CUDA Based Implementation of 2-D Discrete Haar Wavelet Transformation

Hovhannes Bantikyan

State Engineering University of Armenia (Polytechnic), 105 Teryan Str., Yerevan, Armenia

bantikyan@gmail.com

Abstract. The discrete wavelet transform has a huge number of applications in science, engineering, mathematics and computer science. Most notably, it is used for signal coding, to represent a discrete signal in a more redundant form, often as a preconditioning for data compression. Practical applications can also be found in signal processing of accelerations for gait analysis, in digital communications and many others. In this paper presented implementation of 2-D DWT in parallel manner on Graphics Processing Unit, using CUDA technology. Calculating DCT in parallel, using multiple threads, gives us huge improvement in calculation speed.

Keywords

Discrete Haar Wavelet Transform, Parallel computing, GPGPU, CUDA programming.

1 Introduction

The wavelet transform, originally developed as a tool for the analysis of seismic data, has been applied in areas as diverse as signal processing, video and image coding, compression, data mining and seismic analysis. The theory of wavelets bears a large similarity to Fourier analysis, where a signal is approximated by superposition of sinusoidal functions. A problem, however, is that the sinusoids have an infinite support, which makes Fourier analysis less suitable to approximate sharp transitions in the function or signal. Wavelet analysis overcomes this problem by using small waves, called wavelets, which have a compact support. One starts with a wavelet prototype function, called a basic wavelet or mother wavelet. Then a wavelet basis is constructed by translated and dilated (i.e., rescaled) versions of the basic wavelet. The fundamental idea is to decompose a signal into components with respect to this wavelet basis, and to reconstruct the original signal as a superposition of wavelet basis functions; therefore we speak a multiresolution analysis. If the shape of the wavelets resembles that of the data, the wavelet analysis results in a sparse representation of the signal, making wavelets an interesting tool for data compression. This also allows a client-server model of data exchange, where data is first decomposed into different levels of resolution on the server, then progressively transmitted to the client, where the data can be incrementally restored as it arrives ("progressive refinement").

This entire work is aimed to develop a strategy to compute DCT more efficiently and to reduce the time it takes for calculation. In this case the Graphics Processing Unit (GPU) based algorithm can be the cost effective solution. GPU can process large volume data in parallel when working in single instruction multiple data (SIMD) mode. In November 2006, the Compute Unified Device Architecture (CUDA) which is specialized for compute intensive highly parallel computation is unveiled by NVIDIA.

2 Wavelet Transformation

The main idea of (first generation) wavelet decomposition for finite 1-D signals is to start from a signal $c^0 = (c_0^0, c_1^0, ..., c_{N-1}^0)$, with N samples (we assume that N is a power of 2). Then we apply convolution filtering of c^0 by a low pass analysis filter H and downsample the result by a factor of 2 to get an "approximation" signal (or

"band") c^1 of length N/2, i.e., half the initial length. Similarly, we apply convolution filtering of c^0 by a high pass analysis filter G, followed by downsampling, to get a detail signal (or "band") d^1. Then we continue with c^1 and repeat the same steps, to get further approximation and detail signals c^2 and d^2 of length N/4. This process is continued a number of times, say J. Here J is called the number of levels or stages of the decomposition. The explicit decomposition equations for the individual signal coefficients are:

$$c_k^{j+1} = \sum_n h_{n-2k} c_n^j \tag{1}$$

$$d_k^{j+1} = \sum_n g_{n-2k} c_n^j \tag{2}$$

where h_n and g_n are the coefficients of the filters H and G. Note that only the approximation bands are successively filtered, the detail bands are left "as is".

This process is presented graphically in Figure 1, where the symbol $\downarrow_2$ (enclosed by a circle) indicates downsampling by a factor of 2. This means that after the decomposition the initial data vector c^0 is represented by one approximation band c^J and J detail bands $d^1, d^2, ..., d^J$. The total length of these approximation and detail bands is equal to the length of the input signal c^0.

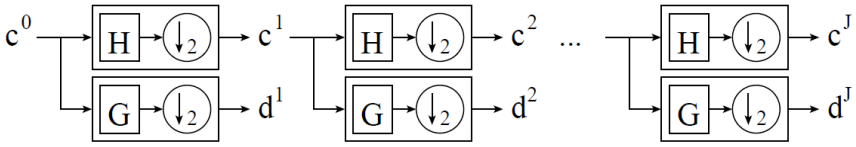

Figure 1. Structure of the forward wavelet transform with J stages: recursively split a signal c^0 into approximation bands c^j and detail bands d^j.

Signal reconstruction is performed by the inverse wavelet transform: first upsample the approximation and detail bands at the coarsest level J, then apply synthesis filters $\widetilde{H}$ and $\widetilde{G}$ to these, and add the resulting bands. (In the case of orthonormal filters, such as the Haar basis, the synthesis filters are essentially equal to the analysis filters.) Again this is done recursively. This process is presented graphically in Figure 2, where the symbol $\uparrow_2$ indicates upsampling by a factor of 2.

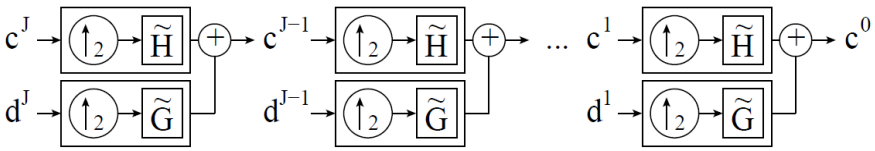

Figure 2. Structure of the inverse wavelet transform with J stages: recursively upsample, filter and add approximation signals c^j and detail signals d^j.

3 NVIDIA CUDA Basics

The fact that the performance of graphic processing units (GPUs) is much bigger than the central processing units (CPUs) of nowadays is hardly surprising. GPUs were formerly focused on such limited field of computing graphic scenes. Within the course of time, GPUs became very powerful and the area of use dramatically grew. So, we can come together on the term General Purpose GPU (GPGPU) denoting modern graphic accelerators. The driving force of rapid raising of the performance are computer games and the entertainment industry that evolves economic pressure on the developers of GPUs to perform a vast number of floating-point calculations within the unit of time. The research in the field of GPGPU started in late 70's. In the last few years, we can observe the renaissance in this area caused by rapid development of graphic accelerators. Two main players in the field of GPGPUs are AMD with their ATI Stream Technology and NVIDIA which introduced Compute Unified Device Architecture (CUDA) - the parallel computing engine accessing GPGPUs resources to software

developers. Through the frameworks extending commonly used programming and scripting languages such as C, Java, Python or MATLAB, CUDA enables easy way to make applications using NVIDIA GPUs.

3.1 Hardware Architecture

Present multi-core CPUs usually consist of 2-8 cores. These cores usually work asynchronously and independently. Thus, each core can execute different instructions over different data at the same time. According to the Flynn's taxonomy, we are talking about Multiple Instruction stream, Multiple Data stream (MIMD) class of computer architectures.

On the other hand, GPUs are designed for parallel computing with an emphasis on arithmetic operations, which originate from their main purpose - to compute graphic scene which is finally displayed. Current graphic accelerators consist of several multiprocessors (up to 30). Each multiprocessor contains several (e.g., 8, 12 or 16) Arithmetic Logic Units (ALUs). Up to 480 processors is in total on the current high-end GPUs. Figure 3 shows the general overview of the CPU and GPU.

Figure 3. Comparison CPU and GPU architectures.

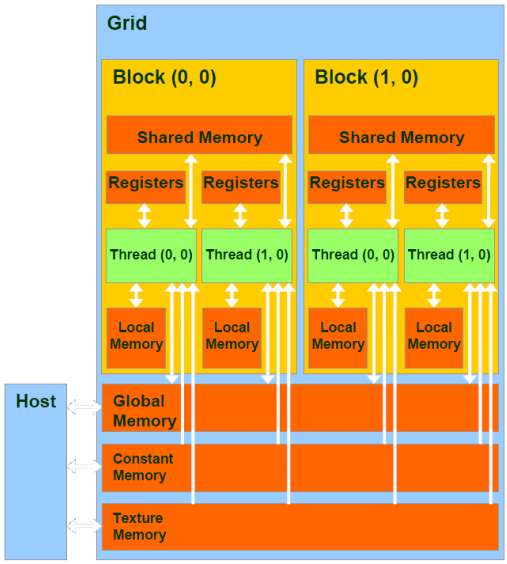

Figure 4. CUDA memory model.

Also the memory hierarchy is specific in the case of graphic accelerators. Each multiprocessor has registers that are used by ALUs. Processors within a multiprocessor can access shared memory of typical size 16KB, or a main memory of the accelerator, which is not cached on the majority of present accelerators. Global memory in terminology of CUDA, is accessible from all the processors on the accelerator. In addition, there are two separate memory areas - constant memory and texture memory, also shared across the multiprocessor and both cached and read-only. When accessing some element from the texture memory, a couple of surrounding elements

are also loaded. This feature is called spatial locality. One of the most limiting factors is a very small capacity of shared memory and registers. If application uses more variables per thread than available registers, they are stored in a local memory which is, in fact, the dedicated part of global memory. Accessing these variables is as time-consuming as accessing any other variable stored in the global memory. For better understanding, we can see the memory hierarchy in Figure 4.

3.2 Programming Model

A CUDA-capable GPU is referred to as a device and the CPU as a host. Thread is the finest grain unit of parallelism in CUDA.

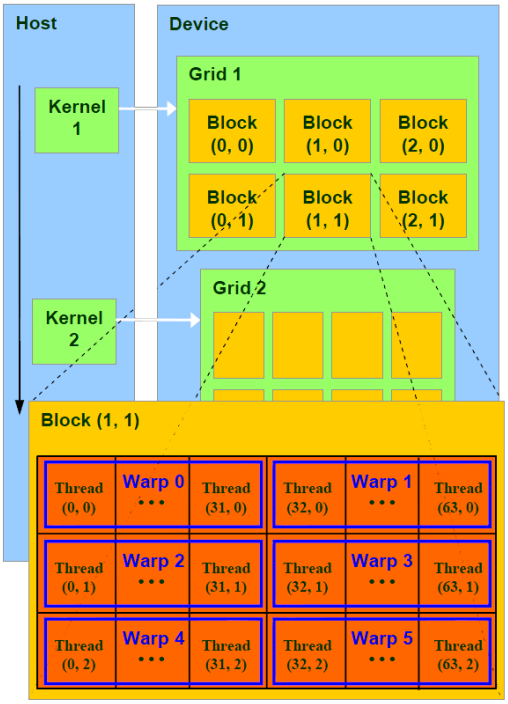

Figure 5. NVIDIA CUDA programming model.

Thousands of threads are able to run concurrently on the device. Threads are grouped into the warps. Size of a warp is usually 32 threads. Warp is the smallest unit that can be processed by multiprocessors. Warps scheduled across processors of one multiprocessor are coupled into a thread blocks. Block is a unit of the resource assignment. Typical size of a thread block is 64-512 threads and depends on the particular application what is the optimal size of a thread block to ensure the best utilization of the device. Thread blocks form a grid. Grid can be viewed as a 1-dimensional, 2-dimensional or 3-dimensional array. Fig. 5 is depicting the described hierarchy.

3.3 Features and Limitations of CUDA

It is easy to learn the CUDA API, but hard to programme efficient applications which utilize the GPU's performance. CUDA API is a set of extensions based on the standard C language. Counterweight to many features of this massively parallel architecture is that there are limitations mostly caused by HW architecture. CUDA belongs to the class of Single Instruction, Multiple Thread (SIMT) according the Flynn's taxonomy. SIMT originates in Single Instruction Stream, Multiple Data Stream (SIMD) class known for example from the supercomputers based on vector processors (e.g., Cray-1). SIMT also implies the divergence in the program that usually leads to the serialization of the run. Recursive functions are not supported either. As introduced before, graphic accelerators were developed with the focus on computing vast amounts of arithmetic operations. Many of them are implemented directly in the hardware with a cost of units of warp-cycles. Besides arithmetic functions there is a set of bitwise operations also implemented "in hardware".

Of course, a set of constructs used in parallel programming is present in CUDA. For example several methods of barrier synchronization primitives, native broadcast of a single variable, scatter and gather functions or atomic operations which prevents from race conditions. The use of shared memory has also significant impact on the overall performance but the limiting factor is its size of 16 KB. Talking about memory, CUDA brought more efficient data transfer operation between the host and the device. Unlike OpenCL, CUDA is closed source belonging to NVIDIA corp. which can be considered as a limitation as well.

4 Implementation and Results

Here implemented 2D discrete Haar wavelet transform for color images. First we perform 1D FWT for all rows, and next, for all columns. For color Images we deal with RGB components of color, and perform Haar Transform for each component separately. Any component (R G B) has values from 0 to 255 to before transformation we scale this values. For displaying image after transformation we scale back transformed values. Let's look at 1D Haar transform on a little example. Suppose you are given N values

$$x = (x_0, x_1, \dots x_{N-1})$$

where N is even. We take pair-wise average of numbers $s_k = (x_{2k} + x_{2k+1})/2$ for $k = 0, \dots, N/2 - 1$. For example

$$x = (6, 12, 15, 15, 14, 12, 120, 116) \rightarrow s = (9, 15, 13, 118)$$

We need second list of data d so that the original list x can be recovered from s and d. For d_k (called directed distances) we have $d_k = (x_{2k} - x_{2k+1})/2$ for $k = 0, \dots, N/2 - 1$. The process is invertible since

$$s_k + d_k = (x_{2k} + x_{2k+1})/2 + (x_{2k} - x_{2k+1})/2 = x_{2k} \tag{3}$$

$$s_k - d_k = (x_{2k} + x_{2k+1})/2 - (x_{2k} - x_{2k+1})/2 = x_{2k+1} \tag{4}$$

So we map $x = (x_0, x_1, \dots x_{N-1})$ to $(s|d) = (s_0, \dots s_{N/2-1}|d_0, \dots d_{N/2-1})$. This process is repeated recursively for s. Using our example values we have

$$(6, 12, 15, 15, 14, 12, 120, 116) \rightarrow (9, 15, 13, 118| - 3, 0, 1, 2)$$

$$(9, 15, 13, 118| - 3, 0, 1, 2) \rightarrow (12, 65.5| - 3, -52.5| - 3, 0, 1, 2)$$

$$(12, 65.5| - 3, -52.5| - 3, 0, 1, 2) \rightarrow (38.75| - 26.75| - 3, -52.5| - 3, 0, 1, 2)$$

So final result is $(38.75| - 26.75| - 3, -52.5| - 3, 0, 1, 2)$. Why might people prefer the data in this form?

- We can identify large changes in the differences portion d of the transform,
- it is easier to quantize the data in this form,
- the transform concentrates the information (energy) in the signal in fewer values,
- and the obvious answer: fewer digits.

To implement optimal algorithm for GPU we have to consider image sizes and GPU limitations. One of the most important parameter is $threadsPerBlock$. This shows number of threads in a single block, which can cooperate together using shared memory. Threads from different blocks can't cooperate. To transform one row we need $N = (width/2 > threadsPerBlock?threadsPerBlock : width/2)$ working threads. We assume that width and height of image are power of 2. Each thread is doing calculation of s_i and d_i, where i is index of thread. If image width is bigger than $2 * threadsPerBlock$, than we cut image to parts and each thread works for each part. For example consider $threadsPerBlock = 512$ and width = 2048. In this case we cut image to two parts and (0, 1024), (1, 1025) ... pixels will be processed by the same thread. So thread will calculate two pair of s and d. After this calculations we need to synchronize threads by calling syncthreads CUDA function, and start second level of transform, which will need to times less threads. So we will run Kernel with

$<< 3 * height, N >>$ parameters, where N is above mentioned threads count, and we multiple height to 3 for R G B components.

In Figure 6 we can see results of our algorithm: 2 level transformed Lena image and 1 level transformed Zelda image.

Figure 6. 2 level FWT for Lena image and 1 level FWT Zelda image.

Experiments done on

- CPU: Intel(R) Core(TM) i3-2100 3,10GHz
- GPU: GeForce 9500 GT, Max threads per block: 512, Max blocks in kernel lunch: 65,535.

We compare the proposed CUDA version of the 2D-FWT with a sequential CPU implementation. We choose images of different sizes (512x512, 1024x1024, 2048x2048 and 4096x4096). We take into account the time needed to copy data and results to and from the GPU.

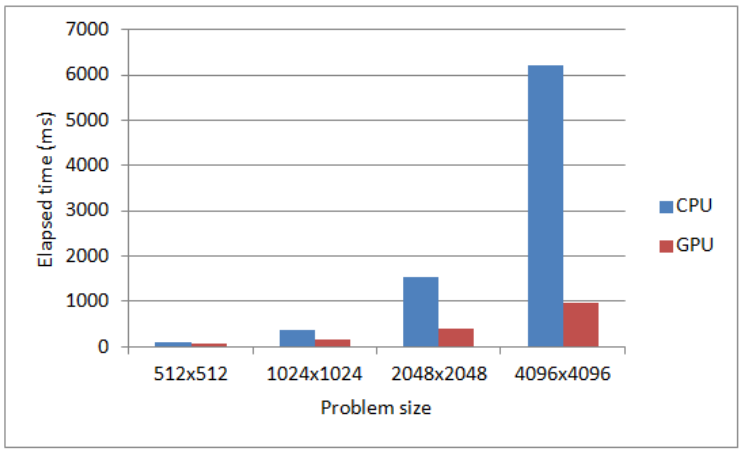

Figure 7. Execution time using CPU and GPU.

5 Conclusion

CUDA is a new hardware and software architecture for issuing and managing computations on the GPU, without the need of mapping them to a graphics API, common to the latest NVIDIA developments. It promises

to simplify the development of applications that take full advantage of current and future powerful GPUs. In this paper we have presented and evaluated an initial implementation of the 2D fast wavelet transform for CUDA-enabled devices. A brief introduction to the wavelet transform and CUDA has been provided prior to explaining our parallelization strategy. We have compared the proposed CUDA version of the 2D-FWT with a sequential implementation. As we can see, we gain in performance using parallel GPU algorithm. Parallelizing the sequential and simple FHT algorithms will be beneficial to control code complexity and minimize execution time of the process.

6 Acknowledgments

I would like to thank Hakob Sarukhanyan for useful and pragmatic suggestions.

References

[1] Wladimir J. van der Laan, Andrei C. Jalba, and Jos B.T.M. Roerdink: Accelerating Wavelet Lifting on Graphics Hardware using CUDA. *IEEE Transactions on Parallel and Distributed Systems*, 132-146, 2011.

[2] Piotr Porwik, Agnieszka Lisowska: The Haar–Wavelet Transform in Digital Image Processing: Its Status and Achievements. *Machine GRAPHICS & VISION vol. 13, no. 1/2* , 79-98, 2004.

[3] Catherine Bénéteau: DISCRETE HAAR WAVELET TRANSFORMS. *PNM Statewide Mathematics Contest*, 2011.

[4] Ivan W. Selesnick: Wavelet Transforms - A Quick Study. *Physics Today magazine*, 2007.

[5] NVIDIA Corp: NVIDIA CUDA C Programming Guide. 2012.

[6] J. Sanders, E. Kandrot: CUDA by Example. 2010.

[7] B. Jähne: Digital Image Processing. 2005.

[8] H. Lensch, R. Strzodka: Massively Parallel Computing with Cuda. 2010.