Accelerating Wavelet Lifting on Graphics Hardware using CUDA

Wladimir J. van der Laan[(a)], Andrei C. Jalba[(b)] and Jos B.T.M. Roerdink[(a)], *Senior Member, IEEE*

Accepted for publication in IEEE Transactions on Parallel and Distributed Systems.

***Abstract*—The Discrete Wavelet Transform (DWT) has a wide range of applications from signal processing to video and image compression. We show that this transform, by means of the lifting scheme, can be performed in a memory and computation efficient way on modern, programmable GPUs, which can be regarded as massively parallel co-processors through NVidia's CUDA compute paradigm. The three main hardware architectures for the 2D DWT (row-column, line-based, block-based) are shown to be unsuitable for a CUDA implementation. Our CUDA-specific design can be regarded as a hybrid method between the row-column and block-based methods. We achieve considerable speedups compared to an *optimized* CPU implementation and earlier non-CUDA based GPU DWT methods, both for 2D images and 3D volume data. Additionally, memory usage can be reduced significantly compared to previous GPU DWT methods. The method is scalable and the fastest GPU implementation among the methods considered. A performance analysis shows that the results of our CUDA-specific design are in close agreement with our theoretical complexity analysis.**

***Index Terms*—Discrete wavelet transform, wavelet lifting, graphics hardware, CUDA.**

I. Introduction

The wavelet transform, originally developed as a tool for the analysis of seismic data, has been applied in areas as diverse as signal processing, video and image coding, compression, data mining and seismic analysis. The theory of wavelets bears a large similarity to Fourier analysis, where a signal is approximated by superposition of sinusoidal functions. A problem, however, is that the sinusoids have an infinite support, which makes Fourier analysis less suitable to approximate sharp transitions in the function or signal. Wavelet analysis overcomes this problem by using small waves, called *wavelets*, which have a compact support. One starts with a wavelet prototype function, called a *basic wavelet* or *mother wavelet*. Then a wavelet basis is constructed by translated and dilated (i.e., rescaled) versions of the basic wavelet. The fundamental idea is to decompose a signal into components with respect to this wavelet basis, and to reconstruct the original signal as a superposition of wavelet basis functions; therefore we speak a *multiresolution analysis*. If the shape of the wavelets resembles that of the data, the wavelet analysis results in a sparse representation of the signal, making wavelets an interesting tool for data compression. This also allows a client-server model of data exchange, where data is first decomposed into different levels of resolution on the server, then progressively transmitted to the client, where the data can be *incrementally* restored as it arrives ('progressive refinement'). This is especially useful when the data sets are very large, as in the case of 3D data visualization [**?**]. For some general background on wavelets, the reader is referred to the books by Daubechies [**?**] or Mallat [**?**].

In the theory of wavelet analysis both continuous and discrete wavelet transforms are defined. If discrete and finite data are used it is appropriate to consider the *Discrete Wavelet Transform* (DWT). Like the discrete Fourier transform (DFT), the DWT is a linear and invertible transform that operates on a data vector whose length is (usually) an integer power of two. The elements of the transformed vector are called *wavelet coefficients*, in analogy of Fourier coefficients in case of the DFT. The DWT and its inverse can be computed by an efficient filter bank algorithm, called Mallat's pyramid algorithm [?]. This algorithm involves repeated downsampling (forward transform) or upsampling (inverse transform) and convolution filtering by the application of high and low pass filters. Its complexity is linear in the number of data elements.

In the construction of so-called *first generation* wavelet bases, which are translates and dilates of a single basic function, Fourier transform techniques played a major role [?]. To deal with situations where the Fourier transform is not applicable, such as wavelets on curves or surfaces, or wavelets for irregularly sampled data, *second generation* wavelets were proposed by Sweldens, based on the so-called *lifting scheme* [**?**]. This provides a flexible and efficient framework for building wavelets. It works entirely in the original time/space domain, and does not involve Fourier transforms.

The basic idea behind the lifting scheme is as follows. It starts with a simple wavelet, and then gradually builds a new wavelet, with improved properties, by adding new basis functions. So the simple wavelet is *lifted* to a new wavelet, and this can be done repeatedly. Alternatively, one can say that a complex wavelet transform is factored into a sequence of simple lifting steps [**?**]. More details on lifting are provided in section III.

Also for first generation wavelets, constructing them by the lifting scheme has a number of advantages [?]. First, it

The authors are with: (a) Institute for Mathematics and Computing Science, University of Groningen, P.O. Box 407, 9700 AK Groningen, The Netherlands; (b) Institute for Mathematics and Computing Science, Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, The Netherlands. E-mail: w.j.van.der.laan@rug.nl, a.c.jalba@tue.nl, j.b.t.m.roerdink@rug.nl

results in a faster implementation of the wavelet transform than the straightforward convolution-based approach by reducing the number of arithmetic operations. Asymptotically for long filters, lifting is twice as fast as the standard algorithm. Second, given the forward transform, the inverse transform can be found in a trivial way. Third, no Fourier transforms are needed. Lastly, it allows a fully in-place calculation of the wavelet transform, so no auxiliary memory is needed. With the generally limited amount of high-speed memory available, and the large quantities of data that have to be processed in multimedia or visualization applications, this is a great advantage. Finally, the lifting scheme represents a universal discrete wavelet transform which involves only *integer* coefficients instead of the usual floating point coefficients [**?**]. Therefore we based our DWT implementation on the lifting scheme.

Custom hardware implementations of the DWT have been developed to meet the computational demands for systems that handle the enormous throughputs in, for example, real-time multimedia processing. However, cost and availability concerns, and the inherent inflexibility of this kind of solutions make it preferable to use a more widespread and general platform. NVidia's G80 architecture [**?**], introduced in 2006 with the GeForce 8800 GPU, provides such a platform. It is a highly parallel computing architecture available for systems ranging from laptops or desktop computers to high-end compute servers. In this paper, we will present a hardware-accelerated DWT algorithm that makes use of the Compute Unified Device Architecture (CUDA) parallel programming model to fully exploit the new features offered by the G80 architecture when compared to traditional GPU programming.

The three main hardware architectures for the 2D DWT, i.e., row-column, line-based, or block-based, turn out to be unsuitable for a CUDA implementation (see Section II). The biggest challenge of fitting wavelet lifting in the SIMD model is that data sharing is, in principle, needed after every lifting step. This makes the division into independent computational blocks difficult, and means that a compromise has to be made between minimizing the amount of data shared with neighbouring blocks (implying more synchronization overhead) and allowing larger data overlap in the computation at the borders (more computation overhead). This challenge is specifically difficult with CUDA, as blocks cannot exchange data at all without returning execution flow to the CPU. Our solution is a sliding window approach which enables us (in the case of separable wavelets) to keep intermediate results longer in shared memory, instead of being written to global memory. Our CUDA-specific design can be regarded as a hybrid method between the row-column and block-based methods. We implemented our methods both for 2D and 3D data, and obtained considerable speedups compared to an *optimized* CPU implementation and earlier non-CUDA based GPU DWT methods. Additionally, memory usage can be reduced significantly compared to previous GPU DWT methods. The method is scalable and the fastest GPU implementation among the methods considered. A performance analysis shows that the results of our CUDA-specific design are in close agreement with our theoretical complexity analysis.

The paper is organized as follows. Section II gives a brief overview of GPU wavelet lifting methods, and previous work on GPU wavelet transforms. In Section III we present the basic theory of wavelet lifting. Section IV first presents an overview of the CUDA programming environment and execution model, introduces some performance considerations for parallel CUDA programs, and gives the details of our wavelet lifting implementation on GPU hardware. Section V presents benchmark results and analyzes the performance of our method. Finally, in Section VI we draw conclusions and discuss future avenues of research.

II. Previous and related work

In [**?**] a method was first proposed that makes use of OpenGL extensions on early non-programmable graphics hardware to perform the convolution and downsampling/upsampling for a 2-D DWT. Later, in [**?**] this was generalized to 3-D using a technique called tileboarding.

Wong *et al.* [**?**] implemented the DWT on programmable graphics hardware with the goal of speeding up JPEG2000 compression. They made the decision not to use wavelet lifting, based on the rationale that, although lifting requires less memory and less computations, it imposes an order of execution which is not fully parallelizable. They assumed that lifting would require more rendering passes, and therefore in the end be slower than the standard approach based on convolution.

However, Tenllado *et al.* [**?**] performed wavelet lifting on conventional graphics hardware by splitting the computation into four passes using fragment shaders. They concluded that a gain of 10-20% could be obtained by using lifting instead of the standard approach based on convolution. Similar to [?], Tenllado *et al.* [**?**] also found that the lifting scheme implemented using shaders requires more rendering steps, due to increased data dependencies. They showed that for shorter wavelets the convolution-based approach yields a speedup of 50-100% compared to lifting. However, for larger wavelets, on large images, the lifting scheme becomes 10-20% faster. A limitation of both [?] and [?] is that the methods are strictly focused on 2-D. It is uncertain whether, and if so, how they extend to three or more dimensions.

All previous methods are limited by the need to map the algorithms to graphics operations, constraining the kind of computations and memory accesses they could make use of. As we will show below, new advances in GPU programming allow us to do in-place transforms in a *single pass*, using intermediate fast shared memory.

Wavelet lifting on general parallel architectures was studied extensively in [**?**] for processor networks with large communications latencies. A technique called *boundary postprocessing* was introduced that limits the amount of data sharing between processors working on individual blocks of data. This is similar to the technique we will use. More than in previous generations of graphics cards, general parallel programming paradigms can now be applied when designing GPU algorithms.

The three main hardware architectures for the 2D DWT are *row-column* (RC), *line-based* (LB) and *block-based* (BB), see

for example [**?**], [**?**], [**?**], [**?**], and all three schemes are based on wavelet lifting. The simplest one is RC, which applies a separate 1D DWT in both the horizontal and vertical directions for a given number of lifting levels. Although this architecture provides the simplest control path (thus being the cheapest for a hardware realization), its major disadvantage is the lack of locality due to the use of large off-chip memory (i.e., the image memory), thus decreasing performance. Contrary to RC, both LB and BB involve a local memory that operates as a cache, thus increasing bandwidth utilization (throughput). On FPGA architectures, it was found [?] that the best instruction throughput is obtained by the LB method, followed by the RC and BB schemes which show comparable performances. As expected, both the LB and BB schemes have similar bandwidth requirements, which are at least two times smaller than that of RC. Theoretical results [?], [?] show that this holds as well for ASIC architectures. Thus, LB is the best choice with respect to overall performance, for a hardware implementation.

Unfortunately, a CUDA realization of LB is impossible for all but the shortest wavelets (e.g., the Haar wavelet), due to the relatively large cache memory required. For example, the cache memory for the Deslauriers-Dubuc $(13, 7)$ wavelet should accommodate six rows of the original image (i.e., 22.5 KB for two-byte word data and HD resolutions), well in excess of the maximum amount of 16 KB of shared memory available per multi-processor, see Section IV-C. As an efficient implementation of BB requires similar amounts of cache memory, this choice is again not possible. Thus, the only feasible strategy remains RC. However, we show in Section V that even an improved (using cache memory) RC strategy is not optimal for a CUDA implementation. Nevertheless, our CUDA-specific design can be regarded as a hybrid method between RC and BB, which also has an optimal access pattern to the slow global memory (see Section IV-A2).

III. Wavelet lifting

As explained in the introduction, lifting is a very flexible framework to construct wavelets with desired properties. When applied to first generation wavelets, lifting can be considered as a reorganization of the computations leading to increased speed and more efficient memory usage. In this section we explain in more detail how this process works. First we discuss the traditional wavelet transform computation by subband filtering and then outline the idea of wavelet lifting.

A. Wavelet transform by subband filtering

The main idea of (first generation) wavelet decomposition for finite 1-D signals is to start from a signal $c^0 = (c^0_0, c^0_1, \ldots, c^0_{N-1})$, with N samples (we assume that N is a power of 2). Then we apply convolution filtering of c^0 by a low pass analysis filter H and downsample the result by a factor of 2 to get an "approximation" signal (or "band") c^1 of length $N/2$, i.e., half the initial length. Similarly, we apply convolution filtering of c^0 by a high pass analysis filter G, followed by downsampling, to get a detail signal (or "band") d_1. Then we continue with c^1 and repeat the same steps, to get further approximation and detail signals c^2 and d^2 of length $N/4$. This process is continued a number of times, say J. Here J is called the number of *levels* or *stages* of the decomposition. The explicit decomposition equations for the individual signal coefficients are:

$$c^{j+1}_k = \sum_n h_{n-2k}\, c^j_n, \;\; d^{j+1}_k = \sum_n g_{n-2k}\, c^j_n$$

where $\{h_n\}$ and $\{g_n\}$ are the coefficients of the filters H and G. Note that only the approximation bands are successively filtered, the detail bands are left "as is".

This process is presented graphically in Fig. 1, where the symbol $\downarrow_2$ (enclosed by a circle) indicates downsampling by a factor of 2. This means that after the decomposition the initial

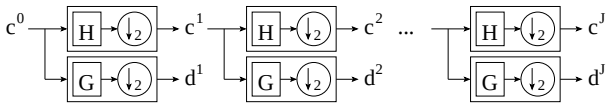

Figure 1. Structure of the forward wavelet transform with J stages: recursively split a signal c^0 into approximation bands c^j and detail bands d^j.

data vector c^0 is represented by one approximation band c^J and J detail bands d^1, d^2, …, d^J. The total length of these approximation and detail bands is equal to the length of the input signal c^0.

Signal reconstruction is performed by the *inverse* wavelet transform: first upsample the approximation and detail bands at the coarsest level J, then apply synthesis filters $\tilde{H}$ and $\tilde{G}$ to these, and add the resulting bands. (In the case of orthonormal filters, such as the Haar basis, the synthesis filters are essentially equal to the analysis filters.) Again this is done recursively. This process is presented graphically in Fig. 2, where the symbol $\uparrow_2$ indicates upsampling by a factor of 2.

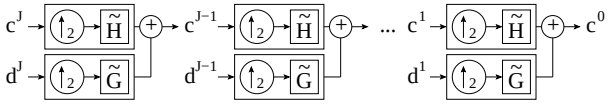

Figure 2. Structure of the inverse wavelet transform with J stages: recursively upsample, filter and add approximation signals c^j and detail signals d^j.

B. Wavelet transform by lifting

Lifting consists of four steps: split, predict, update, and scale, see Fig. 3 (left).

1) **Split**: this step splits a signal (of even length) into two sets of coefficients, those with even and those with odd index, indicated by even^{j+1} and odd^{j+1}. This is called the *lazy wavelet transform*.
2) **Predict lifting step**: as the even and odd coefficients are correlated, we can predict one from the other. More specifically, a prediction operator P is applied to the even coefficients and the result is subtracted from the odd coefficients to get the detail signal d^{j+1}:

$$d^{j+1} = \mathrm{odd}^{j+1} - P(\mathrm{even}^{j+1}) \qquad (1)$$

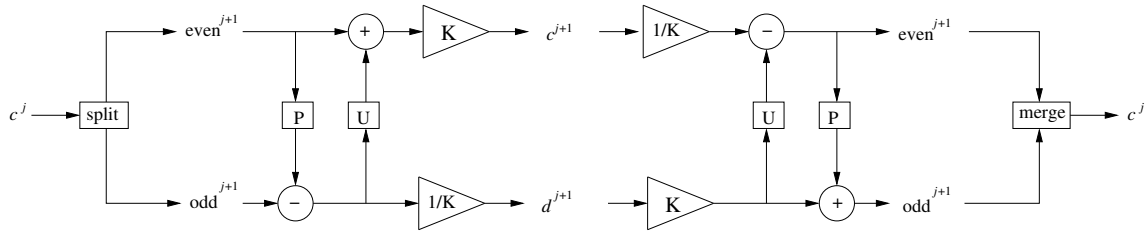

Figure 3. Classical lifting scheme (one stage only). Left part: forward lifting. Right part: inverse lifting. Here "split" is the trivial wavelet transform, "merge" is the opposite operation, P is the prediction step, U the update step, and K the scaling factor.

3) **Update lifting step**: similarly, an update operator U is applied to the odd coefficients and added to the even coefficients to define c^{j+1}:

$$c^{j+1} = \text{even}^{j+1} + U(d^{j+1}) \tag{2}$$

4) **Scale**: to ensure normalization, the approximation band c^{j+1} is scaled by a factor of K, and the detail band d^{j+1} by a factor of $1/K$.

Sometimes the scaling step is omitted; in that case we speak of an *unnormalized* transform.

A remarkable feature of the lifting technique is that the inverse transform can be found trivially. This is done by "inverting" the wiring diagram, see Fig. 3 (right): undo the scaling, undo the update step ($\text{even}^{j+1} = c^{j+1} - U(d^{j+1})$), undo the predict step ($\text{odd}^{j+1} = d^{j+1} + P(\text{even}^{j+1})$), and merge the even and odd samples. Note that this scheme does not require the operators P and U to be invertible: nowhere does the inverse of P or U occur, only the roles of addition and subtraction are interchanged. For a multistage transform the process is repeatedly applied to the approximation bands, until a desired number of decomposition levels is reached. In the same way as discussed in section III-A, the total length of the decomposition bands equals that of the initial signal. As an illustration, we give in Table I the explicit equations for one stage of the forward wavelet transform by the (unnormalized) Le Gall $(5, 3)$ filter, both by subband filtering and lifting (*in-place* computation). It is easily verified that both schemes give identical results for the computed approximation and detail coefficients.

The process above can be extended by including more predict and/or update steps in the wiring diagram [?]. In fact, any wavelet transform with finite filters can be decomposed into a sequence of lifting steps [?]. In practice, lifting steps are chosen to improve the decomposition, for example, by producing a lifted transform with better decorrelation properties or higher smoothness of the resulting wavelet basis functions.

Wavelet lifting has two properties which are very important for a GPU implementation. First, it allows a fully *in-place* calculation of the wavelet transform, so no auxiliary memory is needed. Second, the lifting scheme can be modified to a transform that maps *integers* to *integers* [?]. This is achieved by rounding the result of the P and U functions. This makes the predict and update operations nonlinear, but this does not affect the invertibility of the lifting transform. Integer-to-integer wavelet transforms are especially useful when the input data consists of integer samples. These schemes can avoid quantization, which is an attractive property for lossless data compression.

For many wavelets of interest, the coefficients of the predict and update steps (before truncation) are of the form $z/2^n$, with z integer and n a positive integer. In that case one can implement all lifting steps (apart from normalization) by integer operations: integer addition and multiplication, and integer division by powers of 2 (bit-shifting).

Table I
FORWARD WAVELET TRANSFORM (ONE STAGE ONLY) BY THE (UNNORMALIZED) LE GALL $(5, 3)$ FILTER.

Subband filtering	$c_k^{j+1} = \frac{1}{8}\left(-c_{2k-2}^j + 2c_{2k-1}^j + 6c_{2k}^j + 2c_{2k+1}^j - c_{2k+2}^j\right)$
	$d_k^{j+1} = \frac{1}{2}(-c_{2k}^j + 2c_{2k+1}^j - c_{2k+2}^j)$
Lifting	Split: $c_k^{j+1} \leftarrow c_{2k}^j, \quad d_k^{j+1} \leftarrow c_{2k+1}^j$
	Predict: $d_k^{j+1} \leftarrow d_k^{j+1} - \frac{1}{2}(c_k^{j+1} + c_{k+1}^{j+1})$
	Update: $c_k^{j+1} \leftarrow c_k^{j+1} + \frac{1}{4}(d_{k-1}^{j+1} + d_k^{j+1})$

IV. WAVELET LIFTING ON GPUS USING CUDA

A. CUDA overview

In recent years, GPUs have become increasingly powerful and more programmable. This combination has led to the use of the GPU as the main computation device for diverse applications, such as physics simulations, neural networks, image compression and even database sorting. The GPU has moved from being used solely for graphical tasks to a fully-fledged parallel co-processor. Until recently, General Purpose GPU (GPGPU) applications, even though not concerned with graphics rendering, did use the rendering paradigm. In the most common scenario, textured quadrilaterals were rendered to a texture, with a fragment shader performing the computation for each fragment.

With their G80 series of graphics processors, NVidia introduced a programming environment called CUDA [?]. It is an API that allows the GPU to be programmed through more traditional means: a C-like language (with some C++-features such as templates) and compiler. The GPU programs, now called *kernels* instead of shaders, are invoked through procedure calls instead of rendering commands. This allows the programmer to focus on the main program structure, instead of details like color clamping, vertex coordinates and pixel offsets.

In addition to this generalization, CUDA also adds some features that are missing in shader languages: random access to memory, fast integer arithmetic, bitwise operations, and shared memory. The usage of CUDA does not add any overhead, as it is a native interface to the hardware, and not an abstraction layer.

1) Execution model: The CUDA execution model is quite different from that of CPUs, and also different from that of older GPUs. CUDA broadly follows the data-parallel model of computation [?]. The CPU invokes the GPU by calling a *kernel*, which is a special C-function.

The lowest level of parallelism is formed by *threads*. A thread is a single scalar execution unit, and a large number of threads can run in parallel. The thread can be compared to a fragment in traditional GPU programming. These threads are organized in *blocks*, and the threads of each block can cooperate efficiently by sharing data through fast shared memory. It is also possible to place synchronization points (barriers) to coordinate operations closely, as these will synchronize the control flow between all threads within a block. The Single Instruction Multiple Data (SIMD) aspect of CUDA is that the highest performance is realized if all threads within a *warp* of 32 consecutive threads take the same execution path. If flow control is used within such a *warp*, and the threads take different paths, they have to wait for each other. This is called *divergence*.

The highest level, which encompasses the entire kernel invocation, is called the *grid*. The grid consists of blocks that execute in parallel, if multiprocessors are available, or sequentially if this condition is not met. A limitation of CUDA is that blocks within a grid cannot communicate with each other, and this is unlikely to change as independent blocks are a means to scalability.

2) Memory layout: The CUDA architecture gives access to several kinds of memory, each tuned for a specific purpose. The largest chunk of memory consists of the *global* memory, also known as device memory. This memory is linearly addressable, and can be read and written at any position in any order (random access) from the device. No caching is done in G80, however there is limited caching in the newest generation (GT200) as part of the shared memory can be configured as automatic cache. This means that optimizing access patterns is up to the programmer. Global memory is also used for communication with the CPU, which can read and write using API calls. *Registers* are limited per-thread memory locations with very fast access, which are used for local storage. *Shared memory* is a limited per-block chunk of memory which is used for communication between threads in a block. Variables are marked to be in shared memory using a specifier. Shared memory can be almost as fast as registers, provided that bank conflicts are avoided. *Texture memory* is a special case of device memory which is cached for locality. Textures in CUDA work the same as in traditional rendering, and support several addressing modes and filtering methods. *Constant memory* is cached memory that can be written by the CPU and read by the GPU. Once a constant is in the constant cache, subsequent reads are as fast as register access.

The device is capable of reading 32-bit, 64-bit, or 128-bit words from global memory into registers in a single instruction. When access to device memory is properly distributed over threads, it is compiled into 128-bit load instructions instead of 32-bit load instructions. The consecutive memory locations must be simultaneously accessed by the threads. This is called *memory access coalescing* [?], and it represents one of the most important optimizations in CUDA. We will confirm the huge difference in memory throughput between coalesced and non-coalesced access in our results.

B. Performance considerations for parallel CUDA programs (kernels)

Let us first define some metrics which we use later to analyze our results in Section V-C below.

1) Total execution time: Assume that a CUDA kernel performs computations on N data values, and organizes the CUDA 'execution model' as follows. Let T denote the number of threads in a block, W the number of threads in a warp, i.e., $W = 32$ for G80 GPUs, and B denote the number of thread blocks. Further, assume that the number of multiprocessors (device specific) is M, and that NVidia's occupancy calculator [**?**] indicates that k blocks can be assigned to one multiprocessor (MP); k is program specific and represents the total number of threads for which (re)scheduling costs are zero, i.e., context switching is done with no extra overhead. Given that the amount of resources per MP is fixed (and small), k simply indicates the occupancy of the resources for the given kernel. With this notation, the number of blocks assigned to one MP is given by $b = B/M$. Since in general k is smaller than b, it follows that the number α of times k blocks are rescheduled is $\alpha = \left\lceil \frac{B}{Mk} \right\rceil$.

Since each MP has 8 stream processors, a warp has 32 threads and there is no overhead when switching among the warp threads, it follows that each warp thread can execute one (arithmetic) instruction in four clock cycles. Thus, an estimate of the asymptotic time required by a CUDA kernel to execute n instructions over all available resources of a GPU, which also includes scheduling overhead, is given by

$$T_e = \frac{4\,n}{K}\,\frac{T}{W}\,\alpha\,k\,l_s, \tag{3}$$

where K is the clock frequency and l_s is the latency introduced by the scheduler of each MP.

The second component of the total execution time is given by the time T_m required to transfer N bytes from global memory to fast registers and shared memory. If thread transfers of m bytes can be coalesced, given that a memory transaction is done per half-warp, it follows that the transfer time T_m is

$$T_m = \frac{2\,N}{W\,m\,M}\,l_m, \tag{4}$$

where l_m is the latency (in clock cycles) of a memory access. As indicated by NVidia [**?**], reported by others [**?**] and confirmed by us, the latency of a *non-cached* access can be as large as $400-600$ clock cycles. Compared to 24 cycle latency for accessing the shared memory, it means that transfers from global memory should be minimized. Note that for *cached* accesses the latency becomes about $250-350$ cycles.

One way to effectively address the relatively expensive memory-transfer operations is by using fine-grained *thread parallelism*. For instance, 24 cycle latency can be hidden by running 6 warps (192 threads) per MP. To hide even larger latencies, the number of threads should be raised (thus, increasing the degree of parallelism) up to a maximum of 768 threads per MP supported by the G80 architecture. However, increasing the number of threads while maintaining the size N of the problem fixed, implies that each thread has to perform less work. In doing so, one should still recall (i) the paramount importance of coalescing memory transactions and (ii) the Flops/word ratio, i.e., peak Gflop/s rate divided by global memory bandwidth in words [?], for a specific GPU. Thus, threads should not execute too few operations nor transfer too little data, such that memory transfers cannot be coalesced. To summarize, a tradeoff should be found between increased thread parallelism, suggesting more threads to hide memory-transfer latencies on the one hand, and on the other, memory coalescing and maintaining a specific Flops/word ratio, indicating fewer threads.

Let us assume that for a given kernel, one MP has an occupancy of $k\,T$ threads. Further, if the kernel has a ratio $r \in (0,1)$ of arithmetic to arithmetic-and-memory-transfer instructions, and assuming a *round-robin* scheduling policy, then the *reduction* of memory-transfer latency due to latency hiding is

$$l_h = \sum_{i \geq 0} \left\lfloor \frac{k\,T}{8}\, r^i \right\rfloor. \tag{5}$$

For example, assume that $r = 0.5$, i.e., there are as many arithmetic instructions (flops) as memory transfers, and assume that $k\,T = 768$, i.e., each MP is fully occupied. The scheduler starts by assigning 8 threads to a MP. Since $r = 0.5$ chances are that 4 threads execute each a memory transfer instruction while the others execute one arithmetic operation. After one cycle, those 4 threads executing memory transfers are still asleep for at least 350 cycles, while the others just finished executing the flop and are put to sleep too. The scheduler assigns now another 8 threads, which again can execute either a memory transfer or a flop, with the same probability, and repeats the whole process. Counting the number of cycles in which 4 threads executed flops, reveals a number of 190 cycles, so that the latency is decreased in this way to just $l_m = 350 - 190 = 160$ cycles. In the general case, for a given r and occupancy, we postulate that formula (5) applies.

The remaining component of the total GPU time for a kernel is given by the synchronization time. To estimate this component, we proceed as follows. Assume all active threads (i.e., $k\,T \leq 768$) are about to execute a flop, after which they have to wait on a synchronization point (i.e., on a barrier). Then, assuming again a round-robin scheduling policy and reasoning similar as for Eq. (3), the idle time spent waiting on the barrier is

$$T_s = \frac{T_e}{n} = \frac{4}{K}\,\frac{T}{W}\,\alpha\,k\,l_s. \tag{6}$$

This agrees with NVidia's remark that, if within a warp thread divergence is minimal, then waiting on a synchronization barrier requires only four cycles [?]. Note that the expression for T_s from Eq. (6) represents the minimum synchronization time, as threads were assumed to execute (fast) flops. In the worst case scenario – at least one active thread has just started before the synchronization point, a slow global-memory transaction – this estimate has to be multiplied by a factor of about $500/4 = 125$ (the latency of a non-cached access divided by 4 threads).

To summarize, we estimate the total execution time T_t as $T_t = T_e + T_m + T_s$.

2) Instruction throughput: Assuming that a CUDA kernel performs n flops in a number c of cycles, then the *estimate* of the asymptotic Gflop/s rate is $G_e = \frac{32\,M\,n\,K}{c}$, whereas the *measured* Gflop/s rate is $G_m = \frac{n\,N}{T_t}$; here K is the clock rate and T_t the (measured) total execution time. For the 8800 GTX GPU the peak instruction throughput using register-to-register MAD instructions is about 338 Gflop/s and drops to 230 Gflop/s when using transfers in/from shared memory [?].

3) Memory bandwidth: Another factor which should be taken into account when developing CUDA kernels is the memory bandwidth, $M_b = \frac{N}{T_t}$. For example, parallel *reduction* has very low arithmetic intensity, i.e., 1 flop per loaded element, which makes it bandwidth-optimal. Thus, when implementing a parallel reduction in CUDA, one should strive for attaining peak bandwidth. On the contrary, if the problem at hand is matrix multiplication (a trivial parallel computation, with little synchronization overhead), one should optimize for peak throughput. For the 8800 GTX GPU the pin-bandwidth is 86 GB/s.

4) Complexity: With coalesced accesses the number of bytes retrieved with one memory request (and thus one latency) is maximized. In particular, coalescing reduces l_m (through l_h from Eq. 5) by a factor of about two. Hence one can safely assume that $l_m/(2\,W) \to 0$. It follows that the total execution time satisfies

$$T_t \sim 4\,n\,\frac{N}{W\,M\,D}, \tag{7}$$

where n is the number of instructions of a given CUDA kernel, N is the problem size, D is the problem size per thread, and $\sim$ means that both left and right-hand side quantities have the same order of magnitude.

The *efficiency* of a parallel algorithm is defined as

$$E = \frac{T_S}{C} = \frac{T_S}{M\,T_t}, \tag{8}$$

where T_S is the execution time of the (fastest) sequential algorithm, and $C = M\,T_t$ is the *cost* of the parallel algorithm. A parallel algorithm is called *cost efficient* (or cost optimal) if its cost is proportional to T_S. Let us assume $T_S \sim n_s\,N$, where n_s is the number of instructions for computing one data element and N denotes the problem size. Then, the efficiency becomes

$$E \sim \frac{n_s\,W\,D}{4\,n}. \tag{9}$$

Thus, according to our metric above, for a given problem, *any* CUDA kernel which (i) uses *coalesced* memory transfers (i.e., $l_m/(2\,W) \to 0$ is enforced), (ii) avoids thread divergence (so that our T_s estimate from Eq. 6 applies), (iii) minimizes transfers from global memory, and (iv) has an instruction count

n proportional to $(n_s\, W\, D)$ is *cost efficient*. Of course, the smaller n is, the more efficient the kernel becomes.

C. Parallel wavelet lifting

Earlier parallel methods for wavelet lifting [?] assumed an MPI architecture with processors that have their own memory space. However, the CUDA architecture is different. Each processor has its own shared memory area of 16 KB, which is not enough to store a significant part of the dataset. As explained above, each processor is allocated a number of threads that run in parallel and can synchronize. The processors have no way to synchronize with each other, beyond their invocation by the host.

This means that data parallelism has to be used, and moreover, the dataset has to be split into parts that can be processed as independently as possible, so that each chunk of data can be allocated to a processor. For wavelet lifting, except for the Haar [?] transform, this task is not trivial, as the implied data re-use in lifting also requires the coefficients just outside the delimited block to be updated. This could be solved by duplicating part of the data in each processor. Wavelet bases with a large support will however need more data duplication. If we want to do a multilevel transform, each level of lifting doubles the amount of duplicated work and data. With the limited amount of shared memory available in CUDA, this is not a feasible solution.

As kernel invocations introduce some overhead each time, we should also try to do as much work within one kernel as possible, so that the occupancy of the GPU is maximized. The sliding window approach enables us (in the case of separable wavelets) to keep intermediate results longer in shared memory, instead of being written to global memory.

D. Separable wavelets

For separable wavelet bases in 2-D it is possible to split the operation into a horizontal and a vertical filtering step. For each filter level, a horizontal pass performs a 1-D transform on each row, while a vertical pass computes a 1-D transform on each column. This lends itself to easy parallelization: each row can be handled in parallel during the horizontal pass, and then each column can be handled in parallel during the vertical pass. In CUDA this implies the use of two kernels, one for each pass. The simple solution would be to have each block process a row with the horizontal kernel, while in the vertical step each block processes a column. Each thread within these blocks can then filter an element. We will discuss better, specific algorithms for both passes in the upcoming subsections.

E. Horizontal pass

The simple approach mentioned in the previous subsection works very well for the horizontal pass. Each block starts by reading a line into shared memory using so-called *coalesced reads* from device memory, executes the lifting steps in-place in fast shared memory, and writes back the result using *coalesced writes*. This amounts to the following steps:

1) Read a row from device memory into shared memory.
2) Duplicate border elements (implement boundary condition).
3) Do a 1-D lifting step on the elements in shared memory.
4) Repeat steps 2 and 3 for each lifting step of the transform.
5) Write back the row to device memory.

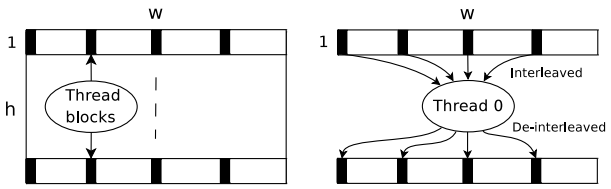

Figure 4. Horizontal lifting step. h thread blocks are created, each containing T threads; each thread performs computations on $N/(hT) = w/T$ data. Black quads illustrate input for the thread with id 0. Here w and h are the the dimensions of the input and $N = w \cdot h$.

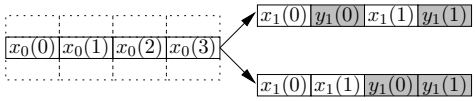

Figure 5. Wavelet lifting for a row of data, representing the result in interleaved (top) and de-interleaved (bottom) form. Here x_i and y_i are the approximation and detail bands at level i.

As each step is dependent on the output in shared memory of the previous step, the threads within the block have to be synchronized every time before the next step can start. This ensures that the previous step did finish and wrote back its work. Fig. 4 shows the configuration of the CUDA execution model for the horizontal step. Without loss of generality, assume that $N = w \cdot h$ integers are lifted at level i. Note that, if the lifting level $i = 0$, then w and h are the dimensions of the input image. For this step, a number $B = h$ of thread blocks are used, with T threads per block. Thus, each thread performs computations on w/T integers. In the figure, black quads illustrate locations which are processed by the thread with id 0. Neither the number nor the positions of these quads need to correspond to the actual number and positions of locations where computations are performed, i.e., they are solely used for illustration purposes.

By reorganizing the coefficients [**?**] we can achieve higher efficiency for successive levels after the first transformation. If the approximation and detail coefficients are written back in interleaved form, as is usually the case with wavelet lifting, the reading step for the next level will have to read the approximation coefficients of the previous level in interleaved form. These reads cannot be coalesced, resulting in low memory performance. To still be able to coalesce, one writes the approximation and detail coefficients back to separate halves of the memory. This will result in a somewhat different memory layout for subbands (Fig. 5) but this could be reorganized if needed. Many compression algorithms require the coefficients stored per subband anyhow, in which case this organization is advantageous.

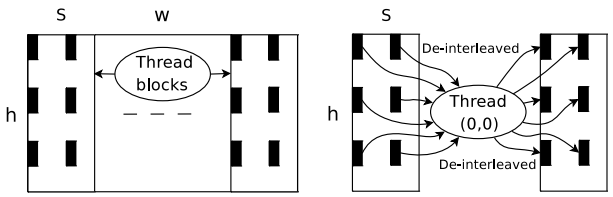

Figure 6. Vertical lifting step. w/S blocks are created, each containing $T = V_x \cdot V_y$ threads; each thread performs computations on $S/V_x \times h/V_y$ data. Black quads illustrate input for the thread with id $(0, 0)$, whereas vertical lines depict boundaries between image-high slabs.

F. Vertical pass

The vertical pass is more involved. Of course it is possible to use the same strategy as for the horizontal pass, substituting rows for columns. But this is far from efficient. Reading a column from the data would amount to reading one value per row. As only consecutive reads can be coalesced into one read, these are all performed individually. The processing steps would be the same as for the horizontal pass, after which writing back is again very inefficient.

We can gain a 10 times speedup by using coalesced memory access. Instead of having each block process a column, we make each block process multiple columns by dividing the image into vertical "slabs", see Fig. 6. Within a block, threads are organized into a 2D grid of size $V_x \times V_y$, instead of a 1D one, as in the horizontal step. The number S of columns in each slab is a multiple of V_x such that the resulting number of slab rows can still be coalesced, and has the height of the image. Each thread block processes one of the slabs, i.e., $S/V_x \times h/V_y$ data. Using this organization, a thread can do a coalesced read from each row within a slab, do filtering in shared memory, and do a coalesced write to each slab row.

Another problem arises here, namely that the shared memory in CUDA is not large enough to store all columns for any sizable dataset. This means that we cannot read and process the entire slab at once. The solution that we found is to use a sliding window within each slab, see Fig. 7(a). This window needs to have dimensions so that each thread in the block can transform a signal element, and additional space to make sure that the support of the wavelet does not exceed the top or bottom of the window. To determine the size of the window needed, how much to advance, and at which offset to start, we need to look at the support of each of the lifting steps.

In Fig. 7(a), `height` is the height of the working area. As each step updates either odd or even rows within a slab, each row of threads updates one row in each lifting step. Therefore, a good choice is to set it to two times the number of threads in the vertical direction. Similarly, `width` should be a multiple of the number of threads in the horizontal direction, and the size of a row should be a multiple of the coalescable size. In the figure, rows in the `top` area have been fully computed, while rows in the overlap area still need to go through at least one lifting step. The rows in the working area need to go through all lifting steps, whilst rows in the bottom area are untouched except as border extension. The sizes of `overlap`, `top` and `bottom` depend on the chosen wavelet. We will elaborate on this later.

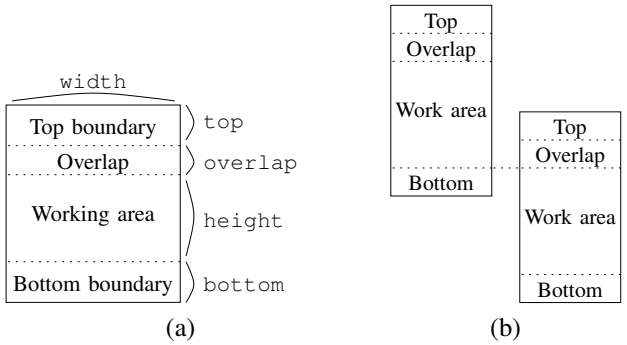

Figure 7. (a): The sliding window used during the vertical pass for separable wavelets. (b): Advancing the sliding window: the next window is aligned at the bottom of the previous one, taking the overlap area into account.

1) The algorithm: Algorithm 1 shows the steps for the vertical lifting pass. Three basic operations are used: **read** copies rows from device memory into shared memory, **write** copies rows from shared memory back to device memory, and **copy** transfers rows from shared memory to another place in shared memory. The shared memory window is used as a cache, and to manage this we keep a read and a write pointer. The read pointer *inrow* indicates where to read from, the write pointer *outrow* indicates where to write back. After reading, we advance the read pointer, after writing we advance the write pointer. Both are initialized to the top of the slab at the beginning of the kernel (line 1 and 2 of Algorithm 1).

The first block has to be handled differently because we need to take the boundary conditions into account. So initially, rows are copied from the beginning of the slab to shared memory, filling it from a certain offset to the end (line 5). Next we apply a vertical wavelet lifting transform (**transformTop**, line 7) to the rows in shared memory (it may be required to leave some rows at the end untouched for some of the lifting steps, depending on their support; we will elaborate on this in the next section). After this we write back the fully transformed rows from shared memory to device memory (line 8). Then, for each block, the bottom part of the shared memory is copied to the top part (Fig. 7(b)), in order to align the next window at the bottom of the previous one, taking the overlap area into account (line 11). The rest of the shared memory is filled again by copying rows from the current read pointer of the slab (line 12).

Further, we apply a vertical wavelet lifting transform (**transformBlock**, line 14) to the rows in the working area. This does not need to take boundary conditions into account as the top and bottom are handled specifically with **transformTop** and **transformBottom**. Then, `height` rows are copied from shared memory row `top` to the current write pointer (line 15). This process is repeated until we have written back the entire slab, except for the last leftover part. When finishing up (line 20), we have to be careful to satisfy the bottom boundary condition.

2) Example: We will discuss the Deslauriers-Dubuc $(13, 7)$ wavelet as an example [**?**]. This example was chosen because it represents a non-trivial, but still compact enough case of

Algorithm 1 The sliding window algorithm for the vertical wavelet lifting transform (see section IV-F). Here `top`, `overlap`, `height`, `bottom` are the length parameters of the sliding window (see Fig. 7), and h is the number of rows of the dataset. The pointer *inrow* indicates where to read from, the pointer *outrow* indicates where to write back.

```
inrow ← 0 {initialize read pointer}
outrow ← 0 {initialize write pointer}
windows ← (h − height − bottom)/height {number of times window fits in slab}
leftover ← (h − height − bottom)%height {remainder}
read(height + bottom from row inrow to row top + overlap) {copy from global to shared memory}
inrow ← inrow+height+bottom {advance read pointer}
transformTop() {apply vertical wavelet lifting to rows in shared memory}
write(height − overlap from row top + overlap to row outrow) {write transformed rows back to global memory}
outrow ← outrow + height − overlap {advance write pointer}
for i = 1 to windows do {advance sliding window through slab and repeat above steps}
    copy(top+overlap+bottom from row height to row 0)
    read(height from row inrow to row top + overlap + bottom)
    inrow ← inrow + height
    transformBlock() {vertical wavelet lifting}
    write(height from row top to row outrow)
    outrow ← outrow + height
end for
copy(top + overlap + bottom from row height to row 0)
read(leftover from row inrow to row top + overlap + bottom)
transformBottom() {satisfy bottom boundary condition}
write(leftover+overlap+bottom from row top to row outrow)
```

the algorithm, that we can go through step by step. The filter weights for the two lifting steps of this transform are shown in Table II. Both the prediction and update steps depend on two coefficients before and after the signal element to be computed. Fig. 8 shows an example of the performed computations. For this example, we choose `top` $= 3$, `overlap` $= 2$, `height` $= 8$ and `bottom` $= 3$. This is a toy example, as in practice `height` will be much larger when compared to the other parameters.

Starting with the first window at the start of the dataset, step 1 (first column), the odd rows of the working area (offset $1, 3, 5, 7$) are lifted. The lifted rows are marked with a cross, and the rows they depend on are marked with a bullet. In step 2 (second column) the even rows are lifted. Again, the lifted rows are marked with a cross, and the dependencies are marked with a bullet. As the second step is dependent on the first, we cannot lift any rows that are dependent on values that were not yet calculated in the last step. In Fig. 8, this would be the case for row 6: this row requires data in rows 3, 5, 7 and 9, but row 9 is not yet available.

Here the `overlap` region of rows comes in. As row 6 of the window is not yet fully transformed, we cannot write it back to device memory yet. So we write everything up to this row back, copy the overlapping area to the top, and proceed with the second window. In the second window, we again start with step 1. The odd rows are lifted, except for the first one (offset 7) which was already computed, i.e., rows 9, 11, 13 and 15 are lifted. Then, in step 2 we start at row 6, i.e., three rows before the first step (row 9), but we do lift four rows.

After this we can write the top 8 rows back to device memory, and begin with the next window in exactly the same way. We repeat this until the entire dataset is transformed. By letting the second lifting step lag behind the first, one can do the same number of operations in each, making optimal use of the thread matrix (which should have a height of 4 in this case).

All separable wavelet lifting transforms, even those with more than two lifting steps, or with differently sized supports, can be computed in the same way. The transform can be inverted by running the steps in reverse order, and flipping the signs of the filter weights.

Table II
FILTER WEIGHTS OF THE TWO LIFTING STEPS FOR THE DESLAURIERS-DUBUC $(13, 7)$ [?] WAVELET. THE CURRENT ELEMENT BEING UPDATED IS MARKED WITH •.

Offset	Prediction	Update
-3		$-\frac{1}{16}$
-2	$\frac{1}{16}$	
-1		$\frac{9}{16}$
0	$-\frac{9}{16}$	•
1	•	$\frac{9}{16}$
2	$-\frac{9}{16}$	
3		$-\frac{1}{16}$
4	$\frac{1}{16}$	

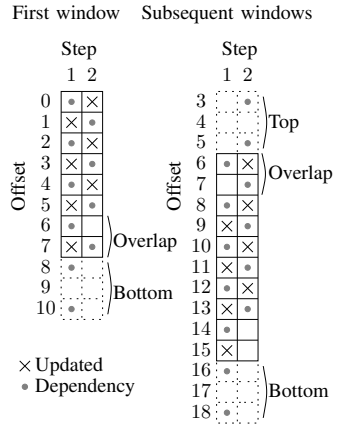

Figure 8. The vertical pass for the Deslauriers-Dubuc $(13, 7)$ [?] wavelet. Lifted rows in each step are marked with a cross, dependent rows are marked with a bullet.

G. 3-D and higher dimensions

The reason that the horizontal and vertical passes are asymmetric is because of the coalescing requirement for reads and writes. In the horizontal case, an entire line of the data-set

could be processed at a time. In the vertical case the dataset was horizontally split into image-high slabs. This allowed the slabs to be treated independently and processed using a sliding window algorithm that uses coalesced reads and writes to access lines of the slab. A consecutive, horizontal span of values is stored at consecutive addresses in memory. This does not extend similarly to vertical spans of values, these will be separated by an offset at least the width of the image, known as the *row pitch*. As a slab is a rectangular region of the image of a certain width that spans the height of the image, it will be represented in memory by an array of consecutive spans of values, each separated by the *row pitch*.

When adding an extra dimension, let us say z, the volume is stored as an array of slices. In a span of values oriented along this dimension, each value is separated in memory by an offset that we call the *slice pitch*. By orienting the slabs in the xz-plane instead of the xy-plane, and thus using the *slice pitch* instead of the *row pitch* as offset between consecutive spans of values, the same algorithm as in the vertical case can be used to do a lifting transform along this dimension. To verify our claim, we implemented the method just described, and report results in section V-B7. More than three dimensions can be handled similarly, by orienting the slabs in the $D_i x$ plane (where D_i is dimension i) and using the pitch in that dimension instead of the row pitch.

V. Results

We first present a broad collection of experimental results. This is followed by a performance analysis which provides insight in the results obtained, and also shows that the design choices we made closely match our theoretical predictions.

The benchmarks in this section were run on a machine with a AMD Athlon 64 X2 Dual Core Processor 5200+ and a NVidia GeForce 8800 GTX 768MB graphics card, using CUDA version 2.1 for the CUDA programs. All reported timings exclude the time needed for reading and writing images or volumes from and to disc (both for the CPU and GPU versions).

A. Wavelet filters used for benchmarking

The wavelet filters that we used in our benchmarks are integer-to-integer versions (unnormalized) of the Haar [?], Deslauriers-Dubuc $(9, 7)$ [?], Deslauriers-Dubuc $(13, 7)$ [?], Le Gall $(5, 3)$ [**?**], (integer approximation of) Daubechies $(9, 7)$ [?] and the Fidelity wavelet – a custom wavelet with a large support [**?**]. In the filter naming convention (m, n), m refers to the length of the analysis low-pass and n to the analysis high-pass filters in the conventional wavelet subband filtering model, in which a convolution is applied before subsampling. They do not reflect the length of the filters used in the lifting steps, which operate in the subsampled domain. The implementation only involves integer addition and multiplication, and integer division by powers of 2 (bit-shifting), cf. section III-B. The coefficients of the lifting filters can be found in [?].

B. Experimental results and comparison to other methods

1) Comparison of 2D wavelet lifting, GPU versus CPU: First, we emphasize that the accuracies of the GPU and CPU implementations are the same. Because only integer operations are used (cf. section V-A) the results are identical.

We compared the speed of performing various wavelet transforms using our optimized GPU implementation, to an optimized wavelet lifting implementation on the CPU, called Schrödinger [?]. The latter implementation makes use of vectorization using the MMX and SSE instruction set extensions, thus can be considered close to the maximum that can be achieved on the CPU with one core.

Table III shows the timings of both our GPU accelerated implementation and the Schrödinger implementation when computing a three-level transform with various wavelets of a 1920×1080 image consisting of 16-bit samples. As it is better from an optimization point of view to have a tailored kernel for each wavelet type, than to have a single kernel that handles everything, we used a code generation approach to create specific kernels for the horizontal and vertical pass for each of the wavelets. Both the analysis (forward) and synthesis (inverse) transform are benchmarked. We observe that speedups by a factor of 10 to 14 are reached, depending on the type of wavelet and the direction of the transform. The speedup factor appears to be roughly proportional to the length of the filters. The Haar wavelet is an exception, since the overlap problem does not arise in this case (the filter length being just 2), which explains the larger speedup factor.

To demonstrate the importance of coalesced memory access in CUDA, we also performed timings using a trivial CUDA implementation of the Haar wavelet, that uses the same algorithm for the vertical step as for the horizontal step, instead of our sliding window algorithm. Note that this method can be considered an improved (using cache) row-column, hardware-based strategy, see Section II. Whilst our algorithm processes an image in 0.80 milliseconds, the trivial algorithm takes 15.23, which is almost 20 times slower. This is even slower than performing the transformation on the CPU.

Note that the timings in Table III do not include the time required to copy the data from (2.4 ms) or to (1.6 ms) the GPU.

2) Vertical step via transpose method: Another method that we have benchmarked consists in reusing the horizontal step as vertical step by using a "transpose" method. Here, the matrix of wavelet coefficients is transposed after the horizontal pass, the algorithm for the horizontal step is applied, and the results are transposed back. The results are shown in columns 3 and 4 of Table III. Even though the transpose operation in CUDA is efficient and coalescable, and this approach is much easier to implement, the additional passes over the data reduce performance quite severely. Another drawback of this method is that transposition cannot be done in-place efficiently (in the general case), which doubles the required memory, so that the advantage of using the lifting strategy is lost.

3) Comparison of horizontal and vertical steps: Table IV shows separate benchmarks for the horizontal and vertical steps, using various wavelet filters. From these results one can conclude that the vertical pass is not significantly slower

Table III
PERFORMANCE OF OUR CUDA GPU IMPLEMENTATION OF 2D WAVELET LIFTING (COLUMN 5) COMPARED TO AN OPTIMIZED CPU IMPLEMENTATION (COLUMN 2) AND A CUDA GPU TRANSPOSE METHOD (COLUMN 3, SEE TEXT), COMPUTING A THREE-LEVEL DECOMPOSITION OF A 1920×1080 IMAGE FOR BOTH ANALYSIS AND SYNTHESIS STEPS.

Wavelet (analysis)	**CPU** (ms)	**GPU** transpose (ms)	**Speed-up**	**GPU** our method (ms)	**Speed-up**
Haar	10.31	5.58	1.9	0.80	12.9
Deslauriers-Dubuc $(9,7)$	16.84	6.01	2.8	1.50	11.2
Le Gall $(5,3)$	14.03	5.89	2.4	1.34	10.5
Deslauriers-Dubuc $(13,7)$	19.52	6.08	3.2	1.62	12.0
Daubechies $(9,7)$	22.66	6.54	3.5	2.05	11.1
Fidelity	28.82	6.45	4.5	2.11	13.7
Wavelet (synthesis)	**CPU** (ms)	**GPU** transpose (ms)	**Speed-up**	**GPU** our method (ms)	**Speed-up**
Haar	9.11	6.33	1.4	0.83	11.0
Deslauriers-Dubuc $(9,7)$	15.93	6.40	2.5	1.45	11.0
Le Gall $(5,3)$	13.02	6.29	2.1	1.28	10.2
Deslauriers-Dubuc $(13,7)$	18.22	6.48	2.8	1.55	11.8
Daubechies $(9,7)$	21.73	7.03	3.1	2.04	10.7
Fidelity	27.21	6.86	4.0	2.18	12.5

Table IV
PERFORMANCE OF OUR GPU IMPLEMENTATION ON 16-BIT INTEGERS, SEPARATE TIMINGS OF HORIZONTAL AND VERTICAL STEPS ON A ONE-LEVEL DECOMPOSITION OF A 1920×1080 IMAGE.

Wavelet (analysis)	**Horizontal (ms)**	**Vertical (ms)**
Haar	0.26	0.19
Deslauriers-Dubuc $(9,7)$	0.44	0.42
Le Gall $(5,3)$	0.39	0.34
Deslauriers-Dubuc $(13,7)$	0.47	0.47
Daubechies $(9,7)$	0.62	0.62
Fidelity	0.63	0.76
Wavelet (synthesis)	**Horizontal (ms)**	**Vertical (ms)**
Haar	0.29	0.19
Deslauriers-Dubuc $(9,7)$	0.39	0.44
Le Gall $(5,3)$	0.35	0.36
Deslauriers-Dubuc $(13,7)$	0.42	0.48
Daubechies $(9,7)$	0.58	0.79
Fidelity	0.59	0.64

(and in some cases even faster) than the horizontal pass, even though it performs more elaborate cache management, see Algorithm 1.

Table V
PERFORMANCE OF OUR GPU IMPLEMENTATION ON 16 VERSUS 32-BIT INTEGERS (3 LEVEL TRANSFORM, 1920×1080 IMAGE).

Wavelet (analysis)	**16-bit (ms)**	**32-bit (ms)**
Haar	0.80	1.09
Deslauriers-Dubuc $(9,7)$	1.50	1.64
Le Gall $(5,3)$	1.34	1.45
Deslauriers-Dubuc $(13,7)$	1.62	1.75
Daubechies $(9,7)$	2.05	2.13
Fidelity	2.11	2.72
Wavelet (synthesis)	**16-bit (ms)**	**32-bit (ms)**
Haar	0.83	1.15
Deslauriers-Dubuc $(9,7)$	1.45	1.81
Le Gall $(5,3)$	1.28	1.66
Deslauriers-Dubuc $(13,7)$	1.55	1.90
Daubechies $(9,7)$	2.04	2.35
Fidelity	2.18	2.80

4) Timings for 16*-bit versus* 32*-bit integers:* We also benchmarked an implementation that uses 32-bit integers, see Table V. For small wavelets like Haar, the timings for 16- and 32-bit differ by a factor of around 1.5, whereas for large wavelets the two are quite close. This is probably because the smaller wavelet transforms are more memory-bound and the larger wavelets are more compute-bound, hence the increased memory bandwidth does not affect the performance significantly.

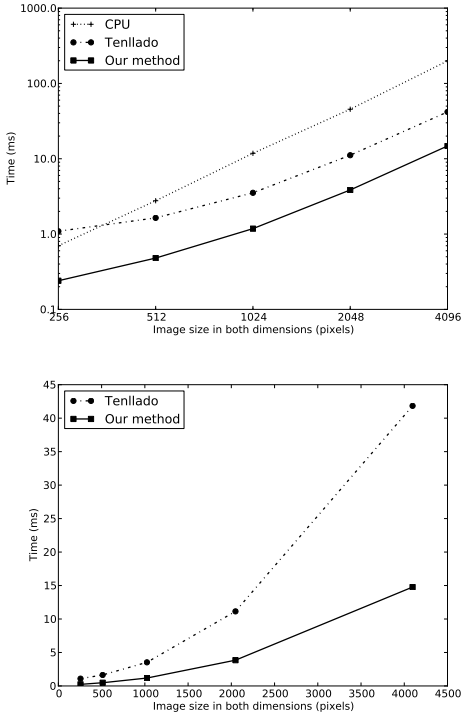

Figure 9. Computation time versus image size for various lifting implementations; 3-level Daubechies $(9,7)$ forward transform. Top: the Schrödinger CPU implementation, Tenllado *et al.* [?] and our CUDA accelerated method in a log-log plot. Bottom: just the two GPU methods in a linear plot.

5) Comparison of 2D wavelet lifting on GPU, CUDA versus fragment shaders: We also implemented the algorithm of Tenllado *et al.* [?] for wavelet lifting using conventional fragment shaders and performed timings on the same hardware. A three-

level Daubechies $(9,7)$ forward wavelet transform was applied to a 1920×1080 image, which took 5.99 milliseconds. In comparison, our CUDA-based implementation (see Table III) does the same in 2.05 milliseconds, which is about 2.9 times faster. This speedup probably occurs because our method effectively makes use of CUDA shared memory to compute intermediate lifting steps, conserving GPU memory bandwidth, which is the bottleneck in the Tenllado method. Another drawback that we noticed while implementing the method is that an important advantage of wavelet lifting, i.e., that it can be done in place, appears to have been ignored. This is possibly due to an OpenGL restriction by which it is not allowed to use the source buffer as destination, the same result is achieved by alternating between two halves of a buffer, resulting in a doubling of memory usage.

Figure 9 further compares the performance of the Schrödinger CPU implementation, Tenllado *et al.* [?] and our CUDA accelerated method. A three-level Daubechies $(9,7)$ forward wavelet decomposition was applied to images of different sizes, and the computation time was plotted versus image size in a log-log graph. This shows that our method is faster by a constant factor, regardless of the image size. Even for smaller images, our CUDA accelerated implementation is faster than the CPU implementation, whereas the shader-based method of Tenllado is slower for 256×256 images, due to OpenGL rendering and state set-up overhead. CUDA kernel calls are relatively lightweight, so this problem does not arise in our approach. For larger images the overhead averages out, but as the method is less bandwidth efficient it remains behind by a significant factor.

Table VI
PERFORMANCE OF OUR GPU LIFTING IMPLEMENTATION IN 3D, COMPARED TO AN OPTIMIZED CPU IMPLEMENTATION; A THREE-LEVEL DECOMPOSITION FOR BOTH ANALYSIS AND SYNTHESIS IS PERFORMED, ON A 512^3 VOLUME.

Wavelet (analysis)	**CPU** (ms)	**GPU** (ms)	**Speed-up**
Haar	1037.4	147.2	7.0
Deslauriers-Dubuc $(9,7)$	2333.1	192.0	12.2
Le Gall $(5,3)$	1636.2	179.3	9.1
Deslauriers-Dubuc $(13,7)$	3056.1	200.5	15.2
Daubechies $(9,7)$	3041.3	234.6	13.0
Fidelity	5918.4	239.2	24.7
Wavelet (synthesis)	**CPU** (ms)	**GPU** (ms)	**Speed-up**
Haar	926.5	150.9	6.1
Deslauriers-Dubuc $(9,7)$	2289.9	184.7	12.4
Le Gall $(5,3)$	1631.1	173.7	9.4
Deslauriers-Dubuc $(13,7)$	2983.9	192.1	15.5
Daubechies $(9,7)$	2943.5	232.0	12.7
Fidelity	5830.7	230.9	25.3

6) Comparison of lifting versus convolution in CUDA: Additionally, we compared our method to a convolution-based wavelet transform implemented in CUDA, one that uses shared memory to perform the convolution plus downsampling (analysis), or upsampling plus convolution (synthesis) efficiently. On a 1920×1080 image, for a three-level transform with the Daubechies $(9,7)$ wavelet, the following timings are observed: 3.4 ms for analysis and 5.0 ms for synthesis. The analysis is faster than the synthesis because it requires less computations – only half of the coefficients have to be computed, while the other half is discarded in the downsampling step. Compared to the 2.0 ms of our own method for both transforms, this is significantly slower. This matches the expectation that a speedup factor of 1.5 to 2 can be achieved when using lifting [?].

7) Timings for 3D wavelet lifting in CUDA: Timings for the 3-D approach outlined in Section IV-G are given in Table VI. A three-level transform was applied to a 512^3 volume, using various wavelets. The timings are compared to the same CPU implementation as before, extended to 3-D. The numbers show that the speed-ups that can be achieved for higher dimensional transforms are considerable, especially for the larger wavelets such as Deslauriers-Dubuc $(13,7)$ or Fidelity.

8) Summary of experimental results: Compared to an optimized CPU implementation, we have seen performance gains of up to nearly 14 times for $2D$ and up to 25 times for $3D$ images by using our CUDA based wavelet lifting method. Especially for the larger wavelets, the gains are substantial. When compared to the trivial transpose-based method our method came out about two times faster over the entire spectrum of wavelets. When regarding computation time versus image size, our GPU based wavelet lifting method was measured to be the fastest of three methods for all image sizes, with the factor mostly independent of the image size.

C. Performance Analysis

We analyze the performance of our GPU implementation, according to the metrics from Section IV-B, for performing one lifting (analysis) step. Without loss of generality, we discuss the Deslauriers-Dubuc $(13,7)$ wavelet, cf. Section IV-F. Our systematic approach consists first in explaining the total execution time, throughput and bandwidth of our method, and then in discussing the design decisions we made. The overhead of data transfer between CPU and GPU was excluded, since the wavelet transform is usually part of a larger processing pipeline (such as a video codec), of which multiple steps can be carried out on the GPU.

1) Horizontal step: The size of the input data set is $N = w \cdot h = 1920 \cdot 1080$ two-byte words. We set $T = 256$ threads per block, and given the number of registers and the size of the shared memory used by our kernel, NVidia's occupancy calculator indicates that $k = 3$ blocks are active per MP, such that each MP is fully occupied (i.e., $kT = 768$ threads will be scheduled); the number of thread blocks for the horizontal step is $B = 1080$. Given that the 8800 GTX GPU has $M = 16$ MPs it follows that $\alpha = 23$, see Section IV-B. Further, we used *decuda* (a disassembler of GPU binaries; see http://wiki.github.com/laanwj/decuda) to count the number and type of instructions performed. After unrolling the loops, we found that the kernel has 309 instructions, 182 of which are arithmetic operations in local memory and registers, 15 instructions are half-width (i.e., instruction code is 32-bit wide), 82 are memory transfers and 30 are other instructions (mostly type conversions). Assuming that half-width instructions have a throughput of 2 cycles, and others take 4 cycles per warp, and since the clock rate of this device is $K = 1.35$ GHz, the

asymptotic execution time is $T_e = 0.48$ ms. Here we assumed that the extra overhead due to rescheduling is negligible, as was confirmed by our experiments.

For the transfer time, we first computed the ratio of arithmetic to arithmetic-and-transfers instructions, which is $r = 0.67$. Thus, from Eq. (5) it follows that as many as 301 cycles can be spared due to latency hiding. As the amount of shared memory used by the kernel is relatively small (i.e., 3×3.75 KB used out of 16 KB per MP) and the size of the L2 cache is about 12 KB per MP [?], we can safely assume that the latency of a global memory access is about 350 cycles, so that $l_m = 49$ cycles. Since $m = 4$ (i.e., two two-byte words are coalesced), the transfer time is $T_m = 0.15$ ms. Note that as two MPs also share a small but faster L1 cache of 1.5 KB, the real transfer time could be even smaller than our estimate. Moreover, as we included also in our counting shared-memory transfers (whose latency is at least 10 times smaller than that of global memory), the real transfer time should be much smaller than its estimate.

According to our discussion in Section IV-E, five synchronization points are needed to ensure data consistency between individual steps. For one barrier, in the ideal case, the estimated waiting time is $T_s = 1.65$ μs, thus the total time is about 8.25 μs. In the worst case $T_s = 0.2$ ms, so that the total time can be as large as 1 ms.

To summarize, the estimated execution time for the horizontal step is about $T_t = 0.63$ ms, neglecting the synchronization time. Comparing this result with the measured one from Table IV, one sees that the estimated total time is 0.16 ms larger than the measured one. Probably this is due to L1 caching contributing to a further decrease of T_m. However, essential is that the total time is dominated by the execution time, indicating a compute-bound kernel. As the timing remains essentially the same (cf. Tables III and V) when switching from two-byte words to four-byte words data, this further strengthens our finding.

The measured throughput is $G_m = 98$ Gflop/s, whereas the estimated one is $G_e = 104$ Gflop/s, indicating on average an instruction throughput of about 100 Gflop/s. Note that with some abuse of terminology we refer to flops, when in fact we mean arithmetic instructions on integers. The measured bandwidth is $M_b = 8.8$ GB/s, i.e., we are quite far from the pin-bandwidth (86 GB/s) of the GPU, thus one can conclude again that our kernel is indeed compute-bound. This conclusion is further supported by the fact that the flop-to-byte ratio of the GPU is 5, while in our case this ratio is about 11. The fact that the kernel does not achieve the maximum throughput (using shared memory) of about 230 Gflop/s is most likely due to the fact that the synchronization time cannot simply be neglected and seems to play an important role in the overall performance.

Let us now focus on the design choices we have made. Using $T = 256$ threads per block amounts to optimal time slicing (latency hiding), see discussion above and in Section IV-B, while we are still able to coalesce memory transfers. To decrease the synchronization time, lighter threads are suggested implying that their number should increase, while maintaining a fixed size of the problem. NVidia's performance guidelines [?] suggest that the optimal number of threads per block should be a multiple of 64. The next higher than 256 multiple of 64 is 320. Unfortunately, using 320 threads per block means that at most two blocks can be allocated to one MP, and thus the MP will not be fully occupied. This in turn implies that an important amount of idle cycles spent on memory transfers cannot be saved, rendering the method less optimal with respect to time slicing. Accordingly, our choice of $T = 256$ threads per block is optimal. Further, our choice on the number of blocks also fulfills NVidia's guidelines with respect to current and future GPUs, see [?].

2) Vertical step: While conceptually more involved than the horizontal step, the overall performance figure for the vertical step is rather similar to the horizontal one. The CUDA configuration for this kernel is as follows. Each 2D thread block contains a number of $16 \times 8 = 128$ threads, while the number of columns within each slab is $S = 32$, see Figure 6. Thus, since the input consists of two-byte words, each thread performs coalesced memory transfers of $m = 4$ bytes, similar to the horizontal step. As the number of blocks is $w/S = 60$, $k = 4$ (i.e., four blocks are scheduled per MP), and the kernel takes 39240 cycles per warp to execute, the execution time for the vertical step is $T_e = 0.46$.

Unlike the horizontal step, now $r = 0.83$ so that no less than 352 cycles can be spared in global-memory transaction. Note that when computing r we only counted global-memory transfers, as in this case more, much faster shared-memory transfers take place, see Algorithm 1. As the shared-memory usage is only 4×1.8 KB, this suggests that the overhead due to slow accesses to global memory can be neglected, so that the transfer time T_m can be neglected. The waiting time is $T_s = 0.047$ μs, and there are 344 synchronization points for the vertical-step kernel, so that the total time is about 15.6 μs. In the worst case, this time can be as large as 1.9 ms. Thus, as $T_t = 0.46$ (without waiting time), our estimate is very close to the measured execution time from Table IV – this being in turn the same as that of the horizontal step. Finally, both the measured and estimated throughputs are comparable to their counterparts of the horizontal step.

Note that compared to the manually-tuned, optimally-designed matrix-multiplication algorithm of [?] which is able to achieve a maximum throughput of 186 Gflop/s, the performance of 100 Gflop/s of our lifting algorithms may not seem impressive. However, one should keep in mind that matrix-multiplication is much easier to parallelize efficiently, as it requires little synchronization. Unlike matrix-multiplication, the lifting algorithm requires a lot more synchronization points to ensure data consistency between steps, as the transformation is done in-place.

The configuration we chose for this kernel is $16 \times 8 = 128$ threads per block and $w/S = 60$ thread blocks. This results in an occupancy of 512 threads per MP, which may seem less optimal. However, to increase the number of threads per block to 192 (next larger multiple of 64, see above), would mean that either we cannot perform essential, coalesced memory accesses, or that extra overhead due to the requirements of the moving-window algorithm would have to be accommodated. Note that we verified this possibility, but the results were

unsatisfactory.

3) Complexity: Based on the formulae from Section IV-B we can analyze the complexity of our problem. For any of the lifting steps using the Deslauriers-Dubuc $(13, 7)$ wavelet, considering that the number of flops per data element is $n_s = 22$ (20 multiply or additions and 2 register-shifts to increase accuracy), the numerator of (9) becomes about $700\,D$. For the horizontal step, $D = w/T = 7.5$, so that the numerator becomes about 5000. In this case the number of cycles is about 1250, so that one can conclude that the horizontal step is indeed *cost efficient*. For the vertical step, $D = (S\,h)/T = 270$, so that the numerator in (9) becomes about 190000, while the denominator is 39240. Thus, the vertical step is also cost efficient, and actually its performance is similar to that of the horizontal step (because $5000/1250 \approx 190000/39240 \approx 5$). Of course, this result was already obtained experimentally, see Table IV. Note that using vectorized MMX and SSE instructions, the *optimized* CPU implementation (see Table III) can be up to four times faster than our T_S estimate above. However, even in this case, both our CUDA kernels are still cost-efficient. Obviously both steps are also *work efficient*, as their CUDA realizations do not perform asymptotically more operations than the sequential algorithm.

VI. Conclusion

We presented a novel, fast wavelet lifting implementation on graphics hardware using CUDA, which extends to any number of dimensions. The method tries to maximize coalesced memory access. We compared our method to an optimized CPU implementation of the lifting scheme, to another (non-CUDA based) GPU wavelet lifting method, and also to an implementation of the wavelet transform in CUDA via convolution. We implemented our method both for 2D and 3D data. The method is scalable and was shown to be the fastest GPU implementation among the methods considered. Our theoretical performance estimates turned out to be in fairly close agreement with the experimental observations. The complexity analysis revealed that our CUDA kernels are cost- and work-efficient.

Our proposed GPU algorithm can be applied in all cases were the Discrete Wavelet Transform based on the lifting scheme is part of a pipeline for processing large amounts of data. Examples are the encoding of static images, such as the wavelet-based successor to JPEG, JPEG2000 [**?**], or video coding schemes [?], which we already considered in [**?**].

VII. Acknowledgements

We thank the anonymous reviewers for several helpful suggestions to improve our paper.

This research is part of the "VIEW: Visual Interactive Effective Worlds" program, funded by the Dutch National Science Foundation (NWO), project no. 643.100.501.

Wladimir J. van der Laan received his M.Sc. (2005) in computing science at the University of Groningen. He is currently pursuing a Ph.D. degree at the Johann Bernoulli Institute for Mathematics and Computer Science of the University of Groningen. His research interests include computer graphics, visualization and GPU computing.

Andrei C. Jalba received his B.Sc. (1998) and M.Sc. (1999) in Applied Electronics and Information Engineering from "Politehnica" University of Bucharest, Romania. He obtained a Ph.D. degree at the Institute for Mathematics and Computing Science of the University of Groningen in 2004. Currently he is assistant professor at the Eindhoven University of Technology. His research interests include computer vision, pattern recognition, image processing, and parallel computing.

Jos B. T. M. Roerdink received his M.Sc. (1979) in theoretical physics from the University of Nijmegen, the Netherlands. Following his Ph.D. (1983) from the University of Utrecht and a two-year position (1983-1985) as a Postdoctoral Fellow at the University of California, San Diego, both in the area of stochastic processes, he joined the Centre for Mathematics and Computer Science in Amsterdam. There he worked from 1986-1992 on image processing and tomographic reconstruction. He was appointed associate professor (1992) and full professor (2003), respectively, at the Institute for Mathematics and Computing Science of the University of Groningen, where he currently holds a chair in Scientific Visualization and Computer Graphics. His current research interests include biomedical visualization, neuroimaging and bioinformatics.