Parallel Fast Fourier Transform

CS597 Concurrency and Algorithms

Francis Joseph Serina

# Introduction

The Fourier Transform is an integral transform to decompose a signal, f(t), into a sum of frequencies, F(ξ). It is an important transform used in numerous applications most commonly in signal processing. The Fourier Transform can be applied on signals of higher dimensions. Due to the nature of the transform, each dimension can be computed in sequence.

However, the transform is computationally expensive. Several algorithms, known as Fast Fourier Transforms or FFT, have been proposed to reduce the execution time. A popular FFT implementation is the Cooley-Tukey FFT Algorithm. The Cooley-Tukey FFT requires that the length, of each dimension, of the input data should be a power of 2.

This document explores 3 ways of implementing Cooley-Tukey FFT and compares their execution times. The input data will be 2D grayscale images.

# Goals

* Implement 2D FFT in 3 different methods – Single Thread, Multi-Thread, Compute Shader.
* Compare the speedup of the Multi-Thread and Compute Shader implementations against the Single Thread implementation.

# Input Data

* 2D Grayscale Image
* Width and Height are powers of 2

The input will go through pre-processing as follows

1. Convert Grayscale to Complex Data
   1. The input value per pixel will be in the range of [0, 1]. This will be treated as the real part of the Complex Data.
2. Shift Frequency Domain
   1. Shift the Frequency Domain by half of the width and height by negating elements when the sum of its row and column indices is odd.

# Fast Fourier Transform

Cooley-Tukey FFT in 2D is as follows

1. Bit Reversal by Row
2. Apply Butterfly by Row
3. Bit Reversal by Column
4. Apply Butterfly by Column

After step (2), the processed data is referred to as *Intermediate*. While the data after step (4) is called *Final*.

Solving the Butterfly network is the bottleneck of the algorithm. All else (Applying Bit Reversal, Computing the Twiddle Factors) is linear in complexity.

It is interesting to note that FFT’s running time is fixed – worst = best = average case – since there are no conditions dependent on the input data other than its size. The running time is **O(NM logNM)** where N and M are the dimensions.

# Output Data

The resulting data will be a 2D array of Complex Numbers. Post-processing the data will help visualize the result. The post-processing is as follows

1. Get Magnitude (or Phase) Spectrum
2. Apply logarithm and scaling
3. Normalize to be between [0, 1]
4. Convert to grayscale

# Inverse FFT

The FFT algorithm is reversible. To compute for the inverse, we perform the following

1. Take the Conjugate per element
2. Apply Forward FFT
3. Take the Conjugate per element again
4. Divide each element by Width \* Height

# Single Thread

This will serve as the baseline comparison for the other 2 implementations.

See *ConcurrencyAndAlgorithms/source/common/fft.cpp* for the 1D and 2D implementations of FFT for a single thread.

# Multi-Thread

In this variation, the main thread divides the work via the following:

1. Compute the Bit Reversal Indices and Twiddle Factors for Width
2. Divide Height by the number of worker threads
3. Each worker thread receives the following:
   1. Portion of Data to work on
   2. Bit Reversal Indices
   3. Twiddle Factors
4. Each worker performs on their assigned portions simultaneously
5. Aggregate partial data from workers
6. Repeat but by column instead

|  |  |
| --- | --- |
| Distribution of Tasks by Row | Distribution of Tasks by Column |
| |  | | --- | | Worker 1 | | Worker 2 | | Worker 3 | | … | | |  |  |  |  | | --- | --- | --- | --- | | Worker 1 | Worker 2 | Worker 3 | … | |

See *ConcurrencyAndAlgorithms/source/common/pfft.cpp* for the 2D implementation of FFT for multithreading.

# Compute Shader

In this approach, each element simultaneously computes for its value in the butterfly network. The main thread synchronizes all elements by dispatching the next stage only when all elements are done in the current stage.

Instead of C++, this is implemented in C# and DirectCompute using Unity3D.

All textures imported have the following import settings:

* *sRGB (Color Texture)* Off
* *Alpha Source* None
* *Generate Mip Maps* Off
* *Wrap Mode* Clamp
* *Filter Mode* Point (no filter)
* *Max Size* 8192
* *Compression* None

Theoretical execution time is reduced to **log2NM** which is the number of synchronizations needed by the main thread.

See *FFTv2.compute* and *FFTComputeHelper.cs* in *ConcurrencyAndAlgorithms/source/Unity/Assets/FFTv2/* for the Compute Shader and the Helper class for the 2D implementation for the GPU.

# Results

|  |  |  |  |
| --- | --- | --- | --- |
| Test Data | Single Thread | Multi-Thread | Compute Shader |
| 4096x2048 (8 MP) | 4.94 s | 2.23 s | 1.96 ms |
| 8192x4096 (32 MP) | 21.2 s | 9.27 s | 2.07 ms |

Hardware Notes:

* Single and Multi-Thread are using Intel Core i7-7700 CPU @3.60 GHz
* Multi-Thread is using 8 cores (1 main thread + 7 worker threads)
* Compute Shader is using NVIDIA GeForce GTX 1070 Ti

Timing Notes:

* Only the processing is considered. Loading/Saving images are not included in the timing.
* Single and Multi-Thread are built and ran in Release
* Compute Shader is performed inside the Unity Editor

# Conclusions

The Multi-Thread approach has a speed of approximately 2.2 times compared to the Single Thread for both 8MP and 32MP test data. This could perhaps be a result of having more data to pass between threads.

The Compute Shader approach is significantly faster compared to the Multi-Thread approach. The difference between the 8MP and 32MP images are small because log2(8M) is 23 while log2(32M) is 25. The stage difference of 2 is insignificant compared to the rest of the algorithm.

# References

Lang, H. W. (2018, April 6). *Bitonic sort*. Retrieved from Hochschule Flensburg - University of Applied Sciences: http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/bitonic/bitonicen.htm

Microsoft Corporation. (2018, May 30). *HLSL*. Retrieved from Windows Dev Center: https://docs.microsoft.com/en-us/windows/desktop/direct3dhlsl/dx-graphics-hlsl

Unity3D. (2017, May 18). *Compute shaders*. Retrieved from Unity Documentation: https://docs.unity3d.com/Manual/class-ComputeShader.html

Williams, A. (2012). *C++ Concurrency in Action.* Manning Publications Co.