Two-Dimensional Discrete Wavelet Transform on Large Images for Hybrid Computing Architectures: GPU and CELL

Marek Błażewicz, Miłosz Ciżnicki, Piotr Kopta,
Krzysztof Kurowski, and Paweł Lichocki

Poznan Supercomputing and Networking Center,
Noskowskiego 10, 61-704 Poznań, Poland
{marqs,miloszc,krzysztof.kurowski,pawel.lichocki}@man.poznan.pl

Abstract. The Discrete Wavelet Transform (DWT) has gained the momentum in signal processing and image compression over the last decade bringing the concept up to the level of new image coding standard JPEG2000. Thanks to many added values in DWT, in particular inherent multi-resolution nature, wavelet-coding schemes are suitable for various applications where scalability and tolerable degradation are relevant. Moreover, as we demonstrate in this paper, it can be used as a perfect benchmarking procedure for more sophisticated data compression and multimedia applications using General Purpose Graphical Processor Units (GPGPUs). Thus, in this paper we show and compare experiments performed on reference implementations of DWT on Cell Broadband Engine Architecture (Cell B.E) and nVidia Graphical Processing Units (GPUs). The achieved results show clearly that although both GPU and Cell B.E. are being considered as representatives of the same hybrid architecture devices class they differ greatly in programming style and optimization techniques that need to be taken into account during the development. In order to show the speedup, the parallel algorithm has been compared to sequential computation performed on the x86 architecture.

Keywords: Discrete Wavelet Transform, JPEG200, GPU, CELL.

1 Introduction

1.1 JPEG2000

The Discrete Wavelet Transform (DWT) is a signal processing technique for extracting information. It is based on sub-coding and can represent data by a set of coarse and detail values in different scales. DWT is frequently used in many practical applications including, audio analysis, image compression and video encoding. In image compression DWT decomposes data into the horizontal and vertical characteristics. It is one-dimensional transform in nature, but applying it in the horizontal and vertical directions forms two-dimensional transform.

M. Alexander et al. (Eds.): Euro-Par 2011 Workshops, Part I, LNCS 7155, pp. 481–490, 2012.

This result in four smaller images. DWT process can be repeated a number of times and it is called dyadic decomposition. The Cohen-Daubechies-Feauveau wavelet is one of the most commonly used set of discrete wavelet transforms in image compression. There are two versions of CDF wavelets: reversible integer-to-integer (CDF 53) and non-reversible real-to-real (CDF 97) wavelet transforms. The reversible transform uses only rational filter coefficients during decomposition and no data is lost due to rounding. It is called *lossless* decomposition. The non-reversible transform called *lossy decomposition* uses non-rational filter coefficients, so it allows for some data to be lost. Both of these transforms are implemented in JPEG2000 image compression standard [1], which has better performance compared to JPEG standard [2]. The detailed description of DWT can be found in [3].

DWT can be realized by iteration of filters with rescaling. This kind of implementation has high complexity, needs a lot of memory and computational power. The better way is to use the lifting-based wavelet transform proposed by Swedlens [4]. Lifting-based filtering is done by using four lifting steps, which update alternately odd or even sample values.

One of the most known application of DWT is JPEG2000 standard. JPEG2000 is a standard for picture encoding in digital movies. The movie with 4K (4096x2160) resolution demands very fast real time encoding solution to distribute it for instance via live broadcasts. Although there are some hardware implementations that offers real time encoding, they are costly as specialized hardware is required. Current consumer-level architectures with software implementations can provide low-cost alternative to hardware solutions. Therefore, our main motivation was to use new hybrid computing architectures: GPGPU and CELL B.E. to implement low-cost software base alternative solutions.

1.2 Cell B.E. Architecture

The Cell Architecture [9], [10] grew from a challenge posed by Sony and Toshiba to provide power-efficient and cost-effective high-performance processing for a wide range of applications, including the most demanding consumer appliance: game consoles. Cell-B.E. (CBEA) - is an innovative solution based on the analysis of a broad range of workloads in areas such as cryptography, graphics transform and lighting, physics, fast-Fourier transforms (FFT), matrix operations, and scientific workloads.

1.3 GPGPU Architecture

General-Purpose GPU is a highly parallel, multithreaded, many core processor with a very high computational power and memory bandwidth, e.g. offered by nVidia. In our work we used recent NVIDIA GPUs: Tesla S1070 Computing System consisting of four T10 computing processors and one gamer's card GTX 280. Each of Tesla T10 processors consists of 30 multiprocessors (MP). Multiprocessor is built from 8 Scalar Processors (SP) cores, two special function units,

a multithreaded instruction unit, one double precision unit and on-chip shared memory. GTX 280 processor has very similar specification, but only possess 1GB memory.

The rest of this paper is organized as follows. The related work is described in Section 2. Section 3 describes generally sequential and parallel DWT algorithm. The optimizations techniques on Cell B.E. and GPU architectures are presented in Section 4 and 5 respectively. Section 6 summarizes and compares results performance obtained on Cell and GPU. Conclusions are given in Section 7.

2 Related Work

In the context of the DWT algorithm several implementations on GPU has been proposed. In [5], it is presented a CUDA algorithm that performs the one-level 2D DWT algorithm in 45ms (without data transfer to and from GPU) on high resolution image with 4096x4096 pixels using NVIDIA Tesla C870. Another highly optimized algorithm is proposed in [6]. This implementation also adopts CUDA, which performs the three-level 2D Daubechies (9,7) wavelet transform in 2.13ms (without data transfer) on 1920x1080 pixels image using NVIDIA GeForce 8800 GTX.

In the case of Cell architecture the efficient implementation of the DWT algorithm is proposed in [7]. The one-level non-reversible wavelet transform executes in 54ms on 3800x2600 pixels image using the IBM QS20 Cell blade server with Cell/B.E. 3.2 GHz chip.

3 Discrete Wavelet Transform

3.1 Sequential Algorithm

First, in the horizontal transform a source image data rows using lifting procedure are decomposed into a set of low pass samples and a set of high pass samples. Than, samples are exposed to the de-interleaving procedure and the image data is transposed to represent rows as columns. The whole process is repeated to create a 2-dimensionally transformed image data.

Lifting Procedure. The lifting procedure consists, in fact of multiple lifting steps: splitting step, two predicting steps, two updating steps and the last scaling step. The splitting step simply splits image data row into two subsequences. One subsequence consists of odd elements and the second one consists even elements. In the predicting step the odd sample values are updated with a weighted sum of even samples and in updating step the even sample values are updated with a weighted sum of odd samples. In order to avoid errors at boundaries of the input row symmetric extension is used. Symmetric extension adds a mirror image of the signal to the outside of the boundaries, refer to [8]. In the last step all sample values are scaled.

De-interleaving. After the lifting procedure the next procedure is de-interleaving. Two subsequences with even and odd samples are called high-pass samples and low-pass samples respectively. They are mixed together in the input array after the lifting procedure. The high-pass and low-pass results have to be moved to the right half and left half of a output array respectively. The computed base position in the input array is linked with the corresponding output pixel in the output array.

3.2 Parallel Algorithm

In the first our approach the sequential algorithm was parallelized without any major changes. This is called the *base* version, it uses the many-loops approach and it is computed in two steps - first horizontally (on entire rows) and than vertically (on entire columns or transposed rows). Columns are processed after rows, so there is a need to synchronize computations. The *base* algorithm was a starting point for further optimizations. Then, we developed a parallel algorithm so-called *tiled* DWT.

The main difference is the problem decomposition. Whereas the *base* DWT works on an image as a whole, the *tiled* version splits images into rectangles of the same size and invokes DWT on them independently. The *tiled* DWT achieves much better performance than the *base* one. However, please note that the result of DWT processing the image in tiles is not identical to the result obtained when working on an image as whole, because *tiling* may introduce artifacts in the resulting image. In other words running the DWT on a whole image at once gives better resulting image quality, but at a very high cost of algorithm performance.

4 DWT on Cell B.E.

4.1 Basic Optimizations

Assumptions. For simplicity we focus on one-level forward transform, since the inverse one is symmetrical. We used a so-called lifting decomposition scheme, which is a very efficient way of computing discrete wavelet transforms. DWT is computed on one-channel grey image of size 4096 pixels (width) on 2048 pixels (height).

Base Version - Parallelization and Vectorization Using SPEs. Parallelization and vectorization of a sequential DWT algorithm resulted in *base* DWT. It uses the many-loops approach and is computed in two steps - first horizontally (on rows) and than vertically (on columns). Columns are processed after rows, to synchronize the computation, in the following way: each SPE thread is executed two times and the POSIX thread join on PPE is used as a natural barrier. The *base* algorithm is fully vectorized and all computations are done on SPEs. The PPE is used only for thread management. Horizontal DWT works on chunks of 4 rows and before computing DWT itself, it shuffles the float order. Vertical DWT works on chunks of 32 columns and arranges the data in correct order on-the-fly during memory *get/put*.

When the buffer size in Local Store is set to 64kb, this allows to use three of them (to enable double-buffering) and leaves 64kb for the code. One chunk of 4 rows of 4096 pixels (floats) fits ideally into the 64kb's buffer, so horizontal DWT is computed on entire rows at once. This is not the case for vertical DWT, where chunk of 128 columns limits the maximum height to 512 pixels. As the image height is set to 2048, the vertical DWT is computed in 4 steps, successively for every quarter of 128 column's chunk.

The *base* DWT works in place, which means the resulting image is stored in the same place in main memory as the input data. De-interleaving the columns in horizontal DWT is relatively easy, as it works on all rows and might be executed on SPEs before storing the results into the main memory. In case of vertical DWT, rows are deinterleaved on-the-fly while storing to the main memory. A complex scheme of loading and storing data was developed in order to enable double-buffering and not to erase the input data before time.

Tiled Version - Problem Decomposition and Memory Issues. The our analysis of the *base* DWT algorithm shown that it requires costly memory transfers and thread management. To address this issue, we improved the *tiled* DWT algorithm. The main idea is to compute both horizontal and vertical DWT at once on tiles of image, thus reducing the cost of synchronization and minimize the number of memory store/load operations. Therefore the main difference between *base* and *tiled* version is the problem of decomposition.

The optimal size of tile images has been estimated experimentally, assuming the tile should fill in entire 64kb buffer and that both height and width should be a power of 2. This resulted in 512 image tiles of the optimal size of 512 pixels on 32 pixels.

4.2 Advanced Optimizations

First optimization technique is reducing the number of separate loops and merge them into a single one. This included merging two predicts and two updates of DWT values, as well as bytes shuffling and reshuffling in case of horizontal DWT (for vertical DWT bytes are reordered on-the-fly during memory transfer). This significantly reduced the cost of handling loop counters and also allowed to apply next optimization, which was moving most computation into registers. In the final algorithm each pixel is read from an array in Local Store only once, all intermediate values computed during DWT are kept in the registers. Then the final results are written just once back to the array in Local Store. Each pixel was read from and written to Local Store only once, whereas all arithmetic calculations were performed in the registers. Such a approach resulted in a very noticeable increase of algorithm efficiency. Tiling an image and loop merging reduced significantly the execution time. Consequently, the initialization phase started to play a major role in the overall effectiveness, and we decided to apply two additional optimization techniques. The algorithm has been balanced for 6

SPEs for Playstation3 and 8 SPEs for QS21. In case of QS21, balancing threads optimization allowed us to run parallel instances of the application, as QS21 has two PPEs. We applied a simple double buffering. Thus, computation and data transfer steps were overlapped and the execution time was reduced. Further optimizations were pointless, as the actual computational part performed on SPEs dropped below 1ms, whereas the thread management part performed by PPE took approximately 10ms (on QS21).

5 DWT on GPGPU

We implemented two versions of parallel DWT algorithm on GPGU: the base DWT and the tiled DWT. DWT algorithm can be effectively paralleled, as data are separated. Naturally, the algorithm can be divided into a number of completely independent tasks, updated by a block of threads and executed by a separate multiprocessor.

5.1 Basic Optimizations

Base Version. The simplest approach to implement DWT on GPGU is using a sequential algorithm and try to parallelize it. First, the image data is simply saved in the global memory. Then, the image data is divided to data chunks which are loaded to the shared memory, to perform lifting and de-interleaving procedures and finally to store back results in the device memory. The whole process is performed two times, on every column and on every row. Each block of threads process one data chunk from the image. A single thread in a block reads one input pixel and generates one output pixel, so thread corresponds to one pixel form the image.

Every thread loads one pixel to the shared memory. During the lifting process, all the pixels at the edge of the data chunk depend on pixels which were not loaded to the shared memory. Around a data chunk within a thread block, there is a two-sided margin of pixels that is required in order to properly do the calculations. This margin of one data chunk overlaps with adjacent data chunks. In order to avoid idle threads, data from the margin should be loaded by threads within block. To avoid large errors at the boundaries the margins of the blocks on the edges of the image should be symmetrically extended.

Before the vertical(column) and the horizontal(row) processing the image should be transposed to access array elements that are adjacent in the global memory. The image data block was loaded to the shared memory array, transposed and written back to the global memory, to avoid uncoalesced access during lifting process. The image was partitioned into square tiles. We used 16x16 threads in block and every thread transposed one pixel.To do processing on columns and rows there were two separate kernel invocations. Between kernel invocations was a global barrier synchronization. It ensured that after every step

the output signal was written back to the global memory. After transposition of the image, we used four lifting loops to calculate the output signal. Firstly, the block of threads was divided into two parts. The left part with threads that referred to even memory cells (low pass samples) and right part with threads that referred to odd memory cells (high pass samples). Therefore only every second thread in a warp was participating in every lifting step. Odd threads were responsible for high resolution pixels and even threads were responsible for low resolution pixels. The margin was updated by few threads that were taken additionally form the left or the right part of threads block. After every lifting step threads were synchronized in order to write back results to the shared memory. However this approach has major disadvantage. During every predict or update step (see Lifting procedure 3.1) the other part of threads in a warp was idle till to synchronization. In order to improve the parallelism, the algorithm was changed and it was used one-loop approach as follows: every thread loaded all necessary data to registers, as it was needed to correctly compute one output pixel. The necessary data were composed of 8 adjacent pixels. Additional improvement was maximizing a number of arithmetic operations by using each thread to load and calculate multiple pixels instead of one.

To sum up every thread read and synchronized two pixels from the global memory to the shared memory. Then, all the threads read adjacent pixels from the shared memory to registers and perform the lifting procedure. When registers were used instead of shared memory, each thread from the block was able to calculate two adjacent output values: high pass and lows pass. It gave us more speedup, as memory transactions were better overlapped by arithmetic operations. After the lifting procedure pixels were scaled and written to the global memory. We restricted our experiment to the shared memory size and a reasonable amount of 2048 pixels. The optimal time we observed was for threads which compute 8 pixels in a block of size 256. The fastest computations run in 11ms.

Tiled Version. A single image can be composed of a single tile or multiple independent tiles. In tiled version of the algorithm the image is divided to multiple tiles and on every tile DWT algorithm is applied. Similar to the base version, the image data chunk was loaded to the shared memory, however the processing on columns and rows was done in one kernel invocation including data transposition. As a result a number of kernel invocations and the global memory calls were reduced. It minimized the amount of global memory transactions.

Data Partition in the Shared Memory. The image was divided into multiple same size data blocks. Every data block was loaded to the shared memory. All the edges of rows and columns in a data block were symmetrically extended. In this case, we did not load pixels from adjacent data blocks. As a result small artifacts on edges were introduced. Each thread within data block loaded one or more pixels, depending on the kernel used in experiments.

Lifting Procedure and Transposition. Algorithm applying the lifting procedure was similar to the algorithm we presented for the base version. If we divided the image into non-overlapping data blocks we were able to apply another optimization. The margin for the row and the column had the same length. When the first transposition was done we had to add new margins between rows in the shared memory in order to do symmetric extension. All performance results depend on the size of the data blocks. The optimal size for the image tile was 64 pixels on 32 pixels using 256 threads in one block, and the execution time was 2,4ms.

5.2 Advanced Optimization

In this section we show more advanced techniques and try to reach maximum efficiency of DWT algorithm on GPU. First optimization is block balancing and maximizing arithmetic operations. Scaling size of the threads block brings more speedup. A number of threads per block should be chosen as multiple of the warp size to avoid wasting computing resources with under-populated warps. Our tests shown that the number of 256 threads in one block is optimal. Second optimization is data transposing. One should note that loading data from the global memory in a column-major fashion is inefficient. Therefore data should be transposed before loading it form the global memory. The next optimization approach was to load all necessary data to registers. We used registers for computations and wrote data back to the shared memory only once. Last optimization was to reduce the data transfer time. The large amount of time was taken by the data image transfer, between CPU and GPU memory. In order to reduce this bottleneck we applied double buffering and split images through all available Tesla modules. Thus, we overlapped a kernel computation with memory copy from different streams.

6 Performance Analysis

6.1 Cell B.E. - Scalability and Execution Time Analysis

To compare various optimization techniques, first we developed a single processor version of the DWT algorithm without any optimization. It was run on one PPE (on Playstation 3 and QS21, with and without Altivec vectorization) and on one 2.16GHz x86 compatible processor.

Each optimization technique helped us to reduce significantly the execution time. The most efficient version of DWT we obtained on QS21 11.5ms. The version of the DWT algorithm in which SPE threads immediately quit with *return* statement, runs for 10.8ms. This means that both DWT computation and memory transfer took less than one millisecond and constitute approx. 6% of the entire computational time.

In order to address the scalability issues we analyzed the algorithm without double buffering and balancing thread procedure (see Figure 1a and Figure 1b

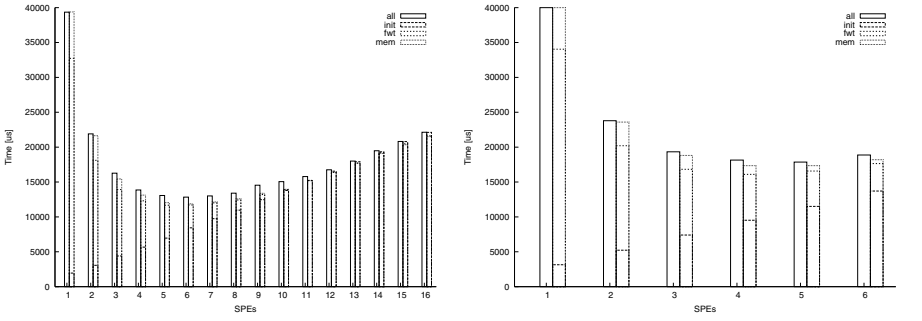

Fig. 1. a) QS21 scalability. b) PS3 scalability.

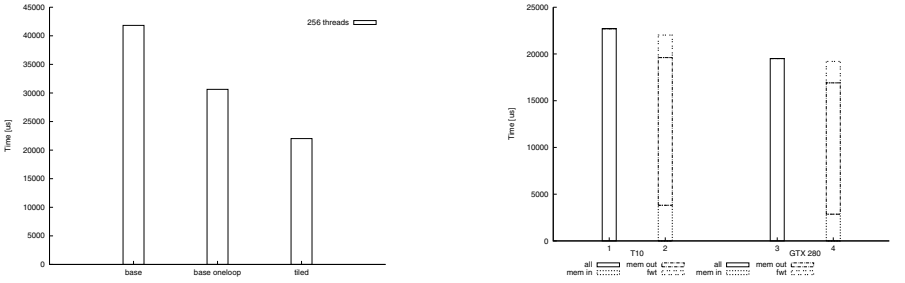

Fig. 2. a) Tesla T10 - best times comparison. b) Tesla T10 and GTX 280.

for details). For both Playstation 3 and QS21 with a rising number of SPEs we discovered that the thread management cost (operations performed on PPE) started to play a major role in the whole execution process.

6.2 GPU - Scalability and Execution Time Analysis

Our program is built on the top of the CUDA 2.1 (Compute Unified Device Architecture). We performed a warm-up computation before of the timed computation to remove the CUDA start up overhead from performance measurements. Figure 2b shows that the best DWT algorithm achieved 19,5ms using gamer's card GTX 280 and 22,7ms using one GPU module on Tesla S1070. The GTX 280 card had higher memory bandwidth and lower computation time than one Tesla S1070 module. According to our tests it has about 16% faster data transfer. The difference in time results between this two device can result from slightly different architectures.

Table 1. Computation time: x86, GPU and CELL

Device	Init.	Mem. copy	Comp.	Speedup
x86	0.0ms	0.0ms	1500ms	1.00x
GTX 280	0.15 ms	16.9ms	2.29ms	76.92x
QS21	10.8ms	0.7ms (double buff.)		130.43x

7 Conclusions

We wish to summarize stressing out the issue which we consider to be the most important difference between Cell B.E. and GPU. Cell B.E. relies on heavy persistent threads, whereas GPU paradigm is to use very light-wighted threads. This has a huge impact on programming style for those architectures, resulting in the development of totally different approaches. Although the concept of parallelization the algorithm for both GPU and Cell B.E. is similar the optimizations details are completely different.

In our opinion the biggest drawback of GPU computing is relatively high cost of memory transfers, which is not a problem in case of Cell B.E. thanks to Element Interconnect Bus that allows the memory transfer and computation to overlap. To some extent this could be addressed by using many GPUs, currently Tesla server consists of four graphical cards.

References

1. ISO/IEC 15444-1: Information technology JPEG 2000 image coding system Part 1: Core coding system (November 2000)
2. ISO/IEC 10918-1: Information technology Digital compression and coding of continuous-tone still images: Requirements and guidelines (1994)
3. Taubman, D., Marcellin, M.: JPEG2000 Image Compression Fundamentals, Standards and Pratice (2002)
4. Sweldens, W.: The lifting scheme: a new philosophy in biorthogonal wavelet constructions. In: Proceedings of the SPIE, Wavelet Applications in Signal and Image Processing III, vol. 2569, pp. 68–79 (September 1995)
5. Franco, J., Bernabé, G., Fernández, J., Acacio, M.: A Parallel Implementation of the 2D Wavelet Transform Using CUDA. In: Proceedings of the 2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing (2009)
6. van der Laan, W., Jalba, A., Roerdink, J.: Accelerating Wavelet Lifting on Graphics Hardware Using CUDA. IEEE Trans. Parallel Distrib. Syst. (January 2011)
7. Bader, D., Agarwal, V., Kang, S.: Computing discrete transforms on the Cell Broadband Engine. Parallel Comput. (March 2009)
8. Aboufadel, E., Elzinga, J., Feenstra, K.: JPEG 2000: The Next Compression Standard using wavelet technology (December 2001)
9. IBM Corporation, Cell Broadband Engine Technology, `http://researchweb.watson.ibm.com/cell/home.html`
10. IBM Corporation, Cell Broadband Engine Technology, `https://www-01.ibm.com/chips/techlib/techlib.nsf/products/Cell_Broadband_Engine`