

GPU-Based DWT Acceleration for JPEG2000

Jiří Matela

Faculty of Informatics, Masaryk University, Botanická 68a, 602 00 Brno,
Czech Republic matela@ics.muni.cz

Abstract. In our paper, we focus on accelerating DWT (Discrete Wavelet Transform) part of the JPEG2000 using general-purpose processing on graphical processing unit (GPU). We utilize Compute Unified Device Architecture (CUDA) platform which has its specific properties and constraints. In particular, a proper memory treatment can greatly affect overall performance. In this paper, we briefly describe corresponding CUDA technical background followed by elaborated description of the algorithm—especially its memory layout part. Resulting implementation of DWT performs very well compared to other implementations available and is able to process an HD sized picture within 1 *ms* or faster.

1 Introduction

JPEG2000[1, 2] is the latest image compression standard from the Joint Photographic Experts Group (JPEG). The aim of JPEG2000 is not only compression performance superior to the current standards but also to provide advanced features demanded by today's emerging applications. The features include progressive image transmission by improving pixel accuracy and resolution, error resilience, or optional mathematically lossless processing. To meet these needs, JPEG2000 adopts a number of contemporary digital signal processing findings. Resulting computational requirements of JPEG2000 are one of drawbacks hindering use of JPEG2000 in common application.

When encoding, an image data is prepared for compression in pre-processing phase first, then entropy-encoded using Embedded Block Coding with Optimal Truncation[3] (EBCOT) algorithm. The decoding procedure is the reverse of encoding. In pre-processing phase, optional color transform is applied and data is then transformed using Discrete Wavelet Transform (DWT) and quantized.

DWT is important and computationally demanding part of JPEG2000 algorithm. Especially real-time processing of images with HD (1920×1080) resolution or higher could be performance sensitive. In section 5 we propose a fast DWT implementation utilizing the CUDA architecture. Exploitation of massively parallel capabilities of current GPUs enabled us to achieve processing rate of 2,5 GPix/s. Section 3 provides an overview of DWT used by JPEG2000 placing emphasis on *lifting scheme*. Section 4 briefly describes CUDA computation model and outlines important constraints of CUDA.

2 Related work

Massively parallel architecture of current GPUs is a platform suitable for acceleration of mathematical computations in the field of digital image analysis. In [4], Wong et al. described a GPU accelerated DWT implementation. They compared their OpenGL and Cg-based implementation with a CPU DWT implementation used by JasPer¹ [5]. The GPU implementation executed on nVidia 7800 GTX showed speedups of up to 7.3 against the CPU implementation executed on AMD Athlon 64x2 dual core processor 3800+ 2.01 GHz. Similarly in [6], Tenllado et al. evaluated implementations of DWT based on OpenGL and Cg. On an nVidia 7800 GTX, they presented speedups ranging from 1.2 to 3.4 over an Intel P4 Prescott 3.4 GHz processor.

Franco et al. [7] proposed implementation based on CUDA architecture. Using nVidia Tesla C870 they achieved speedups ranging from 20 to 40.6 in the execution time over a C version on an Intel Core 2 Quad Q6700 (2.66 GHz).

3 Discrete Wavelet Transform

Discrete Wavelet Transform (DWT) [2] is a broadly used digital signal processing technique with application in diverse areas. DWT allows us to study a digital signal in different resolutions as sets of coarse and fine values. Wavelet transforms are used in domains of digital speech recognition, multi-resolution video processing or data compression. In the context of JPEG2000 standard, DWT is the key prerequisite of the compression process. Most of advanced features of JPEG2000 rely on DWT as well as the superior low-bitrate performance does.

JPEG2000 standard specifies [2] use of *LeGall (CDF) 5/3* DWT filter-banks [8] for lossless compression process and *Daubechies-Feauveau (CDF) 9/7* DWT filter-banks [8] for lossy processing. Wavelet transforms can be implemented by convolution or by lifting scheme.

3.1 Lifting scheme

The advantage of lifting scheme [9, 10] over convolution is in reduced memory and computational complexity. Lifting scheme allows for in-place data manipulation and reduces memory dependencies.

Lifting scheme analysis [11] proceeds as follows. An input signal is split into even and odd subsequences denoted as $\{s_i^0\}$ and $\{d_i^0\}$ respectively. These values are further modified using alternating *prediction* (denoted as p) and *update* (denoted as u) steps. In the prediction step, the algorithm takes an odd sample in a turn and subtracts a linear combination of its (even) neighbours from it; a prediction error $\{d_i^1\}$ is formed:

$$d_i^1 = d_i^0 - p(s_i^0 + s_{i+1}^0) \quad (1)$$

¹ JasPer is an open source reference implementation of Part 1 of JPEG2000 standard

In the update step, a linear combination of already modified adjacent odd samples is added to each even sample and updated even sequence $\{s_i^1\}$ is formed:

$$s_i^1 = s_i^0 + u(d_{i-1}^1 + d_i^1) \quad (2)$$

The output of the last update stage, $\{s_i^j\}$, is actually a low-pass output of DWT filter and similarly output of the last prediction stage, $\{d_i^j\}$, is a high-pass output of the filter. So the result of the wavelet transformation is a signal divided into low-pass and high-pass subbands.

2D signals (e.g., images) are usually transformed in both dimensions. 1D DWT transform is first applied to all rows then to all columns resulting in four subbands LL, HL, LH, and HH. The LL subband is an approximation of the original signal and can be further transformed recursively.

4 GPU computing using CUDA

CUDA [12] is software and hardware platform designed for general purpose computing on GPUs. GPUs have a parallel architecture capable of running thousands of threads in parallel. In CUDA computing model, such threads are grouped into so called *thread blocks*. Threads within a block can cooperate among themselves by sharing data through a *shared memory*. As opposite to a large *global memory*, shared memory is relatively small and very fast². The advantage of global memory is that it is accessible to all threads, whereas shared memory is visible only to threads of the block. Common work flow is to copy data from RAM to global memory of GPU. Once data is ready in global memory, a GPU program can be executed. Each thread block initially fetches a small portion of data from the global memory into the shared memory. Data is then processed by threads in the block and the result is moved back to the global memory.

The global memory access pattern is perhaps the most important performance consideration in programming for the CUDA architecture. In a nutshell, when 16 adjacent threads access adjacent locations in global memory then memory loads and stores are coalesced in one transaction. The details of memory coalescing are described in [12].

5 GPU-accelerated implementation of DWT

The key part of our GPU-accelerated DWT is the design how to split the work between thread blocks in order to provide maximum utilization of the GPU. Since we need to compute the transform in both dimensions, it is natural to choose a 2D partitioning of source image data. Also size and shape of thread blocks needs to be determined. Because the lifting scheme algorithm alternately works with even and odd samples, an efficient approach is to have one even and

² Even global memory is a few times faster than the computer RAM. E.g. nVidia GeForce GTX 285 has global memory bandwidth of 159 GB/s

one odd data sample per each thread in a block—i.e., to have twice as much data samples as threads in each block. Resulting partition of image data is shown in Fig. 1. Each thread block has its dimensions $B_x \times B_y$ where $B_x = D_x$, $B_y = \frac{D_y}{2}$, and B_x, B_y and D_x, D_y denote number of threads and samples in horizontal and vertical direction respectively. Thus that thread blocks are of rectangular shape while data blocks are of squared shape.

Each thread has its exact position within the block, which is determined by indices T_x, T_y where $0 \leq T_x \leq B_x - 1$ and $0 \leq T_y \leq B_y - 1$. In Fig. 1 we can see that threads overlap only the upper half of the block, i.e., $MAX(T_y) = \frac{D_y}{2} - 1$, which means that threads are directly mapped only to samples in the upper half of block data and we will have to change this mapping to be able to process both halves.

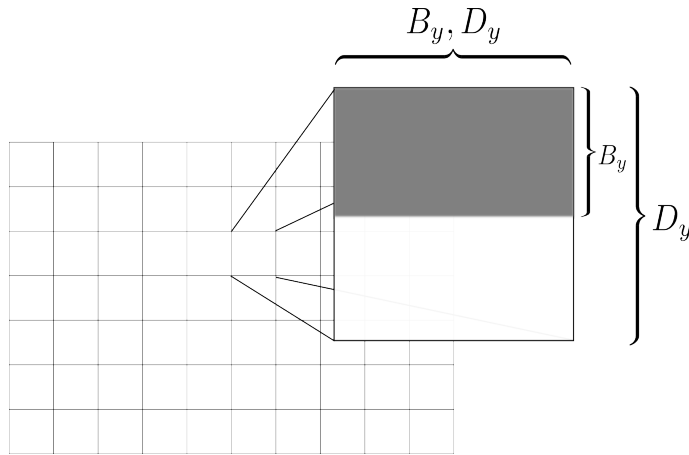


Fig. 1. Source image partitioning.

The first step of computation is to fetch image data from global memory into fast shared memory. It is crucial here to comply with coalesced global memory access. Considering the proposed data partitioning, each thread loads corresponding data sample into the upper half first, and then into the lower half of data block. The horizontal block size should be multiple of 16, so that coalesced access is not broken by thread block misalignment.

DWT coefficients are then computed according to lifting scheme relations 1 and 2. To calculate first dimension of the transform, DWT filters are applied to every row separately. Afterwards, each row contains a sequence of interleaved coefficients of low-pass and high-pass subbands—L, H, L, H, ..., L, H, L, H. Each particular prediction and actualization step is calculated respectively as follows.

$$s[T_x][2T_y + 1] = s[T_x][2T_y + 1] + p \cdot (s[T_x][2T_y] + s[T_x][2T_y + 2]) \quad (3)$$

$$s[T_x][2T_y] = s[T_x][2T_y] + u \cdot (s[T_x][2T_y - 1] + s[2T_y + 1]) \quad (4)$$

Where T_x and T_y determine the thread position in horizontal and vertical direction respectively and $s[row,dx][column,dx]$ is the shared memory 2D array. Note that we propose transposed thread mapping for efficient data processing as follows. Threads are directly mapped into the upper half of block only, so that we have to change the thread mapping to be able to process whole block. In equations 3 and 4, we have swapped³ thread indices T_x, T_y so that the threads cover the left half of the data block instead of the upper half which was covered originally. Equation 3 then *predicts* all odd samples and equation 4 *updates* all even samples in the block.

To calculate the second dimension of the transform, we just apply same filters to the columns.

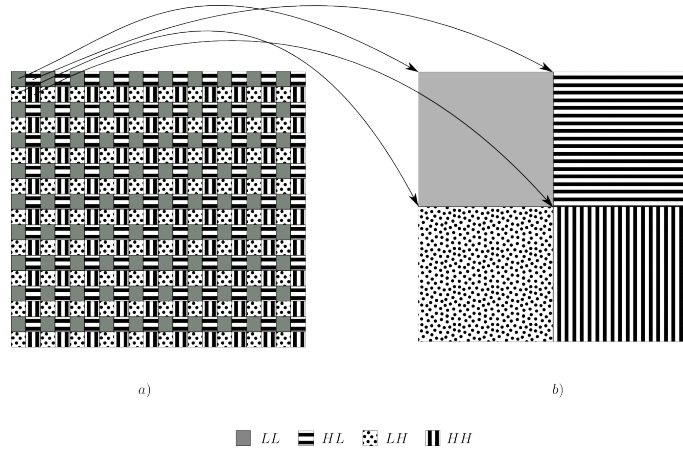


Fig. 2. Moving result of the 2D DWT from shared to global memory.

The result of the application of lifting filters to rows and columns is composed of coefficients of four DWT subbands. Coefficients of LL and HL subbands are alternately located on on even rows and LH and HH coefficients on odd rows of the shared memory s . Shared memory s is depicted in Fig. 2a.

The final step of the CUDA-based transform is to move result from shared memory back to global memory as shown in Fig. 2. Particular subbands, however, needs to be stored separately in global memory. Because there are twice as much data samples as threads in the block, we store even lines first. Even lines contain all LL and HL samples and because those are interleaved, we use first half of threads to store all LL samples and second half to store HL samples. The access to the global memory is hereby coalesced.

³ Note that in C and also in CUDA the notation $s[y][x]$ points to the x -th element on the y -th row.

Note that proposed implementation of DWT is optimized for maximum performance and its limitation is that it does not take into account sample values exchange between blocks borders. The proposed algorithm does not introduce any visual artefacts provided both forward and reverse transformations work with the same data block dimensions.

6 Performance evaluation

In this section we present evaluation results of the proposed CUDA DWT implementation. All measurements have been performed on a computer equipped with Core i7 3.2GHz, 3×2 GB RAM and nVidia GeForce GTX 295. Execution times presented in this section does not include host-to-device transfer times—i.e., times needed to transfer data from RAM to GPU memory.

We also present evaluation of two CPU DWT implementations to compare with the GPU implementation. One of them is optimized DWT implementation used by JasPer [5], the other is very "naïve C implementation we were using to validate GPU results.

6.1 Host to GPU memory memory transfers

Data transfers from PC to GPU are integral part of GPU computation. Since all following performance measurements presented in this chapter do not include the data transfer times, we discuss it here. As opposite to standard C `malloc()` call, CUDA runtime API provides `cudaMallocHost()` call which can be used to accelerate data transfers [13].

Results in Table 1 clearly shows throughput improvement in favour of `cudaMallocHost()` call. Importance of such improvement is more apparent when data transfer times are compared to computation times in Table 2. Results for PCIe 1.0 x16 were measured on PC equipped with dual AMD Opteron Dual-Core 2.6 GHz and 4×1 GB RAM.

Table 1. Host to GPU memory transfers results. Time values are for grayscale 1920×1080 image.

Call	PCIe 1.0 x16	PCIe 2.0 x16
<code>malloc()</code>	1.3 ms \sim 1.5 GB/s	0.40 ms \sim 4.7 GB/s
<code>cudaMallocHost()</code>	0.7 ms \sim 2.8 GB/s	0.33 ms \sim 5.5 GB/s

6.2 Results

Execution times of all three implementations are shown in Table 2. Performance of each implementation was measured on three grayscale pictures of sizes

512×512, 1920×1080, and 2048×2048. All times are in milliseconds and reflect 2D DWT computation only—i.e. data loading or any other preprocessing is not covered in these measurements.

Both, simple C and implementation from JasPer codec, are single threaded, so that they were able to utilize only one CPU core. As expected, simple C is the slowest in the set and needs 120 ms to process HD-sized image. Optimized JasPer DWT implementation computes 2D wavelet transformation about two times faster and is able to process the same image within 55 ms. However, both implementations are not fast enough for real-time applications like real-time HD video processing where the processing needs to take up less than approx. 30 ms for 60 frames per second.

The proposed implementation can process the HD image within 0,81 ms and is about sixty-eight times faster than DWT code from JasPer—particular speedups are shown in Table 3. Table 3 also shows throughputs calculated according to results from Table 2. For reference full HD video at 24 fps is around 50 MPix/s and 4K (4096×2160) video at 24 fps is around 213 MPix/s.

Table 2. Execution times for two CPU and the proposed CUDA DWT implementation.

DWT Implementation	512×512	1920×1080	2048×2048
simple C	12 ms	120 ms	239 ms
JasPer	6 ms	55 ms	111 ms
CUDA DWT	0.12 ms	0.81 ms	1.62 ms

Table 3. Performance and speedups for two CPU and the proposed CUDA DWT implementation.

DWT Implementation	Throughput	Speedup
simple C	17 MPix/s	1.0×
JasPer	37 Mpix/s	2.1×
CUDA DWT	2560 Mpix/s	148×

7 Conclusion

Real-time JPEG2000 compression of high-definition image data is computationally very demanding. New hardware and software architecture CUDA allows to exploit massively parallel computational capabilities of modern GPUs for general purpose computing.

In our paper, we propose fast GPU-based implementation of 2D DWT transformation, which is very important component of JPEG2000 compression algorithm. The measured values indicate that CUDA DWT implementation can process HD and post-HD images in real-time, providing approx. $68\times$ speedup compared to reference optimized implementation. Moreover, considering obtained results, we believe there is a room to accelerate the other components of the compression algorithm.

8 Acknowledgments

This project has been supported by a research intent “Optical Network of National Research and Its New Applications” (MŠM 6383917201) and “Parallel and Distributed Systems” (MŠM 0021622419).

References

1. International Organization for Standardization: Information technology — JPEG 2000 image coding system — part 1: Core coding system. ISO/IEC 15444-1:2004 (2004)
2. Taubman, D.S., Marcellin, M.W.: JPEG2000: image compression fundamentals, standards, and practice. Springer (2002)
3. Taubman, D.: High performance scalable image compression with ebcot. Image Processing, IEEE Transactions on **9**(7) (2000) 1158–1170
4. Wong, T.T., Leung, C.S., Heng, P.A., Wang, J.: Discrete wavelet transform on consumer-level graphics hardware. Multimedia, IEEE Transactions on **9**(3) (2007) 668–673
5. Adams, M., Kossentini, F.: Jasper: A software-based JPEG-2000 codec implementation. In: Image Processing, 2000. Proceedings. 2000 International Conference on. Volume 2. (2000) 53–56 vol.2
6. Tenllado, C., Setoain, J., Prieto, M., Pinuel, L., Tirado, F.: Parallel implementation of the 2d discrete wavelet transform on graphics processing units: Filter bank versus lifting. Parallel and Distributed Systems, IEEE Transactions on **19**(3) (2008) 299–310
7. Franco, J., Bernab, G., Fernández, J., Acacio, M.E.: A parallel implementation of the 2d wavelet transform using cuda. Parallel, Distributed, and Network-Based Processing, Euromicro Conference on **0** (2009) 111–118
8. Cohen, A., Daubechies, I., Feauveau, J.C.: Biorthogonal bases of compactly supported wavelets. Communications on Pure and Applied Mathematics **45** (1992) 485–560
9. Daubechies, I., Sweldens, W.: Factoring wavelet transforms into lifting steps. J. Fourier Anal. Appl **4** (1998) 247–269
10. Sweldens, W.: The lifting scheme: A construction of second generation wavelets. SIAM Journal on Mathematical Analysis **29**(2) (1998) 511–546
11. Rabbani, M., Joshi, R.: An overview of the jpeg 2000 still image compression standard. Signal Processing: Image Communication **17**(1) (2002) 3 – 48
12. NVIDIA: NVIDIA CUDA Programming Guide 2.0. (2008)
13. NVIDIA: NVIDIA CUDA Best Practices Guide 2.3. (2009)