# Achieving Portable Performance For Wavelet Compression Using Data Parallel Primitives

Shaomeng Li[1,2], Nicole Marsaglia[1], Vincent Chen[1], Christopher Sewell[3], John Clyne[2], and Hank Childs[1]

[1]University of Oregon, [2]National Center for Atmospheric Research, [3]Los Alamos National Laboratory

**Abstract**

*We consider the problem of wavelet compression in the context of portable performance over multiple architectures. We contribute a new implementation of the wavelet transform algorithm that uses data parallel primitives from the VTK-m library. Because of the data parallel primitives approach, our algorithm is hardware-agnostic and yet can run on many-core architectures. We also study the efficacy of this implementation over multiple architectures against hardware-specific comparators. Results show that our performance is portable, scales well, and is comparable to native implementations. Finally, we argue that compression times for large data sets are likely fast enough to fit within in situ constraints, adding to the evidence that wavelet transformation could be an effective in situ compression operator.*

## 1. Introduction

As supercomputers get larger and larger, a consistent trend has been that the ability to generate data is increasing faster than the ability to perform I/O. This trend jeopardizes the traditional paradigm for visualizing computer simulation data. In this paradigm, simulations advance and save their states at regular (or irregular) intervals. Each save is effectively a snapshot in time, or "time slice" of what is happening in the simulation. Importantly, these time slices have typically been stored at their native resolution, meaning that the simulation mesh is not modified, and every field value on that mesh is stored (at least for the fields that are stored).

In response to reduced I/O capabilities, there are three main strategies. First, save data less often. As trends worsen, this strategy may become unpalatable for many application domains, since temporal sparsity can result in lost science. Second, do visualization in situ. This strategy is increasingly being preferred for the cases where domain scientists know what they want to see a priori. However, for data exploration-oriented use cases, where new science is often discovered, there often is no a priori knowledge of what to look for. This observation motivates the third strategy, which is to use a combination of in situ and post hoc techniques. In the in situ phase, data is transformed and reduced, with hopes that the reduction will be sufficient to meet I/O requirements. In the post hoc phase, the transformed and reduced form is available for exploration-oriented use cases. That said, the assumption from the traditional paradigm that data be stored at full resolution and all field values are stored, essentially equates to lossless compression, which limits how much reduction can be achieved. So research in this third strategy often assumes that domain scientists will accept lossy techniques when I/O constraints preclude their traditional workflow. This assumption is important, since allowing for some loss in data integrity enables the strategy to be practical.

There have been many interesting ideas for operators that enable explorative visualization with the in situ+post hoc strategy, including the following. Cinema's [AJO*14] main strategy is to transform the data to images, with the idea being that many, many images will still be smaller than simulation data, and that exploration can happen by loading successive images as if they were being generated by a traditional visualization program. The idea with Lagrangian basis flows [ACG*14] is to transform vector field data into pathlines in situ, and then interpolate new pathlines post hoc from the extracted ones. This technique was shown to be more accurate than saving vector field data, and used less storage as well. As some final examples, Analysis-Driven Refinement [NWP*14], or ADR, prioritized the data to save based on the analyses that would be performed, while Lehmann et al. [LJ14] explored a multi-resolution technique in both space and time. With our study, we consider a different operator, wavelet based compression, partially because of its outstanding compression capabilities known in image processing communities. In fact, wavelet compression is the underlying technology of the JPEG2000 still image compression standard [AT04]. The broader applications of wavelet transform go even beyond compression, with examples being signal denoising, boundary detection, and texture analysis, to name a few.

We specifically focus on efficient execution of wavelet compression over multiple architectures. It connects to the overall theme of in situ+post hoc exploration, in that we are studying how to design wavelet compressors that could be run in situ on many-core architectures. We do not study the resulting tradeoffs between compression and data integrity, since that issue has already been investigated [LGP*15].

A particular emphasis of this study is designing code that will be hardware-agnostic and yet still be performant on each architecture it runs on. Ideally, this approach can "future-proof" our code to run not just on today's architectures, but also tomorrow's. Recent research has demonstrated that designing code using "data parallel primitives" (DPPs) as building blocks is a promising direction for achieving this goal. Therefore, the research involved with this work — and the contribution of this paper — is to re-think wavelet transforms using data parallel primitives and to demonstrate the efficacy of the resulting algorithm.

## 2. Related Work

### 2.1. Parallel Wavelet Transforms on CPUs

Domain decomposition is a popular yet effective approach for achieving parallel processing on CPUs. Using this approach, an entire domain is decomposed into smaller subdomains and each subdomain is processed individually. For 2D matrices, the JPEG2000 standard employs this approach [AT04]. A similar application on multi-node settings is also reported in [Uhl95]. For 3D volumes, VAPOR [CR05, CMNR07], an open-source visualization package with a wavelet compression component, decomposes incoming volumes into $64^3$ cubes by default and then processes them in parallel. Although domain decomposition has the advantage of simplicity, the technique can suffer from blocking effects along subdomain boundaries, which arise from wavelet artifacts on finite-length input boundaries.

More complicated parallel approaches treat the entire domain as a whole while performing wavelet transforms in parallel. These approaches eliminate blocking effects, but introduce inter-processor communications. Nielsen et al. [NH00] developed a parallelization strategy that eliminates a time-consuming distributed matrix transpose, and demonstrates strong scalability. Chaver et al. [CPPT01] partitioned 2D matrices into stripes and studied the performance differences between $X$-partitioning and $Y$-partitioning. Chadha et al. [CCC02] further developed a partitioning strategy where intermediate information exchanges are restricted to neighboring processors. Though proven to be effective on multi-core CPUs and distributed systems, it is unclear how similar strategies would perform on many-core architectures. Also, these strategies seem to have, for the most part, not considered 3D volumes.

### 2.2. Parallel Wavelet Transforms on GPUs

Parallel wavelet transforms on GPUs have been predominantly conducted within the CUDA [NBGS08] framework. Natural parallelization strategies on GPU include row-based and column-based processings, which use a GPU thread to process a row or column of an image at a time [AMN14, EALM15]. Domain decomposition is also used to get the CPU and GPU to work together: a CPU sends subdomains to a GPU to process, and retrieves back the results one-by-one [FBFU10].

A trend in GPU-based wavelet transforms is to exploit the many memory hierarchies on GPU devices to achieve higher speedups, including discussions on the use of shared memory [FBFU10], texture memory [GS05], and even registers [EALM15]. While these fine-grained tunings are very effective in making the most out of the hardware, they usually require a good amount of GPU programming skills, and the performance gains are not guaranteed to translate to another version of hardware.

Finally, we point out that an important use of GPU wavelet transform is to perform on-demand decompression at rendering time. The idea is to postpone decompression to the latest possible stage of the rendering pipeline, which is on GPUs, to reduce the expensive data movement costs. An example of this use is GST [KPM16], where supercompressed textures are decoded on GPUs. A detailed survey on this topic is also available at [BRGIG*14].

### 2.3. Visualization Algorithms With DPPs

Several studies have investigated how to re-think a specific algorithm in the framework of data parallel primitives. They include Maynard et al. with thresholding [MMA*13], Larsen et al. with ray-tracing [LMNC15] and unstructured volume rendering [LLN*15], Schroots and Ma with cell-projected volume rendering [SM15], Lessley et al. with external facelist calculation [LBMC16], and Lo et al. with isosurface generation [LSA12]. Our own work differs in that we are considering a different algorithm (wavelet transform).

### 2.4. Other State-of-the-art Floating Point Compressors

Motivated by the I/O bottleneck on supercomputers, several schemes are designed to specifically compress the floating-point data arising from numerical simulations. Some representatives include FPZip [LI06], ZFP [Lin14], SZ [DC15], and ISABELA [LSE*11]. However, the ability of these schemes to perform well on multiple architectures is still not clear, and this work focuses on how to obtain portable performance for wavelet compression.

## 3. Data Parallel Primitives

In the data parallel paradigm, algorithms are made by composing together so-called data parallel primitives, or DPPs. A DPP specifies the pattern of how an input array is processed in parallel to produce outputs, while users take the responsibility to specify operations applied on each individual element. This user-specified operation is sometimes referred to as "functors" or "worklets." A benefit of using data parallel primitives is that execution details such as thread and memory management are abstracted away from general users, which in turn allows specific implementations to optimize for underlying architectures. Algorithm designers then re-think their algorithms using a relatively small set of data parallel primitives to harness the massive parallelism in modern architectures. Here we briefly describe a few data parallel primitives for demonstration purposes. Readers can consult work by Blelloch [Ble90] for theoretical foundations and Nvidia's Thrust [BH11] for examples in an actual product.

**Map** is a simple yet powerful data parallel primitive — it maps each data element from the input array to an element in the output array. The input and output arrays thus have the same size. Map resembles a traditional `for` loop if there are no loop-carried dependencies. Elements are thus processed in parallel with arbitrary order.

**Scan** also maps an input array to an output array with the same size, but resembles a `for` loop that does have loop-carried dependencies. Scan can be efficiently executed in parallel in a bottom-up fashion.

**Reduce** uses all elements from the input array to produce a single output value, for example the sum or the maximum of the input array. Reduce can also be efficiently executed in parallel in a bottom-up fashion.

**Scatter** and **Gather** are data parallel primitives to facilitate data movement — individual elements are moved in parallel to or from designated locations assuming there are no index conflicts.

In practice, more complex data parallel primitives can be constructed by composing the basic data parallel primitives. This process is useful for providing fundamental algorithms, and an example of this is the **Sort** algorithm in Thrust.

## 4. Algorithm Description

Our compression algorithm consists of two primary steps: wavelet transformation followed by coefficient prioritization. In the first step, input data is transformed into coefficients in the wavelet space using filter banks. In the wavelet space, the magnitude of each coefficient is correlated to its information content. Small magnitude coefficients are often insignificant in reconstructing the original field. Further, in general, wavelet transformation results in the vast majority of its coefficients being small. In the second step, coefficients are prioritized based on their information content, and only the most significant ones are saved on disk, resulting in an overall lossy process. The following subsections provide details about our algorithm's two primary steps as well as our implementation details.

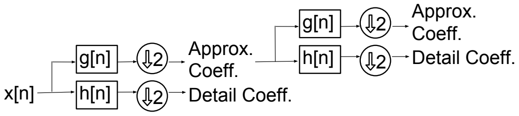

**Figure 1:** *Illustration of a filter bank based wavelet transform workflow. The input signal ($x[n]$) passes through a low-pass and high-pass filter ($g[n]$ and $h[n]$, respectively), and is then down-sampled by a factor of two, resulting in approximation and detail wavelet coefficients. This process is repeated on the approximation coefficients to create a second level wavelet transform.*

### 4.1. Wavelet Transform

Given an input signal, the wavelet transform represents this signal as wavelet coefficients in the wavelet space. There are multiple approaches available to perform the wavelet transform, with filter banks [SN96] and lifting schemes [Swe96] being most popular. We adopted the filter bank approach in this study because of its flexibility; different wavelets can be handled using different filter banks without dramatical changes to the program.

With the filter bank approach, the core operation to calculate wavelet coefficients is discrete convolution. More specifically, we use a two-channel filter bank to perform wavelet transforms, with each filter convolving with the input array (signal) to produce wavelet coefficients. The first channel is a low-pass filter, and the resulting "approximation" coefficients provide a coarsened representation of the signal. The second channel is a high-pass filter, and the resulting "detail" coefficients contain the missing information from the low-pass filtering. The total number of output coefficients is doubled by convolving with two filters. A downsampling step with a factor of two then restores the same number of coefficients to the input array. Despite downsampling, it is still possible to retain all information according to the Nyquist's rule: half frequencies passed through a filter, thus only half coefficients were needed to represent them [Nyq28].

The approximation coefficients are recursively transformed in the same manner — iterating through the filter banks — until a stopping criterion is reached. This practice further decorrelates the approximation coefficients to achieve a better compression. Figure 1 illustrates a two-level wavelet transform workflow.

Wavelet transformation does not directly result in data re-

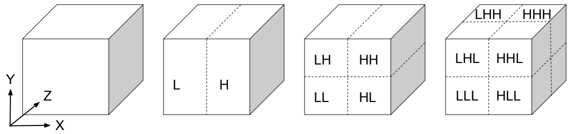

**Figure 2:** *Illustration of one level of wavelet transforms for a three-dimensional cube. Approximation and detail coefficients are denoted using "L" and "H," respectively. From left to right are the original cube, and the resulting coefficients after wavelet transforms in the X, Y, and Z axes, respectively.*

duction. Rather, it "compacts" most information to a few coefficients so coefficient prioritization can effectively reduce the data size. Wavelets with better information compaction capabilities are better suited for lossy compression usage.

#### 4.1.1. Higher Dimensional Wavelet Transform

Using filter banks, wavelet transforms of higher dimensional data can be composed of individual one dimensional transforms along each axis. Consider the three dimensional case as an example. First, each row goes through a wavelet transform pass in the $X$ direction, resulting in approximation and detail coefficients with respect to the $X$ axis. Second, these coefficients then go through wavelet transforms in the $Y$ direction as columns, resulting in approximation and detail coefficients with respect to the $Y$ axis. Third, the output coefficients from the second set of transforms go through wavelet transforms in the $Z$ direction, resulting in approximation and detail coefficients with respect to the $Z$ axis. The motivation of this practice is to decorrelate the signal in each direction for the best compression results. This process is illustrated in Figure 2. This ordering of multidimensional transform is referred to as "non-standard decomposition" in some literatures, and is adopted by all softwares in this study. The pros and cons of "non-standard decomposition" and a few other options are beyond the scope of this paper, and interested readers should consult [SDS95, KP98].

Higher dimensional wavelet transform could also be applied in a recursive fashion. Building on the example in Figure 2, an even coarser version of the data set can be generated by applying additional wavelet transforms to the small cube labeled "LLL." Again, note that the total number of coefficients is constant, regardless of the number of levels of wavelet transforms.

#### 4.1.2. Practical Considerations

Discrete convolution requires special care on the boundaries for finite-length input data. In the general case that the data is not periodic, the data array needs to be extended by half the filter length on both ends, so discrete convolution can perform as usual on the real data. Usually, the extension past the boundary uses the last few elements of the input data array. With an appropriate choice of convolution filter pairs, and careful boundary extensions, mathematically perfect reconstruction is possible with the number of wavelet coefficients matching the number of orginal samples.

The down-sampling step in Figure 1 leaves opportunities to eliminate unnecessary calculation of coefficients, i.e., to skip calculation of coefficients that are meant to be discarded. This is achieved by performing discrete convolution with the low-pass filter on even indexed elements, and the high-pass filter on odd indexed elements.

### 4.2. Coefficient Prioritization

The second step of wavelet compression is to prioritize all coefficients and keep only the ones with the most information content. The heart of this process is a "sort" routine based on the magnitudes of the coefficients. After sorting, a decision is made (typically as input to the compression process) about how many coefficients to save. These coefficients are the largest values. The remaining coefficients are not saved, and treated as zeroes during data reconstruction.

### 4.3. Implementation Specifics

We implemented our algorithm within the VTK-m framework [MSU*16]. VTK-m provides an infrastructure with platform-agnostic data parallel primitives to algorithm developers, and architecture-specific parallelism mechanisms under the hood. Its handling of data parallel primitives means that users can avoid thread scheduling details. Currently, VTK-m has two optimized parallelization mechanisms for its DPPs: CUDA [NBGS08] and Intel TBB [Phe08]. Also, because of the high-level nature of VTK-m, some architectural specifics, such as the different kinds of memories on an Nvidia GPU, are not exposed to its users.

With regards to memory organization, our implementation keeps data in a row-major one-dimensional array regardless of its logical dimensionality. This design means we must face less-than-ideal memory access patterns when accessing data in columns or frames. One potential work-around is to transpose the matrix (or volume) to the desired orientation before performing wavelet transforms along that axis. However, in-place transposition for a matrix (or volume) with different sizes along each dimension is not trivial by itself. We did not choose this optimization for simplicity.

Our implementation supports four wavelets: three members from the CDF [CDF92] wavelet family (CDF 9/7, CDF 8/4, and CDF 5/3), and the Haar wavelet. We used the CDF 9/7 wavelets in this study because it is arguably the best for lossy compression usage (e.g., JPEG2000 in lossy compression mode).

Finally, we note that our 1D and 2D wavelet compressors are already merged into the open-source VTK-m repository, and the 3D case is in the process of being merged.

**Algorithm 1** Worklet for 3D Wavelet Transform in the $X$ Axis

```
Input: $signal$, $workIndex$ {Assigned by VTK-m}
Output: $coefficients$
  $(x,y,z) \leftarrow$ GetLogicalIndex( $workIndex$ )
  if $x$ is even then
    $arr \leftarrow$ ComposeX( $signal, leftExt, rightExt, x, y, z$ )
    $sum \leftarrow$ DiscreteConvolution( $arr, lowWaveletFilter$ )
    $outIdx \leftarrow$ GetOutputIndexApproximationCoeff($x,y,z$)
    $coefficients[\ outIdx\ ] \leftarrow sum$
  else
    $arr \leftarrow$ ComposeX( $signal, leftExt, rightExt, x, y, z$ )
    $sum \leftarrow$ DiscreteConvolution( $arr, highWaveletFilter$ )
    $outIdx \leftarrow$ GetOutputIndexDetailCoeff( $x,y,z$ )
    $coefficients[\ outIdx\ ] \leftarrow sum$
  end if
```

#### 4.3.1. Wavelet Transform with DPPs

We used the "Gather" data parallel primitive to perform signal extension. Gather naturally fits in here since it retrieves elements from designated locations of the signal to extensions (just like gathering). We use specific worklets to guide Gather to correctly handle different dimensionalities and extension directions (e.g., left, right, etc.). Though extending a signal is computationally light because of the small sizes of extensions, implementing them using a data parallel primitive has the additional benefit of avoiding potential data transfers between different computing environments (e.g., between the host and a GPU). This is because DPPs can usually be scheduled to run on designated devices, which allows us to schedule them in the environment where data resides.

Wavelet transforms are carried out using a "Map" data parallel primitive. Details of the transforms, such as wavelet banks and convolution operations, are passed in as worklets. We implemented individual worklets for wavelet transforms in each dimensionality and direction; each worklet resulting in a slightly different Map that performs wavelet transform for one particular case. This practice reduces execution branches inside a worklet, which helps maximize the GPU performance. Algorithm 1 outlines a worklet performing 3D wavelet transforms along the $X$ axis. It assumes that each row of the three-dimensional input is properly extended with an extension on both left and right side (`leftExt` and `rightExt`, respectively), and receives its own work index (`workIndex`) from the VTK-m scheduler, so each instance of the worklet performs convolution on one index: $(x,y,z)$.

#### 4.3.2. Coefficient Prioritization with DPPs

For coefficient prioritization, we used the "Sort" data parallel primitive provided by VTK-m. VTK-m exposes platform-optimized sort when possible. Specifically, it exposes the parallel merge sort from Thrust [BH11] on GPUs, and the parallel quick sort from TBB on CPUs.

## 5. Study Overview

### 5.1. Experiment Overview

We performed our experiments in two rounds. The first round focused on evaluating our own algorithm, while the second round focused on comparing with hardware-specific implementations.

#### 5.1.1. Round 1: Evaluation of the VTK-m Approach

This round was designed to better understand the basic performance of wavelet compression across multiple platforms. It varied two factors:

- Hardware architecture: multi-core CPU and GPU.
- Data sizes: $256^3$, $512^3$, $1,024^3$, and $2,048^3$.

We tested all data sizes on CPU, but skipped the $2,048^3$ data size on GPU due to the GPU memory capacity limitation. We also tested 1D and 2D data inputs for evaluation purposes, and their results yielded similar patterns to 3D inputs. Since 3D data sets are most relevant to HPC applications including simulations and scientific visualizations, we only report 3D results here. We report results from artificial data sets with Gaussian distributions, although the actual data values do not impact performance significantly, because the number of floating point operations and function invocations remains constant for each test size.

#### 5.1.2. Round 2: Comparison with Platform Specific Implementations

This round compared the VTK-m implementation with platform specific implementations for multi-core CPUs and CUDA GPUs, namely VAPOR [CR05, CMNR07] for multi-core CPUs, and a native CUDA implementation for GPUs. These implementations represent the best practices on respective architectures, so they are good comparators for the VTK-m implementation. The total number of configurations for this round is four: VTK-m and VAPOR on multi-core CPUs, and VTK-m and CUDA on GPUs. Again, we opt to only report 3D test results as representatives, and each test is run with multiple problem sizes.

### 5.2. Software Specifications

There are three software packages used in our study: our VTK-m implementation, VAPOR, and a native CUDA implementation. Details about the VTK-m implementation are in Subsection 4.3, so this section focuses on VAPOR and the CUDA implementation.

VAPOR is an open-source software framework consisting of multiple components, including a GUI for post hoc exploration of wavelet-compressed data. For this study, we made use of the standalone wavelet compression utilities included with VAPOR. This program achieves parallel processing through domain decomposition, i.e., a large volume would be decomposed to fixed-sized blocks, and multiple blocks are processed individually and simutaneously using `pthreads`. Coefficient prioritization (described in Subsection 4.2) is performed individually within each block as well using the C++ STL sort.

The native CUDA implementation was written for our study. It followed implementation decisions discussed in [SR16] with adaptations to our GPU. For example, we maxed out the number of threads per block on our GPU to be $1,024$ for larger throughput. Wavelet transforms in each direction ($X$, $Y$, and $Z$) are implemented as separate CUDA kernels for parallel processing. Data is always organized as one-dimensional arrays in the global memory on the GPU without explicit use of shared memory. Thrust sort was used here during coefficient prioritization. Overall, this CUDA implementation has a very similar structure to its VTK-m counterpart, minus the platform-agnostic infrastructure from VTK-m.

Both CPU softwares (VTK-m+TBB and VAPOR) are compiled using GCC, and both GPU softwares (VTK-m+CUDA and native CUDA implementation) are compiled using NVCC with GCC. We turned on `-O2` optimization for all compilations.

### 5.3. Hardware Specifications

To support the tests described in Subsection 5.1, we used the following test systems; both systems are used in both rounds of our testing.

- **CPU System**: Dual socket Intel Xeon Haswell CPUs running at 3.2GHz. There are 16 cores in total, and each core is hyper-threaded to have 2 threads.
- **GPU System**: Nvidia Tesla K40 GPU. There are 2,880 cores in total, each running at 745MHz. This GPU also has 12GB on-board high speed memory.

## 6. Results

The results are organized following the two rounds of our experiments: Subsection 6.1 analyzes the performance of our algorithm over multiple architectures, and Subsection 6.2 compares our performance to hardware-specific implementations.

### 6.1. Performance Analysis of the Algorithm

We separately analyze multi-core CPU performance (6.1.1) and GPU performance (6.1.2).

#### 6.1.1. Multi-core CPU Performance Analysis

Our first set of experiments studied strong scaling of the VTK-m implementation. We ran a baseline of a single core, and then ran additional tests with sixteen cores. In both cases, the compressed volume was the same size. Table 1 shows timing values and speedup factors on four problem sizes. The results show that the transform subroutine achieves near perfect speedups (around $16X$), indicating that the worklet based approach is able to harness the additional CPU cores. In some cases, the speedup numbers are even higher than $16X$. We speculate this is due to the hyper-threading nature of the Xeon CPUs, since VTK-m sees 32 cores through TBB and launches 32 threads for computation. However, the sort subroutine only has speedups from $6.18X$ to $9.49X$. This reduced performance is expected, since sorting requires coordination between the cores.

**Table 1:** *Strong scaling study of VTK-m on 16 Xeon CPU cores. For each problem size, computation time is reported for both transform (shortened as XForm) and sort subroutines (see Section 4) in seconds. The achieved speedup is reported in the last column.*

| Size | Subroutine | 1-core | 16-core | Speedups |
|---|---|---|---|---|
| $256^3$ | XForm | 4.72 | 0.33 | 14.30X |
| | Sort | 1.36 | 0.22 | 6.18X |
| $512^3$ | XForm | 37.22 | 2.06 | 18.07X |
| | Sort | 12.23 | 1.41 | 8.67X |
| $1,024^3$ | XForm | 298.67 | 16.22 | 18.41X |
| | Sort | 103.75 | 13.32 | 7.79X |
| $2,048^3$ | XForm | 2512.10 | 131.40 | 19.12X |
| | Sort | 884.53 | 93.18 | 9.49X |

**Table 2:** *Factor of computational time increase from a smaller to a bigger problem size. Values in this table are derived from the 16-core results in Table 1.*

| Size Incr. | XForm Time Incr. | Sort Time Incr. |
|---|---|---|
| $256^3 \rightarrow 512^3$ | 6.24X | 6.41X |
| $512^3 \rightarrow 1,024^3$ | 7.87X | 9.45X |
| $1,024^3 \rightarrow 2,048^3$ | 8.10X | 7.00X |

Our second set of experiments looked at the execution time increase as the problem size grows. We calculate the ratio of execution times using the sixteen core results and list them in Table 2. The problem size grows by 8 at each step. This table shows that both transform and sort subroutines take close to $8X$ more time to finish processing the next problem size. This result indicates that this implementation is not slowing down as we approach data sizes up to $2,048^3$.

#### 6.1.2. GPU Performance Analysis

Our first set of experiments measure raw performance on the GPU. Table 3 provides the time the GPU takes to perform wavelet compression on three data sizes: $256^3$, $512^3$, and $1,024^3$. We did not test the $2,048^3$ data size because it exceeded the memory capacity on our GPU. These tests show a significant performance boost compared to 16-core CPUs. Given that this is the same code base compiled on two very

**Table 3:** *Wavelet fransform and sorting time on a Tesla K40 GPU in seconds. The factor of time increase from the previous problem size is indicated in parentheses.*

| Size | XForm Time | Sort Time |
|---|---|---|
| $256^3$ | 0.0463 | 0.0445 |
| $512^3$ | 0.3177 (6.86X) | 0.3834 (8.62X) |
| $1,024^3$ | 2.4419 (7.69X) | 3.1766 (8.29X) |

**Table 4:** *Theoretical and achieved occupancy of our wavelet compressor on a Tesla K40 GPU. The transform subroutine was implemented as a worklet, and the sort subroutine was a data parallel primitive provided by VTK-m.*

| | Theoretical Occupancy | Achieved Occupancy |
|---|---|---|
| XForm | 75% | 70.3% |
| Sort | 50% | 49.4% |

distinct architectures, it shows that the performance can be portable. Also, the execution time increase is in line with the problem size growth: it takes roughly $8X$ more time to solve an $8X$ larger problem.

Secondly we use *occupancy* reported by the Nvidia Visual Profiler to assess the efficiency of the VTK-m program. In Nvidia's model, adjacent threads are grouped into warps. There is a maximum number of warps that can be concurrently active on a Streaming Multiprocessor depending on the underlying hardware. Occupancy is then defined as the ratio of active warps to the maximum number of active warps supported by the Streaming Multiprocessor. It is not always possible to achieve a 100% occupancy for a general program because of limiting factors in compilation and GPU invocation specifics (more details can be found in Nvidia documentation [Nvi]). As a result, the Nvidia Visual Profiler reports a theoretical occupancy as well as an achieved occupancy. The achieved occupancy cannot reach the theoretical occupancy when the scheduler is not able to issue sufficient instructions because of data or instruction dependencies. We report both occupancy metrics in Table 4 for two major subroutines in our algorithm: wavelet transform and sort.

The occupancy results are generally good, with the wavelet transform worklet achieving a higher occupancy. This is because of the nature of the wavelet transform that worklets working on individual convolutions are more independent with each other than sorting. For both subroutines, the Nvidia Visual Profiler suggests that the occupancy is large enough that further improvements in occupancy may not improve performance.

We note that for large-scale simulations on supercomputers, a $1,024^3$ cube is on a par with problem sizes a single compute node normally processes. We argue that the achieved compression speed on GPUs, e.g., under six seconds for a $1,024^3$ cube, is likely fast enough to fit within in situ requirements and facilitate the in situ+post hoc strategy to alleviate I/O constraints.

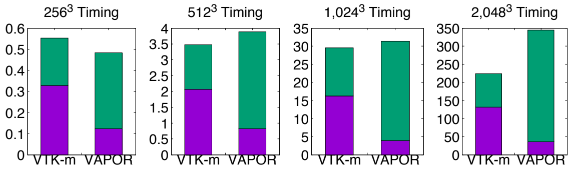

**Figure 3:** *Comparison of execution time (in seconds) between VTK-m and VAPOR. The purple part is for wavelet transforms, and green is for sorting.*

## 6.2. Comparisons With Hardware-Specific Software

### 6.2.1. VAPOR

As previously discussed, VAPOR achieves parallel processing via domain decomposition and `pthreads` (see Subsection 5.2). For the tests on different size data sets, we maintained the number of total subdomains at 64, allowing VAPOR to make full use of our 16-core machine. VAPOR processes each subdomain following the transform and sort subroutines as the VTK-m implementation does. We note that the local sort within each subdomain actually results in fewer calculations than the global sort in VTK-m, but for simplicity in comparison, we consider the sort time to be local for VAPOR and global for VTK-m.

Figure 3 shows the performance comparison between VTK-m and VAPOR. These results show that VTK-m and VAPOR have comparable performance with VTK-m being faster in three of the four test sizes. However, a more prominent difference is how they allocate time differently between their two subroutines. While VTK-m spends more than half its time performing wavelet transforms, VAPOR spends less than a quarter, especially as the problem size grows. This result is interesting since it shows that our DPP-based wavelet transform is $3X$ to $4X$ slower than the best implementations on CPU.

We speculate two design choices by VAPOR contributed to its superior performance: slice-by-slice data processing, and transposition for cache alignment. Both design choices aim to better use the caching mechanism on CPUs. First, a slice from the subdomains that VAPOR processes is most likely to fit into the last level of cache in modern CPUs. For example, a slice from $512^3$ subdomains is 1MByte in 32-bit `float` or 2MByte in 64-bit `double` type, which can easily fit into the 20MB L3 data cache per CPU socket (40MB in total) in our test system. Second, VAPOR transposes data to align arrays in storage to the one dimensional wavelet transforms about to be performed, further increasing cache utilizations in smaller but faster L2 and L1 caches. On the contrary, our data parallel primitive based transform schedules worklets to process arrays as long as one entire volume di-

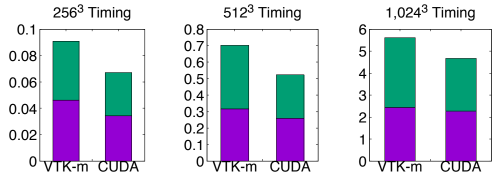

**Figure 4:** *Comparison of execution time (in seconds) between VTK-m and CUDA. The purple part is for wavelet transforms, and green is for sorting.*

mension without certain orderings, hardly making good use of the caching mechanism.

In terms of the time cost for sorting, the STL sort employed by VAPOR does not perform as well as VTK-m's sort, which is TBB's sort for CPUs. One might think that replacing the STL sort in VAPOR to TBB sort could be a simple solution to increase VAPOR's performance. However, it would not be that easy, since VAPOR is already parallelizing across cores for the domain decomposition, and thus the sort for each subdomain can only use a single thread.

#### 6.2.2. Native CUDA Implementation

Figure 4 compares the performance difference between the VTK-m and native CUDA implementations. Since they share similar parallelization strategies (see Section 5.2), this comparison actually quantifies the performance overhead of VTK-m on GPUs. As the results show, this overhead is always within 40% of the CUDA performance. In fact, this overhead has a trend to decrease as data size grows (i.e., from 35% at $256^3$ to 20% at $1,024^3$).

## 7. Conclusions and Future work

This paper explored a new approach to implement a wavelet compression algorithm, distinguished in its aim to achieve portable performance over multiple architectures. This new approach made use of the data parallel primitive paradigm, which aims to future-proof for emerging architectures. We showed that our performance is comparable with two hardware-specific softwares on multi-core CPUs and Nvidia GPUs. The GPU comparison also quantifies the VTK-m overhead to be no more than 40% of its native CUDA counterpart.

For future work, we would like to explore techniques that enable us to process larger data sets on GPUs despite their constrained memory capacity, for example, the greatly enhanced unified memory from CUDA 8.

## Acknowledgement

This work was supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research under Award Number 14-017566 ("XVis: Visualization for the Extreme-Scale Scientific-Computation Ecosystem"), Program Manager Lucy Nowell. It utilized the VTK-m software, developed by the Institute of Scalable Data Management, Analysis and Visualization (SDAV) under the Scientific Discovery through Advanced Computing (SciDAC) program.

National Center for Atmospheric Research (NCAR) also wishes to thank our primary sponsor, the National Science Foundation.

## References

[ACG*14] Agranovsky A., Camp D., Garth C., Bethel E. W., Joy K. I., Childs H.: Improved Post Hoc Flow Analysis Via Lagrangian Representations. In *Proceedings of the IEEE Symposium on Large Data Visualization and Analysis (LDAV)* (Paris, France, Nov. 2014), pp. 67–75. 1

[AJO*14] Ahrens J., Jourdain S., O'Leary P., Patchett J., Rogers D. H., Petersen M.: An image-based approach to extreme scale in situ visualization and analysis. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (2014), IEEE Press, pp. 424–434. 1

[AMN14] Ao J., Mitra S., Nutter B.: Fast and efficient lossless image compression based on CUDA parallel wavelet tree encoding. In *Image Analysis and Interpretation (SSIAI), 2014 IEEE Southwest Symposium on* (2014), IEEE, pp. 21–24. 2

[AT04] Acharya T., Tsai P.-S.: *JPEG2000 Standard for Image Compression: Concepts, Algorithms and VLSI Architectures*. Wiley-Interscience, 2004. 2

[BH11] Bell N., Hoberock J.: Thrust: A productivity-oriented library for CUDA. *GPU computing gems Jade edition 2* (2011), 359–371. 3, 5

[Ble90] Blelloch G. E.: *Vector models for data-parallel computing*, vol. 75. MIT press Cambridge, 1990. 3

[BRGIG*14] Balsa Rodríguez M., Gobbetti E., Iglesias Guitián J., Makhinya M., Marton F., Pajarola R., Suter S. K.: State-of-the-art in compressed GPU-based direct volume rendering. In *Computer Graphics Forum* (2014), vol. 33, Wiley Online Library, pp. 77–100. 2

[CCC02] Chadha N., Cuhadar A., Card H.: Scalable parallel wavelet transforms for image processing. In *Electrical and Computer Engineering, 2002. IEEE CCECE 2002. Canadian Conference on* (2002), vol. 2, IEEE, pp. 851–856. 2

[CDF92] Cohen A., Daubechies I., Feauveau J.-C.: Biorthogonal bases of compactly supported wavelets. *Communications on pure and applied mathematics 45*, 5 (1992), 485–560. 4

[CMNR07] Clyne J., Mininni P., Norton A., Rast M.: Interactive desktop analysis of high resolution simulations: application to turbulent plume dynamics and current sheet formation. *New Journal of Physics 9*, 8 (2007), 301. 2, 5

[CPPT01] Chaver D., Prieto M., Piñuel L., Tirado F.: Parallel wavelet transform for large scale image processing. In *Parallel and Distributed Processing Symposium., Proceedings International, IPDPS 2002, Abstracts and CD-ROM* (2001), IEEE, pp. 6–pp. 2

[CR05] Clyne J., Rast M.: A prototype discovery environment for analyzing and visualizing terascale turbulent fluid flow simulations. In *Electronic Imaging 2005* (2005), International Society for Optics and Photonics, pp. 284–294. 2, 5

[DC15] DI S., CAPPELLO F.: Fast error-bounded lossy hpc data compression with sz. In *IPDPS 2016* (Chicago, IL, 06/2016 2015), IEEE, IEEE. 2

[EALM15] ENFEDAQUE P., AULI-LLINAS F., MOURE J. C.: Implementation of the dwt in a GPU through a register-based strategy. *IEEE Transactions on Parallel and Distributed Systems 26*, 12 (2015), 3394–3406. 2

[FBFU10] FRANCO J., BERNABÉ G., FERNÁNDEZ J., UJALDÓN M.: Parallel 3d fast wavelet transform on manycore GPUs and multicore CPUs. *Procedia Computer Science 1*, 1 (2010), 1101–1110. 2

[GS05] GARCIA A., SHEN H.-W.: GPU-based 3d wavelet reconstruction with tileboarding. *The Visual Computer 21*, 8 (2005), 755–763. 2

[KP98] KOPP M., PURGATHOFER W.: Interleaved dimension decomposition: a new decomposition method for wavelets and its application to computer graphics. 4

[KPM16] KRAJCEVSKI P., PRATAPA S., MANOCHA D.: GST: GPU-decodable supercompressed textures. *ACM Transactions on Graphics (TOG) 35*, 6 (2016), 230. 2

[LBMC16] LESSLEY B., BINYAHIB R., MAYNARD R., CHILDS H.: External Facelist Calculation with Data-Parallel Primitives. In *Proceedings of EuroGraphics Symposium on Parallel Graphics and Visualization (EGPGV)* (Groningen, The Netherlands, June 2016), pp. 10–20. 2

[LGP*15] LI S., GRUCHALLA K., POTTER K., CLYNE J., CHILDS H.: Evaluating the Efficacy of Wavelet Configurations on Turbulent-Flow Data. In *Proceedings of IEEE Symposium on Large Data Analysis and Visualization (LDAV)* (Chicago, IL, Oct. 2015), pp. 81–89. 2

[LI06] LINDSTROM P., ISENBURG M.: Fast and efficient compression of floating-point data. *Visualization and Computer Graphics, IEEE Transactions on 12*, 5 (2006), 1245–1250. 2

[Lin14] LINDSTROM P.: Fixed-rate compressed floating-point arrays. *IEEE transactions on visualization and computer graphics 20*, 12 (2014), 2674–2683. 2

[LJ14] LEHMANN H., JUNG B.: In-situ multi-resolution and temporal data compression for visual exploration of large-scale scientific simulations. In *2014 IEEE 4th Symposium on Large Data Analysis and Visualization* (2014), IEEE, pp. 51–58. 1

[LLN*15] LARSEN M., LABASAN S., NAVRÁTIL P., MEREDITH J., CHILDS H.: Volume Rendering Via Data-Parallel Primitives. In *Proceedings of EuroGraphics Symposium on Parallel Graphics and Visualization (EGPGV)* (Cagliari, Italy, May 2015), pp. 53–62. 2

[LMNC15] LARSEN M., MEREDITH J., NAVRÁTIL P., CHILDS H.: Ray-Tracing Within a Data Parallel Framework. In *Proceedings of the IEEE Pacific Visualization Symposium* (Hangzhou, China, Apr. 2015), pp. 279–286. 2

[LSA12] LO L.-T., SEWELL C., AHRENS J. P.: Piston: A portable cross-platform framework for data-parallel visualization operators. In *EGPGV* (2012), pp. 11–20. 2

[LSE*11] LAKSHMINARASIMHAN S., SHAH N., ETHIER S., KLASKY S., LATHAM R., ROSS R., SAMATOVA N. F.: Compressing the incompressible with isabela: In-situ reduction of spatio-temporal data. In *Euro-Par 2011 Parallel Processing*. Springer, 2011, pp. 366–379. 2

[MMA*13] MAYNARD R., MORELAND K., ATYACHIT U., GEVECI B., MA K.-L.: Optimizing threshold for extreme scale analysis. In *IS&T/SPIE Electronic Imaging* (2013), International Society for Optics and Photonics, pp. 86540Y–86540Y. 2

[MSU*16] MORELAND K., SEWELL C., USHER W., LO L.-T., MEREDITH J., PUGMIRE D., KRESS J., SCHROOTS H., MA K.-L., CHILDS H., ET AL.: Vtk-m: Accelerating the visualization toolkit for massively threaded architectures. *IEEE Computer Graphics and Applications 36*, 3 (2016), 48–58. 4

[NBGS08] NICKOLLS J., BUCK I., GARLAND M., SKADRON K.: Scalable parallel programming with CUDA. *Queue 6*, 2 (Mar. 2008), 40–53. 2, 4

[NH00] NIELSEN O. M., HEGLAND M.: Parallel performance of fast wavelet transforms. *International Journal of High Speed Computing 11*, 01 (2000), 55–74. 2

[Nvi] NVIDIA C.: Achieved occupancy. `https://goo.gl/xHfGlO`. Accessed: Mar 4 2017. 7

[NWP*14] NOUANESENGSY B., WOODRING J., PATCHETT J., MYERS K., AHRENS J.: Adr visualization: A generalized framework for ranking large-scale scientific data using analysis-driven refinement. In *Large Data Analysis and Visualization (LDAV), 2014 IEEE 4th Symposium on* (2014), IEEE, pp. 43–50. 1

[Nyq28] NYQUIST H.: Certain topics in telegraph transmission theory. *Transactions of the American Institute of Electrical Engineers 47*, 2 (1928), 617–644. 3

[Phe08] PHEATT C.: Intel® threading building blocks. *Journal of Computing Sciences in Colleges 23*, 4 (2008), 298–298. 4

[SDS95] STOLLNITZ E. J., DEROSE A. D., SALESIN D. H.: Wavelets for computer graphics: a primer. 1. *IEEE Computer Graphics and Applications 15*, 3 (1995), 76–84. 4

[SM15] SCHROOTS H. A., MA K.-L.: Volume Rendering with Data Parallel Visualization Frameworks for Emerging High Performance Computing Architectures. In *SIGGRAPH Asia 2015 Visualization in High Performance Computing* (2015), SA '15, ACM, pp. 3:1–3:4. 2

[SN96] STRANG G., NGUYEN T.: *Wavelets and filter banks*. SIAM, 1996. 3

[SR16] SCIVOLETTO A., ROMANO N.: Performances of a parallel cuda program for a biorthogonal wavelet filter. In *Proceedings of the International Symposium for Young Scientists in Technology, Engineering and Mathematics (System)* (2016). 6

[Swe96] SWELDENS W.: The lifting scheme: A custom-design construction of biorthogonal wavelets. *Applied and computational harmonic analysis 3*, 2 (1996), 186–200. 3

[Uhl95] UHL A.: A parallel wavelet image block-coding algorithm. In *Proceedings of Intern. Conference on High Performance Computing* (1995). 2