ANN. MULT. GPU PROG.

Parallel 3D Fast Wavelet Transform comparison on CPUs and GPUs

Gregorio Bernabé

Correspondence:
gbernabe@ditec.um.es
Computing Engineering, University of Murcia,Campus de Espinardo, 30071 Murcia, Spain
Full list of author information is available at the end of the article

Abstract

We present in this paper several implementations of the 3D Fast Wavelet Transform (3D-FWT) on multicore CPUs and manycore GPUs. On the GPU side, we focus on CUDA and OpenCL programming to develop methods for an efficient mapping on manycores. 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. We evaluate these proposals and make a comparison between a new Fermi Tesla C2050 and an Intel Core 2 Quad Q6700. Speedups of the CUDA version 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.

Keywords: 3D Fast Wavelet Transform; parallel programming; multicore; CUDA; OpenCL

1 Introduction

Nowadays multicore architectures are omnipresent and can be found in all market segments. In particular, they constitute the CPU of many embedded systems (for example, video game consoles, network processors or GPUs), personal computers (for example, the latest developments from Intel and AMD), servers (the IBM Power6 or Sun UltraSPARC T2 among others) and even supercomputers (for example, the CPU chips used as building blocks in the IBM Blue-Gene/L and Blue-Gene/P systems). This market trend towards CMP (or chip-multiprocessor) architectures has given rise to platforms with a great potential for scientific computing such as the GPGPUs [1][2].

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 [3] 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 [4].

Both companies have also developed hardware products aimed specifically 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. Moreover, it provides more mechanisms to optimize general-purpose applications.

More recently, Open Computing Language (OpenCL) is a framework [6] 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. First, this field has traditionally proven to be a great success for GPUs during its evolution towards high-performance general-purpose computing. Second, 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 and OpenCL provide 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 one of the main focus of this paper.

In previous works [7][8], 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.

In this work, we extend our analysis to the 3D scenario, where speed-up factors have been improved using a new set of optimization techniques. We present different alternatives and programming techniques for an efficient parallelization of the 3D Fast Wavelet Transform on multicore CPUs and manycore GPUs. OpenMP and Pthreads will be selected on the CPU to build different implementations. CUDA and OpenCL will be used for exploiting the potential of the GPU. A comparison between different platforms will be showed to determine the best version. The work presented in this paper is a major revision and an extension of two previous papers published by the authors in [9] and [10].

The rest of this paper is organized as follows. Section 2 summarizes the background to wavelets and the previous work. In Section 3 we describe our implementation effort on multicore CPUs. Sections 4 and 5 focus on the specifics of the GPU programming with CUDA and OpenCL, and Section 6 outlines the different GPU implementations. Experimental results in GPU are analyzed in Section 7. Moreover, this section includes a comparison of different platforms. Section 8 summarizes the work and concludes the paper.

2 Background

2.1 The Wavelet Transform Foundations

The basic idea of the wavelet transform is to represent any arbitrary function f as a weighted sum of functions, referred to as wavelets. Each wavelet is obtained from a mother wavelet function by conveniently scaling and translating it. The result is equivalent to decomposing f into different scale levels (or layers), where each level is then further decomposed with a resolution adapted to that level.

In a multiresolution analysis, there are two functions: 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 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 [11].

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 [11]. 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 [12], we consider Daubechie's W_4 mother wavelet [13] 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. [14] propose two techniques to overcome this problem: row extension and aggregation or tiling.

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.

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 [15][16], 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. 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) [17], a corporate tool, and GCC (GNU Compiler Collection) [18], 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.

2.2 Previous work

In the last few years, a very attractive area of research involves the proposal and evaluation of different transform functions that may overcome the limitations that the DCT used by MPEG-2 presents

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 and command-line flags. (*) Optimal flags are -03, -parallel, -par-threadshold0 -xT.

Programming language and/or tool	Optimizations		Frame size		
	Tiles	Blocks	512 x 512	1K x 1K	2K x 2K
C and g++ compiler	No	No	990.76	5245.62	34798.70
C and g++ (-O3 flags)	Yes	No	392.77 (2.5x)	1592.72 (3.3x)	6404.40 (5.4x)
C and icc (-O3 flags)	Yes	No	309.46 (3.2x)	1272.16 (4.1x)	5023.09 (6.9x)
C using icc optimal (*)	Yes	No	305.37 (3.2x)	1231.34 (4.2x)	4773.52 (7.3x)

for some particular types of video. Wavelet techniques have recently generated much interest in applied areas and the wavelet transform has been mainly applied to image compression. Moreover, the latest image compression standard, JPEG-2000 [19][20], is also based on the 2D discrete wavelet transform with a dyadic mother wavelet transform. The 3D wavelet transform has been also applied for compressing video. Since one of the three spatial dimensions can be considered similar to time, Chen and Pearlman developed a three-dimensional subband coding to code video sequences [21], posteriorly improved with an embedded wavelet video coder using 3D set partitioning in hierarchical trees (SPHIT) [22]. Today, the standard MPEG-4 [23][24] supports an ad-hoc tool for encoding textures and still images, based on a wavelet algorithm. In a previous work [25], we have presented the implementation of a lossy encoder for medical video based on the 3D fast wavelet transform. This encoder achieves both high compression ratios and excellent quality, so that medical doctors can not find longer differences between the original and the reconstructed video. Furthermore, the execution time achieved by this encoder allows for real-time video compression and transmission.

In the last few years the rapid development of the graphics processor, coupled with recent improvement in its programmability have implied and explosion in interest on general purpose computation on graphics processors (GPGPU). The wide deployment of GPU in the last several years has resulted in a widely range of applications implemented on a graphics processor such as physics simulations [26][27], signal and image processing [28][29], computer vision [30], global illumination [31], geometric computing [32] or databases and data mining [33].

In the scope of the mathematical transforms several projects have developed GPU implementations of the Fast Fourier Transform (FFT). The FFT has been implemented in GPUs [34][35][36]. Based on these works, Sumanaweera and Liu [29] presented an implementation of the 2D-FFT in a GPU performing image reconstruction in magnetic resonance imaging (MRI) and ultrasonic imaging. On the same way that the 2D-FWT, the 2D-FFT could be obtained processing 1D-FFT across all columns of an image and then doing another 1D-FFT across all rows. Their implementation automatically balances the load between the vertex processor, the rasterizer, and the fragment processor; it also used other improvements for providing better performance (close to a factor of two) on the Nvidia Quadro NV4x family of GPUs compared to the CPUs in medical image reconstruction at a cheaper cost. Recently, the 1D-FFT, 2D-FFT and 3D-FFT have been included in CUDA libraries [37]. For 2D and 3D transforms, CUFFT (is the CUDA FFT library) performs transforms in row-major (C-order).

In the context of wavelet transform, recently, there has been several implementations of the 2D-FWT on a GPU. In [38], it presented a SIMD algorithm that performs the 2D-DWT completely on the GPU NVidia 7800 GTX. Their implementations adopted OpenGL and Cg for shader development. Evaluations showed speedups between 2.68 and 7.36 in the execution time over a version executed on an AMD Athlon 64x2 dual core processor 3800+ 2.01 GHz. The afore mentioned speedups are apparent for encoding high-resolution images from $1024x1024$ pixels and CPU-GPU time data transfers were ignored. On the same way, Tenllado et. al. [39] explored the implementation of the 2D-DWT on

modern GPUs as NVidia FX5950 Ultra and Nvidia 7800 GTX, and focused on analyzed and compared the actual performance of the Filter Bank Scheme (FBS) [11] and Lifting Scheme (LS) [40], (LS consists in an implementation of the wavelet transform based on integers), which are the most popular choices for implementing the DWT. The implementation are coded using Cg and OpenGL. Ignoring CPU-GPU data transfers, the GPU implementations obtained better performance than a highly tuned CPU implementation on an Intel's Prescott processor of 3.4 GHz. Results showed speedup factors between 1.2 and 3.4. In [41], a novel fast wavelet lifting implementation on graphics hardware using CUDA, which extends to any number of dimensions is proposed. The method tries to maximize coalesced memory access. In [42] a study of the split-and-merge (SM) method in the context of general purpose computation on GPU is presented. SM has been applied to multilevel algorithms such as the 2D wavelet transform and some tridiagonal system solvers. Implementations are coded using CUDA. A comparison to the work presented in [39] demonstrated the SM algorithm is slightly more efficient in the case of the FBS while the efficiency is higher for the LS since, in this case, the SM method saves a greater number of accesses to the global memory. In a previous work [7], we contributed with a CUDA implementation for the 2D-FWT obtaining speedups between 21.06 and 26.81 with regard a sequential C version on a CPU, ignoring the communication time. Our optimized CUDA version is 1.91 and 2.59 better than optimized OpenMP and Pthreads versions implemented on multicore CPUs.

3D wavelet reconstruction has also been implemented on a GPU. Garcia and Shen [43] described a GPU-based algorithm for reconstructing 3D wavelets using fragment programs which uses tileboards as a primary layout to organize 3D wavelet coefficients. They used Cg and OpenGL to evaluate the reconstruction formulae. Results obtained speedup of up to 6.76 on an NVidia Quadro FX 3400 over a 3.0 GHz Intel Xeon processor.

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`.

3.1 Automatic parallelization

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.

1 `-parallel`: Detects simply structured loops capable of being executed safely in parallel, and automatically generates multi-threaded code for those loops.
2 `-par-threshold0`: Used in conjunction with `parallel`, this flag sets a threshold for the automatic parallelization of loops based on the probability of a profitable parallel execution. The value 0 commands the compiler to parallelize loops regardless of their computational workload.
3 `-xT`: Generates specialized code and enables vectorization.

Execution times (see the first and second rows in Table 2) report modest compiler gains, which encourages us to get involved in the parallelization process.

3.2 OpenMP for a semi-automatic parallelization

OpenMP [44] 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

Table 2 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 optimal baseline 3D-FWT using icc compiler is compared against parallel versions with command lines flags. (*) Optimal flags are -03, -parallel, -par-threadshold0 -xT.

Programming language and/or tool	Parallel	Optimizations		Frame size		
		Tiles	Blocks	512 x 512	1K x 1K	2K x 2K
C and icc (-O3 flags)	No	Yes	No	309.46	1272.16	5023.09
C using icc optimal (*)	No	Yes	No	305.37	1231.34	4773.52
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

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 2 (the third row). 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.

3.3 Pthreads for a more explicit effort

Our last effort uses Pthreads to extract parallelism in an explicit way.

This requires a more demanding effort, namely:

- Extract the code for the slave thread to release a separated function, as Pthreads require the entry point of each new thread to be a function. In our particular case, we have extracted the same the loop that it was parallelized with OpenMP in the previous section.
- Create an auxiliary data structure for passing parameters. Functions to be used as the entry point of a thread can only receive a single parameter, which must be a **void** pointer. This forces us to create an auxiliary structure with one field for each parameter, and pass a pointer to this structure.
- Specify an explicit distribution for the frames of the main loop to each CPU core, which in our case is performed depending on the thread and job IDs.

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 2, the fourth row).

3.4 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.

This is translated into the pseudo-code shown in Figure 1 for the main loop, where we also illustrate the OpenMP directive required to derive the semi-automatic version.

The last two rows in Table 2 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.

```
#pragma omp parallel for default(none)
        private(i,j,k) shared(partition, frames, ...)
for (i=0; i<frames; i+=2) {
  j=i/2;
  for (k=0; k<partition; k++) {
    fwt_frames(rows/partition, cols, in_image, tmp1);
    // 2D-FWT with tiling for the Low frame
    fwt2DTiling(rows/partition, ...);
    // 2D-FWT with tiling for the High frame
    fwt2DTiling(rows/partition, ...);
  }
      ...
}
```

Figure 1 The OpenMP version for the 3D-FWT on a multicore CPU when blocking is applied.

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.

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.

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 Fermi parallel architecture is a SIMD (Single Instruction Multiple Data) processor. In C2050 cores are organized into 14 multiprocessors. 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. 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

Table 3 Major hardware and software limitations with CUDA. Constraints are listed for the Fermi Tesla C2050 GPU.

Hardware feature	Value	Software limitation	Value
Multiprocessors (MP)	14	Threads / Warp	32
Processors / MP	32	Thread Blocks / MP	8
32-bit registers / MP	32768	Threads / Block	1024
Shared Memory / MP	16 KB/48KB	Threads / MP	1536
L1 Cache / MP	48 KB/16 KB		
L2 Cache	768 KB		

Table 4 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

parallelism and thread resources must be wisely solved by the programmer to maximize execution efficiency on a certain architecture given its limitations. These limitations are listed in Table 3 for the Tesla C2050.

5 Open Computing Language

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 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 4, although both, CUDA and OpenCL, do mostly the same.

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.

In the OpenCL execution model, kernels are executed by one or more work-items. Work-items are collected into work-groups and each work-group executes on a compute unit. In the OpenCL memory model, kernel data must be specifically placed in one of four address spaces: global memory, constant memory, local memory or private memory. The location of the data determines how quickly it can be processed.

6 Parallelization on a manycore GPU

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

6.1 CUDA implementation of 3D-FWT

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

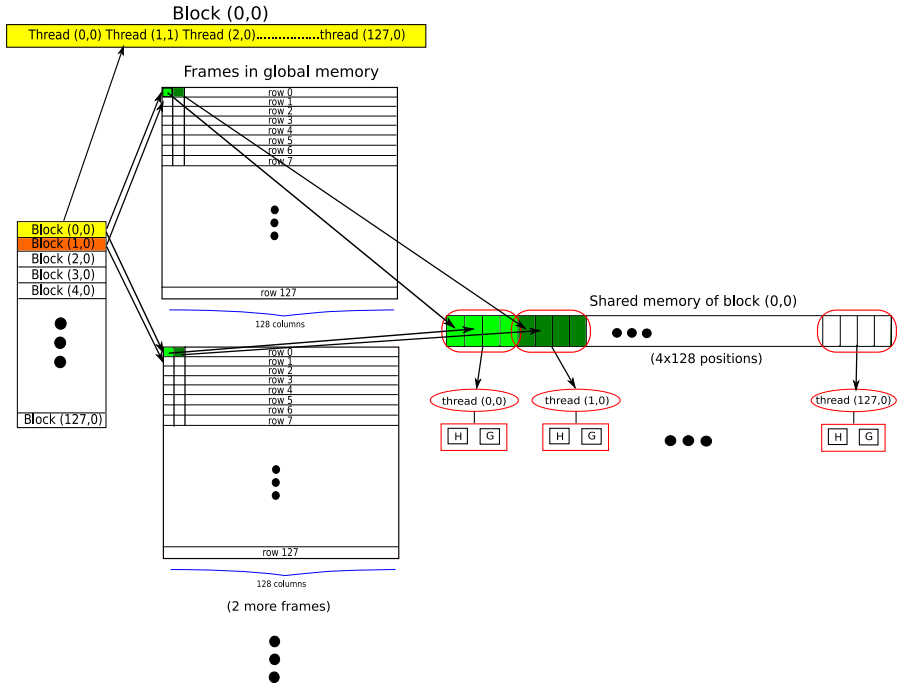

Figure 2 The CUDA kernel for computing the FWT on each video frame when an optimal block size of 128 threads is used. The diagram is tailored to a frame size of 128x128 pixels, which forces a topology of 128x1 blocks for the grid. The i-th block loads 4x128 pixels into shared memory and computes 128 pixels for H and G, storing them in its i-th row.

1 The *host* (CPU) allocates in memory the first four video frames coming from a .pgm file.

2 The wavelet transform is applied one level in each dimension to obtain a better trade off between compression ratio and quality.

3 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 2 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.

4 The 2D-FWT is applied to the frame belonging to the detailed video, and subsequently, to the reference video (see Figure 3). 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 4 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. Conflicts on shared memory banks and coalescing on global memory accesses has been solved.

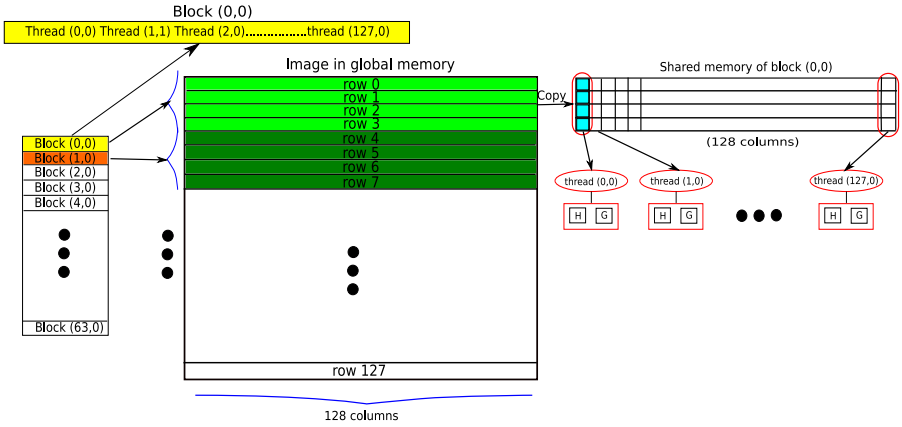

Figure 3 The CUDA kernel for computing the 2D-FWT on a 128×128 image.

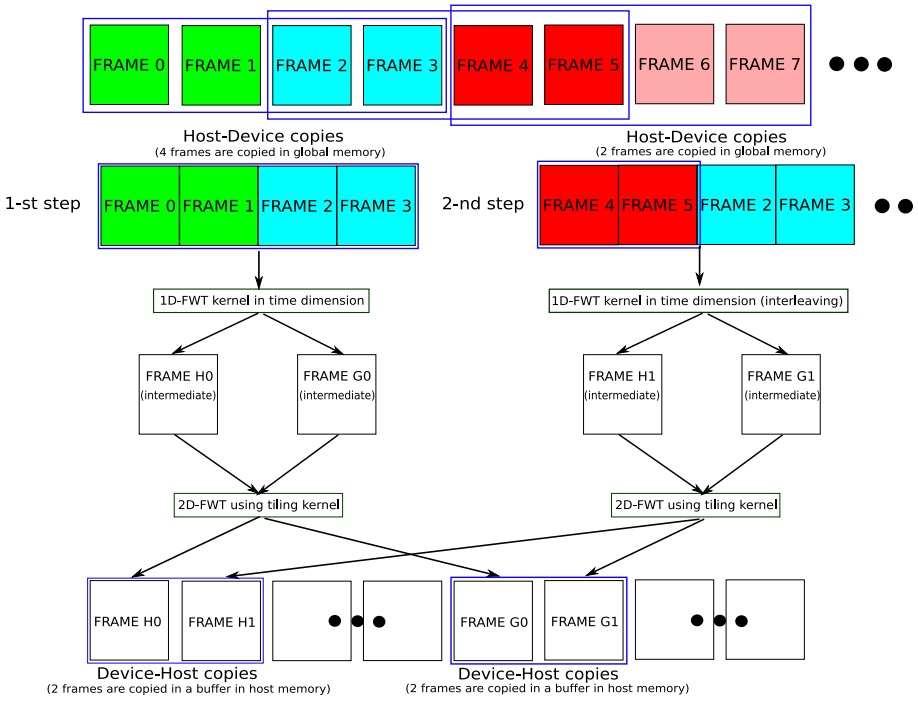

Figure 4 The way 3D-FWT is implemented in CUDA using interleaved accesses to video frames.

6.2 OpenCL implementation

Our 3D-FWT implementation in OpenCL is based on the same CUDA algorithm with the three major steps described in the previous section. We use simple source to source translation to convert the kernels of the implementation of 3D-FWT on CUDA to OpenCL. Our effort have been lesser to that outlined for the CUDA version, although there are some differences between CUDA and OpenCL in terminology, the model is similar and is simple to transform the kernels.

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

Platform and code version	Frame size					
	512 x 512		1K x 1K		2K x 2K	
CPU optimal numbers of threads	156.09		655.33		2843.43	
CUDA implementation	29.21	(5.3x)	100.61	(6.5x)	381.58	(7.4x)
OpenCL implementation	87.12	(1.8x)	276.39	(2.4x)	1011.47	(2.8x)

Table 6 OpenCL and 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.

3D-FWT stage – OpenCL	Frame size		
	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
3D-FWT stage – CUDA	Frame size		
	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

7 Performance analysis

7.1 Comparison between platforms

Table 5 summarizes the optimal execution times we have obtained on each hardware platform (multicore CPU and GPU versions with CUDA and OpenCL) 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.

We have included our optimal tiled 3D-FWT implementation designed with OpenMP and Pthreads. This version is executed on an Intel Core 2 Quad Q6700 CPU (see Table 2, last row).

The GPU version with CUDA exhibits better performance and scalability, with solid gains in all cases. As the size of images increase, the difference between the CUDA implementation and the CPU optimal is bigger. The CUDA version speed-up factor extends into 7.4x factor in the most favorable case. 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.

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. 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.

7.2 GPU profiling

For both GPU versions with OpenCL and CUDA, we may split its execution time into constituent steps for completing a quick profiling process. Table 6 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.

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. In our experiments, the communication time represents an average of 85% and 91% for the different configurations of the CUDA and OpenCL versions, respectively. Therefore, we can obtain an improvement of 15% for CUDA and 9% for OpenCL in the execution time, if we use streaming to overlap the CPU-GPU transfers with the kernels computing in the GPU.

7.3 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 Tables 5 and 6. For example, when communication cost is considered, six and two 1Kx1K 1D-FWTs in the CUDA and OpenCL versions, respectively, will be assigned to the GPU for each one computed on the CPU, and when this cost is neglected, up to eighty and sixty 2Kx2K 1D-FWT in the CUDA and OpenCL implementations, respectively, 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 maximize task parallelism.

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.

8 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 and OpenCL were selected for exploiting data parallelism on a new Fermi Tesla GPU with an explicit memory handling.

Similar programming efforts and hardware costs were invested on each side for a fair architectural comparison. The implementation on CUDA achieves better speedups, ranging from 5.3x to 7.4x for different image sizes. OpenCL presents gains up to 2.8x with regard the best implementation on CPU. However, these outcomes are even lower than those obtained with CUDA. 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. 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.

If we discard the cost of communications between CPU and GPU, profits rise to a factor of 80.6x for larger image in the CUDA version. 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.

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.

References

1. D. Manocha, General-Purpose Computation Using Graphic Processors, IEEE Computer 38 (8) (2005) 85–88.
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 (2011).
7. J. Franco, G. Bernabé, J. Fernández, M. Ujaldón, The 2d wavelet transform on emerging architectures: Gpus and multicores, *Accepted and published online in* Journal of Real-Time Image Processing. http://dx.doi.org/10.1007/s11554-011-0224-7 (October 2011).
8. J. Franco, G. Bernabé, J. Fernández, M. E. Acacio, A Parallel Implementation of the 2D Wavelet Transform Using CUDA, in: 17^{th} Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, Weimar, Germany, 2009.
9. J. Franco, G. Bernabé, J. Fernández, M. Ujaldn, Parallel 3D fast wavelet transform on manycore GPUs and multicore CPUs, in: 10^{th} International Conference on Computational Science, Amsterdam, Netherlands, 2010.
10. G. Bernabé, G. D. Guerrero, J. Fernández, CUDA and OpenCL Implementations of 3D Fast Wavelet Transform, in: 3^{rd} IEEE Latin American Symposium on Circuits and Systems, Playa del Carmen, Mexico, 2012.
11. S. Mallat, A Theory for Multiresolution Signal Descomposition: The Wavelet Representation, IEEE Transactions on Pattern Analysis and Machine Intelligence 11 (7) (1989) 674–693.
12. 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.
13. I. Daubechies, Ten Lectures on Wavelets, Society for Industrial and Applied Mathematics, 1992.
14. 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.
15. 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.
16. 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.
17. ICC, Intel Software Network, http://software.intel.com/en-us/intel-compilers/ (2009).
18. GCC, GCC, the GNU Compiler Collection, http://gcc.gnu.org (2009).
19. M. W. Marcellin, M. J. Gormish, A. Bilgin, M. P. Boliek, An Overview of JPEG-2000, in: Proceedings of Data Compression Conference, 2000.
20. D. Santa-Cruz, T. Ebrahimi, A Study of JPEG 2000 Still Image Coding Versus Others Standards, in: Proceedings of X European Signal Processing Conference, 2000.
21. Y. Chen, W. A. Pearlman, Three-Dimensional Subband Coding of Video Using the Zero-Tree Method, Proc. of SPIE-Visual Communications and Image Processing (1996) 1302–1310.
22. Y. Kim, W. A. Pearlman, Stripe-based spiht lossy compression of volumetric medical images for low memory usage and uniform reconstruction quality, in: Proceedings of International Conference on Acoustics, Speech and Signal Processing, 2000, pp. 2031–2034.
23. S. Battista, F. Casalino, C. Lande, MPEG-4: A Multimedia Standard for the Third Millenium, Part 1, IEEE Multimedia (October 1999).
24. S. Battista, F. Casalino, C. Lande, MPEG-4: A Multimedia Standard for the Third Millenium, Part 2, IEEE Multimedia (January 2000).
25. G. Bernabé, J. M. García, J. González, Reducing 3D Wavelet Transform Execution Time Using Blocking and the Streaming SIMD Extensions, Journal of VLSI Signal Processing 41 (2) (2005) 209–223.
26. P. Sander, N. Tartachuk, J. L. Mitchell, Explicit early-z culling for efficient fluid flow simulation and rendering, Technical Report, ATI Research Journal (August 2004).
27. M. Harris, Fast Fluid Dynamics Simulation on the GPU. In GPU Gems, Addisson Wesley, 2004.
28. I. Viola, A. Kanitsar, M. E. Groller, Hardware-Based Nonlinear Filtering and Segmentation Using High-Level Shading Languages, IEEE Visualization (2003) 309–316.

29. T. Sumanaweera, D. Liu, Medical Image Reconstruction with the FFT. In GPU Gems, Addisson Wesley, 2004.
30. R. Yang, M. Pollefeys, A Versatile Stereo Implementation on Commodity Graphics Hardware, Real Time Imaging 11 (1) (2005) 7–18.
31. D. Weiskopf, T. Schafhitzel, T. Ertl, GPU-Base Bonlinear Ray Tracing, Computer Graphics Forum 23 (3) (2004) 625–633.
32. N. K. Govindaraju, M. Henson, M. C. Lin, D. Manocha, Interactive Visibility Ordering of Geometric Primitives in Complex Environments, Symposium on Interactive 3D Graphics and Games (2005) 49–56.
33. N. K. Govindaraju, B. LLoyd, W. Wang, M. Lin, D. Manocha, Fast Computation of Database Operations Using Graphics Processors, ACM SIGMOD International Conference on Management of Data (2004) 215–226.
34. M. Ansari, Video image processing using shaders, Presentation at Game Developers Conference (2003).
35. J. Sptizer, Implementing a CPU-Efficient FFT, Nvidia Course Presentation, SIGGRAPH.
36. K. Moreland, E. Angel, The FFT on a GPU, Graphics Hardware (2003) 112–119.
37. NVIDIA Corporation, NVIDIA CUDA CUFFT Library Version 1.1 (October 2007).
38. 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.
39. 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.
40. A. R. Calderbank, I. Daubechies, W. Sweldens, B. Yeo, Wavelet Transforms that Map Integers to Integers, Applied and Computational Harmonic Analysis 5 (3) (1998) 332–369.
41. W. J. V. der Laan, A. C. Jalba, J. B. Roerdink, Accelerating wavelet lifting on graphics hardware using CUDA, Parallel and Distributed Systems, IEEE Transactions on 22 (1) (2011) 132–146.
42. F. Argello, D. B. Heras, M. Bo, J. Lamas-Rodrguez, The split-and-merge method in general purpose computation on GPUs, Parallel computing 38 (6) (2012) 227–288.
43. A. García, H. Shen, GPU-Based 3D Wavelet Reconstruction with Tileboarding, The Visual Computer 21 (8–10) (2005) 755–763.
44. OpenMP, The OpenMP API Specification, http://www.openmp.org (2009).