2014 IEEE 26th International Symposium on Computer Architecture and High Performance Computing Workshops

2-D Discrete Wavelet Transform Using GPU

Michal Kucis David Barina Michal Kula Pavel Zemcik
Faculty of Information Technology
Brno University of Technology
Czech Republic

***Abstract*—With the wide spread of the discrete wavelet transform, the need for its efficient implementation becomes increasingly important. This work presents an improved version of an algorithm suitable to compute the 2-D discrete wavelet transform on GPU. Depending on the GPU platform, it is suitable to split the 2-D transform computation into separated horizontal and vertical passes. Considering the horizontal passes, we have examined and chosen the best performing method among the already known ones. Furthermore, we have adapted this method for an existing algorithm computing the vertical transform pass. This step helps to reduce several synchronizations and arithmetic operations in the utilized computation scheme. For large data, the proposed vertical method achieves speed-up about 30 % compared to the current state of the art methods. In contrast to previously published works, the presented approach is built on the OpenCL parallel programming framework.**

I. Introduction

The discrete wavelet transform (DWT) is a mathematical tool which is suitable to decompose discrete signal into several frequency components. It is frequently used as a basis of sophisticated compression algorithms. This paper focuses on the CDF 9/7 wavelet which is often used for image compression (e.g., JPEG 2000 standard). Responses of this wavelet can be computed by a convolution with two FIR filters, one with 7 and the other with 9 taps. In case of two-dimensional transform, the transform can be realized using a separable decomposition scheme. In this paper, we present several algorithms for 2-D transform computation suitable for modern GPUs.

In present personal computers, programmable graphics cards are almost always found. OpenCL is a framework for programming of heterogeneous computer systems, e.g. modern graphics processing units (GPU) found in personal computers, servers or mobile devices. When compared to CUDA framework, CUDA is limited to Nvidia hardware while OpenCL is not platform dependent. The performance analysis [1] remarks that OpenCL offers similar performance to CUDA in general when compared fairly.

Several algorithms for the 2-D DWT computation using GPU have been published in the last decade. Some of them used the pixel shader through the Cg programming language. These were able to take advantage of SIMD operations offered by shader units. Other algorithms were built over the CUDA framework. We are not aware of any approach that use the OpenCL framework.

In this paper, we present several algorithms for 2-D DWT computation focusing on the parallel capabilities of programmable GPUs. Our implementation is based on the OpenCL framework. All the methods presented in this paper are evaluated using Nvidia GeForce GTX 580 graphics card equipped with 3072 MiB RAM and 512 streaming processors and Nvidia Quadro NVS 4200M graphics card equipped with 1024 MiB RAM and 48 streaming processors. Only the forward transform is evaluated since the inverse one has a symmetric nature and performs almost identically. We have also evaluated only one level of the DWT decomposition as the others again perform almost identically. In the chosen memory layout, the sub-bands are interlaced.

The rest of the paper is organised as follows. Related Work section summarizes the state of the art, especially existing GPU implementations. Proposed Approach section reviews significant algorithms and presents the proposed method. Finally, Conclusion section summarizes the paper and outlines the future work.

II. Related Work

The discrete wavelet transform [2] (DWT) is a mathematical tool suitable to decompose a signal into low-pass and high-pass frequency components. Such a decomposition can be performed at several scales resulting in a multi-scale signal representation. It is often used as a basis for sophisticated compression algorithms. A basis of such a transform consists of dilated and shifted wavelets. The Cohen-Daubechies-Feauveau [3] (CDF) 9/7 wavelet is a popular one as used, e.g., in JPEG 2000 image compression standard. One level of the discrete wavelet transform can be computed using the convolution with two mirror filters (a high-pass and a low-pass one). According to the total number of arithmetic operations, the more efficient computational scheme – the lifting – introduced by W. Sweldens in [4] exists. Using this scheme, the whole signal can be transformed in-place. In [5], I. Daubechies and W. Sweldens factored CDF 9/7 wavelet into four successive lifting steps employing short symmetric two-taps FIR filters.

For understanding of the following text, it may be important to understand the lifting scheme in more detail. Any discrete wavelet transform with finite filters can be factored into a finite sequence of N pairs of predict and update convolution operators P_n and U_n. Each predict operator P_n corresponds to a filter $p_i^{(n)}$ and each update operator U_n to a filter $u_i^{(n)}$. These operators alternately modify even and odd signal coefficients.

$$P_n(z) = \sum_{i=-l_n}^{g_n} p_i^{(n)} z^{-i} \tag{1}$$

$$U_n(z) = \sum_{i=-m_n}^{f_n} u_i^{(n)} z^{-i} \tag{2}$$

978-1-4799-7014-8/14 $31.00 © 2014 IEEE
DOI 10.1109/SBAC-PADW.2014.13

The discrete wavelet transform was also extended [6] to two (and more) dimensions. Specifically, the classical 2-D DWT is separable to series of 1-D transforms performed successively on rows and columns (or vice versa). For various requirements, different strategies of 2-D DWT implementation were developed. For example, the simplest row-column methods transform the whole image at once. Furthermore, the block-based methods transform the image using smaller blocks utilizing the row-column method inside. Finally, the pipelined methods such as [7] transform the image using column strips while employing the sliding window on them. Inside this window, the row and column transforms are combined together in a way that a vertical transform is interleaved on multiple columns. This concept was also extended to whole image resulting into the single-loop approach [8].

Implementation of 2-D DWT was also studied on modern programmable graphics cards. In this scenario, the input image have to be initially transferred from main memory into memory on the graphics card. Similarly, the resulting coefficients have to be transferred back.

OpenCL is a framework for general-purpose parallel programming across multiple device types (like GPUs, CPUs, etc.) and platforms. In this framework, a platform independent executable program is called the kernel. The kernel is executed on required number of threads (work-items) that identify their data and control flow by their N-dimensional indices. These threads are organized into work-groups with identical user-defined number of threads. The threads in such a group can cooperate with each other through local memory and barriers. Threads executing a kernel have access to: global memory – device memory that is accessible to all threads (like main GPU memory); local memory – small memory region that is shared by threads in work-group; constant memory – small memory that remains constant during the kernel execution; private memory – the private thread memory. Optionally, a device can support additional functionalities like textures, double type operations, etc.

In recent GPU architectures, the GPU contains the thread scheduler, multiprocessors, L2 cache and a memory controller. The thread scheduler allocates as much work-groups to multiprocessors as their resources allow. Thus, the resources like local memory size should be minimized. The multiprocessor contains blocks of processors, warp schedulers, local memory, load store units, etc. The allocated work-groups created by OpenCL framework is then divided into warps (hardware blocks with 32 threads). Execution instructions of these warps on blocks of processors are provided using warp schedulers dynamically. Due to fact that each instruction is executed on whole warp (half-warp on some architectures) at once, recommendations for ensuring good performance of memory operations exist. Global memory indices in warp should be coalesced. Otherwise, addition memory operations are executed. The local memory is organized into banks. Access to same banks from warp causes serialization. The serialization of local memory operations and uncoalesced global memory access can cause a performance degradation.

In [9] and [10], Ch. Tenllado *et al.* adapted the discrete wavelet transform on pixel (fragment) shaders of GPU. They used the Cg programming language and mapped the input image into textures. The authors compared convolution-based and lifting scheme implementations of CDF 9/7 discrete wavelet transform. The pixel processors support SIMD operations (4-element wide in this case). Using the convolution, the authors used a rearrangement step in order to allow to filter two image rows/columns in parallel. The results of this comparison speaks slightly in favor of convolution scheme. Moreover, the authors compared these results with corresponding CPU implementation using the CDF 9/7 wavelet. Ignoring CPU-GPU data transfer times, the GPU version significantly outperforms the CPU counterpart. Finally, the authors state that the data transfers between the CPU and the GPU are the major bottleneck. However, these works are now obsolete as an instruction set of the modern GPUs does not contain the real SIMD instructions.

The other authors attempted to take advantage of the GPU using the CUDA programming model in [11], [12] or [13]. In [11], the convolution scheme is applied on each row. Then, the image matrix is transposed and the convolutions are applied on each column. Finally, the image is transposed back. The authors point out that important reductions of execution time are obtained for the CUDA version even when they take into account the time needed to copy data and results to and from the GPU memory. However, their CPU implementation seems to be naive compared to the state of the art methods, e.g. [8]. The latter two papers are focused on CDF wavelets and the lifting scheme. Their implementations splits the image into small tiles and performs several independent transforms on each of them. Moreover, in [13], another implementation performs the horizontal transform on the whole image. The horizontal transform is followed by transposition and by vertical filtering. The authors proposed omission of mutual thread synchronization at the cost of loading of more input pixels per each thread. Furthermore, the author of [12] consider the coalesced memory accesses to be crucial for a transform performance.

In [14] and [15], V. Galiano *et al.* compared several CUDA implementations of DWT. They used the CDF 9/7 wavelet and convolution-based algorithm on entire rows/columns. Their fastest implementation uses the coalesced memory access.

In [16], W. J. Laan *et al.* accelerated the Dirac video codec using the CUDA platform. Their DWT implementation is based on the lifting scheme. The authors highlight the coalesced memory access. In the vertical filtering, they divided the image into vertical strips and used a sliding window technique within each strip. However, this paper does not discuss the implementation of DWT in detail. In [17], W. J. Laan *et al.* provided a detailed analysis of the DWT implementation using the lifting scheme on the CUDA platform. They focused on several wavelets (including CDF 9/7) and used a sliding window approach within strips. Their design is a hybrid method between the row-column and block-based methods. Moreover, they implemented the methods for 2-D and 3-D data and compared to optimized CPU counterparts. Also here, the authors point out the importance of coalesced memory accesses.

As it can be seen, the problem of efficient 2-D discrete wavelet transform implementation on GPU was widely studied. Despite this fact, we see a gap in existing implementations. In the section below, we propose several improvements that lead to additional speedups.

III. Proposed Approach

In this section, several existing algorithms for the discrete wavelet transform computation are analysed. Initially, algorithms for horizontal pass of the transform are presented. For further experiments, the best performing algorithm for this pass is adopted. Furthermore, a vertical pass of the transform is discussed. Here, we have proposed several improvements over the state-of-the-art algorithm yielding to an additional speed-up.

In all of the algorithms below, two separated passes needed to compute the 2-D transform are considered. These are referred here to as a horizontal and a vertical pass. In general, both of these passes can share intermediate results between threads that access adjacent data. However, this sharing introduces some requirements for their mutual synchronization. Another approach might be to not share the intermediate results at all for the price that some calculations become redundant. In all cases, the coalesced memory access was used wherever it was possible.

A. Horizontal Pass

At the beginning, we have focused on an algorithm for computation of the horizontal pass. We have implemented and compared plenty of existing algorithms. The most prominent of these algorithms are presented below. All of the implemented algorithms use up to 256 threads for each work group.

Considering the horizontal pass, it can generally consist of the following steps. Firstly, transfer a data row from the global memory to the local memory. Secondly, perform the horizontal transform using data in the shared memory. Thirdly, transfer the computed row of result from the shared memory to the global one. Note that the local memory is shared for the group of threads. An access to this local memory is much faster, but it is limited by relatively small size and it is shared just along one single work group.

Considering the first relevant algorithm, each thread in the work group computes a pair of the output coefficients by the convolution scheme. Each thread loads nine coefficients from the local memory and computes a corresponding pair of resulting coefficients (responses to the FIR filters). The implementation was earlier described in detail in [14], [15]. We refer to as *Horiz-Galiano2011* in this paper.

The second algorithm uses lifting scheme instead of convolution. In this case, every thread loads nine coefficients from local memory and computes all the required computation by itself. No intermediate results are shared between threads. This implementation was described in [13]. We will further denote it as *Horiz-Blazewicz2012-1*. The data-flow graph for a single thread is shown in Fig. 1.

The other algorithm computes 4 pairs of the output coefficients instead of one by each thread. This algorithm employs the lifting scheme which was described in [13]. It is further denoted as *Horiz-Blazewicz2012-4*. The algorithm does not share the intermediate results between threads and does not require synchronization barriers. The implementation requires to load 15 coefficients from the local memory. A group of threads loads and process a single row of the input image.

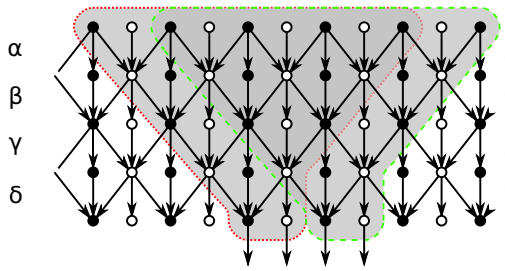

Fig. 1. Lifting scheme of CDF 9/7 wavelet showing the calculation performed by a single thread (dotted and dashed). No intermediate results are shared between threads. No synchronization is required.

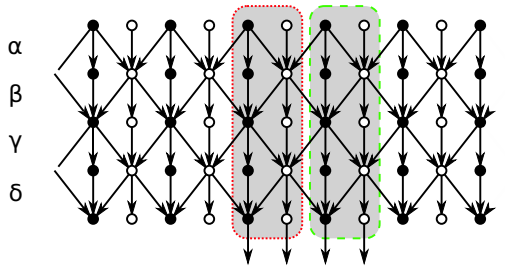

Fig. 2. Lifting scheme of CDF 9/7 wavelet showing the calculation performed by a single thread (dotted and dashed). Intermediate results are shared between neighbouring threads. Synchronization is required.

The last of the implemented algorithms uses the lifting scheme. The algorithm was described in [16], [17] and it is referred to as *Horiz-Laan2009* here. There is no redundant computation considering different threads. However, this implementation requires additional synchronizations in the lifting steps. The algorithm works in the following way. Each of the threads loads three input coefficients from the local memory and performs elementary lifting step. Then, neighbouring threads exchange the intermediate results. These two steps are repeated further. The data-flow graph for a single thread is shown in Fig. 2.

The previously described algorithms are limited to a certain resolution of the input image. One can compute the transform of the input image up to 512 pixels wide if the maximum number of threads in the work group is 256 and if two output coefficients are computed by a single thread. To overcome this limitation, the algorithms are extended in the following way. A group of threads processes the left-most coefficients in a row from the global memory, computes horizontal transform and then saves results into the global memory. The same group processes a following block of coefficients, where previously loaded and processed coefficients are reused. This approach reduces a global memory access and some of the redundancy.

We have implemented and evaluated all of the algorithms described above. The results of the comparison are plotted in Fig. 3 and 4. Several measurements are listed in the TABLE I and II. The *Blazewicz2012-4* algorithm proved to be the fastest one across a whole range of image sizes. This result is caused mainly by maximizing a number of arithmetic operations by using each thread as pointed in [13].

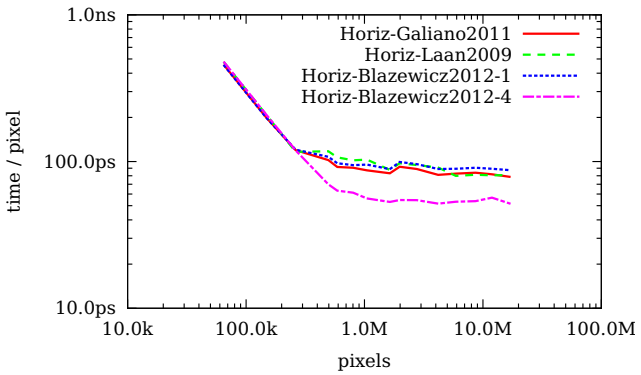

Fig. 3. GeForce GTX 580. Horizontal pass algorithms.

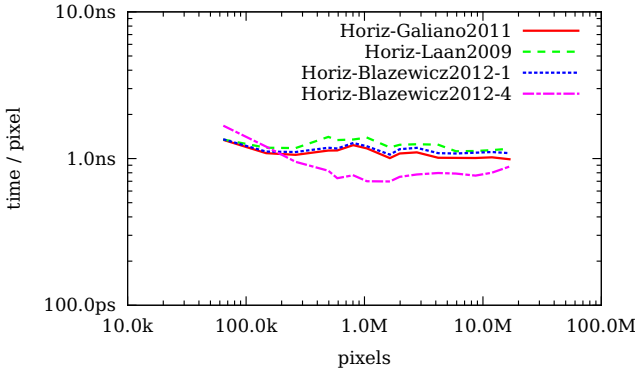

Fig. 4. NVS 4200M. Horizontal pass algorithms.

B. Vertical Pass

In this part, we focus on the vertical pass of the transform. The simplest approach is to use same algorithms that are used in the horizontal pass but transform columns instead of rows. However, such an approach does not consider coalescent access to the global memory that consequently causes a performance degradation. W. J. Laan *et al.* [17] states that this approach is $10\times$ slower than more complex solution, that will be described later.

A more complex approach was presented in [13]. Initially, this approach transposes input data. After that, the unchanged horizontal pass algorithm is performed. Finally, the resulting data are transposed again. We have implemented this algorithm in the following way. The horizontal pass *Blazewicz2012-4* algorithm is used in the heart of the algorithm. We refer this approach to as *Vert-Blazewicz2012*. This process uses fast coalescent access to the global memory and creates lot of working groups that help to utilize computing resources. On the other hand, every transposition requires a separate kernel run. This causes access to global memory for $6-3\times$ for reading and $3\times$ for writing per every element of the data.

Furthermore, we have adopted a vertical transform algorithm presented in [16]. This original algorithm is denoted as *Vert-Laan2009*. The algorithm divides the input image to multiple vertical strips. A width of the strip is based on the size of the successive bytes defined by coalescent memory

algorithm	1 Mpel	10 Mpel
Horiz-Galiano2011	87.1	81.9
Horiz-Laan2009	103.0	80.6
Horiz-Blazewicz2012-1	95.4	89.3
Horiz-Blazewicz2012-4	**56.0**	**56.7**

TABLE I. GTX 580. HORIZONTAL PASS. PICOSECONDS PER PIXEL.

algorithm	1 Mpel	10 Mpel
Horiz-Galiano2011	1173.8	1020.2
Horiz-Laan2009	1389.9	1142.2
Horiz-Blazewicz2012-1	1212.2	1108.4
Horiz-Blazewicz2012-4	**700.7**	**800.8**

TABLE II. NVS 4200M. HORIZ. PASS. PICOSECONDS PER PIXEL.

access. We use width of the strip of 32 coefficients. Every strip is processed by a single work-group of threads. Inside of such a strip, a sliding window approach is used. The width of the sliding window is same as the width of the strip (32 coefficients), the height of the window is 20 coefficients. The algorithm works as follows:

1) The window is placed on the top of the strip, 17 rows are copied from the global memory to the local one. The rows in the window are processed by lifting scheme, where result values are in first 13 rows and 4 rows contain intermediate results. The result values are copied into the global memory, intermediate results stay in the local one.
2) The sliding window is moved by 16 rows down. Missing rows (not in the local memory) are loaded from the global memory.
3) In the window, the lifting scheme is performed .
4) The results are moved to the global memory, the rows with intermediate results are still in the local memory.
5) This process is repeated until the window reach the border of the strip. The last remaining section is processed by a similar process like previous one.

The above described approach performs the lifting computation using barriers after each lifting step. This algorithm is similar to the horizontal pass *Horiz-Laan2009* described before. On the plus side, this approach reduces access to the global memory just for one read and one write per each element. On the negative side, the algorithm creates a small count of work-groups, which can be problematic at the modern GPU with many of multiprocessors.

Furthermore, we have experimented with a different adaptation of this algorithm. As a result, we have created a faster adaptation. The core of our proposed approach is the same as the previously described algorithm presented in [16]. As in the previous case, we have used the sliding window method to process entire tile by a single work-group. The *Horiz-Blazewicz2012-4* algorithm proved to be the fastest one considering the horizontal pass. Therefore, we have adapted this approach to perform the lifting scheme in the vertical direction. This approach requires extension of the sliding window height to process 8 coefficients by a single thread in one particular window position. Consequently, the approach requires the sliding window of 71 rows height. No intermediate results are passed between lifting steps nor different window positions. In the first window position (on the top of strip), 64 rows are computed. These values are computed by the same scheme as the one used in *Horiz-Blazewicz2012-4*. It is required to load

67 rows from global memory to the local one (the window). After processing and moving the results to the global memory, the window is moved by 64 rows down to process additional 64 rows. It is required to have 71 rows in the window to perform the transform correctly. The most bottom part of the strip is processed by separate algorithm to process borders correctly. This approach eliminates part of the synchronisations at the cost of adding some redundant arithmetic operations and increasing the local memory consumption. We refer this adaptation such as *Vert-Proposed*.

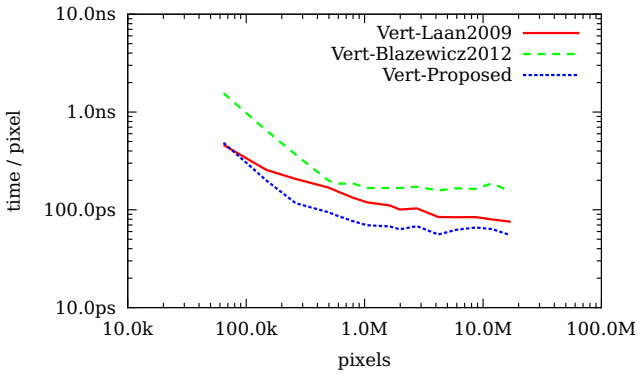

Fig. 5. GeForce GTX 580. Vertical pass algorithms.

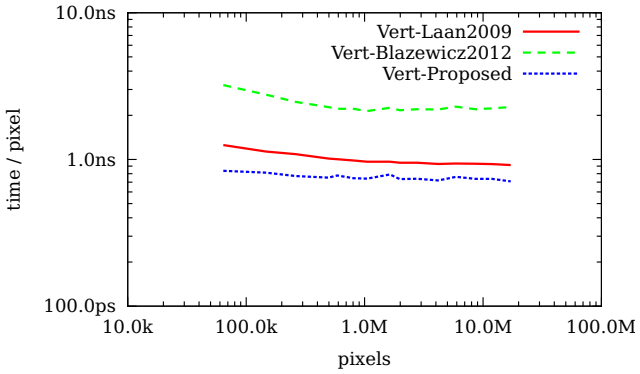

Fig. 6. NVS 4200M. Vertical pass algorithms.

Finally, we have evaluated all of the vertical pass algorithms described in this section. The results are plotted in Fig. 5 and 6. Several measurements are listed in the TABLE III and IV. In all cases, medians of ten measurements are used. The proposed *Vert-Proposed* algorithm has proved to be the fastest one. This algorithm achieved an average speed-up at least 30 % compared to the *Vert-Laan2009* algorithm which is considered to be the state-of-the-art method. The average speedup of *Vert-Proposed* relative to the *Vert-Laan2009* implementation are shown in TABLE V.

algorithm	1 Mpel	10 Mpel
Vert-Laan2009	119.3	79.6
Vert-Blazewicz2012	167.0	186.5
Vert-Proposed	**69.6**	**63.5**

TABLE III. GTX 580. VERTICAL PASS. PICOSECONDS PER PIXEL.

C. Entire Transform

Finally, we decided to evaluate the full transform computation. It may generally consists of the two memory transfers

algorithm	1 Mpel	10 Mpel
Vert-Laan2009	965.3	930.2
Vert-Blazewicz2012	2139.0	2231.7
Vert-Proposed	**739.4**	**737.9**

TABLE IV. NVS 4200M. VERT. PASS. PICOSECONDS PER PIXEL.

graphics card	avg. speed-up
GeForce GTX 580	50 %
NVS 4200M	31 %

TABLE V. VERTICAL PASS. AVERAGE PERCENTAGE SPEEDUPS.

(on-board memory ↔ video memory) and two transform passes (the horizontal and the vertical one). It would not be entirely fair to include the memory transfers in the final comparison. For this reason, the cumulative flow diagram is shown here separating the memory transfers and the two transform passes. The final comparison is plotted in Fig. 7 and 8.

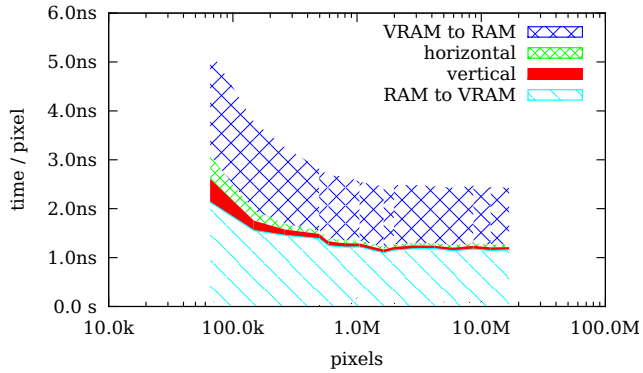

Fig. 7. GeForce GTX 580. Entire transform.

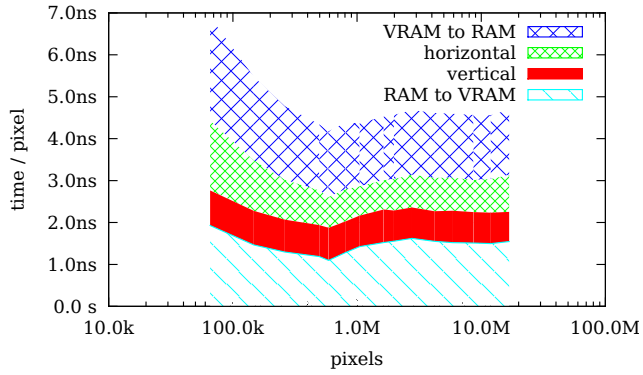

Fig. 8. NVS 4200M. Entire transform.

It would be interesting to compare the GPU implementations described in this section with a tuned CPU counterpart. For this purpose, the state of the art CPU implementation [18] with fused vertical and horizontal passes was used. This implementation utilizes SIMD instructions and have 4 threads running simultaneously. The implementation was evaluated using mainstream PC with Intel x86 CPU. Specifically, Intel Core2 Quad Q9000 running at 2.0 GHz was used. This CPU has 32 kiB of level 1 data cache and 3 MiB of level 2 shared cache (two cores share one cache unit). The comparison is summarized in Fig. 9 and 10.

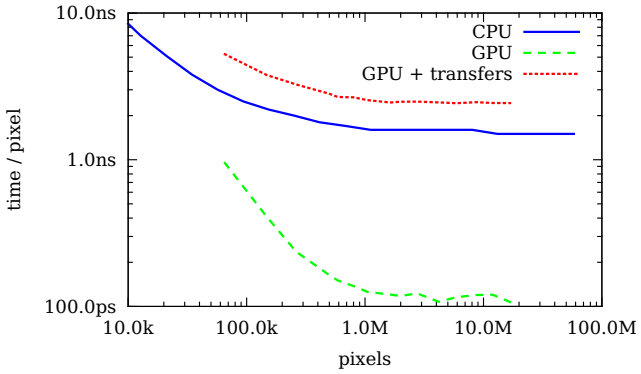

Fig. 9. GeForce GTX 580. Entire transform compared to CPU.

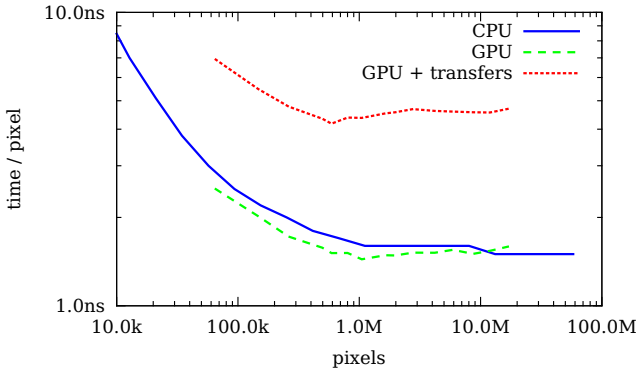

Fig. 10. NVS 4200M. Entire transform compared to CPU.

Note that both GPUs are equipped with an older version (2.0) of PCI-Express $\times 16$ bus. Due to this fact newer GPUs with current version (3.0) of PCI-Express $\times 16$ bus may have $2\times$ faster VRAM to RAM and RAM to VRAM transfers.

IV. Conclusion

We have presented a novel approach to 2-D wavelet transform using GPU reaching an average speedup at least 30 % on tested graphics cards. This approach is focused on utilization of parallel capabilities of modern GPUs. All the methods compared in this paper were evaluated using GeForce GTX 580 and NVS 4200M cards. In addition, we have compared these methods with state of the art implementation on CPU.

In more detail, the computation of single level of the transform is split into horizontal and vertical passes. Initially, we have adapted an existing algorithm to perform the horizontal pass without synchronizations. Furthermore, we have incorporated this algorithm into other existing technique performing the vertical pass using the sliding window. Additionally, we have extended this sliding window height to process 8 coefficients by a single thread in one particular window position. This step helps to reduce the computing redundancy. Finally, we have evaluated the performance of the entire transform and also compared it with transform performed using CPU.

Further work could focus on multi-level decompositions, improvement of the proposed vertical algorithms in order to utilize more work-groups or an exploration of fusion of the horizontal and vertical passes into a single one.

Acknowledgements: This work has been supported by the IT4Innovations Centre of Excellence (no. CZ.1.05/1.1.00/02.0070) and the TACR project V3C (no. TE01020415).

References

[1] J. Fang, A. L. Varbanescu, and H. Sips, "A comprehensive performance comparison of CUDA and OpenCL," in *Proceedings of the 2011 International Conference on Parallel Processing*, ser. ICPP '11. IEEE Computer Society, 2011, pp. 216–225.

[2] S. Mallat, *A Wavelet Tour of Signal Processing: The Sparse Way. With contributions from Gabriel Peyré.*, 3rd ed. Academic Press, 2009.

[3] A. Cohen, I. Daubechies, and J.-C. Feauveau, "Biorthogonal bases of compactly supported wavelets," *Communications on Pure and Applied Mathematics*, vol. 45, no. 5, pp. 485–560, 1992.

[4] W. Sweldens, "The lifting scheme: A custom-design construction of biorthogonal wavelets," *Applied and Computational Harmonic Analysis*, vol. 3, no. 2, pp. 186–200, 1996.

[5] I. Daubechies and W. Sweldens, "Factoring wavelet transforms into lifting steps," *Journal of Fourier Analysis and Applications*, vol. 4, no. 3, pp. 247–269, 1998.

[6] S. Mallat, "A theory for multiresolution signal decomposition: the wavelet representation," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 11, no. 7, pp. 674–693, 1989.

[7] S. Chatterjee and C. D. Brooks, "Cache-efficient wavelet lifting in JPEG 2000," in *Proceedings of the IEEE International Conference on Multimedia and Expo (ICME)*, vol. 1, 2002, pp. 797–800.

[8] R. Kutil, "A single-loop approach to SIMD parallelization of 2-D wavelet lifting," in *Proceedings of the 14th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, 2006, pp. 413–420.

[9] C. Tenllado, R. Lario, M. Prieto, and F. Tirado, "The 2D discrete wavelet transform on programmable graphics hardware," in *Visualization, Imaging and Image Processing Conference 2004*, 9 2004, pp. 808–813.

[10] C. Tenllado, J. Setoain, M. Prieto, L. Pinuel, and F. Tirado, "Parallel implementation of the 2D discrete wavelet transform on graphics processing units: Filter bank versus lifting," *IEEE Transactions on Parallel and Distributed Systems*, vol. 19, no. 3, pp. 299–310, 2008.

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

[12] J. Matela, "GPU-based DWT acceleration for JPEG2000," in *Annual Doctoral Workshop on Mathematical and Engineering Methods in Computer Science*. NOVPRESS s.r.o., 2009, pp. 136–143.

[13] M. Błażewicz, M. Ciżnicki, P. Kopta, K. Kurowski, and P. Lichocki, "Two-dimensional discrete wavelet transform on large images for hybrid computing architectures: GPU and CELL," in *Euro-Par 2011: Parallel Processing Workshops*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, vol. 7155, pp. 481–490.

[14] V. Galiano, O. López, M. Malumbres, and H. Migallón, "Improving the discrete wavelet transform computation from multicore to GPU-based algorithms," in *Proceedings of the 11th International Conference on Computational and Mathematical Methods in Science and Engineering (CMMSE)*, 2011, pp. 544–555.

[15] ——, "Parallel strategies for 2D discrete wavelet transform in shared memory systems and GPUs," *The Journal of Supercomputing*, vol. 64, no. 1, pp. 4–16, 2013.

[16] W. van der Laan, J. B. T. M. Roerdink, and A. Jalba, "Accelerating wavelet-based video coding on graphics hardware using CUDA," in *Proceedings of 6th International Symposium on Image and Signal Processing and Analysis, 2009. ISPA 2009*, 9 2009, pp. 608–613.

[17] W. J. van der Laan, A. C. Jalba, and J. B. T. M. Roerdink, "Accelerating wavelet lifting on graphics hardware using CUDA," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 1, pp. 132–146, 2011.

[18] D. Barina and P. Zemcik, "Diagonal vectorisation of 2-D wavelet lifting," in *IEEE International Conference on Image Processing 2014 (ICIP 2014)*, Paris, France, 10 2014, pp. 2978–2982, accepted.