A Parallel Application of the Fourier Transformation

Justin Spidell – Brett Sumser

October 2021

Abstract

This project intends to implement a parallized version of The Fourier Transform, called the Fast Fourier Transform (hereby referred to as the FFT) to convert analog signals into digital signals and compress images. Specifically, a Discrete Fourier Transform (hereby referred to as the DFT) is used. By taking a waveform and sampling it at a certain constant frequency, we gather discrete signals in the form of a sum of sines and cosines that can be easily stored or analyzed. This is how common storage formats like mp3 or jpg are compressed.

We intend to create an implementation of the FFT to compress images and digital audio, as well as convert analog audio to digital. Our project will be written in C++, and by taking advantage of the University of Oregon's supercomputer, we will test our compression algorithm on its speed, and loss of quality with small and large pictures and audios.

Introduction

The Fourier Transform is an important mathematical concept. It has applications in digital signal processing, convolution in neural networks, image recognition and even speech processing. The main idea behind the Fourier Transform is that it is a "mathematical operation that changes the domain (x-axis) of a signal from time to frequency," [4]. The Fourier transform is denoted by adding a circumflex to the symbol of a function:

$$f \to \hat{f}$$

The Fourier transform is defined as:

$$\hat{f}(\xi) = \int_{-\infty}^{\infty} f(x) e^{-2\pi i x \xi} dx \tag{1}$$

The particular use case for the Fourier Transform that initially started this project was that of digital signal processing, specifically the conversion of analog signals to digital for guitar effects processing. Using the Fourier Series, it is apparent that any complicated wave form can be taken, and represented as an infinite series of sine and cosine functions [3]. Unfortunately, the DFT is on the slower side for algorithms, being performed in $O(n^2)$. It is defined as:

$$Let\ x_0,\ ..., x_{N-1}\ be\ complex\ numbers,$$

$$X_k = \sum_{n=0}^{N-1} x_n e^{-i 2\pi k n / N} \quad k = 0,\ ...,\ N-1 \tag{2}$$

But by using a divide and conquer strategy over the DFT, an algorithm called the Fast Fourier Transform, is formulated. The FFT is much faster than the DFT, being performed in $O(n \log n)$ time. The FFT can be easily implemented using a recursive or iterative programming method, but there are benefits to using an iterative approach. For example, iterative implementations can use less index computations, as well was being able to be parallized far more easily than a recursive version. This leads us nicely into the main impetus for this project, and the class as a whole.

Parallization

There are a few different directions to explore when developing a more parallized implementation of the Fourier Transform. According to Anthony Blake in his thesis paper titled "Computing the Fast Fourier Transform on SIMD Microprocessors", use of the FFT algorithm is extremely widespread in multiple disciplines. He goes on to state that "use of the FFT is even more pervasive, and it is counted among the 10 algorithms that have had the greatest influence on the development and practice of science and engineering in the 20th century," [1]. The widespread use of the FFT algorithm provides great evidence for the need to optimize for different applications, and to understand the methods that can be used to achieve this. Two methods of Parallelization that stand out in particular are SIMD, and multithreading.

SIMD

Single Instruction Multiple Data prcoessors are also known as vector processors, and allow a single instruction to process multiple pieces of data in the same clock cycle. The way that SIMD works is by packing several pieces of data into one data word. This allows the instruction loaded to act on each piece of data from a single instruction. This has applications in situations where large amounts of data are being manipulated. With respect to the Fourier Transform, this can be applied to image manipulation, audio processing, or other functions.

According to this post [2] on StackOverflow, C/C++ contains SIMD functions called vector intrinsics. These are implemented in the compiler, allowing them to load orders of magnitude faster than common library functions. The found that they could use these intrinsics to produce code up to four times faster, and even more in certain cases! For this project, getting a handle on these intrinsics would allow for an increased knowledge in C/C++, while also providing experience in possible future scenarios with performance critical code.

Multithreading

After some preliminary research into a multithreading implementation of FFT [5], it seems there are possible multithreaded implementations of the FFT. Based on the previously cited source, Thulasiraman et al. picked two

different algorithmic approaches to investigate. The first is what they referred to as a sender-initiated algorithm based on the Cooley-Tukey FFT algorithm. The Cooley-Tukey algorithm is the most common of the FFT variations, recursively breaking the DFT into N_1 DFTs of size N_2, multiplying by complex roots of unity, and performing N_2 DFTs of size N_1.

The second is stated as a sender-initited algorithm, utilizing a thread pool fixed at compile time. The amount of threads is equal to N/B, with B being the block list for a list of contiguous data pieces. Threads are then distributed between processors round-robin style, which enables load balancing across available processors. In the end, each processor will perform an FFT computation on B data pieces. This version of the algorithm seems like it may be applicable for use on the University of Oregon supercomputer.

Application

We plan to use our FFT implementation to compress images and audios. We will test our implementations speed, comparing with industry standards, and loss, by comparing with the originals. This process will all be automated, taking place on the University of Oregon's supercomputer. We hope to achieve something similar in time complexity and error to commonly found compression software, however this may be a hard goal to reach.

We will also test our FFT implementation by converting analog audio to digital, and comparing for loss. This is a less intensive but still productive test, being our original goal and very applicable to a real life scenario.

Overall this FFT implementation will have many functions and possible applications, but only testing a few we hope to create an algorithm that is effective and efficient.

Conclusion

By creating a implementation of the Fast Fourier Transform, we hope to gain insight into modern image and audio compression software, as well commonly found analog to digital conversion software. We will attempt to implement a more sophisticated parallelized version of the FFT, using concepts applied from class such as SIMD and multithreading/processing. We are excited to be able to have access the hardware capable of applying these concepts for

parallelized computing. The University of Oregon's supercomputer will be applied for testing and use of our implementation, hopefully with success. This is a lofty goal, and will take a good amount of work, but the end result will be something to be proud of.

References

[1] Anthony Blake. Computing the fast fourier transform on simd microprocessors, 2012.

[2] Konstantin. Improving performance with simd intrinsics in three use cases, 2020.

[3] Bo Liu. Parallel fast fourier transform, N/A.

[4] Cory Maklin. Fast fourier transform, 2019.

[5] Parimala Thulasiraman. Multithreaded algorithms for the fast fourier transform, 2020.