5D interpolation on GPUs using the non-uniform Discrete Fourier Transform

Kai Zhuang, and Daniel Trad

ABSTRACT

We implement 5D interpolation with the Non-Uniform exact location DFT operator on GPUs to address binning-related errors in far offset approximations without compromising computational efficiency. This approach aims to accelerate the relatively slow Non-Uniform DFT operator using GPU computation. Our findings indicate that the GPU implementation effectively resolves both issues. Although the computation of the Non-Uniform DFT is more expensive than the FFT, transitioning to GPU computation significantly reduces runtime compared to the standard CPU implementation of DFT interpolation.

INTRODUCTION

Among the prevalent exploration seismic processing methods employed today, the 5D interpolation algorithm stands out as one of the most widely embraced techniques in recent years. This algorithm is a subset of approaches falling under the overarching category of compressive sensing. Compressive sensing, also referred to as compressive sampling or sparse sampling, is a signal processing methodology that allows for signal acquisition at a substantially reduced rate compared to what the Nyquist-Shannon sampling theorem would recommend. In the context of 5D interpolation, the technique capitalizes on the fact that the seismic wavefield in a shot record is often oversampled. It leverages the idea that data can be extracted from neighboring traces to accurately reconstruct missing traces, provided that the wavefield is adequately, sparsely and randomly sampled. This assumption enables 5D interpolation to effectively interpolate absent data by utilizing information from the wavefield recorded in adjacent and nearby receivers.

Interpolating in 5D presents a formidable computational challenge due to the large data volumes in contemporary seismic acquisitions, often reaching multiple terabytes. This process involves reading and working with every point in the dataset multiple times, consuming a significant amount of computational time. It is not uncommon for days of server time to be dedicated to 5D interpolation to enhance sampling information and ensure data uniformity.

By implementing a 5D interpolation workflow on GPUs, our goal is to bring the runtime of the Discrete Fourier Transform (DFT) on the GPU to a level that can match the speed of the Fast Fourier Transform (FFT) on a CPU. The inherent parallel nature of GPU processors, in contrast to the more serial processing nature of CPUs, provides an opportunity to make the DFT computation competitive with the Fast Fourier Transform (FFT) on a CPU.

It's worth noting that on the CPU, the Non-Uniform Discrete Fourier Transform (DFT) we utilize is significantly slower than the optimized FFT. This comes with the drawback of FFT, which lies in its requirement for even spacing, leading to the coordinate binning step that introduces approximations and potential inaccuracies in the interpolated results. We

seek to remedy this with the use of the non-uniform DFT on GPUs.

FOURIER TRANSFORM OPERATOR SELECTION

The Fast Fourier Transform (FFT) algorithm, introduced by Cooley and Tukey in 1965 (Cooley and Tukey, 1965), significantly reduces the time complexity associated with computing the Discrete Fourier Transform (DFT) of regularly sampled signals. This enhancement is particularly beneficial for processing long-period signals, leading to a substantial decrease in computational time. The time complexity of the DFT is well-known to be $O(N^2)$, where N represents the number of data points.

In contrast, the FFT algorithm achieves a time complexity of $O(N \log N)$ through recursive decomposition and recombination of the DFT. However, it is crucial to recognize that this increased efficiency comes with the requirement that the data must be regularly sampled.

Various strategies have been developed to address this constraint and enable the FFT to operate on irregularly sampled data. One such approach is the non-uniform FFT (NUFFT), proposed by Dutt and Rokhlin in 1993 (Dutt and Rokhlin, 1993). Many of these methods primarily focus on upsampling and/or interpolating the data to transform it into an evenly sampled signal before applying the FFT algorithm.

An alternative method for tackling this issue involves using an iterative approach to solve the NUFFT as a forward-adjoint problem, as discussed by Keiner et al. in 2009 (Keiner et al., 2009). Techniques for computing the NUFFT have been implemented on GPUs, showcasing varying performance levels, as demonstrated by Shih et al. in 2021 (Shih et al., 2021). However, these approaches have drawbacks compared to the straightforward DFT method. For instance, the resampling approach may increase memory usage or introduce approximations (such as sinc, bilinear, etc.) in the interpolation case. In the inversion case, the higher computational time is incurred due to the need for iterative calculations.

For our application in 5D interpolation, we opt to solve the Non-Uniform DFT matrix, referred to as NUDFT (Non-Uniform DFT), instead of the standard FFT. NUDFT accommodates irregular spatial positioning, allowing for more accurate results, especially in wide azimuth data, thereby improving the output image quality. However, as described in Trad (2016), 5D interpolation via NUDFT requires a quadruple nested loop for matrix calculations, resulting in significant computational times due to increased time complexity over FFT. The NUDFT operator is a candidate for GPU application due to the substantial thread advantage provided by GPU architecture. The complexity of applying NUDFT 5D interpolation on GPUs arises from multiple interpolation steps performed between each application of the Fourier transform. These steps must be optimized for GPU applications to prevent a significant algorithm slowdown caused by substantial memory transfers between GPU and CPU. This issue arises because the algorithm computes one frequency slice at a time, using the previous frequency slice to aid in the convergence of other frequencies, leading to the interleaving of operators.

The non-uniform discrete Fourier transform (NUDFT) is crucial in signal processing,

especially when the signal recorded does not exist at regular intervals across certain dimensions. In many NUDFT applications, the signal is interpolated into regular sampling and then processed through an FFT algorithm. However, for our applications focusing on interpolation with NUDFT, resampling/interpolating to regular sampling conflicts with our objective. The standard DFT equation is presented 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 signal processing purposes, the fast Fourier transform (FFT) is generally preferred, taking advantage of regularity and periodicity to reduce computational complexity to $O(n \log n)$. This is in contrast to the direct evaluation of the discrete Fourier transform, with complex $O(n^2)$ complexity, which the 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 costly because, due to unevenly spaced samples, it takes more time to calculate the extra setup needed for the unevenly spaced matrix. This challenge is akin 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 bypasses the need to bin traces for a uniform grid, adding a significant processing requirement.

SPARSE SOLVER

The primary solver for 5D interpolation is the sparse least squares inversion. This solver operates on each frequency slice, transforming the spatial domain into the wave-number domain using the Discrete Fourier Transform (DFT). In this context, the interpolator contains a sampling operator representing the acquisition's signature (Trad, 2016). Our goal is to eliminate this signature through least squares inversion.

First, we begin with a standard forward modeling equation:

$$\mathbf{d} = \mathbf{L}\mathbf{m}. \tag{3}$$

Where, for 5D interpolation, L is a combined operator containing a sampling operator and the Fourier operator of choice. The sampling operator contains the information of the shot signature. This signature is what the solver is trying to remove through iterative inversion.

Sparse inversion aims to find solutions that behave more closely to the least absolute value solution than the least-squares solution. The advantage of using the ℓ_1-norm in the

data space is that the inversion becomes less sensitive to sporadic events commonly associated with noise, a characteristic known as robust inversion. Applying the ℓ_1-norm to the model space encourages a solution with fewer non-zero values, favoring single large numbers over multiple small ones (a sparse model).

By introducing data weights, the algorithm can be directed to converge towards either the least absolute value solution (ℓ_1) or the least squares solution (ℓ_2). Additionally, by incorporating model weights (Trad et al., 2003), the algorithm can be steered towards either a sparse model (ℓ_1-model) or a smooth/least-squares model (ℓ_2-model).

First, we start with a standard optimization problem:

$$\begin{aligned} &\text{Minimize } ||\mathbf{m}||_{\mathbf{p}}^{\mathbf{p}} \\ &\text{Subject to } ||\mathbf{d}-\mathbf{L}\mathbf{m}||_{\mathbf{q}}^{\mathbf{q}}. \end{aligned} \tag{4}$$

We can then express this with the general objective function:

$$\mathbf{J} = ||\mathbf{d}-\mathbf{L}\mathbf{m}|| + \mu||\mathbf{m}||, \tag{5}$$

By adding model and data weights, we can then dictate the algorithm to converge towards the least absolute value solution (ℓ_1) or least squares (ℓ_2) by reformulating the original least-squares equation through the minimization of the modified equation:

$$\begin{aligned} &\text{Minimize } ||\mathbf{W_m}\mathbf{m}||_{\mathbf{2}}^{\mathbf{2}} \\ &\text{Subject to } ||\mathbf{W_r}(\mathbf{d}-\mathbf{L}\mathbf{m})||_{\mathbf{2}}^{\mathbf{2}}. \end{aligned} \tag{6}$$

We can reformulate the cost function in Equation 5 using weights as follows:

$$J = ||\mathbf{W_r}(\mathbf{d}-\mathbf{L}\mathbf{m})||_{\mathbf{2}}^{\mathbf{2}} + \mu||\mathbf{W_m}\mathbf{m}||_{\mathbf{2}}^{\mathbf{2}}. \tag{7}$$

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

$$\begin{aligned} \text{diag}(\mathbf{W_m}) &= |\mathbf{m}|^{(q-2)/2} \\ \text{diag}(\mathbf{W_r}) &= |\mathbf{r}|^{(p-2)/2} \\ \text{where } \mathbf{r} &= \mathbf{d}-\mathbf{L}\mathbf{m}, \end{aligned} \tag{8}$$

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

$$||\mathbf{W_m}\mathbf{m}||_{\mathbf{2}}^{\mathbf{2}} = \mathbf{m^T}\mathbf{W_m^T}\mathbf{W_m}\mathbf{m} = \mathbf{m^T}(\frac{\mathbf{1}}{\sqrt{\mathbf{m}}})^{\mathbf{T}}(\frac{\mathbf{1}}{\sqrt{\mathbf{m}}})\mathbf{m} = ||\mathbf{m}||_{\mathbf{1}}^{\mathbf{1}}. \tag{9}$$

PARALLELIZATION OF DATAFLOW AND MEMORY MANAGEMENT

In general, CPU parallelization tends to perform better in coarse-grained applications, and this complexity increases when considering GPU applications. The preference for coarse-grained parallelization on CPUs is primarily influenced by the processor architecture, which is optimized for executing multiple independent tasks simultaneously. In this context, coarser parallelization is advantageous as CPUs do not anticipate heavy resource sharing. This stands in contrast to GPUs, designed for fine-grained applications where a significant number of resources are shared across threads. The conventional approach to parallelization for a 5D problem on a CPU involves adopting the coarsest-grained strategy, with each CPU operating on distinct trace grouped windows. However, this parallel window approach is unsuitable for GPUs due to architectural constraints, particularly the GPU's inefficiency in handling branching statements, such as if-else statements. In the GPU environment, a branching statement necessitates a device-wide wait of a significant portion of threads since only one branch can be executed at a time. This effectively diminishes a parallel process by a factor of the amount of branching statements in the GPU code.

In recent years, GPU memory sizes have experienced significant growth, alleviating the limitations imposed by the previously restrictive memory constraints. However, concerns persist regarding memory transfer bandwidth and latency across the PCI-Express bus when implementing code on the GPU. Additionally, these challenges exacerbate limitations on the applications of fine-grained parallelism on the GPU. Excessive back-and-forth transfers can lead to substantial slowdowns due to the limitations of the PCIe bus.

The incorporation of the GPU kernel into the broader 5D interpolation framework introduces an additional layer of complexity to the complete 5D implementation on GPUs. We observed the relative speedup between the GPU and CPU Discrete Fourier Transform (DFT) operator to be approximately $1 : 360$ in various tests for our implementation at a common trace length. However, the results following the integration into the overarching 5D code reveal a substantial reduction in speedup. Specifically, the speedup observed by switching just the kernel is approximately $1 : 3$. This outcome was deemed unsatisfactory, as this level of improvement could likely be achieved through better optimization of the CPU code.

The notable decline in speedup can be attributed to the issues mentioned earlier, primarily excessive memory transfers between the CPU and GPU during kernel execution. Due to the kernel's efficiency and the small effective size of the windows, each iteration spends minimal time executing the kernel. However, the need to transfer data back and forth for conjugate gradient calculations, for each frequency slice, contributes significantly to the diminished speedup.

RESULTS

We conducted 5D interpolation on a 3D test dataset, starting with the input decimated data shown in Figure 2. Subsequently, interpolation was performed using FFT as depicted in Figure 3, DFT on CPU in Figure 4, and DFT on GPU in Figure 5. A noticeable distinc-

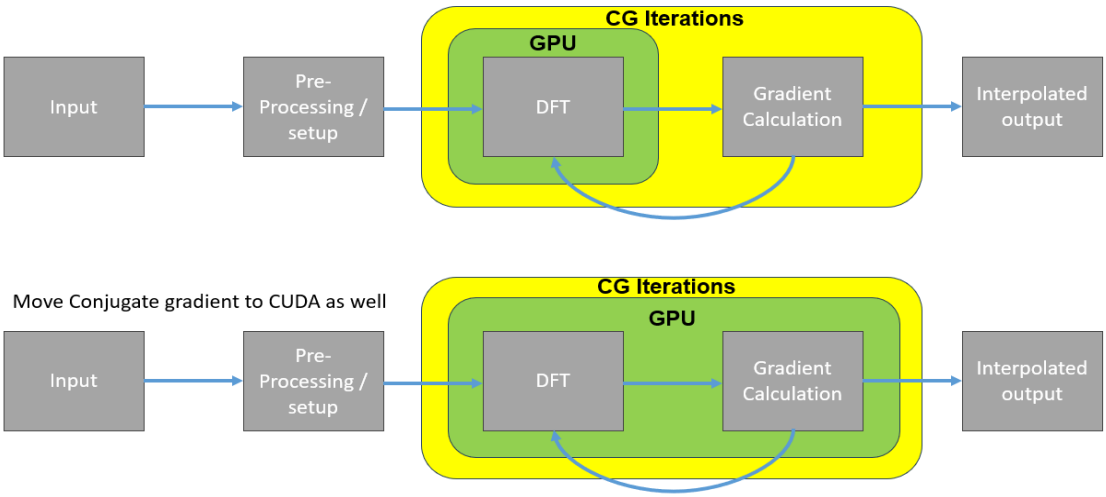

FIG. 1. Ways to perform CUDA interpolation.

tion emerges at far offsets between the FFT and DFT results which can be seen in Figure 7. Near the apex, the DFT exhibits a smoother interpolation compared to the FFT results. Given that the data is synthetic, and the original data is perfectly smooth in that region, this discrepancy underscores the superior output of NUDFT interpolation.

Our initial runtime measurements indicate a general speedup, with the NUDFT on GPU outperforming the CPU at a ratio of approximately 1 to 3. In other words, the GPU accomplishes the task in about 1/3 of the time taken by the CPU. These computations were carried out on a modern Core i5 CPU and a midrange NVIDIA GPU, both at comparable prices. It is noteworthy that this result contrasts with the raw throughput difference observed for the base NUDFT operator, where a 360x runtime difference was evident. We attribute this contrast primarily to the challenges highlighted earlier, specifically the overhead associated with memory transfers between calls. In our next code iteration, we plan to conduct the conjugate gradient calculations on the GPU, theoretically significantly reducing the overhead by minimizing memory transfers.

We also plot the difference between the GPU and CPU implementation of the NUDFT in Figure 6. A noticeable difference in the final output is apparent, particularly notable in the tails of the far offset traces. Specifically, at the tail ends of the far offset, the second event seems to exhibit a slight upward curvature compared to the CPU result. Additionally, there are minor artifacts evident in the GPU result. We are currently investigating the root cause of these differences, and our hypothesis is that there might be some bleedover from the first event into lower events, as indicated in the difference plot. This discrepancy may potentially be attributed to a bug in the GPU code, and further investigation is underway.

CONCLUSION

The adoption of GPUs for non-uniform Discrete Fourier Transforms in 5D interpolation addresses the issue of far offset binning errors commonly encountered in this process. The application of the operator on GPUs enables a reduction in runtime, which constitutes the

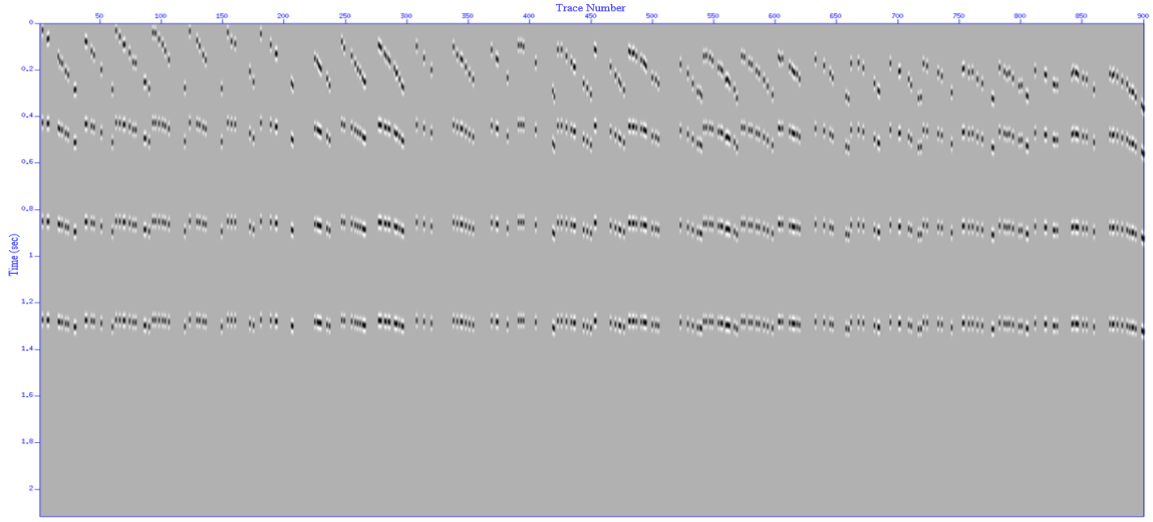

FIG. 2. Decimated shot.

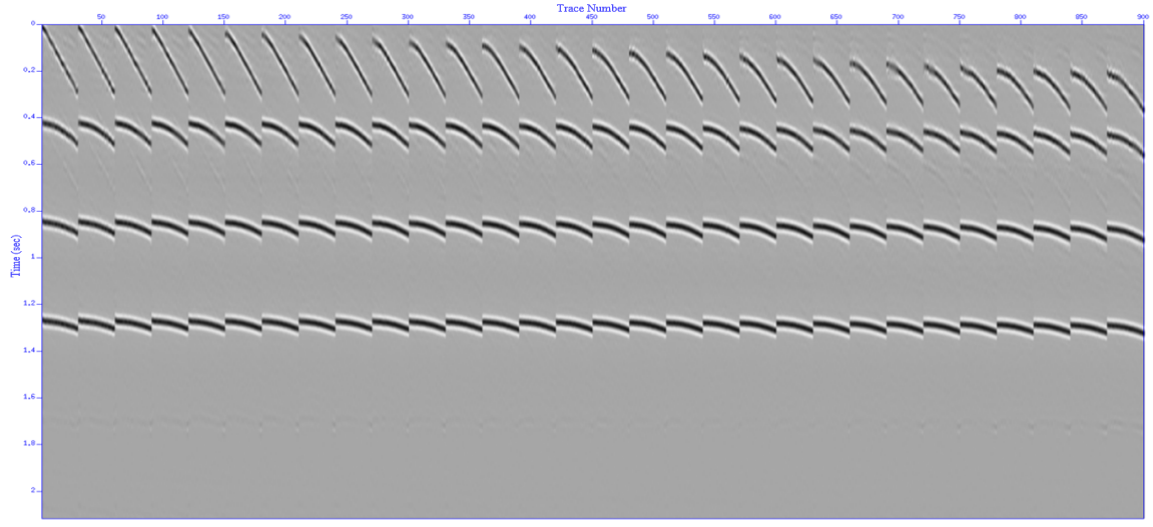

FIG. 3. 5D interpolation using the FFT operator.

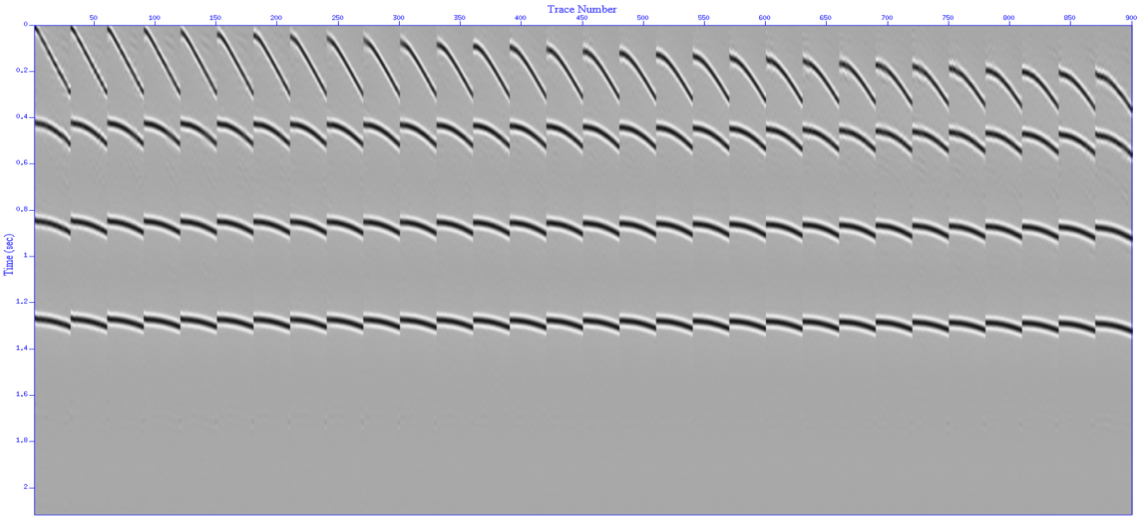

FIG. 4. 5D interpolation using the DFT operator on CPU.

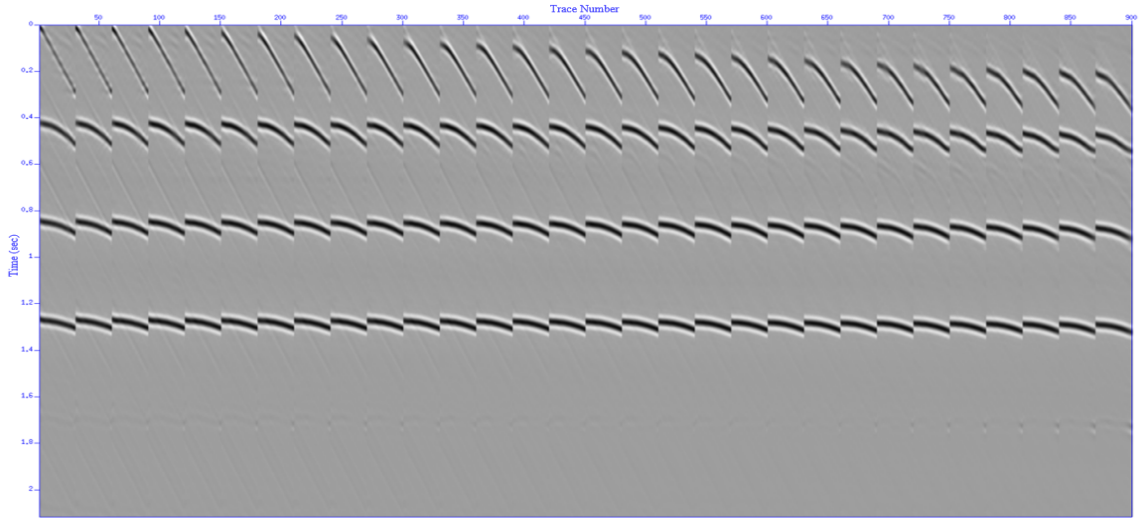

FIG. 5. 5D interpolation using the DFT operator on GPU.

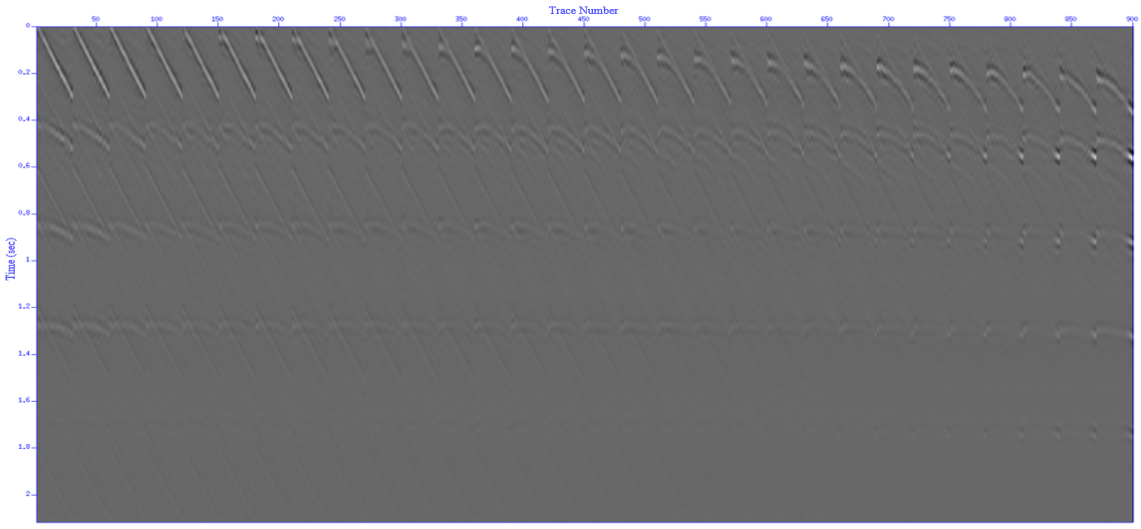

FIG. 6. Difference between GPU and CPU operator results.

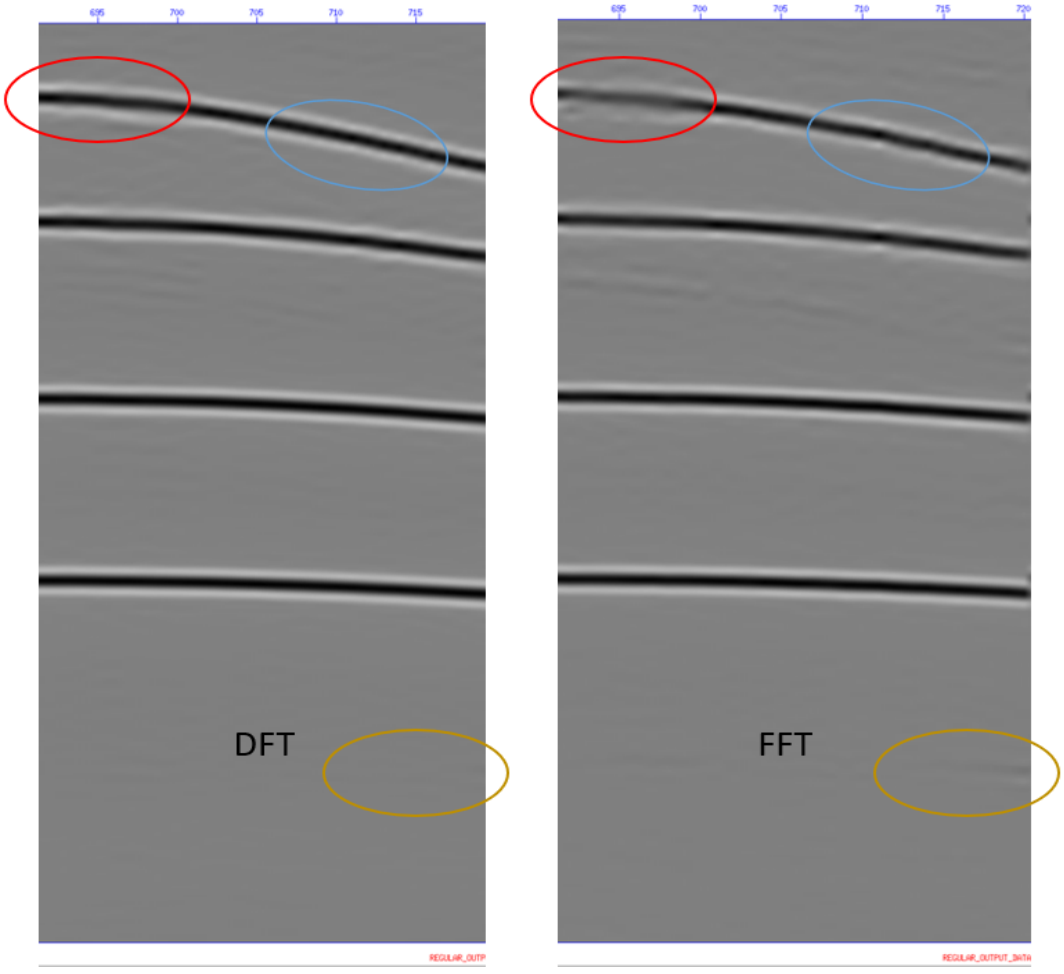

FIG. 7. Zoomed difference between DFT and FFT.

primary limitation in implementing this method.

In the conventional FFT approach, which involves binning traces to transform data into evenly spaced grids, far-offset data is inevitably distorted. However, opting for a Non-Uniform DFT approach instead of an FFT preserves the exact trace locations. The runtime challenges inherent in using Non-Uniform DFT are subsequently resolved with the transition to GPU processing. GPUs exhibit significantly faster performance in matrix operations compared to CPUs, resulting in a notable speedup. This heightened processing speed facilitates the implementation of Fourier transforms without the need for spatial binning.

While we observed a speedup in the 5D program when transitioning from CPUs to GPUs, it was significantly lower than expected. We attribute this to the memory transfer overhead and plan to address it by performing the entire conjugate gradient algorithm on the GPU instead of just the operator. This modification allows for keeping memory on the GPU rather than transferring it off every iteration, aiming to enhance overall performance significantly.

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

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`

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.