Accelerating Wavelet-Based Video Coding on Graphics Hardware using CUDA

Wladimir J. van der Laan, Jos B.T.M. Roerdink
Institute of Mathematics and Computing Science
University of Groningen
Groningen, The Netherlands
{w.j.van.der.laan,j.b.t.m.roerdink}@rug.nl

Andrei C. Jalba
Department of Mathematics and Computer Science
Eindhoven University of Technology
Eindhoven, The Netherlands
a.c.jalba@tue.nl

Abstract

The Discrete Wavelet Transform (DWT) has a wide range of applications from signal processing to video and image compression. 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 method is scalable and the fastest GPU implementation among the methods considered. We have integrated our DWT into the Dirac Wavelet Video Codec (DWVC), of which the overlapped block motion compensation compensation and frame arithmetic have been accelerated using CUDA as well.

1 Introduction

The Discrete Wavelet Transform (DWT) has been widely applied in signal and image processing. To meet the computational requirements for systems that handle very large throughputs, for example in real-time multimedia processing, custom hardware has been developed. Another option is to use a more general, widely available and relatively cheap platform, such as GPU hardware.

We recently developed a hardware-accelerated DWT algorithm that makes use of NVidia's Compute Unified Device Architecture (CUDA) parallel programming model [6] to fully exploit the new features offered by the Tesla architecture, introduced in 2006 with the GeForce 8800 GPU [14]. It is a highly parallel computing architecture available for systems ranging from laptops to high-end compute servers. Our DWT implementation is based on the Lifting Scheme [11] which reduces the number of arithmetic operations compared to the straightforward convolution-based approach. Also, the memory usage is reduced by factoring the wavelet transform into a sequence of steps that can be performed *in-place*. This is a great advantage given the generally limited amount of high-speed memory and the large data sizes that have to be processed in multimedia applications. The method is scalable and the fastest GPU implementation available to date.

In this paper, we will show how to integrate our accelerated wavelet lifting into an implementation of the Dirac Wavelet Video Codec (DWVC) [2]. This codec, first introduced by the BBC, is gaining popularity as a free, open source alternative to H.264 [15]. It is a modern video compression scheme that employs wavelet transforms for inter- and intra- frame image compression, and makes use of motion compensation for compact storage of the difference between successive frames. Moreover, it is on-par with other modern video codec systems, e.g., H.264, which has gained wide acceptance in many applications like Internet broadcasting. It is a good alternative because the usage of H.264 incurs royalty fees, and while these costs are manageable for commercial applications, they could become prohibitive for public domain initiatives such as video archives. DWVC provides an alternative that is free of these fees and is equal in compression rates and quality [13]. Another advantage of wavelet-based video compression is that, as it uses a global transform, it does not suffer from the block artifacts otherwise seen in traditional DCT-based codecs.

The acceleration of video decoding using GPU hardware was also studied by Shen *et al.* [10], with the aim to provide an architecture for video coding on the GPU, with special focus on the motion compensation and frame arithmetic parts. As they were programming the GPU using vertex and fragment shaders, the authors had to overcome the additional complexity of mapping the video decoding process to the rendering pipeline. In this paper we employ a more general programming architecture, which means that we can focus on the actual parallel implementation of the algorithms. Doing so, we achieve speedups of a factor 13 for the image operations, and a factor 5.4 for the entire pipeline. In addition to the wavelet transform, we will discuss how the motion compensation and frame arithmetic steps of this codec can be accelerated using CUDA. Our proposed algorithm applies similarly to other wavelet-based video coding schemes that make use of the lifting scheme, such as [7–9, 12].

The remainder of this paper is organized as follows. In Section 2 we summarize the essentials of the hardware-accelerated DWT algorithm, including a brief discussion of the CUDA programming environment and execution model.

Then in Section 3 we give the details of our implementation of the DWVC in CUDA on the GPU. Section 4 presents benchmark results and analyzes the performance of our method. Finally, in Section 5 we draw conclusions and discuss future avenues of research.

2 CUDA-based implementation of the DWT

2.1 CUDA overview

NVidia's CUDA programming environment allows the GPU to be programmed through traditional CPU means: a C++-like language and compiler. The usage of CUDA does not add any overhead over rendering-based approaches, as it is a native interface to the hardware, and not an abstraction layer. CUDA broadly follows the data-parallel model of computation [6]. The CPU invokes the GPU by calling a *kernel*, which is a special C++ function.

The lowest level of parallelism is formed by scalar execution units called *threads*. A large number of threads can run in parallel. Threads are organized in *blocks*, and the threads within a block can share data through fast shared memory. It is also possible to place synchronization points (barriers) to control flow between all threads within a block. 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 a warp, and the threads take different paths, they have to wait for each other (*divergence*). The highest level, which encompasses the entire kernel invocation, is called the *grid*, which consists of blocks that execute in parallel (if multiprocessors are available). Currently blocks within a grid cannot communicate with each other.

The CUDA architecture provides access to several kinds of memory. *Global* (device) memory can be read and written in any order (random access). *Registers* are limited per-thread memory locations for local storage with very fast access. *Shared memory* is a limited per-block chunk of memory which is used for communication between threads in a block. *Texture memory* is a special case of device memory which is cached for locality. *Constant memory* is cached memory that can be written by the CPU and read by the GPU. To achieve highest throughput, consecutive memory locations must be simultaneously accessed by the threads. This is called *memory access coalescing* [6], and it represents one of the most important optimizations in CUDA.

2.2 Wavelet lifting

In wavelet lifting an input signal or image is decomposed into an *approximation band* x and one or more *detail bands* y_s using polyphase decomposition, see Fig. 1. In lifting theory, this polyphase decomposition is a (trivial) wavelet called the *lazy wavelet transform*, which splits the signal into two parts, containing the even and odd coefficients, respectively. A prediction step computes a prediction $P(x)$ from the approximation band x, after which the predicted

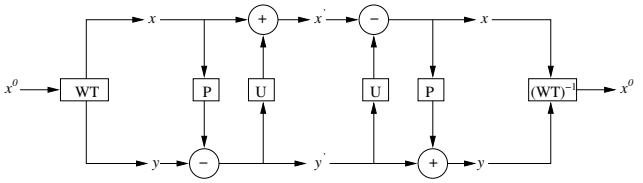

Figure 1. Classical lifting scheme with one detail band. x^0 is the original image, WT is the lazy (trivial) wavelet transform that transforms the signal into approximation band x and detail band y, P is the prediction step, U the update step. $(WT)^{-1}$ is the inverse of the trivial wavelet transform.

values are subtracted from the detail band y_s, to produce a new detail band y'_s, i.e.,

$$y'_s = y_s - P(x). \tag{1}$$

An update step updates the approximation band x using the detail bands y_s,

$$x' = x + U\left(y_s\right). \tag{2}$$

The entire scheme is reversible by applying the steps in reverse order while interchanging $-$ and $+$. For a multilevel transform the process is repeatedly applied to the approximation bands, until a desired number of decomposition levels is reached. Wavelet lifting has the additional property that it can be done entirely with *integer operations*, resulting in a lossless scheme when applied to discrete images.

2.3 Wavelet lifting in CUDA

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. 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.

In 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*. Some duplication of border elements is necessary to properly implement the boundary conditions. Each step is dependent on the output in shared memory of the previous step, therefore the threads within a block have to be synchronized every time before the next step can start. By reorganizing the coefficients [3] we can achieve higher efficiency for successive levels after the first transformation. To be able to coalesce, it must be possible to read back the coefficients consecutively, thus one writes the approximation and detail coefficients back to separate halves of the memory, de-interleaving them.

For the vertical pass we could use the same strategy as for the horizontal pass, substituting rows for columns. But we have shown [14] that a more efficient solution is possible which makes optimal use of 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*. The number of columns in each slab is such that the resulting number of slab rows can still be coalesced. Each thread block then processes one of the slabs, so that 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.

Because the shared memory in CUDA is usually not large enough to store all columns, we use a sliding window within each slab. The dimensions of this window need to be such 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.

3 Accelerating the Dirac Video Codec

The weakest point of DWVC is currently its execution time [13]. Real-time decoding is limited to smaller resolutions (such as 800×600), even for the latest processors. The relatively heavy computational load of the global wavelet transform compared to more traditional block-wise Discrete Cosine Transforms (DCT) has prevented wavelets from being used in mainstream video compression. In an aim to speed up decoding, we implemented all the image operations of DWVC on the GPU, including the wavelet transform, motion compensation and frame arithmetic. As the CPU implementation (called Schrödinger) is already heavily optimized, it provides a good basis for performance comparison of our GPU wavelet lifting algorithm.

A DWVC stream consists of intra- and inter-frames. Intra-frames are self-contained images, while inter-frames store the difference with respect to one or two reference frames.

Decoding consists of three major parts (Fig. 2): arithmetic decoding, motion compensation, and inverse wavelet transform. Arithmetic decoding takes the bitstream and extracts parameters, motion vectors and wavelet coefficients needed to reconstruct the video sequence. It reverses the work of the entropy coder, which removes statistical redundancies from the data by representing common values with shorter bit sequences. This part is most conveniently handled by the CPU, as there is very little inherent parallelism in the process.

Motion compensation exploits the similarity between neighboring video frames. It reconstructs a frame from one or two frames preceding the current one in the stream. Motion compensation is done both on a global and local level. Global motion compensation seeks to describe movements of the camera, while local motion compensation acts on a per-block basis for smaller moving objects.

The images (for intra-frames) and residue (for inter-frames) are stored as wavelet coefficients in a per-component, per-subband basis in the video stream. Subbands that are zero or mostly zero are encoded as empty subbands, represented by only one bit. Only the non-zero subbands are transferred to the hardware. The wavelet filters that are used in the default settings of the encoder are the Deslauriers-Dubuc $(9, 3)$ filter [4] for intra-frames and the LeGall $(5, 3)$ filter [5] for inter-frames. A full list of wavelet filters used in DWVC can be found in [2].

The result of the motion compensation process is a prediction. The reconstructed residue is added to this prediction to form the final decoded frame which is shown on the screen. If the frame was marked as a reference for a future frame, it is stored until the stream tells it to retire.

As we already mentioned, we implemented the wavelet transforms, motion compensation and frame operations like adding, subtracting, conversion and (un)packing on the GPU. In the upcoming sections we will discuss these operations, and show how they can be applied to decoding and encoding of video data.

3.1 Motion compensation

Traditional motion compensation algorithms divide the image into equally-sized, disjoint blocks of pixels. This has the disadvantage that there can be strong discontinuities between neighboring blocks, and moreover, the prediction accuracy on block edges is low. The residual difference image should be as smooth as possible to achieve the best compression, as jumps and discontinuities in the image cause large values in the detail subbands after the wavelet transform, which in turn results in a less compact representation. Overlapped Block Motion Compensation (OBMC) [1] overlaps neighboring blocks a bit, blending them together in the area which they share, thus increasing prediction accuracy.

The coverage of the image by blocks is defined using four parameters. The first two, `xlen` and `ylen`, define the size of the blocks in the horizontal and vertical direction. The second two parameters, `xsep` and `ysep`, define the separation of the beginning of a block to the beginning of the next one in the x- and y-direction, respectively. OBMC is a generalization of traditional motion compensation, as it equals standard disjoint motion compensation if `xlen` $=$ `xsep` and `ylen` $=$ `ysep`.

One or two reference frames can be arbitrarily selected from preceding or subsequent frames. For example, it is possible to do a blending between the previous and the next frame, useful in the case of a fade-in or fade-out, but it is also possible to use an image a few frames back for reference, if that image provides a better match to the current one. If two reference frames are used, these are blended together with weights w_1 and w_2.

A motion vector is a two-dimensional vector that stores the displacement of a block as compared the the reference frame. For example, if the previous frame is used as a reference, and an object moved two pixels to the right since

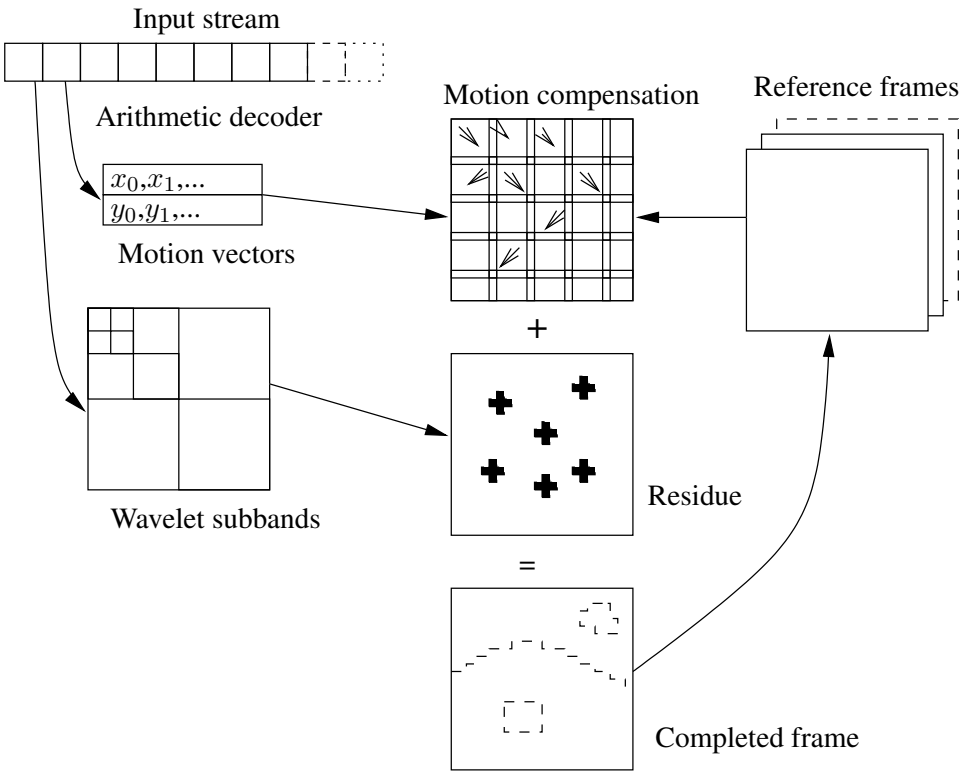

Figure 2. Overview of decoding a DWVC stream. The steps are: arithmetic decoding, motion compensation, and inverse wavelet transform.

the last frame, the motion vector for the block containing the object would be $(2, 0)$. Sub-pixel precision is supported by interpolating the reference frame using a bicubic spline filter. This fits perfectly to texture mapping hardware, if one stores the reference frames as textures.

As each pixel of the resulting image is computed independently, motion compensation is very well suited to a parallel GPU implementation. Each pixel can be part of up to four motion compensation blocks per reference frame. The value of an output pixel (x, y) is calculated by

$$I(x,y) = \sum_{m \in M} w_m(x,y) \Big(w_1 R_1(x + m_{x1}, y + m_{y1}) + w_2 R_2(x + m_{x2}, y + m_{y2}) \Big), \quad (3)$$

in which I is the output frame, M is the set of all blocks, $w_m(x, y)$ is the weight of block m at position (x, y), w_1 and w_2 are the reference frame weights, R_i are the reference frames, and (m_{xi}, m_{yi}), $i = 1, 2$, are the two motion vectors for block m. The block weights are defined so that

$$\sum_{m \in M} w_m(x,y) = 1,$$

and the weights have a linear fall-off at the block edges.

The most obvious approach to a GPU implementation is to divide the image into equally-sized CUDA blocks, whose pixels are then processed by a CUDA thread. This thread determines which motion compensation blocks overlap a pixel, and calculates the output value using Eq. (3).

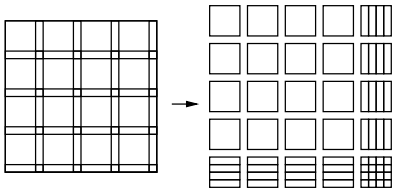

Figure 3. Dividing the frame into four block types according to the number of overlapping blocks, for efficient motion compensation.

Such an approach will result in quite some flow control per pixel, and as neighboring pixels might need values from different motion compensation blocks, thread divergence arises. In other words, in the optimal setting threads within each CUDA block should perform the same operations, while different CUDA blocks can do different computations. An improved algorithm divides the image into regions according to the number of overlapping blocks and orientation of overlap (Fig. 3); in the center of the blocks, it suffices to take a sample from one block. Then there are the cases in which two blocks overlap, horizontally or vertically. Here, the two blocks need to be blended together linearly, so that there is a smooth transition. And finally

there are the diagonal overlaps where four blocks overlap, which must be blended together. A bilinear blending is used to create a smooth transition here.

Therefore, for each region, we know exactly how many, and which blocks it depends on, and how to blend the blocks. Thus, regions can be handled in parallel, but each needs to be handled in a different way. In CUDA, different blocks can execute different kernels when a large conditional statement is put around them that depends on the block identifier. As the threads within the block all take the same path, no divergence is introduced. To use this, we pass a block type parameter to each CUDA block. This block type tells the kernel which of the four regions mentioned before should take part in the computation. As CUDA supports scattered writes, each block can write to the part of the image that it computed.

The motion vectors themselves can be passed to the kernel through an array in constant memory, or by using a texture. We noticed that using a texture is significantly faster than constant memory in a case like this, where each thread potentially accesses a different location, so this is preferred to the other approach. Also, unlike constant memory, textures have no 64KB limit.

3.2 Frame arithmetic

The frames resulting from the inverse wavelet transform and the motion compensation are added together to compute the resulting frame. All computations are done on 16-bit per component, but the final rendering needs an 8-bit image, so the values are clamped between 0 and 255. The output of the algorithm can be directly sent to a texture, without having to pass through the host CPU, by using the CUDA-OpenGL interoperability API. This texture can then be rendered on a quadrilateral to show the frame. If the frame is marked as a reference frame, a copy of the texture is kept for use in a later motion compensation stage.

The fact that DWVC uses 16-bit integers instead of 32-bit complicates the implementation, as both shared and global memory access is geared toward 32-bit values. Two 16-bit integers can be combined into a 32-bit read only as long as the address is aligned to four bytes, which means that reading or writing cannot start from an odd column. To get around this constraint, a one-column border at the left or right of the rectangle must be processed using 16-bit memory operations. Even though this gives some overhead, it is much faster than using only 16-bit memory accesses.

4 Performance results

The benchmarks in this section were run on a machine with a Dual Core AMD Opteron(tm) Processor 280 and a NVidia GeForce GTX280 graphics card, using CUDA version 2.2 for the CUDA programs. The codec was benchmarked in single-threaded mode. Using multiple threads on a multi-core machine would increase the performance of both the CPU and GPU implementations, but the coordination involved in using a GPU from multiple threads, though it became possible in version 2 of CUDA, is quite difficult. For the CUDA implementation, the result was not copied back to the CPU after each frame, as we used direct rendering through OpenGL textures.

In Table 1 we compare the overall performance of the DWVC accelerated by our GPU implementation to the optimized CPU implementation. The experiment was performed using two HD video sequences and one lower resolution sequence. Our method runs at an average of 56.4 frames per second for a 1920×1080 sequence, while the CPU version runs at 10.5 frames per second on the same video sequence. This means we achieve a speedup of 5.4 of the entire process.

A breakdown of the computation time into different stages for two different video sequences is shown in Table 2. To make a valid comparison of the total time spent in each stage, the CPU and GPU were synchronized between stages. This prevents overlap in computation and thus results in a somewhat lower overall performance. Stage 1 performs motion compensation (Section 3.1), and is a factor 8 to 12 faster in our CUDA implementation. Stage 2 performs arithmetic decoding of motion vectors and is the same in both implementations. Stage 3 decodes the wavelet subbands from the input stream. This stage is a bit slower for the CUDA version, because it includes copying the non-zero subbands to GPU memory. Stage 4 performs the inverse integer wavelet lifting transform (Section 2.2) on the decoded residue, and is a factor 9 to 13 faster in the CUDA implementation. Stage 5 combines the motion compensation result and residue (Section 3.2), and is a factor 10 to 28 faster in the CUDA implementation. Stage 6 performs upsampling of reference frames for sub-pixel motion compensation, and is a factor 10 to 17 faster in the CUDA implementation.

By excluding *DECODE* (arithmetic decoding) stages, subtracting 12.64 from both totals for the sequence in Table 2, one can determine the speedup of the GPU accelerated operations compared to their CPU counterpart. This amounts to a factor of about 13.

The frame-rate achieved with our method (56.4) allows for playback of high definition video significantly faster than strictly needed for playback of those movie sequences (25).

5 Conclusion

In this paper, we showed how to accelerate the Dirac Video Codec by a fast wavelet lifting implementation on graphics hardware using CUDA. The method maximizes coalesced memory access. We also accelerated the motion compensation and frame arithmetic stages of this codec.

The experiments on high definition video sequences have demonstrated that one can achieve a speedup factor of 5.4 for the entire decoding process including the CPU steps, and of 13 times for just the GPU part. In our benchmark we

Table 1. Performance in frames per second (FPS) for decoding various DWVC video sequences with: (i) Schrödinger CPU implementation; (ii) Our CUDA implementation

Sequence	Frame size	CPU	CUDA
Big Buck Bunny trailer	1920×1080	10.5	56.4
Elephant's Dream	1024×576	33.7	125.6
2012 movie trailer	1920×800	13.2	71.2

Table 2. Big Buck Bunny trailer (813 frames, 1920×1080) decoded with: (i) Schrödinger CPU implementation; (ii) Our CUDA implementation

Stage	**CPU** (s)	**CUDA** (s)
1 MOTION_DECODE	0.64	0.64
2 MOTION_RENDER	16.16	1.33
3 RESIDUAL_DECODE	12.00	12.94
4 WAVELET_TRANSFORM	22.52	1.63
5 COMBINE	11.27	0.39
6 UPSAMPLE	14.53	0.85
Total	77.13	17.76

could playback a 1080p resolution video at 56.4 frames per second.

As the decoding stages that remain on the CPU are quite involved, further work could involve the acceleration of the arithmetic decoding on (future) GPU hardware, or the development of statistics-based data compression methods that are more paralellizable.

6 Acknowledgements

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.

References

[1] C. Auyeung, J. J. Kosmach, M. T. Orchard, and T. Kalafatis, "Overlapped block motion compensation", In P. Maragos, editor, *Proc. SPIE Visual Communications and Image Processing '92*, pp. 561–572, Nov. 1992.

[2] BBC Research, *Dirac Specification 1.0.0pre7*, Available at `http://dirac.sourceforge.net/specification.html`.

[3] S. Chatterjee and C. D. Brooks, "Cache-efficient wavelet lifting in JPEG 2000", In *Proc. of the IEEE International Conference on Multimedia and Expo*, pp. 797–800, 2002.

[4] G. Deslauriers and S. Dubuc, "Symmetric iterative interpolation processes", *Constructive Approximation*, 5(1), dec 1989, pp. 49–68.

[5] D. LeGall and A. Tabatabai, "Sub-band coding of digital images using symmetric short kernel filters and arithmetic coding techniques", In *IEEE Int. Conf. Acoustics, Speech and Signal Processing*, volume 2, pp. 761–764, 1988.

[6] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A Unified Graphics and Computing Architecture", *IEEE Micro*, 28(2), 2008, pp. 39–55.

[7] L. Luo, J. Li, S. Li, Z. Zhuang, and Y.-Q. Zhang, "Motion compensated lifting wavelet and its application in video coding", In *Proc. IEEE International Conference on Multimedia and Expo (ICME)*, pp. 365–368, 2001.

[8] L. Luo, L. Luo, J. Li, S. Li, and Z. Zhuang, "A motion compensated lifting wavelet codec for 3D video coding", *J. Comput. Sci. Technol.*, 18(2), 2003, pp. 214–222.

[9] B. Pesquet-Popescu and V. Bottreau, "Three-dimensional lifting schemes for motion compensated video compression", In *ICASSP '01: Proc. of the Acoustics, Speech, and Signal Processing Conference*, pp. 1793–1796, 2001.

[10] G. Shen, G. P. Gao, S. Li, H. Y. Shum, and Y. Q. Zhang, "Accelerate video decoding with generic GPU", *IEEE Trans. Circuits and Systems for Video Technology*, 15(5), May 2005, pp. 685–693.

[11] W. Sweldens, "The Lifting Scheme: A Construction of Second Generation Wavelets", *SIAM Journal on Mathematical Analysis*, 29(2), 1998, pp. 511–546.

[12] J. Y. Tham, S. Ranganath, and A. A. Kassim, "Highly scalable wavelet-based video codec for very low bit-rate environment", *IEEE Journal on selected areas in communications*, 16, jan 1998, pp. 12–27.

[13] M. Tun, K. K. Loo, and J. Cosmas, "Error-Resilient Performance of Dirac Video Codec Over Packet-Erasure Channel", *IEEE Transactions on Broadcasting*, 53(3), sep 2007, pp. 649–659.

[14] W. J. van der Laan, A. C. Jalba, and J. B. T. M. Roerdink, "Accelerating wavelet lifting on graphics hardware using CUDA", Technical report, Institute for Mathematics and Computing Science, University of Groningen, 2009, Submitted for publication.

[15] T. Wiegand, G. J. Sullivan, G. Bjntegaard, and A. Luthra, "Overview of the H.264/AVC video coding standard", *Circuits and Systems for Video Technology, IEEE Transactions on*, 13(7), 2003, pp. 560–576.