Implementing Wavelet Transforms on Many-core Architectures — Using CUDA

Samuel Li

May 2015

1 Introduction

Wavelet transform is a technique rooted from the signal processing community [6, 9], and soon people discovered its capacity in data reduction. One of the most prominent uses of wavelet in data reduction has been the JPEG2000 still image compression standard [1, 13].

In the scientific simulation and visualization field, researchers have explored the use of discrete wavelet transforms (DWT) in two relative different directions, both aiming data reduction. In the first direction, researchers use the DWT to provide data access in a multi-resolution fashion, meaning that an approximation of the volume data set is loaded at a lower resolution at first, and the particular region of interest is then reconstructed with higher resolutions [10, 8, 2]. In the second direction, the volume data is reconstructed at the original resolution, but using a subset of the wavelet transformed data. The second direction results in a lossy compression [3, 12].

In a typical use case, the DWT is normally applied on each dimension of a volume data set (e.g. X, Y, and Z dimension). Moreover, in practice, people always apply multiple rounds of DWT to achieve better data reduction. The overall DWT on a volume data set is then becoming a heavy computation task in most systems. As a result, people need a faster way to calculate DWT.

Many-core architectures have emerged recently as accelerators for a traditional computer system. On the one hand, these accelerator cards have much more compute units than a traditional CPU (some NVidia GPUs have thousands of compute units). On the other hand, the compute units on these accelerator cards are relatively simple compared to a CPU, making them not suitable for complex computational tasks.

DWT requires repetitive calculation on arrays of data, with little to no dependency between different arrays. The actual calculations performed on data items are convolutions with the given filters. The width of the wavelet filter decides how many data items are involved in one convolution, which is usually no more than ten. To parallelize this calculation, many convolutions can be performed at the same time. From the perspective of parallel computing, this process is applying a stencil pattern. In this term project, I am going to explore parallelizing the DWT algorithm to fit into the many-core architectures.

2 Implementation Plan

Update: While the project direction stays the same, I made changes to the software frameworks I plan to build on. More specifically, I decided to use CUDA directly for many-core parallelism, instead of the EAVL framework [11] I proposed before. The reason to make this change was that I discovered further that EAVL mainly focuses on the massive parallel operations, like *map*, *reduce*, and *gather*, etc., and not providing an access for a single element in an array to visit its neighbor elements. This ability for a single element to visit its neighbors is essential to apply a stencil pattern. As a result, I switched to CUDA for my parallel implementation. The rest of this section briefly introduces the VAPOR software package that I will be using, and my project plans.

VAPOR [4] is a scientific visualization package, and it highlights the support for data reduction in both directions we talked in Section 1: multi-resolution and lossy compression. VAPOR's data reduction functionality is achieved by using wavelet transforms. Moreover, VAPOR is able to use three different forms of wavelet, which are also known as wavelet kernels: Haar [7], CDF 9/7, and CDF 8/4 [5]. Each wavelet kernel has its advantages and disadvantages in practical use.

I plan to implement my project based on both VAPOR, because VAPOR has already got DWT implemented in serial, including multiple different wavelet kernels. Starting from VAPOR enables me to focus more on the parallel computing component of my project, rather than the details of DWT.

My project plan has three phases:

1. Understand the sequential DWT code from VAPOR, and identify the part(s) that can be parallelized.

2. Parallelize the sequential DWT code using CUDA.

3. Run performance tests and tune my code to achieve a stable version of DWT on many-core architectures.

3 Implementation Details

My implementation focused on parallelizing the calculation of convolution between the wavelet filter and input signal. In the paradigm of CUDA computing, each thread takes care of the calculation of a single element, so instead of using a loop over the entire array, we use a CUDA kernel that identifies the current element index, and does the calculation on that element. Figure 1 shows a CUDA kernel performing convolution on the input signal with the low-pass and high-pass wavelet kernels.

```
 1 __global__ void kernel_conv( double* sigIn, size_t sigInLen,
 2                              double* low_filter, double* high_filter,
 3                              int filterLen, double* cA, double* cD,
 4                              size_t xlstart, size_t xhstart )
 5 {
 6     size_t gindex = threadIdx.x + blockIdx.x * blockDim.x;
 7     size_t yi = gindex * 2;
 8
 9     if( yi < sigInLen ) {
10         size_t yi2 = gindex;
11         cA[ yi2 ] = cD[ yi2 ] = 0.0;
12         size_t xl = xlstart + yi;
13         size_t xh = xhstart + yi;
14         for( int k = filterLen - 1; k >= 0; k-- ) {
15             cA[yi2] += low_filter[k] * sigIn[xl];
16             cD[yi2] += high_filter[k] * sigIn[xh];
17             xl++;
18             xh++;
19         }
20     }
21 }
```

Figure 1: CUDA kernel for performing convolution on the input signal with the low-pass and high0pass wavelet kernels.

Note that the `__global__` syntax indicates that this is a CUDA kernel to be executed on the GPU. Each thread finds out the index of itself in line 6, and checks if it is within the data range before proceeding (line 9).

The invocation of a CUDA kernel also differs from calling a C function. More specifically, we need to pass two pieces of information to the GPU: how many thread in a block, and how many blocks in a grid. These two pieces of information are specified using a triple, indicating the three dimension of a thread block and a grid. The syntax to invoke a CUDA kernel is as follows:

```
kernel_conv<<< (nBlock, 1, 1), (nThread_Per_Block, 1, 1) >>> ();
```

The last mandatory routine to use CUDA kernels is to manage the memory on the GPU card, including allocating memory, copy data back and forth from the system RAM, and free the allocated memory on GPU. The following piece of code illustrates the routine of allocating a chunk of memory on GPU, copying data from RAM to GPU, copy the result from GPU to RAM, and free the allocated memory.

```
double *d_in;
int d_size = sizeof( double );
cudaMalloc( (void**) &d_in, inLen * d_size );
cudaMemcpy( d_in, in, inLen * d_size, cudaMemcpyHostToDevice );
cudaMemcpy( out, d_out, outLen * d_size, cudaMemcpyHostToDevice );
cudaFree( d_in );
```

3.1 Shared Memory

Shared memory lies in the GPU memory hierarchy between the global memory, which can be accessed by all CUDA cores, and the registers, which can only be accessed by the single CUDA cores. Instead, shared memory is associated with each single CUDA multiprocessor, and thus being accessable by all the CUDA cores in that multiprocessor. Because shared memory is significantly faster than global memory, a program can benefit from storing repeatedly accessing data in shared memory, instead of global memory. In my case, the wavelet filters are retrieved by each thread to perform convolution, and shoould be put in the shared memory to increase the performance. The following piece of code shows that we declear a chunck of shared memory to hold the wavelet filter, and each thread fills one data element.

```
__shared__ double s_low_filter[ 9 ];
__shared__ double s_high_filter[ 9 ];
if( threadIdx.x < 9 ) {
    s_low_filter[ threadIdx.x ] = low_filter[ threadIdx.x ];
    s_high_filter[ threadIdx.x ] = high_filter[ threadIdx.x ];
}
__syncthreads();
```

Note that since all threads are executed concurrently, there is a chance that some threads try to read from the shared memory, before it is initialized. To prevent this from happening, we add a `__syncthreads()` line after initializing the shared memory.

4 Performance Study

To study the performance of DWT using CUDA, as well as different parameters of CUDA, I experimented on three perspectives:

1. How much speedup compared to a serial implementation?
2. How much the employment of shared memory improve the performance?
3. How does the thread-per-block parameter affect the performance?

4.1 Speedup Tests

To test the speedups that GPU brings to DWT, I tested my CUDA implementation against the serial implementation on a CPU. The GPU card that I'm testing on is an Nvidia Tesla K40 GPU with 2,880 CUDA cores and 12GB memory. The CPU that I'm testing on is an Intel Xeon E5-2680 v2 Ivy Bridge processor at 2.8GHz. The serial code was also compiled using the Intel compiler and the `-O3` optimization option.

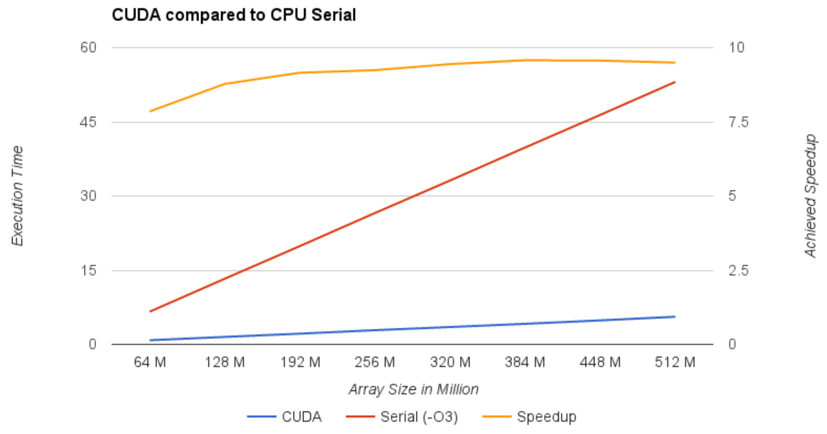


Figure 2: Execution time of DWT on different problem sizes (in seconds), and the corresponding speedups. Notice that the speedup reads are on the right vertical axis.

The test data set was a float array of 512 million points. I tested on this array using part or the whole length of it, eight differences ranging from 64 million to 512 million, to see what the speedup is at each of the problem sizes. Figure 2 summarizes the test results, with the blue showing CUDA execution time, the red line showing serial execution time, and the orange line showing the achieved speedups. Note that the right vertical axis is for the speedups.

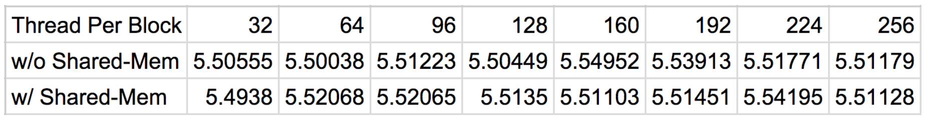
The results show that GPU achieves a consistant 9x speedup when the problem size is big enough, i.e., 256 million. Even at smaller problem sizes, the speedups are also above 7.5x, which are also encouraging. The overall performance of GPU meets my expectation.

4.2 Threads Per Block and Shared Memory Study

Threads per block is an important parameter in launching a CUDA kernel. It should always be a multiplication of 32, bacaues CUDA always groups threads into a "warp" of size 32 to perform tasks. The optimal number is determined by both the hardware capability, e.g., how many CUDA cores are there in a CUDA multiprocessor, as well as the specific problem to tackle. Empirically, the optimal value is somewhere between 128 and 512.

Shared memory speeds up data retrieving (see Section 3.1), so it is expected to help improve the overall performance.

My test varied the thread-per-block parameter from 32 to 256 with eight values in total. With each thread-per-block value, I tested the performance with or without the use of shared memory. These configurations result in sixteen tests in total. I used the 512 million data size for this test. Figure 3 presents the



Thread Per Block	32	64	96	128	160	192	224	256
w/o Shared-Mem	5.50555	5.50038	5.51223	5.50449	5.54952	5.53913	5.51771	5.51179
w/ Shared-Mem	5.4938	5.52068	5.52065	5.5135	5.51103	5.51451	5.54195	5.51128

Figure 3: Execution time of DWT with different threads-per-block parameter, and the configuration with or without shared memory.

execution times with different configurations in seconds.

Surprisingly, the execution times using various configurations are quite consistant: they are all at around 5.5 seconds. Even the use of shared memory did not affect the performance.

One hypothesis that could possibly explain why the use of shared memory did not improve performance, was that the amount of data I manually put in the shared memory is too small (18 doubles in my case). Because these wavelet filters are retrieved once and once again, it is possible that they are kept in the shared memory. This hypothesis needs more evidence to support.

I am not able to explain why the thread per block parameter does not affect the performance at all.

5 Conclusions

We demonstrated in this project that GPU (especially Nvidia GPUs with CUDA) is effective in speeding up programs using a stencil parallel processing pattern. The speedups exhibit good scalability, which means GPU with CUDA could potentially be used in problems with much larger scales.

At the same time, CUDA programming differs from traditional C++ programming in that it requires more tuning to achieve the best performance. This indicates a steeper learning curve of employing CUDA as a productive tool.

References

[1] Michael D Adams. The jpeg-2000 still image compression standard. 2001.

[2] Chuck Baldwin, Ghaleb Abdulla, and Terence Critchlow. Multi-resolution modeling of large scale scientific simulation data. In *Proceedings of the twelfth international conference on Information and knowledge management*, pages 40–48. ACM, 2003.

[3] E Wes Bethel, Hank Childs, and Charles Hansen. *High Performance Visualization: Enabling Extreme-Scale Scientific Insight.* CRC Press, 2012.

[4] John Clyne, Pablo Mininni, Alan Norton, and Mark Rast. Interactive desktop analysis of high resolution simulations: application to turbulent plume dynamics and current sheet formation. *New Journal of Physics*, 9(8):301, 2007.

[5] Albert Cohen, Ingrid Daubechies, and J-C Feauveau. Biorthogonal bases of compactly supported wavelets. *Communications on pure and applied mathematics*, 45(5):485–560, 1992.

[6] Ingrid Daubechies. The wavelet transform, time-frequency localization and signal analysis. *Information Theory, IEEE Transactions on*, 36(5):961–1005, 1990.

[7] Alfred Haar. Zur theorie der orthogonalen funktionensysteme. *Mathematische Annalen*, 69(3):331–371, 1910.

[8] Satoshi Kanai, Hiroaki Date, Takeshi Kishinami, et al. Digital watermarking for 3d polygons using multiresolution wavelet decomposition. In *Proc. Sixth IFIP WG*, volume 5, pages 296–307. Citeseer, 1998.

[9] Stéphane Mallat. *A wavelet tour of signal processing.* Academic press, 1999.

[10] Stephane G Mallat. A theory for multiresolution signal decomposition: the wavelet representation. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 11(7):674–693, 1989.

[11] Jeremy S Meredith, Robert Sisneros, David Pugmire, and Sean Ahern. A distributed data-parallel framework for analysis and visualization algorithm development. In *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*, pages 11–19. ACM, 2012.

[12] Alan Norton and John Clyne. The vapor visualization application. *High Performance Visualization*, 2012.

[13] Bryan E Usevitch. A tutorial on modern lossy wavelet image compression: foundations of jpeg 2000. *Signal Processing Magazine, IEEE*, 18(5):22–35, 2001.