Pre-print of: Enfedaque, P., Aulí, F. and Moure, JC. "Implementation of the DWT in a GPU through a register-based strategy" in IEEE transactions on parallel and distributed systems, vol. 26, issue 12 (December 2015), p. 3394-3406. DOI 10.1109/TPDS.2014.2384047

Implementation of the DWT in a GPU through a Register-based Strategy

Pablo Enfedaque, Francesc Aulí-Llinàs, *Senior Member, IEEE*, and Juan C. Moure

***Abstract*—The release of the CUDA Kepler architecture in March 2012 has provided Nvidia GPUs with a larger register memory space and instructions for the communication of registers among threads. This facilitates a new programming strategy that utilizes registers for data sharing and reusing in detriment of the shared memory. Such a programming strategy can significantly improve the performance of applications that reuse data heavily. This paper presents a register-based implementation of the Discrete Wavelet Transform (DWT), the prevailing data decorrelation technique in the field of image coding. Experimental results indicate that the proposed method is, at least, four times faster than the best GPU implementation of the DWT found in the literature. Furthermore, theoretical analysis coincide with experimental tests in proving that the execution times achieved by the proposed implementation are close to the GPU's performance limits.**

***Index Terms*—Discrete Wavelet Transform (DWT), Graphics Processing Unit (GPU), Compute Unified Device Architecture (CUDA).**

I. Introduction

Over the last ten years, the computational power of Graphics Processing Units (GPUs) has grown notably. Once devised to alleviate the Central Processing Unit (CPU) from the computational burden imposed by the rendering of graphics in computer-aided design or video games, GPUs are nowadays employed for computation tasks in mainstream applications as well. The evolution of GPUs has undergone major revisions. Arguably, the most relevant was the release in November 2006 of the Nvidia Compute Unified Device Architecture (CUDA), which provided tools for general-purpose computing together with the first C compiler for the GPU.

Nowadays, the computational power of GPUs surpasses that of CPUs. The reason behind the GPUs' great improvement lies in their innermost architectural principle. CPUs are mainly based on the Multiple Instruction, Multiple Data (MIMD) principle. They are structured in cores. Each can process a flow of instructions on a piece of data *independently and asynchronously* from the others. GPUs, on the other hand, are mainly based on the Single Instruction, Multiple Data (SIMD) principle. They are devised so that one flow of instructions is applied to different pieces of data *in parallel and synchronously*. This simplifies the chip design, permitting the allocation of more resources to sustain parallel threads. Modern GPUs can execute peaks of almost 30,000 threads, as opposed to the tens of threads that the best CPUs can execute. In addition to provide high computational power, GPUs are more cost and power efficient than CPUs, which make them an ideal choice for a large variety of applications. Both the CPUs and the GPUs are *not* only based on a single architectural principle, but they combine MIMD and SIMD in different ways. As described below, the GPU can be programmed to execute different flows of instructions in a MIMD fashion, though each flow of instructions is applied on multiple pieces of data as dictated by the SIMD principle.

Implementations in GPUs must be carefully realized to fully exploit the potential of the device. Data management is a fundamental aspect. The key is to store the data in the appropriate memory spaces. From a general perspective, the GPU has three main spaces: global, shared, and registers. The largest is the global memory, which is located in the off-chip DRAM and has very high latency. The shared memory and the registers are located on-chip and can be explicitly managed. They are significantly faster than the global memory, but their size is much smaller. The difference between the shared memory and the registers is that the shared memory is commonly employed to store and reuse intermediate results and to efficiently share data among threads. The registers are private to each thread and they are employed to perform the arithmetic and logical operations of the program.

The CUDA performance guidelines [1] recommend the use of the shared memory for data reuse and data sharing. Surprisingly, these recommendations were challenged in November 2008 by Volkov and Demmel [2], [3], who stated that an extensive use of the shared memory may lead to suboptimal performance. This is caused by a combination of three factors. The first is the bandwidth of the shared memory that, despite being very high, may become a bottleneck in applications that need to reuse data heavily. The second is that arithmetic or logical operations carried out with data located in the shared memory implicitly need to move these data to the registers before performing the operations, which requires more instructions. And the third is that the register memory space is commonly larger than that of the shared memory. In their paper, Volkov and Demmel indicated that the only way to increase the GPU's performance is to directly use the registers, minimizing the use of the shared memory. Their results suggest that maximum performance is achieved when

Pablo Enfedaque and Francesc Aulí-Llinàs are with the Department of Information and Communications Engineering, Universitat Autònoma de Barcelona, Spain (phone: +34 935811861; fax: +34 935813443; e-mail: {pablo | fauli}@deic.uab.cat). Juan C. Moure is with the Department of Computer Architecture and Operating Systems, Universitat Autònoma de Barcelona, Spain (e-mail: juancarlos.moure@uab.es). This work has been partially supported by the Spanish Government (MINECO), by FEDER, and by the Catalan Government, under Grants UAB-472-02-2/2012, RYC-2010-05671, TIN2012-38102-C03-03, TIN2011-28689-C02-1, and 2014SGR-691.

the registers are employed as the main local storage space for data reusing, though the results may vary depending on the algorithm. At that time, there was no operations to share the data in the registers among threads, and the register memory space was very limited. This restrained the use of register-based implementations.

The release of the Kepler CUDA architecture in March 2012 unlocked these restrictions. In Kepler, the size of the register memory space has been doubled, the number of registers that each thread can manage has been quadruplicated, and a new set of instructions for data sharing in the register space has been introduced. These improvements facilitate register-based strategies to program the GPU. This is an emerging approach that can significantly enhance the performance achieved. In the fields of image processing and computational biology, for example, this trend has already been employed to achieve good results [4]–[6].

This paper explores the use of a register-based strategy to implement the Discrete Wavelet Transform (DWT). The DWT is a prevalent data decorrelation technique in the field of image and video coding. It is employed in international compression standards such as JPEG2000 [7], [8] or CCSDS-ILDC [9] as well as in numerous widespread coding schemes such as SPIHT [10], EBCOT [11], or SPECK [12]. Realizations of the DWT in GPUs require carefully implemented strategies of data reuse. As seen below, there are many different approaches in the literature. The use of a register-based strategy allows a particular approach that differs from the state-of-the-art methods. The implementation has to be rethought from the scratch. The most critical aspects are data partitioning and thread-to-data mapping (see below). To the best of our knowledge, this is the first implementation of the DWT employing a register-based strategy. The proposed method achieves speedups of 4 compared to the best method found in the literature.

The paper is structured as follows. Section II provides a general description of the DWT, Section III overviews the CUDA architecture, and Section IV reviews state-of-the-art implementations of the DWT in GPUs. Section V describes the method proposed detailing the data partitioning scheme employed and its implementation in the GPU. Section VI assesses the performance of the proposed implementation through extensive experimental results. The last section summarizes this work.

II. Review of the DWT

The DWT is a signal processing technique derived from the analysis of Fourier. It applies a bank of filters to an input signal that decompose its low and high frequencies. In image coding, the forward operation of the DWT is applied to the original samples (pixels) of an image in the first stage of the encoding procedure. In general, coding systems use a dyadic decomposition of the DWT that produces a multi-resolution representation of the image [13]. This representation organizes the wavelet coefficients in different levels of resolution and subbands that capture the vertical, horizontal, and diagonal features of the image. The decoder applies the reverse DWT in the last stage of the decoding procedure, reconstructing the image samples.

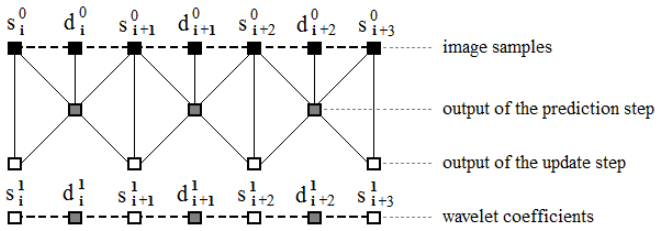


Fig. 1: Illustration of the lifting scheme for the forward application of the reversible CDF 5/3 transform.

The filter bank employed determines some features of the transform. The most common filter banks in image coding are the irreversible CDF 9/7 and the reversible CDF 5/3 [14], which are employed for lossy and progressive lossy-to-lossless compression, respectively. The proposed method implements these two filter banks since they are supported in JPEG2000, though other banks could also be employed achieving similar results.

The DWT can be implemented via a convolution operation [13] or by means of the lifting scheme [15]. The lifting scheme is an optimized realization of the transform that reduces the memory usage and the number of operations performed, so it is more commonly employed. It carries out several steps in a discretely-sampled one-dimensional signal, commonly represented by an array. Each step computes the (intermediate) wavelet coefficients that are assigned to the even, or to the odd, positions of the array. Each coefficient is computed using three samples: that in the even (or odd) position of the array, and its two adjacent neighbors. Such a procedure can be repeated several times depending on the filter bank employed. An important aspect of the lifting scheme is that all coefficients in the even (or odd) positions can be computed in parallel since they do not hold dependencies among them.

Formally expressed, the lifting scheme is applied as follows. Let $\{c_i\}$ with $0 \leq i < I$ be the original set of image samples. First, c_i is split into two subsets that contain the even and the odd samples, referred to as $\{d_i^0\}$ and $\{s_i^0\}$, respectively, with $0 \leq i < I/2$. The arithmetic operations are performed in the so-called prediction and update steps. As seen in Fig. 1, the prediction step generates the subset $\{d_i^1\}$ by applying to each sample in $\{d_i^0\}$ an arithmetic operation that involves d_i^0, s_i^0, and s_{i+1}^0. This operation is generally expressed as

$$d_i^{j+1} = d_i^j - \alpha^j (s_i^j + s_{i+1}^j) \ . \tag{1}$$

The update step is performed similarly, producing the subset $\{s_i^{j+1}\}$ that is computed according to

$$s_i^{j+1} = s_i^j + \beta^j (d_i^{j+1} + d_{i+1}^{j+1}) \ . \tag{2}$$

Depending on the wavelet filter bank, (1) and (2) may be repeated several times. The result of these steps are the subsets $\{d_i^J\}$ and $\{s_i^J\}$, which contain the low and high frequencies of the original signal, respectively, with J denoting the number of iterations performed. α^j and β^j in the above equations depend on the filter bank and change in each step j. The 5/3

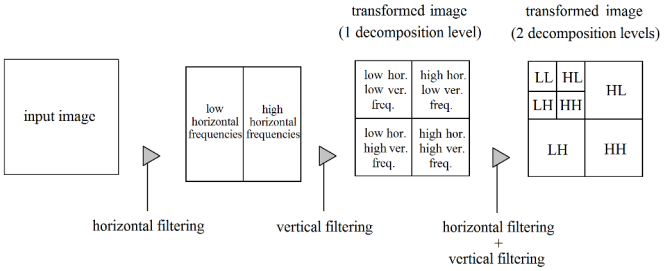


Fig. 2: Application of two levels of DWT decomposition to an image.

transform has $J = 1$ whereas the 9/7 has $J = 2$. The reverse application of the transform applies the same procedure but it swaps additions for subtractions.

The application of the lifting scheme to an image is carried out in two stages. First, the lifting scheme is applied to all rows of the image, which is called horizontal filtering. Then, it is applied to the resulting coefficients in a column-by-column fashion in the so-called vertical filtering. The order in which the horizontal and the vertical filtering are applied does not matter as far as the decoder reverses them appropriately. As seen in Fig. 2, these filtering stages produce a dyadic decomposition that contains four subsets of coefficients called wavelet subbands. Subbands are commonly referred to as LL, HL, LH, and HH, with each letter denoting Low or High frequencies in the vertical and horizontal direction. The size of the LL subband is one quarter of that of the original image. Its content is similar to the original image, so coding systems commonly generate new levels of decomposition by applying the DWT to the resulting LL subband. Fig. 2 depicts this scheme when two levels of decomposition are applied to an image. The reverse DWT applies the inverse procedure starting at the last level of decomposition. In general, five levels of decomposition are enough to achieve maximum decorrelation.

III. Overview of CUDA

CUDA is the most popular architecture for general-purpose GPU computing. The CUDA programming model defines a computation hierarchy formed by threads, warps, and thread blocks. A CUDA thread represents a single lane of a vector instruction. Warps are sets of (currently 32) threads that advance their execution in a lockstep synchronous way. Warps are the smallest scheduling units in a GPU. Commonly, all threads in a warp are executed simultaneously, as a single vector operation. Control flow divergence among the threads results in the sequential execution of the divergent paths, so it is commonly avoided. Thread blocks group several warps that are executed independently but that can cooperate via synchronization operations that permit the sharing of data. Warps from multiple blocks are scheduled for execution on a SIMD processing unit called streaming multiprocessor (SM).[1] The occupancy of the GPU (or of a SM) is the percentage of allocated threads relative to the theoretical maximum. Current GPUs include up to 15 SMs. The unit of work sent from the CPU (host) to the GPU (device) is called a kernel. The host can launch some kernels for parallel execution, each composed from tens to millions of thread blocks. The thread blocks are scheduled for independent execution in multiple SMs.

As described before, the memory is organized in three logical spaces: global, shared, and registers. The global memory is shared by all threads in a kernel and has a capacity of several GBs. Data already accessed in the global memory is kept in two levels of on-chip cache for their (possible) reusing. The shared memory is accessible by all warps in a block, while the registers are local to each thread. The communication between the threads in a thread block is commonly carried out via the shared memory. Threads in a warp can also communicate using the shuffle instruction, which permits the access to another thread register of the same warp. The Kepler architecture provides 48 KB and 256 KB for the shared memory and the registers, respectively, per SM. The number of threads allocated in a SM (i.e., its occupancy) is constrained by the amount of shared memory and registers assigned to its threads. The registers have the highest bandwidth and lowest latency, whereas the shared memory bandwidth is significantly lower than that of the registers. The shared memory provides flexible accesses, while the accesses to the global memory must be coalesced to achieve higher efficiency. Among other ways, a coalesced access occurs when consecutive threads of a warp access consecutive memory positions.

IV. Previous and related work

The pre-CUDA GPU-based implementations of the DWT employed manifold devices and programming languages.The implementation proposed in [16], for instance, was based on OpenGL, whereas [17], [18] employed OpenGL and Cg. Most of these earliest methods used convolution operations and were tailored to each filter bank. [18] evaluated for the first time the use of the lifting scheme, though the convolution approach was preferred because the lifting requires the sharing of intermediate values among coefficients. At that time there were no tools to implement that efficiently. This was experimentally confirmed in [19], in which both the convolution and the lifting approach were implemented.

The aforementioned pre-CUDA implementations were constrained by the lack of a general-purpose GPU architecture and its programming tools. The operations of the DWT had to be mapped to graphics operations, which are very limited. Though these works accelerated the execution of the DWT with respect to a CPU-based implementation, their performance is far from that achieved with current GPUs that have an enhanced memory hierarchy and support general-purpose computing. Key in current CUDA implementations is how the image is partitioned to permit parallel processing. Fig. 3 illustrates the three main schemes employed in the literature. They are named row-column, row-block, and block-based.

The first DWT implementation in CUDA was proposed in [20]. It employs the row-column scheme. First, a thread block loads a row of the image to the shared memory and the threads compute the horizontal filtering on that row. After

[1]The streaming multiprocessor is named slightly different in each CUDA architecture, namely, SM in Fermi, SMX in Kepler, and SMM in Maxwell. For simplicity, the acronym SM is adopted herein for all architectures.

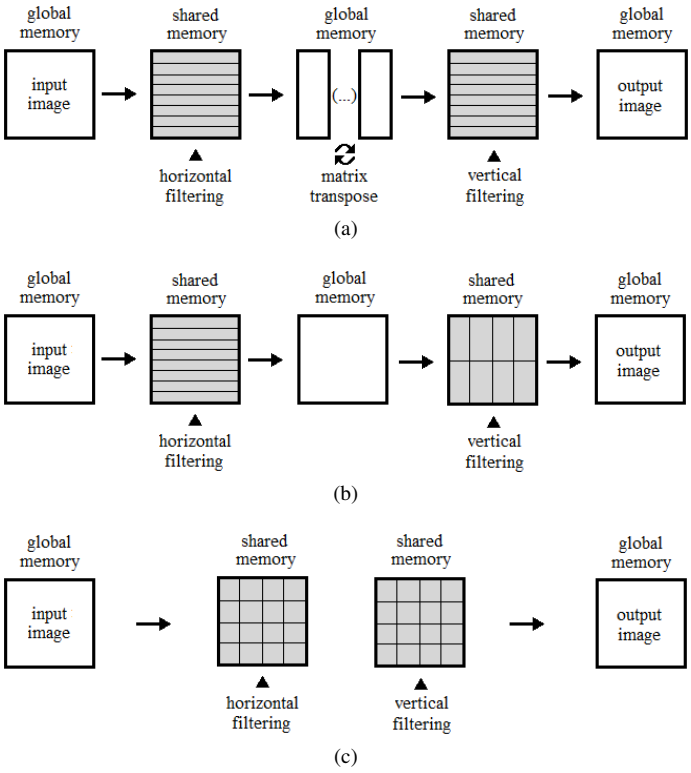


Fig. 3: Illustration of the a) row-column, b) row-block, and c) block-based partitioning schemes to allow parallel processing in the GPU. Horizontal arrows indicate the main data transfers to/from the global memory.

the first filtering stage, all rows of the image are returned to the global memory, in which the image is stored as a matrix. After transposing it, the same procedure is applied, with the particularity that in this second stage the rows are in reality the columns of the original image, so the vertical filtering is actually executed (see Fig. 3(a)). Even though this work still uses convolution operations, speedups between 10 to 20 compared to a multi-core OpenMP implementation are achieved. Its main drawback is the matrix transpose, which is a time-consuming operation. A similar strategy is utilized in [21] and [22] for other types of wavelet transform.

The first CUDA implementation based on the lifting scheme was presented in [23] employing the block-based scheme. The main advantage of this scheme is that it minimizes transfers to the global memory since it computes both the horizontal and the vertical filtering in a single step. It partitions the image in rectangular blocks that are loaded to the shared memory by a thread block. Both the horizontal and the vertical filtering are applied in these blocks, neither needing further memory transfers nor a matrix transpose. The only drawback of such an approach is that there exist data dependencies among adjacent blocks. In [23] these dependencies are not addressed. The common solution to avoid them is to extend all blocks with some rows and columns that overlap with adjacent blocks. These extended rows/columns are commonly called halos.

The fastest implementation of the DWT found in the literature is that proposed in [24], in which the row-block scheme is introduced. The first step of this scheme is the same as that of the row-column, i.e., it loads rows of the image to the shared memory to apply the horizontal filtering on them. Then, the data are returned to the global memory. The second step is similar to what the block-based scheme does. It partitions the image in vertically stretched blocks that are loaded to the shared memory. Consecutive rectangular blocks in the vertical axis are processed by the same thread block employing a sliding window mechanism. This permits the thread block to reuse data in the borders of the blocks, handling the aforementioned problem of data dependencies. The only drawback of such a scheme is that it employs two steps, so more accesses to the global memory are required. The speedups achieved by [24] are approximately from 10 to 14 compared to a CPU implementation using MMX and SSE extensions. They also compare their implementation to convolution-based implementations and to the row-column scheme. Their results suggest that the lifting scheme together with the row-block partitioning is the fastest. The implementation of [24] is employed in the experimental section below for comparison purposes.

Other works in the literature implement the DWT in specific scenarios. [25] employs it in a real-time SPIHT decoding system that uses Reed-Solomon codes. The partitioning scheme used is similar to the row-block but without the sliding window, which forces the reading of more data from the global memory. [26] utilizes a block-based scheme for the compression of hyperspectral images. [27], [28] examines the convolution approach again, whereas [29] implements a variation of the DWT.

Regardless of the partitioning scheme employed, all works in the literature store each partition of the image in the shared memory and assign a thread block to compute each one of them. This is the conventional programming style recommended in the CUDA programming guidelines.

V. Proposed Method

A. Analysis of the partitioning scheme

As most modern implementations, our method employs the lifting scheme. Contrarily to previous work, the proposed approach stores the data of the image partitions in the registers –rather than in the shared memory. It is clear in the literature that the row-column scheme is the slowest [23], [24]. Also, the analysis in [24] indicates that the block-based scheme may not be effective due to its large need of shared memory. This is the main reason behind the introduction of the row-block scheme in [24]. Nonetheless, that analysis is for CUDA architectures prior to Kepler. So it is necessary to study the differences between the row-block and the block-based scheme in current architectures to decide which is adopted in our method. The following analysis assesses memory accesses, computational overhead, and task dependencies.

Both the row-block and the block-based schemes permit data transfers from/to the global memory via coalesced accesses. The main difference between them is the number of global memory accesses performed. The row-block requires the reading and writing of the image (or the LL subband)

twice. All data are accessed in a row-by-row fashion in the first step. After the horizontal filtering, the data are returned to the global memory. The whole image is accessed again using vertically stretched blocks to perform the vertical filter. For images with a large width, it may be more efficient to divide the rows in slices that are processed independently due to memory requirements. Data dependencies among the first and the last samples of the slice have to be handled appropriately. This may slightly increase the number of accesses to the global memory, though not seriously so. A similar issue may appear with images with a large height.

Let m and n respectively be the number of columns and rows of the image. When the row-block scheme is applied, the computation of the first level of decomposition requires, at least, two reads and two writes of all samples to the global memory, i.e., $4mn$ accesses. In general, the application of L levels of wavelet decomposition requires $4M$ accesses, with M being

$$M = \sum_{k=0}^{L-1} \frac{m \cdot n}{4^k} \ . \tag{3}$$

Contrarily to the row-block, the block-based scheme reuses the data after applying the horizontal filtering. If it were not for the data dependencies that exist on the borders of the blocks, this partitioning scheme would require $2M$ accesses, half those of the row-block scheme. To address these dependencies, the partitioning has to be done so that the blocks include some rows and columns of the adjacent blocks, the so-called halo. The size of the halo depends on the lifting iterations of the wavelet filter bank (i.e., J). Let $\hat{m}$ and $\hat{n}$ denote the number of columns and rows of the block –including the halo. The application of an iteration of the lifting scheme in a sample involves dependencies with 2 neighboring coefficients. For the reversible CDF 5/3 transform (with $J = 1$), for instance, these dependencies entail two rows/columns on each side of the block. The samples in these rows/columns are needed to compute the remaining samples within the block, but they must be disregarded in the final result.[2] The number of samples computed without dependency conflicts in each block is $(\hat{m} - 4J) \cdot (\hat{n} - 4J)$. The ratio between the size of the block with and without halos is determined according to

$$H = \frac{\hat{m} \cdot \hat{n}}{(\hat{m} - 4J) \cdot (\hat{n} - 4J)} \ . \tag{4}$$

Since the halos have to be read but not written to the global memory, the number of global memory accesses required by this partitioning scheme is $HM + M$.

Table I evaluates the number of memory accesses needed by the row-block and block-based partitioning schemes for different block sizes and wavelet transforms. The row-block scheme always requires $4M$ accesses. For the block-based scheme, the larger the block size, the fewer the accesses to the global memory, with the lower bound at $2M$. Except for blocks of 16×16 and the use of the 9/7 transform, the number of accesses required by the block-based scheme is always lower than for the row-block scheme. So compared to the row-block scheme, results of Table I suggest that the block-based scheme can reduce the execution time devoted to the memory accesses in a similar proportion. Furthermore, the accesses corresponding to the halos may be accelerated by means of the on-chip caches.

[2]Sides of blocks that coincide with the limits of the image do not have rows/columns with data dependencies. The number of samples corresponding to this is negligible for images of medium and large size, so it is not considered in the discussion for simplicity.

TABLE I: Evaluation of the number of accesses to the global memory required by two partitioning schemes. The row-block scheme requires the same number of accesses regardless of the lifting iterations and block size.

block size ($\hat{m} \times \hat{n}$)	row-block	block-based	
		CDF 5/3	CDF 9/7
16×16	$4M$	$2.78M$	$5M$
32×32		$2.31M$	$2.78M$
64×64		$2.14M$	$2.31M$
128×128		$2.07M$	$2.14M$
64×20		$2.33M$	$2.90M$

The evaluation reported in Table I is theoretical. The following test evaluates the real execution time that is devoted to the memory accesses achieved by the block-based strategy. In this artificial test none logical or arithmetic operation is performed. A warp is assigned to read and write the block data from/to the global memory to/from the registers. Blocks are of 64×20 since this block size fits well our implementation. Results hold for other block sizes too. The same experiment is carried out with and without using the aforementioned halos. Evidently, the writing of data to the global memory is carried out only for the relevant samples when halos are used. When no halos are utilized, $2M$ accesses are performed, whereas the use of halos performs $HM + M$ accesses as mentioned earlier.

Table II reports the results achieved when using different image sizes, for both the reversible CDF 5/3 (with $J = 1$) and the irreversible CDF 9/7 (with $J = 2$) transform. The theoretical increase in memory accesses (i.e., $(HM+M)/2M$) due to the use of halos for this experiment is 1.17 and 1.45, respectively for the 5/3 and 9/7 transform. The experimental results suggest that the theoretical analysis is approximately accurate for the 5/3 transform, especially for images of medium and large size. Contrarily, the real increase for the 9/7 transform is larger than the theoretical. This is caused because the writing of samples is *not* done in a fully coalesced way. The threads assigned to the halo hold irrelevant data, so they are idle when writing the results. In spite that the real increase is higher than the theoretical, we note that it is always below 2, which is the point at which the row-block scheme would be more effective than the block-based.

Another aspect that may be considered when analyzing these partitioning schemes is the arithmetic operations that are performed. The row-block applies the lifting scheme to all image coefficients (or to those in the LL subband) once, so it performs λM operations, with λ denoting the number of arithmetic operations needed to apply the lifting scheme to each coefficient. Samples within the halos in the block-

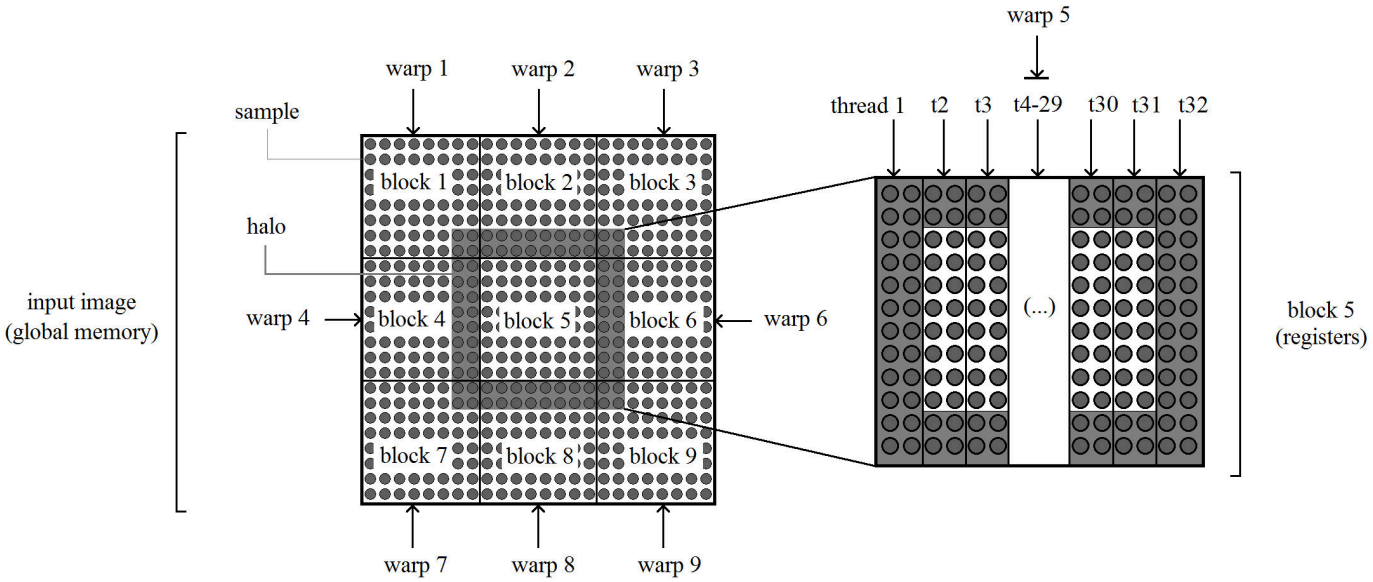


Fig. 4: Illustration of the partitioning scheme and the thread-to-data mapping employed by the proposed method when applying a CDF 5/3 transform.

TABLE II: Evaluation of the practical and theoretical increase in computational time due to the inclusion of halos in each block. Blocks of size 64×20 are employed. Experiments are carried out with a Nvidia GTX TITAN Black.

	image size ($m \times n$)	exec. time (in μs) no halos	exec. time (in μs) halos	real inc.	theor. inc.
CDF 5/3	1024×1024	17	23	1.35	1.17
	2048×2048	63	74	1.17	
	4096×4096	245	279	1.14	
	8192×8192	981	1091	1.11	
CDF 9/7	1024×1024	19	34	1.8	1.45
	2048×2048	67	116	1.73	
	4096×4096	255	444	1.74	
	8192×8192	1015	1749	1.72	

based scheme compel the application of the lifting scheme in some coefficients more than once. Again, the increase can be determined through the percentage of samples within the halos in each block, resulting in λHM. Although the block-based scheme performs more arithmetic operations, in practice this becomes inconsequential since the performance of the DWT implementation is bounded by the memory accesses (see next section).

The final aspect of this analysis studies the task dependencies of the algorithm. There are two task dependencies that must be considered. They are the application of the horizontal and vertical filtering, and the application of the prediction and update steps. In both cases, the tasks have to be applied one after the other. The steps dependency can be handled in both partitioning schemes with local synchronization within each thread block. The horizontal-vertical filtering dependency has different constrains in each partitioning scheme. The row-block needs to synchronize all the thread blocks after each filter pass. This can only be implemented via two kernels that are executed sequentially. The block-based scheme does not require synchronization among different thread blocks since all data is within the block, so local synchronization can be employed. This is generally more effective than executing two sequential kernels.

B. Thread-to-data mapping

The analysis of the previous section indicates that the block-based partitioning scheme requires fewer global memory accesses and that it can be implemented employing effective synchronization mechanisms. The proposed method uses a scheme similar to the block-based. Besides storing the data of the image partitions in the registers, another important difference with respect to previous works is that each partition is processed by a warp instead of using a thread block. This strategy does not need shared memory since threads within a warp can communicate via shuffle instructions. It also avoids the use of synchronization operations required by inter-lifting dependencies since the threads in a warp are intrinsically synchronized and there is no need to communicate data between warps. The removal of all synchronization operations elevates the warp-level parallelism.

Fig. 4 illustrates the partitioning strategy employed. The rectangular divisions of the image represent the samples within each block that can be computed with*out* dependency conflicts. The surrounding gray rows/columns of block 5 in the figure represent the real extension (including the halo) of that block. For the 5/3 transform, two rows/columns are added to each side to resolve data dependencies. The real size of all other blocks in the figure also includes two rows/columns in each side, though it is not illustrated for clarity.

The size of the block directly affects the overall performance of the implementation since it has impact on the memory access pattern, the register usage, the occupancy, and the total

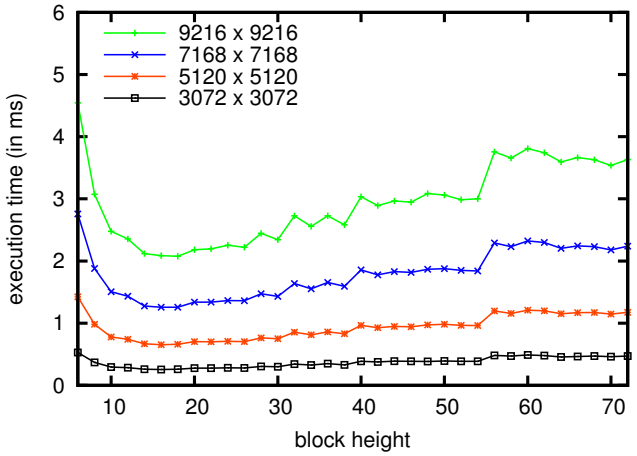


Fig. 5: Evaluation of the execution time achieved by the proposed method when employing blocks of different height. 5 decomposition levels of forward CDF 5/3 wavelet transform are applied to images of different size. Experiments are carried out with a Nvidia GTX TITAN Black.

number of instructions executed. Key to achieve maximum efficiency is that the threads in a warp read and write the rows of the block performing coalesced accesses to the global memory. To achieve it, the block width must be a multiple of the threads within a warp, which is 32 in current architectures. To assign pairs of samples to each thread is highly effective. The processing of two adjacent samples per thread permits the application of the lifting scheme first for samples in the even positions of the array and then for samples in the odd positions. We note that to assign only one sample per thread would generate divergence since only half the threads in a warp could compute the (intermediate) wavelet coefficients. So the width of the block that we employ is $\hat{m} = 64$. This is illustrated in the right side of Fig. 4.

In our implementation, each thread holds and processes all pairs of samples of two consecutive columns. With such a mapping, the application of the vertical filtering does not require communication, whereas the horizontal filtering requires collaboration among threads. With this mapping, the threads collaboratively request $\hat{n}$ rows from the global memory before carrying out any arithmetic operation. This generates multiple on-the-fly requests to the global memory, key to hide the latency of the global memory since arithmetic operations are then overlapped with memory accesses.

The height of the block permits some flexibility. Fig. 5 reports the results that are achieved by the proposed method when employing different block heights. Results are for images of different size and for the forward CDF 5/3 transform, though they hold for other wavelet filter banks and for the reverse application of the transform. The results in this figure indicate that the lowest execution times are achieved when the height of the block is between 12 to 26, approximately.

Table III extends the previous test. It reports the registers employed by each thread, the device occupancy, and the number of instructions executed when applying the proposed method to an image of size 7168 $\times$ 7168 employing blocks

TABLE III: Evaluation of some GPU metrics when the proposed method is applied to an image of size 7168 $\times$ 7168 employing different block heights, for the forward CDF 5/3 transform. Experiments are carried out with a Nvidia GTX TITAN Black.

block height	registers used	device occupancy	instructions executed (x10^3)
10	34	69%	70655
20	57	45%	45153
30	78	33%	39749
40	97	22%	37106
50	121	22%	35599
60	141	16%	34604
70	161	16%	34222

of different heights. We assure that register spilling does not occur in any of these, and following, tests. The smaller the block height, the fewer registers used per thread and the higher the occupancy of the device. Then, more instructions are executed due to larger halos. As seen in Fig. 5, the tradeoff between the device occupancy and the number of instructions executed is maximized with blocks of 64 $\times$ 20, approximately. For this block size, the occupancy of the device is 45% and the number of instructions executed is much lower than when using blocks of 64 $\times$ 10. These results hold for the CDF 9/7 transform and for images of other sizes. The results of the next section employ a block size of 64×20.

C. Algorithm

Algorithm 1 details the CUDA kernel implemented in this work for the forward application of the wavelet transform. We recall that a CUDA kernel is executed by all threads in each warp identically and synchronously. The parameters of the algorithm are the thread identifier (i.e., T), the first column and row of the image corresponding to the block processed by the current warp (i.e., X, Y), and the first column and row of the wavelet subbands in which the current warp must leave the resulting coefficients (i.e., X_S, Y_S). The height of the block is denoted by $\bar{Y}$ and is a constant.

The first operation of Algorithm 1 reserves the registers needed by the thread. The registers are denoted by $\mathcal{R}$, whereas the global memory is denoted by $\mathcal{G}$. From line 2 to 5, the thread reads from the global memory the two columns that it will process. The reading of two consecutive columns can be implemented with coalesced accesses to the global memory. The reading of all data before carrying out any arithmetic operation generates the aforementioned on-the-fly memory accesses.

The horizontal filtering is carried out in lines 6-13 as specified in Eq. (1) and (2) employing that α^j and β^j corresponding to the wavelet filter bank. Since this filtering stage is applied along each row, the threads must share information among them. The operation $\Phi(\cdot)$ in lines 8,10 is the shuffle instruction introduced in the CUDA Kepler architecture. This operation permits thread T to read a register from any other thread in the warp. The register to be read is specified in the first parameter

Algorithm 1 Forward DWT kernel

Parameters:
T thread with $T \in [0, 31]$
X, Y first column and row of the block in the image
X_S, Y_S first column and row of the block in the S subband

1: **allocate** $\mathcal{R}[\bar{Y}][2]$ **in register memory space**
2: **for** $y \in \{0, 1, 2, ..., \bar{Y} - 1\}$ **do**
3: $\quad \mathcal{R}[y][0] \leftarrow \mathcal{G}[Y + y][X + T * 2]$
4: $\quad \mathcal{R}[y][1] \leftarrow \mathcal{G}[Y + y][X + T * 2 + 1]$
5: **end for**
6: **for** $j \in \{0, 1, 2, ..., J - 1\}$ **do**
7: $\quad$ **for** $y \in \{0, 1, 2, ..., \bar{Y} - 1\}$ **do**
8: $\quad\quad \mathcal{R}' \leftarrow \Phi(\mathcal{R}[y][0], T + 1)$
9: $\quad\quad \mathcal{R}[y][1] \leftarrow \mathcal{R}[y][1] - \alpha^j(\mathcal{R}[y][0] + \mathcal{R}')$
10: $\quad\quad \mathcal{R}' \leftarrow \Phi(\mathcal{R}[y][1], T - 1)$
11: $\quad\quad \mathcal{R}[y][0] \leftarrow \mathcal{R}[y][1] - \beta^j(\mathcal{R}[y][1] + \mathcal{R}')$
12: $\quad$ **end for**
13: **end for**
14: **for** $j \in \{0, 1, 2, ..., J - 1\}$ **do**
15: $\quad$ **for** $y \in \{1, 3, 5, ..., \bar{Y} - 1\}$ **do**
16: $\quad\quad \mathcal{R}[y][0] \leftarrow \mathcal{R}[y][0] - \alpha^j(\mathcal{R}[y - 1][0] + \mathcal{R}[y + 1][0])$
17: $\quad\quad \mathcal{R}[y][1] \leftarrow \mathcal{R}[y][1] - \alpha^j(\mathcal{R}[y - 1][1] + \mathcal{R}[y + 1][1])$
18: $\quad$ **end for**
19: $\quad$ **for** $y \in \{0, 2, 4, ..., \bar{Y} - 2\}$ **do**
20: $\quad\quad \mathcal{R}[y][0] \leftarrow \mathcal{R}[y][0] - \beta^j(\mathcal{R}[y - 1][0] + \mathcal{R}[y + 1][0])$
21: $\quad\quad \mathcal{R}[y][1] \leftarrow \mathcal{R}[y][1] - \beta^j(\mathcal{R}[y - 1][1] + \mathcal{R}[y + 1][1])$
22: $\quad$ **end for**
23: **end for**
24: **for** $y \in \{2J, 2J + 2, ..., \bar{Y} - 2J\}$ **do**
25: $\quad \mathcal{G}[Y_{LL} + y/2][X_{LL} + T] \leftarrow \mathcal{R}[y][0]$
26: $\quad \mathcal{G}[Y_{HL} + y/2][X_{HL} + T] \leftarrow \mathcal{R}[y][1]$
27: **end for**
28: **for** $y \in \{2J + 1, 2J + 3, ..., \bar{Y} - 2J + 1\}$ **do**
29: $\quad \mathcal{G}[Y_{LH} + y/2][X_{LH} + T] \leftarrow \mathcal{R}[y][0]$
30: $\quad \mathcal{G}[Y_{HH} + y/2][X_{HH} + T] \leftarrow \mathcal{R}[y][1]$
31: **end for**

of this function. The second parameter of $\Phi(\cdot)$ is the thread identifier from which the register is read.

The vertical filtering is applied in the loops of lines 14-23. In this case, the thread has all data needed to apply it, so the prediction step is carried out first in the loop of lines 15-18 followed by the update step. Note that in the horizontal filtering, the prediction and update steps were carried out within the same loop since all threads process the same row simultaneously.

The last two loops in Algorithm 1 (lines 24-31) write the resulting coefficients in the corresponding wavelet subbands stored in the global memory. In this case, accesses to the global memory are not fully coalesced as mentioned earlier. These loops only transfer the rows that do not belong to the halos. Our implementation also takes into account that the threads containing the first and last columns of the block do not write their coefficients in the global memory since they have dependency conflicts, though it is not shown in Algorithm 1 for simplicity.

This algorithm details the forward application of the wavelet transform for one decomposition level. The application of more decomposition levels carries out the same procedure but taking the resulting LL subband of the previous decomposition level as the input image. Also, the reverse operation of the transform is implemented similarly as the procedure specified in Algorithm 1.

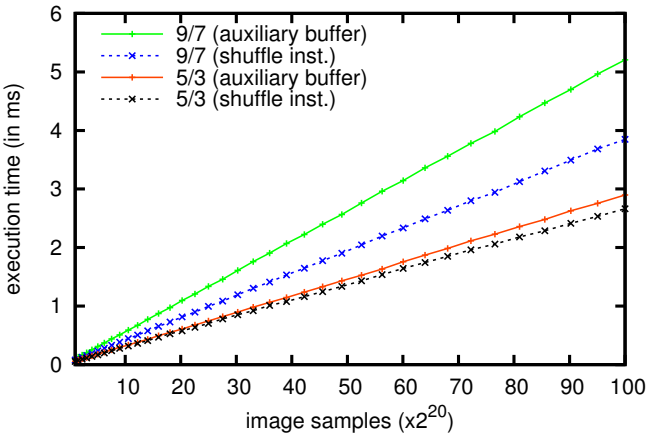


Fig. 6: Evaluation of the execution time achieved by the proposed method when employing shuffle instructions or an auxiliary buffer in the shared memory to communicate data among threads. Five decomposition levels of forward CDF 5/3 or 9/7 wavelet transform are applied to images of different size. Experiments are carried out with a Nvidia GTX TITAN Black.

Although the proposed method has been devised for the Kepler CUDA architecture and following, Algorithm 1 could also be employed in previous architectures. Before Kepler, the data sharing among threads in a warp can be implemented by using an auxiliary buffer in the shared memory. By only replacing the shuffle instructions in lines 8,10 by the use of this auxiliary buffer, our register-based strategy could be employed in pre-Kepler architectures. This strategy is employed in the next section to assess the performance achieved with GPUs of the Fermi architecture. Evidently, the shuffle instruction is faster than the use of an auxiliary buffer due to the execution of fewer instructions. See in Fig. 6 the execution time spent by the proposed method when employing shuffle instructions or the auxiliary buffer. Shuffle instructions accelerate the execution of the 9/7 transform in approximately 20%.

On another note, the proposed method can also be employed to perform strategies of wavelet transformation that involve three dimensions in images with multiple components, such as remote sensing hyperspectral images or 3D medical images. The conventional way to apply such strategies is to reorder the original samples and apply the DWT afterwards [30].

VI. Experimental results

A. Overall performance

Except when indicated, the experimental results reported in this section are carried out with a Nvidia GTX TITAN Black GPU using the CUDA v5.5 compiler. This GPU has 15 SMs and a peak global memory bandwidth of 336 GB/s. The proposed algorithm is compiled and executed conventionally, without needing any assembly edit. Results have been collected employing the Nvidia profiler tool nvprof. All the experiments apply five levels of decomposition to images of

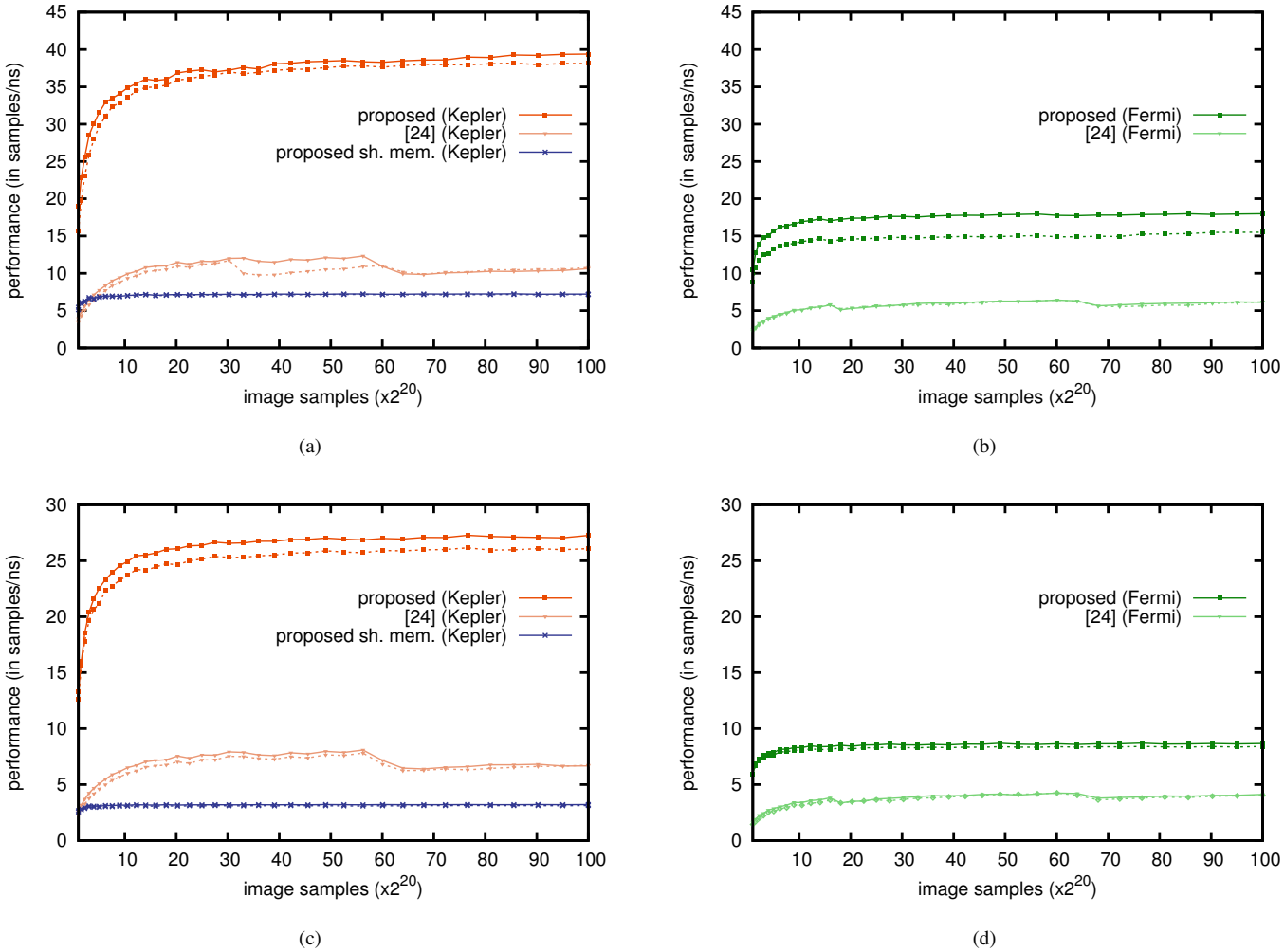


Fig. 7: Evaluation of the performance achieved by the proposed method and [24], for (a),(b) the CDF 5/3 transform and (c),(d) the CDF 9/7 transform. Solid lines indicate the forward application of the transform and dashed lines indicate the reverse.

size ranging from 1024×1024 to 10240×10240. The data structures employed to store the image samples are of 16 bits. Floats of 32 bits are employed to perform all computations of the CDF 9/7 since they provide enough arithmetic precision for image coding applications.

The first test evaluates the performance achieved by the proposed method and compares it to the best implementation found in the literature [24]. The implementation in [24] is configured to obtain maximum performance in this GPU using the maximum shared memory size. Fig. 7(a) and (c) depict the results achieved for both the reversible CDF 5/3 and the irreversible CDF 9/7 wavelet transforms. The horizontal axis of the figures is the size of the image, measured as the number of image samples, whereas the vertical axis is the performance. The metric employed to evaluate the performance is the number of samples processed per unit of time. The plots with the label "proposed" depict the performance of our method when the data of the blocks are stored in the registers. To compare the performance achieved by the use of registers vs the use of shared memory, these figures also report the performance achieved with the GTX TITAN Black when our implementation uses a buffer in the shared memory to hold all the data.[3] To assess the increase in performance achieved with different Nvidia architectures, Fig. 7(b) and (d) depict the results achieved with a Tesla M2090 GPU (Fermi architecture). Data blocks of size 64×20 and thread blocks of 128 are employed for all implementations. In our implementation, thread blocks of 128 achieve the best results. Thread blocks of 64 may obtain slightly better performance in some applications, especially when using Maxwell architectures. In our case the differences between blocks of 128 and 64 are negligible. As seen in Fig. 7, both the forward and the reverse application of the wavelet transform achieve similar performance since they perform practically the same operations in the inverse order.

The experimental results of Fig. 7 indicate that the performance speedup between Fermi and Kepler achieved by [24] is approximately 1.7 for both the CDF 5/3 and 9/7 transform. The performance speedup achieved by our implementation is

[3]Such a strategy does *not* explicitly use the registers, so all data are kept in the shared memory. It is configured to avoid bank conflicts in the shared memory and to employ the maximum shared memory size, maximizing performance.

TABLE IV: Evaluation of the total number of instructions executed and global memory accesses performed by the proposed method and by the implementation in [24] (Kepler architecture). Results are for the forward transform.

		instructions executed (x10^3)					mem. accesses (x10^3)		
	image size	proposed	[24]	increase	prop. (sh. mem.)	increase	proposed	[24]	increase
CDF 5/3	1024×1024	982	3554	3.62	1618	1.64	208	348	1.67
	2048×2048	3804	12350	3.25	6224	1.63	822	1395	1.70
	4096×4096	14926	44541	2.98	24357	1.63	3287	5587	1.70
	8192×8192	59083	167320	2.83	96462	1.63	13230	22360	1.69
CDF 9/7	1024×1024	2026	5370	2.65	3743	1.84	210	366	1.74
	2048×2048	7847	18921	2.41	14456	1.84	825	1396	1.69
	4096×4096	31200	69266	2.22	57513	1.84	3313	5588	1.69
	8192×8192	123963	262917	2.12	228583	1.84	13458	22514	1.67

2.3 and 3, respectively for the 5/3 and 9/7. This difference is because our implementation exploits very efficiently the resources of the Kepler architecture. Furthermore, note that the performance achieved by our implementation when using the Tesla M2090 GPU (Fermi architecture) is even higher (1.4 on average) than that of [24] when using the GTX TITAN Black (Kepler architecture). When comparing the results achieved by both implementations with the Kepler architecture, the results of Fig. 7 show that the proposed register-based implementation is significantly faster than the method presented in [24]. Though it depends on the wavelet transform and the size of the input data, speedups ranging approximately from 3.5 to almost 5 are achieved. The gain in performance is caused by the novel programming methodology based on the use of registers, which results in a lower number of instructions executed and a lower number of global memory accesses. This can be seen in Table IV. The columns with the label "increase" in this table report the increase ratio in the number of instructions or accesses with respect to the proposed method. The implementation in [24] executes three times more instructions and around 70% more memory accesses than ours. This corresponds with the theoretical analysis of Section V (Table I).

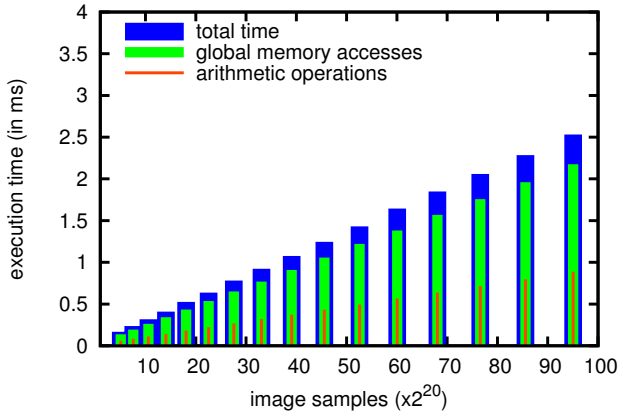
The results of Fig. 7(a) and (c) also indicate that, in our implementation, the use of registers speedups the execution from 3.5 to 9 times with respect to the use of shared memory. As mentioned previously, the use of shared memory significantly decreases the performance due to a low occupancy of the device and the fact that data have to be moved from the shared memory to the registers to perform arithmetic operations. The occupancy achieved by "proposed sh. mem." is 12%, as opposed to the 45% achieved when using registers. The low occupancy achieved by the use of shared memory is constrained by the amount of shared memory assigned per thread block. Though this occupancy could be increased by reducing the data block size, the number of instructions executed is then significantly increased and so the overall performance is reduced. As seen in Table IV, when the proposed method employs shared memory instead of registers, the number of instructions is increased in approximately 60% and 80% for the 5/3 and 9/7 transform, respectively. This is due to the operations that move the data from the shared memory to the registers. The number of memory accesses performed by the proposed method is the same regardless of using registers or shared memory, so it is not shown in the table. We note that these results correspond with [24], in which it was already indicated that the block-based scheme employing shared memory is *not* efficient. The use of the proposed register-based strategy enhances the performance of such a scheme greatly.

Another observation that stems from Fig. 7 is that our method achieves regular performance regardless of the image size, indicating that it scales well with the size of the input data. This is seen in the figure as the almost straight plot of our implementation. Only for small images the performance decreases due to low experimental occupancy.

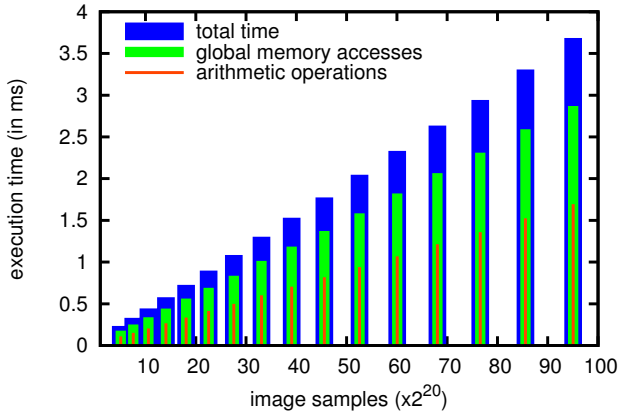
B. Analysis of execution bottleneck

The aim of the next test is to identify the execution bottleneck. To this end, it separately evaluates the time spent by the arithmetic operations and the time spent by the global memory accesses in our register-based implementation. Fig. 8 depicts the results achieved by the forward application of both the 5/3 and 9/7 transform. Again, the horizontal axis of the figure is the number of image samples, whereas the vertical axis is the execution time. The plot with the label "global memory accesses" reports the time spent by reading and writing all data from/to the global memory in an implementation in which all arithmetic operations are removed. The plot with the label "arithmetic operations" reports the time spent by the arithmetic operations in an implementation in which all memory accesses are removed. The plot with the label "total time" is our original implementation with both memory accesses and arithmetic operations.

It is worth noting in Fig. 8 that the time spent to perform the arithmetic operations is less than that spent for the memory accesses. As it is also observed in the figure, the overlapping of the arithmetic operations with the memory accesses is carried out efficiently. If the arithmetic operations were not overlapped with the memory accesses, the total execution time should be the sum of both. If the overlapping were realized perfectly, the execution time should be the maximum of both. These results indicate that the proposed method is mostly memory bounded, especially for the 5/3 transform. As previously stated, such an overlapping is achieved thanks to the large number of



(a)



(b)

Fig. 8: Evaluation of the execution time spent to perform only the arithmetic operations, accesses to the global memory, and both the arithmetic operations and accesses (total time), for the forward application of the (a) CDF 5/3 transform and (b) the CDF 9/7 transform.

on-the-fly requests to the global memory carried out in our implementation.

As seen in Fig. 8 (and also in Fig. 7), the performance achieved with the reversible CDF 5/3 transform is approximately 40% higher than that achieved with the irreversible CDF 9/7. This difference is caused by two factors. First, the 9/7 has $J = 2$, so its lifting scheme requires twice the number of arithmetic operations as that of the 5/3 (see Table IV). The amount of time required to execute the arithmetic instructions of the 9/7 is also twice that required by the 5/3 (see Fig. 8). Thanks to the overlapping of the arithmetic operations with the memory accesses, the total execution time of the 9/7 is *not* doubled. Nonetheless, the overlapping achieved by the 9/7 is not as effective as that achieved by the 5/3. The second factor behind the lower performance achieved with the 9/7 is that the blocks in the 9/7 need larger halos than with the 5/3, requiring more memory accesses. The time spent by the 9/7 transform to carry out the memory accesses is approximately 30% higher than that spent by the 5/3. Even so, in Table IV the

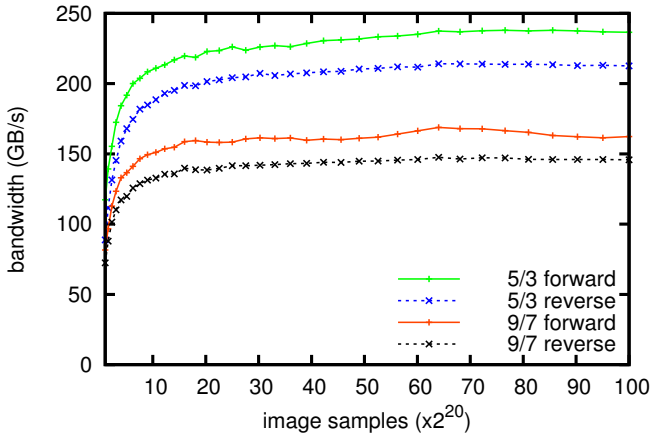


Fig. 9: Evaluation of the global memory bandwidth usage achieved by the proposed method.

number of memory accesses carried out by both transforms is the same because the extra memory accesses (corresponding to the larger halos of the 9/7) are reused in the on-chip cache. In practice, the 9/7 performs 40% more memory accesses to the on-chip cache than the 5/3 (data not shown), increasing the time spent by the memory accesses. We note that large on-chip caches can hold more data reused for the halos, reducing the computational time.

The aim of the next test is to appraise whether our implementation is bounded by the memory bandwidth or by the memory latency. The results of Fig. 9 depict the experimentally measured bandwidth. As reported by Nvidia, 100% usage of the peak memory bandwidth is not attainable in practice. The maximum attainable can be approximated by that obtained by the Nvidia SDK bandwidth test. In the GTX TITAN Black, this test achieves an usage of 70%. Our implementation achieves an average bandwidth of 65% and 50% for the 5/3 and 9/7 transform, respectively. These results reveal that the implementation applying the 5/3 transform is bounded by the global memory bandwidth. The 9/7 does not reach the maximum bandwidth usage and so more parallelism (by means of more thread- and instruction-level parallelism) could improve its performance. The reverse application of the transforms achieves lower bandwidth usage due to the more scattered access pattern that they use when reading the image since each warp fetches data from four different subbands.

C. Evaluation in other devices

The last test evaluates the performance of our register-based implementation in four different GPUs. The features of the employed devices are shown in Table V. The Tesla M2090 has a Fermi architecture, the GTX 680 and GTX TITAN Black have a Kepler architecture, and the GTX 750 Ti has the latest architecture called Maxwell. The four devices have different memory bandwidth. The experimental bandwidth depicted in the table is computed with the Nvidia SDK bandwidth test. The results achieved with these devices can be found in Table VI and Fig. 10. The table reports the execution

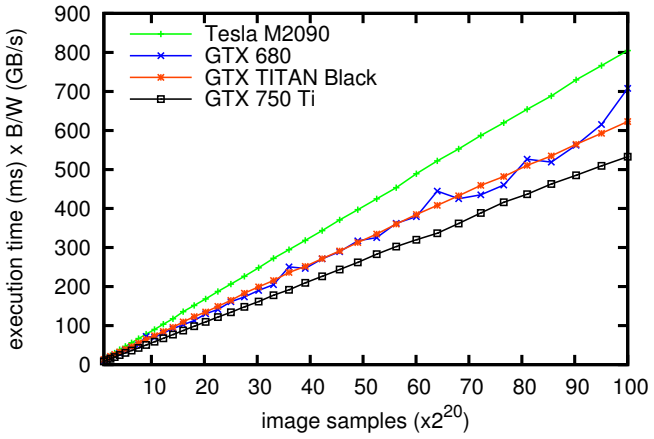


Fig. 10: Evaluation of the execution time weighted by the experimentally measured global memory bandwidth in different GPUs, for the forward application of the 5/3 transform.

time for both the 5/3 and 9/7 transform, whereas the figure depicts the execution time multiplied by the global memory bandwidth of each device, for the forward application of the 5/3 transform. As seen in Table V, the execution time achieved is mainly related to the memory bandwidth. The GTX TITAN Black has the highest bandwidth and so the lowest execution time, followed by the GTX 680, the GTX 750 Ti, and the Tesla M2090. Despite the differences seen in this table, note in Fig. 10 that the performance of both the GTX TITAN Black and the GTX 680 is almost the same considering their difference in the global memory bandwidth. This figure also discloses that the GTX 750 Ti, which has the highest compute capability and the largest on-chip cache, achieves slightly better performance than the remaining devices considering its memory bandwidth. The small irregularities achieved by the GTX 680 in Fig. 10 may be because this GPU has the smallest on-chip cache and the fewest number of SMs, which may affect its performance for some image sizes. As seen in the figure, the Tesla M2090 achieves the lowest performance because the Fermi architecture has half the register memory space per SM as that of Kepler and Maxwell architectures, which reduces the device occupancy. Also, because it does not employ shuffle instructions.

VII. Conclusions

This paper introduces an implementation of the DWT in a GPU through a register-based strategy. This kind of implementation strategy has recently become feasible in the latest CUDA architectures due to the expansion of the register memory space and the introduction of instructions to allow data sharing in the registers. Despite improvements on other aspects of the device, it is likely that future generations of GPUs will maintain or enlarge the register space and enhance the communication capabilities of registers. The key features of the proposed method are the use of the register memory space to perform all operations and an effective block-based partitioning scheme and thread-to-data mapping that permit the assignment of warps to process all data of a block. Experimental evidence indicates that the proposed register-based strategy obtains better performance than the use of shared memory since it requires fewer instructions and achieves higher GPU occupancy.

Experimental analyses suggest that the proposed implementation is memory bounded. The global memory bandwidth achieved is close to the experimental maximum and most of the computation is overlapped with the memory accesses. Since most of the global memory traffic is unavoidable (i.e., employed to read the input image and to write the output data), we conclude that the execution times achieved by the proposed implementation are close to the limits attainable in current architectures. Compared to the state of the art, our register-based implementation achieves speedups of 4, on average. The implementation employed in this work is left freely available in [31].

Conceptually, the application of the DWT can also be seen as a stencil pattern [32]. Stencils, and other algorithms with similar data reuse patterns, may also benefit from a implementation strategy similar to that described in this work.

Acknowledgment

The authors thank A. C. Jalba, W. J. van der Laan and J. B. T. M. Roerdink for providing the implementation of the method proposed in [24].

References

[1] "CUDA, C Programming guide," Tech. Rep., Feb. 2014.
[2] V. Volkov and J. W. Demmel, "Benchmarking GPUs to tune dense linear algebra," in *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, Nov. 2008, pp. 31–42.
[3] V. Volkov, "Better performance at lower occupancy," in *Proceedings of the GPU Technology Conference*, vol. 10, Sep. 2010.
[4] F. N. Iandola, D. Sheffield, M. Anderson, P. M. Phothilimthana, and K. Keutzer, "Communication-minimizing 2D convolution in GPU registers," in *Proceedings of the IEEE International Conference on Image Processing*, Sep. 2013, pp. 2116–2120.
[5] A. Chacón, S. Marco-Sola, A. Espinosa, P. Ribeca, and J. C. Moure, "FM-index on GPU: a cooperative scheme to reduce memory footprint," in *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing with Applications*, 2014, in Press.
[6] ——, "Thread-cooperative, bit-parallel computation of Levenshtein distance on GPU," in *Proceedings of the 28th ACM International Conference on Supercomputing*, Jun. 2014, pp. 103–112.
[7] *Information technology - JPEG 2000 image coding system - Part 1: Core coding system*, ISO/IEC Std. 15 444-1, Dec. 2000.
[8] A. Skodras, C. Christopoulos, and T. Ebrahimi, "The JPEG2000 still image compression standard," *IEEE Signal Process. Mag.*, vol. 18, no. 5, pp. 36–58, Sep. 2001.
[9] *Image Data Compression*, Consultative Committee for Space Data Systems Std. CCSDS 122.0-B-1, Nov. 2005.
[10] A. Said and W. A. Pearlman, "A new, fast, and efficient image codec based on set partitioning in hierarchical trees," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 6, no. 3, pp. 243–250, Jun. 1996.
[11] D. Taubman, "High performance scalable image compression with EBCOT," *IEEE Trans. Image Process.*, vol. 9, no. 7, pp. 1158–1170, Jul. 2000.
[12] W. A. Pearlman, A. Islam, N. Nagaraj, and A. Said, "Efficient, low-complexity image coding with a set-partitioning embedded block coder," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 14, no. 11, pp. 1219–1235, Nov. 2004.
[13] S. Mallat, "A theory of multiresolution signal decomposition: the wavelet representation," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 11, pp. 674–693, Jul. 1989.

TABLE V: Features of the GPUs employed.

	Tesla M2090	GTX 680	GTX TITAN Black	GTX 750 Ti
compute capability	2.0	3.0	3.5	5.0
clock frequency	1301 MHz	1006 MHz	889 MHz	1020 MHz
SMs	16	8	15	5
number of cores	512	1536	2880	640
register space per SM	128 KB	256 KB	256 KB	256 KB
shared memory per SM	48 KB	48 KB	48 KB	64 KB
size of global memory	6144 MB	2048 MB	6144 MB	2048 MB
memory bandwidth (theoretical)	177.6 GB/s	192.2 GB/s	336 GB/s	86.4 GB/s
memory bandwidth (experimental)	138.11 GB/s	146.4 GB/s	234 GB/s	67.1 GB/s
size of on-chip L2 cache	768 KB	512 KB	1536 KB	2048 KB
peak GFLOPS (single precision)	1331	3090	5121	1306

TABLE VI: Evaluation of the execution time achieved with different GPUs, for the forward application of the 5/3 and 9/7 transform using 5 decomposition levels.

		execution time (in μs)			
	image size	Tesla M2090	GTX 680	GTX TITAN Black	GTX 750 Ti
CDF 5/3	1024×1024	100	70	55	125
	2048×2048	278	204	139	370
	4096×4096	983	698	467	1305
	8192×8192	3782	3038	1741	5021
CDF 9/7	1024×1024	176	97	79	171
	2048×2048	538	282	194	540
	4096×4096	1997	1024	652	1990
	8192×8192	7811	3960	2490	7686

[14] D. S. Taubman and M. W. Marcellin, *JPEG2000 Image compression fundamentals, standards and practice*. Norwell, Massachusetts 02061 USA: Kluwer Academic Publishers, 2002.

[15] W. Sweldens, "The lifting scheme: A construction of second generation wavelets," *SIAM Journal on Mathematical Analysis*, vol. 29, no. 2, pp. 511–546, Mar. 1998.

[16] M. Hopf and T. Ertl, "Hardware accelerated wavelet transformations," in *Proceedings of the EG/IEEE TCVG Symposium on Visualization*, May 2000, pp. 93–103.

[17] A. Garcia and H.-W. Shen, "GPU-based 3D wavelet reconstruction with tileboarding," *The Visual Computer*, vol. 21, no. 8-10, pp. 755–763, Sep. 2005.

[18] T.-T. Wong, C.-S. Leung, P.-A. Heng, and J. Wang, "Discrete wavelet transform on consumer-level graphics hardware," *IEEE Trans. Multimedia*, vol. 9, no. 3, pp. 668–673, Apr. 2007.

[19] C. Tenllado, J. Setoain, M. Prieto, L. Piñuel, and F. Tirado, "Parallel implementation of the 2D discrete wavelet transform on graphics processing units: filter bank versus lifting," *IEEE Trans. Parallel Distrib. Syst.*, vol. 19, no. 3, pp. 299–310, Mar. 2008.

[20] J. Franco, G. Bernabé, J. Fernández, and M. E. Acacio, "A parallel implementation of the 2D wavelet transform using CUDA," in *Proceedings of the 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, Feb. 2009, pp. 111–118.

[21] J. Franco, G. Bernabé, J. Fernández, and M. Ujaldón, "Parallel 3D fast wavelet transform on manycore GPUs and multicore CPUs," *Procedia Computer Science*, vol. 1, no. 1, pp. 1101–1110, May 2010.

[22] Z. Wei, Z. Sun, Y. Xie, and S. Yu, "GPU Acceleration of integer wavelet transform for TIFF image," in *Proceedings of the Third International Symposium on Parallel Architectures, Algorithms and Programming (PAAP)*, Dec. 2010, pp. 138–143.

[23] J. Matela *et al.*, "GPU-based DWT acceleration for JPEG2000," in *Annual Doctoral Workshop on Mathematical and Engineering Methods in Computer Science*, Nov. 2009, pp. 136–143.

[24] W. J. van der Laan, A. C. Jalba, and J. B. Roerdink, "Accelerating wavelet lifting on graphics hardware using CUDA," *IEEE Trans. Parallel Distrib. Syst.*, vol. 22, no. 1, pp. 132–146, Jan. 2011.

[25] C. Song, Y. Li, and B. Huang, "A GPU-accelerated wavelet decompression system with SPIHT and Reed-Solomon decoding for satellite images," *IEEE J. Sel. Topics Appl. Earth Observations Remote Sens.*, vol. 4, no. 3, pp. 683–690, Sep. 2011.

[26] M. Ciznicki, K. Kurowski, and A. Plaza, "Graphics processing unit implementation of JPEG2000 for hyperspectral image compression," *Journal of Applied Remote Sensing*, vol. 6, no. 1, pp. 061 507–1, Jan. 2012.

[27] V. Galiano, O. López, M. P. Malumbres, and H. Migallón, "Parallel strategies for 2D discrete wavelet transform in shared memory systems and GPUs," *The Journal of Supercomputing*, vol. 64, no. 1, pp. 4–16, Apr. 2013.

[28] V. Galiano, O. López-Granado, M. Malumbres, and H. Migallón, "Fast 3D wavelet transform on multicore and many-core computing platforms," *The Journal of Supercomputing*, vol. 65, no. 2, pp. 848–865, Aug. 2013.

[29] J. Chen, Z. Ju, C. Hua, B. Ma, C. Chen, L. Qin, and R. Li, "Accelerated implementation of adaptive directional lifting-based discrete wavelet transform on GPU," *Signal Processing: Image Communication*, vol. 28, no. 9, pp. 1202–1211, Oct. 2013.

[30] B. Penna, T. Tillo, E. Magli, and G. Olmo, "Transform coding techniques for lossy hyperspectral data compression," *IEEE Trans. Geosci. Remote Sens.*, vol. 45, no. 5, pp. 1408–1421, May 2007.

[31] P. Enfedaque. (2014, Nov.) Implementation of the DWT in a GPU through a register-based strategy. [Online]. Available: https://github.com/PabloEnfedaque/CUDA_DWT_RegisterBased

[32] M. Krotkiewski and M. Dabrowski, "Efficient 3D stencil computations using CUDA," *ELSEVIER Parallel Computing*, vol. 39, pp. 533–548, Oct. 2013.

Pablo Enfedaque is a Ph.D student with the Department of Information and Communications Engineering, Universitat Autònoma de Barcelona, Spain. He received the B.E. degree in computer science and the M.Sc. degree in high performance computing and information theory in 2012 and 2013, respectively, from Universitat Autònoma de Barcelona. His research interests include image coding, high performance computing and parallel architectures.

Francesc Aulí-Llinàs (S'06-M'08-SM'14) is a Ramón y Cajal Fellow with the Department of Information and Communications Engineering, Universitat Autònoma de Barcelona, where he received the B.Sc., B.E. (with honors), M.Sc., and Ph.D. (cum laude) degrees in computer science in 2000, 2002, 2004, and 2006, respectively. He carried out two research stages of one year each with D. Taubman, at the University of New South Wales, and M. Marcellin, at the University of Arizona. In 2013, he was awarded with a distinguished R-Letter given by the IEEE Communications Society for a paper co-authored with M. Marcellin. In 2014, he was recipient of an Intensification Young Investigator Award (I3 program) given by the Spanish Government. He is reviewer for various magazines and symposiums and has authored numerous papers in journals and conferences. His research interests lie in the area of image and video coding, computing, and transmission.

Juan C. Moure received his B.Sc. degree in computer science and his Ph.D. degree in computer architecture from Universitat Autònoma de Barcelona (UAB). Since 2008 he is associate professor with the Computer Architecture and Operating Systems Department at the UAB, where he teaches computer architecture and parallel programming. He has participated in several European and Spanish projects related to high-performance computing. His current research interest focuses on the usage of massively parallel architectures and the application of performance engineering techniques to open research problems in bioinformatics, signal processing, and computer vision. He is reviewer for various magazines and symposiums and has authored numerous papers in journals and conferences.