CUDA and OpenCL Implementations of 3D Fast Wavelet Transform

Gregorio Bernabé and Ginés D. Guerrero and Juan Fernández
Computer Engineering Department
University of Murcia (Spain)
e-mail: {gbernabe,gines.guerrero,peinador}@ditec.um.es

***Abstract*— We present in this paper several implementations of the 3D Fast Wavelet Transform (3D-FWT) on CUDA and OpenCL running on a new Fermi Tesla architecture. We evaluate these proposals and make a comparison with others optimal executed on multicores CPU and Nvidia Tesla C870. Speedups of the CUDA version on Fermi architecture are the best results, improving the execution times on CPU, ranging from 5.3x to 7.4x for different image sizes, and up to 81 times faster when communications are neglected. Meanwhile, OpenCL obtains solid gains which range from 2x factors on small frame sizes to 3x factors on larger ones.**

I. Introduction

Efforts to exploit the Graphics Processing Unit (GPU) for non-graphical applications have been underway by using high-level shading languages such as DirectX, OpenGL and Cg. These early efforts that used graphics APIs for General Purpose computing were known as GPGPU programs.

Nvidia was first to launch a solution to exploit the GPU computational power beyond a traditional graphics processor and simplify the programming. CUDA [1] is Nvidia's solution as a simple block-based API for programming. How could it be otherwise, its main competitor AMD introduced its own product called Stream Computing [2].

Both companies have also developed hardware products aimed specifically at the scientific General Purpose GPU (GPGPU) computing market: The Tesla products [3] are from NVIDIA, and Firestream [2] is AMD's product line. Between Stream Computing and CUDA, we chose the latter to program the GPU for being more popular and complete. Moreover, it provides more mechanisms to optimize general-purpose applications.

More recently, Open Computing Language (OpenCL) is a framework [4] that emerges and attempt to unify those two models. It provides parallel computing using task-based and data-based parallelism. It is an open standard. Up to now, it has been adopted by Intel, AMD, Nvidia and ARM. It allows you to program several architectures dependent upon each of the previous manufacturers and hence not specialized for any particular compute device.

Novel scientific applications are good candidates to take the opportunity offered by CUDA and OpenCL for accelerating codes on GPUs, the release of the Tesla GPU based on Fermi architecture offers a new stage on the development of GPGPU, and the 3D Fast Wavelet Transform (3D-FWT) represents a solid opportunity in the video processing field.

In previous works [5][6], we contributed with a CUDA implementation for the 2D-FWT running more than 20 times faster than a sequential C version on a CPU, and more than twice faster than optimized OpenMP and Pthreads versions implemented on multicore CPUs. We extend our analysis to the 3D-FWT scenario, different alternatives and programming techniques have been introduced for an efficient parallelization of the 3D Fast Wavelet Transform on multicore CPUs and manycore GPUs. OpenMP and Pthreads were used on the CPU to expose task parallelism, CUDA was selected for exploiting data parallelism on the Tesla C870 with an explicit memory handling, where GPU speed-up extends between 3x and 15x depending on problem size.

In this paper, we present several implementations of the 3D-FWT on CUDA and OpenCL. A comparison between both them and our previous results is carried out.

The rest of the paper is organized as follows. Section II presents the foundations of the 3D-FWT. Section III focuses on the specifics of the GPU programming with CUDA and outlines the GPU implementation. Section IV describes peculiarities of the GPU implementation on OpenCL. Section V analyzes performance and Section VI concludes.

II. The Wavelet Transform Foundations

The wavelet transform can be implemented by quadrature mirror filters (QMF), $G = g(n)$ and $H = h(n)\ n\epsilon Z$. H corresponds to a low-pass filter, and G is a high-pass filter. For a more detailed analysis of the relationship between wavelets and QMF see [7].

The filters H and G correspond to one step in the wavelet decomposition. Given a discrete signal, s, with a length of 2^n, at each stage of the wavelet transformation the G and H filters are applied to the signal, and the filter output downsampled by two, thus generating two bands, G and H. The process is then repeated on the H band to generate the next level of decomposition, and so on. This procedure is referred to as the 1D Fast Wavelet Transform (1D-FWT).

It is not difficult to generalize the one-dimensional wavelet transform to the multi-dimensional case [7]. The wavelet representation of an image, $f(x, y)$, can be obtained with a pyramid algorithm. It can be achieved by first applying the 1D-FWT to each row of the image and then to each column, that is, the G and H filters are applied to the image in both the horizontal and vertical directions. The process is repeated

several times, as in the one-dimensional case. This procedure is referred to as the 2D Fast Wavelet Transform (2D-FWT).

As in 2D, we can generalize the one-dimensional wavelet transform for the three-dimensional case. Instead of one image, there is now a sequence of images. Thus, a new dimension has emerged, the time (t). The 3D-FWT can be computed by successively applying the 1D wavelet transform to the value of the pixels in each dimension.

Based on previous work [8], we consider Daubechie's W_4 mother wavelet [9] as an appropriate baseline function. This selection determines the access pattern to memory for the entire 3D-FWT process. Let us assume an input video sequence consisting of a number of frames (3^{rd} dimension), each composed of a certain number of rows and columns (1^{st} and 2^{nd} dimension). The 1D-FWT is performed across all frames for each row and column, that is, we apply the 1D-FWT $rows \times cols$ times in the third dimension. The first 1D-FWT instance requires four elements to calculate the first output element for the reference video and the detailed video, with these elements being the first pixel belonging to the first four frames. The second output element for the reference and detailed video are calculated using the first pixel of the third, fourth, fifth and sixth video frames. We continue this way until the entire reference and detailed video are calculated, and these data are the input used for the next stage.

The 2D-FWT is performed $frames$ times, i.e., once per frame. This transform is performed by first applying the 1D-FWT on each row (*horizontal filtering*) of the image, followed by the 1D-FWT on each column (*vertical filtering*). The fact that *vertical filtering* computes each column entirely before advancing to the next column, forces the cache lines belonging to the first rows to be replaced before the algorithm moves on to the next column. Meerwald et al. [10] propose two techniques to overcome this problem: row extension and aggregation or tiling.

Other studies [11][12], have also reported remarkable improvements when applying the *tiling* technique over the 2D-FWT algorithm. Our experience implementing on a CPU the sequential 2D-FWT algorithm revealed a reduction of almost an order of magnitude in the overall execution time with respect to a baseline version. This process can straightforwardly be applied to the 3D case. In our previous work [6], we report solid gains on execution times as well, which range from 2-3x factors on small frame sizes to 5-7x factors on larger ones. From now on, only the tiled 3D-FWT version is taken for parallelization purposes, either on CPU or GPU.

III. Compute Unified Device Architecture

The Compute Unified Device Architecture (CUDA) [1] is a programming interface and set of supported hardware to enable general purpose computation on Nvidia GPUs. The programming interface is ANSI C extended by several keywords and constructs which derive into a set of C language library functions as a specific compiler generates the executable code for the GPU. Since CUDA is particularly designed for generic computing, it can leverage special hardware features not visible to more traditional graphics-based GPU programming, such as small cache memories, explicit massive parallelism and lightweight context switch between threads.

All the latest Nvidia developments on graphics hardware are compliant with CUDA: For low-end users and gamers, we have the GeForce series; for high-end users and professionals, the Quadro series; for general-purpose computing, the Tesla boards.

Focusing on Tesla, the C870, D870 and S870 models are respectively endowed with one, two and four computing nodes using a 1U rack-mount chassis. They are all based on the G80 GPU, upgraded with the GT200 GPU to release the Tesla C1060 and S1070 models. Our base architecture [6], the Tesla C870, contains 128 cores and 1.5 GB of video memory to deliver a peak performance of 518 GFLOPS (single precision), a peak on-board memory bandwidth of 76.8 GB/s and a peak main memory bandwidth of 4 GB/s under its PCIe x16 interface.

The Fermi architecture is the most significant leap forward in GPU architecture since the original G80. Fermi implements IEEE 754-2008 and significantly increased double-precision performance. It added error-correcting code (ECC) memory protection for large-scale GPU computing, 64-bit unified addressing, cached memory hierarchy, and instructions for C, C++, Fortran, OpenCL, DirectCompute and other languages.

The Tesla C2050 contains 448 cores and 3 GB of video memory to deliver a peak performance of 1.03 TFLOPS (simple precision) and 515 GFLOPS (double precision), a peak on-board memory bandwidth of 144 GB/s and a peak main memory bandwidth of 8 GB/s under its PCIe x16 interface of second generation.

The G80 and the Fermi parallel architectures are a SIMD (Single Instruction Multiple Data) processors. In C870 and C2050, cores are organized into 16 and 14 multiprocessors, each having a large set of 8192 and 32768 registers, respectively. The first generation of Tesla GPU has a 16 KB shared memory very close to registers in speed (both 32 bits wide), and constant and texture caches of a few kilobytes. On the Fermi Tesla, the shared memory can be configured from 16KB to 48 KB. In both architectures, each multiprocessor can run a variable number of threads, and the local resources are divided among them. In any given cycle, each core in a multiprocessor executes the same instruction on different data based on its `threadId`, and communication between multiprocessors is performed through global memory.

At the highest level, a program is decomposed into kernels mapped to the hardware by a grid composed of blocks of threads scheduled in warps. No inter-block communication or specific schedule-ordering mechanism for blocks or threads is provided, which guarantees each thread block to run on any multiprocessor, even from different devices, at any time. Threads belonging to the same block must all share the registers and the shared memory on a given multiprocessor. This tradeoff between parallelism and thread resources must be wisely solved by the programmer to maximize execution efficiency on a certain architecture given its limitations. These

TABLE I

MAJOR HARDWARE AND SOFTWARE LIMITATIONS WITH CUDA. CONSTRAINTS ARE LISTED FOR THE G80 AND FERMI GPUS.

Hardware feature	C870	C2050
Multiprocessors (MP)	16	14
Processors / MP	8	32
32-bit registers / MP	8192	32768
Shared Memory / MP	16 KB	16 KB/48 KB
L1 Cache / MP	No	48 KB/16 KB
L2 Cache	No	Yes – 768 KB
Software limitation	C870	C2050
Threads / Warp	32	32
Thread Blocks / MP	8	8
Threads / Block	512	1024
Threads / MP	768	1536

limitations are listed in Table I for the cases of the Tesla C870 and C2050.

A. Implementation of 3D-FWT on CUDA

Our 3D-FWT implementation in CUDA consists of the following three major steps:

1) The *host* (CPU) allocates in memory the first four video frames coming from a .pgm file.
2) The first four images are transferred from main memory into video memory. The 1D-FWT is then applied to the first four frames over the third dimension to obtain a couple of frames for the detailed and reference videos. The grid is composed of $rows \times cols/128$ blocks.
3) The 2D-FWT is applied to the frame belonging to the detailed video, and subsequently, to the reference video. Results are then transferred back to main memory.

The whole procedure is repeated for all remaining input frames, taking two additional frames on each new iteration. On each new iteration, two frames are copied, either at the beginning or at the second half depending on the iteration number. In particular, the first iteration copies frames number 0, 1, 2 and 3 to obtain the first detailed and reference video frames, the second iteration involves frames 2, 3, 4 and 5 to obtain the second detailed and reference video frames, and so on. Note that frames 4 and 5 occupy the memory formerly assigned to frames 0 and 1, which requires an interleaved access to frames in the second iteration. Conflicts on shared memory banks and coalescing on global memory accesses has been solved.

IV. OPEN COMPUTING LANGUAGE IMPLEMENTATION OF 3D-FWT

Open Computing Language (OpenCL) is an open royalty-free standard for general purpose parallel programming across CPUs, GPUs and other processors, giving software developers portable and efficient access to the power of these heterogeneous processing platforms.

OpenCL includes a host C API for for controlling and interacting with GPU devices, a C language for writing device kernels and an abstract device model that maps very well to NVidia and ATI hardware. There are some differences between CUDA and OpenCL in terminology, as we can observe in table II. Therefore, we use simple source to source translation

TABLE II

DIFFERENCES IN TERMINOLOGY BETWEEN CUDA AND OPENCL

CUDA Terminology	OpenCL Terminology
GPU	Device
Multiprocessor	Compute Unit
Scalar core	Processing element
Global memory	Global memory
Shared (per-block) memory	Local memory
Local memory (automatic, or local)	Private memory
kernel	program
block	work-group
thread	work item

TABLE III

SUMMARY OF EXECUTION TIMES (MSECS.) FOR THE 3D-FWT ON EACH PLATFORM, WITH THE GPU GAINS BETWEEN PARENTHESIS.

	Frame size					
Code version	512x512		1Kx1K		2Kx2K	
CPU optimal	156.09		655.33		2843.43	
CUDA C870	57.65	(2.7x)	216.66	(3.0x)	843.11	(3.4x)
CUDA C2050	29.21	(5.3x)	100.61	(6.5x)	381.58	(7.4x)
OpenCL C2050	87.12	(1.8x)	276.39	(2.4x)	1011.47	(2.8x)

to convert the kernels of the implementation of 3D-FWT on CUDA to OpenCL.

Setting up the GPU for kernel execution differs substantially between CUDA and OpenCL. Their APIs for context creation and data copying are different, and different conventions are followed for mapping the kernel onto the GPUs processing elements. These differences could affect the length of time needed to code and debug a GPU application, but here we mainly focus on runtime performance differences.

V. PERFORMANCE ANALYSIS

Table III summarizes the optimal execution times we have obtained on each hardware platform at the end of our parallelization effort when the 3D-FWT is applied to a video of 64 frames of different sizes. Input data were recovered from files in PGM format, where a single component (grayscale) was used. I/O time to read grayscale images from file was not considered. A similar programming effort and hardware cost was invested on each platform.

We have included our optimal tiled 3D-FWT implementation designed with OpenMP and Pthreads [6]. This version is executed on an Intel Core 2 Quad Q6700 CPU (see Table III, upper side). Also, the best parallelization strategy on a GPU using CUDA has been used to obtain the results in C870 and C2050. The original CUDA version implemented has been adjusted to the Fermi architecture with the memory optimizations needed. The size of the shared memory has been configured to 16 KB and 48 KB without influence in the results. The OpenCL implementation has been obtained from the CUDA version, translating line to line and following the same model (last row of Table III).

First of all, results with Fermi Tesla clearly improve their obtained with our original C870 as might be expected. As the size of images increase, the difference between the original Tesla and the last one is bigger. The average gap between them is about 2x for the different image sizes, which confirms the potential of the Fermi architecture. The increase in the number of processors and cached memory hierarchy introduced are

TABLE IV

OPENCL AND CUDA EXECUTION TIMES (IN MSECS.) FOR OUR OPTIMAL TILED 3D-FWT IMPLEMENTATION ON AN INPUT VIDEO CONTAINING 64 FRAMES OF INCREASING SIZES. THE COMMUNICATION COST IS REMOVED IN THE LAST ROW.

	Frame size		
3D-FWT stage – OpenCL C2050	512x512	1Kx1K	2Kx2K
1. CPU to GPU transfer	25.19	86.38	325.52
2. 1D-FWT on frames	3.53	6.64	11.73
3. 1D-FWT on rows	3.85	5.89	6.97
4. 1D-FWT on cols	3.80	9.82	29.29
5. GPU to CPU transfer	50.75	167.66	637.96
Computational time (2-4)	**11.18**	**22.35**	**47.99**
GPU/CPU speed-up	14.0x	29.3x	59.3x
	Frame size		
3D-FWT stage – CUDA C2050	512x512	1Kx1K	2Kx2K
1. CPU to GPU transfer	11.62	45.6	181.63
2. 1D-FWT on frames	2.11	4.18	7.73
3. 1D-FWT on rows	2.37	2.39	2.39
4. 1D-FWT on cols	2.29	6.86	25.15
5. GPU to CPU transfer	10.82	41.58	164.68
Computational time (2-4)	**6.77**	**13.43**	**35.27**
GPU/CPU speed-up	23.1x	48.8x	80.6x

responsible for most of the achieved improvement. The programming effort to obtain the results in the C2050 has been minimal, but it was hoped a greater improvement because there are two generations of GPUs between both Tesla architectures and the number of processors is 3.5 times in the Fermi GPU than in the C870.

The OpenCL implementation obtains better results than the optimal CPU. Speedups are considerable and present a good scalability. The GPU speed-up factor extends into 2.8x factor in the most favorable case. However, these outcomes are very far from those collected through GPUs Tesla with CUDA. In fact, results are below our initial speedups with CUDA on the Tesla C870. This is due to the semantic gap between OpenCL and compute devices because it is vendor independent and hence not specialized for any particular compute device.

A. GPU profiling

For both optimal GPU versions with OpenCL and CUDA, we may split its execution time into constituent steps for completing a quick profiling process. Table IV reveals this breakdown, where we can see that each 1D-FWT phase is lower in CUDA option than in the OpenCL implementation. This is because of the additional layer introduced by OpenCL. The major difference extends into 1.7x factor for the computational time revealing an important and substantial discrepancy in favor of CUDA. If we eliminate the communication time in each configuration, accelerations obtained with CUDA are very considerable and important. Likewise, speedups obtained by OpenCL are highly competitive.

With regard to the communication time, this one predominates clearly over calculations in both implementations. This is a consequence of the nature of a 3D-FWT algorithm, which lacks of arithmetic intensity but handles big data volumes. Now, the gap between CUDA and OpenCL is very important and speedups go up 4.7x favorable to CUDA. Thus, it is still unclear that OpenCL can achieve the same performance as other programming frameworks that are designed for particular compute devices.

VI. SUMMARY AND CONCLUSIONS

In this work, we have presented and evaluated several methods to implement the 3D Fast Wavelet Transform on CUDA and OpenCL on a new Fermi architecture. We have compared these implementations with others optimal executed on multicores CPU and Tesla C870 GPU. The implementation on CUDA achieves better speedups, ranging from 5.3x to 7.4x for different image sizes. If we discard the cost of communications between CPU and GPU, profits rise to a factor of 80.6x for larger image. OpenCL presents gains up to 2.8x with regard the best implementation on CPU. However, these outcomes are even lower than those obtained with Tesla C870. OpenCL is hardly competitive with CUDA in terms of performance because the first one has and environment setup overhead that is large and should be minimize. Moreover, the difference in the communication time between CUDA and OpenCL is very significant, because the last one has been designed for general compute devices.

ACKNOWLEDGMENTS

This work was supported by the Spanish MICINN, Consolider Programme and Plan E funds, as well as European Commission FEDER funds, under Grants CSD2006-00046 and TIN2009-14475-C04-02/01. It was also partly supported by the Fundación Séneca (Agencia Regional de Ciencia y Tecnología, Región de Murcia) under grant 00001/CS/2007.

REFERENCES

[1] Nvidia, "CUDA Zone maintained by Nvidia," http://www.nvidia.com/object/cuda.html, 2009.

[2] AMD, "AMD Stream Computing," http://ati.amd.com/technology/streamcomputing/index.html, 2009.

[3] Nvidia, "Tesla GPU Computing Solutions," http://www.nvidia.com/object/tesla_computing_solutions.html, 2009.

[4] The Khronos Group, "The OpenCL Core API Specification," http://www.khronos.org/registry/cl.

[5] J. Franco, G. Bernabé, J. Fernández, and M. Ujaldón, "The 2D Wavelet Transform on Emerging Architectures: GPUs and Multicores," Still to be published in *Journal of Real-Time Image Processing. http://dx.doi.org/10.1007/s11554-011-0224-7. Special Issue*, September 2011.

[6] J. Franco, G. Bernabé, J. Fernández, and M. Ujaldn, "Parallel 3D fast wavelet transform on manycore GPUs and multicore CPUs," in *10^{th} International Conference on Computational Science (ICCS 2010). http://dx.doi.org/10.1016/j.procs.2010.04.122. Procedia Computer Science*, vol. 1, Amsterdam, Holland, June 2010, pp. 1101–1110.

[7] S. Mallat, "A Theory for Multiresolution Signal Descomposition: The Wavelet Representation," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 11, no. 7, pp. 674–693, July 1989.

[8] G. Bernabé, J. González, J. M. García, and J. Duato, "A New Lossy 3-D Wavelet Transform for High-Quality Compression of Medical Video," in *Proceedings of IEEE EMBS International Conference on Information Technology Applications in Biomedicine*, November 2000, pp. 226–231.

[9] I. Daubechies, *Ten Lectures on Wavelets*. Society for Industrial and Applied Mathematics, 1992.

[10] P. Meerwald, R. Norcen, and A. Uhl, "Cache Issues with JPEG2000 Wavelet Lifting," in *Proceedings of Visual Communications and Image Processing Conference*, January 2002, pp. 626–634.

[11] J. Tao, A. Shahbahrami, B. Juurlink, R. Buchty, W. Karl, and S. Vassiliadis, "Optimizing Cache Performance of the Discrete Wavelet Transform Using a Visualization Tool," *Procs. of IEEE Intl. Symposium on Multimedia*, pp. 153–160, December 2007.

[12] A. Shahbahrami, B. Juurlink, and S. Vassiliadis, "Improving the Memory Behavior of Vertical Filtering in the Discrete Wavelet Transform," in *Proceedings of ACM Conference in Computing Frontiers*, September 2006, pp. 253–260.