Parallel Computing Solutions for Linear Combination of Filters

Andrea Cavarra
University of Catania
Viale A. Doria 6, 95125 Catania, Italy

Dario Caramagno
University of Catania
Viale A. Doria 6, 95125 Catania, Italy

***Abstract*—The GPU (Graphics Processing Unit) are the future of high performance computing and provides a parallel programming model for general purpose applications thanks to CUDA programming interface. The programming model of GPU architecture is significantly different from the traditional CPU one. This paper presents the advantages of GPU architecture by proposing an algorithm for the linear combination of digital filters in image processing, implemented as a parallel GPU version (a sequential CPU version has been also implemented for comparison). The use of parallel processing CUDA architecture has enabled us to take advantage of the GPU allowing an increase of the performance compared to CPU.**

***Index Terms*—Matrices Convolution, Image processing, GPGPU, Parallel Computing, CUDA.**

I. Introduction

The evolution of computers has allowed to develop more complex calculations and has improved the performance of the simulations in scientific fields. Technology has increased the CPU (Central Processing Unit) frequency with the upper limit of 3 GHz, processing multiple cores: dual core, quad core and octa core in a few decades, unfortunately with the limits due to power dissipation and the increasing temperature.

One of the solutions proposed to push such limits is the use of the GPU (Graphics Processing Unit). Typically, GPUs handle the huge amount of data for graphical applications in three-dimensions with high performances, so the GPGPU (General-Purpose Computing on GPU) has been introduced. GPGPU programming has been implemented using two types of APIs (Application Programming Interface): the complex OpenGL [1] and DirectX [2].

In the recent years, the computer house NVIDIA has developed CUDA (Compute Unified Device Architecture). This platform is much simpler than the previous APIs and and it allows to program using a high level programming language based on C, using a model of parallel computing. NVIDIA CUDA technology has opened a new era for GPGPU computing allowing the design and implementation of parallel GPU-oriented algorithms without needing any knowledge on OpenGL. The computational power of these architectures, nowadays, has received a considerable growth compared to CPUs and due to the GPU ability to perform a huge number of simple operations in parallel. GPUs have cores much simpler compared to CPUs. This allows the realization of GPU architectures in which the silicon area for control and management is very reduced compared to that employed in the CPU and a big increase in the number of cores [3].

GPUs allow us to compute optimally, in parallel mode, a code with a certain complexity and for the application of digital filters in two-dimension arrays. This paper proposes the use of GPU architecture for the implementation of digital filters with a parallel code that uses a linear combination of digital filters processing images. The aim is to compare the parallel execution speed to the sequential code. The remaining parts of the paper will be dedicated to explain the parallel and serial approaches.

II. Digital Filters

A digital filter performs mathematical operations on discrete-time sampled signals in order to enhance or reduce certain characteristics of the signal. Such filters are implemented by software components, often through mathematical functions or matrices, and loaded inside the processors with programmable hardware. Cost and speed are closely dependent on the used processor.

The application of digital filters has enormous advantages over analog filters as the possible transfer functions are much more flexible for digital filters. The main advantages are:

1) high accuracy due to the absence of physical components;
2) a digital filter is easier to design and implement automatically modifing its frequency response and changing the input;
3) flexibility to change the digital filter parameters, without changing the system hardware;
4) easy simulation and design of the filters being implemented by software with a reduction of system complexity.

these properties are essential to the implementation of a high quality filter. As mentioned previously these advantages are accompanied by the limitations of speed and cost, related to the processor used, and frequency. The frequency limit is described by Nyquist theorem, using the following formula:

$$f_s > 2B \tag{1}$$

which imposes the filter with a maximum limit of the frequencies in order to reduce the effects of additional aliasing distortion of the signals. B term is the signal band and f_s

is sampling frequency. The digital filters are widely used in image filtering and image processing. This paper will examine, implement and test an algorithm to filter digital images with a GPU implementation.

A. Digital filters for image processing

Digital filters are an essential tool for image processing. The digital image can be defined as a two dimensional array or matrix and each element represents a pixel of the image. Images are processed by an algorithm. The process of filtering is also known as convolution of a mask with a standard size 3x3, 5x5 or 9x9, which is a matrix. This mask applies different mathematical operators by means of a convolution to the image to achieve digital image processing.

The use of filters to digital image processing allows us to make operations like:

1) the extraction of information from the image, such as the detection of boundaries in images;
2) the exaltation of details, such as the increase of light intensity or color contrast;
3) the elimination and reduction of disturbances in images.

This processing brings a high computational cost especially for large images. For this reason a sequential approach, based on the CPU, is not very efficient, so parallel approaches have been introduced to increase performances. This paper presents an algorithm for the application of a linear combination of filters to local processing based on a model of parallel computation on GPU architectures. For our local processing filters, the image processing method consists of applying a function to each original pixel values and to an appropriate range of pixels, within the radius of the filter matrix. This method is based on the convolution operator for which a brief explanation is needed.

B. Convolution

As previously said, the algorithm executes an operation on an input matrix and pre-established filters, in order to provide as a resalt a linear combination of the input. The 2D convolution between two continuous functions, $f(x, y)$ and $g(x, y)$, is defined by the formula:

$$f(x,y) * g(x,y) = \int_{-\infty}^{\infty} f(\alpha,\beta) g(x-\alpha, y-\beta) d\alpha d\beta \tag{2}$$

This expression will be brought to the discrete domain, by assuming $f(x, y)$ and $g(x, y)$ be two discrete arrays of a limited size, as a double sum expressed by:

$$f(x,y) * g(x,y) = \sum_{\alpha=0}^{M-1} \sum_{\beta=0}^{N-1} f(m,n) g(x-\alpha, y-\beta) d\alpha d\beta \tag{3}$$

for $x = 0, 1, \ldots, M-1$ and $y = 0, 1, \ldots, N-1$.

This convolution operation is defined on discrete-time as a simple operation of "local media" in a range of amplitude defined by the kernel size used obtaining a value for the output matrix for the same position of the source data as shown in figure 1.

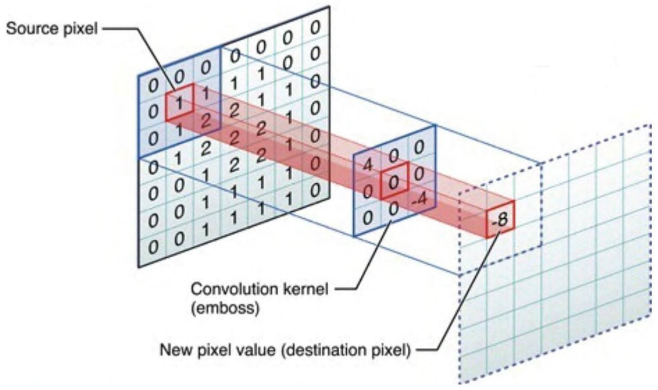
Source pixel
Convolution kernel (emboss)
New pixel value (destination pixel)
-8

Fig. 1: 2D Convolution

This operation is done from left to right, for each of the original matrix element obtaining the matrix of the resulting convolution. Therefore, the convolution algorithm implements the digital filtering operation. Then, it suffices to modify the kernel in order to change the type of filter. We have studied a digital filtering operation of am image, in ASCII format, through two filters of ninth order. The next section describes the convolution operation implemented as a parallel algorithm that can be executed on a CUDA GPU.

III. Parallel computing on GPU

This paper presents the convolution algorithm implementation for a bidimensional matrix of input data with a linear combination of filters using a parallel model. In this work two versions of the same algorithm are presented in order to compare the efficiency of the parallel computation and the sequential one, and then demonstrate how the parallel computing support is far more suitable for high performance computing.

A. GPU Architecture

In recent decades, high performance computing sector has been having an exponential growth with the widespread use of GPGPU. The use of GPU in this ever-expanding purview can exploit the power of parallel computing for increasingly complex simulations due to the inherent parallel nature of GPUs.

CUDA compatible GPUs are, in fact, based on an architecture made up of a number of MIMD (Multiple Instruction Multiple Data) multiprocessors called *Streaming MultiProcessors*, whose number depends on the specification and class of GPU performance. This *Streaming MultiProcessors* are the basic units of the GPU architecture and are implemented as SIMD (single instruction multiple data), and called by NVIDIA as SIMT (Single Instruction Multiple Threads), which has 8 processors said *Streaming Processor* or CUDA *Cores*. In this architecture, each *Streaming MultiProcessor* is able to create, manage, schedule and execute groups of 32 threads called *warp*. A warp executes one instruction at a time, so as to maximize the efficiency when all 32 threads of a warp agree on their execution path. When one or more blocks of threads are assigned to a multiprocessor to run, they are then partitioned into *warps*, scheduled by a *warp scheduler* and executed one at a time. Each of these processors can perform simple mathematical operations (such as addition, subtraction, multiplication, ect.) for integers or floating point numbers. Inside each multiprocessor there is also a shared memory, accessible only by the processors in the same multiprocessor, caches for instructions and for data, and, finally, a unit for decoding the instructions.

Each multiprocessor has access to a *global memory* shared among all GPU multiprocessors and called *Device Memory* [3]. The programming model in CUDA C organizes the program in a sequential part executed on the CPU, *host*, mainly performing memory allocation and kernel calls, and in a parallel part called kernel and executed on the GPU, *device*. This structure requires the presence, inside the program, of instructions that are sequentially executed on the host and interspersed with calls to the kernel that allow us to carry out entire parts of the program in parallel, on the device, as is shown in figure 2.

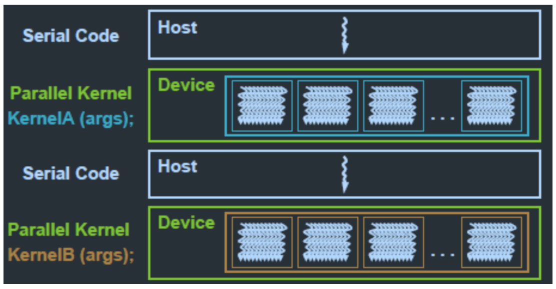

Fig. 2: Model of programming in CUDA

Kernel is the model of parallel execution of the code in a *device* and it is defined as a grid divided in to a certain number of blocks to which it is assigned a multi-processor for each. Inside each block there is a number of fundamental computational units defined as *thread*.

Kernel, or grid, are sequentially executed between them while blocks and threads are executed in parallel by adopting a SIMT data-parallel model. Each of these *thread* belongs to a single block and is uniquely identified within the *kernel* by assigning it an index. In this manner the memory addressing will be simplyfied especially in the case of processing multi-dimensional data.

In each of the blocks the kernel also has a shared memory accessible only to the threads of the same block. The logical subdivision of a *kernel* in *grid* and *blocks* is a crucial aspect in a code in CUDA for obtaining the parallelization of code. The organization and management of the internal threads allows the implementation of a more efficient code.

IV. Presentation of the Algorithm

The algorithm developed in this work executes the convolution operation between an input matrix A and some pre-established kernels generating an output matrix $\tilde{A}$ given by the linear combination of results obtained using this formula

$$A \Longrightarrow \tilde{A} = \alpha(k_1 * A) + \beta(k_2 * A) + \gamma A \qquad (4)$$

While this operation is executed, we want to generate the best parallelized code for the purpose of improving its performances compared to the serial code version. The algorithm shown can be summed up in three basic steps:

1) acquisition of input data matrix;
2) convolution of the input matrix with pre-established kernels;
3) linear combination of the results.

Obviously, each of these steps is performed by both the sequential version of the code and then the parallel one. The following will detail the basic steps previously exposed focusing our attention on the parallel algorithm.

A. *Parallel algorithm solution*

The parallel solution of the algorithm starts in a host by means of code that acquires the input and takes data from a text file in ASCII code and store it in an array on *shared memory*. Instead of a 2D array for data, complex to manage, we have used a single dimensional array where the rows of the input matrix are reported one after the other, then data are reconstructed by an index. In this way the data will be transfer from host to GPU with the standard function *cudaMemcpy()*. After a given number of threads and having organized the blocks, the call to kernel is performed, in order to do parallel computations.

Inside the *kernel* it is assigned a thread to any given input so that we can entrust the execution of the code to a set of parallel *threads* arranged in blocks and indexed using the following formula:

```
id = threadIdx.x + (blockIdx.x*blockDim.x);
```

To each thread will be assigned a position in the input array, and the thread will compute the convolution, in parallel to other threads in each block. The input data acquired in the form of array was necessary to implement a mechanism of jump allowing us to move between the various locations in the array in order to perform the products between the elements of the kernel (in our case a simple 3x3 filter), and the components located in the neighborhood of the source location. This problem is solved using the radius of the kernel defined as:

```
Kernel_radius = (Kernel_order -1)/2;
```

We can identify all the elements lying in the range of the position of interest. Then *kernel radius* is subtracted to the input array index and added a variable that increases cyclically through a simple *for* loop. This let us multiply in parallel the elements of the input with the kernel element lying in the right position around the central element of the first row of the kernel, as shown below:

```
for(i=0;i<N;i++)
    s[id]+=A[id-z+(i%3)+(i-(i%3)/3)]*k[i];
```

Then the convolution algorithm performs the operation just described for each row of the kernel:

```
for each row in kernel;
   for each element in kernel row;
      Multiply the element to the
      location of the Kernel with
      the corresponding element
      in the input data matrix.

Sum results, save them on central position.
```

This algorithm is executed as many times as the kernel number we want to apply to our array of the input data. In our specific case we chose to apply two kernels of ninth order with filter function for the input matrix obtained by ASCII encoding of a two-dimensional image. For a linear combination of the results we have used, finally, a simple program, *add*, shown below:

```
if(id<((rig-1)*(col-1)))
  sum[id]=(a*s[id]+b*s2[id]+g*A[id]);
else return;
```

The *add* program, called from the host and executed in the device, sums the results of the two performed convolutions loaded in arrays s and $s2$ with the values of the input data A. Each of these addends is regulated by a default weight which calibrates the effect of a convolution comparated to the other in the final matrix indicated with *sum* array. An input image having 256 colors has been used as input array, so to ensure the output image had the same range of color variation, two *if* statements impose this condition to the elements of *sum* array.

In the following the results and performances of the measures on the execution time in CUDA C and the C versions are compared, hence assessing parallel and sequential versions of the algorithm.

V. Experimental Results

A comparison of the execution time using the function $clock$ belonging library $time.h$ has been performed to the algorithm in the sequential and parallel versions. The execution time measures the time interval between the $start$ and the end $time$. Matrices of various sizes were used as input data in order to compute the efficiency of the two implementations when varying the amount of input data. Such matrices are matrices representing images, coming from a text file in ASCII code.

The sequential algorithm for processing data runs on a AMD i686 Dual-Core processor with up to 1GHz clock speed, while for the GPU algorithm we have used a NVIDIA GeForce GTX 480 with 480 CUDA cores and 1536 MB GDDR5 video RAM with CUDA 4.2. We developed our GPU code using NVIDIA CUDA API while the CPU code is compiled under nvcc. The GPU results are compared with the CPU results under Linux. For our simulations we used a subdivision of $kernel$ in blocks of 440 threads by applying filters of order 9.

Table I shows the sizes of the matrices used in data input and averages, calculated on 20 samples, the execution times of the sequential and parallel versions.

TABLE I: Averange Execution Time

	Averange Execution Time (s)	
Matrix Size	Sequential code	Parallel code
256x256	0.185	0.099
512x512	0.281	0.195
800x600	0.508	0.375
1024x1024	0.983	0.658
1920x1200	1.891	1.164
2024x2024	3.292	1.890
3000x3000	6.942	3.570
4096x4096	13.023	7.695
8192x8192	63.652	41.085
9000x9000	83.693	55.421

Note that the execution time of the algorithm on the GPU includes the transfer times of data (HostToDevice and DeviceToHost) and the kernel running time. The experimental results are represented in Table I and Figure 3.

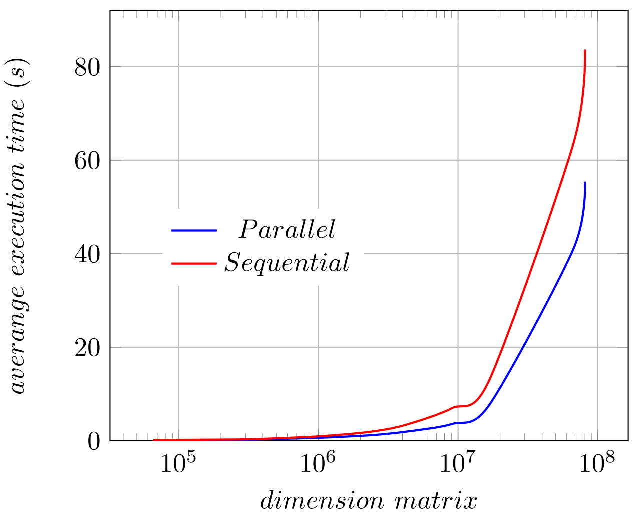

Fig. 3: Comparison between sequential and parallel

Figure 3 shows the execution times, sequential and parallel, where the parallel code is faster. The result of this comparison shows that the parallel code is more efficient. The comparison between parallel and sequential version has been calculated as the difference between the $Sequential\ Execution\ Time$ e il $Parallel\ Execution\ Time$, and has been reported using a histogram in which the parallel code presents a significant performance, while increasing the size of the input image, hence more benefits in the use of a parallel CUDA architecture than the typical CPU.

Figure 4 shows the trend of the difference between the $Sequential\ Execution\ Time$ and the $Parallel\ Execution\ Time$. The trend of this curve is exponential, observing the parallel version it has an $execution\ time$ smaller than the sequential version, with a greater efficiency and better performances, while increasing the amount of data input. The more calculations the more benefits are gained from CUDA parallelism.

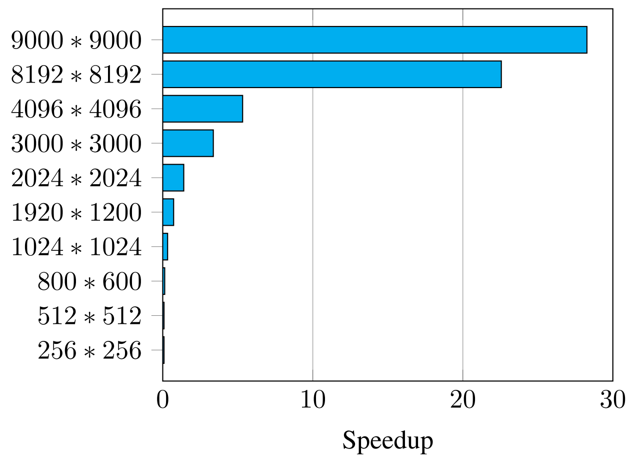

Fig. 4: Speedup

To test the validity of the results, we were carried out other tests based on the execution of iterated code in order to simulate the processing of a set of images of equal size, then measuring the *Execution Time* of the whole process. The results of these tests are shown in Figure 5.

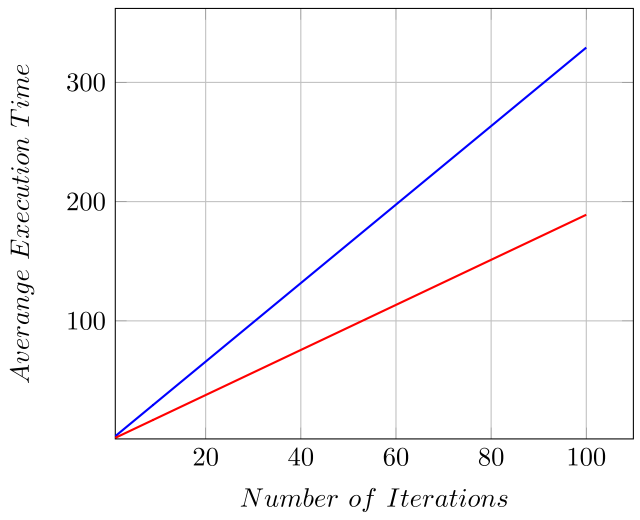

Fig. 5: Number of iterations

The advantages of the use of CUDA architecture are directly proportional to the increase in computation. The same tests were performed using a computing architecture more complex and powerful in order to see if an increase in hardware performance could lead to different results. Such tests were performed by loading the source files on a server equipped with Intel Xenon E5- 2630 with 6 cores 2.6 GHz clock and RAM 48 GB, NVIDIA Tesla K10 with 2x1536 Cuda cores and 8 GB GDDR5 video RAM. The two versions of the algorithm, sequential and parallel, varying the size of the input data array, have led to the results shown in Table II and plotted in Figure 6.

TABLE II: Averange Execution Time on Server

	Averange Execution Time (s)	
Matrix Size	Sequential code	Parallel code
256x256	0.022	1.832
512x512	0.090	1.885
800x600	0.191	1.955
1024x1024	0.037	2.037
1920x1200	0.784	2.093
2048x2048	1.422	2.415
3000x3000	3.047	3.649
4096x4096	5.651	4.829
8192x8192	28.327	19.304
9000x9000	34.557	23.358

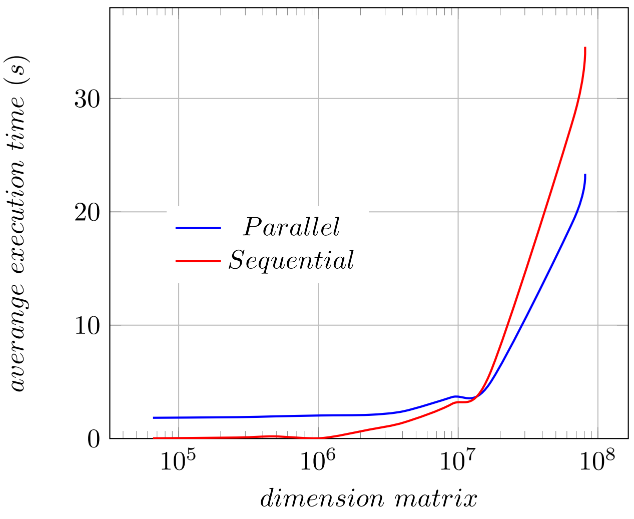

Fig. 6: Comparison between sequential and parallel code on server

The numbers give the result of an average of 10 samples of the running times. The results obtained reveal experimentally that by increasing CPU hardware performance and for images having smaller sizes, the sequential version is faster than its parallel version. This is possibly due to CUDA cores of common GeForce having a low clock speed, determing slower performances for a few calculations. On the other hand, the performance of the parallel code is better than the serial version for the larger processed input array. The tests based on the execution of iterated code produce the results shown in Figure 7.

Tests run on the server estimate the trend when increasing the amount of input data. A better hardware has obtained better performance for both versions of the algorithm, producing however, a difference depending on the size of the input

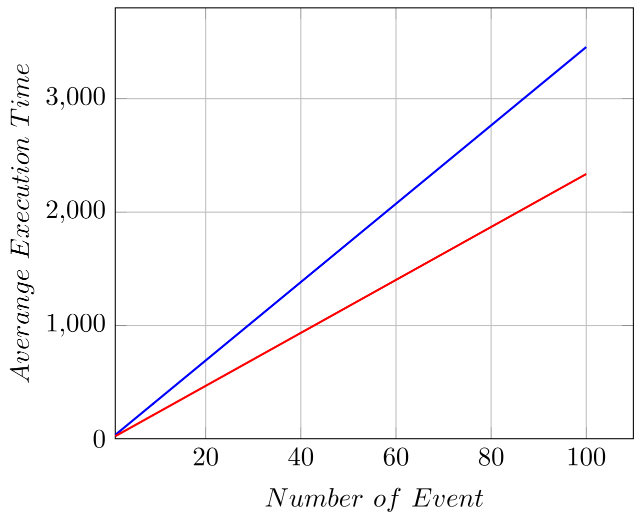

Fig. 7: Number of iterations on Server

data array. In fact, the calculation contained in the algorithm running on the terminal using the GeForce GTX 480 has a better speedup over the sequential version on CPU. As soon as the need grows, it is always better the computational performance with a parallel version. Assuming that the matrices are 9000x9000, the Tesla K10 has a speedup almost 140x with respect to the GeForce GTX 480. So the processing on GPU cards is most often preferable than the computaion on the CPU, the choice of the GPU itself must be weighted on the basis of the knowledge of the computational power needed. The strategy of parallelization brings more efficient algorithms.

VI. Related Works

In recent years the field of GPGPU has aroused great interest inspiring many research works in order to include the best strategies that make it possible to exploit the full potential of the structure of parallel computing and the CUDA parallel programming model. In this regard it is possible to see the work [4] aimed to investigate the GPU architecture and advantages/disadvantages of the CUDA model. Although the core API for programming graphics cards remain, for the moment, OpenGL [1] and DirectX [2], the latter is less prone to general purpose programming. To overcome this lack CUDA has emerged [5], which though limited only to NVIDIA graphics cards makes programming easier for general purpose [5], [6].

A lot of work has been performed for applying the CUDA model for the image processing field. A paper useful to understand how to perform image convolution, essential for image processing, has been produced by Podlozhnyuk [7], and other useful for understanding more complex issues are those produced by Park [8] and Castaño [9].

Finally, the work relating to the performance and application of parallel programming model is inherent in a huge range of topics, such as e.g. the work in [10]–[17]. In [18], a parallel solution for GPU Integrated generation systems (IGSs) has been achieved, whereas in [19], it has been described the realization of a system of distributed and parallel processing for the identification of events.

VII. Conclusion

This paper has presented an algorithm for the application of a linear combination of digital filters implemented with parallel programming for image processing on CUDA compatible GPUs, and has compared with its sequential version on the CPU. The solution presented has the aim to exploit the programming model CUDA parallel, more efficient than a sequential processing code with an appropriate management and organization of blocks and thread.

The tests have shown that the implementation of the parallel code on the GPU increases in speed compared to serial implementation of CPU. In general the speed is not less than 40x with peaks higher than 90x. This increase in speed is also evident with the increase of computational power required in processing. The execution time of the parallel code also counts the time of data transfer between the host device and the penalization of some performance of parallel code which remains higher than the serial version. It is possible to improve the result by a more detailed analysis of the strategies of parallelization in order to manage the computing resources provided by the GPU architectures.

For the realization of the C code we have used CUDA programming interface which has made it easier the GPGPU approach, and allowing us to exploit very effectively the potential hidden in GPU computing. The algorithm presented in this article, though designed for the application of filters on digital images, is well suited for many applications including mobile ones that are outside the scope of image processing such as signal analysis or application of spectral masks sampled signals.

Another possible application for our algorithm is that in the industrial field for the detections of the boundary in an image useful for the detection of objects. In this case you could apply the Sobel operator by simply modifying the matrix of the filter. With this work we have shown only a small part the great potential of the GPU, which could gain popularity in the scientific community thanks to the computing power and to its easy availability, which makes GPGPU one of the best choices in high performance computing.

References

[1] M. Woo, J. Neider, T. Davis, and D. Shreiner, *OpenGL programming guide: the official guide to learning OpenGL, version 1.2*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[2] K. Gray, *Microsoft DirectX 9 programmable graphics pipeline*. Microsoft Press, 2003.

[3] C. Nvidia, “Nvidia cuda c programming guide,” *NVIDIA Corporation*, vol. 120, p. 18, 2011.

[4] J. Ghorpade, J. Parande, M. Kulkarni, and A. Bawaskar, “Gpgpu processing in cuda architecture,” *arXiv preprint arXiv:1202.4347*, 2012.

[5] J. Sanders and E. Kandrot, *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010.

[6] H. Nguyen, *Gpu gems 3*. Addison-Wesley Professional, 2007.

[7] V. Podlozhnyuk, “Image convolution with cuda,” *NVIDIA Corporation white paper, June*, vol. 2097, no. 3, 2007.

[8] I. K. Park, N. Singhal, M. H. Lee, S. Cho, and C. W. Kim, “Design and performance evaluation of image processing algorithms on gpus,” *Parallel and Distributed Systems, IEEE Transactions on*, vol. 22, no. 1, pp. 91–104, 2011.

[9] D. Castaño-Díez, D. Moser, A. Schoenegger, S. Pruggnaller, and A. S. Frangakis, “Performance evaluation of image processing algorithms on the gpu,” *Journal of structural biology*, vol. 164, no. 1, pp. 153–160, 2008.

[10] G. Pappalardo and E. Tramontana, “Automatically discovering design patterns and assessing concern separations for applications,” in *Proceedings of ACM Symposium on Applied Computing (SAC)*, Dijon, France, April 2006, pp. 1591–1596.

[11] R. Giunta, G. Pappalardo, and E. Tramontana, “Aspects and annotations for controlling the roles application classes play for design patterns,” in *Proceedings of IEEE Asia Pacific Software Engineering Conference (APSEC)*, Ho Chi Minh, Vietnam, December 2011, pp. 306–314.

[12] G. Pappalardo and E. Tramontana, “Suggesting extract class refactoring opportunities by measuring strength of method interactions,” in *Proceedings of Asia Pacific Software Engineering Conference (APSEC)*. Bangkok, Thailand: IEEE, December 2013, pp. 105–110.

[13] F. Bonanno, G. Capizzi, G. Lo Sciuto, C. Napoli, G. Pappalardo, and E. Tramontana, “A cascade neural network architecture investigating surface plasmon polaritons propagation for thin metals in openmp,” in *Proceedings of International Conference on Artificial Intelligence and Soft Computing (ICAISC)*, ser. Springer LNCS, vol. 8467, Zakopane, Poland, June 2014, pp. 22–33.

[14] C. Napoli, G. Pappalardo, E. Tramontana, R. Nowicki, J. Starczewski, and M. Woźniak, “Toward work groups classification based on probabilistic neural network approach,” in *Proceedings of International Conference on Artificial Intelligence and Soft Computing (ICAISC)*, ser. Springer LNCS, Zakopane, Poland, June 2015, vol. 9119, pp. 79–89.

[15] C. Napoli, G. Pappalardo, and E. Tramontana, “An agent-driven semantical identifier using radial basis neural networks and reinforcement learning,” in *XV Workshop ”From Objects to Agents” (WOA)*, vol. 1260. Catania, Italy: CEUR-WS, September 2014.

[16] M. Woźniak, D. Połap, M. Gabryel, R. Nowicki, C. Napoli, and E. Tramontana, “Can we process 2d images using artificial bee colony?” in *Proceedings of International Conference on Artificial Intelligence and Soft Computing (ICAISC)*, ser. Springer LNCS, Zakopane, Poland, June 2015, vol. 9119, pp. 660–671.

[17] C. Napoli, G. Pappalardo, and E. Tramontana, “Improving files availability for bittorrent using a diffusion model,” in *Proceedings of IEEE International WETICE Conference*, Parma, Italy, June 2014, pp. 191–196.

[18] F. Bonanno, G. Capizzi, G. L. Sciuto, C. Napoli, G. Pappalardo, and E. Tramontana, “A novel cloud-distributed toolbox for optimal energy dispatch management from renewables in igss by using wrnn predictors and gpu parallel solutions,” in *Power Electronics, Electrical Drives, Automation and Motion (SPEEDAM), 2014 International Symposium on*. IEEE, 2014, pp. 1077–1084.

[19] C. Napoli, G. Pappalardo, E. Tramontana, and G. Zappalà, “A cloud-distributed gpu architecture for pattern identification in segmented detectors big-data surveys,” *The Computer Journal*, p. bxu147, 2014.