Translation-Invariant Two-Dimensional Discrete Wavelet Transform on Graphics Processing Units

Dietmar Wippig, and Bernd Klauer

***Abstract*—** The Discrete Wavelet Transform (DWT) is used in several signal and image processing applications. Due to the computational expense various approaches have been proposed. One approach is using graphics processing units (GPUs) as stream processors to speed up the calculation of the DWT. This paper presents a GPU implementation of the translation- invariant wavelet transform computed by the "algorithme à trous". Our approach focuses on processing of infrared images, but can be easily used in different image processing applications. We extend our work by the integration of our implementation in wavelet-based edge detection and wavelet denoising. Experiments show that the computation performance could be significantly improved. Initialisation and data transfer are already existing bottlenecks, which could dramatically reduce the GPU performance, if it can't be hided by the application.

***Keywords*—**parallel discrete wavelet transform, algorithme à trous, image processing, GPU, shader.

I. Introduction

THE Discrete Wavelet Transform (DWT) has a broad application field in signal and image processing. In spite of fast filterbank implementations the calculation is still compute intensive especially for large problem sizes and real-time processing. Therefore various approaches have been developed to speed up the calculation of the Discrete Wavelet Transform, which range from special-purpose, fixed-function hardware implementations (ASICs) to universal software implementation on general purpose processors (GPP). These approaches have made tradeoffs between performance and flexibility.

In the last years the development of digital entertainment, not scientific and military applications, drives the development of computing. Besides mobile devices the mass market is increased characterized by customer personal computers. The high performance requirements of these applications lead on the one hand to multimedia extensions of general-purpose processors like MMX, SSE and on the other hand to the evolution of graphics processing units (GPUs) to powerful and programmable processors, which supports general calculations.

Manuscript received September 30, 2010.

Dietmar Wippig was with the Helmut-Schmidt-University, Hamburg, D-22043 Germany (e-mail: wippig@hsu-hamburg.de). He is now with Ministry of Defence, Rheinbach, D-53359 Germany.

Bernd Klauer is with the Helmut-Schmidt-University, Hamburg, D-22043 Germany (e-mail: bernd.klauer@hsu-hamburg.de).

The GPU is an attractive platform for a broad field of applications, because it still remains a significant higher arithmetic processing capability than the GPPs with multimedia extensions and is often less utilized. Therefore it can be used as a powerful accelerator without extra cost [1]. The key for performance increase of applications on the GPUs is a large amount of parallelism and locality, which can be exploited by the GPU. The wavelet transform in general fulfil this requirement and was early an area of research for a GPU implementation [2, 3].

In this paper we present a GPU implementation of the translation invariant wavelet transform computed by the "algorithme à trous". Our approach focus on a two-dimensional (2D) solution for processing of infrared images, which significant increase the computation performance compared with an implementation on GPPs and can be easily used in different image processing applications. We successfully integrate our implementation in wavelet-based edge detection and wavelet denoising. The computation performance could be improved significantly.

II. Related Work

Hopf and Ertl [2, 3] realize the first implementation of the 2D DWT for the fixed-function pipeline. The approach perform the convolution, down- and upsampling of the wavelet transform by special OpenGL extensions, which are not commonly implemented in graphics hardware.

In [4] a new implementation of the 2D DWT for programmable GPUs is proposed, which is based on user defined fragment shader. A texture twice the size of original image size is used for the results as well as for temporary results. The convolution of the image borders with separate kernels prevents border effects. Besides Mallat's pyramid algorithm a GPU implementation of Swelden's lifting algorithms is presented, which has only advantages for long filters.

Beside, Wang et al. [5] have developed another GPU implementation of the 2D DWT using Mallat's pyramid algorithm. The implementation is based on horizontal and vertical convolutions with position-dependent filters, which are mapped to user defined fragment shaders. Border effects are prevented by indirect access on image positions stored in a dynamically generated texture. The integration in Jasper [6, 7], which is a reference implementation of JPEG2000 [8], show the usability and the performance of Wang's implementation. Because of initialization and data transfer times the GPU implementation surpass the classical implementation on image sizes greater than 600x600 pixel [9].

Most applications of the DWT especially in data compression use non redundant algorithms like Mallat's pyramid or Swelden's lifting algorithm. In image analysis and processing a translation-invariant representation can often deliver better results. Holschneider et al. [10] are developed the "algorithme à trous" for the calculation of the translation-invariant DWT. Similar as Mallat's pyramid algorithm the "algorithme à trous" can be implemented by a fast filter bank. Because the resulting subbands are not downsampled, input and output image sizes are equal at each dyadic scale. Especially for large images this results in high memory consumption storing the wavelet transform decomposition and high computation expense for the calculation of the wavelet transform. On this matter hardware requirements have limit the use of the "algorithme à trous". Compared with Mallat's pyramid algorithm the "algorithme à trous" has a higher arithmetic intensity. Consequently on the one side a GPU implementation seems to be especially advantageous. On the other side the large memory consumption is a challenging requirement especially for the graphics hardware. Since two years consumer level graphics hardware with large video memory is available, which made a GPU implementation of the "algorithme à trous" attractive.

III. GPGPU

In the last decade, the GPU became a powerful, flexible stream processor. The development of high level programming languages made a broader use of general purpose applications on graphic processing units (GPGPU) possible. More over with stream programming languages like CUDA, ATI Stream or OpenCL general purpose applications can be easily implemented.

The target of our implementation of the "algorithme à trous" on the GPU was the reusability in future applications. On the one side current stream programming languages must be integrated in the common software development process in a special way. On the other side like CUDA only one hardware vendor is supported. So we still directly use the graphics API with high level shading languages, which could be well integrated in all common programming languages like Basic, Fortran, C/C++/C# or Java and is supported by most of the graphics hardware. Because the algorithms were already available in Java, our implementation is based on Java. As graphics API we use OpenGL, which is bound by the JOGL class library.

IV. DWT

Popularly the DWT is seen to be equal with Mallat's pyramid algorithm, which is based on the multiresolution analysis (MRA) of signals. The decomposition of signals using the DWT can be expressed by a pyramidal filter structure of quadrature mirror filter (QMF) pairs. Therefore the DWT of the discrete signal $f[n] = a_0[n]$ can be computed successive. At each scale $2^j, j > 0$ it decomposes the higher resolute approximation signal $a_j[n]$ into a coarser resolute approximation signal $a_{j+1}[n]$ and a detail signal $d_{j+1}[n]$ [11]:

$$\begin{aligned} a_{j+1} &= a_j * \bar{h}[2n], \\ d_{j+1} &= a_j * \bar{g}[2n], \end{aligned} \tag{1}$$

whereas d_{j+1} are the wavelet coefficients at resolution 2^{-j}. The other way around the original signal $a_0[n]$ can be reconstructed successive from the wavelet decomposition, called inverse discrete wavelet transform (IDWT):

$$a_j[n] = \breve{a}_{j+1} * h[n] + \breve{d}_{j+1} * g[n], \tag{2}$$

with

$$\breve{x}[k] = \begin{cases} x[n] & ,k = 2n \\ 0 & ,k = 2n+1 \end{cases}.$$

Mallat's pyramid algorithm is decimated: During the convolutions of the decomposition only every second filter coefficients is considered. This can be obtained by downsampling with the factor two after the convolution. Before the convolutions of the reconstruction zeros are inserted between every pair of values of the approximation and detail signals. This results in the upsampling with the factor two.

The "algorithme à trous" is translation-invariant and therefore not decimated. The DWT is computed with scale-dependent filters $\bar{h}_j$ and $\bar{g}_j$ obtained by inserting $2^j - 1$ zeros (french trous) between every pair of filter coefficients:

$$\begin{aligned} a_{j+1} &= a_j * \bar{h}_j[n], \\ d_{j+1} &= a_j * \bar{g}_j[n]. \end{aligned} \tag{3}$$

The appropriate scale-dependent reconstruction filters $\tilde{h}_j$ and $\tilde{g}_j$ are biorthogonal. The IDWT is

$$a_j[n] = \frac{1}{2}\left(a_{j+1} * \tilde{h}_j[n] + d_{j+1} * \tilde{g}_j[n]\right). \tag{4}$$

The DWT can be easily extended in two dimensions if the wavelet can be written as separable products of functions in spatial directions. Mallat [12] proposed a two-dimensional quadratic spline wavelet for the "algorithme à trous", which is often used in image processing. It decomposes the images, respectively, in two subbands with horizontal and vertical orientation and an approximation. Because the wavelet is separable the DWT can be computed by separable convolution in horizontal and vertical direction:

$$\begin{aligned} a_{j+1}[n_x, n_y] &= a_j * \bar{h}_j \bar{h}_j[n_x, n_y], \\ d^1_{j+1}[n_x, n_y] &= a_j * \bar{g}_j \delta[n_x, n_y], \\ d^2_{j+1}[n_x, n_y] &= a_j * \delta \bar{g}_j[n_x, n_y], \end{aligned} \tag{5}$$

with $\delta[n_x, n_y]$ discrete Dirac function.

The IDWT uses the reconstruction filters h_j, k_j and l_j:

$$a_j[n_x,n_y] = a_{j+1} * h_j h_j[n_x,n_y] + d^1_{j+1} * k_j l_j[n_x,n_y] + d^2_{j+1} * l_j k_j[n_x,n_y]. \quad (6)$$

Although the filters of the spline wavelet are short, the complexity for decomposition and reconstruction remains $O(N^2 \log_2 N)$. Hence, the “algorithme à trous” is still very computing intensive. Moreover the representation has $(2j+1)N^2$ values, which must be stored in memory. Especially in real-time applications, general-purpose processors could not deliver the necessary performance for the computation of the “algorithme à trous”. The need for a fast implementation is therefore obvious.

V. The GPU as Stream Processor

The advantages of the GPU for the computation of the DWT can be explained from the perspective of computer architecture. Because the GPU is a powerful stream processor and the DWT fits well to the stream processing model, the starting point is the stream processing model.

A stream program is expressed as a sequence of kernels that is applied to data streams. The data streams contain a set of elements of the same type (e.g. image pixel). Each element of each stream is processed equally by the kernels. A kernel can only access on its input and output streams.

Therefore the inheriting parallelism of applications and locality of their data can be expressed by a stream program and efficiently exploited by a stream processor. Because the kernels apply the same calculation to each element of the input streams kernels, several elements can be processed simultaneously and a large amount of data parallelism can be exploited. Within a kernel, independent operations can be executed in parallel to exploit instruction level parallelism (ILP). Finally, thread level parallelism can be exploited, because of the pipelined execution of the kernels [13]. On the one hand data parallelism reduces the necessary instruction bandwidth. On the other side the organisation of communication use the locality of stream processing to deliver the required memory bandwidth. Data, which is accessed only inside a kernel, is stored in local register. The communication via streams requires high bandwidth. To deliver high bandwidth the memory modules are placed close to the processing elements. Stream processing requires less global communication. Global communication is especially necessary for the communication with external I/O-devices, whose bandwidth is limited to the bandwidth of the interface. Because most of the data in global communication are also streams, the data access pattern is predictable and can be used to speed up the communication through block access. Nevertheless global communication is often the bottleneck for stream processing [14].

Generally speaking, architectures for efficient stream processing contain a large amount of processing elements (PEs) grouped in several cluster. The clusters support the task parallel execution of different kernels. Data parallelism and instruction level parallelism can be exploited by independent processing of different stream elements and instructions by the processing elements. Inside the clusters shared memory modules are available for temporary results. The streams are stored in memory modules close to the clusters which deliver a high bandwidth.

The GPU can be seen as a powerful stream processor. The previously described properties of stream architectures can be also found in modern GPU. Fig. 1 shows a block diagram of the GeForce-8-GPU with 112 Streaming Processor Cores (SP Cores), which are organised in 14 Streaming Multiprocessors (SMs). Two Multiprocessors build one independent Texture/Processor Cluster. Because the GPU is originally

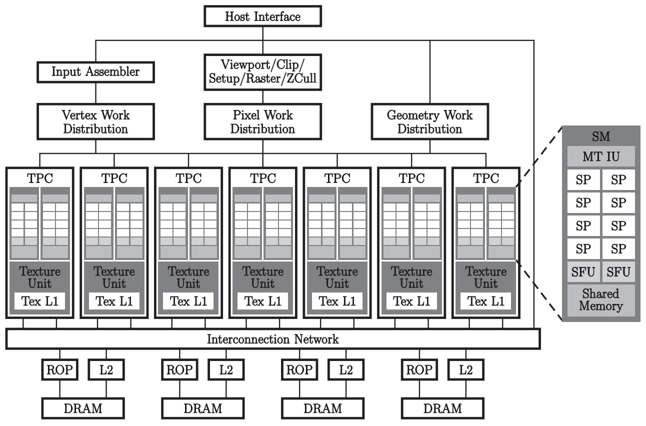


Fig. 1. NVIDIA GeForce-8-Series after [15, 16]. TPC: Texture/Processor Cluster; SM: Streaming Multiprocessor; MT IU: Multithreaded Instruction Unit; SP: Streaming Processor; SFU: Special Function Unit; Tex: Texture; ROP: Raster Operation Processor.

designed for computing of the graphics pipeline, GPU architectures additionally contain special fixed-function units for the efficient computation of some stages of the pipeline [15, 16].

The development of most programming languages and APIs targeted at writing graphics applications. The increasing flexibility of GPUs leads to higher level languages, which hide the underlying graphics programming languages and APIs. One advantage of using higher level languages is that they simply the use of the GPU especially for the development of general purpose applications. Currently the most widespread language in this domain is CUDA, which only supports NVIDIA hardware. On the other hand low level graphics programming using the APIs (OpenGL, Direct3D) may achieve better performance at the cost of explicit expression of non graphics applications in graphics idioms. Furthermore most of graphics hardware is supported by the APIs [17]. Our approach is based on OpenGL to support as much graphics hardware as possible.

Independent from the used programming language some restrictions of graphics architectures have to be considered by the programmer of streaming applications. These architectures efficiently process data-independent control-flow, while data-dependent jumps and branches decrease the performance. Because the images are processed in blocks, the access to pixels, which are far away, is slow. Data transfer between GPU and CPU is a bottleneck.

VI. Implementation

Our GPU implementation of the two-dimensional “algorithme à trous” is a direct implementation of the filter bank according to (5) and (6) by a set of fragment shaders. First of all an input image has to be transferred from main memory via PCI express bus to the video memory of the GPU. The input image is then successive decomposed by the wavelet filters. These two-dimensional filters are separable product filters, which are achieved by one-dimensional filtering first in horizontal and then in vertical direction. The filters are computed by fragment shaders that are executed for each input pixel. In order to get a one-to-one-mapping of texels and pixels, textured rectangles are drawn with an orthographic parallel projection. The resulting subbands are stored in a new texture, which become the input image for the calculation of the next scale. After the decomposition of the input image in the desired number of levels all subbands and the approximation of the coarsest level are hold in a set of textures. From this decomposition the input image can be reconstructed successively in the same way as the decomposition.

Textures and shaders supply the processing up to four color channels. For gray-level images, three color channels are used for storing and processing of the subbands and the approximation images. Therefore at the decomposition two filters and three filters at the reconstruction can be folded in parallel to increase the efficiency of using the GPU resources.

Our shaders of the wavelet filters are implemented in GLSL based on [18]. The texture coordinates are exclusively computed in these fragment shaders. Therefore no special vertex shaders are needed. Instead of stretching the filters with zeros for the “algorithme à trous”, the coefficients of the unstretched filters are multiplied with input values at the corresponding distances. These distances have to be set before applying the shaders. Because the dynamic range of the high-pass filter is [-2.0,2.0], the convolution sum is adapted to the texture range [0.0, 1.0] before storing the decomposition. While reconstruction this adaption is considered. To avoid errors at the boundaries at reconstruction a symmetrical periodic extended version of the image is used for decomposition. Beside the reconstructed image all subbands and approximation images can be transferred to CPU main memory for further processing.

VII. Results

A. Test setup

All tests were conducted on a PC running the Windows XP operation system. Because non real-time operation systems allocate the computing resources stochastically by priorities, computing times have a coincident variance. Therefore one measurement has not an adequate measure for our comparison. The measurements have to be repeated in a sufficient number to have approximate normally distributed computing times.

Our implementation is object-oriented based on Java and OpenGL. GLSL is used for the shader development. OpenGL is bound to Java by the JOGL class library, which only supports up to OpenGL 2.1. Our implementation uses rectangle textures with single precision floating point values and frame buffer object for direct rendering to textures. The CPU implementations are also object-oriented in Java. Because the interface of the CPU and GPU classes is equal, the implementation can easily be interchanged.

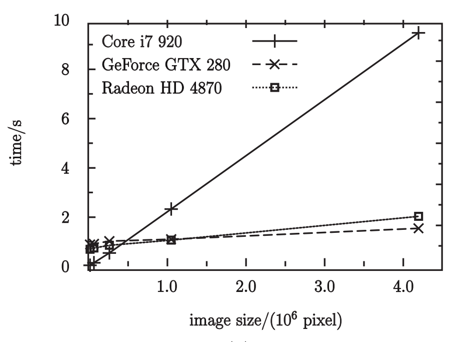
All the tests were performed on a PC with an Intel Core i7 920 CPU (4 Cores, 2.67 GHz, 4 GBytes DDR3 RAM), and an ATI Radeon HD 4870 or Nvidia GTX 280 GPU. The GPUs have 1 GBytes of video memory. The CPU implementations only use one single core. Five test images based on the often used well known Lena image [19] ranging from 128x128 to 2048x2048 are used. Furthermore, 200 images of a resolution build a test image sequence. For a more detailed analysis the execution times are subdivided in times for the initialisation, the data transfer, the decomposition and reconstruction.

B. Test results

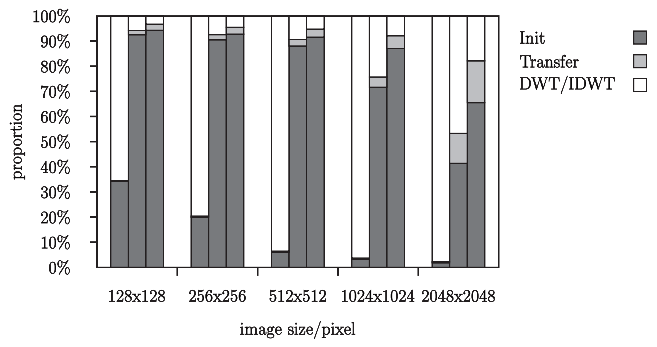
In the evaluation, the test images are decomposed in three levels and reconstructed from this decomposition. First the processing of gray-level images is considered. Fig. 2a shows the execution times of the CPU and GPU implementations. Due to GPU’s significant longer initialisation and transfer times the CPU implementation is faster for small image sizes. Otherwise the GPU implementation clearly outperforms the CPU implementation for larger image sizes. The arithmetic performance of the GPUs has a positive effect on the total execution time, if the initialisation and transfer times less dominates the total execution times on the GPU (Fig. 2b). This effect is greater on the processing of color images1. While initialisation and transfer times only increase modest, the arithmetic intensity is three times as high as for the processing of gray-level images. As shown in Fig. 2c, this effect can be also observed analysing the speedup of the GPU implementations relative to the CPU implementation. Furthermore it can be seen, that for large images also assuming a perfect partition of the computation on the CPU cores the GPU implementations are still faster.

Real-time multimedia applications are often more interested in the number of processed frame per second (fps), also called

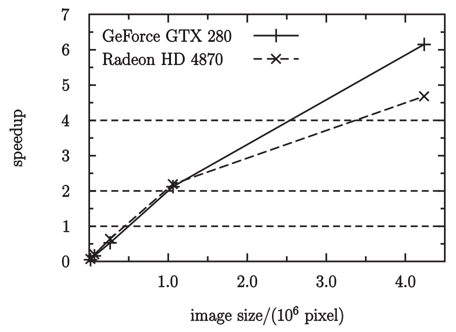
[1] Due to memory allocation problems the processing of 2048x2048 color images fails.



(a)



(b)



(c)

Fig. 2. DWT of gray-level images for different image sizes. (a) execution times CPU vs. GPU, (b) distribution of execution times: 1st column – Core i7 920, 2nd column – Radeon HD 4870, 3rd column – GeForce GTX 280, (c) speedup of execution times of GPUs relative to CPU.

image size/ pixel	Core i7 920	Radeon HD 4870	GeForce GTX 280
128x128	34,54	308,36	59,73
256x256	8,49	185,75	59,34
512x512	2,05	79,21	59,59
1024x1024	0,46	22,00	25,24
2048x2048	0,11	6,40	7,49

Tab. 1. CPU and GPU frame rates of the DWT of graylevel streams in fps.

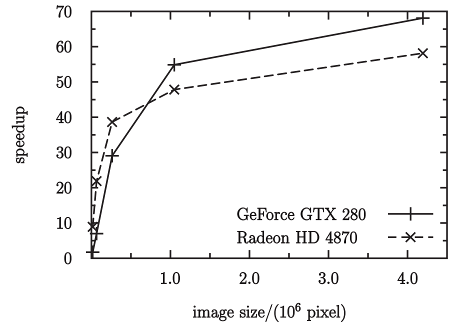


Fig. 3. DWT of gray-level streams for different image sizes:

as frame rate. Table 1 shows the frame rates of the CPU and GPU implementations. Because the initialisation is done only once at begin of the stream processing, the initialisation times can be neglected. Therefore higher frame rates for all image sizes relative to the CPU implementation can be achieved by the GPU implementations. This has the corresponding effect on the speedup of the GPU implementations relative to the CPU implementation. The maximum speedup of the GPU implementations for gray-level streams is 68.1 and for color streams 133.4. Hence, it is obvious, that a multicore CPU could not provide a comparable computing performance for the wavelet transform.

In practice frame rates are more interested than speedup. Mostly hundreds of frames per second mostly have not to be processed. But it can be approximated, how much resources can be used on the GPU for further applications.

C. *Wavelet-based edge detection*

For all scales the “algorithme à trous” compute horizontal and vertical subbands from different smoothed original images (approximation images). The used high-pass filters enhance high frequency image components like edges. Like the canny edge detector, the local maxima of the modulus of both subbands corresponds with sharp intensity variations in the original image [20]. Due to the smoothness often the scales 22-24 are used for edge detection.

Thus the wavelet decomposition at the corresponding scale is needed for computing the edges at scale 2j. First, our GPU implementation of the wavelet transform is used for the decomposition up to the scale 2^j. Then, additionally shaders for computing the magnitude and its maxima are applied to the stored wavelet decomposition.

The experimental results are comparable with the results of the wavelet transform above. Because only the decomposition is performed, the arithmetic intensity is lower, so that the achieved speedups are smaller. Nevertheless the speedups are still considerable. For gray-level streams a maximum speedup of 35.1 and for color streams of 81.9 is achieved.

D. *Wavelet denoising*

The wavelet transform can also be used for noise reduction. Because the image information is concentrated in a few, big wavelet coefficients, so that the important wavelet coefficients can be extracted by a threshold algorithm. The reconstruction from the subbands with the thresholded wavelet coefficients and the approximation of the coarsest scale leads to a denoised version of the original image [21]. For the calculation of the threshold a estimation of the noise level of the wavelet coefficients is needed.

The challenge for the implementation of wavelet denoising on the GPU is the calculation of the thresholds. Due to the sequential character of the estimators, the calculation has to be done by the CPU. The subbands have to be transferred to the CPU for threshold estimation, which is the bottleneck for the GPU implementation of the wavelet denoising. For that reason the achieved speedups are much smaller than before. We got a maximum speedup of 10.0 for gray-level streams and 13.3 for color streams.

VIII. Conclusions

In this paper we have presented a GPU implementation, which significantly speedup the computation of the 2D-DWT with the “algorithme à trous”. We have shown that the GPU can be used as a powerful stream processor to speed up streaming applications like the DWT. Our implementation was based on OpenGL to support as much graphics hardware. The object-oriented approach can be easily integrated in new applications. This is shown in two applications, which perform the wavelet-based edge detection and wavelet denoising of images. Experimental results show, that initialisation and data transfer times could dramatically reduce the GPU performance, if it can’t be hided by the application. Our future work will be in further improvements of the implementation and an OpenCL implementation.

IX. Conclusion

A conclusion section is not required. Although a conclusion may review the main points of the paper, do not replicate the abstract as the conclusion. A conclusion might elaborate on the importance of the work or suggest applications and extensions.

References

[1] Owens, J.D., Sengupta, S., and Horn, D., “Assessment of Graphic Processing Units (GPUs) for Department of Defense (DoD) Digital Signal Processing (DSP) Applications”, Technical Report, ECE-CE-2005-3, Computer Engineering Research Laboratory, University of California, Davis, October 2005.

[2] Hopf, M., and Ertl, T., “Accelerating 3D Convolution using Graphics Hardware”, IEEE Visualization '99, pp. 471-474, IEEE Computer Society, October 1999.

[3] Hopf, M., and Ertl, T., “Hardware Based Wavelet Transformations”, Vision, Modeling and Visualisation '99, pp. 317-328, IEEE and GI Infix Press, November 1999.

[4] Tenllado, C., Lario, R., Prieto, M., and Tirado, F. “The 2D Discrete Wavelet Transform on Programmable Graphics Hardware”, Proceedings of the 4th IASTED International Conference on Visualization, Imaging and Image Processing (VIIP '04), pp. 808-813, ACTA Press, September 2004.

[5] Wang, J., Wong, T.-T., Heng, P.-A., and Leung, C.-S., “Discrete Wavelet Transform on GPU”, Proceedings of ACM Workshop on General-Purpose Computing on Graphics Processors (GP2 2004), C-41, July 2004.

[6] Adams, M.D., and Kossentini, F. “JasPer: A software-based JPEG-2000 codec implementation”, Proceedings of IEEE International Conference on Image Processing (ICIP '00), vol. 2, pp. 53-56, October 2000.

[7] Adams, M.D., and Ward, R.K., “JasPer: A portable flexible open-source software tool kit for image coding/processing”, Proceedings of the 2004 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP '04), vol. 5, pp. 241-244, October 2004.

[8] Christopoulos, C., Skodras, A., and Ebrahimi, T., “The JPEG2000 still image coding system: An overview”, IEEE Transactions on Consumer Electronics, vol. 46, no. 4, pp. 1103-1127, November 2000.

[9] Wong, T.T., Leung, C.S., Heng, P.A., and Wang, J., “Discrete Wavelet Transform on Consumer-Level Graphics Hardware”, IEEE Transactions on Multimedia, vol. 9, no. 3, pp. 668-673, April 2007.

[10] Holschneider, M., Kronland-Martinet, R., Morlet, J., and Tchamitchian, P., “A real-time algorithm for signal analysis with the help of the wavelet transform”, in: “Wavelets, Time-Frequency Methods and Phase Space”, pp. 289-297, Springer, Berlin, 1989.

[11] Mallat, S., “A Theory for Multiresolution Signal Decomposition: The Wavelet Representation”, IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI), vol. 11, no. 7, pp. 674-693, July 1989.

[12] Mallat, S., and Zhong, S., “Complete Signal Representation With Multiscale Edges”, Technical Report, no. 483 Courant Institute of Mathematical Science, New York University, New York, December 1989.

[13] Dally, W.J., Kapasi, U.J., Khailany, B., Ahn, J.H., and Das, A., “Stream Processors: Programmability with Efficiency”, ACM Queue, vol. 2, no. 1, pp. 52-62, March 2004.

[14] Khailany, B., Dally, W.J., Rixner, S., Kapasi, U.J., Owens, J.D., and Towles, B., “Exploring the VLSI Scalability of Stream Processors”, Proceedings of 9th International Symposium on High Performance Computer Architecture (HPCA '03), pp. 153-164, IEEE, February 2003.

[15] Lindholm, E., Nickalls, J., Oberman, S., and Montrym, J., “NVIDIA Tesla: A unified Graphics and Computing Architecture”, IEEE Micro, vol. 28, no. 2, pp. 39-55, March/April 2008.

[16] Nickolls, J., Buck, I., Garland, M., and Skadron, K., “Scalable Parallel Programming with CUDA”, ACM Queue, vol. 6, nor. 2, pp. 40-53, March/April 2008.

[17] Blythe, D., “Rise of the Graphics Processor”, Proceedings of the IEEE, vol. 96, no. 5, pp. 761-778, May 2008.

[18] Rost, R.J., “OpenGL® Shading Language”, Addison-Wesley, 2nd edition, Upper Saddle River [et al.], January 2006.

[19] Hutchison, J., “Culture, Communication, and an Information Age Madonna”, IEEE Personal Communication Society Newsletter, vol. 45, no. 3, pp. 1 and 5-7, May/June 2001.

[20] Mallat, S., and Zhong, S., “Characterization of Signals from Multiscale Edges”, IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI), vol. 14, no. 7, pp. 710-732, July 1992.

[21] Donoho, D.L., and Johnstone, I.M., “Ideal Spatial Adaption via Wavelet shrinkage”, Biometrica, vol. 81, pp. 425-455, 1994.