GPU Acceleration of Image Processing Algorithm Based on Matlab CUDA

Layla Horrigue [1], Refka Ghodhbane [2], Taoufik Saidani [3] and Mohamed Atri [4]

[1,2,3,]4Electronics and Micro-Electronics Laboratory, Faculty of Sciences, Monastir, Tunisia
[2,3]Faculty of computing and Information Technology, Northern Border University, Rafha, Saudi Arabia

Summary
MATLAB is one of most commonly used platforms in multiple scientific applications like digital image processing, digital signal processing etc…
The high level programming syntax and friendly user of MATLAB makes it best suited to write technical code. Among, the significance of MATLAB's programming is its rich library of built-in function that makes programming more easily. Nerveless, the standard MATLAB uses an interpreter which decelerates the processing, particularly while executing loops. This becomes a bottleneck performance in programs that make excessive use of loops. Therefore, MATLAB is frequently exposed to the memory latency and the issues of slow execution. In order to accelerate MATLAB's processing, we use NIVIDIA'S CUDA parallel processing architecture. We note that processing can be speed up significantly by interfacing MATLAB with CUDA and parallelizing the most time consuming portion of MATLAB's code white balance. The obtained results indicate, that the speedup is proportional to the image size until it attains a maximum at 2056*3088 pixels, beyond these values the speedup decreases. The performance with GPU enhances above a factor of 14~15 compared with CPU.

Key words:
CUDA, MATLAB, GPU, CPU, White Balance.

1. Introduction

MATLAB is a most commonly high-level programming language used in different scientific numerical computation. Among the features supported by MATLAB: including flexibility development environment for managing programming code, using friendly interface for technical computing, interactive tools for iterative exploration, solving problem design and functions for integration MATLAB with other languages and external applications like C, C++, Mex, and CUDA.
However MATLAB exploits high- level programming features because it has a rich library and simple programming paradigm which makes easy to write a technical code for users who do not have much programming expertise. However, MATLAB uses an interpreter that provokes to demean its performance particularly in executing iterative logic. Indeed the huge data (eg image/ matrix operations) causes the memory latency. For those reasons, MATLAB is not much efficient for real time image processing operations. In fact with high volumes of graphics cards deployed work stations or in personal computers, graphics processors units (GPUs) become a parallel computing platform accessible to a wide range of users. In the light of teraflops computing capability and high memory bandwidth, there is an intense incentive to use GPUs for general purpose computation and there have been successful reports in the literature on such effort [9, 2]. NVIDA's compute unified Device Architecture (CUDA) is a revolutionary standard which allows programmers to use the power of computing engine used in NVIDIA's CUDA- enabled GPUs.
However, CUDA uses the parallel computing engine in NVIDIA's GPUs and all the pipeline stages can be combined to execute a number of operations simultaneously. Therefore CUDA can solve the complex computational, taking better than the CPU [5, 6].
This paper presents a demonstration of mixed programming concept by integrating MATLAB with CUDA. This is done by replacing the time consuming parts of the algorithms running in MATLAB CPU and porting to the Graphical Processing Unit (GPU). By applying this methodology, we not only use the rich programming features of MATLAB, but also minimize its performance bottleneck.
This paper is organized as follows: Section 2 and section 3 provide an overview of the architecture of CUDA and its benefits and the related work for accelerating MALAB. Section 4 briefly describes how MATLAB can be integrated with CUDA. In section 5 we present the proposed application. The systematic approach of analyzing MATLAB's slow processing of over proposed application and their CUDA implementation are presented in section 6. The results of the comparative analysis of CPU vs GPU processing are presented in section 7. Finally conclusion is drawn in section 8.

Manuscript received June 5, 2018
Manuscript revised June 20, 2018

2. General purpose programming with GPU

Graphics Processing Units (GPUs) have emerged as powerful accelerators for many regular algorithms that operate on dense vectors and matrices, and it designed to process blocks of pixels at high speed and massive parallel operations on application data like data visualization, graphic rendering etc. However usually these applications include a single program executing in parallel on several elements of data.
On the other hand, GPUs are designed for parallel computing with an emphasis on arithmetic operations, which originate from their main purpose to compute graphic scene which is finally displayed. Current graphic accelerators consist of several multiprocessors (up to 30). Each multiprocessor3 contains several (e.g., 8, 12 or 16) Arithmetic Logic Units (ALUs). Up to 480 processors are in total on the current high-end GPUs. Figure 1 shows the general overview of the CPU and GPU.

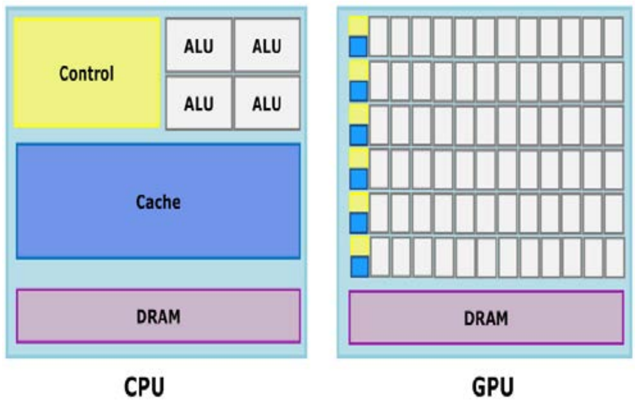


Fig. 1 Comparison CPU and GPU architectures.

2.1 Programming Model:

In 2006, NVIDIA introduced which is hardware and software platform designed for general purpose computing on GPUs, having a novel programming model with new instruction set architecture. CUDA comes with a special software paradigm that allows users to use C as a high level programming language. Other language and applications programming interface are also supported in the futures such as Open ACC, Direct Compute, and FORTRAN.
A CUDA-capable GPU is referred to as a device and the CPU as a host. As shown in fig.1 CUDA computing model provides thread which is the finest grain unit of parallelism. Block is a unit of the resource assignment. The standard size of a thread block is 64-512 threads. It banks on the particular application which is the optimum size of a thread block to certify the best utilization of the device. Thread blocks form a grid can be viewed as a 1-dimensional, 2-dimensional or 3-dimensional array. It means that it guide the developer to partition the problem into sub-problems such a way that can be solved independently in parallel by blocks of threads and every sub-problem into finer part that can be resolved in parallel by all threads within block as shown in figure 2.

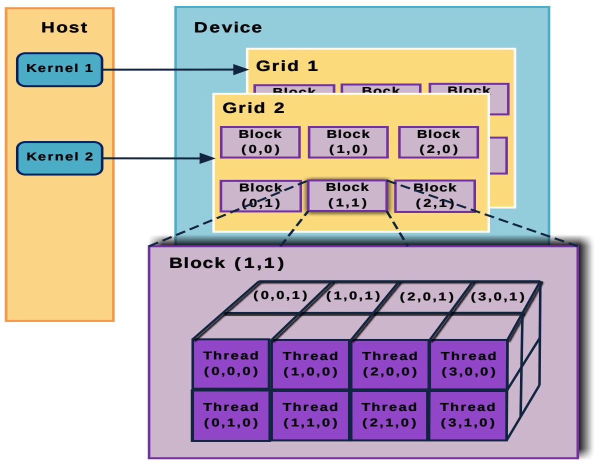


Fig. 2 NVIDIA CUDA programming architecture

CUDA provides different memory spaces which are during execution accessible for threads. As shown in figure 2, the threads grouped into thread blocks can cooperate among themselves by sharing data through a shared memory. Each thread that excites kernel has its own private local memory. The global memory is accessible to all threads, whereas shared memory is visible only to threads of the block. Also threads have access to other memories called texture memory and constant memory as mentioned in figure 2. All these memories have very small access time and their optimized use notably speed up the overall execution program.

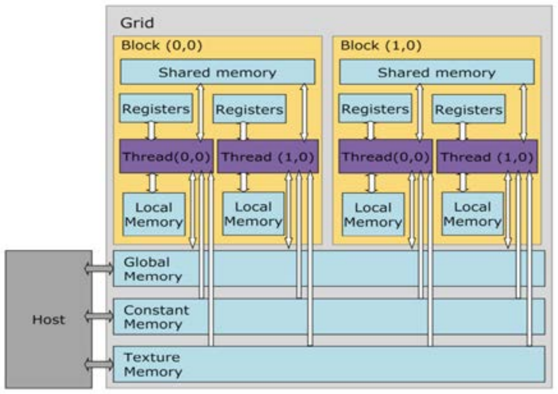


Fig. 3 CUDA memory architecture

• Global memory: The largest volume of memory available to all multiprocessors in a GPU, from 256 MB to 1.5 GB in modern platform. It offers high bandwidth, over 100 GB/s for top solutions from NVIDIA, but it suffers from very high latencies (several hundred cycles).
• Constant storage: memory area of 64 KB (the same concerns modern GPUs), read only for all multiprocessors. It's cached by 8 KB for each multiprocessor. This memory is rather slow latencies of several hundred cycles, if there are no required data in cache.
• Texture memory: is available for reading to all multiprocessors. Data are fetched by texture units in a GPU, so the data can be interpolated linearly without extra overheads. Slow as global memory -- latencies of several hundred cycles, if there are no required data in cache.

2.2 Features and Limitations of CUDA:

It is easy to learn the CUDA API, but hard to program efficient applications which utilize the GPU's performance. CUDA API is a set of extensions based on the standard C language. Counterweight to many features of this massively parallel architecture is that there are limitations mostly caused by HW architecture. CUDA belongs to the class of Single Instruction, Multiple Thread (SIMT) according the Flynn's taxonomy. SIMT originates in Single Instruction Stream, Multiple Data Stream (SIMD) class known for example from the supercomputers based on vector processors (e.g., Cray-1). SIMT also implies the divergence in the program that usually leads to the serialization of the run. Recursive functions are not supported either [14].
As introduced before, graphic accelerators were developed with the focus on computing vast amounts of arithmetic operations. Many of them are implemented directly in the hardware with a cost of units of warp-cycles5. Besides arithmetic functions there is a set of bitwise operations also implemented "in hardware".
Of course, a set of constructs used in parallel programming is present in CUDA. For example several methods of barrier synchronization primitives, native broadcast of a single variable, scatter and gather functions or atomic operations which prevents from race conditions.
The use of shared memory has also significant impact on the overall performance but the limiting factor is its size of 16 KB. Talking about memory, CUDA brought more efficient data transfer operation between the host and the device. Unlike OpenCL6, CUDA is closed source belonging to NVIDIA corp. which can be considered as a limitation as well.

3. Related work

The relevant literature prove that parallel architecture of current GPUs has become progressively powerful and more programmable, which allows GPUs to be the main computation device and invade various domains such as physics and mathematical simulations and even image compression analysis. In deed it explained many solutions of MATLAB's performance bottleneck.
The work presented in [2] discusses the mixed programming principles and methods where MATLAB is integrated with other language like FORTRAN and Visual C++ (VC).
The results of this work show that a mixed programming with different tasks can be achieved by compiling different MATLAB program, doing the necessary settings and replacing the corresponding C++ code. Also the problem of MATLAB's memory latency is discussed on many works.
The work presented in [9] explains who different types of CUDA memories like (global, texture and constant can be used to avoid performance bottleneck and to attain the maximum performance from MATLAB.
Work [10] describes the benefits of integration MATLAB with CUDA and explains who implement an application like 2D.DWT on CUDA, C and Warps. The algorithm is parallelized using CUDA and called through the use of mex function in MATLAB environment. Then the nvmex function compile the CUDA (.cu) file and produces a binary mexw32 file, which can be called like any other MATLAB function in MATLAB.
The work presented in [10] described the potential advantages of using CUDA to accelerate its performance. By employing NVIDIA Tesla C870 the authors achieved speedups varying from 20 to 40.6 in the execution time over a C version on an Intel Core 2 Quad Q6700 (2.66 GHz).
The Results presented in [6] display that the implementation of the application ADL-based wavelet transforms using GPU attains a speedup 10 times greater than that procured by the optimized CPU implementation with an AMD ATHLON II X2 240 CPU and a NVIDIA GeForce 8800 GTX 768MB.
In [18] the authors have implemented the DWT "Le Gall 5/3" and "Cohen-Daubechies-Feauveau 9/7" filters on a low cost NVIDIA's GPU (NVIDIA GeForce GT 640M LE) and realized on MATLAB (version R2013b) using the in-house parallel computation toolbox (PCT). The obtained results show, that the speedup is proportional to the image size and the performance with GPU enhances above a factor of 2~3 compared with CPU.

4. MATLAB and CUDA

MATLAB is very efficient and easy to program development environment, the use of memory latency problem and interpreter has a large negative impact on its performance. The standard MATLAB uses an interpreter which is slower in executing loops compared to the compilers of other languages (C, C++)[12,14].

Image processing tasks require excessive use of loops because it operates on each pixel of an image, for this reasons, in order to get the maximum throughput, we accelerate MATLAB processing by using NVIDIA's CUDA parallel processing architecture. In fact MATLAB could be easily extended via MEX files to take advantage of the computational power offered by the latest NVIDIA graphics processor unit (GPU). MATLAB executable ("MEX") is an essential utility that allow to call and compile the codes written in other languages like (C, FORTRAN), into a dynamically linked.
Mex-files are dynamically linked subroutines written in other languages, which that can be called from within MATLAB as regular MATLAB function.

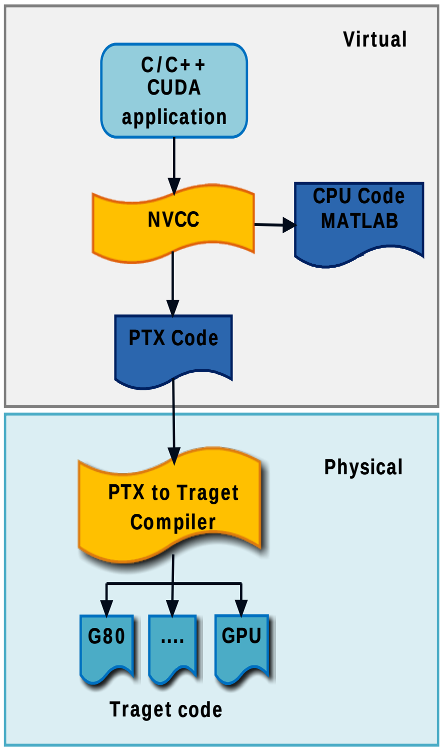


Fig.4 Compiling CUDA code

The external interface function can provide means to call a MATLAB functions written in (C or FORTRAN) and also to transfer data between MATLAB and Mex-files. To integrate MATLAB with CUDA, NVIDIA provides a MATLAB plug-in which allows programmer to write CUDA enabled Mex-files and parallelize different time consuming portions of MATLAB's algorithm with CUDA and call them in MATLAB.

4.1 GPU computing using MATLAB:

Furthermore to perform parallel computing with MATLAB in order to use computer's graphics processing unit, various options are available for using GPU in MATLAB [15, 17and 18]. For example the function "gpuArray ()" is used to transfer data like vector, array etc., from the MATLAB workspace to the GPU, where the computation is effectuated. When the execution is achieved, the results are displaced from the GPU to the MATLAB workspace by the function "gather ()". This allows the GPU data to be available as regular variable in the MATLAB workspace.

4.2 System configuration:

Our proposed implementation is performed on representative commercial products from the GPU markets as mentioned in table 1

Table 1: System configuration

HOST		GPU	
Name	Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz	Name	GeForce GT 620
Clock	3401 MHz	Clock	1620 MHz
Cache	1024 KB	Num. Processors	1
Num. Processors	4	Compute Capability	2.1
OS.Type	Windows	TotalMemory	2.00 GB
OS.Version	Microsoft Windowsÿ7 dition Int,grale	CUDAVersion	6
Name	Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz	DriverVersion	8.17.13.328 8 (332.88)

The GPU is an NVIDIA GeForce GT 620 (384 CUDA cores clocked at 1620 MHz) with 2.0 GB of GPU device memory and a 2.1 kb per-block shared memory per SM) with NVIDIA driver version (332.88)and CUDA 6.0. We also used a multicore CPU based system with an Intel® core™ (i7-3770 CPU @3.40GHz). We have elaborated our GPU code using MATLAB version R2014a, "Transferring data between the MATLAB workspace and the GPU" to accelerate our algorithm using nvcc () function and .ptx file. Table 1 resumes our system configuration while table2 show the peak performance of various GPUs using the same MATLAB version which is R2014a.
The peak performance indicated in table 2 is usually attained when dealing with extremely large arrays. The results are obtained using the CPUs on the host PC and included then for comparison. Since MATLAB works mainly in double precision the devices are classfy according to how well they effect double-precision calculations. Single precision results are included for completeness.

Table 2: GPU comparison report: Host PC

	Results for data-type 'double' (In GFLOPS)			Results for data-type 'single' (In GFLOPS)		
	MTimes	**Backslash**	**FFT**	**MTimes**	**Backslash**	**FFT**
Quadro K6000	1092.86	421.36	160.04	3017.89	831.15	334.22
Host PC	83.89	56.66	10.68	156.74	118.54	18.51
Quadro 2000	38.34	33.77	14.10	223.55	133.56	49.63
GeForce GT 620	12.22	9.63	4.16	55.97	24.29	11.09

5. The Application White Balance

NVIDIA GPUs are recently been a popular device for several scale computations like image processing, signal processing, and other many applications, on account to their high computational throughput and also their parallel architecture. In fact CUDA allows programmers exploit all the power of this architecture by providing specific control over how computations are divided between parallel threads and executed on the device. The resulting algorithms written for the GPU are often much faster than the traditional codes written for the CPU and the process of building a framework for developing can take a long time. In the literature, several programmers write CUDA kernels with the hope that they will be integrated into C or Fortran programs for production. For this reason, they frequently use these languages to test their kernels, which require writing important amounts of “glue code” for tasks like managing GPU memory by transferring data to the GPU, initializing and launching CUDA kernels then visualizing kernel outputs. The writing of “glue code” consumes a lot of time and it is difficult to be modified. In this context, using an image of test “Lena” our article presents how MATLAB supports CUDA kernel development by providing a language and development environment in order to evaluate kernels, analyze and visualize kernel results, and write test harnesses to validate kernel results.

The basic principle of our application is to adjust the colors of an image so that the image does not have a reddish or bluish tint. This technique is called White balancing. In fact this technique involves calculating the average amount of each color present in the image, then applying scale factors to assure that the processed image has an equal amount of each color.

As shown in figure 5, the algorithm eliminates the reddish tint from the original image.

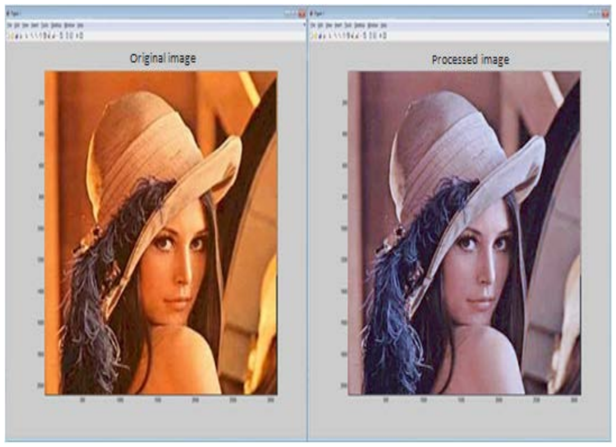
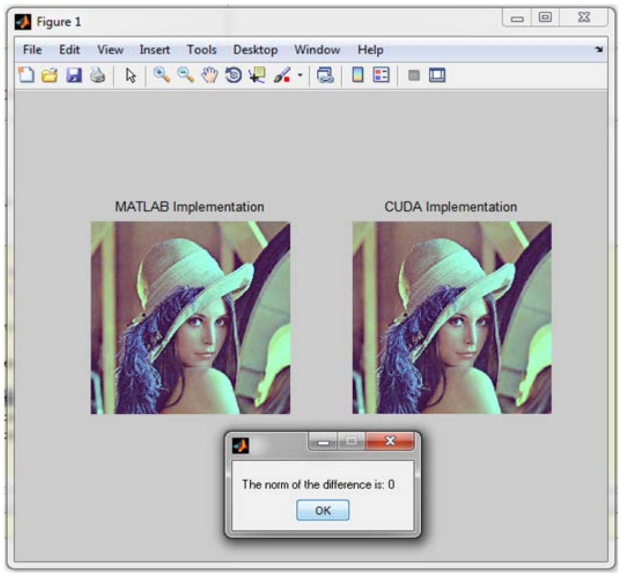


Fig. 5 Image “Lena 512 *512” before and after using white Balance adjustment.

ndeed the development of the algorithm with MATLAB only consumes five lines of code which is much less than that in C. One of the causes is that MATLAB is a high level interpreted language and that it is therefore not necessary to perform administrative tasks such as variable declaration and memory allocation. In addition, MATLAB includes thousands of integrated mathematical, engineering, and tracing functions, and can be extended with domain-specific algorithms in signal and image processing and other fields. The main purpose of our paper is to implement the white balance algorithm in C, with each computational step written as a CUDA kernel. Before starting with CUDA, we utilize the MATLAB white balance code in order to explore the algorithm and decide how to decompose it into kernels. In that event we start by using the MATLAB Profiler to know how long each section of code takes to execute. Consequently the bottleneck areas which means where we will need to spend extra effort to develop efficient CUDA kernels will be indicate by the profiler. Accordingly the most time-consuming section of our algorithm is the code which multiplies every element in the image data with a suitable scale factor. It is obviously an operation that can be parallelized and it could be accelerated notably on the GPU. Therefore we implement our proposed code in CUDA C/C++ and we write kernels for the computational proceeding in the white balance algorithm, then we will evaluate and test this kernel to ensure that it runs correctly and gives right results.

6. Evaluation of the proposed code

In order to load the proposed kernel into MATLAB, we should use paths to the compiled PTX file and source code. Then we must setting the sizes of the thread blocks and grid before we can initiate it, thereafter the kernel can then be used just like any other MATLAB function, save that we launch the kernel utilizing the feval command.



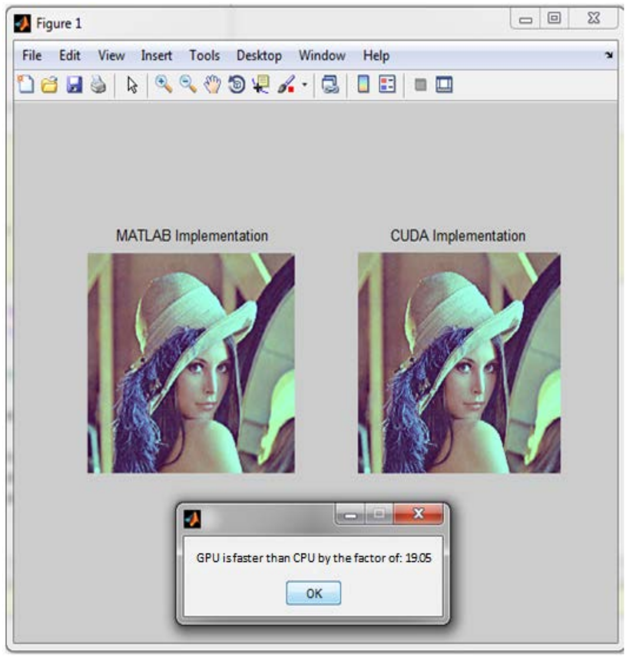
The norm of the difference is: 0

Fig. 6 Comparison between the output images of CUDA and MATLAB (CPU)

To evaluate our approach we test if the results are correct by comparing our new version with the original MATLAB implementation of the white balance algorithm. As indicated in figure 6, the comparison shows that the output images seem identical.
The visual validation proves that the kernel is working properly and the calculated norm of the difference of the output images is zero, which validates our approach numerically.

7. Comparative analysis of GPU vs CPU processing

Furthermore the use of CUDA in order to process an image of size 512 * 512, allows reducing the total time for the image scaling operation from 36.2 ms on the CPU to 7.5 ms on the GPU. The GPU run time distribution is presented as follows: 1.9 ms is execution time and 5.59 ms is for loading and transferring the data to the GPU. That amounts to a 19.05 times speedup on the GPU.



GPU is faster than CPU by the factor of: 19.05

Fig.7 Comparison between the speedup of CUDA (GPU) and MATLB (CPU)

Table 3 summarizes the execution time of four test images (Lena, Barbara, Peppers and Baboon) whit different sizes for the proposed application, White_balance. As indicated in this table, it is clear to note that for images of small size, less than 512 pixels, the computing with the CPU and GPU is almost equal, for all test images.

Table 3: Time cost comparison for the white balance between CPU and GPU for four test images with different sizes.

CARTE GRAPHIQUE: Ge Force GT 620							
Image test	Size of image (unit 8)	CPU time (ms)	GPU time		GPU time (ms)	Speed up of GPU	Speed up of GPU without time of distribution
			Run time Execution (ms)	Run time distribution (ms)			
Lena	256*256	9.6	1.75	5.14	6.9	1.39	5.48
	512*512	36.2	1.9	5.59	7.5	4.82	19.05
	1024*1024	83	1.98	5.82	7.8	10.64	41.92
	1536*1536	124.5	2.77	8.14	10.92	11.4	44.94
	2048*2048	149.4	3.04	8.94	11.99	12.46	49.14
	2056*3088	189	3.24	9.55	12.8	14.76	58.33
	3072*3072	224.22	4.08	12.01	16.1	13.92	54.95
Barbara	256*256	12.1	2.511	7.38	9.9	1.22	4.818
	512*512	36.7	2.13	6.26	8.4	4.36	17.23
	1024*1024	86	2.18	6.41	8.6	10	39.44
	1536*1536	130	3.12	9.19	12.32	10.55	41.66
	2048*2048	155.4	2.96	8.72	11.69	13.29	52.5
	2056*3088	196.2	3.3	9.85	12.8	15.32	59.45
	3072*3072	238.01	4.38	12.91	17.3	13.75	54.33
Peppers	256*256	11.7	2.25	6.64	8.9	1.31	5.2
	512*512	37.5	2.15	6.34	8.5	4.41	17.44
Peppers	1024*1024	88	3.01	8.8	11.9	7.39	29.23
	1536*1536	133.5	2.68	7.9	10.6	12.59	49.81
	2048*2048	161.4	3.31	9.76	13.08	12.33	48.76
	2056*3088	197,1	3.55	10.45	14.1	13.97	55.52
	3072*3072	239.2	5.17	14.38	19.56	12.21	46.26
Baboon	256*256	10.3	1.8	5.29	7.1	1.45	5.72
	512*512	36.4	2.25	6.64	8.9	4.08	16.17
	1024*1024	87.8	2.23	6.56	8.8	9.97	39.37
	1536*1536	130.8	2.43	7.16	9.6	13.62	53.82
	2048*2048	162.4	3.34	9.85	13.19	12.31	48.62
	2056*3088	198.2	3.7	10.89	14.6	13.57	53.56
	3072*3072	242.7	5.29	14.56	19.81	12.25	45.87

As shown in table3, we found that for a Lena image of size 256*256 pixels, the GPU time, which is equal to 6.9 ms, is lower than CPU, which is equal to 9.6. The result in this case is converged, this is due to the time spent to transfer data from the host's memory to GPU's global memory, which requires a large fraction of total execution time for a small picture sizes. If we eliminate the data transfer time in this case the computing with the GPU will be faster than CPU as indicated in the table by the factor of 5,48. By increasing the size of images, it is obvious that the computation with GPU becomes faster than the CPU for all test images. For example, for test image (Baboon) of size 3072*3072 pixels the GPU execution time is faster than CPU by the order of 12.25 and 45.87 when we don't count the transfer data time. In fact the data transfer time often negligible in larger algorithms, because data transfer needs to be completed only once, and we can then compare execution times only.

In addition, the speedup versus different image sizes for four testing images, of white balance application is given in Figures 8 and 9. All test images have similar behavior conditioned by the image size. We note that, for an image size equal to 256*256 pixels, the acceleration factor CPU/GPU is almost equal to 1.5, but for the rest test image sizes, the acceleration factor develops until reached 15,32. The speedup is proportional to the image size until it reaches a maximum at the size 2056*3088 pixels beyond these values the curve decreases to 13.75. Therefore the figure 8 shows that when we increase the image size of all using image test, the speedup increases and attains a maximum at 2056*3088 pixels equal to 59.45 for image Barbara, after that the speedup decreases to 54.33.

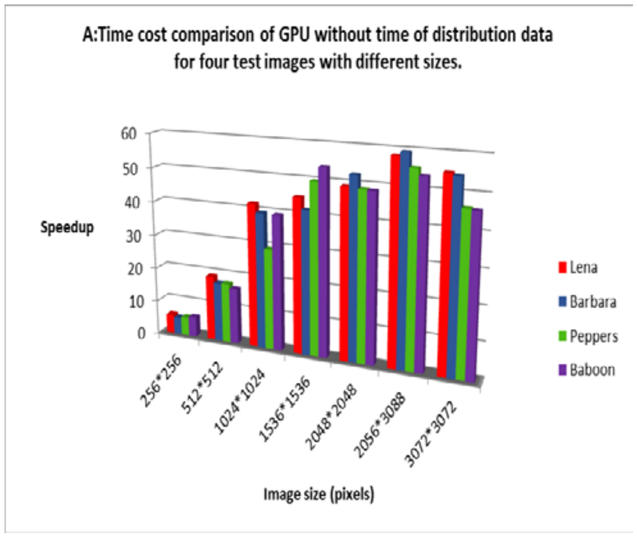


Fig. 8 Speedup factor of GPU for With Balance without of distribution data

For the figure 9, the speedup of the smallest image we tested (Baboon 256*256) was 1.45. By increasing the size of this image, the speedup augments and reaches a maximum at 2056*3088 pixels equal to 13.57. The observed decrease in the acceleration on the other used image test after 2056*3088 was likely due to memory limitations [15, 17and 18].

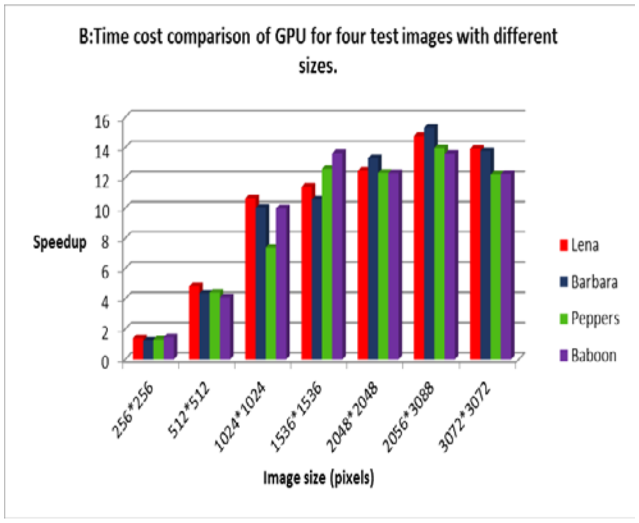


Fig. 8 Speedup factor of GPU for Withebalance

8. Conclusion

This paper presents a demonstration of mixed programming concept by integration MATLAB with CUDA. This is done by replacing the time consuming parts of the algorithms running in MATLAB CPU and porting to the Graphical Processing Unit (GPU). By applying this methodology, we not only use the rich programming features of MATLAB, but also minimize its performance bottleneck. Furthermore we presented in this paper a novel fast method of white balance implementation using MATLAB and CUDA, for achieving load balance between CPUs and GPUs in a dynamic context. It is based on an accurate prediction of the CPU and GPU execution times of codes, using the results of a profiling of those codes. In fact the efficiency of our GPU based implementation is measured and compared to CPU based algorithms using Ge Force GT 120.

Our future plans consist extending this work to handle other types of hardware and larger systems of kernels. Finally, the current system is focused towards performance, but with slight modifications it could be adapted to improve energy and time consumption.

References

[1] M. E. Belviranli, L. N. Bhuyan, and R. Gupta. “A dynamic self-scheduling scheme for heterogeneous multiprocessor architectures”. ACM Trans. Archit.Code Optim., 9(4),57:1-57:20, Jan. 2013.

[2] J.-F. Dollinger and V. Loechner. “Adaptive runtime selection for GPU”. In 42nd International Conference on Parallel Processing - ICPP, Lyon, France, 2013. IEEE.

[3] T. Komoda, S. Miwa, H. Nakamura, and N. Maruyama. “Integrating multi-GPU execution in an OpenACC compiler”. In 42nd International Conference on Parallel Processing - ICPP, Lyon, France, 2013. IEEE.

[4] J. Lee, M. Samadi, Y. Park, and S. Mahlke. “Transparent CPU-GPU collaboration for data-parallel kernels on heterogeneous systems”. In Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques, PACT '13, pages 245{256, Piscataway, NJ, USA, 2013. IEEE Press.

[5] NVIDIA Corporation. Cu BLAS-XT. https://developer.nvidia.com/cublasxt, 2014.

[6] J. Chen, Z. Ju, C. Hua, B. Ma, C. Chen, L. Qin, R. Li, “Accelerated implementation of adaptive directional lifting-based discrete wavelet transform on GPU”, Signal Processing: Image Communication, 28, 1202–1211, 2013.

[7] NVIDIA: NVIDIA's Next Generation CUDA Compute Architecture: Fermi. NVIDIA (2009)

[8] Volkov, V.: “Better Performance at Lower Occupancy. In: GPU Technology Conference 2010. (2010)

[9] T. Kim, H. M. Kim, P. sing Tsai, and T. Acharya, “Rate-distortion optimization algorithm for JPEG 2000”, Mathematics of Data/Image Coding, Compression, and Encryption V, with Applications, 4793 (2003), pp. 36-41.

[10] J. Franco, G. Bernab, J. Fernndez, M. E. Acacio, “A parallel implementation of the 2D wavelet transform using cuda”, 16th Euromicro Conference on Parallel, Distributed and Network-Based Processing, Weimar, Germany, 111-118, 2009.

[11] NVIDIA Corporation, NVIDIA CUDA Programming Guide 3.0, 2010.

[12] J. Matela, “Implementation of JPEG2000 compression on GPU”, Master's thesis, Faculty of Informatics, Masaryk University, 2009.

[13] A. Weiss, M. Heide, S. Papandreou, and N. F□urst, CUJ2K: a “JPEG2000 encoder in CUDA”, (2009).

[14] Masarykova univerzita, Fakulta informatiky, “Design and Implementation of Arithmetic Coder for CUDA Platform”, Thesis, Brno, 2010

[15] D. S. Smith, J. C. Gore, T. E. Yankeelov, E. Brian Welch, “Real-Time Compressive Sensing MRI Reconstruction Using GPU Computing and Split Bregman Methods”, Int. Journal of Biomedical Imaging, Article ID 864827, 1-6, 2012.

[16] J. Aceituno, P. Albert, J. Jegard, J. Virey, « Programmation sur périphérique GPGPU », Université de Bourgogne. 2010. http://barbuk.org/rapports/rapport_m1_sys3.pdf, Accessed 5 Septembre 2015.

[17] Mathworks Parallel Computing Toolbox, User’s Guide, R2014a, 2016.

[18] R. Khemiri et al: “Implementation and Comparison of the Lifting 5/3 and 9/7 Algorithms in MATLAB on GPU”, Journal of Electrical Systems 12-3 (2016):490-499

Layla Horrigue received her M.S. degree in Micro-electronics from Faculty of Science of Monastir, Tunisia in 2013. Her major research interests include VLSI and embedded system in video compression.

Refka Ghodhbani received her M.S. degree in Micro-electronics from Faculty of Science of Monastir, Tunisia in 2013. Her major research interests include Circuit and System Design, Image compression embedded block coding.

Taoufik Saidani received his M.S. degree in Micro-electronics from Faculty of Science of Monastir, Tunisia in 2007. His major research interests include VLSI and embedded system in video and image compression. His current research interests include digital signal processing and hardware–software co-design for rapid prototyping in telecommunications.

Mohamed Atri born in 1971, received his Ph.D. degree in Microelectronics from the Science Faculty of Monastir in 2001. He is currently a member of the Laboratory of Electronics and Microelectronics. His research includes Circuit and System Design, Image processing, Network Communication, IPs and SoCs.