A Parallel Implementation of the 2D Wavelet Transform Using CUDA

Joaquín Franco, Gregorio Bernabé, Juan Fernández and Manuel E. Acacio
Dpto. de Ingeniería y Tecnología de Computadores
Universidad de Murcia, Campus de Espinardo s/n, 30100 Murcia (SPAIN)
email:{j.franco,gbernabe,juanf,meacacio}@ditec.um.es

Abstract

There is a multicore platform that is currently concentrating an enormous attention due to its tremendous potential in terms of sustained performance: the NVIDIA Tesla boards. These cards intended for general-purpose computing on graphic processing units (GPGPUs) are used as data-parallel computing devices. They are based on the Computed Unified Device Architecture (CUDA) which is common to the latest NVIDIA GPUs. The bottom line is a multicore platform which provides an enormous potential performance benefit driven by a non-traditional programming model. In this paper we try to provide some insight into the peculiarities of CUDA in order to target scientific computing by means of a specific example. In particular, we show that the parallelization of the two-dimensional fast wavelet transform for the NVIDIA Tesla C870 achieves a speedup of 20.8 for an image size of 8192x8192, when compared with the fastest host-only version implementation using OpenMP and including the data transfers between main memory and device memory.

Keywords: 2D fast wavelet transform, parallel programming, multicore processor, CUDA, NVIDIA Tesla.

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 [25] whose best representative is the NVIDIA Tesla GPGPU series [20].

CUDA [21] is a new hardware and software architecture for issuing and managing computations on the GPU, without the need of mapping them to a graphics API [12], common to the latest NVIDIA developments. Each CUDA-enabled device behaves as a massively-threaded computing device with a significant amount of on-board memory. Thus, it is composed of a variable number of *thread processors* and a *thread execution manager* that handles threading automatically. Both the number of thread processors and the amount of on-board memory depend on the specific GPU model. Data set is divided into smaller chunks stored in the on-board memory in order to feed the thread processors. Thereafter threads are intended to run in lockstep in a SIMD fashion acting as a data-parallel computing device. In this sense, CUDA programmers don't have to write explicitly threaded code. Instead, the design of a correct data layout becomes the crucial task to obtain good performance and requires writing some explicit code.

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 discrete cosine transform (DCT) used by MPEG-2 presents 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 images. Several coders have been developed using the 2D wavelet transform [1] [16] [28]. The latest image compression standard, JPEG-2000 [18] [27], is also based on the 2D 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 [7], later improved with an embedded wavelet video coder using 3D set partitioning in hierarchical trees (SPHIT) [15]. Today, the standard MPEG-4 [2] [3] supports an ad-hoc tool for encoding textures and still images, based on a wavelet algorithm. In a previous work [5], 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 cannot 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 case of parallelizing the 2D fast wavelet transform, automatic parallelization methods do not yield enough benefits, while manual parallelization methods pose a considerable burden in software development [4].

1066-6192/09 $25.00 © 2009 IEEE
DOI 10.1109/PDP.2009.40

IEEE computer society

In this paper, we attempt to take advantage of the NVIDIA Tesla C870 that complies with the CUDA model to improve the execution time of the 2D fast wavelet transform. In particular, we provide performance results from an initial implementation which required a moderate development effort while achieving a remarkable speedup in terms of overall execution time.

The rest of this paper is organized as follows. Section 2 summarizes the background to wavelets. In Section 3 we provide an introduction to CUDA and the main details of our parallelization strategy for the 2D wavelet transform using CUDA. Experimental results are analyzed in Section 4. Finally, Section 5 summarizes the work and concludes the paper.

2. Background

In this section, we review the theory behind wavelets along with the previous work for wavelets and GPU programming.

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.

One way to achieve such a decomposition writes f as an integral over a and b of $\psi^{a,b}$ with appropriate weighting coefficients. However, it is preferred to write f as a discrete superposition. If a discretization is started with $a = a_0^m$ and $b = nb_0a_0^m$, with $m, n\epsilon Z$, and $a_0 > 1,\ b_0 > 0$ fixed. The wavelet decomposition is then

$$f = \sum c_{m,n}(f)\psi_{m,n} \tag{1}$$

with

$$\psi_{m,n}(t) = \psi^{a_0^m, nb_0a_0^m}(t) = a_0^{-\frac{m}{2}}\psi(a_0^{-m}t - nb_0) \tag{2}$$

In a multiresolution analysis, there are two functions: the mother wavelet ψ and its associated scaling function ϕ. Therefore, the wavelet transform can be implemented by quadrature mirror filters (QMF), $G = g(n)$ and $H = h(n)\ n\epsilon Z$, where $h(n) = \frac{1}{2} < \phi\left(\frac{x}{2}\right), \phi(x - n) >$, and $g(n) = (-1)^n h(1 - n)$ (<> denotes the space L^2 of all square integrable functions). H corresponds to a low-pass filter, and G is a high-pass filter. The reconstruction filters have impulse response $h^*(n) = h(1 - n)$, and $g^*(n) = g(1 - n)$. For a more detailed analysis of the relationship between wavelets and QMF see [17].

Table 1. Filter Coefficients for H of W_4.

Daubechie's W_4
$h_0 = 0.4829629131445341$
$h_1 = 0.8365163037378079$
$h_2 = 0.2241438680420134$
$h_3 = -0.1294095225512604$

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 down-sampled 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. It is important to note that the wavelet decomposition of a set of discrete samples has exactly the same number of samples as in the original, due to the orthogonality of wavelets. This procedure is referred to as the 1D Fast Wavelet Transform (1D-FWT). The inverse wavelet transform can be obtained in a way very similar to that for the forward transform by simply reversing the above procedure following. But the order of the g's and h's has to be reversed.

It is not difficult to generalize the one-dimensional wavelet transform to the multi-dimensional case [17]. The 2D case, introduces, like in the one-dimensional case, a scaling function $\phi(x, y)$ such that:

$$\phi(x, y) = \phi(x)\phi(y) \tag{3}$$

where $\phi(x)$ is a one-dimensional scaling function. 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).

Firstly, we consider Daubechie's W_4 wavelet mother [8] as a baseline function. We have chosen this function because some previous work have proved its effectiveness [6]. The mother wavelet basis for W_4 is shown in Table 1, from which the taps filters for G and the reconstruction filters can be derived.

In a traditional implementation based on wavelets, the 1D-FWT is applied to a row of an image, the pixels belonging to this row are only needed. Let us suppose that the W_4 mother wavelet (with four taps filters) is applied to images that have n by n pixels. Applying the 1D-FWT to a n-bits row, creates $n/2$ low-pixels and $n/2$ high-pixels. In order to compute the first-low-pixel and the first-high-pixel, the first, second, third and fourth original pixels are needed. The second-low-pixel and second-high-pixel depend on the third, fourth, fifth and

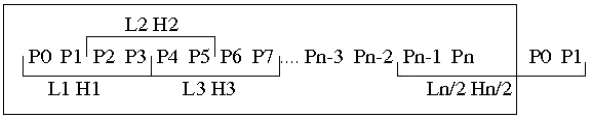


Figure 1. A traditional implementation with W_4.

sixth original pixels, and so on. Finally, in the last stage, the (n/2)th-low-pixel and (n/2)th-high-pixel depend on the (n-1)th, nth, first and second original pixel, so that when the last original pixel is reached, the process comes back to the beginning, as shown in Figure 1.

2.2. Previous Work

In the last few years the rapid development of the graphics processor, coupled with recent improvements in its programmability, have implied and explosion in interest on general purpose computation on graphics processing units (GPGPU). The wide deployment of GPUs in the last few years has resulted in a widely range of applications implemented on a graphics processor such as physics simulations [13] [26], signal and image processing [30] [32], computer vision [35], global illumination [33], geometric computing [10] or databases and data mining [11].

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 [19] [29]. Based on these works, Sumanaweera and Liu [30] presented an implementation of the 2D-FFT in a GPU performing image reconstruction in magnetic resonance imaging (MRI) and ultrasound 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 [23]. For 2D and 3D transforms, CUFFT (is the CUDA FFT library) performs transforms in row-major order (C-order).

In the context of the wavelet transform, there have been several implementations of the 2D-FWT on a GPU. In [34], it is presented a SIMD algorithm that performs the 2D-DWT completely on the GPU NVIDIA 7800 GTX. This implementation adopted Cg and OpenGL for shader development. Evaluation showed speedups between 2.68 and 7.36 in the execution time over a version executed on an AMD Athlon 64X2 dual-core 3800+ at 2.01 GHz. These speedups are apparent for encoding high-resolution images from 1024×1024 pixels when CPU-GPU data transfer times are ignored. On the same way, Tenllado *et al.* [31] explored the implementation of the 2D-DWT on modern GPUs as NVIDIA FX5950 Ultra and NVIDIA 7800 GTX. Their implementations were coded using Cg and OpenGL. Ignoring CPU-GPU data transfer times, the GPU implementations obtained better performance than a highly tuned CPU implementation on an Intel Prescott processor at 3.4 GHz. Results showed speedup factors between 1.2 and 3.4.

Finally, 3D wavelet reconstruction has also been implemented on a GPU. Garcia and Shen [9] described a GPU-based algorithm using fragment programs which uses tile-boards as a primary layout to organize 3D wavelet coefficients. They used Cg and OpenGL to evaluate the reconstruction formulae. Results obtained speedups of up to 6.76 on an NVIDIA Quadro FX 3400 over an Intel Xeon processor at 3.0 GHz.

3. Compute Unified Device Architecture

All the latest NVIDIA developments such as GeForce 8 series, Quadro FX 5600/4600 and Tesla solutions are compliant with the *Compute Unified Device Architecture* (CUDA). Nevertheless, the NVIDIA Tesla boards, namely C870, D870 and S870, are the only ones that have been specifically designed for general-purpose computing given the fact that they have no graphics output. In particular, the NVIDIA Tesla C870 is a homogeneous CMP, with 128 cores and 1.5 GB of on-board memory, attached to the main CPU through a PCIe x16 interface. Under this configuration, the NVIDIA Tesla features a theoretical 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. In turn, the NVIDIA Tesla D870 and S870 computing solutions comprise two and four C870 units in a desktop and a 1U rack-mount chassis, respectively. Therefore, they come with 256 and 512 cores, and 3 GB and 6 GB of on-board memory, to provide a theoretical peak performance of 1036.8 Gflops and 2073.6 Gflops, respectively.

3.1. Hardware Architecture

Each CUDA-compliant device is a set of multiprocessor cores (see Figure 2(a)), capable of executing a very high number of threads concurrently, that operates as a coprocessor to the main CPU or host. In turn, each multiprocessor has a SIMD architecture, that is, each processor of the multiprocessor executes a different thread but all the threads run the same instruction, operating on different data based on its `threadId`, at any given clock cycle. The NVIDIA Tesla C870 has sixteen multiprocessors with eight processors each.

Both the host and the device maintain their own DRAM, referred to as *host memory* and *device memory* (on-board memory). Device memory can be of three different types (see Figure 2(b)): *global memory*, *constant memory* and *texture memory*. They all can be read from or written to by the host and are persistent through the life of the application. Nevertheless, global, constant and texture memory spaces are optimized for different memory usages.

Multiprocessors have on-chip memory that can be of the four following types: *registers*, *shared memory*, *constant cache* and *texture cache* (see Figures 2(a) and 2(b)). Each processor in a multiprocessor has one set of local 32-bit read-write *registers* per processor. A parallel data cache of *shared memory* is shared by all the processors. A read-only *constant cache* is shared by all the processors and speeds up reads from the constant memory. A read-only *texture cache* is shared by all the processors and speeds up reads from the texture memory. The local and global memory spaces are implemented as read-write regions of device memory and are not cached. The NVIDIA Tesla C870 has 8192 registers and 16 KB of shared memory per multiprocessor.

3.2. Software Architecture

A portion of a parallel application that is executed many times, but independently on different data, can be isolated into a function that is executed on the device by many threads running on different processors of the multiprocessors. Such a function, called a *kernel*, is compiled to the instruction set of the device and downloaded into it.

A kernel is organized as a set of thread blocks as shown in Figure 2(c). A *thread block* is a batch of threads that can cooperate together by efficiently sharing data through the shared memory and synchronizing their execution to coordinate memory accesses using the primitive `__syncthreads()`. Each thread block executes on one multiprocessor. Each thread has its own *thread ID*, which is the number of the thread within a one-, two- or three-dimensional array of arbitrary size. The use of multidimensional identifiers helps to simplify memory addressing when processing multidimensional data.

The number of blocks in a thread block is limited (512 in the NVIDIA Tesla C870). Therefore, blocks of equal dimension and size that execute the same kernel can be batched together into a *grid of thread blocks*. This comes at the expense of reduced thread cooperation, because threads in different thread blocks from the same grid cannot communicate and synchronize with each other. In contrast, this model allows thread blocks of the same kernel grid to run on any multiprocessor, even from different devices, at any time. Again, each block is identified by its *block ID*, which is the number of the block within a one- or two-dimensional array of arbitrary size for the same reasons as above. It is worth noting that kernel threads are extremely lightweight, i.e. creation overhead is negligible and context switching is essentially free.

In this scenario, a thread can access device memory through the following memory spaces: read-write per-thread registers, read-write per-thread local memory, read-write per-block shared memory, read-write per-grid global memory, read-only per-grid constant memory and read-only per-grid texture memory. Note that the global, constant, and texture memory spaces are persistent across kernel launches.

3.3. Programming

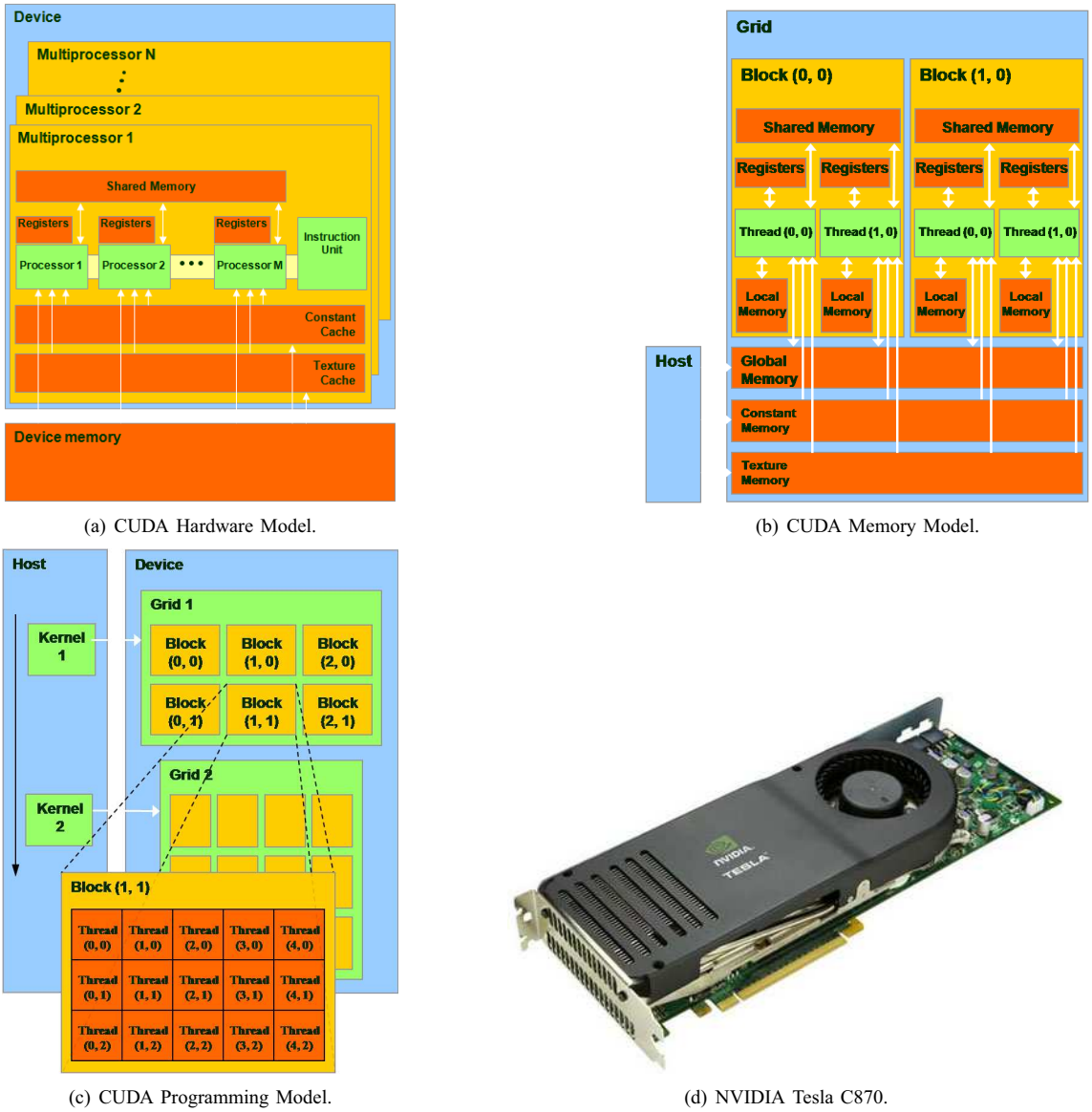
CUDA tries to simplify the programming model by hiding thread handling from programmers, i.e. there is no need to write explicit threaded code in the conventional sense. Instead, CUDA includes C/C++ software-development tools that allow programmers to combine host code with device code [12]. To do so, CUDA programming requires a single program written in C/C++ with some extensions [21]:

- Function type qualifiers to specify whether a function executes on the host or on the device and whether it is callable from the host or from the device (`__device__`, `__global__` and `__host__`).
- Variable type qualifiers for variables that reside on device memory (`__device__`, `__shared__` and `__constant__`).
- Four built-in variables that specify the grid and block dimensions, the block index within the grid and thread index within the block (`gridDim`, `blockDim`, `blockIdx` and `threadIdx`), accessible in `__global__` and `__device__` functions.
- An *execution configuration construct* to specify the dimension of the grid and blocks when launching kernels, declared with the `__global__` directive, from host code (for example, `function«gridDim, blockDim, shm_size»(parameter_list)`).

Each source file containing these extensions must be compiled with the CUDA `nvcc` compiler [12].

Besides, CUDA comes with a runtime library, split into a host component, a device component and a common component, that supports built-in vector data types and texture types, and provides a number of mathematical functions, type conversion and casting functions, thread synchronization functions, and device and memory management functions. Finally the CUDA environment also includes two higher-level mathematical libraries, namely CUBLAS [22] and CUFFT [23].

Kernel launches using the above mentioned execution configuration construct are asynchronous, that is, control returns to host immediately. Then, the main CPU is free to do whatever is required until `cudaThreadSynchronize()` is invoked so that the host blocks until all previous CUDA calls complete. In this way, memory allocation and movement of data between host memory and device memory



(a) CUDA Hardware Model.

(b) CUDA Memory Model.

(c) CUDA Programming Model.

(d) NVIDIA Tesla C870.

Figure 2. CUDA Architecture.

is left to programmers. They must allocate the required buffers in either host or device memory, and also copy data back and forth between host memory and device memory, using the functions provided by the runtime library (`cudaMalloc()`, `cudaMallocHost()`, `cudaFree()` and `cudaMemcpy()`).

3.4. Parallelization Strategy for 2D-FWT

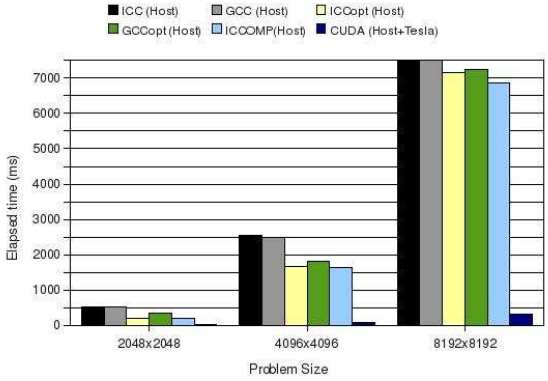
Firstly, we briefly describe the process to obtain the standard decomposition of an image using the 2D-FWT. Each image is processed as follows:

- A `.pgm` file is read into a page-locked main memory buffer.
- The main memory buffer is copied to a global memory buffer.
- The 1D-FWT is applied to each row of the image.
- The image matrix is transposed in order to be able to apply the 1D-FWT again but on each column.
- The image matrix is transposed again to recover the initial data layout.
- Finally, the resulting image is copied back from the global memory buffer to the page-locked main memory buffer.

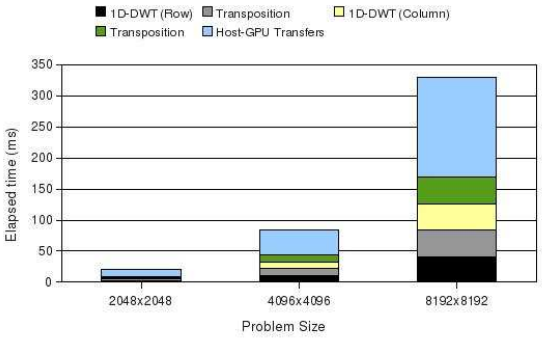
Table 2. Main configuration parameters.

Hardware	
Processor	Intel Core 2 Quad Q6700 (2.66 GHz)
L1 caches (I + D)	64KB + 64KB
L2 data cache	2×4MB
Main Memory	4 GB
GPU Architecture	NVDIA Tesla C870
Software	
OS	OpenSUSE 10.2
`icc`	v10.0
`gcc`	v4.1.2
CUDA	v1.1

In this scenario, two different kernels were identified: 1D-FWT and matrix transposition. Secondly, the parallelization strategy with CUDA for both kernels is explained given an image of dimension `m rows x n columns`. To parallelize the computation of the 1D-FWT for every row of the image, we empirically determined the optimal configuration, using the CUDA occupancy calculator and following a simple set of heuristics [24]. Each thread requires 13 registers and 128-thread blocks need 1032 Bytes of shared memory ((256 + 2) image elements x 4 Bytes per image element). Thus, four active thread blocks per multiprocessor require around 6.6 K registers and 4 KB of shared memory, which does not exceed the maximum allowed values for the NVIDIA Tesla C870. Such a configuration consists of a different thread to compute every pair of G and H values. In this way, we define one-dimensional thread blocks of 128 threads and two-dimensional grids of (`m rows x n/256 columns`) thread blocks. To parallelize the matrix transposition we have used the matrix transpose example from the CUDA development kit.



(a) Execution times using `gcc`, `icc` and CUDA.



(b) Execution time breakdown on Tesla C870.

Figure 3. 2D-FWT execution times using `gcc`, `icc` and CUDA for different images sizes.

4. Performance Evaluation

In this section we present the evaluation results obtained for the parallel 2D-FWT described in Section 3.

4.1. Evaluation environment

All the tests have been executed on a Intel Core 2 Quad processor. Additionally, we have also employed the NVIDIA Tesla C870 for the CUDA version of the 2D-FWT described in Section 3. Consequently, Table 2 summarizes the main hardware and software parameters of our evaluation environment and their corresponding values.

We compare the proposed CUDA version of the 2D-FWT (compiled with the `-O3` flag) with a sequential implementation compiled with both the `gcc` and the `icc` compilers. For the `gcc` compiler we consider both the performance of the binary generated without any compiler flags, and with the `-O3` flag activated. For the `icc` compiler we also consider the latter two cases, although for the second case we also activate the following flags (besides `-O3`):

- `-parallel`: Detects simply structured loops capable of being executed safely in parallel and automatically generates multi-threaded code for these loops.
- `-par-threshold0`: Used in conjunction with `-parallel`, this flag sets a threshold for the auto-parallelization of loops based on the probability of profitable execution of the loop in parallel. In our case, we use the value 0 for the threshold which commands the compiler to parallelize loops regardless of computation work volume.
- `-xT`: Generates specialized code and enables vectorization.

More details regarding these compilation flags can be found in [14]. Additionally, our CUDA version of the 2D-FWT has been compared with a parallel implementation of the algorithm that has been developed using OpenMP. This implementation required a moderate programming effort and takes advantage of all the four cores of the host platform.

4.2. Results

Execution times (in milliseconds) that have been obtained for the configurations previously described are plotted in Figure 3. Results are shown for three image sizes: 2048×2048, 4096×4096, and 8192×8192. In Figure 3(a) we show the execution times reached for the sequential algorithm of the 2D-FWT when the `gcc` and `icc` compilers are used without any flags (*GCC (Host)* and *ICC (Host)* bars), and when the optimization flags already commented are used (*GCCopt (Host)* and *ICCopt (Host)* bars). Additionally, the *ICCOMP(Host)* and *CUDA (Host+Tesla)* bars plot the execution times reached for the OpenMP and CUDA versions of the 2D-FWT proposed in this paper, respectively. A breakdown of the *CUDA (Host+Tesla)* bars is shown in Figure 3(b). In particular, we split each bar into five components: time spent in the application of the 1D-FWT (*1D-FWT (Row)* and *1D-FWT (Column)*), time spent in the transpositions (*Transposition*), and time taken in copying data between main memory and the NVIDIA Tesla (*Host-GPU Transfers*).

As it can be seen, the utilization of the compiler flags have limited effect on final execution time, even when several of them are aimed at parallelizing the loops found in the sequential code (those employed in `icc`). Moreover, the speedups reached for the *GCCopt* and *ICCopt* configurations decrease as the image size grows. On the contrary, important reductions in terms of execution time are obtained for the CUDA version of the 2D-FWT even when we take into account the time needed to copy data and results to and from the NVIDIA Tesla. Compared to the best results for the sequential code (*ICCopt*), the speedups achieved with our CUDA implementation of the 2D-FWT range from 10 (2048×2048 size) to 21.7 (8192×8192 size). In turn, compared to the OpenMP implementation (*ICCOMP*), the speedups range from 9.5 (2048×2048 size) to 20.8 (8192×8192 size). The improvement of the OpenMP implementation over the automatic parallelization provided by `icc` is minimal due to the memory access pattern. In general, the time needed to copy data from the host's main memory to the NVIDIA Tesla's global memory and conversely, represents a large fraction of the total execution time, and keeps almost constant as we change image size (approximately 50% of the elapsed time). Thus, if we do not include these copies to calculate the speedups, we would get speedups ranging from 20 (2048×2048 size) to 40.64 (8192×8192 size).

5. Conclusions and Future Work

CUDA is a new hardware and software architecture for issuing and managing computations on the GPU, without the need of mapping them to a graphics API, common to the latest NVIDIA developments. It promises to simplify the development of applications that take full advantage of current and future powerful GPUs.

In this paper we have presented and evaluated an initial implementation of the 2D fast wavelet transform for CUDA-enabled devices. A brief introduction to the wavelet transform and CUDA has been provided prior to explaining our parallelization strategy. We have compared the proposed CUDA version of the 2D-FWT with a sequential implementation compiled with both the `gcc` compiler and the `icc` compiler, and with a parallel implementation of the algorithm that has been developed using OpenMP. Performance results, obtained on a Intel Core 2 Quad processor with an NVIDIA Tesla C870, indicate that significant speedups can be achieved with a moderate programming effort when compared with automatic parallelization methods (up to 21.7), or an OpenMP parallel implementation that requires a similar development effort (up to 20.8).

Future work involves a more in depth analysis of the parallelization strategy in order to fully exploit all the computing resources provided by these devices, along with an exhaustive comparison with optimized OpenMP and Pthread implementations of the algorithm.

Acknowledgements

The authors would like to thank the anonymous reviewers for their detailed comments and valuable suggestions, which have been helped to improve the quality of the paper. This work has been jointly supported by Spanish MEC under grant "TIN2006-15516-C04-03" and European Comission FEDER funds under grant "Consolider Ingenio-2010 CSD2006-00046".

References

[1] M. Antonini and M. Barlaud. Image Coding Using Wavelet Transform. *IEEE Transactions on Image Processing*, 1(2):205–220, April 1992.

[2] S. Battista, F. Casalino, and C. Lande. MPEG-4: A Multimedia Standard for the Third Millenium, Part 1. *IEEE Multimedia*, 6(4), October 1999.

[3] S. Battista, F. Casalino, and C. Lande. MPEG-4: A Multimedia Standard for the Third Millenium, Part 2. *IEEE Multimedia*, 7(1), January 2000.

[4] G. Bernabé, R. Fernández, J. M. García, M. E. Acacio, and J. González. An Efficient Implementation of a 3D Wavelet Transform Based Encoder on Hyper-Threading Technology. *Journal of Parallel Computing*, 33(1):54–72, February 2007.

[5] G. Bernabé, J. M. García, and J. González. Reducing 3D Wavelet Transform Execution Time Using Blocking and the Streaming SIMD Extensions. *Journal of VLSI Signal Processing*, 41(2):209–223, 2005.

[6] 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.

[7] Y. Chen and W. A. Pearlman. Three-Dimensional Subband Coding of Video Using the Zero-Tree Method. *Proc. of SPIE-Visual Communications and Image Processing*, pages 1302–1310, March 1996.

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

[9] A. García and H. Shen. GPU-Based 3D Wavelet Reconstruction with Tileboarding. *The Visual Computer*, 21(8–10):755–763, September 2005.

[10] N. K. Govindaraju, M. Henson, M. C. Lin, and D. Manocha. Interactive Visibility Ordering of Geometric Primitives in Complex Environments. *Symposium on Interactive 3D Graphics and Games*, pages 49–56, April 2005.

[11] N. K. Govindaraju, B. LLoyd, W. Wang, M. Lin, and D. Manocha. Fast Computation of Database Operations Using Graphics Processors. *ACM SIGMOD International Conference on Management of Data*, pages 215–226, 2004.

[12] T. R. Halffill. Parallel Processing with CUDA. *MicroProcessor Report Online*, January 2008.

[13] M. Harris. *Fast Fluid Dynamics Simulation on the GPU. In GPU Gems*. Addisson Wesley, March 2004.

[14] Intel Corporation. *Intel C++ Compiler Options (Document Number: 307776-002US)*, 2007.

[15] Y. Kim and W. A. Pearlman. Stripe-Based SPIHT Lossy Compression of Volumetric Medical Images for Low Memory Usage and Uniform Reconstruction Quality. *Proceedings of International Conference on Acoustics, Speech and Signal Processing*, 2000.

[16] A. S. Lewis and G. Knowles. Image Compression Using the 2-D Wavelet Transform. *IEEE Transactions on Image Processing*, 1(2):244–256, April 1992.

[17] S. Mallat. A Theory for Multiresolution Signal Decomposition: The Wavelet Representation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 11(7):674–693, July 1989.

[18] M. W. Marcellin, M. J. Gormish, A. Bilgin, and M. P. Boliek. An Overview of JPEG-2000. In *Proceedings of Data Compression Conference*, March 2000.

[19] K. Moreland and E. Angel. The FFT on a GPU. *Graphics Hardware*, pages 112–119, July 2003.

[20] NVIDIA Corporation. *NVIDIA Tesla Computing Solutions for HPC*.

[21] NVIDIA Corporation. *NVIDIA Compute Unified Device Architecture (CUDA) Programming Guide Version 1.1*, November 2007.

[22] NVIDIA Corporation. *NVIDIA CUDA CUBLAS Library Version 1.0*, June 2007.

[23] NVIDIA Corporation. *NVIDIA CUDA CUFFT Library Version 1.1*, October 2007.

[24] NVIDIA Tutorial at PDP'08. *CUDA: A New Architecture for Computing on the GPU*, February 2008.

[25] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell. A Survey of General-Purpose Computation on Graphics Hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[26] P. Sander, N. Tartachuk, and J. L. Mitchell. Explicit Early-Z Culling for Efficient Fluid Flow Simulation and Rendering. *Technical Report, ATI Research Journal*, August 2004.

[27] D. Santa-Cruz and T. Ebrahimi. A Study of JPEG 2000 Still Image Coding Versus Others Standards. In *Proceedings of X European Signal Processing Conference*, September 2000.

[28] J. M. Shapiro. Embedded Image Coding Using Zerotrees of Wavelets Coefficients. *IEEE Transactions on Signal Processing*, 41(12):3445–3462, December 1993.

[29] J. Sptizer. Implementing a CPU-Efficient FFT. *Nvidia Course Presentation, SIGGRAPH*, 2003.

[30] T. Sumanaweera and D. Liu. *Medical Image Reconstruction with the FFT. In GPU Gems*. Addisson Wesley, March 2004.

[31] C. Tenllado, J. Setoain, M. Prieto, L. Pi nuel, and 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):299–310, February 2008.

[32] I. Viola, A. Kanitsar, and M. E. Groller. Hardware-Based Nonlinear Filtering and Segmentation Using High-Level Shading Languages. *IEEE Visualization*, pages 309–316, October 2003.

[33] D. Weiskopf, T. Schafhitzel, and T. Ertl. GPU-Base Bonlinear Ray Tracing. *Computer Graphics Forum*, 23(3):625–633, September 2004.

[34] T. T. Wong, C. S. Leung, P. A. Heng, and J. Wang. Discrete Wavelet Transform on Consumer-Level Graphics Hardware. *IEEE Transactions on Multimedia*, 9(3):668–673, April 2007.

[35] R. Yang and M. Pollefeys. A Versatile Stereo Implementation on Commodity Graphics Hardware. *Real Time Imaging*, 11(1):7–18, February 2005.