a result corresponding to the image size of $250 \times 250 \times 250$, since, as we have already mentioned, this particular computation fails on our system's GPU due to insufficient memory size.

8.2.3 Number of Threads

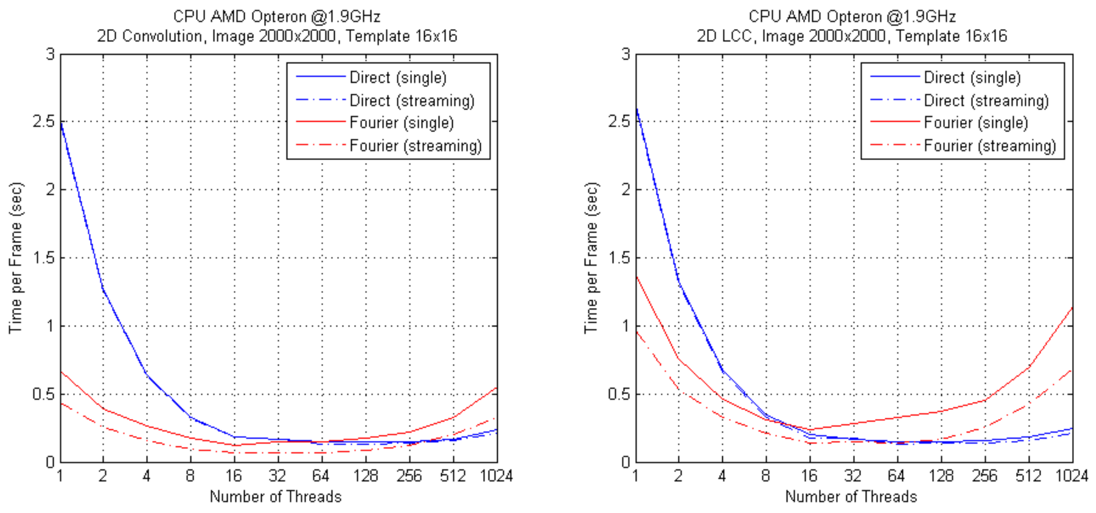

Figure 8.13: CPU Execution Times 2D for Varying Number of Threads

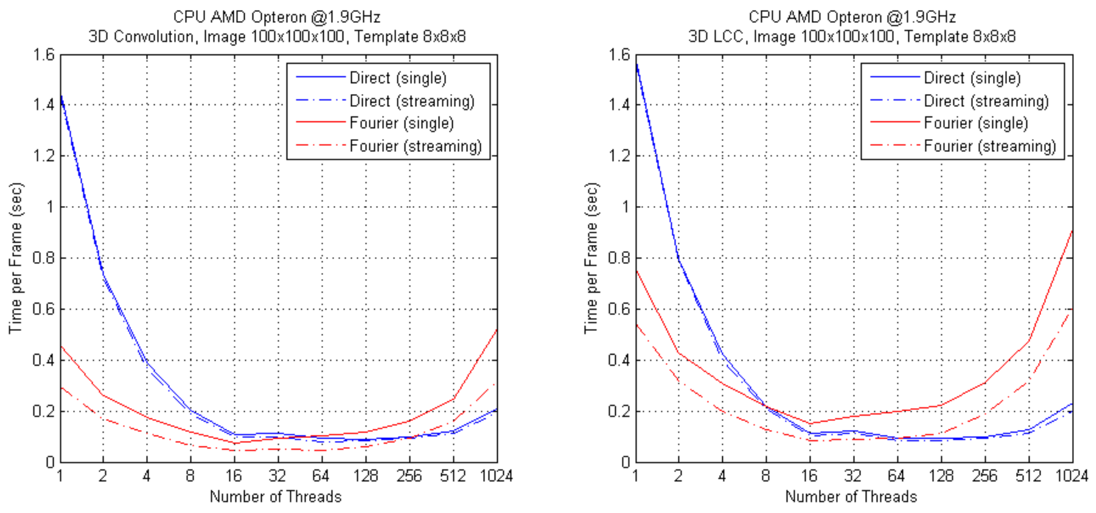

Figure 8.14: CPU Execution Times 3D for Varying Number of Threads

8.3 Behavior of Plan-Execute Mechanism

8.3.1 CPU Results

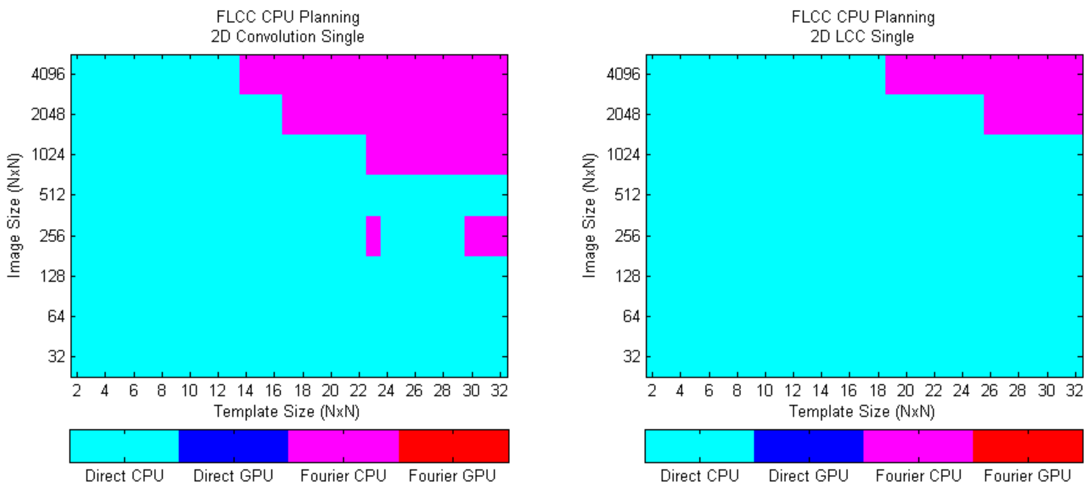

Figure 8.15: Method Selection on CPU for 2D Single Image

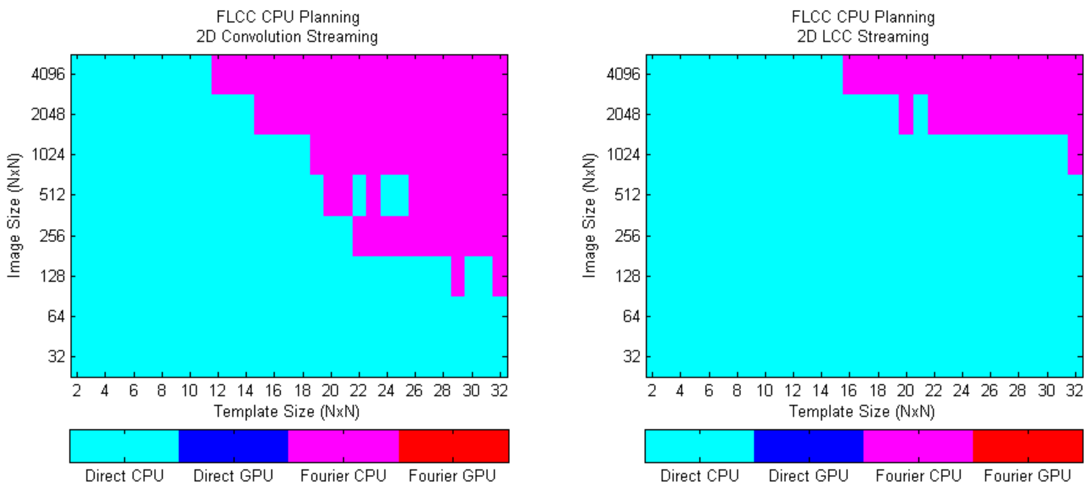

Figure 8.16: Method Selection on CPU for 2D Streaming

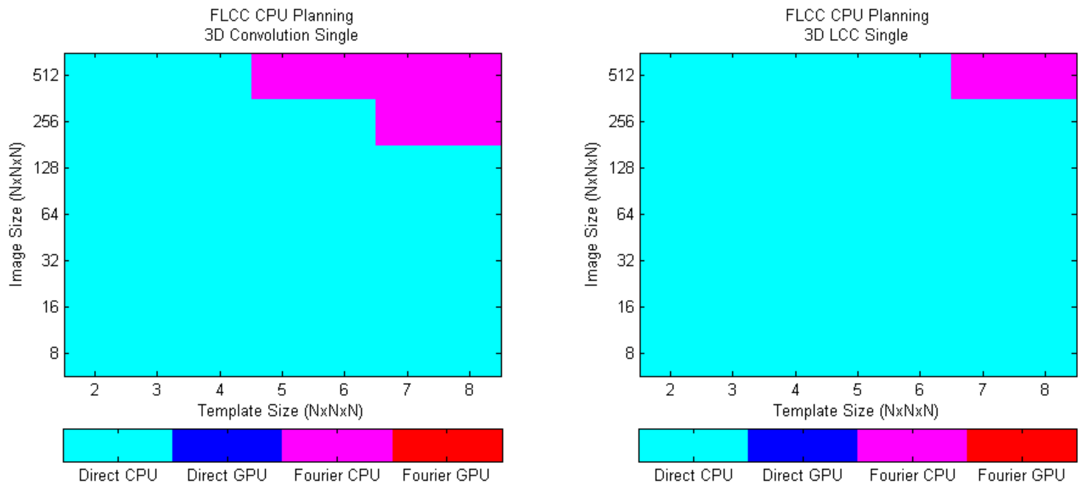

Figure 8.17: Method Selection on CPU for 3D Single Image

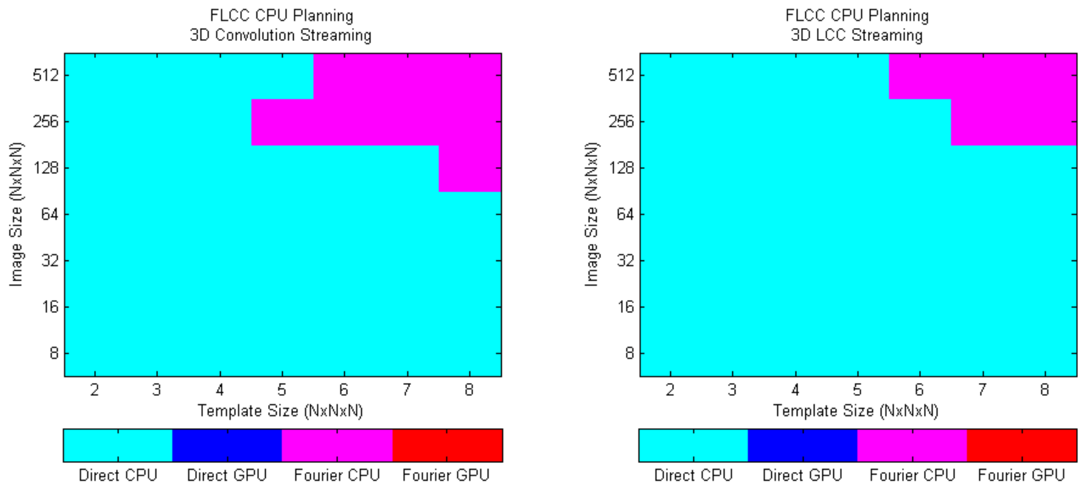

Figure 8.18: Method Selection on CPU for 3D Streaming

8.3.2 GPU Results

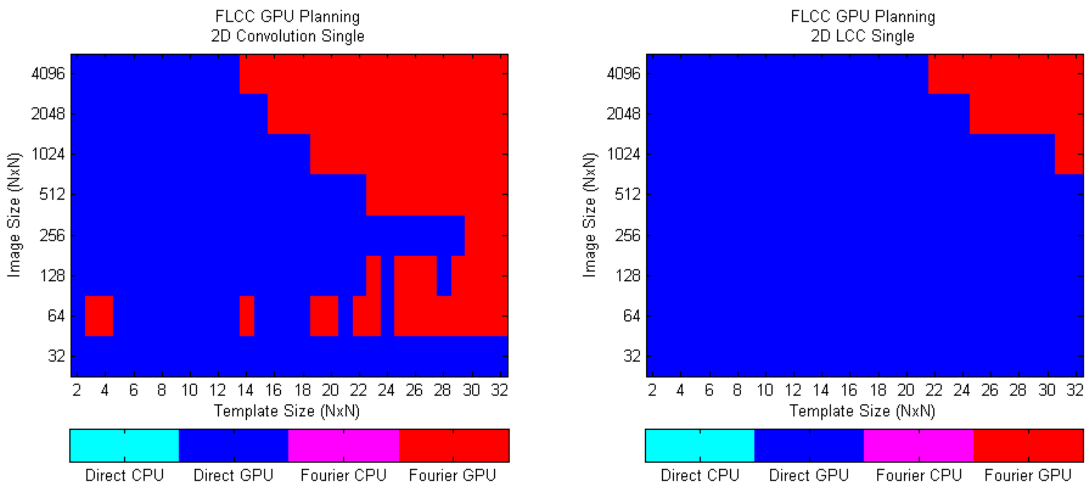

Figure 8.19: Method Selection on GPU for 2D Single Image

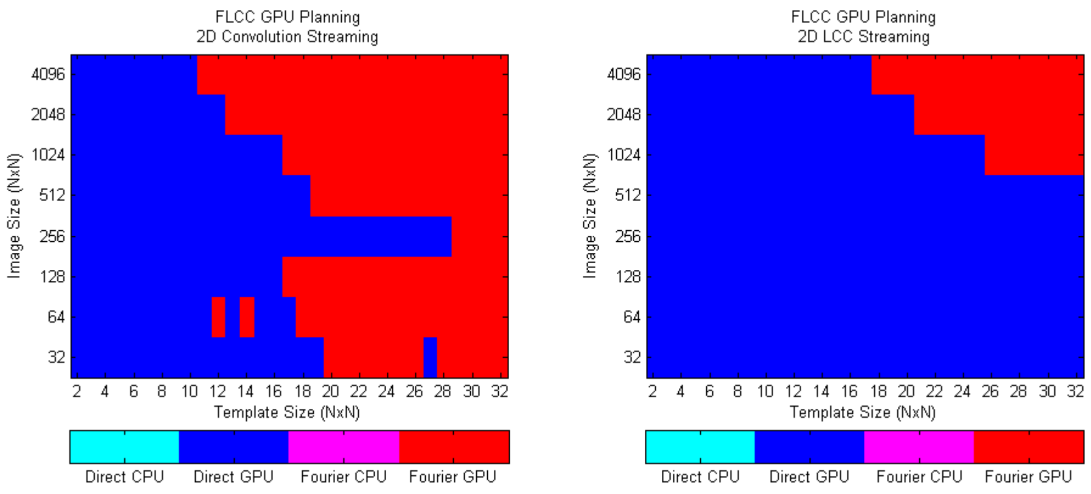

Figure 8.20: Method Selection on GPU for 2D Streaming

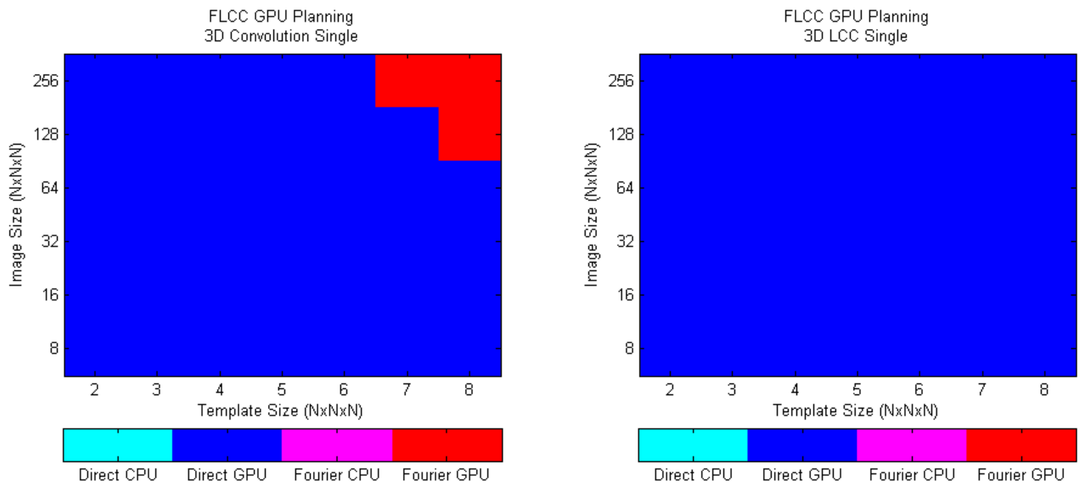

Figure 8.21: Method Selection on GPU for 3D Single Image

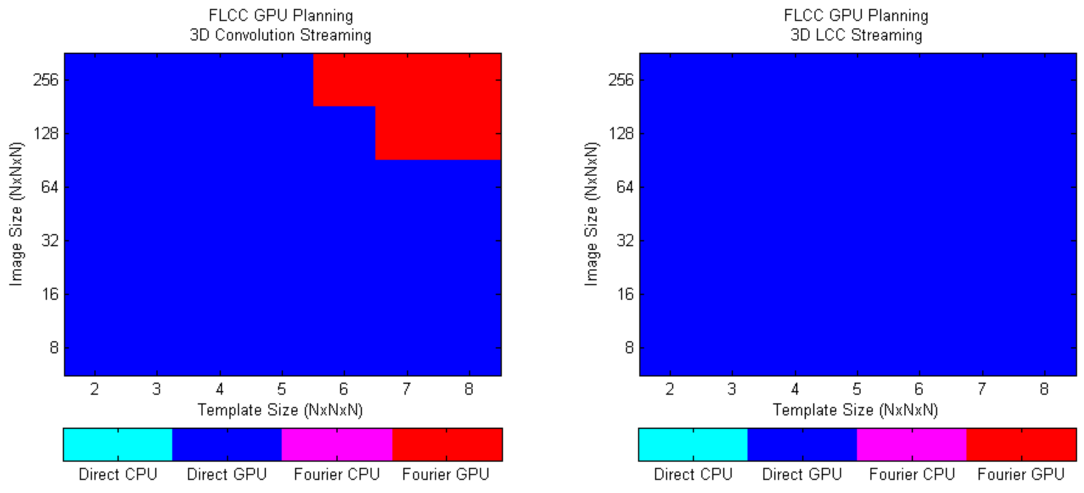

Figure 8.22: Method Selection on GPU for 3D Streaming

We shall note here that in Figure 8.21 and in Figure 8.22 the color map depicting the results does not exceed the image size of $256 \times 256 \times 256$. This happens because, due to the insufficient memory size of our system's GPU, the entire set of FLCC computation routines inevitably fail for the image size of $512 \times 512 \times 512$. Therefore, the planning function is not able to select any of them and returns with an error.

Bibliography

Apicella, Anthony, Jonathan Shane Kippenhan, and Joachim H. Nagel. "Fast Multi-modality Image Matching." *Proc. SPIE Medical Imaging III: Image Processing.* 1989. 252.

Barney, Blaise. *Introduction to Parallel Computing.* Lawrence Livermore National Laboratory. https://computing.llnl.gov/tutorials/parallel_comp/.

—. *POSIX Threads Programming.* Lawrence Livermore National Laboratory. https://computing.llnl.gov/tutorials/pthreads/.

Bracewell, Ronald N. *The Fourier Transform and Its Applications 3rd Edition.* McGraw-Hill, 2000.

Brunelli, R., and T. Poggio. "Face Recognition: Features versus Templates." *IEEE PAMI.* 1993. 1042-1052.

Cahn von Seelen, Ulf M., and Ruzena Bajcsy. *Adaptive Correlation Tracking of Targets with Changing Scale.* Department of Computer and Information Science, University of Pennsylvania, June 1996.

Chang, Dar-Jen, Ahmed H. Desoky, Ming Ouyang, and Eric C. Rouchka. "Compute Pairwise Manhattan Distance and Pearson Correlation Coefficient of Data Points with GPU." *Proceedings of the 10th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing.* IEEE, May 2009. 501-506.

Cooley, James, and John Tukey. "An Algorithm for the Machine Calculation of Complex Fourier Series." *Mathematics of Computation*, 1965.

Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 2nd Edition.* MIT Press, 2001.

Dimitriadis, Stavros, Andreas Pomportsis, and Evangelos Triantafyllou. *Multimedia Technology Theory and Practice.* Tziola Publications, 2004 (in Greek).

Eibl, Patrick, Dennis Healy, Nikos P. Pitsianis, and Xiaobai Sun. "Fast Pattern Matching in 3D Images on GPUs." *HPEC Proceedings.* 2009.

Fragkakis, Charalampos N. *Statistics.* University Studio Press, 2001 (in Greek).

Frigo, Matteo, and Steven G. Johnson. "FFTW Manual v3.2.2." July 2009.

—. "The Design and Implementation of FFTW3." *Proc. IEEE, vol. 93, no. 2, pp. 216–231.* 2005.

Gembris, Daniel, Markus Neeb, Markus Gipp, Andreas Kugel, and Reinhard Männer. "Correlation Analysis on GPU Systems Using NVIDIA's CUDA." *Journal of Real-Time Image Processing*, Dec. 2011: 275-280.

Halfhill, Tom R. "Parallel Processing with CUDA." *Microprocessor Report*, January 2008.

Hayes, Monson H. *Schaum's Outline of Theory and Problems of Digital Signal Processing.* McGraw-Hill, 1999.

Hopf, Matthias, and Thomas Ertl. "Accelerating 3D Convolution Using Graphics Hardware." *Visualization '99. Proceedings .* San Francisco, Oct. 1999. 471-564 .

Iliopoulos, Alexandros-Stavros. "AutoGPU: Automatic Production of Optimized CUDA Kernel Code." Thessaloniki: Aristotle University of Thessaloniki, July 2011 (in Greek).

Itti, L., N. Dhavale, and F. Pighin. "Realistic Avatar Eye and Head Animation Using a Neurobiological Model of Visual Attention." *Proc. SPIE 48th Annual International Symposium on Optical Science and Technology.* Bellingham: SPIE Press, Aug. 2003. 64-78.

Johnson, Andrew E., and Martial Hebert. "Efficient Multiple Model Recognition in Cluttered 3-D Scenes." *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition.* June 1998. 671-677.

Karafyllias, Ioannis, and Isaac Manolis. "The Computation of Local Correlation Coefficients of Images on Graphics Processing Units." Aristotle University of Thessaloniki, Apr. 2009 (in Greek).

Karniadakis, George Em, and Robert M. Kirby. *Parallel Scientific Computing in C++ and MPI: A Seamless Approach to Parallel Algorithms.* Cambridge University Press, 2002.

Kirk, David B., and Wen-mei W. Hwu. *Programming Massively Parallel Processors A Hands-on Approach.* Elsevier Inc., 2010.

Kugiumtzis, Dimitris. *Statistics for Electrical Engineers.* Thessaloniki: Aristotle University of Thessaloniki, 2005 (in Greek).

Lindholm, Erik, John Nickolls, Stuart Oberman, and John Montrym. "NVIDIA Tesla: A Unified Graphics and Computing Architecture." *Micro, IEEE* , March-April 2008 : 39-55 .

Lowe, David G. "Object Recognition from Local Scale-Invariant Features." *Proc. of the International Conference on Computer Vision.* Corfu: IEEE, 1999. 1150-1157.

Mattlof, Norman. *Programming on Parallel Machines.* Davis: University of California.

Moustakides, George. *Basic Techniques in Digital Signal Processing.* Tziola Publications, 2004 (in Greek).

Mutch, Jim, Ulf Knoblich, and Tomaso Poggio. "CNS: a GPU-based Framework for Simulating Cortically-organized Networks." Massachusetts Institute of Technology, Cambridge, MA, Feb. 2010.

Muyan-Özçelik, Pınar, John D. Owens, Junyi Xia, and Sanjiv S. Samant. "Fast Deformable Registration on the GPU: A CUDA Implementation of Demons." *Proceedings of the 2008 International Conference on Computational Science and Its Applications (ICCSA).* IEEE Computer Society Press, June 2008.

Nichols, Bradford, Dick Buttlar, and Jacqueline Proulx Farrell. *Pthreads Programming.* O'Reilly & Associates Inc., 1996.

Nickolls, John, Ian Buck, Michael Garland, and Kevin Skadron. "Scalable Parallel Programming with CUDA." *ACM Queue*, March/April 2008: 40-53.

NVIDIA. "CUDA API Reference Manual v4.0." Feb. 2011.

NVIDIA. "CUDA C Best Practices Guide v4.0." May 2011.

NVIDIA. "CUDA CUFFT Library Manual v3.1." May 2010.

NVIDIA. "CUDA Programming Guide v4.0." June 2011.

Panas, Stavros M. *Analog Signal Processing.* Thessaloniki: Aristotle University of Thessaloniki, 2001 (in Greek).

—. *Stochastic Signal Processing.* Thessaloniki: Aristotle University of Thessaloniki, 2001 (in Greek).

Papamakarios, Georgios, and Georgios Rizos. "FLCC Manual v1.3." 2011.

Papamakarios, Georgios, Georgios Rizos, Nikos P. Pitsianis, and Xiaobai Sun. "Fast Computation of Local Correlation Coefficients on Graphics Processing Units." *Proc. SPIE 7444, 744412.* 2009.

Papamarkos, Nikos. *Digital Image Processing.* V. Giourdas Editions, 2005 (in Greek).

Pitas, Ioannis. *Digital Image Processing.* Thessaloniki, 2001 (in Greek).

Podlozhnyuk, Victor. *FFT-Based 2D Convolution.* NVIDIA Corporation, June 2007.

Podlozhnyuk, Victor. *Image Convolution with CUDA.* NVIDIA Corporation, June 2007.

Press, William H., Saul A. Teukolsky, William T. Flannery, and Brian P. Vetterling. *Numerical Recipes The Art of Scientific Computing.* Cambridge University Press, 2007.

Savvidis, Evangelos. "Automatization of Code Production for GPU Architectures." Thessaloniki: Aristotle University of Thessaloniki, May 2011 (in Greek).

Serre, Thomas, Lior Wolf, and Tomaso Poggio. "Object Recognition with Features Inspired by Visual Cortex." *Proceedings of 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR).* San Diego: IEEE Computer Society Press, June 2005.

Smith, Steven W. *The Scientist's and Engineer's Guide to Digital Signal Processing 2nd Edition.* California Technical Publishing, 1999.

Stallings, William. *Operating Systems: Internals and Design Principles, 4th Edition.* Prentice Hall, 2001.

Strintzis, Michael G. *Time Series Analysis.* Kyriakidis Brothers Publishing House S.A., 2000 (in Greek).

Sun, Xiaobai, Nikos P. Pitsianis, and Paolo Bientinesi. "Fast Computation of Local Correlation Coefficients." *Proc. of SPIE Vol. 7074 707405-1.* Sep. 2008.

Tanenbaum, Andrew S. *Modern Operating Systems 3rd Edition.* Prentice Hall Inc., A Pearson Education Company, 2008.

Tanenbaum, Andrew S., and James R. Goodman. *Structured Computer Organization 4th Edition.* Prentice Hall Inc., A Pearson Education Company, 1999.

Van Loan, Charles. *Computational Frameworks for the Fast Fourier Transform.* Philadelphia, PA: Society for Industrial and Applied Mathematics, 1992.

Zitova, Barbara, and Jan Flusser. "Image registration methods: a survey." *Image and Vision Computing*, October 2003: 977-1000.