Parallel Implementation of the 2D Discrete Wavelet Transform on Graphics Processing Units: Filter Bank versus Lifting

Christian Tenllado, Member, IEEE Computer Society, Javier Setoain, Manuel Prieto, Member, IEEE, Luis Piñuel, Member, IEEE, and Francisco Tirado, Senior Member, IEEE

Abstract—The widespread usage of the discrete wavelet transform (DWT) has motivated the development of fast DWT algorithms and their tuning on all sorts of computer systems. Several studies have compared the performance of the most popular schemes, known as Filter Bank Scheme (FBS) and Lifting Scheme (LS), and have always concluded that LS is the most efficient option. However, there is no such study on streaming processors such as modern Graphics Processing Units (GPUs). Current trends have transformed these devices into powerful stream processors with enough flexibility to perform intensive and complex floating-point calculations. The opportunities opened up by these platforms, as well as the growing popularity of the DWT within the computer graphics field, make a new performance comparison of great practical interest. Our study indicates that FBS outperforms LS in current-generation GPUs. In our experiments, the actual FBS gains range between 10 percent and 140 percent, depending on the problem size and the type and length of the wavelet filter. Moreover, design trends suggest higher gains in future-generation GPUs.

Index Terms—Graphics processors, parallel processing, parallel algorithms, parallel and vector implementations, wavelets and fractals, SIMD processors, optimization, parallel discrete wavelet transform, lifting, filter bank, GPU, stream processors.

1 INTRODUCTION

Custom designs have been extensively used during the last decade in order to meet the computational demand of image and multimedia processing. However, the difficulties that arise in adapting specific designs to the rapid evolution of applications have hastened their decline in favor of other architectures. Programmability is now a key requirement for versatile platforms designed to follow new generations of applications and standards.

At the other extreme of the design spectrum, we find general-purpose architectures. The increasing importance of media applications in desktop computing has promoted the extension of programmable processor cores with multimedia enhancements such as single-instruction, multiple-data (SIMD) instruction sets (the Intel’s MMX/SSE of the Pentium family and the IBM-Motorola’s Altivec are well-known examples). Unfortunately, the cost of delivering instructions to the arithmetic logic units (ALUs) poses a serious bottleneck in these architectures, and they are still unsuited to meeting more stringent multimedia demands.

Graphics Processing Units (GPUs) seem to have taken the best from both worlds. Initially designed as expensive application-specific units with control and communication structures that enable the effective use of many ALUs, they have evolved into highly parallel multipipelined processors with enough flexibility to allow a (limited) programming model. Their numbers are impressive. Today’s fastest GPUs can deliver a peak performance in the order of 360 Gflops, which is more than seven times the performance of the fastest x86 dual-core processor [1]. Moreover, they evolve faster than more specialized platforms because the high-volume game market fuels their development.

Obviously, GPUs are optimized for the demands of 3D scene rendering, which makes software development of other applications a complicated task. Nevertheless, they are general enough to perform computation beyond their specific domain [2]. In fact, their astonishing performance has captured the attention of many researchers in different areas, who are using GPUs to speed up their own applications [3].

In this paper, we have explored the mapping of the discrete wavelet transform (DWT) on modern GPUs, focusing on analyzing and comparing the actual performance of the Filter Bank Scheme (FBS) [4] and Lifting Scheme (LS) [5], which are the most popular choices for implementing the DWT.

From an implementation standpoint, one of the most interesting advantages of the LS is that it requires less arithmetic operations.1 The exact number depends on the type and length of the wavelet filters [6] and, for large filter lengths, theoretically tends to half the number of arithmetic operations involved in the FBS. Actual empirical comparisons between FBS and LS reflect this feature, and LS is seen as the most efficient strategy [7], [8].

However, these arithmetic advantages of LS are at the expense of introducing some data dependencies, which may become a bottleneck on certain parallel platforms. Our
main goal is to explore this trade-off between arithmetic cost and data parallelism on GPUs, and we envision how it will evolve with new developments in the graphics hardware technology. In summary, the focus of this paper is to provide guidance to application designers on the parallel implementation of wavelet-based kernels on GPUs. These guidelines are of great practical interest, given the growing importance of the DWT.

The rest of this paper is organized as follows: Section 2 introduces some related work. Sections 3 and 4 provide background on the DWT and on the GPU architecture and programming model, respectively. In Section 5, we explicitly adapt LS and FBS to this model. Section 6 analyzes the efficiency of both implementations, and finally, Section 7 summarizes the main conclusions and provides some hints on future work.

2 Related Work

The DWT has emerged in recent years as a key part of image and video coding [9], [10], [11] and as a valuable tool for a wide variety of image processing applications [12], [13], [14], [15], [16]. Moreover, we have also witnessed the increasing presence of the DWT in computer graphics [17], which emphasizes the relevance of our study. Driven by its popularity and by the real-time needs of many image processing tasks, significant research work has been done on its efficient implementation on all sorts of computer systems.

In desktop computing, performance problems arise out of the discrepancies between the memory access patterns of the main components of the 2D-DWT (the vertical and the horizontal filtering), especially when large input images are processed, which causes one of these components to exhibit poor data locality in the straightforward implementations of the transform. Consequently, most research has concentrated on cache-aware DWT implementations, which exploit loop transformations and specific data layouts to improve data locality [18], [19], [20]. More recent proposals build on this work and extend it, focusing on the exploitation of multimedia SIMD extensions [21], [22], [23].

Parallel implementations of the DWT for multiprocessor systems are usually based on the general principles of data-domain decomposition; that is, the input images are statically or dynamically partitioned, and the different threads perform the same work on their share of the data [24]. Locality is even more critical in these systems, and performance tuning also focuses on memory hierarchy issues [25], [26], [27].

In embedded systems for wavelet-based image compression, efficiency refers not only to execution time, but also to other design objectives such as hardware budget or power consumption. Given these additional constraints, several authors have shown that LS produces the most efficient designs [8], [28], [29], [30].

Gnavi et al. presented an interesting performance comparison between LS and FBS on a programmable platform (a Texas Instrument DSP) [7]. They assessed the real empirical performance of both schemes in terms of execution speed in the context of the JPEG 2000, concluding that LS was always faster than its FBS counterpart. The actual LS gains heavily depended on the length of the wavelet filters and the number of LS steps, but they were lower than theoretically expected.

Focusing on general-purpose computing on GPUs (GPGPU), most of the research activities work toward finding efficient strategies for algorithm mapping on these platforms. Generally speaking, it involves developing new implementation strategies following a stream programming model, in which the available data parallelism is explicitly uncovered so that it can be exploited by the hardware. This programming challenge has been studied by several researchers, who have successfully ported a large number of scientific applications [3]. In particular, the image and video processing community is becoming very active in this context [31], [32], [33], [34], [35], [36], [37], [38], [39], [40].

The most relevant contributions in our context are [32], [33], [34]. In [33], the authors proposed the first implementation of the 2D-DWT on graphics hardware. They developed an OpenGL-based model of the FBS algorithm on a Silicon Graphics workstation by using high-level OpenGL routines (OpenGL convolution filters), with no direct mapping on hardware, which limits the efficiency of their implementation.

Moreland and Angel described the first implementation of the 2D-FFT on GPUs [32]. They used a lower level model, in which all the FFT computations were directly related to texture mapping operations performed on the GPU’s fragment processors (FP). This strategy allowed them to exploit all the programmable capabilities of the FPs and develop much more sophisticated implementations than simply using the OpenGL convolution filters.

Wang et al. [34] developed a new GPU implementation of the FBS that used some of the ideas introduced in [32] and integrated it on Jasper, one of the reference implementations of the JEG 2000. This implementation was based on position-dependent filtering and convolution by means of predicated execution and indirect texture lookup with computed texture coordinates. This approach does not efficiently exploit all the hardware resources available in current GPUs such as the hardware interpolators of texture coordinates. Furthermore, predicated execution limits the effective performance of the GPU’s FPs.

In [41], we proposed an alternative mapping of FBS, which overcame the limitations mentioned above, and introduced some hints for implementing LS. In this paper, we extend this preliminary work with more elaborate and efficient implementations based on OpenGL Framebuffer Objects. Furthermore, we provide a comparative study of both schemes for several wavelet families and a new performance assessment using a more recent GPU, which allows us to envision future trends.

3 Discrete Wavelet Transform Algorithms

DWT has been traditionally implemented using two different algorithms, usually known as FBS and LS. Their respective one-dimensional (1D) versions are described in Sections 3.1 and 3.2.

The 1D-DWT produces a pyramid decomposition of a given input signal into different resolution bands. Each level generates a pair of Approximation and Detail signals from the Approximation band of the previous level. As their names suggest, Approximation is a coarse-grained representation of its predecessor, and Detail contains the high-frequency details that have been removed. Both of them have half the resolution of their predecessor.

The 2D-DWT is usually obtained by applying a separate 1D transform along each dimension. The most common
approach, known as the square decomposition, alternates between computations on image rows and columns. This process is applied recursively to the quadrant containing the coarse scale approximation in both directions. This way, the data on which computations are performed is reduced to a quarter in each step (see Fig. 1a).

3.1 Filter Bank Algorithm

The FBS, which is the original Fast Wavelet Transform algorithm proposed by Mallat [4], uses a pair of Quadrature Mirror Filters.

As Fig. 1b illustrates, the discrete signal is convolved with a low-pass (LP) filter \(H(z)\) and a high-pass (HP) filter \(G(z)\), and their outputs are downsampled to obtain the Approximation (low band, LP in Fig. 1b) and Detail signals (high band, HP in Fig. 1b).

The full decomposition is obtained by iterating this process on the Approximation signal. The mathematical definition of one stage of this decomposition process for the CDF(9, 7) biorthogonal wavelet family, applied to a signal \(x\), is shown as follows:

\[
\begin{align*}
LP_n &= \sum_{k=-4}^{4} h_k x_{2n+k}, \\
BP_n &= \sum_{k=-2}^{4} g_k x_{2n+k},
\end{align*}
\]

with \(n\) ranging from 0 to \(\frac{N-1}{2}\), where \(N\) is the number of signal coefficients.

3.2 Lifting Scheme

In the early 1990s, Sweldens demonstrated that an equivalent definition of the wavelet decomposition can be obtained in terms of prediction and update steps [5] (generally referred to as LS steps). By concatenating several groups of these predict and update steps, it is possible to obtain equivalent transformations for the different wavelet families. Daubechies and Sweldens further showed that every FBS can be easily factorized into several LS steps by using the Polyphase Matrix representation of the filters [6].

From a mere implementation standpoint, the main advantage of LS is that it exploits the redundancy between the HP and LP filters, allowing a reduction in the number of arithmetic operations. Fig. 1c shows a block diagram of the forward wavelet transform using LS.

The mathematical definition of the LS that results from the Polyphase matrix factorization for the CDF(9, 7) biorthogonal wavelet family is shown as follows:

\[
\begin{align*}
sl_0 &= x_{2l}, \\
sl_1 &= x_{2l+1}, \\
dl_0 &= d_{l-1}, \\
dl_1 &= d_{l-1} + \alpha (sl_0 + sl_1), \\
su_0 &= sl_0 + \beta dl_0, \\
su_1 &= su_0 + dl_0, \\
du_0 &= du_1 + \gamma (sl_1 + sl_0), \\
du_1 &= du_0 + dl_1.
\end{align*}
\]

The LS is traditionally considered as the most efficient alternative, since LS and FBS are usually compared in terms of their arithmetic cost, that is, the number of multiplications and additions involved in both schemes. Daubechies and Sweldens [6] showed that the FBS tends, asymptotically for long filters, to require up to twice the number of operations needed by its LS counterpart. However, this arithmetic advantage does not guarantee that LS is the most efficient scheme. The actual performance also depends on other algorithmic features such as the inherent data parallelism, the memory access behavior or the target computing platform.

4 GPU Computing Model

In this section, we provide background on the GPU architecture and programming model. Section 4.1 outlines a high-level description of the traditional rendering pipeline, whereas Section 4.2 illustrates how an algorithm can be mapped to the GPU using a stream programming model.

4.1 The Graphics Pipeline

Modern GPUs such as the latest Nvidia GeForce or ATI Radeon cards, implement a generalization of the traditional rendering pipeline for 3D computer graphics [42], [43]. The inputs to this pipeline are vertices from a 3D polygonal mesh, with attached information such as their colors and texture coordinates, plus a virtual camera viewpoint, and the output is a 2D array of pixels to be displayed on the screen. It consists of several stages, but the bulk of the work is performed in three of them: vertex processing, rasterization, and fragment processing.

In the vertex stage, the 3D coordinates of the input vertices are transformed (projected) onto a 2D screen position, applying lighting as well to determine their colors. Once transformed, vertices are grouped into rendering primitives, such as triangles, which in turn are scan converted by the rasterizer into a stream of pixel fragments. These fragments are discrete portions of the triangle surface that correspond to the pixels of the rendered image. In the fragment stage, the color of each fragment is computed
In the last stages of the rendering pipeline, the output from the fragment processing stage is combined with the existing data stored at the associated 2D locations in the color buffer (frame buffer) to produce the final colors.

Until only a few years ago, commercial GPUs were implemented using a fixed-function rendering pipeline. However, most GPUs today include fully programmable Vertex and Fragment stages. The programs that they execute are usually called vertex and fragment programs (or shaders), respectively, and can be written using C-like high-level languages such as Cg [44]. This is the feature that allows for the implementation of nongraphics applications on the GPUs. Parallelism is exploited at different levels within this graphics pipeline:

- **Deep pipelining.** The actual hardware of a modern GPU has hundreds of stages to exploit functional parallelism and increase the throughput and the GPU’s clock frequency.

- **Data parallelism.** GPUs also incorporate replicated stages to take advantage of the inherent data parallelism of the rendering process. For instance, the vertex and fragment processing stages include several replicated units known as vertex processor (VP) and fragment processor (FP), respectively. The overall idea is that the GPU launches a thread per incoming vertex (or per group of fragments), which is dispatched to an idle processor [43], [45].

- **Multithreading.** The VP and FP processors access the memory via a texture cache, which captures 2D locality [46]. Furthermore, they exploit multithreading to hide memory accesses; that is, they support multiple in-flight threads.

- **Instruction-level parallelism.** Apart from multithreading, VPs and FPs can execute independent shader instructions in parallel as well. For instance, these processors have often supported short-vector instructions that operate on four-element vectors (Red/Green/Blue/Alpha (RGBA) channels) in SIMD fashion [43].

In our target area, that is, image processing applications, most GPU implementations take advantage of the FPs, since they typically fit well with the programming model that they offer. Furthermore, FPs usually outperform the VPs, both in number and memory access performance. The following section focuses on this programming environment.

### 4.2 Programming Model

For nongraphics applications, the GPU can be better thought of as a stream coprocessor that performs computations through the use of streams and kernels. A stream is just an ordered collection of elements requiring similar processing. A kernel is a data-parallel function that processes input streams and produces new output streams; that is, its outcome must not depend on the order in which output elements are produced, which forces programmers to explicitly expose data parallelism to the hardware.

As mentioned above, in our target area, kernels are expressed as fragment programs, and data streams are abstracted as textures [47]. The fragment programs are coded using a high-level shading programming language such as Cg. However, we must still use a 3D graphics API such as OpenGL to organize data into streams, transfer those data streams to and from the GPU as 2D textures, upload kernels, and perform the sequence of kernel calls dictated by the application flow. In order to illustrate these concepts, Fig. 2 shows some of the OpenGL commands and the Cg code to perform a simple example that adds some matrices. The implementation consists of two kernels (fragment programs) denoted as $fp1$ and $fp2$, respectively. $fp1$ adds $A$ and $B$ and stores the result in $C$, whereas $fp2$ consumes $C$ to update a large portion of the original matrix $A$.

Basically, the main program does the following:

```
for( i=0; i < ROWS; i++)
for( j=0; jj=0; j < COLUMNS; jj+=2 .jj++)
B[i*COLUMNS + jj] = 0.0;
B[i*COLUMNS + jj + COLUMNS/2] = 0.0;
```

```
for( k=0; k < FILTER_H; k++)
B[i*COLUMNS + jj] +=
(k) A[i*COLUMNS + j + k - FILTER_H/2];
```

```
for( k=0; k < FILTER_G; k++)
B[i*COLUMNS + jj + COLUMNS/2] +=
  (k) A[i*COLUMNS + j + k - FILTER_G/2];
```
Fig. 2). The input and output streams of these kernels are read from and written into the active texture. Technically, it involves mapping the color buffer to a specific region of the active texture (this is known as render to texture). Delimit the output and input areas of the active texture. This definition is performed by drawing a rectangle and is equivalent to specifying the loop boundaries for the outer parallel loops. The vertex coordinates of this rectangle delimit the output area (matrix C in the first kernel and a portion of matrix A in the second one), whereas the associated texture coordinates define the corresponding reading areas (matrices A and B in the first kernel and a portion of matrices C and A in the second one). These texture coordinates are hardware interpolated to obtain the coordinates of the input elements associated with each output fragment. These coordinates are then used for fetching the actual input elements by means of texture lookups.

• Force synchronization to avoid race conditions. As shown in Fig. 2, it is quite frequent that a given kernel consumes data from a previous one (fp2 consumes the result of fp1). If needed, the synchronization between consumers and producers is performed using the OpenGL glFinish() command, which does not return until the effects of all previously called GL commands are completed.

5 2D-DWT on GPUs

In this section, we describe the stream implementation of the two algorithms under study by using the programming model described in Section 4.2.

The first design decision concerns the mapping of a 2D gray-scale image onto a 2D texture, constituted by four-float elements referred to as texels. Given the real nature of the DWT, each pixel of the gray-scale image is converted to a 32-bit floating-point value. These pixels are arranged in groups of four using a 2D layout; that is, two pixels from two consecutive rows of the original array are packed into a single RGBA texel. This is the most intuitive mapping, since it allows for a symmetric design of the algorithms. The main program of our implementation carries out the required data reordering to perform this mapping and transfers the input image into the GPU memory by using a texture that is twice its size. The bottom half of this texture is used for saving intermediate results.

The next step in the design process consists of performing an efficient partitioning of the target algorithms. Here, we decide which part of the work will be done in the FP (define the kernels) and which part will be left to the CPU. Load balancing between the CPU and the GPU is a key performance factor. One common option is to leave the parallel sections of the code to the GPU and perform the other parts of the application such as the control flow on the CPU. Obviously, this partitioning process heavily depends on the target algorithm. In our implementation, we have assumed that the DWT is part of a more complex application, as this is often the case. Our goal is then to perform the whole transformation on the GPU in order to free up the CPU for other processing tasks of the application. Under this assumption, the partitioning should enlarge the size of the streams to maximize the granularity of the parallelism and leave only the flow-control activity to the CPU.

The results of this partitioning process is a chain of kernels that produces, for each position of the output texture, the corresponding wavelet coefficient. Each kernel can read other parts of the texture (through interpolated texture coordinates passed to the kernel as input parameters) to get all the necessary information to compute the wavelet coefficients.

5.1 Filter Bank Algorithm

The pseudocode in Fig. 3 shows a generic implementation of the horizontal DWT using the FBS. The inner loops perform the LP and HP filters, respectively. The index variable of the central loop $j$ is incremented by two on every iteration to perform downsampling. Notice also that the approximation and detail wavelet coefficients, that is, the output from the LP and HP filters, are stored on the left and right halves of the destination array, respectively (see Fig. 3).

Fig. 4a graphically illustrates a convenient partitioning of this loop (Fig. 4b shows the analogous partitioning for the vertical DWT). Basically, we extract the different kernels by finding different convex regions of the destination array that are produced the same way. Since approximations and
details have to be stored on different halves of the destination array, we distinguish two classes of kernels: LP and HP kernels (denoted as \(H\) and \(G\) in Fig. 4a). Each class is further split into three different kernels, denoted by the prefixes LEFT, CENTRAL, and RIGHT. Both the LEFT and RIGHT programs are designed to perform mirror extensions on the boundaries of the image.

The flow control is performed by the CPU, as in the simple example in Fig. 2. The main program then launches the different kernels and forces synchronization between the horizontal and the vertical filtering. The horizontal kernels read from the top half of the allocated texture and write into the bottom half. Downsampling is performed by specifying input areas that are twice as large as the output ones.

Fig. 5a describes the internals of the horizontal \(H_{\text{CENTRAL}}\) kernel (the other kernels are similar). Each kernel thread produces four approximation coefficients packed into a single texel using a 2D layout. To load the input elements necessary for computing these four DWT coefficients, texture lookup is performed using the interpolated texture coordinates given as input to the fragment program. The lookup method used is GL_NEAREST. Fig. 5b shows how the input texture coordinates are generated. The main program in the CPU selects different spatial regions of the original image, which are shifted versions of the central one. The texture interpolation hardware of the GPU interpolates the texture coordinates that delimit these regions, providing the six texture coordinates of the texels. In general, the number of texels needed depends on the filter length and the data layout chosen. Since the elements of the original array are also packed using a 2D layout, an initial data rearrangement step is needed to fully exploit the SIMD capabilities of the FPs.

All the DWT horizontal kernels can be performed in parallel, since they write (shade) on separate regions of the memory (texture). Once the image is horizontally filtered, the vertical filtering can be applied, which, as shown in Fig. 4b, is analogous to the horizontal filtering (instead of splitting the image into left-hand and right-hand boundaries and inner columns, the image is divided into upper and lower boundaries and inner rows). Since the vertical DWT kernels consume the output from the horizontal DWT kernels, as mentioned above, the main program introduces a synchronization barrier in between them.

5.2 Lifting Scheme

In general-purpose architectures and DSPs, the LS steps are usually fused into a single loop to optimize data locality. However, the inner loop of such implementations exhibits loop-carried dependencies that inhibit its parallel implementation on a GPU.

A more natural and efficient stream-based LS implementation can be obtained instead by using loop fission. Fig. 6 shows this transformation for the CDF(9, 7) wavelet

![Fig. 6. Loop structure for the LS with separate loops for each stage.](image)

```c
for(i=0;i<ROWS;i++)
for(j=0;j<COLS/2;j++)
{ A[i][j]=A[i][j*2];
  A[i][j+COLS/2]=A[i][j*2+1];
}
```

```c
for(i=0;i<ROWS;i++)
for(j=COLS/2;j<COLS-1;j++)
}
```

```c
for(i=0;i<ROWS;i++)
for(j=1;j<COLS/2;j++)
}
```

```c
for(i=0;i<ROWS;i++)
for(j=COLS/2;j<COLS-1;j++)
```

```c
for(i=0;i<ROWS;i++)
for(j=1;j<COLS/2;j++)
}
```

```c
for(i=0;i<ROWS;i++)
for(j=COLS/2;j<COLS-1;j++)
```

```c
for(i=0;i<ROWS;i++)
for(j=1;j<COLS/2;j++)
```

Fig. 6. Loop structure for the LS with separate loops for each stage. Specific processing on the array boundaries is not shown.
family. The basic idea is that every LS step is performed by a different kernel, and the CPU main program chains and serializes these kernels to satisfy data dependencies.

Fig. 7a illustrates the actual kernel flow of our implementation. The first two kernels perform the lazy wavelet transform; that is, they reorder the even and odd elements of the input image and store them separately in the bottom half. The subsequent kernels apply the LS steps and normalize the wavelet coefficients in place; that is, they read from the bottom half of the texture and also write into that area. As in the FBS implementation, each of these kernels has to be specialized on the boundaries. Fig. 7b graphically describes one of these kernels, which specifically performs the horizontal $\alpha$-step of the CDF(9, 7) family, but which is also similar to the other LS steps of the CDF(9, 7) or even other wavelet families. The fragment programs that implement these kernels are simpler, and each one requires less texture accesses than the FBS counterparts. That is, the total number of memory accesses is the same, but in FBS, only two fragment programs perform all the memory accesses, whereas in the LS case, these accesses are performed by six simple fragment programs. The overall arithmetic cost is also lower. However, this implementation requires additional synchronization barriers to satisfy data dependencies and more rendering passes (the whole LS flow has less temporal locality), which may degrade performance.

6 PERFORMANCE EVALUATION

Several factors can potentially influence the relative performance of the two DWT implementations under study: the filter lengths, the number of LS steps, the target image size, and the underlying architecture. We have tried to analyze the impact of these factors by using different GPUs and exploring a wide range of wavelet families.

6.1 GPU Architectures

As experimental platforms, we have used two different GPUs—the Nvidia FX 5950 Ultra and the Nvidia 7800 GTX—which represent an evolution of two years in the graphics hardware technology. The main characteristics of these cards are summarized in Table 1. The most significant difference between them is the number of FPs. Following a general trend in graphics hardware, this number has grown substantially. A comparison between them allows us to envision how well LS and FBS will exploit future GPU generations.

6.2 Wavelet Families

We have explored five wavelet families that exhibit different arithmetic cost/LS steps ratios. Table 2 summarizes their main characteristics in terms of the number of elements of the input image ($N^2$). Their respective LS factorizations are described in [6] and [7]. Both schemes always have the same complexity order $O(n)$, but LS requires less operations, which makes this scheme the most appropriate choice in most general-purpose systems. Nevertheless, the actual gains from LS are always smaller than these theoretical values [7]. For instance, on Intel’s x86 processors, our hand-tuned implementation of LS is only around 20 percent more efficient than its FBS counterpart for the CDF(9, 7) family [21], [48].

6.3 Performance Analysis: FBS versus LS

Table 3 compares the performance of FBS and LS by using different image sizes. The execution times shown correspond to the processing of a single color image using a five-level wavelet decomposition.

7. $N$ is the number of gray-scale pixels in the image, that is, the number of rows multiplied by the number of columns. In the case of color images, it is the number of pixels multiplied by the number of channels.

8. Our CPU implementations of LS and FBS both include aggressive optimizations to improve data locality and intensively exploit Intel’s SIMD extensions [21], [48].
The $LS/FBS$ ratio is always smaller than theoretically expected. In fact, FBS is more efficient than the LS counterpart in most cases ($LS/FBS > 1$), especially on the Nvidia 7800 GTX. In this GPU, LS only outperforms FBS for large images using the SWE(13,7) family, which is the most favorable of the five analyzed families for LS: the $LS/FBS$ arithmetic cost ratio is similar to the other families, but LS only requires four LS steps.

The execution time of both schemes scales linearly with the image size, which emphasizes the ability of the GPU to perform the 2D-DWT. Therefore, the $LS/FBS$ ratio (see Fig. 8) tends asymptotically to a constant value:

$$\lim_{\text{Size} \to \infty} \frac{T_{LS}}{T_{FBS}} = \lim_{\text{Size} \to \infty} \frac{\alpha_{LS} \text{Size} + \beta_{LS}}{\alpha_{FBS} \text{Size} + \beta_{FBS}} = \frac{\alpha_{LS}}{\alpha_{FBS}}.$$  

As shown in Fig. 9, the execution time of both schemes improves significantly in the 7800 GTX (apart from the smallest image sizes using LS, for two of the wavelet families considered). However, these performance gains are quantitatively different. Asymptotically, the speedup factor is around 3.7-4.3 times for FBS, but only around 2.1-2.4 times for LS; that is, the reductions in their respective slopes ($\alpha_{FBS}$ and $\alpha_{LS}$) are higher for the FBS.

This behavior is a consequence of the differences in data locality and degree of parallelism between both schemes. The GPU-based implementation of LS has a lower arithmetic cost than the FBS counterpart. However, the key performance factor is not this but the number of rendering passes and synchronization barriers. FBS is superior to LS in exploiting the temporal locality of the DWT. The long pipeline of a modern GPU hides memory latencies and mitigates locality problems. However, synchronization barriers cause a pipeline flush, and hence, FBS is more effective in exploiting this capability. We have measured the speedup obtained on the LS implementation by removing these barriers for the (9, 7) wavelet family. The speedup on the 7800 GTX ranges from 1.09 to 1.78, depending on the image size (from 1.02 to 1.38 on the FX 5950 Ultra). Obviously, the output from this implementation is incorrect; however, it serves as an estimation of the impact that these barriers have on the performance.

### 6.4 Performance Analysis: GPU versus CPU

To put these results into perspective, we conclude our analysis by assessing the relative performance between modern GPU and CPU implementations. As a CPU reference to the Nvidia 7800 GTX, we have used a contemporary Intel’s Prescott processor (3.4 GHz, 2-Gbyte RAM, 800-MHz front-side buffer (FSB), 2-Mbyte level-2 (L2) cache). The GPU implementations were developed using the Intel C/C++ compiler and include aggressive optimizations to improve data locality and exploit Intel’s SIMD extensions [21].

Table 4 shows the execution times of the best 2D-DWT implementations on both platforms (FBS on the GPU and LS on the CPU) using the CDF(9, 7) wavelet family (similar results are obtained for the other families). Ignoring CPU-GPU data transfer times, the GPU implementation clearly outperforms the CPU counterpart. Furthermore, it scales better with image size (see Fig. 10).

Finally, we should highlight that data transfers between the CPU and the GPU are currently a major bottleneck. Therefore, only those non-graphics applications that can hide these transfers with useful computation will benefit from the GPU. However, at some point in the future, we might see something like a GPU integrated on the same die as a multicore processor, and the GPU might serve as a coprocessor to speed up data parallel algorithms such as the 2D-DWT.

### 7 Conclusions and Future Research

In this paper, we have explored the implementation of the 2D-DWT on modern GPUs. We have analyzed the performance of both FBS and LS in order to determine which algorithm is more suitable for this kind of platform. The main contributions of this paper can be summarized as follows:

- Even though the LS has been shown to be the most efficient on general-purpose architectures, our experiments indicate that this is not always the case for current state-of-the-art GPUs, on which the FBS beats LS performance in most cases.
- The execution times of our implementations scale linearly with problem size, which makes the LS/FBS execution time ratio tend to a constant value when the problem size grows. Our experiments suggest that this ratio grows as the number of FPs (shader processors) increases. Taking into account the

$9$. For instance, AMD has recently announced the CPU/GPU “Fusion Project” [49].
expected evolution of graphics hardware, this kind of system will progressively favor FBS.

- Ignoring CPU-GPU data transfers, our GPU implementations have shown better performance than a highly tuned CPU implementation (speedup factors of 1.2-3.4 times). Although the overall GPU performance is significantly impaired by these GPU-CPU data transfers, a tighter CPU/GPU integration promises to mitigate these bottlenecks [49].

- Results also suggest that fusing several consecutive kernels might significantly speed up the execution, even if the complexity of the resulting fused fragment program is higher. This kind of transformation is similar to *loop fusion*, a well-understood optimization performed by compilers to improve temporal locality and instruction level parallelism within basic blocks.

We plan to investigate how to translate and adjust these techniques within the GPU context and develop...
automatic methodologies for guiding loop fusion and distribution transformations. CPU/GPU integration will also open the door to exploring new application partitioning methodologies.

ACKNOWLEDGMENTS

This work has been supported by the research contracts CICYT-TIN 2005/5619 and Ingenio 2010 Consolider CSD2007-00050.

REFERENCES


Luis Piñuel received the PhD degree in computer science from the Universidad Complutense de Madrid (UCM). He is an associate professor in the Department of Computer Architecture, UCM. His research interests include computer system architecture and processor microarchitecture, with emphases on adaptive architecture, power and temperature-aware computing, and code optimization. He is a member of the IEEE and the IEEE Computer Society.

Francisco Tirado received the bachelor’s degree in applied physics and the PhD degree in physics from the Universidad Complutense de Madrid (UCM) in 1973 and 1977, respectively. He has held several positions with the Computer Science and Automatic Control Department, UCM. Since 1978, he has been an associate professor, and since 1986, he has been a professor of computer architecture and technology. He has worked on different fields within computer architecture, parallel processing, and design automation. His current research areas are parallel algorithms and architectures, and processor design. He is a coauthor of more than 200 publications: 15 book chapters, 47 magazine articles, and 133 conference proceedings. He has served in the organization of more than 60 international conferences as the general chair, a member of steering committee, the program chair, a member of program committee, an invited speaker, and the session chair. He is the director of the Center for SuperComputation (CSC) and Madrid Science Park. From 1994 to 2002, he was the dean of the Physics Science and Electronic Engineering Faculty. He is member of the Informatics Advisory Board of the UCM, and he was also the vice dean of the Physics Science Faculty and head of the Computer Science and Automatic Control Department. From 1988 to 1992, he was the general manager of the Spanish National Program for Robotics and Advanced Automation. He is the adviser of the National Agency for Research and Development (CICYT). He served the research evaluation committee in Spain for three years and chaired it from 2001 to 2002. He is a senior member of the IEEE, the IEEE Computer Society, and of several European institutions and committees.