Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 1 (2010) 1101–1110

Procedia Computer Science

www.elsevier.com/locate/procedia

International Conference on Computational Science, ICCS 2010

Parallel 3D fast wavelet transform on manycore GPUs and multicore CPUs

Joaquín Franco[a,1], Gregorio Bernabé[a], Juan Fernández[a], Manuel Ujaldón[b]

[a]*Computer Engineering Department, University of Murcia (Spain)*
[b]*Computer Architecture Department, University of Malaga (Spain)*

Abstract

GPUs have recently attracted our attention as accelerators on a wide variety of algorithms, including assorted examples within the image analysis field. Among them, wavelets are gaining popularity as solid tools for data mining and video compression, though this comes at the expense of a high computational cost. After proving the effectiveness of the GPU for accelerating the 2D Fast Wavelet Transform [1], we present in this paper a novel implementation on manycore GPUs and multicore CPUs for a high performance computation of the 3D Fast Wavelet Transform (3D-FWT). This algorithm poses a challenging access pattern on matrix operators demanding high sustainable bandwidth, as well as mathematical functions with remarkable arithmetic intensity on ALUs and FPUs. On the GPU side, we focus on CUDA programming to develop methods for an efficient mapping on manycores and to fully exploit the memory hierarchy, whose management is explicit by the programmer. On multicore CPUs, OpenMP and Pthreads are used as counterparts to maximize parallelism, and renowned techniques like *tiling* and *blocking* are exploited to optimize the use of memory. Experimental results on an Nvidia Tesla C870 GPU and an Intel Core 2 Quad Q6700 CPU indicate that our implementation runs three times faster on the Tesla and up to fifteen times faster when communications are neglected, which enables the GPU for processing real-time videos in many applications where the 3D-FWT is involved. © 2010 Published by Elsevier Ltd.

Keywords:
3D Fast Wavelet Transform, parallel programming, GPU, multicore

1. Introduction

The newest versions of programmable GPUs provide a compelling alternative to traditional CPUs, delivering extremely high floating point performance for scientific applications which fit their architectural idiosyncrasies [2]. Both Nvidia and AMD have released software components aimed to exploit the GPU computational power beyond a traditional graphics processor. CUDA [3] is Nvidia's solution as a simple block-based API for programming; AMD's solution is called Stream Computing [4]. Those companies have also developed hardware products aimed specifically

Email addresses: `j.franco@ditec.um.es` (Joaquín Franco), `gbernabe@ditec.um.es` (Gregorio Bernabé), `juanf@ditec.um.es` (Juan Fernández), `ujaldon@uma.es` (Manuel Ujaldón)

[1]Corresponding author

at the scientific General Purpose GPU (GPGPU) computing market: The Tesla products [5] are from NVIDIA, and Firestream [4] is AMD's product line. Between Stream Computing and CUDA, we chose the latter to program the GPU for being more popular and complete. More recently, the OpenCL framework [6] emerges as an attempt to unify those two models with a superset of features, for which we are confident on an eventual portability for the methods described throughout this paper without loss of generality.

On its evolution towards high-performance general-purpose computing, image processing has proven to be of great success for GPUs, and within this field, the Fast Wavelet Transform (FWT) constitutes an extraordinary opportunity for a GPU acceleration for two primary reasons: Its computational cost, and more important, the leading role it is assuming in many applied areas like biomedical analysis, video compression and data mining in general.

The FWT is a memory intensive application containing assorted access patterns where memory optimizations constitute a major challenge. Fortunately, CUDA provides a set of powerful low-level mechanisms for controlling the use of memory and its hierarchy. This affects performance at the expense of a programming effort, which is the main focus of this paper.

There have already been several implementations of the 2D-FWT on a GPU. In [7], a SIMD algorithm runs the 2D-DWT on a GeForce 7800 GTX using Cg and OpenGL, with a remarkable speed-up. A similar effort in [8] combined Cg and the 7800 GTX to report a 1.2x-3.4x speed-up versus a CPU counterpart. In a previous work [1], 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. This work extends our analysis to the 3D scenario, where speed-up factors have been improved using a new set of optimization techniques.

The rest of the paper is organized as follows. Section 2 presents the foundations of the 3D-FWT. Section 3 describes our implementation effort on multicore CPUs. Section 4 focuses on the specifics of the GPU programming with CUDA, and Section 5 outlines the GPU implementation and optimizations carried out. Section 6 analyzes performance and Section 7 concludes.

2. The Wavelet Transform

The basic idea of the wavelet transform is to represent any arbitrary function f as a weighted sum of functions, referred to as wavelets. In multiresolution analysis, two functions exist: the mother wavelet and its associated scaling function. Therefore, the wavelet transform can be implemented by quadrature mirror filters (QMF), $G = g(n)$ and $H = h(n)\ n\epsilon Z$. H and G correspond to low and high pass filters, respectively, calculated at each step of the wavelet decomposition process. For a more detailed analysis of the relationship between wavelets and QMF see [9].

Given a discrete signal s, with a length of 2^n, each of those output filters are down-sampled by two. The process is then repeated on the H band to generate the next level of decomposition. This procedure is referred to as the 1D Fast Wavelet Transform (1D-FWT).

It is not difficult to generalize the 1D-FWT to a multi-dimensional case [9]. The 2D-FWT of an image computes the 1D transform over image rows and columns separately, and then down-samples to start the next decomposition level. The 3D case of a video sequence introduces the temporal dimension to represent an array of 2D images (the video frames). Here, the 3D-FWT is computed by applying the 1D-FWT across all frames for each row and column to obtain two volumes: The reference video, or low-pass filter, and the detailed video, or high-pass filter. Then, the 2D-FWT is applied on both volumes, once per frame, to finalize the process.

2.1. Memory footprint

Based on previous work [10], we consider Daubechie's W_4 mother wavelet [11] 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).

First, 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

Table 1: Execution times in milliseconds on an Intel Core 2 Quad Q6700 CPU when running a 3D-FWT for an input video containing 64 frames. A baseline 3D-FWT is compared against assorted versions using different C compilers, command-line flags and parallelization strategies.

		Optimizations		Frame size		
Programming language and/or tool	Parallel	Tiles	Blocks	512 x 512	1K x 1K	2K x 2K
C and g++ compiler	No	No	No	990.76	5245.62	34798.70
C and g++ (-O3 flags)	No	Yes	No	392.77 (2.5x)	1592.72 (3.3x)	6404.40 (5.4x)
C and icc (-O3 flags)	No	Yes	No	309.46 (3.2x)	1272.16 (4.1x)	5023.09 (6.9x)
C using icc optimal (*)	No	Yes	No	305.37 (3.2x)	1231.34 (4.2x)	4773.52 (7.3x)
(*) Optimal flags for the icc compiler at command-line are: {-03, -parallel, -par-threadshold0, -xT}						
OpenMP (4 threads)	Yes	Yes	No	186.47	762.59	3142.29
” + Pthreads	Yes	Yes	No	176.91	715.85	2889.97
OpenMP (4 threads)	Yes	Yes	Yes	166.79	687.17	2831.32
” + Pthreads	Yes	Yes	Yes	156.09	655.33	2843.43

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.

Second, the 2D-FWT is performed $frames$ times, i.e., once per frame. This is translated into $rows$ instances of the 1D-FWT to sweep over the first dimension, followed by $cols$ instances of the 1D-FWT to complete the second dimension.

2.2. *Improving data locality*

For the sake of simplicity, let us focus now on the 2D-FWT process. 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. [12] propose two techniques to overcome this problem: *row extension* and *aggregation*.

Row extension adds some dummy elements so that the image width is no longer a power of two, but co-prime with the number of cache sets. This technique makes sense when we use large images with a width equal to a power of two, and filter length is greater than four on a four-way associative cache. But in our specific case, we use a eight-way associative cache.

Aggregation filters a number of adjacent columns consecutively before moving on to the next row, instead of performing *vertical filtering* on a column by column basis. When the number of columns filtered consecutively matches the image width, *aggregation* is called *tiling*.

Other studies [13, 14] 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. The upper side of Table 1 reports solid gains on execution times as well, which range from 2-3x factors on small frame sizes to 5-7x factors on larger ones. Selected compilers are ICC (Intel C Compiler) [15], a corporate tool, and GCC (GNU Compiler Collection) [16], a free compiler developed by the GNU project. 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. From now on, only the tiled 3D-FWT version is taken for parallelization purposes, either on CPU or GPU.

3. Parallelization on a multicore CPU

For an efficient 3D-FWT parallelization on multicore CPUs, three different paths are explored: (1) Automatic parallelization driven by compilers, (2) semi-automatic parallelization using OpenMP, and (3) explicit thread-level parallelism with `pthreads`.

Our first attempt uses the following flags in the C compiler (besides `-O3`): `-parallel` generates multi-threaded code for loops; `-par-threadshold0` sets a threshold for automatic parallelization of loops based on the probability of a profitable parallel execution; finally, `-xT` generates specialized code and enables vectorization. Execution times (see central rows in Table 1) report modest compiler gains, which encourages us to get involved in the parallelization process.

OpenMP [17] is an API for multi-platform shared-memory parallel programming in C/C++ and Fortran. An OpenMP parallel version for the 3D-FWT with *tiling* requires a moderate programming effort, and can fully exploit as many cores as the CPU may have. In our case, we limit the study to a quad-core platform, focusing on scalability and raw performance. Minimal changes were required with respect to the sequential code for the 3D-FWT due to the high-level expression of OpenMP. In particular, a single directive **#pragma omp parallel for** was applied to define a parallel region on the main `for` loop sweeping over frames. Execution times for this 3D-FWT version are shown in the lower side of Table 1. Performance was studied depending on the number of running threads, with four threads to provide the best results versus counterparts based on one, two and eight threads. This parallel version reduces the execution time around 40% with respect to the previous optimization effort using the sequential C language.

Our last effort uses Pthreads to extract parallelism in an explicit way. This OpenMP version combined with Pthreads improves execution times between 5% for the small image and 9% for the larger one, and a good scalability is preserved on the multicore CPU (see Table 1, lower side).

3.1. Blocking for further optimizations

We decompose frames processing with the aim of improving data locality in our 3D-FWT code: Instead of calculating two entire frames, one for the detailed video and another one for the reference video, we split them into smaller blocks, on which the 2D-FWT with *tiling* is applied.

The last two rows in Table 1 report marginal gains when blocking is enabled, suggesting that the 3D-FWT is not a memory bandwidth bound kernel when running on a multicore CPU.

4. Compute Unified Device Architecture

The Compute Unified Device Architecture (CUDA) [3] 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.

4.1. Hardware platforms

All graphics hardware by Nvidia is 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, we have either the C870 card or the S870 1U rack-mount chassis endowed with four GPUs. These are based on the G80 architecture, recently upgraded with the GT200 GPU to release the Tesla C1060 and S1070 models. Our target architecture, the Tesla C870 card, 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.

4.2. Execution model

The G80 parallel architecture is a SIMD (Single Instruction Multiple Data) processor consisting of 128 cores. Cores are organized into 16 multiprocessors, each having a large set of 8192 registers, a 16 KB shared memory very close to registers in speed (both 32 bits wide), and constant and texture caches of a few kilobytes. 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.

Table 2: Major hardware and software limitations with CUDA. Constraints are listed for the G80 GPU.

Hardware feature	Value	Software limitation	Value
Multiprocessors (MP)	16	Threads / Warp	32
Processors / MP	8	Thread Blocks / MP	8
32-bit registers / MP	8192	Threads / Block	512
Shared Memory / MP	16 KB	Threads / MP	768

4.3. Programming Elements

The key elements for understanding optimizations of a CUDA program are the following:

- A program is decomposed into **blocks** running in parallel. Assembled by the developer, a block is a group of threads that is mapped to a single multiprocessor, where they share 16 KB of memory. All threads in blocks concurrently assigned to a single multiprocessor divide the multiprocessor's resources equally amongst themselves. This tradeoff between parallelism and thread resources must be wisely solved by the programmer on a certain architecture given its limitations, which are listed in Table 2 for the case of the Tesla C870.
- The developer explicitly divides the data amongst the *threads* in a block in a SIMD fashion. Context switch is very fast in CUDA, which encourages the programmer to declare a huge amount of threads structured in blocks to maximize parallelism and hide memory latency.
- The threads of a block are executed in groups of 32 threads called **warps**. A warp executes a single instruction at a time across all its threads in 4 clock cycles. The threads of a warp may follow their own execution path and this *divergence* is handled in hardware, but the GPU is much more efficient when all threads execute the same instructions on different data based on their unique thread ID.
- A **kernel** is the code to be executed by each thread. When a kernel is invoked on the GPU, the hardware scheduler launches thread blocks on available multiprocessors Threads run on different processors of the multiprocessors sharing the same executable and global address space, though they may not follow the same path of execution. A kernel is organized into a **grid** as a set of **thread blocks**.

4.4. Memory optimizations

4.4.1. Conflicts on shared memory banks

Attention must be paid to how the threads access the 16 banks of shared memory, since only when the data resides in different banks can all of the available ALU bandwidth truly be used.

Each bank only supports one memory access at a time; the rest are serialized, stalling multiprocessor's running threads until their operands arrive. The use of shared memory is explicit within a thread, which allows the programmer to solve bank conflicts wisely.

4.4.2. Coalescing on global memory accesses

Coalescing is another critical issue related to memory performance. A coalesced access involves a contiguous region of global memory where the starting address must be a multiple of region size and the k^{th} thread in a half-warp must access the k^{th} element in a block being read. This way, the hardware can serve completely two coalesced accesses per clock cycle, which maximizes memory bandwidth. Otherwise, scattered *(uncoalesced)* accesses results in *memory divergence* and requires to perform one memory transaction per thread.

It is programmer's responsibility to organize memory accesses sufficiently close together so that they can be *coalesced* on each GPU architecture. In general, memory accesses to consecutive positions are more than ten times faster than those using strides.

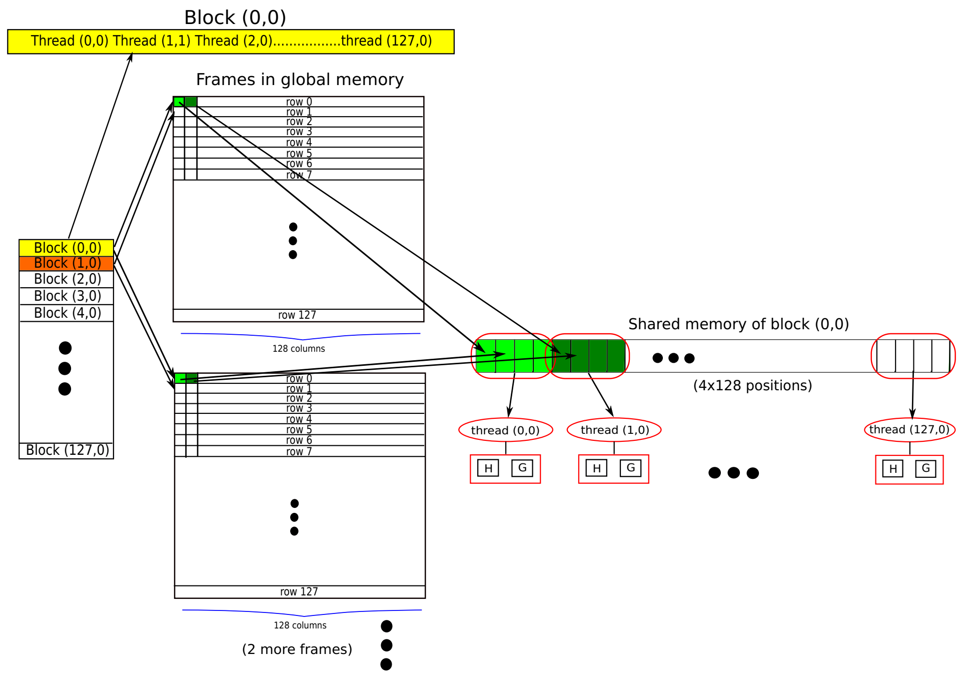

Figure 1: The CUDA kernel for computing the FWT on each video frame when an optimal block size of 128 threads is used. The diagram is tailored to a frame size of 128x128 pixels, which forces a topology of 128x1 blocks for the grid. The i-th block loads 4x128 pixels into shared memory and computes 128 pixels for H and G, storing them in its i-th row.

5. Parallelization on a manycore GPU

This section describes our parallelization strategies on a GPU using CUDA, along with some optimizations performed investing a similar effort to that outlined for the CPU case in section 3.

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. Figure 1 illustrates this procedure for an optimal block size of 128 threads. Each thread loads four pixels into shared memory and computes an output H and G pair. 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 (see Figure 2). 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. Figure 3 summarizes the way the entire process is implemented in CUDA. 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.

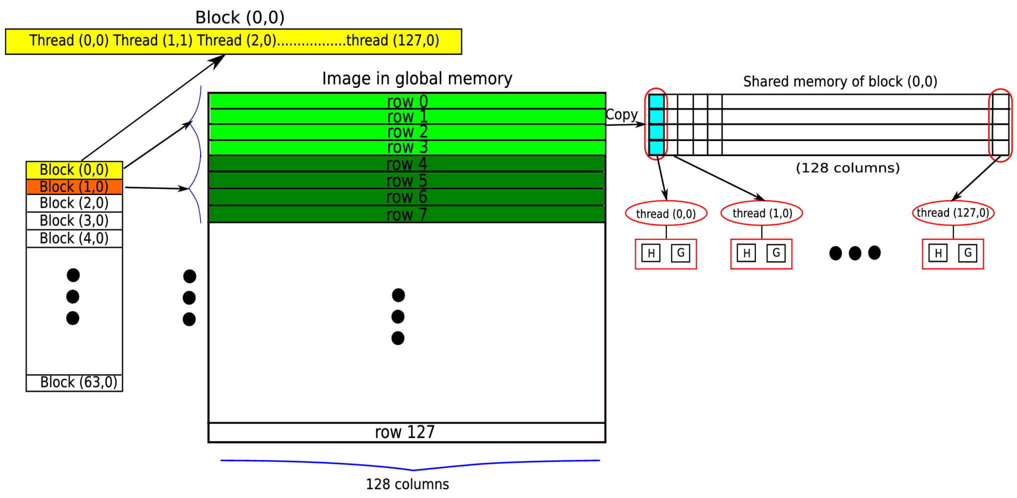

Figure 2: The CUDA kernel for computing the 2D-FWT on a 128 × 128 image.

Table 3: The reduction in the execution time (msecs.) when our CUDA optimizations are applied to the 3D-FWT code for a video of 64 frames of different sizes.

	Frame size		
Code version	512 x 512	1K x 1K	2K x 2K
CUDA baseline	59.97	225.95	879.34
CUDA optimized	57.65	216.66	843.11

5.1. CUDA optimizations

When each thread accesses to global memory to load its four pixels into shared memory, coalescing is ensured but conflicts arise on shared memory banks: The stride in the access pattern to memory is $s = 4$, and therefore, only four out of the sixteen banks are utilized on each half-warp. We have solved this issue by inserting a fifth null value from each thread, which transforms the stride into $s = 5$ to guarantee concurrent access to all sixteen banks. Table 3 reveals modest performance gains, in line with our implementation effort.

6. Performance analysis

6.1. GPU profiling

Now that the optimal GPU version has been attained, we may split its execution time into constituent steps for completing a quick profiling process. Table 4 reveals this breakdown, where we can see that each 1D-FWT phase contributes with a similar computational weight, but communication time predominates over calculations. This is a consequence of the nature of a 3D-FWT algorithm, which lacks of arithmetic intensity but handles big data volumes.

We believe this communication cost can be removed as long as the 3D-FWT represents a small fraction of a video processing application where the input video has to be transferred into GPU memory anyway, which represents a frequent case in real practice. Moreover, newer CUDA programming tools and underlying hardware allow to overlap data transfers via PCI-express with internal GPU computations, which may alleviate this overhead.

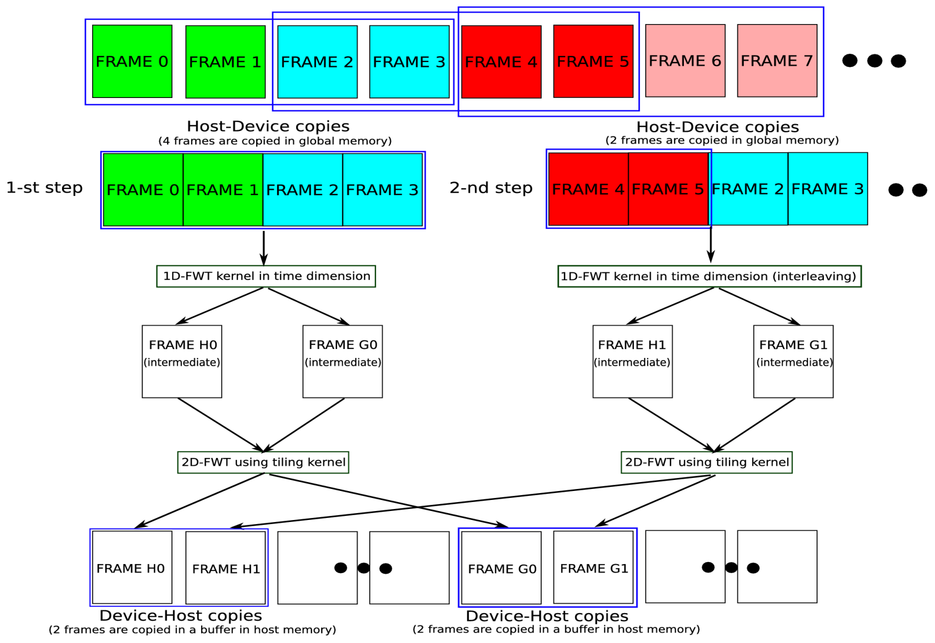

Figure 3: The way 3D-FWT is implemented in CUDA using interleaved accesses to video frames.

6.2. Comparison between platforms

Table 5 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. A similar programming effort and hardware cost was invested on each platform.

The GPU version exhibits better performance and scalability, with solid gains in all cases. The GPU speed-up factor extends into 15x factor in the most favorable case, the bigger we may run on the GPU due to video memory constraints. In general, the GPU acceleration keeps in the expected range for a class of algorithms like the 3D-FWT with low arithmetic intensity, pseudo-regular access patterns and intricate loop traversing.

6.3. Combined tuning

The CPU and the GPU execution time may be confronted from a performance analysis viewpoint, but with a more realistic and profitable perspective we may travel to a tantalizing scenario: A bi-processor platform, where each hardware contributes to speed-up the application, either enabling few CPU cores or an army of streaming processors.

A straightforward high-level process may partition the 3D-FWT loops to assign 1D-FWT computations to CPU or GPU inversely to their estimated latency, which can be taken from those times reported in Table 5. For example, when communication cost is considered, three 1Kx1K 1D-FWTs will be assigned to the GPU for each one computed on the CPU, and when this cost is neglected, up to fifteen 2Kx2K 1D-FWT may be computed on the GPU for a single on the CPU.

A combined effort is also feasible on applications performing 3D-FWTs over a list of queued videos on a batch processing basis: Larger videos are mapped to the GPU, whereas smaller ones stay on the CPU for an even workload balance on each platform to mazimize task parallelism.

Table 4: CUDA execution times (in msecs.) for our optimal tiled 3D-FWT implementation on an input video containing 64 frames of increasing sizes. Breakdown into major stages, where steps 3 and 4 complete the 2D-FWT and steps 1 and 5 represent the communication cost, which is removed in the last row.

	Frame size		
3D-FWT stage	512x512	1Kx1K	2Kx2K
1. CPU to GPU transfer	21.57	83.42	332.17
2. 1D-FWT on frames	4.03	14.26	55.79
3. 1D-FWT on rows	5.04	18.02	68.54
4. 1D-FWT on cols	5.19	17.33	65.94
5. GPU to CPU transfer	21.82	83.63	320.67
Computational time (2-4)	**14.61**	**49.59**	**190.27**

Table 5: Summary of execution times (msecs.) for the 3D-FWT (GPU factor gains between parenthesis).

Platform and	Frame size		
code version	512 x 512	1K x 1K	2K x 2K
CPU using an optimal number of threads	156.09	655.33	2843.43
CUDA optimized (see 1-5 in Table 4)	57.65 (2.7x)	216.66 (3.0x)	843.11 (3.4x)
CUDA computational (see 2-4 in Table 4)	14.61 (10.7x)	49.59 (13.2x)	190.27 (15.0x)

Overall, our programming efforts on multicore CPUs and manycore GPUs provide multiple chances for a compatible scenario where they may cooperate for an additional performance boost.

7. Summary and conclusions

In this paper, 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, whereas CUDA was selected for exploiting data parallelism on the GPU with an explicit memory handling.

Similar programming efforts and hardware costs were invested on each side for a fair architectural comparison, where GPU speed-up extends between 3x and 15x depending on problem size. This fulfills our expectations for a class of algorithms like the 3D-FWT, where we face low arithmetic intensity and high data bandwidth requirements. Our performance gains also enable the GPU for real-time processing of the 3D-FWT in the near future, given that GPUs are highly scalable and become more valuable for general-purpose computing in the years to come.

Following this trend, the 3D-FWT may benefit extraordinarily given its leading role for understanding video contents in applied scientific areas and performing video compression in multimedia environments. Our work is part of the developing of an image processing library oriented to biomedical applications. Future achievements include the implementation of each step of our image analysis process so that it can be entirely executed on GPUs without incurring penalties from/to the CPU. Our plan also includes porting the code to CPU/GPU clusters.

Acknowledgments

This work was supported by the Spanish MEC and MICINN, as well as European Comission FEDER funds, under Grants CSD2006-00046, TIN2009-14475-C04 and TIC2003-06623. It was also partly supported by the Junta de Andalucía of Spain under Grant TIC-02109.

References

[1] J. Franco, G. Bernabé, J. Fernández, M. Acacio, M. Ujaldón, The 2D Wavelet Transform on Emerging Architectures: GPUs and Multicores, *Submitted for publication to* Journal of Real-Time Image Processing.
[2] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, T. J. Purcell, A Survey of General-Purpose Computation on Graphics Hardware, Computer Graphics Forum 26 (1) (2007) 80–113.
[3] Nvidia, CUDA Zone maintained by Nvidia, http://www.nvidia.com/object/cuda.html (2009).
[4] AMD, AMD Stream Computing, http://ati.amd.com/technology/streamcomputing/index.html (2009).
[5] Nvidia, Tesla GPU Computing Solutions, http://www.nvidia.com/object/tesla_computing_solutions.html (2009).
[6] The Khronos Group, The OpenCL Core API Specification, http://www.khronos.org/registry/cl.
[7] T. T. Wong, C. S. Leung, P. A. Heng, J. Wang, Discrete Wavelet Transform on Consumer-Level Graphics Hardware, IEEE Transactions on Multimedia 9 (3) (2007) 668–673.
[8] C. Tenllado, J. Setoain, M. Prieto, L. P. . nand F. Tirado, Parallel Implementation of the 2D Discrete Wavelet Transform on Graphics Processing Units: Filter Bank versus Lifting, IEEE Transactions on Parallel and Distributed Systems 19 (2) (2008) 299–310.
[9] S. Mallat, A Theory for Multiresolution Signal Descomposition: The Wavelet Representation, IEEE Transactions on Pattern Analysis and Machine Intelligence 11 (7) (1989) 674–693.
[10] G. Bernabé, J. González, J. M. García, 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, 2000, pp. 226–231.
[11] I. Daubechies, Ten Lectures on Wavelets, Society for Industrial and Applied Mathematics, 1992.
[12] P. Meerwald, R. Norcen, A. Uhl, Cache Issues with JPEG2000 Wavelet Lifting, in: Proceedings of Visual Communications and Image Processing Conference, 2002, pp. 626–634.
[13] J. Tao, A. Shahbahrami, B. Juurlink, R. Buchty, W. Karl, S. Vassiliadis, Optimizing Cache Performance of the Discrete Wavelet Transform Using a Visualization Tool, Procs. of IEEE Intl. Symposium on Multimedia (2007) 153–160.
[14] A. Shahbahrami, B. Juurlink, S. Vassiliadis, Improving the Memory Behavior of Vertical Filtering in the Discrete Wavelet Transform, in: Proceedings of ACM Conference in Computing Frontiers, 2006, pp. 253–260.
[15] ICC, Intel Software Network, http://software.intel.com/en-us/intel-compilers/ (2009).
[16] GCC, GCC, the GNU Compiler Collection, http://gcc.gnu.org (2009).
[17] OpenMP, The OpenMP API Specification, http://www.openmp.org (2009).