Applied Mathematics & Information Sciences
An International Journal

http://dx.doi.org/10.18576/amis/100131

Accelerating Fourier Descriptor for Image Recognition Using GPU

Bahri Haythem[*], *Sayadi Fatma, Chouchene Marwa, Hallek Mohamed and Atri Mohamed*

Electronics and Microelectronics Laboratory, Faculty of Sciences, University of Monastir, 5000 Monastir, Tunisia

Received: 1 Jul. 2015, Revised: 28 Aug. 2015, Accepted: 29 Aug. 2015
Published online: 1 Jan. 2016

Abstract: In the next few years, the rate of enhancement in GPUs (Graphics Processing Units) performance is expected to outshine that of CPUs (Central Processing Units), increasing the demand of the GPU as the processor chosen for image processing. In light of tremendous advance in computer vision research of recognition shape domain, we proposed a GPU technology of programming and computing to accelerate the Fourier descriptor technique invariant to color images classification. It is a simple and powerful technique to represent objects based on their shapes. It has attractive properties such as rotational, scale, and translational invariance. Since the heaviest part of Fourier descriptor computing time is the Fast Fourier Transform (FFT), we decided to bring it out on GPU. We used CUDA: Compute Unified Device Architecture, the specific programming language of GPU, and its CUFFT library to accelerate the computation of FFT. To showcase this implementation, we studied the performance of GPU versus a traditional implementation on CPU for single and double precision.

Keywords: Fast Fourier Transformation, GPU, CUDA, Generalized Fourier Descriptor, Clifford Generalized Fourier Descriptor, CUFFT

1 Introduction

Development of storing systems, transmission and acquisition systems induced the creation of a great base of image. Diversity of these databases (medicals image, objects image and faces image) makes image representation one of dynamic domain of research. Indexing image by her contain is not the only way to characterize it. We must find a speedily representation available and easily to compare. For all image, we can extract different attributes using some mathematic tools. Descriptor is a tool that represents all information of image in its coefficients vector. In our paper, we are interested in shapes descriptor and more precisely the Fourier descriptor on recognition and classification images domain. We proposed a new implementation of GFD (Generalized Fourier Descriptor) and GCFD (Generalized Clifford Fourier Descriptor) based on GPU technology. For GPU implementation, it is more important to indicate the appropriate descriptor in term of time and efficiency. Indeed, the performance and speedy factors have becoming in the same importance degree.

In fact Fourier descriptor is used as feature vector in various papers. The first ideas are started by Gauthier et al. [1], when they proposed a family of invariants in translation, rotation, and scale. Fourier descriptor is also used on object classification and image retrieval domain by Y. Raj Bahadur [2] and F. Javier Diaz [3]. F. Smach and al. [4] were implemented for the first time the GFD as a co-design (HW/SW), when the hardware part was implemented in FPGA. Finally, J. Mennesson and al. [5] have created the notion of GCFD. This method was implemented purely on software using matlab and C. In order to present some papers using the Fourier descriptor on GPU, T. Li presented in [6] an optimal method based on Fourier descriptors to detect and track an active contour of images (convex and concave) using CUDA. In [7], H. Heidari and al. implemented also a color based image retrieval system in CUDA. However no publications exist about GPU implementation of recognition shape image based Fourier descriptor. Accelerating the mechanism of Fourier descriptor on GPU is an original idea to have a suitable result of recognition for so less time.

We divided our paper in five main parts. First of all, we gave an overview of the Fourier descriptor with each pattern. Secondly, we defined CUDA hierarchical and its

[*] Corresponding author e-mail: bahri.haythem@hotmail.com

specifications architecture. Thirdly, we detailed CUDA CUFFT library and our algorithms using this last. Fourthly, we discussed about experimental results of GFD and GCFD. Finally, we compared between FFT and Fourier descriptor implementation for each pattern and precision.

2 Fourier descriptor for a color image

Fourier descriptor can be used as a feature vector component in various applications, such as:

- Image retrieval (on the web or in large database),
- Shape recognition (tumors in medical images),
- Object tracking (someone in the metro) etc ...

In case of shape recognition domain, we used Fourier descriptor for color object image processing. It gathers the full information of object in image. It conserves the invariance characters of geometric transform, noisy and lighting. Fourier descriptor can be in global mode representing all pixels of image or in local mode, where it shows some regions and interesting points. We quote among shapes descriptor: Fourier descriptor, geometric moments ...

In this context, we studied the Fourier descriptor for recognition and classification images. Complex Fourier Transform or Fast Fourier Transform (FFT) is used by Fourier descriptor to extract different features of an object in an image. Those features were represented in a coefficients vector that will be used as an input of classifier (SVM, neuron network ...) to identify their class.

2.1 Fourier transform

2.1.1 Definition

Fourier transform is used in a wide range of applications, such as image analysis, image filtering, image reconstruction and image compression. It is the analogous of theoretical Fourier series for non-periodic functions. It is an operator that transforms an integral function to another function giving its frequency spectrum. This transform is an important image processing tool which is used to decompose an image into its sine and cosine components [8]. The output of the transformation represents the image in the Fourier or frequency domain, while the input image is the spatial domain equivalent. In the Fourier domain image, each point represents a particular frequency contained in the spatial domain image. The Fourier transform of spatial function $f(x)$ gives a frequency function $(equation\ 1)$. It can be recovered without losing information using two variables u and v $(equation\ 2)$[1]:

$$F(v) = \int_{-\infty}^{+\infty} f(x)\, e^{-(jvx)} \mathrm{d}x \tag{1}$$

$$F(u,v) = \iint_{-\infty}^{+\infty} f(x,y)\, e^{-j2\Pi(ux+vy)} \mathrm{d}x\mathrm{d}y \tag{2}$$

2.1.2 Fast Fourier Transform in image processing

For a digital image processing, Fourier descriptor used DFT (Discrete Fourier Transform) which is the basis of most digital images processing $(equation\ 3)$. The intensity of digital image f (x,y) defined in R^2 function will be transformed in R^2 frequency function using DFT $(equation\ 4)$:

$$F(u) = \sum_{1}^{M} f(x)\, e^{-j2\Pi(\frac{ux}{M})} \tag{3}$$

$$F(u,v) = \sum_{x=1}^{M}\sum_{y=1}^{N} f(x,y)\, e^{-j2\Pi(\frac{ux}{M}+\frac{vy}{N})} \tag{4}$$

For a discrete function, DFT uses a summation operator, while for a continuous function, it uses an integral operator. DFT is a sampled Fourier transform; therefore it does not contain all frequencies forming an image. Only a set of samples is large enough to fully describe the spatial domain image. The FFT is based on the complex DFT, a developed model of the real DFT. Even with these computational savings, the ordinary one dimensional DFT has N^2 complexity but FFT can reduce it to Nlog2N. There are various forms of FFT but most of them restrict the size of the input image that may be transformed. It is used to access the geometric characteristics of a spatial domain image. In most implementation the Fourier image is shifted in such a way that the output image $F(0,0)$ is displayed in the center.

2.2 GFD: Generalized Fourier descriptor

The image intensity relations were defined in Cartesian coordinates but D.Zhang and G.Lu [9] have used them in 2 dimensions Polar coordinates.

$$F(\rho,\theta) = \sum_{r}\sum_{i} f(r,\theta_i)\, e^{j2\Pi(\frac{r}{R}\rho+\frac{2\Pi}{T}\varphi)} \tag{5}$$

A classical method applied on GFD, consist in computing Fourier descriptor vector on each color channel separately. The resulting vectors will be concatenated in simply vector as an input of classifier.

[1] *y, u: Spatial and frequency variable (as respectively of x, v). u=1...M and v=1...N. j^2= -1.*

This approach is given by F. Smach and al. [10] defined some invariants for all digital color images. These invariants are noted by this following relation:

$$D_f(r) = \int_0^{2\Pi} \| f(r,\theta)\|^2 \, \mathrm{d}\theta \tag{6}$$

The integral is replaced by a finite sum in discrete domain to produce the Fourier descriptor vector. In fact, we summarize the compute of descriptors in five main steps. Firstly, we decompose a color image into 3 separate channels images (red, blue and green). Then, we apply the FFT algorithm for each channel. Therefore we sum the square module of FFT located on the circle of radius ”r” [4]. Then, we gather these values to construct the Fourier descriptor vector $D_f(r)$. Finally, we normalize all coefficients by $D_f(0)$ (E.g. $FD_b(1) = D_{fb}(1)/D_{fb}(0)$). The following figure shows how we compute $FD_b(r)$ for each channels of image:

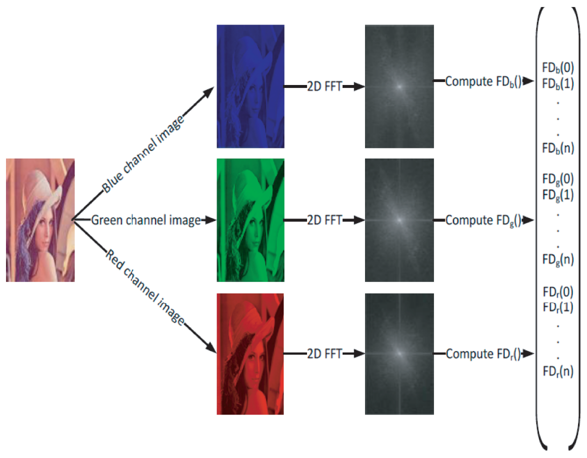

Fig. 1: Compute of Generalize Fourier Descriptor

2.3 GCFD: Generalized Clifford Fourier Descriptor

In this section, we presented the second model of Fourier descriptor: GCFD or Generalized Clifford Fourier Descriptor. It can minimize the loss of data relative of GFD. This descriptor is based on Fourier transform defined in Clifford algebra [5]. For this transform, it used 2 bi-vectors B and I_4B, they are equivalent of two perpendicular plans [11]. To bring out the totality of color image information, the same method consists to compute the Fourier descriptor vector. For GCFD, we worked in two channels instead of three channels. To define GCFD, we decomposed an image into parallel part according the bi-vector B and other perpendicular part according I_4B. We chose a $B_b = blue \wedge e_4$ as a bi-vector replacing B. This choice specifies the analysis plan and generates another orthogonal plan. We noticed that decomposing of color image is sensitive to the bi-vector chosen. It varied the result of descriptor, thereafter the precision of classification. When applying the Clifford Fourier Transform (CFT), we obtained some invariant noted by GCFD, it can be defined in 2 vectors $GCFD_{\|B}$ and $GCFD_{\perp B}$:

$$GCFD_{\|B} = \int_0^{2\Pi} \left\|F_{\|B}(r,\theta)\right\|^2 \mathrm{d}\theta \tag{7}$$

$$GCFD_{\perp B} = \int_0^{2\Pi} \|F_{\perp B}(r,\theta)\|^2 \, \mathrm{d}\theta \tag{8}$$

We determinate the $GCFD_{\|B}$ values when applying the CFT in a parallel analyze plan of B. As the $GCFD_{\perp B}$ values were obtained by the same transform for the second plan I_4B. Applying the CFT for a function allows decomposing them in parallel and perpendicular parts. To compute the Fourier descriptor for each component, we follow the same steps of GFD for each plan. Firstly, we compute the CFT for the both channels. Next, we determine the module of CFT and its square values. Then we extract the coefficients vector [12] and finally we normalize these coefficients with the first term.

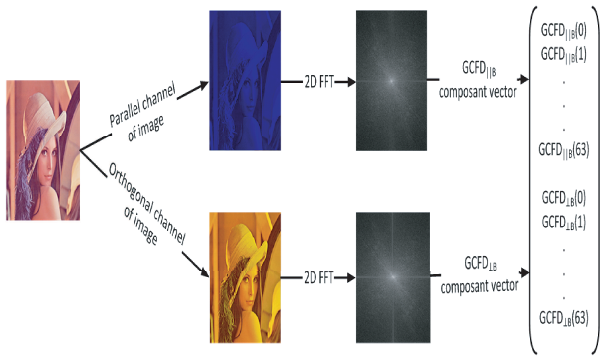

Fig. 2: Compute of Generalized Clifford Fourier Descriptor

The GCFD vector is the concatenation of the parallel and orthogonal part of descriptor. The number of coefficients for this descriptor is $2 \times m$ instead of $3 \times m$ from the marginal method of GFD.

3 Parallelism approaches of CUDA

FFT is used in image transform, video compression, speech and audio. For each field it makes its specific signification, it has a compute intensive and big volume of data [13]. For this reason, we estimated that GPU based FFT algorithm can be the cost effective solution. In fact GPU has becoming a hot research topic that works in image processing. It can execute the FFT application in parallel model using the SIMD (Single Instruction Multiple Data) mode. To execute this transform, we used a programming language of graphics card which is CUDA (Compute Unified Device Architecture).

3.1 Programming under GPU : CUDA

CUDA is a programming language under GPU. It was introduced by the NVIDIA Corporation in November 2006. This architecture provides a complete solution for General Purpose-Computing on Graphics Processing Unit, including new hardware, instruction sets and programming models. The API (Application Programming Interface) of CUDA allows communication between the CPU and GPU, ultimately allowing the user to control the execution of code on the GPU with the same degree of control as on CPU. It primarily allows the C programming language to be used as a high-level interface, although other languages are supported. Other important features are flexibility of data structures and explicit access on the different physical memory levels of the GPU. CUDA presents also a good framework for programmers including a compiler, CUDA Software Development Kit (CUDA SDK), a debugger, a profiler, CUFFT and CUBLAS scientific libraries [14].

3.2 CUDA Programming Hierarchical

The CUDA computing engine virtualizes graphics hardware available to the programmer through the use of uniquely numbered threads that are organized into 1D, 2D, or 3D blocks of arbitrary size [15]. The threads are executed on the graphics device equipped with a GPU, hereafter referred to as the device, serving as a coprocessor that enhances the computational capabilities of the workstation, referred to as the host (CPU).

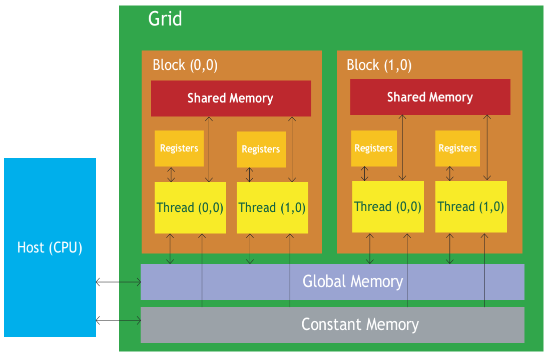

Fig. 3: GPU Device Memory Hierarchy [16]

The memory of the device presented in figure 3, is disjoint from the memory of the host, making it necessary to allocate and transfer blocks of data to the device prior to executing threads. In addition to off-chip random access memory, termed global memory, the device offers a limited amount of low-latency on-chip memory accessible to all threads within a block, referred to as shared memory. On-chip memory is also available in the form of registers, which are only accessible to individual threads. The device code is encapsulated in special functions called kernels that are invoked by the host, and executed in parallel by multiple threads. At run time, the threads within the block are executed in groups of 32, called warps. The execution follows the single instruction multiple thread (SIMT) model. This model guarantees parallel execution as long as the threads in a warp do not experience a divergence of code due to branching instructions. To ensure peak performance, it is imperative to maximize the occupancy of the multiprocessors and to minimize the latency associated with global memory access by selecting the appropriate granularity of computations and the proper assignment of thread block dimensions [18].

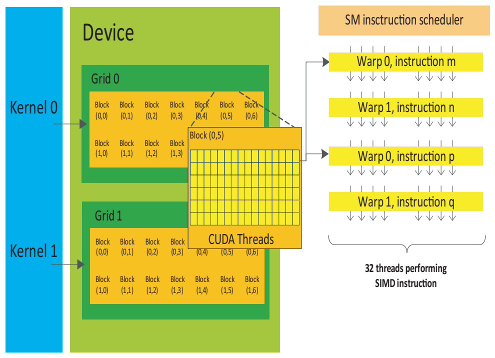

Fig. 4: CUDA thread organization and execution [17]

Each CUDA kernel runs a virtualized set of threads which are hierarchically organized into three dimensional thread blocks. The programming model conceptually partitions these thread blocks into a three dimensional grid (left side of Fig4 above). CUDA dynamically assigns each thread block to a single streaming multiprocessor. A thread block is further broken down into a collection of multiple warps, each group of scalar threads that execute in SIMD fashion (right side of Fig4). This implicit SIMD model can reduce the programming complexity and improve the code maintainability. However, this model may limit the instruction throughput on SIMD vector units since warp formation is statically determined.

4 CUFFT library of CUDA

As a new method of implementation on GPU, we used a specific library of CUDA programming language which is

CUFFT [19,20]. It is the CUDA FFT library that implements the Fourier transform algorithms and provides a simple interface for computing parallel FFT on GPU. This allows users to leverage the floating point power and parallelism of the GPU without having to develop a custom. The CUFFT library implements several FFT algorithms, each with different performances and accuracy. The 4.0 version [21] of the CUFFT library supports the following features:

- 1D, 2D, and 3D transforms of complex and realvalued data
- Batch execution for doing multiple transforms of any dimension in parallel
- Transform sizes up to 64 million elements in single precision and up to 128 million elements in double precision in any dimension, limited by the available GPU memory
- Inplace and outofplace transforms for real and complex data
- Doubleprecision transforms on compatible hardware
- Support for streamed execution, enabling simultaneous computation together with data movement . . .

Algorithm 1: 2D FFT of an image using CUDA (CPU part)

```
Input:
img_input                  ▷ the input image
matrix_img_h               ▷ complex image matrix in the host
matrix_img_d               ▷ complex image matrix in the device
plan                       ▷ the transform place of CUFFT
Output:
matrix_img_Rh              ▷ real image matrix in the host
matrix_img_Rd              ▷ real image matrix in the device
img_ouput                  ▷ the output image
1  for $i \leftarrow 0$ to $width$ do
2      for $j \leftarrow 0$ to $height$ do
3          $matrix\_img\_h(i*width+j).x \leftarrow img\_input(i,j)$
4          $matrix\_img\_h(i*width+j).y \leftarrow 0$
5  $matrix\_img\_Rd \leftarrow matrix\_img\_Rh$
6  $matrix\_img\_d \leftarrow matrix\_img\_h$
7  cufftPlan2d(plan, CUFFT_Z2Z)   ▷ 2D transform plan for complexe data
8  cufftExecZ2Z (plan, matrix_img_d, matrix_img_d)   ▷ complex to complex transform function
9  FFT_shift (matrix_img_Rd, matrix_img_d)   ▷ FFT shifting kernel
10 for $l \leftarrow 0$ to $width$ do
11     for $k \leftarrow 0$ to $height$ do
12         $img\_output(l,k) \leftarrow matrix\_img\_Rh(l*width+k)$
13 return $img\_output$
```

Applying CUFFT can be resumed in 4 easy steps. Initially, we should allocate space on GPU where the FFT functions compute its algorithm [22,23,24]. Then,we create plan specifying transform configuration like the size and type (real, complex, 1D, 2D and so on). The following step is to execute the plan as many times as required, providing the pointer to the GPU data created in the first step. Finally, we destroy the space initially created to release the GPU memory.

Algorithm 2: KERNEL of FFT_shift on CUDA (GPU part)

```
Input:
matrix_img       ▷ complex image matrix in the device
aux              ▷ auxiliary matrix for exchange
Output:
matrix_img_R     ▷ real image matrix in the device
1  col1 : 0 to width/2   ▷ pointer to the x-coordinate until the middle of image
2  row1 : 0 to height/2  ▷ pointer to the y-coordinate until the middle of image
3  col2 : width/2 to width   ▷ pointer to the x-coordinate from the middle until the end of image
4  row2 : height/2 to height ▷ pointer to the y-coordinate from the middle until the end of image
5  Phase 1: Exchange the data between the four quadrants   Exchange the data between the first quadrant and the third quadrant for the real part.
6  $aux(row1*width+col1) \leftarrow matrix\_img(row1*width+col1).x$
7  $matrix\_img(row1*width+col1).x \leftarrow matrix\_img(row2*width+col2).x$
8  $matrix\_img(row2*width+col1).x \leftarrow aux(row1*width+col1)$
9  Do the same step for the imaginary part of the previous quadrants.   Exchange the data between the second quadrant and the forth quadrant for the real part.
10 $aux(row1*width+col1) \leftarrow matrix\_img(row1*width+col2).x$
11 $matrix\_img(row1*width+col2).x \leftarrow matrix\_img(row2*width+col1).x$
12 $matrix\_img(row2*width+col1).x \leftarrow aux(row1*width+col1)$
13 Do the same step for the imaginary part of the previous quadrants. Phase 2: Define the module of those previous results
14 $matrix\_img\_R(row1*width+col1) \leftarrow \{matrix\_img(row1*width+col1).x*matrix\_img(row1*width+col1).x+matrix\_img(row1*width+col1).y*matrix\_img(row1*width+col1).y\}$
15 Do in the same way the second, the third and the fourth quadrants.
16 return $matrix\_img\_R$
```

For more details, we tried to define some programming specification of CUFFT library API [23]. First of all, CUFFT stands to compute real and complex data for each precision such as *cufftReal*, *cufftComplex*, *cufftDoubleReal* and *cufftDoubleComplex*. This temporary space will be allocated separately for each individual plan. *CufftPlan1D()*, *cufftPlan2D()* and *cufftPlan3D()* create a simple plan for a *1D*, *2D* and *3D* transform respectively. Then, when the execution function is called, the actual transform takes place following the plan of execution. Before talking about execution, we should notice that *cufftHandle* creates after *cufftPlan()* to store and access CUFFT plans. *CufftExecC2C* (or *cufftExecZ2Z*) executes a single precision (or double precision) complex to complex transform plan in the transform direction as specified by direction parameter *CUFFT_FORWARD* (or *CUFFT_INVERSE*). Also *cufftExecR2C* (or *cufftExecD2Z*) executes a single precision (or double precision) real to complex (implicitly forward) CUFFT transform plan. At the end of the transformation, we free all GPU resources associated with a CUFFT plan and destroy the internal plan data structure by *cufftDestroy()*. In Algorithm 1, extract of CUDA C programs is presented to show how CUFFT used on Fourier descriptor. We describe also how CUFFT used the GPU memory pointed to by the input data parameter. Then the execution function stores the Fourier coefficients in the output data array. Besides this, we added a relevant kernel called *fft_shift* to reorder the output of Fourier transform. This kernel is shown in Algorithm 2.

5 Experimental results

For the experiment, we used NVIDIA's GeForce GT 525M graphics card based GPU with Compute Capability 2.1. It belongs to the Fermi architecture and it supports 96 CUDA cores, running at 1.2 GHz. It is connected with Intel® Core^{TM} i3-2350M based CPU with a clock speed 2.30 GHz built in x64 PC Main Boards PCI Express. For smooth running the process of FFT based image recognition using GPU, we have used different resolution of images to figure out the performance of FFT on CPU and GPU based implementation for Fourier descriptor. Images are chosen with an odd resolution for some processing obligation. CPU performance results are obtained by computing FFT of images in C++ using OpenCV and measuring the execution time using the Visual Studio special profiler. Then GPU performance results are obtained by applying the CUFFT library for each image in CUDA and measuring the execution time using the special NVIDIA Compute Visual Profiler.

5.1 Processing of GFD computing for a color image

According to these experimental materials features, we started processing image with the first pattern of Fourier descriptor which is GFD. We divided our work in two parts: firstly we processed the FFT function for each image on both computing platforms (CPU and GPU). Secondly, we processed the totality of GFD also on both computing platforms. Based on the processing described previously, the results measured can be seen in table I. The time is shown in microseconds and is calculated from when either the CPU or GPU starts allocating memory and calculating the FFT response to the FFT responses are ready for use in host memory.

Table 1: FFT features on GFD

Image size	Processing time of FFT		Speedup factor of FFT
	CPU (ms)	GPU (ms)	
15×15	0.75	0.1	7.5×
31×31	1.55	0.13	11.9×
63×63	5.77	0.28	20.6×
127×127	20.22	0.89	22.7×
255×255	85.61	3.52	24.3×
511×511	380.03	18.36	20.7×
1023×1023	1626.66	103.4	15.7×

Table 2: GFD features

Image size	Processing time of GFD		Speedup factor of GFD
	CPU (ms)	GPU (ms)	
15×15	2.91	0.3	9.7×
31×31	4.98	0.97	5.1×
63×63	13.54	3.7	3.6×
127×127	47.13	15.35	3×
255×255	197.68	61.47	3.2×
511×511	552.76	90.77	6×
1023×1023	1981.25	182.68	10.8×

As expected, GPU implementation outperforms CPU version in all size of image. The previous table indicates that GPU seems to run FFT faster than CPU. While increasing the image size, the results show an improvement performance of GPU as about 1.52 second as a difference with CPU for 1023x1023 resolutions. It can be noticed from table 1 that the speedup factor can reach 24X at 255x255 image size. The speedup of FFT computing was impacting certainly in the full treatment of GFD. The results in table 2 show that GPU reduced enormously the GFD execution time. We notice that the speedup factor can reach in highest 10X.

5.2 Processing of CGFD for a color image

As the first model of Fourier descriptor, we implemented the GCFD on both processors. This descriptor includes less proceeding than GFD. On one hand, we reduced the execution time of processing; on the other hand, we used a shorter descriptor vector than GFD.

Table 3: FFT features on GCFD

Image size	Processing time of FFT		Speedup factor of FFT
	CPU (ms)	GPU (ms)	
15×15	0.26	0.07	3.7×
31×31	0.44	0.09	4.8×
63×63	0.95	0.18	5.2×
127×127	2.86	0.59	4.8×
255×255	14.82	2.34	6.3×
511×511	45.18	12.14	3.7×
1023×1023	185.42	64.62	2.8×

Table 4: GCFD features

Image size	Processing time of GCFD		Speedup factor of GCFD
	CPU (ms)	GPU (ms)	
15×15	2.48	0.2	12.4×
31×31	4.5	0.79	5.7×
63×63	11.19	2.54	4.4×
127×127	42.84	12.45	3.4×
255×255	159.91	50.17	3.1×
511×511	484.3	60.01	8×
1023×1023	1848.72	99.31	18.6×

Table 3 shows the comparison time of FFT computing of GCFD between CPU and GPU. This last present a great enhancement versus CPU, since the difference time can reach 0.12 second. In the same table, the speedup factor is outstanding for all image size. Maximum speedup achieved 6.3X in 255x255 image size, whereas minimum is 2.8X in 1023x1023 resolutions. This performance will decrease the execution time of GCFD computing. As shown in the table 4 above, GPU highly reduces the time of computing. It achieves about 18X as a speedup for 1023x1023 resolutions.

6 A comparative study of implementation results

To showcase this work, we studied the efficiency of each implementation for single and double precision. We started this study by comparing the result of FFT implementation on both computing platforms. Then, we compared GFD and GCFD implementation on CPU than on GPU.

6.1 Comparative study of FFT

6.1.1 FFT of GFD

In Figure 5, we observe that the performance boost for FFT is gradually enhanced. This performance for both CPU and GPU is nearly the same for image size between 15x15 until 63x63. On both descriptors, while the resolution increases the performance of CPU still delayed versus GPU.

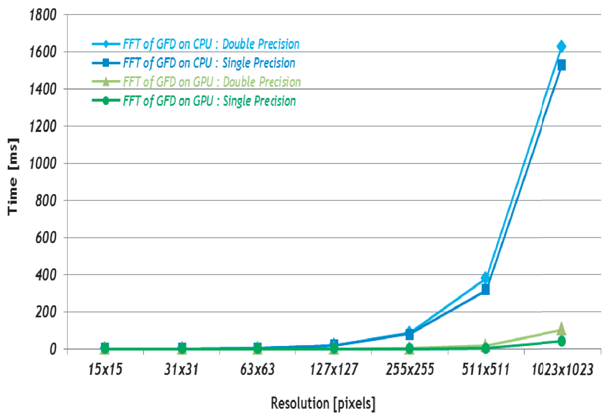

Fig. 5: FFT for GFD in single and double precision: CPU & GPU

In the same figure, we show the relative performance using each different precision. For CPU, both curves of single and double precision are very close together and do not separate in the most points. Whereas for GPU, we notice that single precision is best than double precision from 511x511 images size but after this resolution the performance is nearly the same.

6.1.2 FFT of GCFD

In the same context, we compared the execution time of FFT for GCFD. Figure 6 shows that when the size of image increases, the gap between CPU and GPU increases from the 127x127.

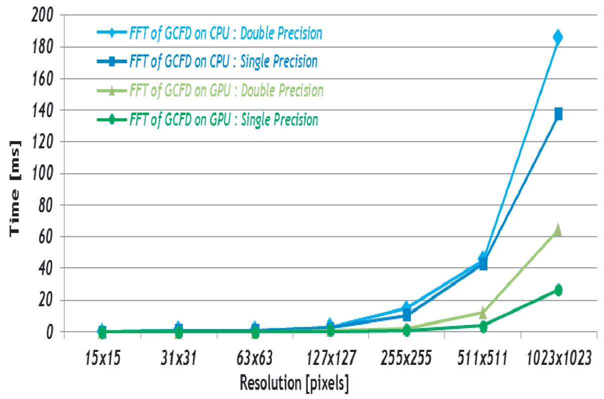

Fig. 6: FFT for GCFD in single and double precision: CPU & GPU

In the same figure, we should focus the reduction of execution time when using single precision on both platforms. For CPU the gap between both precision started from 511x511 image size while for GPU it began from 255x255.

6.2 Comparative study of Fourier descriptor

In this work we are interested mainly in the execution time of Fourier descriptor, therefore and by analogy to the previous study, we divided this part according to the descriptor type and the precision type.

6.2.1 GFD

Figure 7 below shows that the performance of GFD on CPU and GPU is nearly common before 127x127 image sizes. For a large resolution, GPU performs better results than those of CPU, so if we increase the size of image, the margin rises immediately.

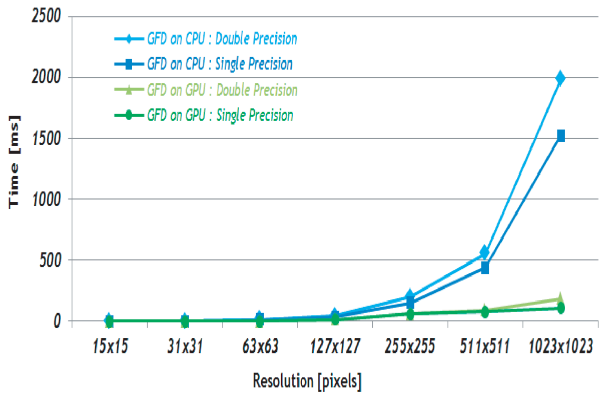

Fig. 7: GFD in single and double precision: CPU & GPU

The results of execution time for GFD indicate a great performance of single precision then double precision. This brings more flexibility of programming giving the same results. On CPU, single precision reduces enormously the execution time compared to double precision. For 1023x1023 image size, it can reach up to 455 ms of execution time difference. However, on GPU the difference between simple and double precision achieved only 72 ms.

6.2.2 GCFD

For the second pattern of descriptor GCFD, figure 8 shows that single precision get best time versus double precision for both CPU and GPU. It can reduce up to 383 ms on CPU and up to 24 ms on GPU. Implementing GCFD on GPU makes fewer vectors of descriptor and less time. This method improves better recognition and classification than classical method.

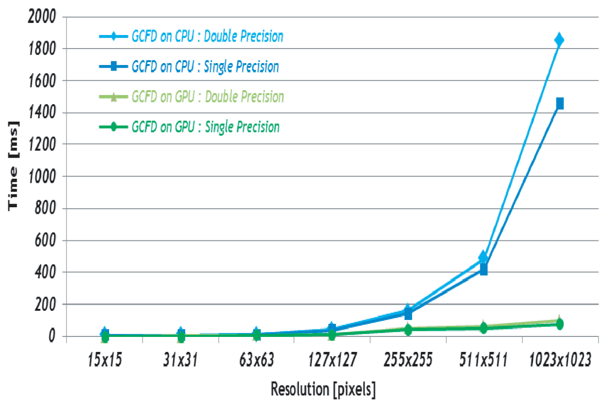

Fig. 8: GCFD in single and double precision: CPU & GPU

6.3 Comparative study of speedup factor

6.3.1 Speedup of FFT

To showcase the performance of this work, we compared the speedup of each implementation of FFT for the both models of descriptor. In figure 9, we notice that the performance of GPU is more important for GFD than GCFD. We noted that the least value of speedup is around 2.8X and the highest can reach 24.3X.

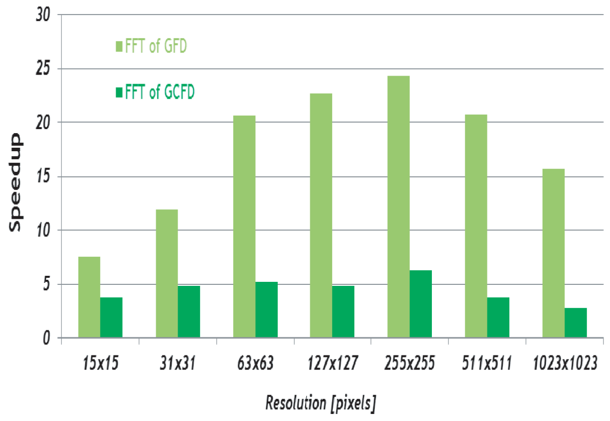

Fig. 9: Speedup of FFT: GFD & GCFD

Due to their massive parallel architecture, using GPU enables the completion of computationally intensive assignments much faster compared with CPU. In fact, we realized that GFD has a speedup better then GCFD because the feature of heavy work of GPU. This is why GPU has enormous potential particularly in areas where data and compute intensive basic research requires the processing of large volumes of measurement data. When we have more data, the performance of GPU was great.

6.3.2 Speedup of Fourier descriptor

To finish this study, we compared the speedup of GFD and GCFD average time: We found that GCFD achieves superior speedups compared to GFD; it can reach 18.6X of speedup. In otherwise, GFD gives maximum 10.8X of speedup according to figure 10.

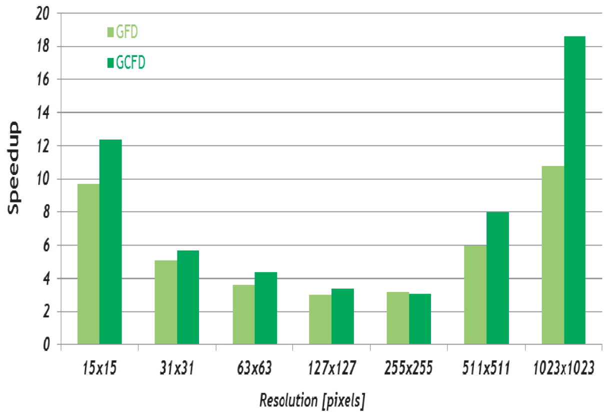

Fig. 10: Speedup of GFD & GCFD

7 Conclusion and Future Work

This paper describes the acceleration of the Fourier descriptor vector computing using CUDA platform and discusses the feasibility and profit of different kinds of optimized CUDA program. We perform also the computing of FFT, which is the heaviest part of GFD and GCFD using CUDA CUFFT library. We can find out that a good CUDA program could speed up the parallel programs significantly. Hence, we optimize the computing time of Fourier descriptor vector. Then, we compare between both models of descriptor for single and double precision and finally we show that GPU success to reduce enormously the execution time. As future work, we proposed to realize the classification part on GPU using the descriptor vector to perform the SVM based classification and we compare it with another implementation on a FPGA.

References

[1] J.P. Gauthier, G. Bornard, M. Silbermann. Harmonic analysis on motions groups and their homogenous spaces. IEEE transactions on systems, man, and cybernetics, 1991, 159-172.

[2] Y.R. Bahadur, K. Naveen Nishchal, K. Arun Gupta, K. Vinod Rastogi. Retrieval and classification of shape-based objects using Fourier, generic Fourier, and wavelet Fourier Descriptors technique: a comparative study. Optics and Lasers in Engineering, ScienceDirect, 2007, 695-708.

[3] F.J. Diaz, A.M. Buron, J.M. Solana. Haar wavelet based processor scheme for image coding with low circuit complexity. Computers and Electrical Engineering, SienceDirect, 2007, 109-126.

[4] F. Smach, C. Lematre, J.P. Gauthier, J. Miteran, M. Atri. An FPGA-based accelerator for Fourier descriptors computing for color object recognition using SVM. Journal of Real-Time Image Processing, 2007, 249-258.

[5] J. Mennesson, C. Saint-Jean, L. Mascarilla. Color Object Recognition Based On Clifford Fourier Transform. Guide to Geometric Algebra in Practice, Springer Verlag, 2011, 175-191.

[6] T. Li. Control a robot tele-echography by visual servoing. PhD thesis, University of Rennes, 2013.

[7] H. Heidari, A. Chalechale and A.A. Mohammadabadi. Parallel Implementation of Color Based Image Retrieval Using CUDA on the GPU. International Journal Information Technology and Computer Science, 2014, 33-40.

[8] R. Fisher, S. Perkins, A. Walker, E. WolfartImage. Processing learning resources, Contents and Index, Image Transforms, Fourier Transform (2003). [Online]. Available: http://homepages.inf.ed.ac.uk/rbf/HIPR2/fourier.htm.

[9] D. Zhang, G. Lu. Shape-based image retrieval using generic Fourier descriptor. Signal Processing: Image Communication, 2002, 825-848.

[10] F. Smach, C. Lemaitre, J.P. Gauthier, J. Miteran, M. Atri. Generalized Fourier descriptors with applications to objects recognition in svm context. Journal of Mathematical Imaging and Vision, 2008, 4371.

[11] J. Mennesson, C. Saint-Jean, L. Mascarilla. A new set of Clifford Fourier descriptors for color images. The GCFD3. Signal processing (signal, image, speech), 2012, 359-382.

[12] J. Mennesson. Frequency methods for color image recognition - An approach based on Clifford algebras. PhD thesis, University of Rochelle, 2011.

[13] M. Nazmul Haque, M. Shorif Uddin. Accelerating Fast Fourier Transformation for Image Processing using Graphics Processing Unit. Journal of Emerging Trends in Computing and Information Sciences CIS journal, 2011, 367-375.

[14] J. Balfour. Introduction to CUDA CME343/ME339. NVIDIA Research, 2011.

[15] C. Ding. CUDA Tutorial: The Golden Energy Computing Organization. [Online]. Available: http://geco.mines.edu/tesla/cuda_tutorial_mio/index.html.

[16] Best Practices in GPU-Based Video Processing. GTC : GPU technology conference, 2012

[17] T. Nguyen, D. Hefenbrock, J. Oberg, R. Kastner, S. Baden. A software-based dynamic-warp scheduling approach for load-balancing the ViolaJones face detection algorithm on GPUs. Journal of Parallel and Distributed Computing, 2013, 677-685.

[18] NVIDIA. NVIDIA CUDA Programming Guide Version 4.0. NVIDIA Corporation, 2011.

[19] NVIDIA. NVIDIA CUDA Programming Model Overview. NVIDIA Corporation, 2006.

[20] M. Fatica. CUDA Libraries and CUDA FORTRAN. NVIDIA Corporation, 2011.

[21] NVIDIA. CUDA CUFFT Library Version 4.0. NVIDIA Corporation, 2011.

[22] C. Trujillo, J.G. Sucerquia. Graphics processing units more than the pathway to realistic video games. National University of Colombia, Dyna: general scientific publication of the area of technological sciences, 2011, 164-172.
[23] NVIDIA. CUFFT LIBRARY USER'S GUIDE. NVIDIA Corporation, 2012.
[24] M. Abdellah, S. Saleh, A. Eldeib, A. Shaarawi. High Performance Multi-dimensional (2D/3D) FFT-Shift Implementation on Graphics Processing Units (GPUs). 6th Cairo International Biomedical Engineering Conference, Cairo, Egypt, 2012.

Bahri Haythem is currently a PhD student at Laboratory of Electronics and Micro-electronics the University of Monastir Tunisia. He received a M.S degree in micro-electronics and nano-electronics from Faculty of Science of Monastir, Tunisia in 2012. His research interests are focused on processing image and video in graphics processor.

Sayadi Fatma was born in 1974. She received the PhD Degree in Micro-electronics from Faculty of Science of Monastir, Tunisia in collaboration with the LESTER Laboratory, University of South Brittany Lorient FRANCE., in 2006. She is currently a member of the Laboratory of Electronics & Micro-electronics. Her research includes image and video processing in graphics processor, motion tracking and pattern recognition, circuit and system design.

Chouchene Marwa is currently a PhD student at Laboratory of Electronics and Micro-electronics the University of Monastir Tunisia. She received a M.S. degree in Electronic Materials and Devices from Faculty of Science of Monastir, Tunisia, in 2010. Her research includes image and video processing in graphics processor, motion tracking and pattern recognition, multimedia application, video surveillance.

Hallek Mohamed is currently a PhD student at Laboratory of Electronics and Micro-electronics the University of Monastir Tunisia. He received a M.S degree in micro-electronics and nano-electronics from Faculty of Science of Monastir, Tunisia in 2012. His research interests are focused on pattern recognition and stereo matching.

Atri Mohamed born in 1971, he received his PhD Degree in Micro-electronics from Faculty of Science of Monastir, Tunisia, in 2001 and his Habilitation in 2011. He is currently a member of the Laboratory of Electronics & Micro-electronics. His research includes circuit and system design, pattern recognition, image and video processing.