FLCC: A Library for Fast Computation of Convolution and Local Correlation Coefficients

Georgios Papamakarios[1]
gpapamak@auth.gr

Georgios Rizos[1]
grizos@auth.gr

Nikos P. Pitsianis[1,2]
nikos.pitsianis@eng.auth.gr

[1] Department of Electrical and Computer Engineering
Aristotle University of Thessaloniki
Thessaloniki, Greece

[2] Department of Computer Science
Duke University
Durham, NC, U.S.A.

***Abstract*—The 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 paper 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.**

Keywords-fast convolution; normalized correlation coefficients; multi-core processors; graphics processing units

I. Introduction

Convolution is without doubt one of the most important and fundamental mathematical operations in the field of digital image processing, as well as in general signal processing and control systems theory. Discrete ν-dimensional convolution between real signals x and y is defined in (1).

$$(x * y)[\boldsymbol{n}] = \sum_{\boldsymbol{k}\in\mathbb{R}^{\nu}} x[\boldsymbol{k}]y[\boldsymbol{n}-\boldsymbol{k}] \qquad (1)$$

The significance of convolution is of both theoretical and practical nature. Convolution is found in the center of Linear and Shift Invariant (LSI) systems description. In fact, the output of such a system is precisely equal to the convolution between its input and its impulse response [1]. Moreover, as far as digital images are concerned, convolution represents the filtering process between an image and an image filter in the spatial domain [2]. As such, convolution is frequently used as a basic computational block in several image analysis processes, such as pattern recognition, object detection, feature extraction and many more [3][4].

A similar to convolution and at the same time equally important mathematical operation is correlation (also known as cross-correlation). Discrete ν-dimensional correlation between real signals x and y is defined in (2).

$$(x \star y)[\boldsymbol{n}] = \sum_{\boldsymbol{k}\in\mathbb{R}^{\nu}} x[\boldsymbol{k}]y[\boldsymbol{n}+\boldsymbol{k}] \qquad (2)$$

Correlation is used to express the similarity degree between a signal and a signal template in every possible relative positioning. That way a "best match" for the template can be determined within the signal. An even more accurate similarity measure can be obtained if the template and each overlapping signal section are normalized to zero mean and unit standard deviation before correlation is performed. The above leads to the definition of the so called table of correlation coefficients with local normalization (or simply Local Correlation Coefficients – LCCs). In (3) we express the LCC between a template T and the overlapping signal section (or panel) P – both of N_T elements – in the case of a 2-dimensional digital signal (such as an image). By $\mu(P)$ and $\sigma(P)$ we denote respectively the mean and the standard deviation of P (similarly for T).

$$r(P,T) = \frac{1}{N_T}\frac{\sum_{x,y}\left(P_{x,y}-\mu(P)\right)\left(T_{x,y}-\mu(T)\right)}{\sigma(P)\sigma(T)} \qquad (3)$$

Due to the well-known Cauchy-Schwartz inequality, $r(P,T)$ will always fall within the close interval [-1, 1], thus often being interpreted as the cosine of the "angle" between the panel and the template. That practically means that a coefficient of value 1 will denote a perfect proportional match between the selected panel and the template, while a coefficient of value -1 will denote a perfect, but inverse, match. Of course, a zero coefficient indicates a complete mismatch between the two or, as otherwise stated, the panel being orthogonal to the template. Therefore, thanks to their capability of effectively indicating local similarity, LCCs find numerous applications in digital image processing such as template or pattern matching, image registration, change or motion detection, to name only a few [5][6].

Unfortunately, usefulness does not come at no cost. Both convolution and LCCs have always been considered operations of high computational intensity, often rendering their usage rather troublesome. Especially in the case of LCCs, the local normalization characteristic increases the computational requirements significantly, thus discouraging their usage against unnormalized correlation. The above problem is far more evident in situations of high processing requirements, such as applications involving images of high resolution and/or 3 dimensions or real-time processing of images (e.g. real-time video processing).

Several efforts have been made towards the direction of developing methods for fast and efficient calculation of the operations under consideration. In the case of convolution, the well-known convolution theorem has been widely used to reduce the arithmetic complexity to a minimum. In particular, this theorem expresses convolution in the Fourier – or frequency – domain, enabling the usage of the efficient Fast Fourier Transform (FFT) algorithm for its calculation [1]. Still though, the performance benefit of this method evaporates when small templates are concerned. Another promising approach for multi-dimensional convolution in particular, known as separable convolution, takes advantage of the separability of the template, managing to reduce a multi-dimensional convolution to a series of one-dimensional convolutions. However, this approach is not equally effective for all templates, since it is highly dependent on the degree of the template's separability.

The case of LCCs has seemed to be even more challenging. Several methods have been proposed that try to reduce computation time by approximating the result, such as making ad-hoc assumptions about the image properties or relaxing the requirement of locality. Others attempt to adapt the techniques developed for fast convolution to the computation of LCCs, such as expressing the computation to a certain degree in the Fourier domain. An effective method is presented in [7], where the computational load is reduced via the usage of FFTs and precomputed integrals over the image, the latter being a technique first introduced in [8] for rapid low-pass image filtering. However, major part of the computation is still done in the spatial domain since the LCCs are not yet fully expressed in the Fourier domain. In [9], the method of precomputed integrals is further exploited, managing to reduce computation time substantially. However, the method proposed is still approximate, depending crucially on the specific form of the template. A fully Fourier-based method for calculating the LCCs was recently introduced in [10], with a full algorithmic description in [11]. The proposed algorithm handles effectively large templates and computations on a stream of images but, as with FFT based convolution, performs rather poorly for small templates.

Alongside addressing the problem by algorithmic means, notable efforts have been made in taking advantage of the computational capability of various hardware architectures in order to achieve accelerated computation. Naturally, hardware developments over the years have been increasing performance continuously. For the computation of convolution and LCCs, parallel architectures seem to be particularly suitable, since these two operations exhibit considerable parallelization potential. Apart from the more traditional parallelization architectures, such as multi-CPU systems, array processors (many-cores) have been employed for effectively carrying out the computation. CUDA-enabled Graphics Processing Units (GPUs) are used in [11], [12] and [13] for convolution implementations and in [11], [14], [15] and [16] for LCC implementations, together with various algorithmic techniques, in order to produce worthy results.

In this paper we attempt to develop a unified scheme in addressing the computation of convolution and LCCs on digital images. We present the FLCC library, a computational tool which provides a simple yet powerful interface for fast computation of convolution and LCCs on a wide range of practical cases. Our approach is based on three levels. Firstly, we utilize a set of efficient algorithms, reducing arithmetic complexity to a minimum. Secondly, we exploit top-notch modern parallel architectures, namely multi-core processors and CUDA-enabled GPUs, in order to effectively carry out the computations. Thirdly, we combine the advantages of the different algorithmic and architectural approaches under an easy-to-use and portable unifying scheme, namely the plan-execute model.

In section II we present the full set of algorithms used in FLCC library. In section III we describe the interface and the various characteristics of FLCC, together with hardware implementation and details on the plan-execute model. We conclude with indicative experimental results in section IV and some final remarks in section V.

II. Algorithms

The first level of our approach consists of developing efficient algorithms for the calculation of both convolution and LCCs. We make no previous assumptions on the content of the two signals – apart from their being real-valued – and so we are interested in algorithms that guarantee the correct result in any case; in other words we do not accept approximations or numerical sacrifices. For this purpose we develop two separate algorithmic methods for each operation; we refer to them as “direct method” and “Fourier domain method”, the former addressing the computation in the spatial domain while the latter doing so in the frequency domain.

A. *Direct Method*

In the case of convolution, the direct method is quite straightforward; it simply consists of calculating the result directly as it is suggested by its definition in (1). That is, we consider the filter (typically the smaller in size of the two signals) to be sliding along every dimension. In each position we simply calculate the dot product between the filter and the according panel, obtaining the convolution value for that position. This method, albeit straightforward, exhibits certain properties that, if properly exploited, can lead to respectable performance. Firstly, it is highly parallelizable, up to pixel level from the writer’s point of view, i.e. every convolution value can be calculated totally independently from the others. Secondly, the method shows high space locality, which can facilitate efficient memory caching. In section III we further explain the way in which these properties are exploited in the implementation level of FLCC.

In the case of LCCs, things are less straightforward though. Attempting to directly perform the computation guided by the definition in (3) poses difficulties which stem from the need of calculating the local mean and standard deviation for every single panel in the signal. To do that in an efficient way in the spatial domain, we need to effectively reformulate the LCC definition formula. We first note that the template T can be pre-normalized once for all in the beginning. Typically the template will be small enough for this operation to be of trivial cost. Denote by $\hat{T}$ this normalized template. We progressively rewrite (3) as shown below.

$$r(P,T) = \sum_{x,y} \left(\frac{P_{x,y} - \mu(P)}{\sqrt{N_T}\sigma(P)} \right) \hat{T}_{x,y} \tag{4}$$

$$r(P,T) = \frac{\sum_{x,y} P_{x,y} \hat{T}_{x,y} - \mu(P) \sum_{x,y} \hat{T}_{x,y}}{\sqrt{\sum_{x,y} \left(P_{x,y} - \mu(P) \right)^2}} \tag{5}$$

$$r(P,T) = \frac{\sum_{x,y} P_{x,y} \hat{T}_{x,y}}{\sqrt{\sum_{x,y} P_{x,y}^2 - \frac{1}{N_T} \left(\sum_{x,y} P_{x,y} \right)^2}} \tag{6}$$

Note that $\sum_{x,y} \hat{T}_{x,y} = 0$ since we have assumed $\hat{T}$ to be of zero mean. Note also that calculating the LCC by (4) would require three separate passes over each panel while by (6) the entire calculation can be performed within a single pass over each panel. Therefore, (6) provides us with the desired direct method of efficiently computing LCCs in the spatial domain.

B. Fourier Domain Method

This method consists of performing the major part of the calculation in the Fourier (frequency) domain. To achieve that for convolution, we base on the well-known convolution theorem, which is stated in (7).

$$\mathcal{F}\{x * y\} = \mathcal{F}\{x\}\mathcal{F}\{y\} \tag{7}$$

In (7), $\mathcal{F}\{x\}$ denotes the Fourier transform of signal x (same for y). This theorem provides us with an efficient way of calculating convolution. In particular, we can obtain convolution between signals x and y by first Fourier-transforming them, then calculating the element-wise complex product of the transforms in the Fourier domain and finally transforming the product back to the spatial domain. The efficiency of this method arises from the possibility to perform the transforms via the highly efficient FFT algorithm. Furthermore, we have assumed that x and y are real-valued (since they represent digital images). As it is well-known, the Fourier transform of a real-valued signal exhibits conjugate symmetry, i.e. half of its values can be expressed as the complex conjugate of the other symmetrical half. The above property can be further exploited in a way that the necessary operations to perform a real-valued FFT be reduced by approximately half compared to a regular complex FFT [11][18]. Referring to the real-valued FFT as "half-FFT", we easily see that the whole operation can be completed with only 1.5 FFTs (i.e. 3 half-FFTs). Finally, one more optimization can be considered in the – quite frequent – case where a stream of images needs to be processed by the same filter. In that case, to which we refer as "streaming", the FFT of the filter need only be calculated once; assuming the number of images in the stream is sufficiently large, we end up with convolution by just 1 FFT (i.e. 2 half-FFTs) per image.

Moving the above method to the case of LCCs is not trivial though. We present here a variation of the algorithm firstly described in detail in [11]. Denote by S the signal to be processed (typically an image) and by T the processing template of size N_T. We describe every panel P in S of size N_T as in (8).

$$P(u,v) = S(x,y)B_T(x-u, y-v) \tag{8}$$

Here, B_T is the spatial support of the template, i.e. a signal of the same size as the template having units as values. With (8) in mind, we rewrite (6) as in (9).

$$r(u,v) = \frac{S \star \hat{T}}{\sqrt{(S^2 \star B_T) - \frac{1}{N_T}(S \star B_T)^2}} \tag{9}$$

$$S \star \hat{T} = \sum_{x,y} S(x,y)\hat{T}(x-u, y-v) \tag{10}$$

$$S^2 \star B_T = \sum_{x,y} S^2(x,y)B_T(x-u, y-v) \tag{11}$$

$$S \star B_T = \sum_{x,y} S(x,y)B_T(x-u, y-v) \tag{12}$$

In (9) we manage to express the whole LCC table in terms of three correlations, described in (10), (11) and (12). But since correlation is essentially a convolution where one of the two signals has been reversed along every dimension (as it follows by their definitions), it is possible to obtain these three correlations by making use of the convolution theorem in (7). We are therefore in the position now to present the Fourier domain method of calculating the LCCs.

- Perform the FFTs of signals S, S^2, $\hat{T}$ and B_T.
- Calculate the element-wise complex products $\mathcal{F}\{S\}\mathcal{F}\{\hat{T}\}$, $\mathcal{F}\{S^2\}\mathcal{F}\{B_T\}$ and $\mathcal{F}\{S\}\mathcal{F}\{B_T\}$.
- Perform the inverse FFTs of the three products to obtain $S \star \hat{T}$, $S^2 \star B_T$ and $S \star B_T$ respectively.
- Combine the three correlations element-wise as in (9).

It is therefore evident that the full LCC computation can be effectively performed via 7 FFTs. Note though that in the spatial domain all the quantities involved are real-valued. That means the 7 FFTs can be efficiently reduced to only 3.5 FFTs. Furthermore, in the frequent case of streaming, the FFTs of $\hat{T}$ and B_T need only be computed once for the entire stream. This finally leads to an efficient LCC full computation with just 2.5 FFTs per image.

C. *Analysis and Comparison*

Denote by N_y the size of the convolution/LCC result and by N_T the size of the filter/template. In Table I we present the arithmetic complexity in terms of number of multiplications for each algorithm we have previously described.

In calculating the complexity, we took into account the convention that a complex FFT of N points takes no more than $5N \log N$ operations [11]. We also accepted that the complexity of the Fourier domain method is dominated by the FFTs, disregarding the additive linear factor contributed by the element-wise operations (complex product, division, root extraction, etc.).

Observing the arithmetic complexities of Table I leads us to some particularly interesting conclusions. Firstly, we note that the complexity of the direct method is of class $O(N_y N_T)$ while the complexity of the Fourier domain method is of class $O(N_y \log N_y)$. The class of each method is the same for both convolution and LCCs; the latter exhibit higher constant coefficients which account for the additional need for local normalization. Secondly, the number of operations of the direct method shows strong (proportional) dependence on the size of the template, while in the Fourier domain method the number of operations is determined primarily by the size of the image (assuming the typical case where the image is much larger than the template), its performance being practically independent by the size of the template. Therefore, the theoretical determinant of which method is more efficient for each case is the comparison between N_T and the "critical size" N_T^{crit}, as it follows:

- When $N_T < N_T^{crit}$ the direct method performs better.
- When $N_T > N_T^{crit}$ the Fourier domain method performs better.

The critical size designates the template size for which both methods perform equivalently and it can be theoretically estimated by equating the arithmetic complexities of the two methods. In Table II we show the theoretically estimated value of N_T^{crit} for each possible case. We denote by N_S the size of the image and in our calculations we assume that $N_S \cong N_y$. In section III we demonstrate how we exploit the above conclusion for developing a universal scheme of efficient calculation of convolution and LCCs in FLCC library.

TABLE I. ARITHMETIC COMPLEXITY OF ALGORITHMS

Operation	**Algorithmic Method**		
	Direct	***Fourier (single image)***	***Fourier (streaming)***
Convolution	$N_y N_T$	$7.5 N_y \log N_y$	$5 N_y \log N_y$
LCCs	$2 N_y N_T$	$17.5 N_y \log N_y$	$12.5 N_y \log N_y$

TABLE II. CRITICAL TEMPLATE SIZES

Operation	**Computational Case**	
	Single image	***Streaming***
Convolution	$7.5 \log N_S$	$5 \log N_S$
LCCs	$8.75 \log N_S$	$6.25 \log N_S$

III. FLCC LIBRARY

Apart from the algorithmic methods described in section II, another way to further accelerate the computation of convolution and LCCs is to utilize the potential of modern parallel computer architectures. Among the wide variety of currently available architectures we selected to work on two particular platforms, namely multi-core processors and Graphics Processing Units (GPUs), the latter being specifically those manufactured by NVIDIA. The choice was made after taking into account their computational power, effectiveness regarding the calculations that concern us, wide availability, affordability, relatively easy-to-learn programmability and the existence of already established optimized libraries providing the functions for performing FFTs.

On multi-core processors multithreading was done using the POSIX threads API (or just pthreads) [17] and all FFTs were made possible using version 3 of FFTW library [18]. Likewise, on GPUs the programming was done with NVIDIA's CUDA programming environment [19] and the FFT functions were provided by the CUFFT library [20], again by NVIDIA.

A. *FLCC Library Presentation*

As a next step to theoretical algorithmic development, we proceeded in creating the FLCC library, a powerful and versatile computational tool capable of a holistic approach towards the fast calculation of convolution and LCCs. FLCC is a library for the C (or C++) programming language and it is available for use in the form of a complete software package, easy-to-use and hopefully practically useful. Its goal is to provide its user with an interface to facilitate the execution of the operations under consideration. Its efficiency is achieved in three ways. Firstly, it contains a collection of functions each implementing one of the algorithms described in section II. Secondly, it utilizes the strengths of the chosen parallel architectures along with those of established libraries for the efficient execution of FFTs. It should be noted that the various algorithms and architectures can achieve different efficiency, relative to the type and size of the problem, as already shown in section II. Thus, finally, in order to combine every asset for the best possible result, the library uses a versatile, open to expansion plan-execute mechanism, similarly to the FFTW and CUFFT libraries. At the time of syntax of this paper, FLCC library is at version 1.3 (FLCC v1.3). Despite the library's being fully operational, it is still at an early stage and will be complemented with new features in the future. A complete list of FLCC's current features follows:

- Fast computation of convolution for 2D and 3D images and filters of any size.
- Fast computation of locally normalized correlation coefficients distributions (LCCs) for 2D and 3D images and templates of any size.
- Single precision arithmetic.
- Accelerated computation of convolution and LCCs between a stream of images and the same filter/template.
- Capability of parallel computation either on a multi-core CPU or a (currently one) GPU.

Finally, we report that FLCC v1.3 is supported for use on UNIX/Linux and MS Windows (via Cygwin) systems. Each new version is freely available for installation and use from the webpage http://autogpu.ee.auth.gr/. FLCC is distributed under the FreeBSD license, complete with full documentation.

B. *FLCC Interface Description*

Central idea to the design of FLCC is to not relate its user with practical issues concerning the choice of an appropriate algorithm or architecture and to provide them with a simple and succinct API that is responsible for all the work; from making the correct choice, to performing the computation. Version 1.3 of FLCC library provides six main functions for the computation of convolution and LCCs. In addition, there are two more memory management functions intended for the allocation and deallocation of memory for the input and output tables, which should be used instead of the traditional ones. Note that as part of the documentation of FLCC there is a detailed up-to-date manual [21], fully describing the API. An overview of FLCC's API is shown in Table III.

A short description of the plan-execute mechanism used by FLCC would be the following. Initially, the user calls a planner function in which they describe the problem (number of dimensions, size of each dimension, single image or streaming). The planner function processes that input and decides on the best algorithm-architecture combination for the calculations. The planner proceeds to creating an object called plan, which includes all the information pertaining to this choice. Afterwards, the user calls an executor function that uses the above plan and the input images and returns the resulting output table. The advantage of this approach is that the execution can be performed for an unrestricted number of times (though for the same type of computation and size of images) while the planning need only be done once for each different problem. This can lead to substantial efficiency in the (quite frequent) case of several calculations of the same type. Finally, there exists another type of function that deallocates all the resources occupied by the plan. It is intended to be called when the plan is no more needed.

C. *Parallel Implementation of Direct Method*

We will analyze the direct method for both convolution and LCCs alongside because, as we saw in section II, they are similar in that they both have to do with a sweep of the filter/template over the image and the execution of some sort of operation between the filter/template and the overlapping panel. Another thing they have in common is their high parallelizability. Every output element can be calculated independently from all others, by a different thread, just by knowing the filter/template and the corresponding panel.

TABLE III. FLCC API FUNCTIONS

Operation	**Computational Stage**		
	Planning	***Execution***	***Plan deallocation***
Convolution	`conv_plan()`	`conv_exec()`	`conv_destroy()`
LCCs	`lcorr_plan()`	`lcorr_exec()`	`lcorr_destroy()`
Memory Management			
`flcc_malloc()`		`flcc_free()`	

In the case of a CPU with one or more cores the parallelization is done by creating threads using the pthreads library. The number of threads created is dictated by the user during FLCC's installation and of course it is recommended to be set equal to the number of the CPU cores available. Each thread is given access to the filter/template and the padded image and it is charged with calculating its share of output elements. There has also been an effort for optimization. The majority of filters/templates found in practical problems are of equal dimension sizes (i.e. square or cubic) and those managed by the direct method have a relatively small size. This lends the opportunity for an optimization which lies on the idea of hard-coding the few most frequent cases, allowing among others for explicit loop unrolling. Indeed, apart from the general, naive thread functions that perform convolution and LCCs, FLCC retains optimized hard-coded versions of that general function that refer to certain filter/template sizes (2x2 up to 32x32 for 2D and 2x2x2 up to 10x10x10 for 3D). Since the size is already known it is possible to fully unroll the iteration loop for each panel. Regarding the optimized thread functions, this is done with a smart use of macro-instructions. We define a macro-instruction that does the work of the thread function and the sizes of the filter/template's dimensions are given as arguments. During installation multiple calls of this macro-instruction are compiled for the different sizes we discussed and since the number of iterations for each case is a constant number the compiler fully unrolls the loop. This way, we achieve optimized performance for the majority of common cases while it is ensured that the calculation is always performed correctly even for the less common ones.

Given the capabilities of GPUs and CUDA, it is possible to start as many CUDA threads as the number of output elements; each thread executing the kernel function that calculates its according element using the direct method. We organize CUDA threads to fixed-size blocks and consequently the blocks to a grid. We take steps though to avoid a control structure inside the kernel that checks whether the output element is within the logical limits of the output array. We do this by allocating more memory than needed so as to have an exact fit of all the blocks into this enlarged output table. After the calculations are done we discard the additional values. This whole process is incorporated during the padding process and thus does not affect the temporal efficiency negatively. Again, there has been an effort for further optimization. The primary memory modules of NVIDIA's GPUs are the following three: the local memory private to each thread, a fast shared memory module present in each of the GPU's multiprocessors that all threads in a block share and the larger, relatively slower global memory of the GPU available to all threads [19]. We perform caching of all the data needed by the block threads on the shared memory module which costs very little in time and ensures faster multiple accesses to the data needed. Said data are the filter/template and the section of the image needed by the block threads. Mirroring the implementation on a CPU, we proceed in performing full hard-coded loop unrolling in the common cases of filters/templates with the certain sizes we discussed. This time we have created a code generator using MATLAB that generates the code of a kernel for a given filter/template size. We pre-generated kernels for the sizes of interest that both utilize shared memory and perform full loop unrolling to the effect we described earlier in the case of CPUs. The code of these kernels is pre-included in the library and it is compiled during installation.

We should note that the latter method for performing loop unrolling is slightly faster than using macro-instructions but its major disadvantage is that it dramatically increases the size of the library's source code, leading to high compilation times during installation. In the case of the CPU though, the difference in optimization achieved by code generators is negligible, so we choose to work with macro-instructions instead, avoiding further increasing FLCC's source code.

D. Parallel Implementation of Fourier Domain Method

As we described in section II, the significant part of the Fourier domain method for LCCs is the computation of three convolutions. Therefore we describe the implementation of the Fourier domain method focusing on the stage of convolution.

In the CPU implementation of the Fourier domain method we extensively use the established FFTW library that provides optimized functions for performing the 2D and 3D half-FFTs needed. This way the FFTs are performed at roughly half the time and with half the memory requirements. FFTW also supports multithreading as it utilizes internally the pthreads library. The element-wise complex multiplications are easily executed by sharing the workload among different threads. Moreover, FFTW's functions work faster for tables whose dimension sizes may be expressed as small prime number products [18]. In order to exploit this characteristic, we expand each table dimension to meet this requirement. This process is included in the padding process and of course in the end we discard the additional elements of the result. A final detail is that after an inverse FFT by an FFTW function the signal we get is not the wanted signal. The latter will result after a division of each element of said signal with its total number of elements. This division is included in the element-wise multiplication thread function so as to be done in parallel.

In the GPU implementation we use in the same manner NVIDIA's CUFFT library for the half-FFTs. As for the element-wise multiplication, we created a simple kernel that performs it. The same also applies in this case concerning the time and memory space benefits of the half-FFTs, the fact that the FFTs are faster for tables whose sizes are small prime number products and the need for a division of each of the signal's elements by the number of elements of the signal. So we act similarly by lengthening each table dimension as in [13] to achieve faster FFT execution and performing the division inside the CUDA kernel so as to be done in parallel.

E. Streaming on GPUs

Apart from the algorithmic benefits of streaming on the Fourier domain method we already discussed in section II, there is also the possibility of an additional optimization in the case of computations on a GPU using either the direct or the Fourier domain method. In order for data to be processed by the multiprocessors of a GPU, they first need to be transferred from RAM to the GPU's global memory. Regarding computations between a stream of N images and a filter/template, all of the N images and the filter/template have to be transferred to the global memory and then N output tables have to be transferred back to RAM. This would be seriously time-consuming but CUDA offers a possibility that we can exploit in order to "hide" the transfer time behind the computation time. CUDA streams lend the opportunity of asynchronous concurrent execution of kernels and transferring of data for certain GPUs [19]. So, while the i-th output table is being calculated, we transfer the (i-1)-th output table back to RAM and the (i+1)-th image to the GPU's global memory simultaneously. With this concurrency trick the only visible transfer times are the first image's and the filter/template's ones to the GPU and the last output table's one back to RAM. In case asynchronous concurrent execution and transfer is not supported by the user's GPU, the transfers and calculations are still performed correctly yet serially.

F. FLCC's Plan-Execute Model

FLCC contains functions that implement all the different algorithms on both chosen parallel architectures with all combinations possible. During planning all of the available functions are called to perform calculations on random tables. Their individual execution times are measured with precision and the function to be determined the fastest is designated to perform the computations during the execution stage. In the case where one or more of these functions fail due to some internal error, FLCC recognizes the failure and excludes them from the selecting process. One final detail is that both FFTW's and CUFFT's planning stages are included in FLCC's planning stage and thus do not affect the execution time.

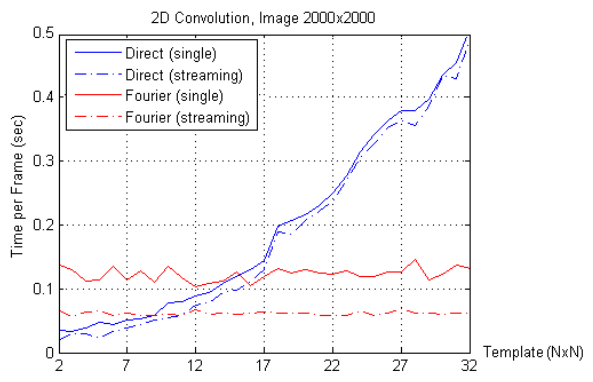
IV. Experimental Results

We present experimental results on the usage of FLCC after performing two different sets of experiments. Both experiments were carried out on a multi-core SMP system equipped with a CUDA-enabled GPU. The multi-core is an AMD Opteron 6168 of totally 24 cores with a clock speed of 1.90 GHz. The GPU is an NVIDIA GeForce GTX 480, consisting of a total of 480 cores and having a clock speed of 1.40 GHz.

A. First Set of Experiments: Algorithmic Comparison

We time the performance of every algorithm implemented in FLCC for a 2D image of 2000x2000 pixels against a template of sizes ranging from 2x2 up to 32x32 pixels. The results are illustrated in Fig. 1 for both convolution and LCCs on the multi-core and similarly in Fig. 2 on the GPU.

The results verify our initial hypothesis, i.e. the performance of the direct method decreases significantly as the template becomes larger, while the performance of the Fourier domain method remains stable for all template sizes. We also observe the critical template sizes for each case, whose existence serves as the basis for the FLCC planning strategy. One can also notice the performance benefit (up to approximately 50%) introduced by the process of streaming, thanks to both the algorithmic and the architectural optimizations it allows for. Last but not least, we see that the utilization of high-performance parallel architectures can alleviate the additional computational load introduced by local normalization; indeed, the performance on LCCs is very close to the one of convolution, thus enabling their usage in cases where local normalization would previously be considered too expensive to perform.



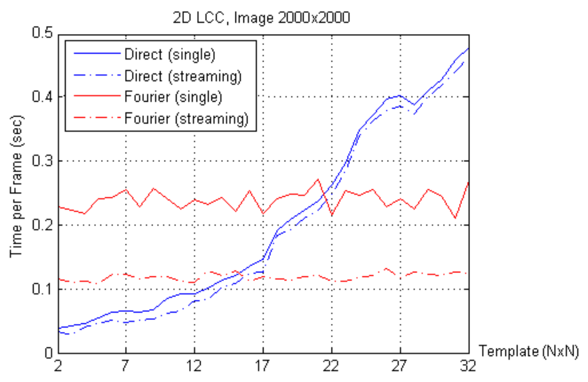
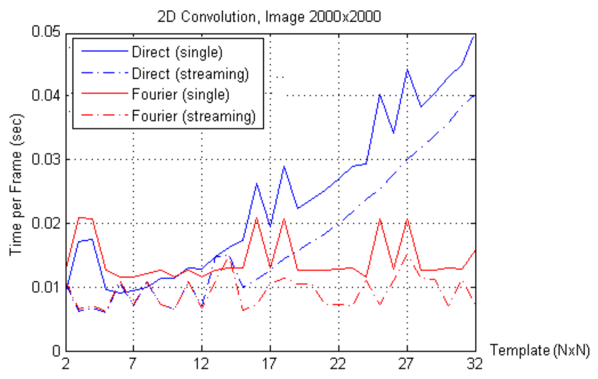


Figure 1. Algorithmic performance of convolution (left) and LCCs (right) on multi-core.



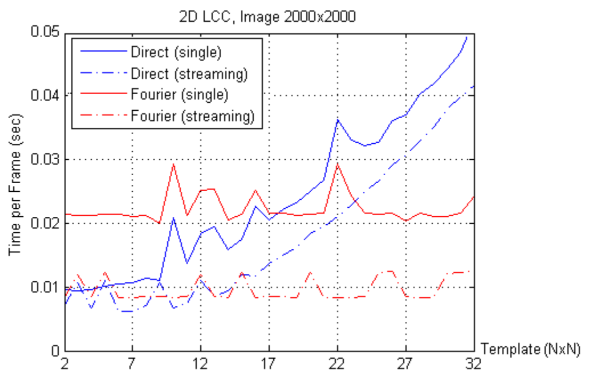


Figure 2. Algorithmic performance of convolution (left) and LCCs (right) on GPU.

B. *Second Set of Experiments: FLCC Planning*

Having examined the performance of each algorithm-architecture combination, we now focus on the behavior of the FLCC planning process. We illustrate the selections made by the planner functions of the FLCC interface for 2D convolution on a single image or a stream of images, separately on the multi-core in Fig. 3 and the GPU in Fig. 4 (our system's GPU is circa 10 times faster than the multi-core so planning between them would not be of much interest). The image sizes to plan for range from 32x32 up to 4096x4096 pixels and the template sizes range from 2x2 up to 32x32 pixels.

One can notice that, in principle, FLCC selects correctly according to our theoretical predictions and first set of experiments. Indeed, the direct method is selected for smaller templates while the Fourier domain method is preferred for larger templates. It is also noteworthy that FLCC seems to take correctly into account whether the computation refers to a single image or a stream of images, selecting the Fourier domain method – which as we have explained benefits algorithmically by the streaming – more often when streaming is concerned. Finally, we observe how FLCC selections take into account the differences between the two platforms; the direct method is more often selected on the GPU than on the multi-core, since its greater parallelizability allows it to perform better on a massively parallel array processor like the GPU. Similar results – not shown here – have also been obtained when planning for LCCs and can be found in [22].

V. Concluding Remarks

In this paper we began by examining algorithmically the calculation of convolution and LCCs in both the spatial and the frequency domain. We described two methods that correspond to each domain and we referred to them as direct method and Fourier domain method respectively. We showed that the direct method is more suitable for calculations with relatively small filters/templates, whilst the Fourier domain method is more suitable for larger ones. Secondly, we examined the suitability of convolution and LCC calculation in a parallel environment. We advocated in favor of the usage of multi-cores and CUDA-enabled GPUs, based on their computational power, availability and affordability. Thirdly, we explained how the various advantages of the different algorithmic and architectural approaches can be combined through the utilization of a plan-execute mechanism in execution time, leading to an efficient tackling of a wide range of practical cases.

We took one step forward in developing the FLCC library, an efficient, easy-to-use and portable computational tool that makes a holistic attempt towards the practical calculation of convolution and LCCs. We described how FLCC implements each algorithm on each architecture and how the best algorithmic-architectural combination for each case and system is selected in execution time via the plan-execute mechanism. Finally, we presented how FLCC's functionality is accessed through its interface, a minimal set of functions designed to be practical and easy-to-learn, without requiring any specialized knowledge by its user.

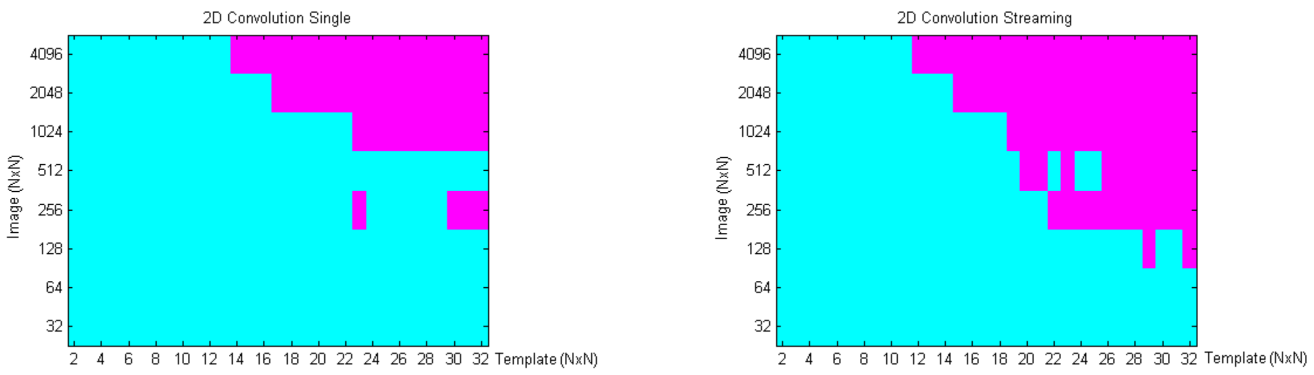


Figure 3. FLCC planning for a single image (left) and a stream of images (right) on multi-core. Cyan for direct method and magenta for Fourier domain method.

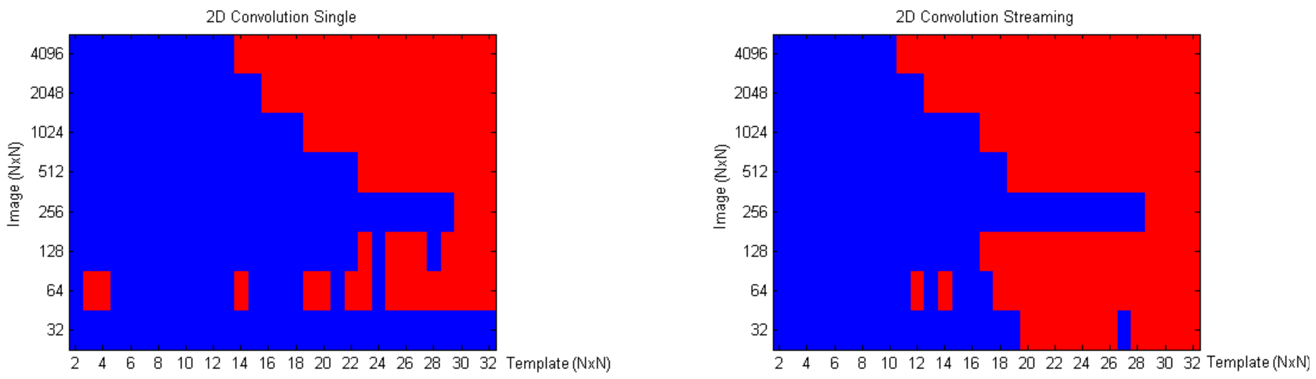


Figure 4. FLCC planning for a single image (left) and a stream of images (right) on GPU. Blue for direct method and red for Fourier domain method.

Of course, as any live piece of software, current FLCC version 1.3 manifests its own weaknesses and limitations. Further development and expansion of its functionality and utility are possible, such as exploration and incorporation of new algorithms or support for new architectures and platforms. Examples for immediate improvement include separable convolution, support for double precision or utilization of more than one GPU concurrently. Finally, less restraining (but more complicated) interfaces can be added, allowing the user an advanced accessibility to FLCC's internal power.

REFERENCES

[1] M. H. Hayes, Schaum's Outline of Theory and Problems of Digital Signal Processing, McGraw-Hill, 1999.

[2] I. Pitas, Digital Image Processing, Thessaloniki, 2001.

[3] D. G. Lowe, "Distinctive image features from scale-invariant keypoints," International Journal of Computer Vision, vol. 60, no. 2, pp. 91-110, 2004.

[4] L. Itti, 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, pp. 64-78, Aug. 2003.

[5] R. Brunelli, and T. Poggio, "Face recognition: features versus templates," IEEE PAMI, pp. 1042-1052, 1993.

[6] T. Serre, L. Wolf, and T. 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.

[7] J. P. Lewis, "Fast normalized cross-correlation," Industrial Light and Magic.

[8] F. Crow, "Summed-area tables for texture mapping," Computer Graphics, vol. 18, no. 3, pp. 207-212, 1984.

[9] K. Briechle, and U. D. Hanebeck, "Template matching using fast normalized cross correlation," Proceedings of SPIE, vol. 4387, AeroSense Symposium, Orlando, Florida, USA, Apr. 2001.

[10] X. Sun, N. P. Pitsianis, and P. Bientinesi, "Fast computation of local correlation coefficients," Proc. of SPIE, vol. 7074, 707405-1, Sep. 2008.

[11] G. Papamakarios, G. Rizos, N. P. Pitsianis, and X. Sun, "Fast computation of local correlation coefficients on graphics processing units," Proc. SPIE 7444, 744412, 2009.

[12] V. Podlozhnyuk, "Image convolution with CUDA," NVIDIA Corporation, June 2007.

[13] V. Podlozhnyuk, "FFT-based 2D convolution," NVIDIA Corporation, June 2007.

[14] I. Karafyllias, and I. Manolis, The Computation of Local Correlation Coefficients of Images on Graphics Processing Units, Dissertation for Diploma, Aristotle University of Thessaloniki, Apr. 2009.

[15] D.-J. Chang, A. H. Desoky, M. Ouyang, and E. 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, pp. 501-506, May 2009.

[16] P. Eibl, D. Healy, N. P. Pitsianis, and X. Sun, "Fast pattern matching in 3D images on GPUs," HPEC Proceedings, 2009.

[17] B. Nichols, D. Buttlar, and J. P. Farrel, Pthreads Programming, O'Reilly & Associates Inc., 1996 .

[18] M. Frigo, and S. G. Johnson, "The design and implementation of FFTW3," Proc. IEEE, vol. 93, no. 2, pp. 216-231, 2005.

[19] NVIDIA, CUDA Programming Guide v4.0, June 2011.

[20] NVIDIA, CUDA CUFFT Library Manual v3.1, May 2010.

[21] G. Papamakarios, and G. Rizos, FLCC Manual v1.3, Nov. 2011.

[22] G. Papamakarios, and G. Rizos, FLCC: A Library for Fast Computation of Convolution and Local Correlation Coefficients, Dissertation for Diploma, Aristotle University of Thessaloniki, Dec. 2011.