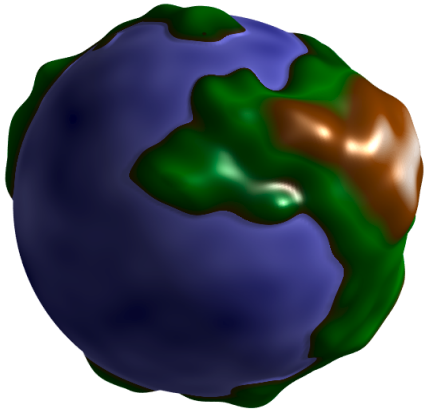
Utilizing GPU Parallelism to Improve Fast Spherical Harmonic Transforms

Max Carlson
University of Utah
Salt Lake City, Utah
mcarlson@cs.utah.edu

Hari Sundar
University of Utah
Salt Lake City, Utah
hari@cs.utah.edu



***Abstract*—Spherical harmonics form an orthogonal basis for functions that live on the surface of a sphere and are useful for solving partial differential equations and for numerical integration. The complexity of transforming a set of function samples to their corresponding spherical harmonic coefficients is largely dominated by the computation of the associated Legendre transform. This associated Legendre transform requires the computation of $(L+1)$ dense matrix-vector products where L is the order of the spherical harmonic expansion. Since the number of rows and columns of each of these matrices depends on L, this step is essentially $O(L^3)$. In this paper, we explore the GPU parallelism available to improve the butterfly compression approach. We present some preliminary results showing performance increases for large problem sizes and eventually plan to release the MonarchSHT library for GPU spherical harmonic transforms.**

I. Introduction

Representing functions with the Fourier basis is a very useful and well studied technique for solving PDEs, numerically evaluating integrals, and for processing and analyzing signals. Unfortunately, the Fourier basis is not suitable for representing functions that live on the surface of a sphere and thus a spherical basis is necessary. The spherical harmonic basis is one such representation and is analogous to the Fourier basis in that each basis function is an eigenfunction of the Laplace operator and satisfies many of the same properties.

The Spherical Harmonic Transform can then be used for many of the same things the Fourier transform can be used for. For instance, for a time-dependent PDE defined on the surface of a sphere, the SHT can be analytically applied to the problem to get an ODE in terms of the spherical harmonic coefficients. Then by solving these ODEs for the coefficients, the solution can be reconstructed using an SHT.

Another potential application for SHTs is in computer generated graphics. Fourier transforms can be used to generate fractal noise which makes realistic looking terrain. Using the same process, SHTs can be used to create fractal noise over a sphere to get realistic looking planets or asteroids. A toy example of this can be seen in the title graphic which was produced using a small value of L. Typically this method isn't used much since SHTs can be expensive to compute without efficient algorithms.

For 3D models that are topologically like a sphere, each vertex can be thought of as a point on the sphere that has been displaced by some vector. In this case, these shapes can be transformed into spherical harmonic coefficients. This is potentially useful in shape analysis by performing SHT on many such models and then using statistical analysis on the resulting coefficients to find ways of classifying different shapes. [1]

In order to reduce the amount of time required to compute a spherical harmonic transform (forward or backward), the choice of discretization goes a long way. By choosing points on the sphere surface such that they form horizontal rings, the transform can be decomposed into a series of matrix-vector products followed by a fast Fourier transform, resulting in an algorithm with reduced complexity compared to the naive transformation. [2] This improvement is widely used among fast SHT algorithms and is fairly standard practice. There are then two algorithms that improve on this standard fast algorithm using two different approaches.

The first of these is the algorithm used by the S2hat [3] software library that takes this fast SHT algorithm and utilizes GPU parallelism to further reduce the complexity of the direct transform. Then there is Wavemoth [4] which uses a butterfly compression scheme to reduce the amount of data needed to compute and store the associated Legendre transform to $O(L^2 \log L)$ but does so without utilizing GPU parallelism. The goal of this paper is to combine the Wavemoth compression idea with GPU acceleration to create a highly scalable, fast, spherical harmonic transform.

II. Background

A. Spherical Harmonic Transform

Similar to the Fourier basis, a discrete function defined on the surface of a sphere can be represented exactly as an infinite sum of spherical harmonic coefficients multiplied by the basis functions typically denoted by Y_l^m. Then by assuming that the basis functions evaluate to 0 for basis functions with degree larger than some parameter L, a function f can be written as the following finite sum

$$f(\theta_k, \phi_{k,j}) = \sum_{m=-L}^{L} \sum_{l=|m|}^{L} a_{lm} Y_l^m(\theta_k, \phi k, j) \tag{1}$$

The spherical harmonic basis functions can then be separated such that $Y_l^m(\theta, \phi) = \overline{P}_l^m(\cos(\theta))e^{im\phi}$ where $\overline{P}_l^m$ are the normalized associated Legendre functions. This then means that the spherical harmonic expansion (1) can be rewritten to get (2).

$$f(\theta_k, \phi_{k,j}) = \sum_{m=-L}^{L} e^{im\phi_{k,j}} [\sum_{l=|m|}^{L} a_{lm} \overline{P}_l^m(\theta_k, \phi_{k,j})] \tag{2}$$

From here it can be seen that the inner summation is the matrix-vector product seen in (3) where Λ_m is the matrix of size $(N_{\text{rings}} \times L + 1 - |m|)$ and the k, l-th entry is $\overline{P}_l^m(\cos\theta_k)$.

$$q_m^{(k)} = \sum_{l=|m|}^{L} \overline{P}_l^m(\cos\theta_k) a_{l,m} \qquad \mathbf{q}_m = \mathbf{\Lambda}_m \mathbf{a}_m \tag{3}$$

Then using the computed $\mathbf{q}_m$, the spherical harmonic expansion reduces to (4). At this point, if there are L evenly spaced points on the ring starting at $\phi = 0$, this can be computed simply by using the standard FFT. For more interesting discretizations, this step requires a bit more processing before it is ready for FFT but as long as the points are arranged in horizontal rings, this step can be accelerated using FFT. This is the traditional first reduction in complexity for fast SHT algorithms. [4], [5], [2], [6], [7]

By using the FFT, the total complexity of the spherical harmonic transform is dominated by applying the matrices Λ_m of associated Legendre coefficients. The butterfly compression scheme that follows is an attempt to reduce the complexity of this step, allowing for larger transforms to be computed in a reasonable amount of time.

$$f(\theta_k, \phi_{k,j}) = \sum_{m=-L}^{L} q_m^{(k)} e^{im\phi_{k,j}} \tag{4}$$

Computing the spherical harmonic coefficients given function samples is slightly more complex. Instead of simply evaluating a sum, the integral in equation (5) must be evaluated. This can be done by choosing a discretization such that the value of $\cos(\theta_k)$ for each ring is the k-th Legendre quadrature node and then evaluating the sum seen in (6).

$$a_{lm} = \int f(\theta, \phi) \tilde{P}_l^m(\cos(\theta)) e^{-im\phi} \tag{5}$$

This sum can again be separated like in the inverse transform. Then computing the coefficients is simply a matter of computing the inverse FFT along each ring, and then applying the transpose of Λ_m in addition to the quadrature weights.

$$\sum_j w_j f(\theta_j, \phi_j) \tilde{P}_l^m(\cos(\theta_j)) e^{-im\phi_j} \tag{6}$$

B. Butterfly Compression

The first property of Λ_m to note is that each of these matrices is dense, with no non-zero entries. They are also full rank and can't be compressed using usual techniques such as singular value decomposition or interpolative decomposition. However, for $m \in \{0, 1, ..., M\}$, Λ_m can be broken into two rows and q columns of blocks as seen in (7) such that each block A_j is rank deficient.

$$\Lambda_m = \begin{bmatrix} A_0 & A_1 & ... & A_{q-2} & A_{q-1} \\ A_q & A_{q+1} & ... & A_{2q-2} & A_{2q-1} \end{bmatrix} \tag{7}$$

For the Λ's that satisfy this property, a technique known as butterfly factorization [4], [8] can be applied to reduce the memory needed to store Λ in addition to reducing the number of operations necessary to apply Λ to a vector.

According to Tygert (2010) [9], the amount of data needed to store the compressed Λ_m is roughly $O(L \log L)$. Since there are $O(L)$ matrices, the total amount of work to apply the compressed associated Legendre transform is $O(L^2 \log L)$.

Since each of the blocks that make up Λ_m are rank deficient, they can be factored into the form seen in (8) using interpolative decomposition (ID). The output of this factorization is the three matrices $\hat{A}_j^i$, T_j^i, and P_j^i. The permutation matrix P_j^i can be encoded as a vector and in this form will be referred to as p_j^i.

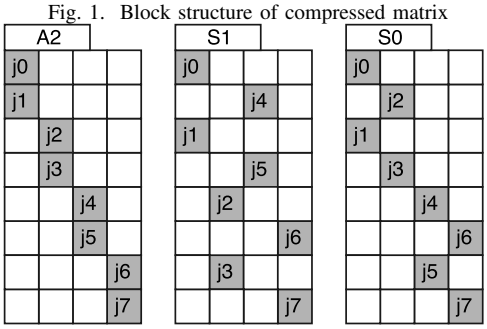
$$A_j^{(i)} = (\hat{A}\,[I|T]\,P)_j^{(i)} = \hat{A}_j^{(i)} S_j^{(i)} P_j^{(i)} \tag{8}$$

The compression scheme then works by computing the ID of each block, storing the $\tilde{A}_j$ matrices and permutations for later use in multiplication and then joining the column skeletons to form the next set of blocks for step i, A_j^i. The full process is outlined below in Algorithm 1.

Algorithm 1 Butterfly Compression

```
procedure ButterflyCompress
    Input: Matrix A of size m × n
    Break A into 2q blocks A_j^(0)
    for i ∈ {1, ..., Q} do
        for j ∈ {0, 1, ..., 2q − 1} do
            Compute interpolative decomposition of A_j^(i−1)
            Store Ã_j^(i) and p_j^(i)
        s_i = 2^(Q−i+1)
        join-and-split(i, s_i) to get A^i
```

Fig. 1. Block structure of compressed matrix



The final compressed matrix is then of the form seen in (9) where each matrix $A^{(i)}$ contains the stored matrices $\tilde{T}_j^{(i)}$ and permutation vectors $p_j^{(i)}$. Then $A^{(Q)}$ is the block diagonal matrix where each block is $\hat{A}_j$ and $\hat{A}_{j+s_Q}$ joined vertically.

$$A = A^{(Q)} P^{(Q-1)} A^{(Q-1)} P^{(Q-2)} A^{(Q-2)} \cdots P^{(0)} A^{(0)} \tag{9}$$

$$S_i = \begin{bmatrix} I_0 & T_0^{(i)} p_0^{(i)} & & & \\ I_{s_i} & T_{s_i}^{(i)} p_{s_i}^{(i)} & & & \\ & & \ddots & & \\ & & & I_{2q-s_i-1} & T_{2q-s_i-1}^{(i)} p_{2q-s_i-1}^{(i)} \\ & & & I_{2q-1} & T_{2q-1}^{(i)} p_{2q-1}^{(i)} \end{bmatrix} \tag{10}$$

After each round of interpolative decompositions, the column skeletons are joined horizontally and then split vertically (12) resulting in the next round of blocks to be factorized. The way the blocks are indexed, the j-th skeleton is joined with the $(j+1)$-th and then given a new index depending on s_i and $x_k^{(i)}$ as seen in (11).

$$s_i = 2^{Q-i+1} \qquad x_k^{(i)} = 2^{Q-i} \lfloor \frac{k2^i}{q} \rfloor + k \tag{11}$$

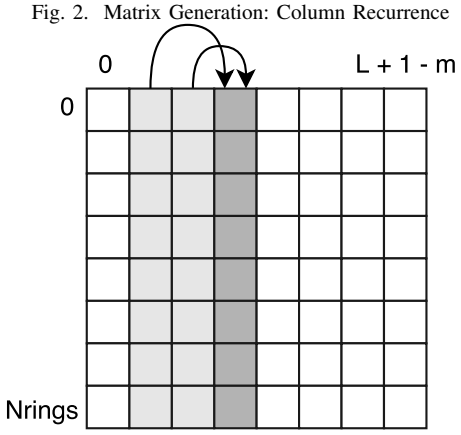
$$\begin{bmatrix} \hat{A}_{2k}^{(i-1)} & \hat{A}_{2k+1}^{(i-1)} \end{bmatrix} = \begin{bmatrix} A^{(i)}_{x_k^{(i)}} \\ A^{(i)}_{x_k^{(i)}+s_i} \end{bmatrix} \quad \forall k \in \{0, 1, ..., q-1\} \tag{12}$$

Finally, the permutation matrices $P^{(i)}$ are applied to their corresponding $A^{(i)}$ matrix to get $S^{(i)}$ and each sub-block's location within $S^{(i)}$ is recorded for use in multiplication. For example, if Q = 2, then the compressed matrices block structure will be as seen in Figure 1.

III. Method

In order to explore the GPU parallelism available in this algorithm, we developed a C library that will be referred to as MonarchSHT in this paper. The details of this implementation can be seen below.

Fig. 2. Matrix Generation: Column Recurrence



A. GPU Parallelism

Since transferring data to and from the GPU can be a very time-consuming process, these transfers are to be avoided whenever possible. For this implementation of the spherical harmonic transform, the goal is to then only transfer the input coefficients to the device, and then the output function values back to the host machine. This means that the generation and compression of the matrices, and the matrix-vector products, all happen within a single device's memory. In the distributed setting, an extra step of moving the data from each device to the root node is required but it is negligible in comparison to the complexity of the rest of the algorithm.

1) Matrix Generation: Since each matrix Λ_m is of size $(N_{\text{rings}} \times L + 1 - m)$, a single buffer of size $(N_{\text{rings}} \times L + 1)$ can be used to store each uncompressed Λ_m until it is either multiplied against the input coefficients or compressed using the butterfly factorization then written to file.

The first column of the first matrix Λ_0 is simply the normalizing constant $\sqrt{\frac{1}{4\pi}}$. Then each thread in a block of the GPU handles a single row and computes the values for each column using the associated Legendre recurrence relation. Then for the next matrix generation, the first column contains P_{m-1}^{m-1} which is used to generate P_m^m and then the process repeats.

Using this approach, each matrix can be computed using a single kernel launch which helps reduce overhead. Using Magma or cuBLAS to update each column as a daxpy operation needs many small kernel launches and ends up incurring a large amount of kernel launch overhead.

2) Matrix Compression: In order to compress a given Λ_m, the matrix is partitioned into blocks based on the input Q and then each block is factored using interpolative decomposition. Then the interpolation matrices T_j^i and permutation vectors p_j^i are written to a file for later use. The remaining column skeletons are then joined together horizontally and split vertically to form the next round of blocks to factor. The final round of column skeletons are simply discarded since they consist only of columns of the original matrix which can be generated as needed.

Once each block has been processed by column pivoted QR, the resulting triangular R matrix must be observed to approximate the rank of the block. For a given matrix with rank k, in exact arithmetic, R would be of the form seen in (13). However, in finite precision, the bottom right block will not be exactly the zero matrix. The approximate rank of a given block is the value of k such that the top-left block is k by k and the bottom-right block has Frobenius norm less than some tolerance. By using a binary search, this rank can be approximated in a relatively small amount of time to arbitrary precision.

$$R = \begin{bmatrix} R_{11} & R_{12} \\ 0 & 0 \end{bmatrix} \tag{13}$$

For compression that is as close to lossless as possible, this rank needs to be computed very accurately. However, if a lossy compression is acceptable, the tolerance can be reduced, resulting in an approximate rank less than the true rank. For the experiments in this paper, the compression is computed to be as lossless as possible. Wavemoth's precomputation also has a tolerance parameter that determines how lossy the compression is and is set to be as lossless as possible.

Since each block requires some data movement followed by column pivoted QR and then a triangular system solve, the goal is to interleave the kernels so that data for one block is being moved while another block is being processed using dgeqp3 or dtrsm. Unfortunately, Magma's column pivoted QR (dgeqp3) doesn't naturally have support for multiple streams and required modification. Specifically, unlike other Magma routines, dgeqp3 does not take a CUDA stream and operates on the default stream. Magma's column pivoted QR launches a large number of small single operation kernels that depend on the results from previous kernels. This means that even with our modification, very little overall concurrency is achieved for the matrix compression and is apparent in the resulting performance.

3) Matrix-Vector Multiplication: For the Λ_m that are not compressed, applying the matrix is simply a matter of calling Magma's dgemv routine. For the matrices that are compressed, the process is more complex. Recall that the butterfly algorithm results in a factorization of the form (14).

$$A = A^Q S^{Q-1} ... S^0 \tag{14}$$

This compressed matrix-vector multiplication can be broken up into a number of operations. Applying S^i (15) to a vector requires applying each permutation p^i_j and then T^i_j for each of its blocks. The final block matrix A^Q does not have any permutations encoded in the blocks and can be applied simply with dgemv.

$$y_j = S^i_j x_j = \begin{bmatrix} I & T^i_j \end{bmatrix} \begin{bmatrix} x^u_j \\ x^l_j \end{bmatrix} = x^u_j + T^i_j x^l_j \tag{15}$$

The data that is stored during the precomputation has to be read in from file and then moved to the GPU memory.

Fig. 3. Example of the GPU concurrency. Each uncompressed operation needs to wait for file reads on the CPU, the GPU is kept busy by applying the matrices that don't need CPU file reads. (Not to scale)

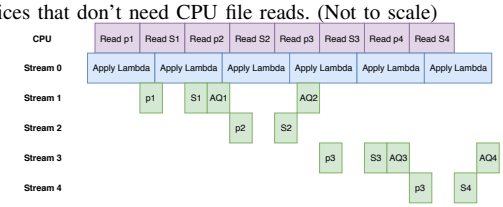


Fig. 4. Given 14 matrices and 4 processes, ownership would be distributed as follows

Process	Responsible for
0	0,4,8,12
1	1,5,9,13
2	2,6,10
3	3,7,11

Compared to a single dgemv, this is a very expensive process and care needs to be taken to "hide" these transfer times. One way to do this is to launch some tunable amount of kernels to apply the uncompressed matrices while the compressed data is being read. Once the data for the current compressed matrix is read, the kernels for applying S^i and A^Q can be launched and the process is repeated.

By utilizing pinned memory and asynchronous data transfers, the GPU can stay busy while the CPU is performing the lengthy file reads. This method of applying the uncompressed matrices while the compressed matrix data is being read will be referred to as the "hybrid" method. An example of what each stream is doing can be seen in Fig.3.

B. MPI Parallelism

For a fixed m, the generation, compression, and multiplication of the matrix Λ_m is entirely independent of any other m. This means that for all $L+1$ matrices that need to be handled, each can be processed entirely independently in a distributed setting. Each process is assigned a range of m values that they are responsible for in a round-robin fashion to ensure that each process is responsible for roughly equal amounts of work. A small example of m ownership per process is illustrated in Fig.4.

Once the Legendre Transform portion of the algorithm is complete, the results of the matrix-vector multiplications are all gathered to the root process. This data is then moved back onto the GPU at root for a final FFT to complete the spherical harmonic transform. Typically, moving data to or from a GPU or between MPI nodes is an expensive step. However, the amount of time required to compress, generate, and apply the matrices dominates this single round of communication and FFT as the problem size scales.

IV. Results

In order to measure the performance of our proposed implementation, we ran our code on the University of Utah Center for High Performance Computing's Kingspeak cluster

Fig. 5. Run-time comparison of libpsht, Wavemoth, and MonarchSHT's direct and hybrid method. Wavemoth and MonarchSHT's compression is done in as lossless a manner as possible. Further study is needed to understand what is happening at L = 2048 to cause the GPU methods to perform nearly the same as Wavemoth.

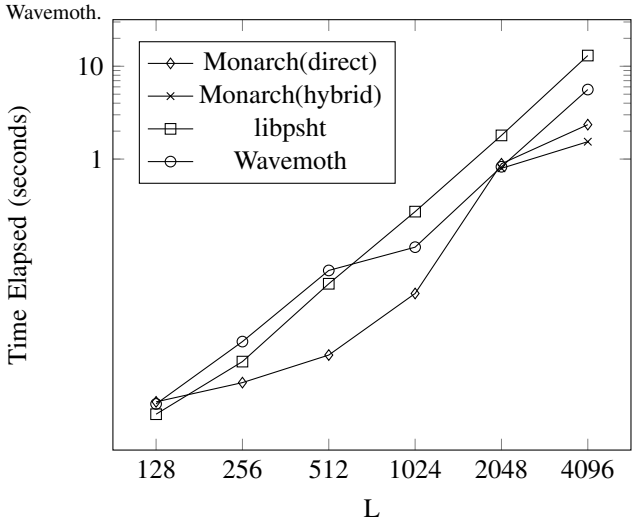
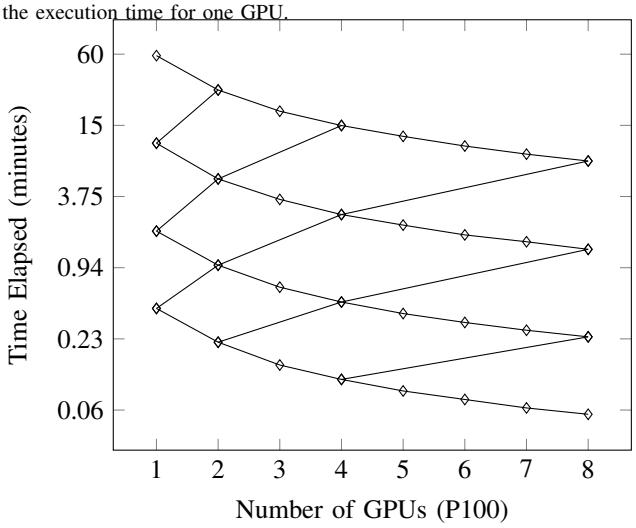


Fig. 6. MPI strong (solid lines) and weak (dashed lines) scaling for L = (1024,2048,4096,8192), T = (100,200,400,800). The work divides very nicely between many processes and the execution time is very nearly T/p where T is the execution time for one GPU.



using the available GPU nodes. Each node in the cluster is equipped with two Nvidia P100 graphics cards and 28 CPU cores (14 per socket). The following experiments were then run using various configurations of GPUs and CPU cores as detailed in each section.

A. GPU vs CPU

In order to get a baseline for the performance of our code, we first tested how the direct, uncompressed, GPU-accelerated transformation performs against the sequential CPU standard library libpsht. For this comparison, our GPU code is running with a single P100 GPU and the CPU libpsht code is running using every available CPU core.

Then the experiment was rerun using the hybrid approach using the same configuration. Before L = 2048, there simply isn't enough uncompressed work to be done to hide the file reads and thus the hybrid method doesn't start to pull ahead of the direct method until L = 2048 and L = 4096.

The experiment is run again for each problem size using Wavemoth's compressed algorithm and all of the results are put together in Fig.5. The hybrid method is only plotted for the problem sizes in which it beats the direct method since in the other cases, the direct method would be used.

B. MPI Scaling

In order to measure the scaling of the algorithm (compressed and uncompressed) with respect to the number of GPUs, we ran the code with $L = 8096$ and $T = 1000$ where T is the number of matrices to compress before switching to the uncompressed algorithm. The GPU cluster on Kingspeak then has 8 total P100 GPUs available and the code was run with 1,2,...,8 GPUs giving the results that can be seen in Fig.6.

Since each matrix-vector multiplication is independent from any other and dominates the total complexity of the algorithm, the scaling seen in Fig.6 is to be expected. The total run time is divided nearly evenly among the total available GPUs.

V. Conclusions

Using this method, if a large number of spherical harmonic transforms need to be computed using a single discretization, the compressed Λ matrices can be precomputed to reduce the total amount of work to compute each transform. This pre-computation is expensive and can require a very large amount of space to store but if enough transforms are computed, this cost can be offset by the improved performance overall. Specifically, let t_G be the time to generate all of the matrices, t_M be the time it takes to apply all of the matrices in the uncompressed manner, t_B be the time it takes to apply of the matrices after butterfly compression, and t_C be the time it takes to compress all of the matrices, then the two methods take the same amount of total time when the total number of transform to be computed is equal to n in (16).

$$n = \frac{t_C}{t_M - t_B} \tag{16}$$

This is a somewhat conservative estimate since the compressed matrices can be stored for later but the uncompressed matrices are typically not. Then if we let k be the number of times the uncompressed matrices are re-generated over the course of n total transforms, the condition for equivalent performance becomes (17).

$$n = \frac{t_C - (k-1)t_G}{t_M - t_B} \tag{17}$$

The unfortunate downside to this method of compressing and storing the matrices ahead of time is that by $L = 4096$, the amount of storage space required starts to become unmanageable. For $L = 4096$, the required space to store the data is around 50 or 60 gigabytes. By $L = 8192$, the amount

of space required would be measured in tens or hundreds of terabytes. For very large transforms, in a distributed computing setting, each node could potentially store only the precomputed data for the range of m for which the node is responsible to alleviate these storage requirements.

We are then left with the question of whether or not the GPU-accelerated version of the butterfly compression algorithm actually improves anything over the Wavemoth method. Even without the butterfly compression, utilizing GPU parallelism for computing the matrix-vector products drastically reduces the total amount of time required. For the compressed matrices, even further performance improvements can be obtained through concurrent multiplication of the sub-blocks.

As can be seen in the previous results section, even without compression, our direct method performs faster than Wavemoth's compressed algorithm. With compression, for large enough L, even further gains are obtained.

A. Future Work

The major bottleneck in the GPU compression algorithm seems to be in the column pivoted QR stage of the interpolative decomposition. Magma's dgeqp3 routine seems to require a lot of synchronization and there does not seem to be any gains from running dgeqp3 for each block concurrently. Using Magma for the ID step gives results comparable to the existing non-GPU Wavemoth compression but should be able to be improved further. [10]

Once the GPU compression scheme is improved, application specific features should be included into the method such as spin-weighting and support for complex coefficients and function values. Since the butterfly compression algorithm's usefulness hinges on the number of transforms performed for a specific discretization, an application that requires a large number of transforms would be ideal to illustrate its effectiveness.

Both the direct and hybrid MonarchSHT methods are highly unoptimized and yet still see performance improvements over CPU methods due to GPU parallelism. A next step in this project would be to add a number of optimizations to further improve the performance. For instance, the direct method generates the matrices Λ_m with a kernel and then uses Magma's dgemv to apply it in a separate kernel. These two operations can be rolled into a single optimized kernel that will likely see performance increases. Since the hybrid method uses the matrix generation kernel to construct the residual matrices, an optimized, combined kernel will result in gains for this method as well.

VI. Acknowledgements

This work was supported in part by the National Science Foundation grants CCF-1643056 and CCF-1704715. I'd also like to thank the University of Utah School of Computing and my adviser Hari Sundar for helping make this work possible.

References

[1] H. Dette, V. B. Melas, and A. Pepelyshev. Optimal designs for three-dimensional shape analysis with spherical harmonic descriptors. *ArXiv Mathematics e-prints*, March 2006.

[2] Martin J Mohlenkamp. A fast transform for spherical harmonics. *Journal of Fourier analysis and applications*, 5(2):159–184, 1999.

[3] I. O. Hupca, J. Falcou, L. Grigori, and R. Stompor. Spherical harmonic transform with GPUs. *ArXiv e-prints*, October 2010.

[4] DS Seljebotn. Wavemoth-fast spherical harmonic transforms by butterfly matrix compression. *The Astrophysical Journal Supplement Series*, 199(1):5, 2012.

[5] Mark Tygert. Fast algorithms for spherical harmonic expansions, ii. *Journal of Computational Physics*, 227(8):4260–4279, 2008.

[6] Michael O'Neil, Franco Woolfe, and Vladimir Rokhlin. An algorithm for the rapid evaluation of special function transforms. *Applied and Computational Harmonic Analysis*, 28(2):203–226, 2010.

[7] Nathanaël Schaeffer. Efficient spherical harmonic transforms aimed at pseudospectral numerical simulations. *Geochemistry, Geophysics, Geosystems*, 14(3):751–758, 2013.

[8] Yingzhou Li, Haizhao Yang, Eileen R Martin, Kenneth L Ho, and Lexing Ying. Butterfly factorization. *Multiscale Modeling & Simulation*, 13(2):714–732, 2015.

[9] Mark Tygert. Fast algorithms for spherical harmonic expansions, III. *CoRR*, abs/0910.5435, 2009.

[10] Andrés Tomás, Zhaojun Bai, and Vicente Hernández. Parallelization of the qr decomposition with column pivoting using column cyclic distribution on multicore and gpu processors. In Michel Daydé, Osni Marques, and Kengo Nakajima, editors, *High Performance Computing for Computational Science - VECPAR 2012*, pages 50–58, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.