WAVELET ENCODING AND MULTI-GPU PROGRAMMING

ANDRE KESSLER

Abstract. We investigate compression of large-volume spatial data using the wavelet transform, computed massively in parallel on NVIDIA graphics processing units (GPUs). In particular, Haar basis wavelets are used to achieve compression ratios of [100x] or more. Computation is done over a set of computing nodes consisting of multiple nodes and multiple GPUs per node. Significantly more data than can be stored on-board the individual GPUs is streamed on and successfully compressed. After the compression, the data is ready to be analyzed or manipulated by other tools, after which the changed data or extracted features will be decompressed and stored.

Keywords. CUDA, wavelets, multi-GPU, C++, Haar basis, data compression

1. Introduction & Motivation

The ability to process large volumes of spatial data is very important to a wide range of scientific disciplines. Tools to analyze and perform computation on such data are used everywhere from visualizing the results of nuclear magnetic resonance imaging (MRI) scans to processing flight video for aeronautics. Such large-scale computation lends itself well to the growing field of general-purpose computing on graphics processing units (GPGPU), which allows parallelism on a massive scale for a fraction of the cost of such machines in the past.

Raw computation is a particular strong suit of current GPUs–individual cards can hold up to 3.52 teraflops of compute power alone–but the dedicated on-board memory available to GPUs has a lot of room to grow. This is a problem for applications that are very "data-intensive," since the speed of memory transfers from host memory to a GPU is on the scale of an order of magnitude slower than on-chip L1 cache and shared memory accesses.[cite].

We would like to have some way of significantly compressing spatial data so that it fits in the limited on-board memory of a GPU, and decompressed as needed for computation on the same device. Realistically, we may have terabytes of data–in three spatial dimensions, a uniform `float` grid of size $4,000 \times 4,000 \times 4,000$ is already over 2 TB in size.

The wavelet transform will prove to be particularly effective for our purposes. Originally developed for feature extraction of seismic data[cite], the wavelet transform is particular adept at "picking out" areas of data that are well-correlated locally, and thereby can represent sections of highly self-similar data with a small amount of information, while retaining enough to reconstruct fine features in areas of detail.

2. Wavelet Transform

We will use the Haar wavelet, which can be described as follows. Given a sequence of 2^n data points $c_0, c_1, \ldots, c_{2^n-1}$ which we wish to compress, we will split it into two sets: the even-indexed values and the odd-indexed values. The even-indexed values will be kept as *coarse coefficients*, and the odd-indexed ones will be replaced by *details*. We can accomplish this by setting

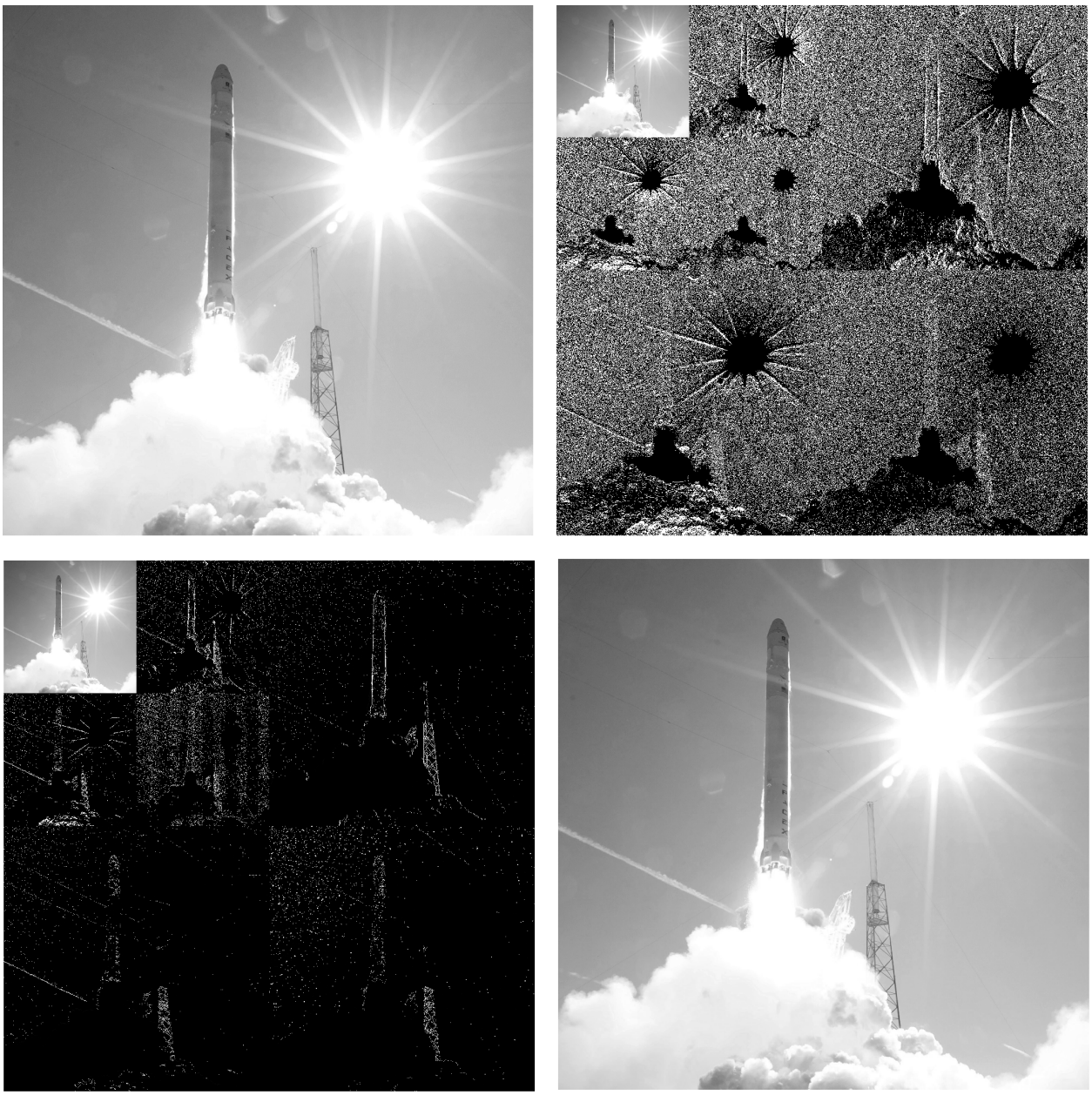

FIGURE 1. The 2-D wavelet transform as computed by our GPU code. From top down and left to right: original, forward transform, thresholding (9.8× compression), and inverse transform.

$$c_k^{+1} = (c_{2k} + c_{2k+1})/2$$
$$d_k^{+1} = c_{2k+1} - c_{2k}$$

That is, the coarse coefficient on the next level is set to the average of the even and odd point, while the detail coefficient is set to be equal to their difference. Notice that given our coarse coefficient c_k^{+1} and detail d_k^{+1}, we can *perfectly reconstruct* the original values c_{2k} and c_{2k+1}. Furthermore, the coarse coefficients represent a downsampled version of the original data, while the details represent corrections

to the average of the data. Details are less important, and we may ignore some while still retaining the ability to reconstruct our data fairly well. See in particular Latu [3] and Sweldens [5].

For example, consider Figure 1. We have an image of a rocket launch (upper left) that we wish to compress. The forward transform of the image is to the right: the downsampling ("coarse coefficients") can be seen as the smaller copy of the rocket image in the upper leftmost corner. Surrounding the small image are three sets of "details": the one to the right are the x-details, to the bottom are the y-details, and diagonally adjacent are the xy-details[1]. This lower level is surrounded by all of the even finer detail coefficients.

After the forward transform, we wish to compress the data. This we accomplish by thresholding, as can be seen in the lower left image. For each level, detail coefficients that are less than a particular threshold value are chosen to be zeroed out. In this particular image, we achieve 9.8× compression, and the inverse transform of the thresholded values (shown to the lower right) is still virtually indistinguishable from the original image. The particular threshold value was chosen so that some details would still be visible in the picture; see Figure **??** for much more signficant compression. Note that despite the "blockiness" seen in more uniform areas of the picture, details like the writing on the rocket, sounding tower, and cloud structure are still remarkably well preserved.

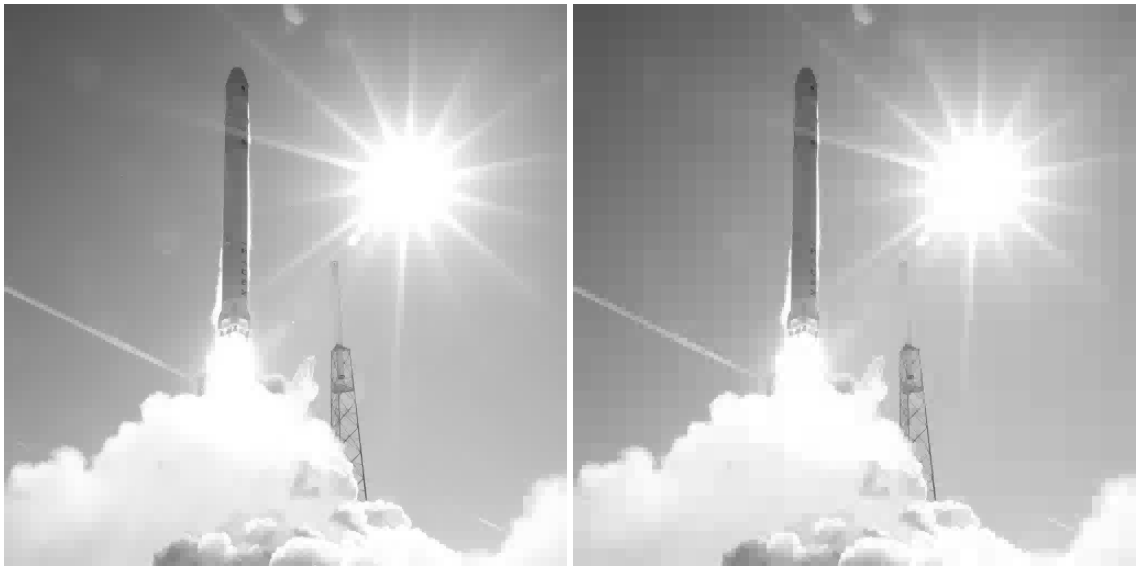

FIGURE 2. Much harsher thresholdings result in more artifacts: 545.33× compression on the left and 1342.18× compression on the right.

3. Data Storage Techniques

Indexing into our data array presents a variety of challenges. Row-major order is inconvenient for several reasons, the first of which being that our data is structured spatially. For any dimension higher than 1-D, row-major order does not well relate to the spatial locality of the data (since adjacent cells are, on average, `row_stride/2` apart). This is a particular concern for GPUs, where memory locality allowing coalesced reads is of utmost importance [6].

[1]Fundamentally, the 2-D wavelet transform has this structure due to its being written as a tensor product of two 1-D wavelet transforms. When we move to the 3-D transform, the 7 adjacent detail regions will correspond to $\{x, y, z, xy, yz, zx, xyz\}$-details.

Table 1. Morton-order indexing of a 4×4 array

0	1	4	5
2	3	6	7
8	9	12	13
10	11	14	15

The solution to these potential issues that we used is the Morton-order indexing scheme, which can be seen in Table 1. It relates to the natural depth-first order of traversal for a quadtree, and exhibits good "chunking" of the data into spatially local sections. For simplicity, the 2-D curve and code is depicted here, but an equivalent traversal exists for 3-D data using a natural depth-first traversal of an octree.

The code for to convert between (x, y) coordinates and Morton order can be accomplished as follows: suppose $x = x_1x_2x_3x_4$ and $y = y_1y_2y_3y_4$, where the right-hand sides are the digits of x and y in binary. Expanding the digits $x_10x_20x_30x_4$ and $y_10y_20y_30y_40$ and logical-OR-ing produces $y_1x_1y_2x_2y_3x_3y_4x_4$, which turns out to be the desired order.

```
__host__ __device__ __forceinline__ size_t expand_bits( size_t x ) {
  x = ( x | ( x << 16 ) ) & 0x0000ffff0000ffffL;
  x = ( x | ( x <<  8 ) ) & 0x00ff00ff00ff00ffL;
  x = ( x | ( x <<  4 ) ) & 0x0f0f0f0f0f0f0f0fL;
  x = ( x | ( x <<  2 ) ) & 0x3333333333333333L;
  x = ( x | ( x <<  1 ) ) & 0x5555555555555555L;
  return x;
}

__host__ __device__ __forceinline__ size_t xy_to_index( size_t x, size_t y ) {
  return expand_bits( x ) | ( expand_bits( y ) << 1 );
}
```

Listing 1. Morton-order conversion using bit-shifts

In our program, all multi-dimensional data will be stored in a flat 1-D array and indexed according to the Morton order. This will also allow us to create a simple data-packing scheme after the wavelet transform is complete: sparse data may be associated to its spatial location with a Morton index. Sorting the array produces an implicit quadtree in memory, which we can then binary search to find particular nodes.

For a fast key-value radix sort in CUDA, we turn to Duane Merrill's excellent CUDA Unbound (CUB)[2] components library. The library's `cub::DeviceRadixSort` can sort up to 0.97 billion 32-bit key-value pairs per second on a Tesla K20. On the

4. Predictions & Results

The computing environment is a cluster of four nodes dedicated for Propulsion Analysis at Space Exploration Technologies (SpaceX). Each node has four NVIDIA Tesla K20m cards for computation;

[2] `http://nvlabs.github.io/cub/`

TABLE 2. Computing environment specifications

Specifications	**1× Tesla K20m**	**4× Tesla K20m**	**Total for cluster**
`double` flops	1.17 Tflops	4.68 Tflops	18.72 Tflops
`float` flops	3.52 Tflops	14.08 Tflops	56.32 Tflops
RAM (GDDR5)	4.8 GB	19.2 GB	76.8 GB
Bandwidth max	208 GB/s	832 GB/s	3328 GB/s

the total computing power is described in Table 2. Each node is hooked up with Infiniband and has GPUDirect[3] enabled; these allow transfers at speeds of up to about 25 Gbits/sec any two nodes.

Given N GB of data on any single node that needs to be compressed, we are first clearly limited by the bandwidth of the network. The next most immediate limitation is the amount of RAM on any one individual card. The theoretical minimum time to completely fill the onboard RAM of one Tesla K20m according to these specifications is approximately 0.02 sec. Of course; we cannot completely fill the RAM of a card; we will need to start off with perhaps half of the RAM filled, wavelet-transform the onboard data, compress, and continue. The important theoretical information is that the time of this host → device memory transfer should be on the order of 10^{-2} seconds. For the test cases, we used an SVG file scaled to the desired resolution and then exported as a binary PGM[4] file. The threshold was chosen as a decaying fixed value: on the finest level, anything less than a 100 on the color scale black (0) → white (255) is removed; on the next coarser level, 50 and lower is removed, 25 on the next, and so on. By way of example, this compression is more coarse than Figure 1, but better resolution than the compression in Figure 2. The jump to 8 GB is the point at which the multi-GPU setup kicks in;

TABLE 3. 2-D Timing Runs

Dimensions	Memory	CPU Fwd	CPU Inv	GPU Fwd	GPU Inv	Compression
$2^{10} \times 2^{10}$	2.0 MB	0.05 s	0.05 s	0.01 s	0.01 s	87.15×
$2^{11} \times 2^{11}$	32.0 MB	0.20 s	0.20 s	0.05 s	0.05 s	97.981×
$2^{12} \times 2^{12}$	128.0 MB	0.79 s	0.76 s	0.20 s	0.18 s	115.47×
$2^{13} \times 2^{13}$	512.0 MB	3.34 s	3.25 s	0.81 s	0.82 s	120.375×
$2^{14} \times 2^{14}$	2.0 GB	19.74 s	17.14 s	3.28 s	3.01 s	234.295×
$2^{15} \times 2^{15}$	8.0 GB	131.18 s	128.01 s	3.31	3.45	346×[5]

we split up the data into four chunks and, in parallel, dispatch them to the GPUs.

Hence, we will be most successful we can stagger the memory copying so that it completes when the wavelet transform of previous data completes, and then the computation can continue without pause onto the next chunk of data.

5. SAMPLE CODE

We store results of the transform in the Mallat ordering as described in Latu [3]. This is then transcribed in the Morton-order curve form to actually be indexed in memory.

[3] `https://developer.nvidia.com/gpudirect`

[4] Portable Gray Map, from the netpbm standard. See `http://netpbm.sourceforge.net/doc/pgm.html`.

```
template<typename T>
void forward_base( T *field_device, size_t N, size_t &finestStep, int maxLevels ) {
  T *temp = new T[N*N];
  int levels = 0;
  for( int step = N; step > 1; step >>= 1 ) {
    wavelet_gpu::splice_x ( field_device, temp, step );
    wavelet_gpu::predict_x( field_device, step, FORWARD_DIRECTION );
    wavelet_gpu::update_x ( field_device, step, FORWARD_DIRECTION );

    wavelet_gpu::splice_y ( field_device, temp, step );
    wavelet_gpu::predict_y( field_device, step, FORWARD_DIRECTION );
    wavelet_gpu::update_y ( field_device, step, FORWARD_DIRECTION );
    levels++; finestStep = step;
    if( levels >= maxLevels ) {
      break;
    }
  }
  delete [] temp;
}
```

Listing 2. C++ wrapper for kernel launches in computation of the forward transform.

The inverse is, indeed, the exact inverse of the predict step; in the 3-D code, splice/predict/update of z occurs last in the forward transform while update/predict/combine occurs first in the inverse transform.

```
template<typename T>
void inverse_base( T *field_device, size_t N, size_t finestStep ) {
  T *temp = new T[N*N];
  for( size_t step = finestStep; step <= N; step <<= 1 ) {
    wavelet_gpu::update_y ( field_device, step, INVERSE_DIRECTION );
    wavelet_gpu::predict_y( field_device, step, INVERSE_DIRECTION );
    wavelet_gpu::combine_y ( field_device, temp, step );

    wavelet_gpu::update_x ( field_device, step, INVERSE_DIRECTION );
    wavelet_gpu::predict_x( field_device, step, INVERSE_DIRECTION );
    wavelet_gpu::combine_x ( field_device, temp, step );
  }
  delete [] temp;
}
```

Listing 3. C++ wrapper for the inverse computation.

Conveniently, the Mallat order memory splice and choice of blocks prior to the update step guarantees the memory predicted/updated will not be overwritten.

```
__global__ void mallat_predict_x( double *field_d, size_t N, int direction ) {
  size_t x, y, loadIndex, storIndex, globalIndex, global_x, global_y;
  double predict;
  x = threadIdx.x; y = threadIdx.y;
  loadIndex = xy_to_index( BLOCKDIM_X * blockIdx.x + x,
                           BLOCKDIM_Y * blockIdx.y + y );
  global_x = index_to_x( globalIndex );
  global_y = index_to_y( globalIndex );
  storIndex = xy_to_index( global_x + N / 2, global_y );

  predict = field_d[loadIndex];
  __syncthreads();
  field_d[storIndex] = field_d[storIndex] - direction * predict;
}
```

Listing 4. Sample kernel for the forward-predict step, launched by wavelet_gpu::predict_x

The updates/predicts for x, y and z are all similarly structured. In all, the major cost as determined by profiling is the memory restructuring between full-level predict/updates.

6. Future work

There are many directions in which this project may be further developed. The first point to note is that this program is itself a basis for a library, rather than a pure application: aside from a basic faster wavelet transform, this project allows for the development of programs that need to deal with extremely large data sets on the GPU– and would benefit from having this data compressed but "on-demand" to be partially decompressed entirely in device memory. It will be interesting to produce visualizations for flight data or MRI scans using this library and explore the various statistics and information about the data that we can compute using the GPU.

With respect to the project itself, both the multi-GPU and wavelet aspects has several possible improvements. We implemented a variant of the Haar basis wavelet, but there are many other families of wavelets available, some of which may offer better compression and more accuracy at the expense of slightly more computation. The Haar wavelet is itself the most basic case of the more general Daubechies wavelet, and there are many other possibilities including Bezier surface patches.

The JPEG2000 and JP3D standards themselves are based on wavelet methods, and with some development effort the parallelization techniques used here could be used in an implementation of those standards. While on average the file formats do not see much use by the general public, they are still highly useful image and video formats for industrial applications, and

Inter-node communication was less explored with this project than it might have been, and it will be useful to see how the full cluster can be exploited for maximum performance. From the use of the shared network file system, one can always split up the data initially so each node will work on its own piece and be responsible for that piece only, but if certain pieces of the data end up being thresholded much more than others, it will most likely result in much better performance if some kind of load-balancing can be implemented over the network.

7. Acknowledgments

I would like to thank Dr. Adam Lichtl of SpaceX for suggesting the focus of the project, providing access to some of SpaceX's extensive GPU computing resources–without which this project simply would not have been possible–as well as for many helpful discussions. I would also like to thank Prof. Alan Edelman of MIT for the opportunity to work on this project for his 6.338/18.337 Parallel Computing course.

References

1. Pamela Cosman and Kenneth Zeger, *Memory Constrained Wavelet Based Image Coding*, **5** (1998), no. 9, 221–223.
2. Tero Karras, *Maximizing parallelism in the construction of bvhs, octrees, and kd trees*, Eurographics/ACM SIGGRAPH Symposium on High Performance Graphics, The Eurographics Association, 2012, pp. 33–37.
3. G Latu, *Sparse data structure design for wavelet-based methods*, ESAIM: Proceedings **34** (2010), no. December 2011, 240–276.
4. Anthony E. Nocentino and Philip J. Rhodes, *Optimizing memory access on gpus using morton order indexing*, Proceedings of the 48th Annual Southeast Regional Conference (New York, NY, USA), ACM SE '10, ACM, 2010, pp. 18:1–18:4.
5. Wim Sweldens and Peter Schroder, *Building Your Own Wavelets at Home*.
6. N. Wilt, *The cuda handbook: A comprehensive guide to gpu programming*, Pearson Education, 2013.

Dept. of Mathematics, Massachusetts Institute of Technology, Cambridge, MA
E-mail address: akessler@mit.edu
URL: www.mit.edu/∼akessler