Teraflop FFT computation for OFDM using GPGPU

Selcuk Keskin[1] and Taskin Kocak[2]
Department of Computer Engineering, Bahcesehir University, Istanbul 34353, Turkey
Email: {selcuk.keskin[1],taskin.kocak[2]}@eng.bau.edu.tr

***Abstract*—In this paper, we propose to use General Purpose Computing on Graphical Processing Unit (GPGPU) to increase the required throughput for multi-gbps Orthogonal Frequency Division Multiplexing (OFDM) system. A parallel approach for a 512-point Fast Fourier Transform (FFT) is investigated and implemented by a general purpose parallel computing platform called CUDA. Several optimization methods are applied and efficient data structures are used to achieve 1 teraflop FFT computation throughput.**

I. Introduction

Next generation wireless and wired digital communication systems have started to choose OFDM technology because of its high speed data rates, high spectral efficiency, high quality service and robustness. The main idea behind OFDM is to split the data stream to be transmitted into N parallel streams of reduced data rate and to transmit each of them on a separate subcarrier. These carriers are made orthogonal by appropriately choosing the frequency spacing between them. The advantage of applying OFDM in high data rate communication systems is a relatively long symbol duration compared with the delay spread of the channel, in which inter-symbol interference (ISI) can be eliminated by adding a guard interval [1]. The realization of OFDM technique over any communication system faces some difficulties.

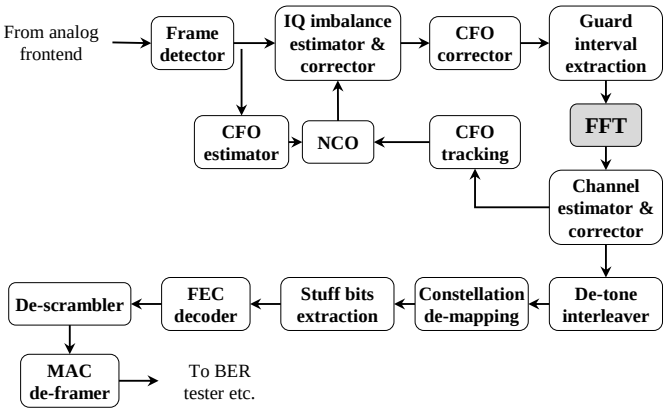

Fig. 1: Receiver diagram of the OFDM system

OFDM is a method of encoding digital data on multiple carrier frequencies [2]. Basic block diagram of the OFDM system in a receiver mode can be seen in Fig. 1. FFT and FEC decoding computations are the most time consuming parts. Therefore, any performance gains in these blocks can potentially improve the throughput of the whole system significantly. The acceleration of algorithms such as these is of critical importance for high throughput communications systems [3].

Increasing data traffic and multimedia services in recent years have paved the way for the development of optical transmission methods to be used in high bandwidth DSP-assisted optical communications systems. In this paper, we present an FFT computation process on GPU instead of DSPs, that can support the high-throughput requirement.

GPUs have only been used for 3D graphics rendering in the first years of their evolution. By increasing technology of GPUs, they offer high performance of general purpose processing by executing thousands of threads simultaneously. A GPU provides a parallel architecture, which combines raw computation power with programmability [4]. GPU provides extremely high computational throughput by employing many cores working on a large set of data in parallel. Compute Unified Device Architecture (CUDA), developed by NVIDIA, is a widely used programming approach in massively parallel computing applications [5].

II. FFT algorithm design

An FFT process computes the Discrete Fourier Transform (DFT) for a set of signal data and it produces exactly the same result as evaluating the DFT definition directly; the only difference is that an FFT is much faster. The DFT is obtained by decomposing a sequence of values into components of different frequencies. An FFT is a way to compute the same result more quickly: computing the DFT of N points in naive way. The difference in speed can be enormous, especially for long data sets where N may be in the thousands or over.

If we assume that $x_0 ... x_{N-1}$ are complex numbers, Eq. 1 defines the DFT. Evaluating this definition directly requires $O(N^2)$ operations: there are N outputs (X_k), and each output requires a sum of N terms.

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

The publication by Cooley and Tukey in 1965 of an efficient algorithm for the calculation of the DFT was a major turning point in the development of digital signal processing [6]. Then, various extensions and modifications were made to the original algorithm [7]. By far the most commonly used FFT is the Cooley-Tukey algorithm against the others like Prime-factor FFT algorithm [8], Bruun's FFT algorithm [9], Rader's FFT algorithm [10], and Bluestein's FFT algorithm [11].

The FFT butterfly operation which is the basic calculation element in the FFT process takes two complex points and

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

converts them into two other complex points. In the case of the 2-point (radix-2) Cooley-Tukey algorithm, the butterfly is simply a DFT of size-2 that takes two complex inputs (x_0, x_1) which are corresponding outputs of the two sub-transforms and gives two complex outputs (y_0, y_1) by using Eq. 2. The w^k is called twiddle factor. A twiddle factor in FFT algorithms, is any of the trigonometric constant coefficients that are multiplied by the data in the course of the algorithm.

$$\begin{aligned} y_0 &= x_0 + x_1 w^k \\ y_1 &= x_0 - x_1 w^k \end{aligned} \tag{2}$$

An N point signal is decomposed into N signals each containing a single point. Each stage of FFT uses an interlace decomposition, separating the even and odd numbered samples [12]. The DFT of a N-point sequence can be simply calculated from the two $N/2$-point DFT's of the even index terms $x_0, x_2 ... x_{N-2}$ and the odd index terms $x_1, x_3 ... x_{N-1}$, then those two results are combined to produce the DFT of the whole sequence [13]–[15]. This idea can then be performed recursively to reduce the overall runtime to $O(NlogN)$.

A. GPGPU Architecture

GPU provides extremely high computational throughput by employing many cores working on a large set of data in parallel. Compute Unified Device Architecture (CUDA), developed by NVIDIA, is a widely used programming approach in massively parallel computing applications [5]. CUDA comes with a software environment that allows developers to use C as a high-level programming language. CUDA C provides a simple path for users familiar with the C programming language to easily write programs for execution by the device. It consists of a minimal set of extensions to the C language and a runtime library. CUDA C extends C by allowing the programmer to define C functions, called kernels, that, when called, are executed N times in parallel by N different CUDA threads, as opposed to only once like regular C functions.

The NVIDIA GPU architectures consist of multiple stream multiprocessors (SM) which consist of pipelined cores and instruction dispatch units. During execution, each dispatch unit can issue a numerous wide single instruction multiple data (SIMD) instruction, which is executed on a group of cores. Although, CUDA provides the possibility to unleash GPU's computational power, several restrictions prevent programmers from achieving peak performance. The programmer should pay attention to the hardware-based aspects to achieve near-peak performance.

Pascal is NVIDIA's latest architecture for CUDA compute applications. Pascal retains and extends the same CUDA programming model provided by previous NVIDIA architectures such as Kepler and Maxwell. Like Maxwell, each GP102 SM provides four warp schedulers managing a total of 128 single-precision (FP32) and four double-precision (FP64) cores. A Pascal GP102 in TITAN X, used in this paper, implementation includes 28 SM units and six 64-bit memory controllers. The SM processing core architecture of TITAN X can be seen in Fig. 2. The six SMs are grouped into a Graphics Processing Cluster (GPC) which has one Raster Engine.

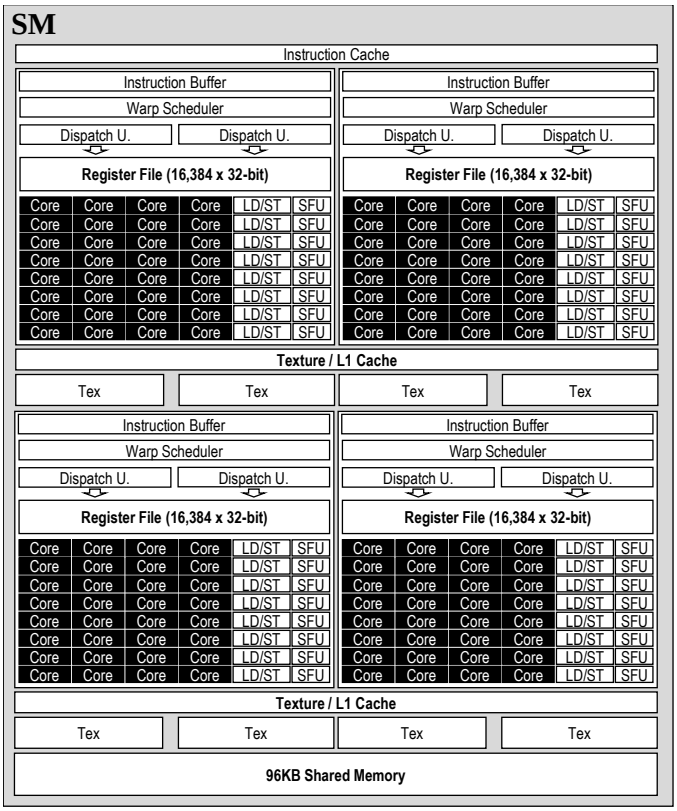

Fig. 2: Pascal GP102 Streaming Multiprocessor

Compared to previous chip architectures Kepler and Maxwell, the SMs memory hierarchy has also changed. Rather than implementing a combined shared memory/L1 cache block as in Kepler SMX, Pascal SM units feature a 96 KB dedicated shared memory, while the L1 caching function has been moved to be shared with the texture caching function. Global memory of GPU is an off-chip memory. Whole SM can access the global memory, but access time is the slowest.

III. GPGPU Based Parallel FFT Algorithm Design

Recent multi-gbps wireless communication specifications such as WirelessHD [16] and IEEE802.15.3c [17] use commonly 512 subcarriers in their OFDM processes. Hence, we also decided to design and implement a 512-point FFT in this paper. A 512-point FFT process can be performed as a 9-stage computation module, where a 2-point FFT is calculated in each stage. To achieve better performance values, we must optimize parallel FFT algorithm considering GPU device architecture. One of the major optimizations for a CUDA algorithm by considering device architecture is related with memory transfer operations and the other one is related with threads.

A. Thread Organization

To achieve the size of 512, we will use 256 independent threads ($256 \times 2 = 512$). By computing these threads in parallel

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

on different cores, parallel computing of the FFT process will be implemented as shown in Fig. 3. However, a kernel can be executed by multiple equally-shaped thread blocks. The input data will be transferred, as number of "$batchsize$" which represents number of FFTs to be executed together.

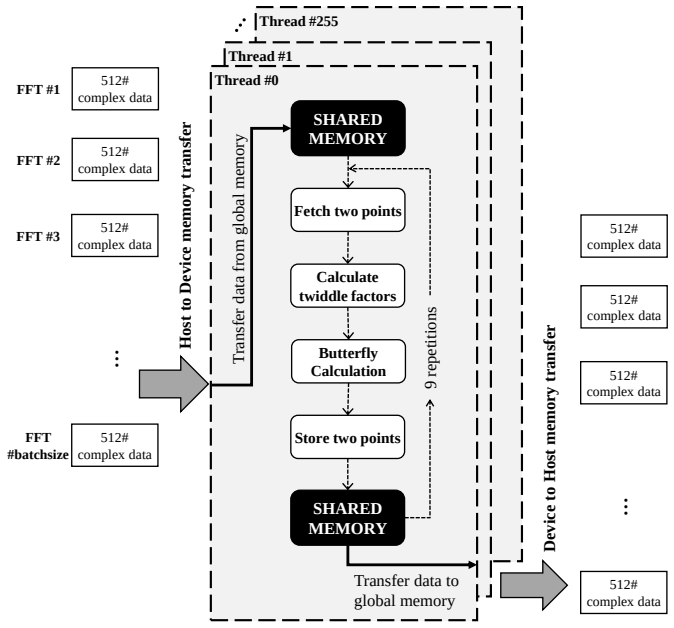

Fig. 3: Thread organization of the GPU parallel calculation process

The major steps of the each stage can be summarized as fetch two points from memory, calculate twiddle factors and implement butterfly operations for each radix and then store back to memory as shown in Fig. 4. The process is repeated 9 times to achieve 512-point FFT. Inputs of FFT process are transferred host to device(global) memory (H2D), and after the calculations, outputs on global memory are transferred from device to host memory (D2H).

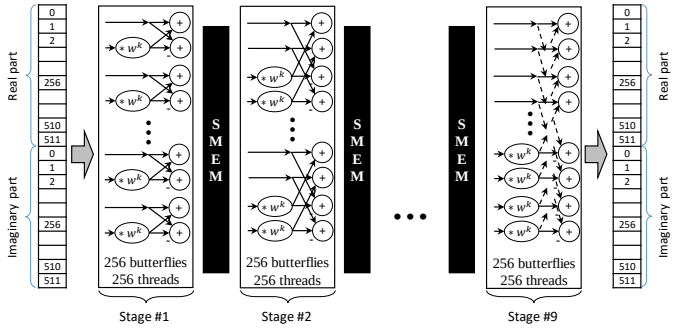

Fig. 4: 512-point FFT architecture with 9-stage

B. Reduction in Shared Memory Access

In the first-pass form of the algorithm, inputs were written to the shared memory from global memory as shown in Fig. 3. In the iteration, inputs were written to registers from shared memory and results were calculated with registers. After the iteration, outputs were read again from shared memory and written to global memory. To reduce shared memory access, we changed progress. Inputs are transferred from global memory to registers. In the iteration part, calculations are done and written to shared memory. After the calculation, the values are transferred from shared memory to registers for the next iteration. At the end of the iterations, outputs are already on the registers so these values are used to calculate outputs which are written to global memory. The new algorithmic flow for each thread can be seen in Fig. 5.

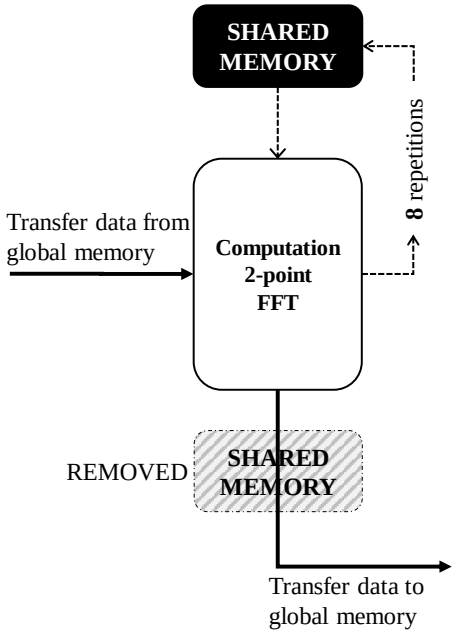

Fig. 5: Illustration of one thread's algorithmic flow after enhancement

IV. EXPERIMENTAL RESULTS

The designed Cooley-Tukey based parallel FFT computation algorithm, so called BauFFT (Bahcesehir University Fast Fourier Transform) is implemented with CUDA 8.0 environment and executed on NVIDIA TITAN X. All the experiments are conducted on a 3.20 GHz Intel Core i7-960 CPU with 12GB of memory.

The data copy time between host and device is excluded from our measurements. Different number of $batchsize$ is evaluated to show the effect on performance of multiple FFTs with the same sizes. The calculation time per FFT is reduced by increasing $batchsize$ as seen in Fig. 6. When $batchsize$ is 4096 and higher, the performance is nearly the same. A single 512-point FFT signal with 32-bit data representation is calculated within 23 ns computation time by using BauFFT code on Pascal architecture based TITAN X GPU. The computation throughput of the FFT block for peak

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

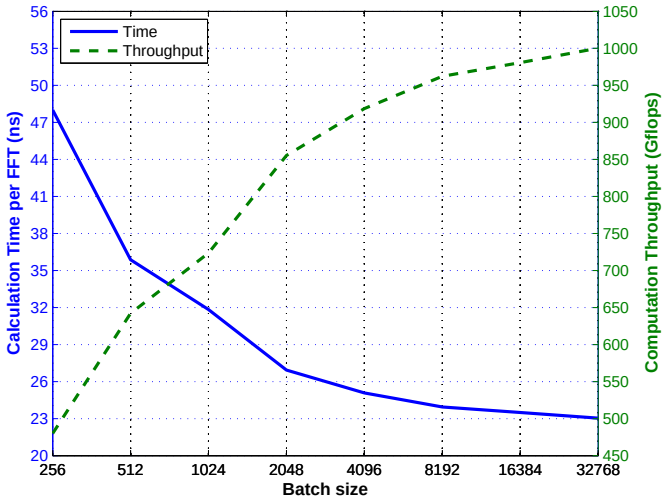

Fig. 6: The performance of the BauFFT code on GPU

performance becomes as

$$\frac{5 \times 512 \times 9}{23 \times 10^{-9}\, sec} = \frac{23040}{23 \times 10^{-9}\, sec} = 1\, TFlops \qquad (3)$$

Our GPU based FFT algorithm can be compared with NVIDIA's cuFFT algorithm. As we can see from Fig. 7, we perform better than cuFFT. Moreover, cuFFT is a closed source algorithm. Before calling cuFFT algorithm, a plan function is executed by using the transform size, data type and number of transforms. The process must be waited until this plan function is finished. In the other words, the next process in the OFDM system cannot be started. Considering to accelerate all processes in the OFDM system by GPU, only one kernel can be used to implement whole OFDM process without too much kernel initialization overhead for each process.

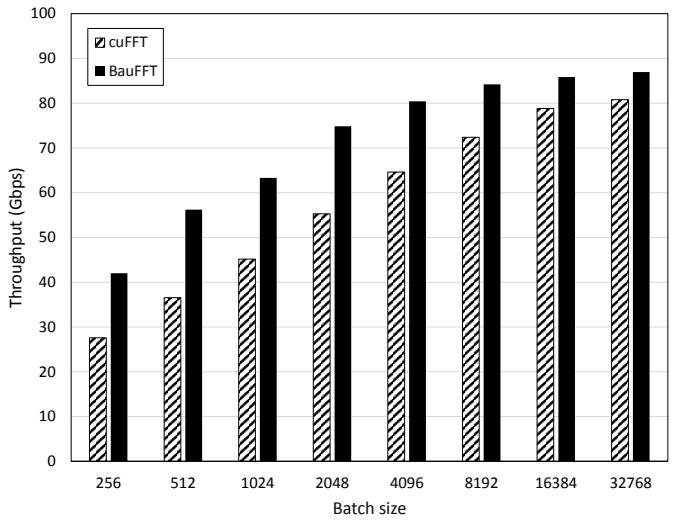

Fig. 7: The performance comparison between BauFFT and NVIDIA's cuFFT

V. Conclusion and Future Work

A GPGPU-based 512-point FFT algorithm is proposed in this paper. Global memory accesses are reduced and data representation is improved to use memory coalescing. The flow of BauFFT algorithm is modified to reduce passed of shared memory. As a result, our algorithm achieves over 1 TFlops FFT computation throughput.

The result shows that GPUs can efficiently calculate FFT in OFDM system. We would like to implement FEC decoding on GPU which is the other time consuming part of an OFDM receiver.

Acknowledgment

A part of this work is financially supported by KDDI R&D Laboratories Inc., Japan. As a member of CUDA Research Center, we also acknowledge the support of NVIDIA Corporation with the donation of the GPU used for this research.

References

[1] M. Yoshida and T. Taniguchi, "An LDPC-coded OFDM receiver with pre-FFT iterative equalizer for ISI channels," in *IEEE 61st Vehicular Technology Conference*, vol. 2, May 2005, pp. 767–772.

[2] V. Kanwar, H. Thakur, and N. Sharma, "Performance Evaluation of OFDM System under Various Modulation Techniques and Various Channels," *International Journal of Research in Engineering & Advanced Technology*, vol. 1, no. 3, pp. 1–5, 2013.

[3] C.-H. Peng, K.-T. Shr, M.-H. Lin, and Y.-H. Huang, "A baseband receiver for optical OFDM systems," in *International Symposium on VLSI Design, Automation and Test (VLSI-DAT)*, April 2011, pp. 1–4.

[4] J. B. Srivastava, R. Pandey, and J. Jain, "Implementation of Digital Signal Processing Algorithm in General Purpose Graphics Processing Unit (GPGPU)," *International Journal of Innovative Research in Computer and Communication Engineering*, vol. 1, no. 4, pp. 1006–1012, 2013.

[5] D. B. Kirk and W.-M. W.Hwu, *Programming Massively Parallel Processors*, 2nd ed. Morgan Kaufmann, 2012.

[6] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex Fourier series," *Mathematics of Computation*, vol. 19, no. 90, pp. 297–301, 1965.

[7] J. W. Cooley, P. A. W. Lewis, and P. D. Welch, "Historical notes on the fast Fourier transform," *Proceedings of the IEEE*, vol. 55, no. 10, pp. 1675–1677, 1967.

[8] I. J. Good, "The Interaction Algorithm and Practical Fourier Analysis," *Journal of the Royal Statistical Society, Series B*, vol. 20, pp. 361–372, 1958.

[9] G. Bruun, "z-Transform DFT filters and FFTs," *IEEE Trans. on Acoustics, Speech, and Signal Processing*, vol. 26, no. 1, pp. 56–63, 1978.

[10] C. Rader, "Discrete Fourier transforms when the number of data samples is prime," *Proceedings of the IEEE*, vol. 56, no. 6, pp. 1107–1108, 1968.

[11] L. I. Bluestein, "A linear filtering approach to the computation of discrete Fourier transform," *IEEE Trans. on Audio and Electroacoustics*, vol. 18, no. 4, pp. 451–455, 1970.

[12] S. W. Smith, *The scientist and engineer's guide to digital signal processing*. California Technical Publishing, 1997.

[13] E. O. Brigham, *The fast Fourier transform and its applications*, 1st ed. Prentice-Hall, Inc., 1988.

[14] A. V. Oppenheim and R. Schafer, *Discrete-Time Signal Processing*, 3rd ed. Prentice-Hall, Inc., 2009.

[15] L. R. Rabiner and B. Gold, *Theory and Application of Digital Signal Processing*, 1st ed. Prentice-Hall, Inc., 1975.

[16] (2017) The WirelessHD Consortium. [Online]. Available: http://www.wirelesshd.org/

[17] (2017) IEEE standards association. [Online]. Available: http://standards.ieee.org/findstds/standard/802.15.3c-2009.html