CPEG655 - Final Project

FFT on GPU - Project Report

Group Members:

Kshitij Srivastava

Kun XIa

Introduction

Programmers in the field of Computer Science have always tried to make programs run faster without compromising on efficiency. Fourier Transform is one of the most powerful and effective approach used not only in the field of mathematics but also in other fields such as numerical simulations, signal processing and so on and so forth. Main advantage of using fourier transform is that many differential equations are easier to solve in frequency domain rather than in time domain. The Navier Stokes equation, used for determining velocity field of a fluid flow, is one such example.

The main goal of our project is to implement fourier transform of a function on a GPU in order to reduce its execution / running time. Generally fourier transform of an input signal is calculated using Discrete Fourier Transform (DFT). However, this is a serial version and can take a lot of time to execute especially for large size input. Fast fourier transform is an optimized version of DFT which works on the principle of divide and conquer. This is also a serial version but it decreases the execution time w.r.t to DFT. Our goal is to implement FFT in parallel on a GPU. This will substantially reduce the execution time.

Theory

Any function which is continuous in time domain could be converted to its frequency domain . This converted frequency domain is called as Fourier Transform. It can be mathematically denoted as :

$$\hat{u}(k) = \int_{-\infty}^{\infty} e^{-ikx} u(x)\, dx$$

We can perform inverse fourier transform to get the function back in time domain / physical domain.

$$u(x) = \int_{-\infty}^{\infty} e^{ikx} \hat{u}(k) dk$$

Discretization:

When we look at the equation closely, we find a problem. We cannot have infinite domain and differentiate it at the same time. We need a finite size domain and a finite value for dx. After discretizing the equation , we get Discrete Fourier Transform which can be expressed mathematically as :

$$\hat{u}_k = \sum_{j=0}^{N-1} u_j e^{-\frac{2\pi i}{N} kj} \qquad \text{k=0,1,....(N-1)}$$

and we can get its fourier transform by :

$$u_j = \sum_{j=0}^{N-1} \hat{u}_k e^{-\frac{2\pi i}{N} kj} \qquad \text{j=0,1,....(N-1)}$$

Problem:

Although we will get correct results if we use above mentioned equations, but it will be very time consuming. Precisely, it will take 0(N^2) operations. So we need to use a slight variant of DFT which is based on divide and conquer approach called as Cooley-Tukey algorithm.

Cooley - Tukey Algorithm:

In 1965, James Cooley and John Tukey introduce the FFT to public, and reduced the computation cost of DFT from O(N^2) to O($Nlog_2N$).

According to the article on *Pythonic Perambulations* website, Cooley and Tukey's idea divided the DFT computation into two small parts, and the function is shown below.

$$
\begin{aligned}
X_k &= \sum_{n=0}^{N-1} x_n \cdot e^{-i\,2\pi\,k\,n\,/\,N} \\
&= \sum_{m=0}^{N/2-1} x_{2m} \cdot e^{-i\,2\pi\,k\,(2m)\,/\,N} + \sum_{m=0}^{N/2-1} x_{2m+1} \cdot e^{-i\,2\pi\,k\,(2m+1)\,/\,N} \\
&= \sum_{m=0}^{N/2-1} x_{2m} \cdot e^{-i\,2\pi\,k\,m\,/\,(N/2)} + e^{-i\,2\pi\,k\,/\,N} \sum_{m=0}^{N/2-1} x_{2m+1} \cdot e^{-i\,2\pi\,k\,m\,/\,(N/2)}
\end{aligned}
$$

After getting those two small DFT function, we can continue divided each of them into another smaller two DFT functions. As long as the single DFT function is small enough, we can reduce the computation cost from O(N^2) to O($Nlog_2N$). Chart 1 from *versci.com* is shown below. In this chart, we can find an eight point sequence input signal, and we start with 2 point Fourier transforms.Then, we get 4 points, and 8 points Fourier transforms.

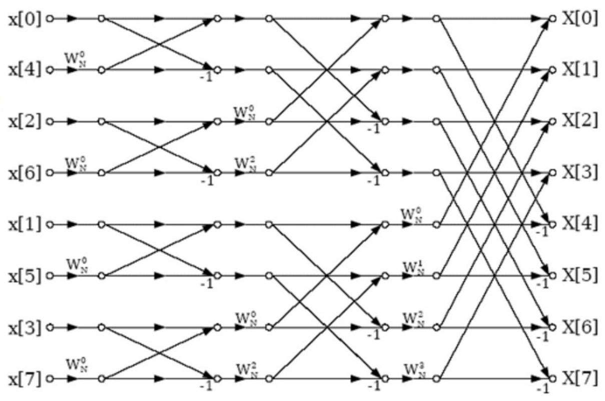


Chart 1

The reason for reduction in order of operation from O(N^2) to O($Nlog_2N$) is because:

Cooley-Tukey algorithm is an example of a divide and conquer algorithm. A divide and conquer (D&C) is an algorithm design paradigm based on multi-branched recursion. A divide and conquer algorithm works by recursively breaking down a problem into two or more subproblems of the same (or related) type (divide), until these become simple enough to be solved directly (conquer). The solutions to the subproblems are then combined to give a solution to the original problem.

The FFT method employs a divide-and-conquer strategy, using the even-indexed and odd-indexed coefficients separately to define two new polynomials of degree-bound n/2 each of which are evaluated recursively. These subproblems have exactly the same form as the original problem, but are half the size. To determine the running time of this procedure, we note that exclusive of the recursive calls, each invocation takes time O (n), where n is the length of the input vector.

Methodology

1. Serial version of DFT and FFT were implemented in C. Structure was used to store the real part and imaginary part. We tried using arrays to store both the parts but it complicated the code.
2. RMSE value was calculated for the two versions upto N=16384. Above this value, it was almost impossible to calculate DFT because our program was terminated by the server as our code exceeded the time limit. RMSE value was within the permissible limit.
3. Next, parallel version of FFT was implemented in CUDA. RMSE value and difference in execution time between three versions was calculated.
4. We use cuComplex library for our parallel FFT code so that we could use inbuilt structure provided by CUDA to store the real part and imaginary part. We used it for one more reason. cuFFT library also uses same complex structure and so it would be easier for us to compare our parallel version with cuFFT version in the future.
5. Before FFT kernel function is called, we use Rader's FFT algorithm to reorder the input data sequence for our parallel version FFT.
6. We used "int" to store some variables in our code but as the input size increased, these variables overflowed. So we had to change int to long to avoid overflow.
7. Initially we were calculating twiddle factor on the fly. As we know that there could be slight difference between DFT and FFT as the input size increases, eventually we

realised that calculating twiddle factor on the fly made the situation worse because it had a cascading effect. So we calculated twiddle factor once.

8. For our parallel version, one thread calculated one element and number of threads in a block were 512.

EXECUTION OF THE CODE

Machine Specification: We are using university server which has two Tesla K40c GPUs. Some of its specifications are:

1. Name: Tesla K40c
2. Max threads / block: 1024
3. Max thread size: 1024 * 1024 * 64
4. Shared mem /block: 48KB
5. Constant memory: 64KB
6. Wrap size: 32
7. Registers / block: 65535
8. Clock rate: 745MHz

Compiling and running the program:

The following method explains how the codes were run on server. Method was similar for all the questions, therefore, I am explaining it for just one part.

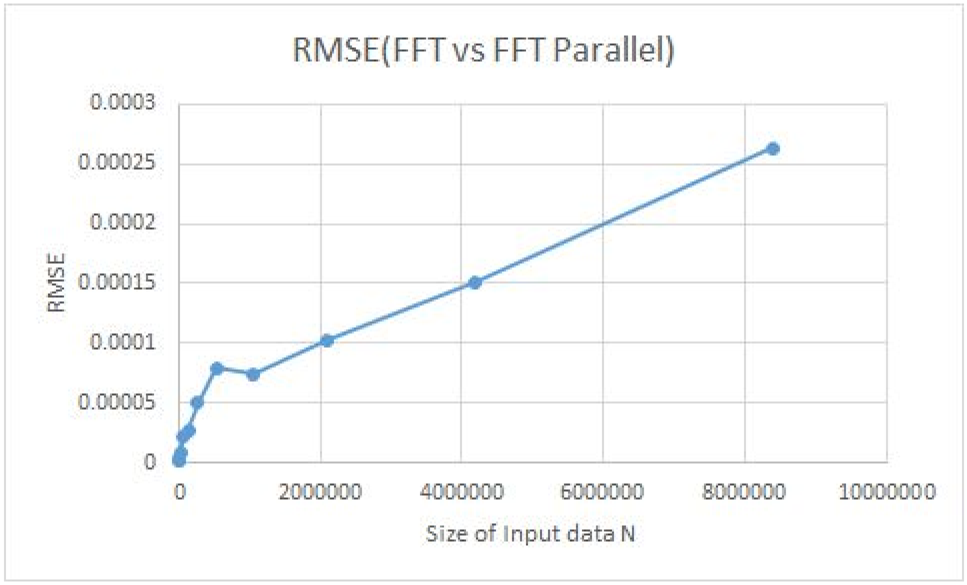
1. After logging into mlb.acad.ece.udel.edu, we were supposed to ssh into cpeg655.ece.udel.edu
2. After this, in order to compile the code, I used: /usr/local/cuda/bin/nvcc -o temp device.cu –lcuda
3. In order to run the code, I used: srun -N1 --gres=gpu:1 temp

So the above code executes device.cu file which extracts all the properties of the GPU and runs it. Similar approach was used for all other files.

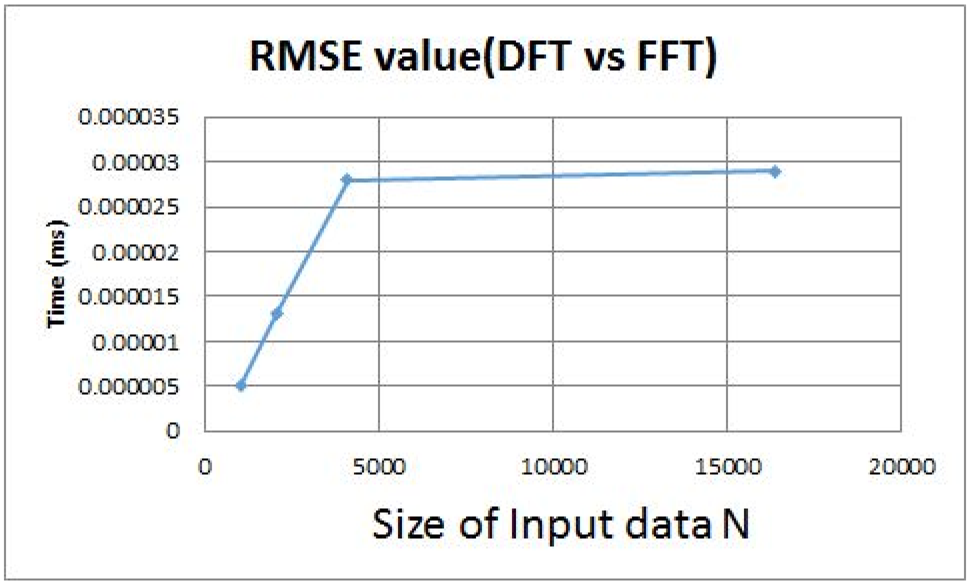
Our code consists of Serial DFT , Serial FFT and Parallel FFT. At the time of submitting the code, value of N = 131072, so we had to comment the DFT part otherwise our code will time out. If you want to run the DFT part then, please comment out line number : 298 - 305 , 315-325 and 329-334. This will run DFT part as well.

Results

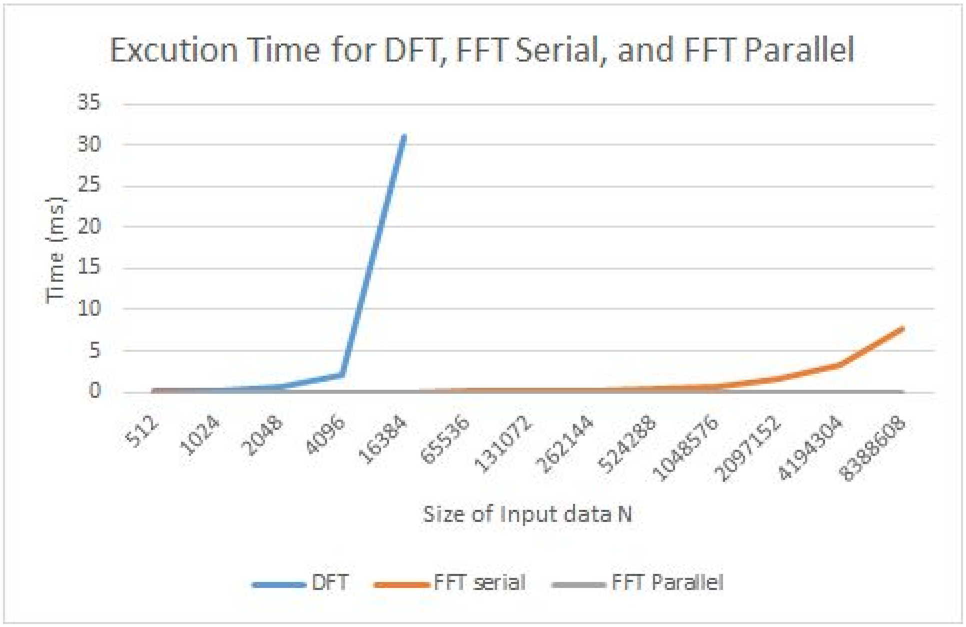
N	DFT(millisecond)	FFT Serial(millisec)	FFT Parallel (millisec)	RMSE value (FFT serial v DFT)	RMSE value (FFT Parallel v FFT serial)
512	0.045761	0.000294	0.000037	0.000002	0.000001
1024	0.175117	0.000628	0.000041	0.000005	0.000002
2048	0.607248	0.001178	0.000045	0.000013	0.000003
4096	2.034476	0.002497	0.000049	0.000028	0.000002
16384	31.012565	0.012119	0.000057	0.000029	0.000009
65536	-	0.053196	0.000064	-	0.000022
131072	-	0.087167	0.000068	-	0.000027
262144	-	0.170273	0.000072	-	0.000050
524288	-	0.357368	0.000075	-	0.000079
1048576	-	0.737331	0.000073	-	0.000074
2097152	-	1.540531	0.000088	-	0.000103
4194304	-	3.237195	0.000092	-	0.000151
8388608	-	7.65048	0.000097	-	0.000263



As you can see , RMSE value increases linearly but on having a closer look at the graph, it can be seen that the magnitude of increase in RMSE is very small.



It is interesting to see that RMSE value increased sharply initially but increased rather slowly afterwards. However, it is interesting to note that magnitude increase in RMSE value still remains very less.



It can be seen that running time of DFT increased exponentially whereas running time of FFT serial implementation increased almost linearly, however, on the other hand running time of parallel version remained almost constant.

Conclusion:

FFT and Parallel FFT can reduce the computation time of DFT tremendously as can be seen from our results. We verified that our serial implementation and parallel implementation of FFT are correct as can be seen from the RMSE value which we calculated at each step. Our parallel version has sped up the process. It can be seen that running time has been reduced considerably for parallel FFT. We could not show the running time of DFT version after N=16384 because it exceeded the time limit on the server.

In the future , we would like to compare our code with cuFFT library and also use shared memory in order to comprehend its effect on running time.

In the end , we would like to thank Dr. Li for his guidance and support.

References:

1. “An Algorithm for the Machine Calculation of Complex Fourier Series” by James W. Cooley and John W. Tukey, Math. Comp, 19 (1965), 297-301. □
2. GPU Computing with CUDA: Christopher Cooper, Boston University □
3. Introduction to Algorithms (3rd edition) by THOMAS H. CORMEN, CHARLES E. LEISERSON, RONALD L. RIVEST, CLIFFORD STEIN
4. http://en.literateprograms.org/Cooley-Tukey_FFT_algorithm_(C)
5. https://jakevdp.github.io/blog/2013/08/28/understanding-the-fft/
6. http://docs.nvidia.com/cuda/cufft/
7. https://en.wikipedia.org/wiki/Discrete_Fourier_transform□