Exact location 5D interpolation on GPUs and a look into processing kernels

Kai Zhuang, and Daniel Trad

ABSTRACT

We look at the computation of the naive DFT interpolation on GPUs in an attempt to solve two main issues commonly seen in the application of 5D interpolation, compute times and far offset approximations. Our results show that the implementation on GPUs help solve both problems, although the computation of the naive DFT is significantly more expensive than the FFT, the switch to GPU computation for the dataflow decreases the runtime compared to standard CPU implementation of DFT interpolation.

INTRODUCTION

Generally, 5D interpolation is very compute intensive due to the sheer data volume sizes seen in today's seismic acquisitions size in multiple terabytes of data. As 5D interpolation must be performed on the entire dataset, it represents a significant amount of computational time spent on processing a dataset. Often, days of server time are spent interpolating data to improve sampling information and data uniformity. With our application of the 5D workflow into GPUs, we can attempt to significantly reduce the runtime due to the massively parallel nature of GPU processors. As a simple anecdote, GPU processing is more akin to parallel processing whereas CPU processing is more in line with 'multi-serial processing'. This computational advantage afforded to GPUs allows us to leverage some computationally more expensive operators to perform our interpolation at a higher accuracy. Mainly the Fourier transform algorithm that is used for standard 5D interpolation uses evenly sampled bins so that the data can be used in an FFT algorithm. This use of bins causes approximations in the 5D algorithm that can result in inaccuracies in the interpolated result. In our approach, we remove the binning part of the workflow in favor of using a more expensive naive DFT operator instead of the standard FFT operator.

CUDA

CPUs are generally not built to specialize in the parallel processing applications we scientists use them for every day. These applications mainly consist of highly repetitive matrix operations for multi-dimensional data. CPUs are generally built for executing multiple independent tasks at the same time, where each task operates independently from another and are barely related, thus CPU threads are generally considered independent. This independence comes at a sacrifice to repeatedly do the same workload, which GPUs accel at. For comparison, CPUs generally consist of 2 threads per core where the threads of each core share some processing resources. GPUs on the other hand consist of streaming multiprocessors (SMs) which hold about 2048 threads each(with shared resources), divided into 64 thread warps per SM. Although these cores are weaker/slower than CPU cores the massive thread count of these SMs more than makes up the difference in full parallel compute scenarios.

DATA TRANSFER AND PLACEMENT

The memory management for a heterogeneous computing system is very different from a native CPU system. The CPU and GPU are separate and a high-speed PCI Express bus connects the two. This results in two distinct, physically linked memory spaces, one on the GPU and one on the CPU. In these heterogeneous systems the CPU and GPU cannot directly address each other's memory and explicit copying is required for cross-access. Although implementations are in place for programmers to address CPU and GPU memory as a single entity (CUDA unified memory), explicit copying still occurs in the background (forces transfers every time it's addressed on a different device). When poorly managed, either explicitly or implicitly, a significant amount of overhead is added to the workflow of the algorithm resulting in poor runtimes. In the worst case, these runtime overheads can push GPU algorithms to become slower than their CPU counterparts. GPU memory is generally separated into three tiers, global memory, shared memory, and constant memory. Global memory is comparable to the CPU's ram, it consists of the largest memory space on the GPU and the GPU can read directly from it. All data transfers between CPU and GPU also happen in global memory. Shared memory on a GPU is equivalent to the L3 cache on a CPU, where each SM possesses a small amount of shared memory that can stream at extremely high speed to the GPU core. On a CPU the L3 cache is automatically managed by the system which stores what it thinks are data that are frequently accessed. The fact that the L3 cache is automated makes CPU algorithms less predictable as direct control over what is being cached is not available and cache misses can be a common occurrence. On the other hand, the need to explicitly program shared memory is another complication that a programmer must face when working on GPU algorithms. The last type of memory unique to GPUs is known as constant memory. Constant memory exists as very small on the DRAM and is buffered on the GPU. Constant memory stores constant values, these values cannot be modified during GPU kernel execution hence the constant name. The advantage of constant memory is that all threads can access the same value at the same time without any overhead even though it only possesses a single read port. This is to say that constant memory optimization for broadcasting information, where simultaneous access is significantly optimized. The main limitation for constant memory is that if access to values is not simultaneous across threads then access is reduced to serial when being read from, due to the single port. Another memory management topic to take into consideration for GPUs is memory coalescence. Memory coalescence for GPUs works in the opposite way that one would expect for CPUs due to the parallel nature of GPUs. For a CPU, memory is best accessed sequentially to improve read speed to avoid the need to seek different parts of the data. For a GPU, memory is best laid out for sequential parallel loading of the memory where memory is spread across computational threads doing the work. Data placement is a very important factor when it comes to proper workflow performance on GPUs, especially if the throughput is memory bound.

NON-UNIFORM DISCRETE FOURIER TRANSFORM

Of the main exploration seismic processing methods in use today, the 5D interpolation algorithm (Trad, 2009) is one of the most ubiquitously adopted in recent years. 5D interpolation falls under the umbrella of techniques known as compressive sensing (Lin and Herrmann, 2009). The main assumption of compressive sensing and 5D interpolation is that

the seismic wavefield in a shot record is heavily oversampled. Through this assumption, 5D interpolation seeks to interpolate missing data from wavefield data recorded in adjacent and nearby receivers. 5D interpolation works by performing least squares Fourier interpolation in the 5 data dimensions (inline, crossline, offset, azimuth, and time) frequency slice by frequency slice.

The Fast Fourier Transform (FFT) algorithm (Cooley and Tukey, 1965) reduces the time complexity of computing the DFT of a regularly sampled signal thus significantly reducing computational time for long-period signals. However, this increase in efficiency requires the data to be regularly sampled. There are several approaches to overcome this issue and allow for the FFT to operate on irregular data, for example, the non-uniform FFT or NUFFT (Dutt and Rokhlin, 1993). The majority of these approaches focus mainly on either upsampling and/or interpolating the data into an evenly sampled signal, then applying the FFT algorithm. Another common way to approach this problem is to use an iterative method to solve for the NUFFT as a forward-adjoint problem (Keiner et al., 2009). Many of these techniques to calculate the NUFFT have already been implemented on GPUs (Shih et al., 2021), with varying degrees of performance. All of these approaches come with drawbacks compared to the naive DFT approach. The resampling approach can cause unwanted increased memory use in the upsampling case or approximations (sinc, bilinear, etc...) in the interpolation case. The inversion case would result in higher computational time due to the need to solve the problem iteratively.

For our application into 5D interpolation, we will be solving the naive DFT matrix, which we shall refer to as NUDFT (Non-Uniform DFT) instead of the standard FFT. NUDFT allows for irregular spacial positioning which allows for more accurate results, especially in wide azimuth data, to increase the quality of the output image. However, as described in Trad (2016), for 5D interpolation via NUDFT a quadruple nested loop is required for the matrix calculations which will result in significant computational times due to increased time complexity over FFT. The naive NUDFT operator is a candidate for application on GPUs because of the massive thread advantage provided by the GPU architecture. The complexity of the application of NUDFT 5D interpolation on GPUs comes in the form of multiple steps for interpolation that are performed between each application of the Fourier transform. These multiple steps must then be optimized for application on the GPU which otherwise would cause a significant slowdown of the algorithm. The slowdown is caused by significant memory transfers between GPU and CPU as the data pipeline would become interleaved between the two. This is mainly due to the algorithm computing one frequency slice at a time, then using the previous frequency slice to aid in the convergence of other frequencies causing the interleaving of operators.

The non-uniform discrete Fourier transform (NUDFT) is important in the field of signal processing as in many applications the signal recorded does not exist in regular intervals across some dimensions. In many applications of the NUDFT, the signal is interpolated into regular sampling and then run through an FFT algorithm. For our applications, because we are seeking to interpolate with NUDFT, resampling/interpolating to regular sampling is in contention with our objective. The standard DFT equation can be seen in equation 1.

$$H(\omega) = \sum_{k=0}^{N-1} h_k e^{-i\omega k \Delta t} \tag{1}$$

Where $H(\omega)$ is the Fourier coefficient, h_k is the signal at the k^{th} sample ω is the wavenumber and N is the number of frequencies. For the purposes of signal processing, the fast Fourier transform (FFT) is generally used, this transform takes advantage of the regularity due to its periodicity, reducing computational complexity to $O(nlog(n))$. This is different from the direct evaluation of the discrete Fourier transform with complex $O(n^2)$ which NUDFT shares. In general, the calculation for the NUDFT becomes:

$$H(\omega) = \sum_{x_1=X_1min}^{X_1max} \sum_{x_2=X_2min}^{X_2max} \sum_{x_3=X_3min}^{X_3max} \sum_{x_4=X_4min}^{X_4max} h_k e^{-i\omega k \Delta t} \tag{2}$$

Processing using NUDFT is also costly in that because the samples are not evenly spaced NUDFT takes more time to calculate due to the extra calculations needed to set up the unevenly spaced matrix. This problem is similar to the computer science problem of sparse matrix-vector multiplication where sparse matrices must account for the non-uniform way in which data is stored. Implementing 5D interpolation using a NUDFT operator then bypasses the need to bin traces for a uniform grid, however, this adds a significant processing requirement by significantly increasing the computational complexity of the problem from $O(nlog(n))$ to $O(n^2)$.

NUDFT KERNEL APPROACHES

As we are applying a direct assessment of the Fourier transform, the main operation for the transform being applied is a modified matrix-vector multiplication (mvm). The matrix represents the mapping from the spatial plane to the wave-number domain and the vector represents each frequency slice of the data after FFT is applied in time to the dataset. For the multiplication algorithm, we compute the single precision matrix-vector multiplication. For our DFT implementation filters must be applied in our mvm that are not present in the example. We make a comparison of general CPU and GPU approaches to mvm as well as standard library packages that are commonly used.

Figure 1 shows the performance results of CPU and GPU kernels for mvm. Something interesting to note is that at very small matrix sizes the CPU implementations lead the runtimes, while the GPU code is seen to be slower. This speed difference at small matrices can be attributed to the lack of a full population of threads on the GPU. In general, in these cases of small calculations, the sheer speed difference in processor threads pushes the advantage toward CPU threads. As the GPU threads populate with data, it can be seen that the GPU kernels start to take the lead while the CPU begins to lag significantly as the graph diverges. Within the CPU results, The consistently worst result is the generic BLAS implementation. This low performance can be mainly attributed to the lack of parallel support present in the library forcing the BLAS to work single-threaded. The second slowest implementation is the mvm operator we wrote, which uses openMP for parallelization.

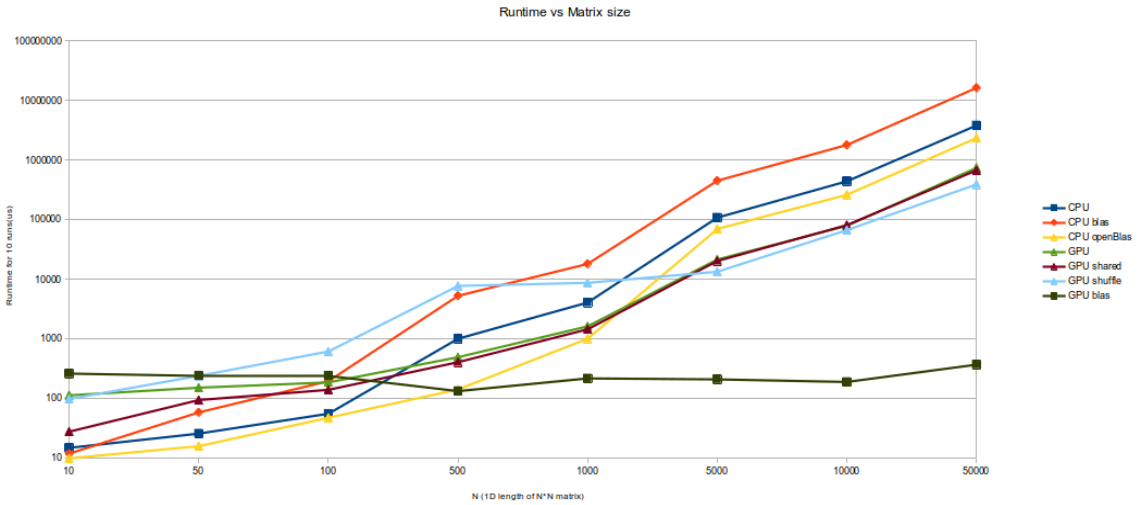

FIG. 1. Compairson of GPU and CPU runtimes for matrix-vector multiplication, CPU is an intel 8 core 8 thread 9000 series CPU, GPU is an Nvidia Geforce RTX 3060.

Algorithm 1 matrix vector multiplication pseudo code

```
function MVM
    #PRAGMA OMP PARALLEL FOR          ▷ insert for parallelization of loop
    for im = size m do
        for in = size n do
            tmp[im] += matrix[im][in] * vec[in]
        end for
    end for
end function
```

Of the CPU results the openBLAS implementation performs the best at all data sizes, this is mainly attributed to the optimization of the kernel at the assembly level as well better parallelization control. The GPU kernels show interesting changes as matrix size increases. The simplest implementation labeled simply as GPU in figure 1 is a direct port of the basic CPU implementation to the GPU which we will use as a baseline in our comparison. The second implementation which utilizes shared memory operates significantly faster than the baseline approach at small matrix sizes, then converges to a similar runtime as size increases. The shared memory approach separates the work into high-speed blocks of memory which are read from rather than directly from global memory on the GPU. The shared memory pseudo-code can be seen in **Algorithm 2**.

Algorithm 2 gpu shared MVM pseudo code based on Nvidia guide (Cook, 2022)

```
__global__
function MVM
    blockRow = blockIdx.y;
    blockCol = blockIdx.x;
    Csub = GetSubMatrix(C, blockRow, blockCol);
    row = threadIdx.y;
    col = threadIdx.x;
    for m < (A.width/BLOCK_SIZE); do
        Asub = GetSubMatrix(A, blockRow, m);
        Bsub = GetSubMatrix(B, m, blockCol);
        __shared__As[BLOCK_SIZE][BLOCK_SIZE];
        __shared__Bs[BLOCK_SIZE][BLOCK_SIZE];
        As[row][col] = GetElement(Asub, row, col);
        Bs[row][col] = GetElement(Bsub, row, col);
        sync
        for e < BLOCK_SIZE do
            Cvalue+ = As[row][e] * Bs[e][col];
        end for
    end for
    SetElement(Csub, row, col, Cvalue);
end function
```

The speed difference early can most likely be attributed to the speed benefits from the use of shared memory. Where at the larger matrix sizes, the speed is once again limited to the global memory streaming speed as the small size of the shared memory becomes less significant. The third GPU approach we took into consideration is the memory shuffle approach. In the memory shuffle approach rather than storing small blocks of reused data into shared memory, we do a shuffling of the memory using the primitive $\_\_shfl\_sync()$, to directly transfer data between registers rather than through the shared memory pipeline. The runtimes from the shuffle code are the oddest of the GPU codes as at small matrices it performs the worst of all the approaches, where around n=1000 to 5000 it becomes one of the best performant approaches. Generally, however, Nvidia's own CUBLAS implementation is the most performant GPU approach by far for large matrices as the runtimes for our benchmarks did not increase with increased matrix size. This flat line in computational time indicates that the resources were not fully saturated, as the matrix size increased the

saturation of the pipeline increased but as the pipeline was not full it could still do the calculation in the same amount of clock cycles. Although we would like to analyze how this performance was achieved, due to the closed-source nature of the CUBLAS library, analysis is not possible. Although the CPU kernels perform better at very small sizes, these small sizes are wholly unrealistic for most standard applications of mvm that are not ML applications.

LEAST SQUARES DEBLENDING

The main solver for 5D interpolation is the sparse least squares inversion. The solver works on each frequency slice, transforming the spatial domain into the wave-number domain using DFT. In this case the interpolator contains a sampling operator that represents the signature of the acquisition (Trad, 2016). Our objective is to remove this signature through lease squares inversion. Sparse inversion seeks to find the solution to problems that are expected to behave more closely to the least absolute value solution as opposed to the least-squares solution. The advantage of the ℓ_1-norm for the data space is that the inversion is less sensitive to sporadic events commonly associated with noise which is known as robust inversion. The ℓ_1-norm applied to the model space then pushes the model space to the solution with the most zero values favoring single large numbers instead of multiple small ones (sparse model). By adding data weights, we can then force the algorithm to converge towards the least absolute value solution (ℓ_1) or least squares (ℓ_2) solution. Also, by adding model weights (Trad et al., 2003), we can push the algorithm to converge towards either a sparse model (ℓ_1-model) or the smooth/least-squares model (ℓ_2-model). We can reformulate the cost function in equation **??** using weights to

$$J = ||\mathbf{W_r}(\mathbf{b} - \mathbf{\Gamma L m})||_\mathbf{2}^\mathbf{2} + \mu||\mathbf{W_m m}||_\mathbf{2}^\mathbf{2}. \tag{3}$$

Here $\mathbf{W_r}$ is the data weight operator, and $\mathbf{W_m}$ is the model weight operator. These weights allow the user to adjust the norm of the residual or model. These weight operators are usually diagonal matrices. These weights can be constructed as

$$\begin{aligned} \text{diag}(\mathbf{W_m}) &= |\mathbf{m}|^{(q-2)/2} \\ \text{diag}(\mathbf{W_r}) &= |\mathbf{r}|^{(p-2)/2} \\ \text{where } \mathbf{r} &= \mathbf{b} - \mathbf{\Gamma L m}, \end{aligned} \tag{4}$$

where p and q are generally either 1 or 2 for inverting based on either the ℓ_1 or ℓ_2 norm. As a way to avoid diving by zero, a threshold is chosen as a percentile of the data. Using $q = 1$ we can solve for the ℓ_1 solution while our algorithm solves a ℓ_2 equation. For example, to calculate the ℓ_1 norm for the model, we can use the model weight below

$$||\mathbf{W_m m}||_\mathbf{2}^\mathbf{2} = \mathbf{m^T W_m^T W_m m} = \mathbf{m^T}(\frac{\mathbf{1}}{\sqrt{\mathbf{m}}})^\mathbf{T}(\frac{\mathbf{1}}{\sqrt{\mathbf{m}}})\mathbf{m} = ||\mathbf{m}||_\mathbf{1}^\mathbf{1}. \tag{5}$$

This can also be applied to the data weight $\mathbf{Wr}$ to implement robust inversion.

RESULTS

A nuance we noticed when comparing CPU and GPU results is that although the results are the same there is a micro-rounding difference between GPU and CPU results that are

not seen visually but show up in a random pattern akin to random Gaussian noise. This difference is not seen between CPU runs or GPU vs GPU results. We think this difference is mainly attributed to a difference in the handling of float rounding causing differences between CPU and GPU outputs.

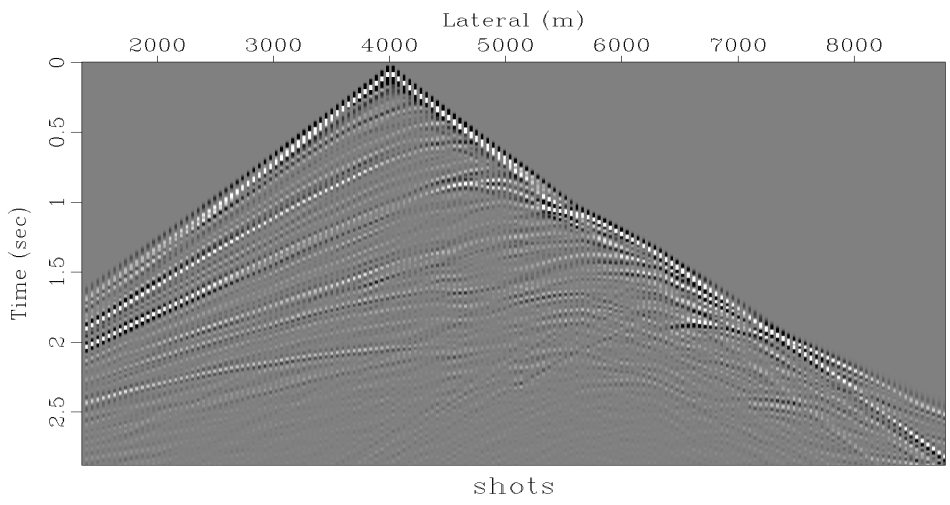

FIG. 2. Decimated shot.

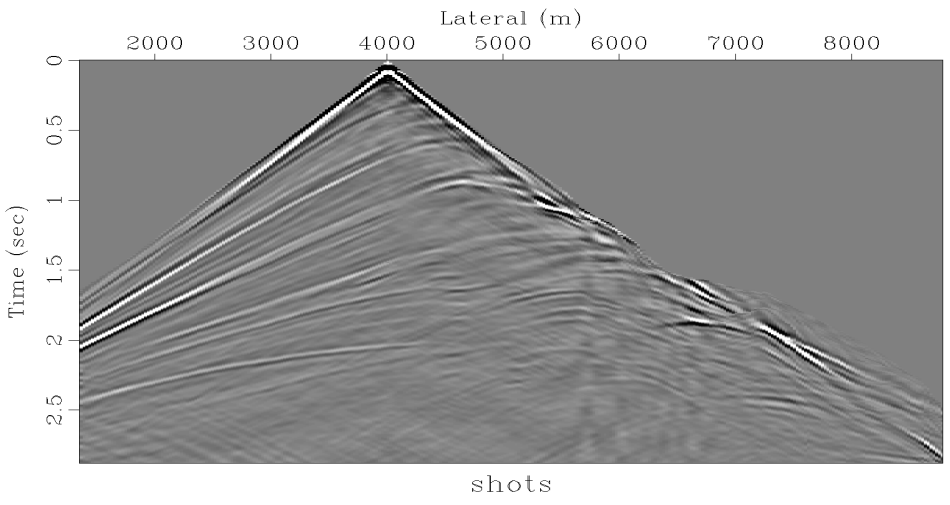

FIG. 3. Shot reconstruction using DFT interpolation.

The general 2D interpolation has been implemented, the reconstruction can be seen in figure **??** and the decimation in figure 2. The decimated shot gather has two traces between every live trace removed for the interpolation. it can be seen that although the interpolation was effective, the quality of the output is lacking, this is mainly due to the 2D nature of the example as the operator lacks the extra information afforded to the other dimensions. Work is still ongoing for the 5D part of the study.

CONCLUSION

The use of GPUs for the application of matrix-vector multiplication required for the application of naive DFT solves two issues present with 5D interpolation. The computation time and far offset issues of 5D interpolation are both solved through the implementation of GPUs. Due to binning of traces to transform the data into evenly spaced grids, far offset data is lost, by implementation a naive DFT approach rather than an FFT one exact trace locations can be used. The runtime issues due to the use of naive DFT are solved with the move to GPU processing as they can perform matrix operations significantly faster than CPUs. This speedup allows for the implementation of Fourier transforms without the need for spatial binning at very low runtimes.

ACKNOWLEDGMENTS

We thank the sponsors of CREWES for their continued support. This work was funded by CREWES industrial sponsors and NSERC (Natural Science and Engineering Research Council of Canada) through the grant CRDPJ 543578-19.

REFERENCES

Cook, S., 2022, Cuda c++ programming guide.
URL https://docs.nvidia.com/cuda/cuda-c-programming-guide/

Cooley, J. W., and Tukey, J. W., 1965, An algorithm for the machine calculation of complex fourier series: Mathematics of Computation, **19**, 297–301.

Dutt, A., and Rokhlin, V., 1993, Fast fourier transforms for nonequispaced data: SIAM Journal on Scientific Computing, **14**, No. 6, 1368–26, copyright - Copyright] © 1993 Society for Industrial and Applied Mathematics; Last updated - 2021-09-11.

Keiner, J., Kunis, S., and Potts, D., 2009, Using nfft 3—a software library for various nonequispaced fast fourier transforms: ACM Trans. Math. Softw., **36**, No. 4.
URL https://doi.org/10.1145/1555386.1555388

Lin, T., and Herrmann, F., 2009, Designing simultaneous acquisitions with compressive sensing, cp–127–00,269.

Shih, Y.-h., Wright, G., Andén, J., Blaschke, J., and Barnett, A. H., 2021, cufinufft: a load-balanced gpu library for general-purpose nonuniform ffts.
URL https://arxiv.org/abs/2102.08463

Trad, D., 2016, Five-dimensional interpolation: exploring different fourier operators: CREWES Annual Research Report, **28**.

Trad, D., Ulrych, T., and Sacchi, M., 2003, Latest views of the sparse radon transform: GEOPHYSICS, **68**, No. 1, 386–399.

Trad, D. O., 2009, Five-dimensional interpolation: Recovering from acquisition constraints: GEOPHYSICS, **74**, No. 6, V123–V132.
URL https://doi.org/10.1190/1.3245216