Parallelizing the Fast Fourier Transform

Mitchell Gilmore, Austin Heath, Emre Karabay, Chen Zhang

Boston University
Department of Electrical and Computer Engineering
8 Saint Mary's Street
Boston, MA 02215
www.bu.edu/ece

May. 3, 2024

Introduction

For our final project, we focused primarily on parallelization methods of the Fast Fourier Transform (FFT). The Fourier transform, and its efficiently computable variant called Fast Fourier Transform (FFT), is a mathematical technique used to convert a signal or a function of time or space into a frequency domain representation. It's a fundamental tool in the field of digital signal processing, instrumental for operations such as image analysis, data compression, and solving partial differential equations. In practical applications, it's paramount for processing information in a wide range of areas, including engineering, physics, and data analysis. FFT's ability to break down a complex signal into a series of simple sine waves allows for important insights into the individual components of the signal, making it a cornerstone of spectral analysis. Figure 1 depicts a standard butterfly computation diagram.

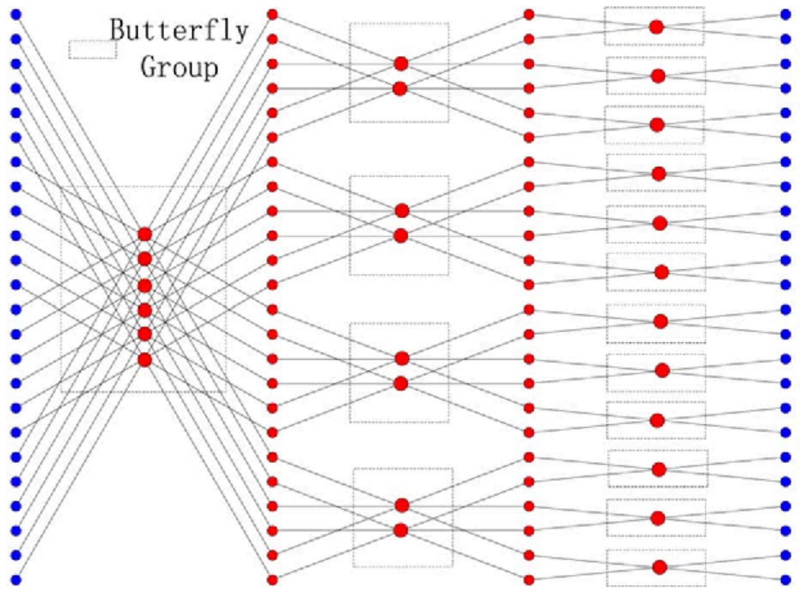

Figure 1: Butterfly FFT

We created several variants of the algorithm in C++ and Cuda, wrote timing code to compare performance, and investigated the Sparse & Pruned FFT variant algorithms. We ran out of time to properly test and compare these other types of algorithms, but we find the algorithms to be extremely interesting when dealing with large amounts of data. In addition to investigating the Sparse FFT, we also studied sample rate changing in the fourier domain. This may have neat applications in machine learning. We do not need to modify the full algorithm and instead, we tweak the parameters (roots of unity) of the FFT directly. This paper discusses some of our findings and experiments.

Pruned & Sparse FFTs

A pruned FFT is a specialized version of the transformative algorithm that strategically eliminates certain computations to increase efficiency. While the traditional FFT computes all complex exponentials, the pruned FFT only executes computations for specific frequencies of the input data, discarding others regarded as unimportant. It's employed in cases where only a subset of the FFT outputs is required, thereby saving computational resources. This approach proves particularly useful in high-speed digital signal processing, real-time systems, audio analysis, and other spectral analysis applications, where computational efficiency and speed are essential.

However, the performance enhancements achieved through pruned FFTs are typically modest. Though they manage to reduce the complexity when compared to the full FFT, they save only a minimal factor in the logarithm. Optimizing an FFT requires considerable effort, and the gains from a pruned FFT might be offset by the sacrifices made in optimizing full FFTs. Therefore, one should only consider pruned 1D FFTs when a minute fraction of the outputs is needed for a minute fraction of the inputs is non-zero. Pruned FFTs might benefit higher dimensions, such as in zero-padded convolutions. Still, other alternatives, such as the naive DFT formula or the Goertzel algorithm, offer different advantages, including storage requirement reduction. However, these methods may lack the precision of FFT-based methods.

FFTW currently does not implement any general pruned FFT algorithm, but it is possible to implement pruned FFTs using FFTW. We attempted this and the incomplete codes are in the Github project. We also investigated the solution and code proposed by a team of researchers at Washington State University for taking advantage of known sparsity within the FFT input. This is relevant when the user expects only a couple frequencies with large magnitudes. The paper claims to get O(S log N) performance where S is much smaller than N and represents the number of frequencies with large magnitudes. This Sparse FFT (Fast Fourier Transform) is primarily used when you have a signal that is mostly zero-valued in the frequency domain, meaning there are only a few non-zero frequencies. It aims to exploit this sparsity to achieve computational efficiency.

This solution, called the Discrete MSFT (DMSFT) algorithm, assumes there exists an algorithm that will always deterministically return an s-sparse vector v ∈ C N satisfying the equation below:

$$\left\|\hat{\mathbf{f}} - \mathbf{v}\right\|_2 \leq \left\|\hat{\mathbf{f}} - \hat{\mathbf{f}}_s^{\mathrm{opt}}\right\|_2 + \frac{33}{\sqrt{s}} \cdot \left\|\hat{\mathbf{f}} - \hat{\mathbf{f}}_s^{\mathrm{opt}}\right\|_1 + 198\sqrt{s}\,\|\mathbf{f}\|_\infty N^{-r}$$

Equation 1: WSU MSFT Theorem 1

The algorithm labeled as #1 in the paper, called "A Generic Method for Discretizing a Given SFT Algorithm A" satisfies the O(S log N) runtime and the equation above. From our brief research, this algorithm is not used outside of academia, however there have been several other attempts to take advantage of sparsity in the frequency domain. Some of these implementations are listed below. Note that none of these algorithms are parallelized at the moment:

- SFFT by MIT
- Sparse FFT on top of FFTW, written by the authors of FFTW. However, this implementation relies on an old version of FFTW.
- DMSFT by Ruochuan Zhang. This is the algorithm developed at Washington State.

FFT Sample Rate Modifications

A FFT is a method to decompose discrete time signals into discrete frequency signals. Though the FFT algorithm, namely the cooley-tukey algorithm is just a factorization of the sum of monotonically increasing exponentials on a regular grid. The algorithm is specialized to the FFT by simply using the fourier basis as the base of the exponentials, but the factorization of the algorithms is far more general than this. The FFT specifically is popular because it is very interpretable given the extensive research of signal processing and the DFT and therefore a highly optimized operation in many hardware platforms, but the divide and conquer is applicable to any base.

Therefore you can modify the FFT implementation to realize different operations on a signal that is not trivial in either domain. Within this project we implemented one such application that allows for time and frequency scaling to be realized within the transformation of the FFT. This can be done simply through the understanding that the DFT simply decomposes a signal into different continuous basis functions. By evaluating these basis functions on different grids you can accomplish both time and frequency scaling. To show this capability we have included a CUDA implementation of the Cooley-Tukey Radix-2 FFT with the option to scale the signals.

FFT Experiments

We were able to implement some of the following algorithms completely. Note, this does not include algorithms written with CUDA or the FFT Sparse partial solutions. Some of these codes can be run by following the README in the GitHub repository. The timing experiments in Figure 2 were taken on SCC 1. Results may vary depending on configuration parameters and machine.

Naive DFT	0.56181500 MS
Custom Standard FFT	0.25074100 MS
FFT from FFTW	0.32383100 MS
Threaded FFT (using std::thread, not pthreads)	0.16408700 MS
Attempted Sparse FFT	In progress
Attempted FFT with MPI	In progress
Attempted FFT in CUDA	In progress

Figure 2: FFT timing experiments

As expected, the threaded FFT performed the fastest in a three dimensional fourier transform. However, there are cases where the standard FFT outperformed the threaded variant when N was small. This coincides with the added setup and tear down costs associated with threads. Please see the GitHub page for execution and timing instructions.

Conclusion

Our project focused on parallelization methods for the Fast Fourier Transform (FFT), essential in digital signal processing and machine learning. We explored variants like Pruned FFTs and Sparse FFTs, emphasizing efficiency improvements and computational complexities. While Pruned FFTs offer modest gains by strategically eliminating computations, Sparse FFTs leverage sparsity in the frequency domain for efficiency. Our experiments with various FFT implementations provided insights into their performance, with threaded FFTs showing superiority in three-dimensional transforms. Despite time constraints limiting comprehensive testing, our code lays a foundation for future optimizations and exploration in FFT parallelization methods.

Sources

Ghazi, B., Hassanieh, H., Indyk, P., Katabi, D., Price, E., & Shi, L. (2013). Sample-optimal average-case sparse fourier transform in two dimensions. *UIUC*. https://doi.org/10.1109/allerton.2013.6736670

Gilbert, A. C., Indyk, P., Iwen, M. A., & Schmidt, L. (2014). Recent Developments in the Sparse Fourier Transform: A compressed Fourier transform for big data. *IEEE Signal Processing Magazine*, *31*(5), 91–100. https://doi.org/10.1109/msp.2014.2329131

Kapralov, M. (2016). Sparse fourier transform in any constant dimension with nearly-optimal sample complexity in sublinear time. *MIT*. https://doi.org/10.1145/2897518.2897650

Merhi, S., Zhang, R., Iwen, M. A., & Christlieb, A. (2017, June 8). *A new class of fully discrete sparse fourier transforms: Faster stable implementations with guarantees*. arXiv.org. https://arxiv.org/abs/1706.02740

Pruned FFTs with FFTW. (n.d.). https://www.fftw.org/pruned.html