SERIES & PARALLEL FFT's on GPU

(GACS-7306 Final Project Report)

Submitted by: Srinivas Kasturi

Submitted to: Dr. Christopher Henry

Submitted on: 12/11/2017

Abstract

To solve a significant amount of computational problems with tremendous performance, we use GPUs (graphics processing units) and GPU computing is one of the newest trends in computational science. Nvidia's CUDA GPUs are highly parallel and designed for maximum throughput. 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. Main advantage of using Fourier transform is that many differential equations are easier to solve in frequency domain rather than in time domain.

Keywords: CUDA, FFT, DFT, GPU, Parallel

1. Introduction and Project Idea

The main idea of this project is to focus on using CUDA programming environment to parallelize our problem instead of traditional C++ environment. 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, serial version 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 and also it decreases the execution time with respect to to DFT. Our goal is to implement FFT in parallel on a GPU. This will substantially reduce the execution time.

2. Literature Review

The basic idea of this project was taken from the research work of Microsoft professionals whose topic is "High Performance Discrete Fourier Transforms on Graphics Processors". To reduce the memory transpose overheads in hierarchical algorithms by combining the transposes into a block based multi-FFT algorithm. For non-power-of-two sizes, using a combination of mixed radix FFTs of small primes and Bluestein's algorithm & Cooley's algorithm. [1]

In-order to implement a parallel processing algorithm for FFT we choose Cooley-Tukey's algorithm because of reduced complex operations & parallel processing.[2] There are many existing papers discussing this problem [4][5][3][2]. Many high performance open source and vendor FFT libraries are widely used by research and industry. CUFFT is NVidia's implementation of an FFT solver on their CUDA architecture. CUFFT employs a radix-n algorithm, and operates by taking a user-defined plan as input which species the parameters

of the transform. It then optimally splits the transform into more manageable sizes if necessary. These sub-FFT's are then farmed out to the individual blocks on the GPU itself which will handle the FFT computation.

3. Problem Description

In this project, we look forward to analyze the performance of Parallel FFT with respect to traditional Series FFT. Although we will get correct results if we use our basic approach DFT equations, but it will be very time consuming. Precisely, it will take $O(N^2)$ operations. So we need to use a slight variant of DFT which is based on divide and conquer approach called as Cooley-Tukey algorithm to achieve this we will implement few algorithms and compare their overall performance with respect to time of execution.

4. Theoretical Framework

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 :

$$\hat{u}(k) = \int_{-\infty}^{\infty} e^{-ikx} u(x)\, dx \qquad \text{-eq.1}$$

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

$$u(x) = \int_{-\infty}^{\infty} e^{ikx} \hat{u}(k)\, dk \qquad \text{-eq.2}$$

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:

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

-eq.3

and we can get its Fourier transform by :

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

-eq.4

Problem:

Although we will get correct results if we use above mentioned equations, but it will be very time consuming. Precisely, it will take O(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.

A Fast Fourier transform (FFT) is an algorithm to compute the discrete Fourier transform (DFT) and it's inverse. There are many different FFT algorithms involving a wide range of mathematics, from simple complex-number arithmetic to group theory and number theory. The DFT is obtained by decomposing a sequence of values into components of different frequencies. This operation is useful in many fields (see discrete Fourier transform for properties and applications of the transform) but computing it directly from the definition is often too slow to be practical. An FFT is a way to compute the same result more quickly: computing the DFT of N points in the naive way, using the definition, takes O(N^2) arithmetical operations, while an FFT can compute the same DFT in only $O(N \log_2 N)$ operations.

5. Implementation

5.1 Detailed Description of Algorithm

Cooley - Tukey Algorithm:

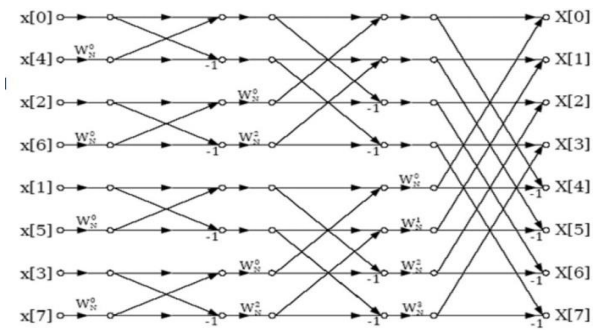
In 1965, James Cooley and John Tukey introduce the FFT to public, and reduced the Computation cost of DFT from O(N^2) to O($N\log_2 N$).

According to the article on *Numerical recipes* website, Cooley and Turkey's idea Divided the DFT computation into two small parts, and the function is shown below.

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

-eq.5

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($Nlog_2N$). According to this algorithm, for an eight point sequence input signal, and we start with 2 point Fourier transforms. Then, we get 4 points, and 8 points Fourier transforms.



8 points Fourier transforms fig5.1.1

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 sub problems of the same (or related) type (divide), until these become simple enough to be solved directly (conquer). The solutions to the sub problems 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 sub problems 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.

5.2 System Description:

The below block diagram (fig 5.1.1) shows CUDA programming model is CPU as the host, GPU as the equipment end, this programming model by CPU and GPU work together to accomplish complex transaction processing, CPU is mainly responsible for the small amount of data of the serial arithmetic, GPU is mainly responsible for performing high speed thread parallel processing tasks, CPU has an independent memory address space host memory, GPU has an independent memory address space the device memory, in memory of the operation process needs to call the CUDA memory management functions, including open, release and initializing the memory space, as well as in the host side and the terminal equipment for data transmission. For the large-scale data and parallel operation part can consider this part of the work to the GPU to complete, run on the GPU CUDA parallel computing by the kernel function to achieve the kernel, the kernel function is not a complete program but the program can be executed in parallel steps. Therefore, a complete CUDA program is composed of a series of equipment of kernel functions parallel steps and the host side of serial processing steps to complete. CPU serial code complete work is mainly the preparation of data and the initialization of the device, to the greatest extent possible to improve the calculation speed, should try to avoid the data in CPU and GPU pass between the frequency, but because the GPU function is very limited, most of the work needs to be carried out in CPU. In a CUDA program, the host has mainly completed the following functions.

(1) CUDA starts, after allocating memory in CPU, then the memory space allocation is done in the device for the input data.

(2) The host copies the memory of the data to the memory of device.

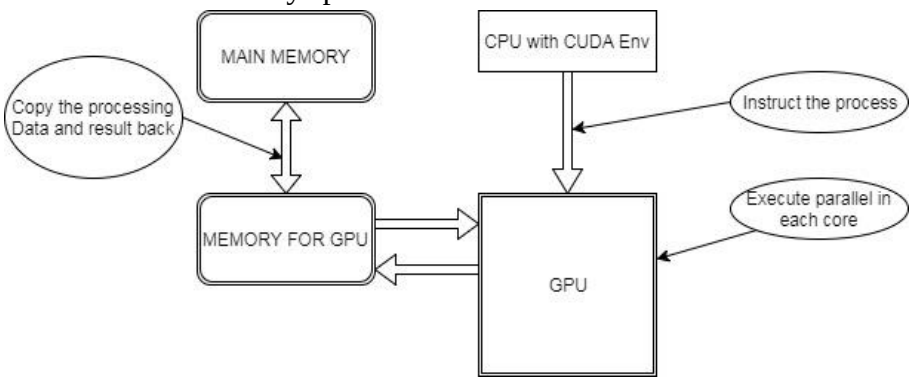
(3) In the GPU, the distribution of memory is used for storing the output data.

(4) Calling kernel function for calculation, the results into the memory of the corresponding region.

(5) CPU memory allocation, used to store the output data came from GPU.

(6) The memory of the results using CPU for data and other processing.

(7) The release of memory space in GPU.



Block Diagram Fig 5.2.1

5.3 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 realized 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.

5.4 System Architecture

Software:

A. C++ & CUDA toolkit 6.0+

Hardware:

A. Device- CUDA enabled GPU device with compute capability of 2.0+ higher.
B. Host - Windows 7 or later with minimum 2GB ram and 2ndgen dual core processor.
C. ACS GPU (GeForce GTX 780)
    - CUDA cores: 2304
    - Memory speed: 6.0 Gbps
    - Standard Memory : 3072 MB
    - Memory Interface Width : 384-bit
    - Memory Bandwidth (GB/sec) : 288.4

5.5 Code Screenshots

CUDA FFT KERNEL CODE:

```
/*CUDA FFT kernel*/
__global__ void FFT(cuFloatComplex *d_out, cuFloatComplex *d_in, long stage, long Bsize, long NeachL,cuFloatComplex *WnS)
{

    long k = NeachL*(blockIdx.x*blockDim.x+threadIdx.x);

    //cuFloatComplex Wn;
    if(k<N){
        //butterfly
        for(long as = 0; as<Bsize;as++){
            if(k+as+NeachL/2<N){

                d_out[k+as]=cuCaddf(d_in[k+as],cuCmulf(WnS[as*N/NeachL],d_in[k+NeachL/2+as]));
                d_out[k+as+NeachL/2]=cuCsubf(d_in[k+as],cuCmulf(WnS[as*N/NeachL],d_in[k+as+NeachL/2]));
__syncthreads();
d_in[k+as]=make_cuFloatComplex(cuCrealf(d_out[k+as]),cuCimagf(d_out[k+as]));
                d_in[k+as+NeachL/2]=make_cuFloatComplex(cuCrealf(d_out[k+as+NeachL/2]),cuCimagf(d_out[k+as+NeachL/2]));

            }
        }

    }
```

Simple FFT series code:

```
/* simple FFT  */
t* FFT_simple(t* x, long n) {
    t* X2 = (t*) malloc(sizeof(struct complex_t) * n);
    t * d, * e, * D, * E;
    long k;

    if (n == 1) {
        X2[0] = x[0];
        return X2;
    }

    e = (t*) malloc(sizeof(struct complex_t) * n/2);
    d = (t*) malloc(sizeof(struct complex_t) * n/2);
    for(k = 0; k < n/2; k++) {
        e[k] = x[2*k];
        d[k] = x[2*k + 1];
    }

    E = FFT_simple(e, n/2);
    D = FFT_simple(d, n/2);

    for(k = 0; k < n/2; k++) {
        /* Multiply entries of D by the twiddle factors e^(-2*pi*i/N * k) */
        D[k] = complex_mult(wnn[k*N/n], D[k]);
    }

    for(k = 0; k < n/2; k++) {
        X2[k]       = complex_add(E[k], D[k]);
        X2[k + n/2] = complex_sub(E[k], D[k]);
    }

    free(D);
    free(E);
    countE++;
    return X2;
}
```

6. Experiments & Results

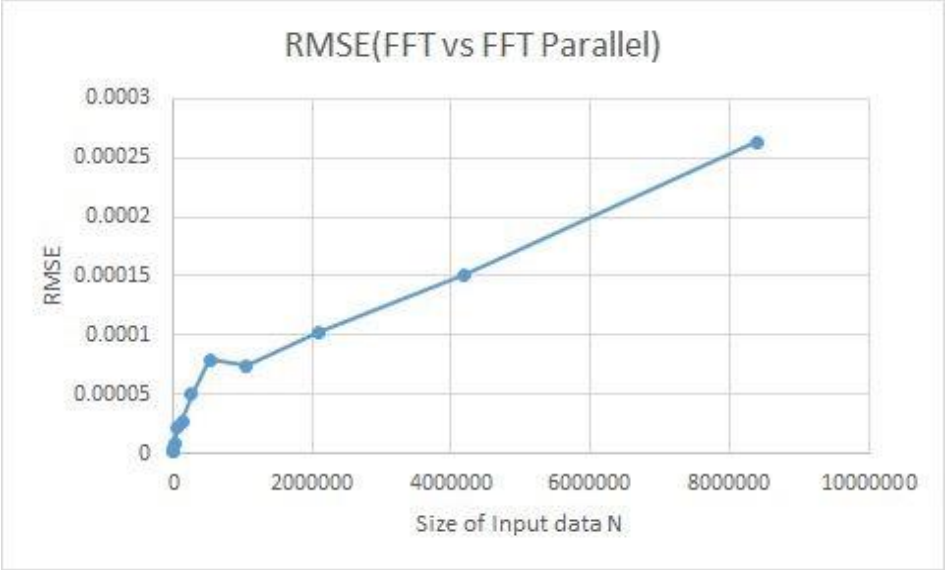
According to the algorithm, input of this system should be N value to the power of 2. We'll be testing this algorithm with different kinds of N values for both series & parallel.

A. As part of this project, verification of the implemented code might be done by calculating the RMSE (Root mean- square error) value for both series and parallel outputs.
B. Reliability can be decided by comparing both the execution/run times.

Below table(6.1) contains all the results of the experiments conducted with different N values.

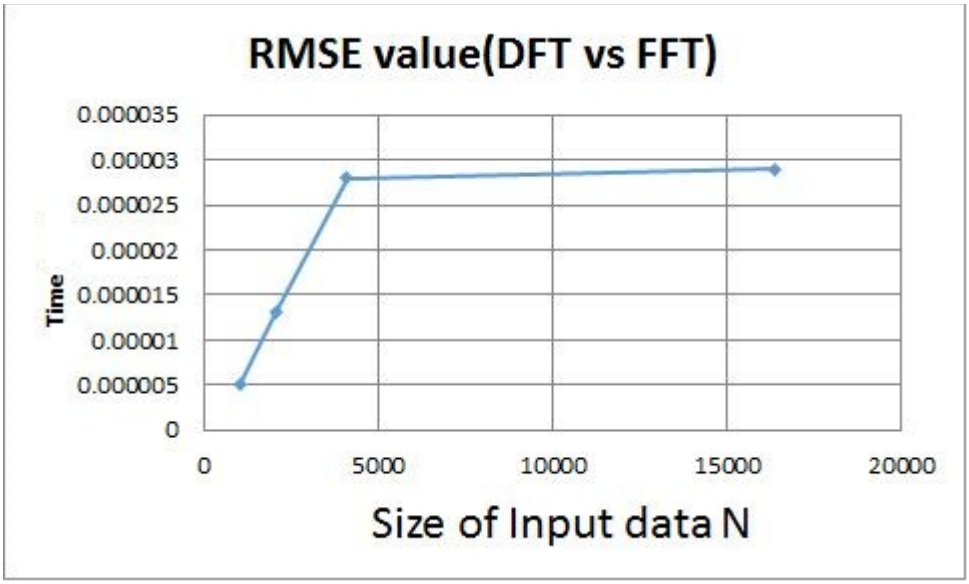
N	DFT(second)	FFT Serial(second)	FFT Parallel (second)	RMSE value (FFT serial v DFT)	RMSE value (FFT Parallel v FFT serial)
512	0.045761	0.000213	0.000064	0.000002	0.000001
1024	0.175117	0.000447	0.000070	0.000005	0.000002
2048	0.607248	0.001022	0.000073	0.000013	0.000003
4096	2.034476	0.002183	0.000078	0.000028	0.000002
16384	31.012565	0.019998	0.000088	0.000029	0.000009
65536	-	0.038309	0.000096	-	0.000022
131072	-	0.087167	0.000099	-	0.000027
262144	-	1.168698	0.000108	-	0.000050
524288	-	13.514978	0.000116	-	0.000079
1048576	-	70.937331	0.000122	-	0.000074
2097152	-	115.540531	0.000146	-	0.000103
4194304	-	370.237195	0.001162	-	0.000151
8388608	-	497.650408	0.001397	-	0.000263

Results Table (6.1)



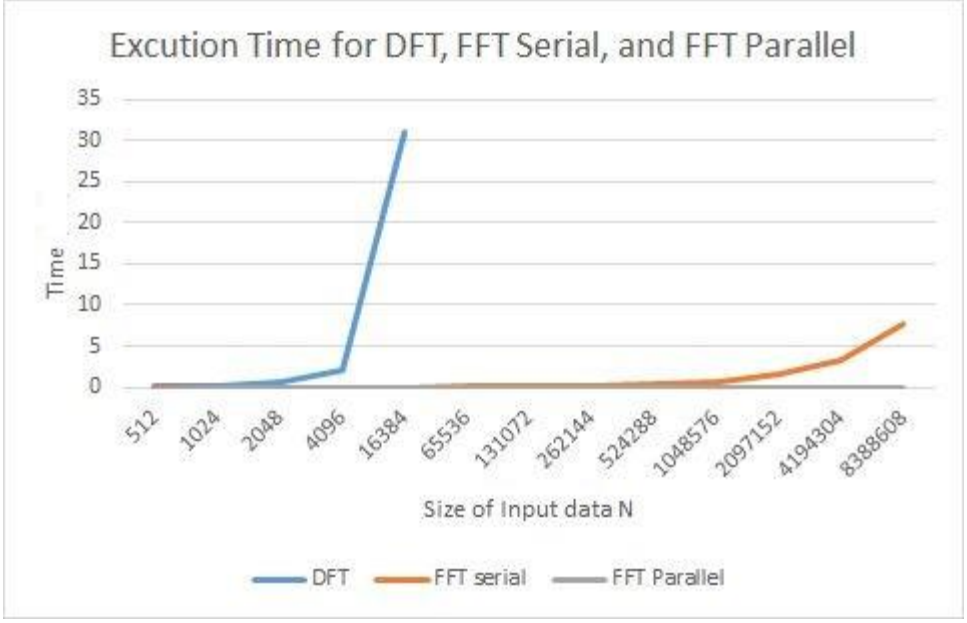
RMSE FFT vsFFT -P Graph 6.2

As you can see, RMSE value increases linearly but on having a closer look at the graph 6.2, it can be seen that the magnitude of increase in RMSE is very small.



RMSE DFT vs FFT Graph 6.3

It is interesting to see that RMSE value increased sharply initially but increased rather slowly afterwards in graph 6.3. However, it is interesting to note that magnitude increase in RMSE value still remains very less.



Execution Time for DFTvs FFT-S vs FFT-P Graph 6.4

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.

7. Analysis

In summary, CUDA allows for massive parallelism of computations allowing for much faster processing time for large number of operations. For processes such as matrix multiplication or in this case convolution it is perfect. CUDA is simple to implement in parallel with sequential programming. For small sets of instructions use the individually faster CPU and when you need to do lots of computation for large data sets, offload the process to the GPU. CUDA comes with an easy to use and decently documented toolkit for both Windows and Linux and with Nvidia GPU's becoming more prevalent in most systems, there is going to be even more applications where CUDA will greatly aid processing.

8. 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 our Prof Christopher Henry for his guidance & support. I would like to thank my friends who helped in developing this project.

9. References

[1] Naga K. Govindaraju, Brandon Lloyd, Yuri Dotsenko, Burton Smith, and John Manferdelli. High performance discrete fourier transforms on graphics processors. In Proceedings of the 2008 ACM/IEEE conference on Supercomputing, SC '08, pages 2:1{2:12, Piscataway, NJ, USA, 2008.IEEE Press.

[2] An Algorithm for the machine calculation of Complex Fourier Series. By James W. Cooley and John W. Turkey. Mathematics of Computation, Vol 19,No. 90.(Apr.,1965), pp.297-301.

[3] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. Proceedings of the IEEE, 93(2):216{231, 2005. Special issue on \Program Generation, Optimization, and Platform Adaptation".

[4] BAILEY, D. H. 1988. A High-Performance FFT Algorithm for Vector Supercomputers, International Journal of Supercomputer Applications 2, 1, 82–87.

[5] T. Jansen, B. von Rymon-Lipinski, N. Hanssen, and E. Keeve, “Fourier volume rendering on the GPU using a split-stream-FFT,” in Proceedings of the Vision, Modeling, and Visualization Conference 2004, 2004, pp.395–403

Web References:

[1] http://en.literateprograms.org/Cooley-Tukey_FFT_algorithm_(C)

[2] https://jakevdp.github.io/blog/2013/08/28/understanding-the-fft/

[3] http://docs.nvidia.com/cuda/cufft/

[4] https://en.wikipedia.org/wiki/Discrete_Fourier_transform