Parallelization of Fast Fourier transformation algorithm on the GPU

Halilović Kemal
department of Computing and Informatics
Faculty of Electrical Engineering
University of Sarajevo
Sarajevo, Bosnia and Herzegovina
khalilovic3@etf.unsa.ba

Mehmedović Faris
department of Computing and Informatics
Faculty of Electrical Engineering
University of Sarajevo
Sarajevo, Bosnia and Herzegovina
fmehmedovi1@etf.unsa.ba

Memić Miralem
department of Computing and Informatics
Faculty of Electrical Engineering
University of Sarajevo
Sarajevo, Bosnia and Herzegovina
mmemic2@etf.unsa.ba

Omić Kenan
department of Computing and Informatics
Faculty of Electrical Engineering
University of Sarajevo
Sarajevo, Bosnia and Herzegovina
komic1@etf.unsa.ba

***Abstract*—This paper describes the fast Fourier transform, it's theoretical setting with the mathematical formulation and application on hardware platform. An analysis of this topic is performed using the literature in this research area. An analysis is performed using technologie suitable for parallel implementation of fast Fourier transform calculations and the reasons why the CUDA platform was chosen are explained. Finally, the expected goals of the work are described.**

***Keywords*—fast Fourier transform, CUDA platform, parallel implementation**

I. Introduction

The Fourier transform has many applications in science and engineering. It is used in a wide range of areas such as digital signal processing, voice recognition and so on. Discrete Fourier Transform (DFT) is a special type of Fourier transform. If the Discrete Fourier Transform is implemented straightforward, the time complexity is $O(n^2)$. In addition, this approach is not the best in practice, as it would be more suitable to use the Fast Fourier Transform (FFT) with time complexity $O(nlog(n))$, which is based on fewer mathematical operations [1]. In this paper algorithm for computing FFT with high performance on graphics processing units (GPUs) is presented.

GPU architecture efficiently processes vector data (a series of numbers) and is often referred to as vector architecture. With GPUs, more space is allocated for computing and less for caching and control. As a result, GPU hardware explores less parallelism at the instruction level and relies on software parallelism to achieve performance and efficiency. Performance is achieved by multi-threaded execution of large and independent data, thus amortizing costs more easily controls and smaller caches [2].

In this paper NVIDIA's CUDA API is targeted, though many of the concepts have broader application. The rest of the paper is organized as follows. After discussing related work in Section II an overview of FFT computation is presented in Section III. The parralel implementation is presented in Section IV and comparison results of the algorithm on different GPUs were presented in Section V. Results are compared with other FFT implementation and concluded with some ideas for future work in Section VI.

II. Related work

Due to the important role of Fast Fourier Transform in various scientific fields, great efforts have been made to optimize it. FFT is often used to process images and videos in real-time so the speed of calculation is essential.

In paper [3], two approaches to storing complex numbers are considered: storing an ordered pair of real and imaginary parts in one buffer and storing in two buffers in which real and imaginary parts are individually arranged, where the first approach proved to be faster.

Various tests have shown that the higher the input data the more noticeable the parallelization of FFT is. Authors in [4] have compared FFT variants with and without parallelization, where the results showed that the performance did not differ in images with 512x512 resolution. However, when processing, images had a resolution of 1024x1024, the version with parallelization had over 4x faster execution time. In papers [5] and [6], the cuFFT library from CUDA Toolkit is used to make the most of the NVIDIA GPU.

III. Fast Fourier transform

Discrete Fourier Transform is a special type of Fourier transform, which requires input discrete function and that nonzero values have finite length [1]. The input function can be a finite sequence of real or complex numbers, thus the DFT is ideal for processing information stored in computers. But, if the data is continuous, the data has to be sampled when DFT is used. There is a possibility that if the sampling interval is too wide, it may cause the aliasing effect, however, if it's too narrow, the size of the digitalized data might be increased [1]. The definition is expressed as follows:

The given sequence $x(n)$ for $k = 0, 1, 2, ..., N$ is transformed into the sequence $X(k)$ by DFT via the formula (1):

$$X(k) = DFT_N\{x(n)\} = \sum_{n=0}^{N-1} x(n)e^{\frac{-2\pi i}{N}kn} \tag{1}$$

The basic idea of the fast Fourier transform is to apply divide and conquer. The coefficient vector of the polynomial is divided into two vectors, recursively compute the DFT for each of them, and combine the results to compute the DFT of the complete polynomial. So let there be a polynomial $A(x)$ with degree $n-1$, where n is a power of 2, and $n > 1$ [7]:

$$A(x) = a_0 + a_1x + a_2x^2 + ... + a_nx^n = \sum_{i=0}^{n} a_ix^i \tag{2}$$

Polynomial multiplication is formulated as a divide and conquer algorithm with the following steps for a polynomial $A(x)\ \forall x \in X$:

1) Algorithm is divided into two smaller polynomials, the one containing only the coefficients of the even positions, and the one containing the coefficients of the odd positions:

$$A_{even}(x) = \sum_{k=0}^{\lceil \frac{n}{2}-1 \rceil} a_{2k}x^k = \langle a_0, a_2, a_4, ...\rangle \tag{3}$$

$$A_{odd}(x) = \sum_{k=0}^{\lfloor \frac{n}{2}-1 \rfloor} a_{2k+1}x^k = \langle a_1, a_3, a_5, ...\rangle \tag{4}$$

2) Recursively conquer $A_{even}(y)$ for $y \in X^2$ i $A_{odd}(y)$ for $y \in X^2$, where $X^2 = \{\ x^2 \mid x \in X\ \}$.
3) Combine the terms. $A(x) = A_{even}(x^2) + x \cdot A_{odd}(x^2)$ for $x \in X$.

However, the recurrence realtion for this algorithm is

$$T(n, |X|) = 2 \cdot T(\frac{n}{2}, |X|) + O(n + |X|) = O(n^2) \tag{5}$$

which is no better than before.

Better performances can be achieved if X is reduced, or $|X| = 1$ (base case), or $X^2 = \frac{|X|}{2}$ and X^2 is (recursively) reducing. Then the recurrence relation is of the form

$$T(n) = 2 \cdot T(\frac{n}{2}) + O(n) = O(nlog(n)). \tag{6}$$

Reduction of time complexity is achieved with the "Divide and Conquer" algorithm, which allows to write formula (1) in the following format [3]:

$$T(2k) = DFT_{N/2}\{x(n)\ \ + x(n + N/2)\} \tag{7}$$

$$T(2k) = DFT_{N/2}\{x(n)\ \ + x(n + N/2)\} \tag{8}$$

where $k = 0, 1, ...N/2 - 1$ and $W_N^n = exp(-i2\pi n/N)$. Then decomposition formulas 7 and 8 can be applied for $N = 2^p$, where p is an integer.

IV. Parallel implementation

Having identified the hotspots and having done the basic exercises to set goals and expectations, the problem that is left is to parallelize the code. Design of the FFT algorithm requires a certain amount of refactoring to expose it's inherent parallelism nature. CUDA family of parallel programming languages aims to make the expression of this parallelism as simple as possible and in the same time enabling operation on CUDA-capable GPUs which are designed for maximum parallel throughput [10].

The main goal of project is to implement Fourier transform of a function on a GPU in order to reduce its execution time. Fast Fourier Transform is an optimized version of Discrete Fourier Transform, which works on the principle of divide and conquer and also it decreases the execution time with respect to DFT [2]. By refactoring the code, the aim is to expose the parallelism and substantially reduce the execution time. With that being said, the key spots of the algorithm which obtained the parallelisation property and enabled the operation on CUDA-capable GPUs were CUDA fft kernel (Algorithm 1) and CUDA bit_reverse_copy kernel (Algorithm 2). When these kernels are launched, number of parallel blocks and threads that the runtime system should use, are specified. A collection of these parallel blocks is a grid, where each block in a grid has its own number of threads that the kernels will run on. By having the kernels run in parallel, it significantly reduces the amount of time it takes to run the FFT on input functions with large amounts of data.

Algorithm 1: CUDA FFT kernel being refactored to enable the parallelism nature of the algorithm

```
Function fft (Argument *A, Argument m):
    $th$ = $blockIdx.x$ * $blockDim.x$ + $threadIdx.x$
    $k$ = $th/(m/2)$
    $j$ = $th\%(m/2)$
    $w$ = complexp(((2 * CUDART_PI) / $m$) * $j$)
    $t$ = cuCmul($w$, $A[k + j + m/2]$)
    $u$ = A[k + j]
    $A[k + j]$ = cuCadd($u$, $t$)
    $A[k + j + m/2]$ = cuCsub($u$, $t$)
```

Algorithm 2: CUDA bit_reverse_copy kernel being refactored to enable the parallelism nature of the algorithm

```
Function bit_reverse_copy(Argument size, Argument *A, Argument *R):
    $n$ = $blockIdx.x$ * $blockDim.x$ + $threadIdx.x$
    if $n > size$ then
        Return
    end
    $s$ = $(int)log2((double)size)$
    $revn$ = 0
    for $(int\ i = 0;\ i < s;\ i++)$ do
        $revn+ = ((n >> i)\&1) << ((s-1)-i)$
    end
    $aux$ = $A[n]$
    $R[revn]$ = $aux$
```

As seen in Alg. 1. i 2. the functions fft and bit_reverse_copy are set up as kernel functions, in which the value of a specific variable is determined by the current block and thread in which the function is running in. For fft this is the variable th, while for bit_reverse_copy it is n. In both cases, in order to find the number of the current thread in which the function is running, the calculation of:

$$n = blockIdx.x \cdot blockDim.x + threadIdx.x \qquad (9)$$

is used, where $blockIdx.x$ and $threadIdx.x$ are the current block and thread the functions is in, while $blockDim.x$ is the number of threads in each block. The required number will change based on the number of elements in the input sequence. Now that there is a different value in every parallel version of the functions, implementation of the FFT can be used to determine the index of the sequence it is are working on. As the algorithm is working on individual parts of the sequence there is no need to synchronize the threads at the end. It is important to add that variables, used to specify the size of the launch, are defined as dim3 type, which represents a three-dimensional tuple [10].

V. Results

FFT algorithm is tested on three different NVIDIA GPUs: GTX 980, GTX 1650 Ti and GTX 1050 Ti. The specifications for these GPUs are summarized in Table I. One of the key difference between the GPUs is the memory bandwidth. The GTX 980 has the most bandwidth and the GTX 1650 Ti has the least. The GTX 980 also has more multiprocessors, which gives it the highest peak performance, when execution time of Fast Fourier transformation algorithm is analyzed. Thus the GPUs ability to manage larger amounts of information is utilized, extending to milions of samples. Because each processor in such system is assigned to perform a specific function, it can perform its task, pass the instruction set on to the next processor, and begin working on a new set of instructions [9].

In the results, it's noticeable that the results of the experiment for over a million samples are approximately the same. This is an indication to us that during the execution of arithmetic-logical operations on the graphics cards there was saturation. Although parallelization has been performed, due to the lack of better graphics card specifications it will not work as expected with millions of samples.

Table II
TABLE REPRESENTING EXECUTION TIME OF FFT ON GPUS

Number of samples	Execution time of Fast Fourier transformation measured in miliseconds		
	NVIDIA GeForce GTX 980	NVIDIA GeForce GTX 1650 Ti	NVIDIA GeForce GTX 1050 Ti
2^9	0.800416	0.609888	1.490944
2^{10}	1.130592	1.221664	2.507776
2^{11}	1.228992	1.353696	2.763776
2^{12}	1.340256	1.489664	3.152896
2^{14}	1.53136	2.162752	3.415424
2^{16}	3.710272	3.217888	7.834624
2^{17}	7.577696	6.41168	15.92832
2^{18}	15.02586	13.18976	32.05594
2^{19}	34.28919	27.47283	68.08167
2^{20}	72.39171	59.49994	135.8438
2^{21}	147.2	236.8	540.8

The input and output arrays were aligned to a multiple of the cache line width. Performances for computational throughput are reported in GB/s, which are computed as [10]:

$$Effective\_bandwith = ((B_r + B_w)/10^9)/time \qquad (10)$$

where B_r is the number of bytes read per kernel, B_w is the number of bytes written per kernel, and $time$ is the execution time of parallelized FFT algorithm in seconds.

Table III
TABLE REPRESENTING EFFECTIVE BANDWIDTH OF FFT ON GPUS

Number of samples	Effective bandwith of Fast Fourier transformation measured in GB/s		
	NVIDIA GeForce GTX 980	NVIDIA GeForce GTX 1650 Ti	NVIDIA GeForce GTX 1050 Ti
2^9	0.003026	0.001887	0.001637
2^{10}	0.008251	0.005789	0.002639
2^{11}	0.01492	0.011351	0.004384
2^{12}	0.029055	0.021701	0.012931
2^{14}	0.098851	0.072226	0.042515
2^{16}	0.172804	0.17671	0.078487
2^{17}	0.17073	0.186167	0.077887
2^{18}	0.18097	0.198893	0.081006
2^{19}	0.164397	0.193567	0.077813
2^{20}	0.163784	0.190368	0.076007
2^{21}	2.037156	1.809524	4.88375

In next graph (Fig. 1) line segments are observed that represent effective bandwidth for different GPUs for number of samples, which are in range ($2^9 - 2^{20}$).

Table I
GPUS USED IN EXPERIMENT

GPU	Base Clock	Multiprocessors	Memory Bus	Memory Size	Bandwidth
GTX 980	1127 MHz	22	256 bit	4 GB	224.4 GB/s
GTX 1650 Ti	1350 MHz	16	128 bit	4 GB	128 GB/s
GTX 1050 Ti	1350 MHz	6	128 bit	4 GB	192.0 GB/s

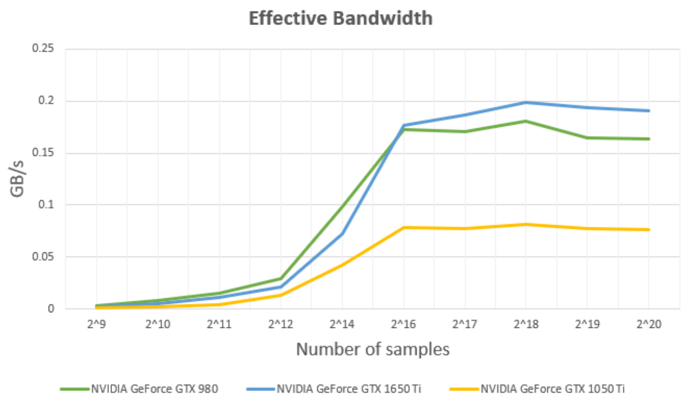

Fig. 1. Graph representing effective bandwidth of FFT on GPUs

NVIDIA GeForce GTX 980 has the best effective bandwidth (presented in Table III and Fig. 1), i.e. the least time access to data by the program, which is expected, because it has the largest memory bus of 256 bits and has 22 multiprocessors. These are the demonstrated results for measured bandwidth, which is a measure of data throughput. Another metric very important to performance is computational throughput, which is computed as [10]:

$$Computational\_throughput = ((2 * N)/10^9)/time \quad (11)$$

where N is the number of elements in the experiment, and $time$ is the execution time of parallelized FFT algorithm in seconds. The comparison of effective bandwith results for FFT on selected hardware platforms are presented in Table IV.

Table IV
TABLE REPRESENTING COMPUTATIONAL THROUGHPUT OF FFT ON GPUS

Number of samples	Computational throughtput of Fast Fourier transformation measured in GB/s		
	NVIDIA GeForce GTX 980	NVIDIA GeForce GTX 1650 Ti	NVIDIA GeForce GTX 1050 Ti
2^{9}	0.000504	0.000315	0.000273
2^{10}	0.001375	0.000965	0.00044
2^{11}	0.002487	0.001892	0.000731
2^{12}	0.004843	0.003617	0.002155
2^{14}	0.016475	0.012038	0.007086
2^{16}	0.028801	0.029452	0.013081
2^{17}	0.028455	0.031028	0.012981
2^{18}	0.030162	0.033149	0.013501
2^{19}	0.027399	0.032261	0.012969
2^{20}	0.027297	0.031728	0.012668
2^{21}	0.339526	0.301587	0.081396

In next graph (Fig. 2) line segments are observed that represent computational throughput for different GPUs for number of samples, which are in range $(2^9 - 2^{20})$.

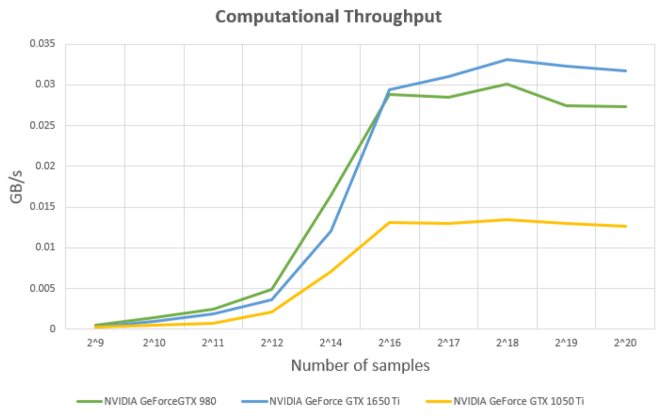

Fig. 2. Graph representing computational throughput of FFT on GPUs

As is seen in Table IV and Fig. 2, once again the best performance metrics are achieved with NVIDIA GeForce GTX 980, which has far better specifications, which are presented in Table I. CUDA fft functon reads 16 bytes per element computed, but performs only a single multiply-add instruction, so it's pretty clear that it will be bandwidth bound, and so in this case, bandwidth is the most important metric to measure and optimize.

This paragraph shows the comparison of the testing results of FFT algorithm presented in this paper with the results provided in [11]. Representative for the testing results is going to be NVIDIA GeForce GTX 980, since analyzing all results together, indicate that the parallel FFT algorithm has had it's best overall performance on GTX 980. Orlando Ayala, Lian-Ping Wang in their paper *Parallel implementation and scalability analysis of 3D Fast Fourier Transform using 2D domain decomposition* [11], did the analysis on Bluefire – IBM cluster, with different environment of development and code configuration. Comparing execution time of Fast Fourier transformation (Table V) shows that algorithm ran on GTX 980 has much better performance than algorithm ran on Bluefire – IBM cluster at NCAR, an IBM Power 575 cluster (4064 POWER6 processors running at 4.7 GHz, bandwidth of 20 GB/s each direction, switch latency of 1.3 μs at peak performance).

Table V
TABLE REPRESENTING EXECUTION TIME OF FFT ON GTX 980 AND BLUEFIRE – IBM CLUSTER

Number of samples	Execution time of Fast Fourier transformation measured in miliseconds	
	NVIDIA GeForce GTX 980	Bluefire - IBM cluster
2^9	0.800416	1527.0
2^{12}	1.134256	1514.0
2^{18}	15.02586	1542.0
2^{22}	147.2	1520.0

The reason why the GTX 980 has overall better performance output than the IBM cluster lies in the code configuration. Algorithm ran on GTX 980 is greatly simplified if locations of inputs/outputs for butterfly operators are rearranged. Special interest is the problem of devising an in-place algorithm that overwrites its input with its output data using only $O(1)$ auxiliary storage. The most well-known reordering technique involves explicit bit reversal for in-place radix-2 algorithms which is used in this paper. The results indicate that algorithms characterized by unified structures (having identical stages) are better suited for GPU implementations. In addition it can be concluded that structures that are simpler in the sense of indices calculations are also more computationally efficient.

VI. CONCLUSION

Application of Fast Fourier algorithm is presented on several GPUs for efficiently performing parallelization. This algorithm provides the best performance for a given input, because the speed of calculation is essential in real-time. We address the memory bandwidth, computational throughput and execution time issues. Comparasion is made using NVIDIA's CUDA API and three different NVIDIA GPUs: GTX 980, GTX 1650 Ti and GTX 1050 Ti. Results of this paper indicate a better performance metrics of FFT algorithm on NVIDIA GeForce GTX 980. We also address execution time issues. There are several avenues for future work. Interesting direction is mapping the FFT algorithms onto multiple GPUs, one reason to create a system like this presently is that current GPU devices are manufactured with a fixed amount of memory that is often limited in size and bandwidth compared to the memory allocation available to the CPU. Aggregating the resources from multiple GPUs can potentially solve this [12].

REFERENCES

[1] Liu, B., *Parallel Fast Fourier Transform*, Studies in Parallel and Distributed System, Lecture notes, Massey University, Palmerston North, New Zealand, 2021.

[2] Intel, *"Compare Benefits of CPUs, GPUs, and FPGAs for Different oneAPI Compute Workloads"*, Published: 04/15/2021. Accessed on: January, 2022. [Online].

[3] Puchała, D., Stokfiszewski, K., Yatsymirskyy, M., Szczepaniak, B., *Effectiveness of Fast Fourier Transform Implementations on GPU and CPU*, in 2015 16th International Conference on Computational Problems of Electrical Engineering (CPEE). IEEE, 2015, pages 162–164.

[4] Haque, Mohammad Nazmul, and Mohammad Shorif Uddin. *Accelerating Fast Fourier Transformation for Image Processing using Graphics Processing Unit*, 2011.

[5] Govindaraju, Naga K., et al. *High performance discrete Fourier transforms on graphics processors*, SC'08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing. IEEE, 2008.

[6] Mejia-Parra, Daniel, et al. *Fast simulation of laser heating processes on thin metal plates with FFT using CPU/GPU hardware*, Applied Sciences 10.9.2020: 3281.

[7] Leiserson C., Demaine E., *6.046J Introduction to Algorithms (SMA 5503)*, Massachusetts Institute of Technology: MIT OpenCourseWare, https://ocw.mit.edu., Fall 2005.

[8] Guide, Design. *CUDA C Best Practices Guide*, NVIDIA, July 2013.

[9] Britannica, The Editors of Encyclopaedia. *"Multiprocessing"*. Encyclopedia Britannica, Accessed January, 2022 [Online].

[10] Harris, Mark. *How to Implement Performance Metrics in CUDA C/C++*, nVIDIA Developer Zone, 2012.

[11] Ayala, O., Wang, L. P., *Parallel implementation and scalability analysis of 3D fast Fourier transform using 2D domain decomposition*. Parallel Computing, 39(1), 58-77, 2013.

[12] Nandapalan, Nimalan. *Use of multi-GPU systems for large FFTs: with applications in ultrasound simulations*, 2013.