An efficient parallel implementation of 3D-FFT on GPU

Selcuk Keskin[1], Ertunc Erdil[2] and Taskin Kocak[1]
[1]Department of Computer Engineering, Bahcesehir University, Istanbul 34353, Turkey
[2]Faculty of Engineering and Natural Sciences, Sabanci University, Istanbul 34956, Turkey
Email: {selcuk.keskin,taskin.kocak}@eng.bau.edu.tr, ertuncerdil@sabanciuniv.edu

***Abstract*—Three-dimensional fast Fourier transform (3D-FFT) is a very data and compute intensive kernel encountered in many applications. In order to meet the very high throughput requirements, dedicated application specific integrated circuit and field programmable gate array solutions for FFT computation are proposed in recent years. However, these solutions have a long development cycle, high design cost and fixed functionality. Conversely, software solutions are less expensive, scalable, and flexible and have shorter development cycle. In this paper, we propose an 3D-FFT algorithm on GPU. We demonstrated the performance of the proposed approach on artificially generated 3D images with various sizes. Experimental results show that the proposed GPU-based 3D-FFT implementation achieves up to 486 GFlops with memory and algorithmic optimizations.**

I. Introduction

Fast Fourier Transform (FFT) is a well-known algorithm that have made a huge impact in various fields of science and engineering such as digital and wireless communications, broadcast systems, image, audio, and video processing, digital signal processing. The algorithm provides a fast implementation for the Discrete Fourier Transform (DFT) with low complexity $O(nlogn)$ whereas that of the DFT is $O(n^2)$. Due to the time complexity of DFT, it becomes inefficient in the applications that need real time processing for high dimensional data sets.

Image processing and computer vision are two of the research areas where 3D-FTT has been most commonly used. Some applications of these fields include image denoising, texture segmentation and image reconstruction. Dabov et al. [1] propose a denoising approach that enhances each sparse representation of a 3D image using Fourier domain. Also, a non-exhaustive survey of image denoising literature includes various Fourier domain algorithms [2]. Matej et al. propose an FFT-based iterative approach for image reconstruction [3]. FFT have also been exploited for image segmentation since it has nice properties to extract textural information from images. One of the pioneering FFT-based texture segmentation method was proposed in [4].

Graphics Processing Units (GPUs) have become one of the most popular platform for high performance computing. GPUs provide parallel architecture, which combines raw computation power with programmability [5]. By executing thousands of threads simultaneously and much larger bandwidth, GPUs have become an important platform to implement high performance FFT. The only public FFT library on GPU is cuFFT provided by NVIDIA which supports multidimensional transforms of complex single precision data and 1D batched execution [6]. In this paper, we propose an 3D-FFT implementation that runs on GPU with support of high-throughput requirement. In order to test the performance of our 3D-FFT, we artificially generate 3D images with various sizes. Our experimental results demonstrate the efficiency of the proposed approach over the existing 3D-FFT implementations.

II. 3D-FFT Architecture

FFT is an efficient way of computing DFT for a set of signals. DFT converts a finite sequence of samples from time or space domain to frequency domain and vice versa. The forward transform is expressed as:

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

where N is the input length, n is spatial index and k is the frequency index.

The efficient algorithm proposed by Cooley and Tukey in 1965 for computing DFT was a major turning point in the development of digital signal processing [7]. The FFT butterfly operation which is the basic calculation element in the FFT process takes two complex points and 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) and gives two complex outputs (y_0, y_1) by using Eq. 2. The twiddle factors in FFT algorithms are trigonometric coefficients that multiplies the data [8].

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

where the twiddle factor, w^k, is given as:

$$w^k = e^{-\frac{2\pi i}{N}nk} = cos(\frac{2\pi}{N}nk) - isin(\frac{2\pi}{N}nk) \tag{3}$$

There are two different decomposition strategies for parallel 3D-FFT computation [9]. The 1D or slab decomposition is the easiest and the most common decomposition technique used in existing parallel FFT libraries [10]. The 3D input data are decomposed along one axis into slabs. 3D-FFT consists of 1D-FFTs for each XYZ dimensions.

Fig. 1 illustrates the steps of 3D-FFT computation strategy: (a) 1D-FFT along x dimension, (b) 1D-FFT along y dimension, and (c) 1D-FFT along z dimension.

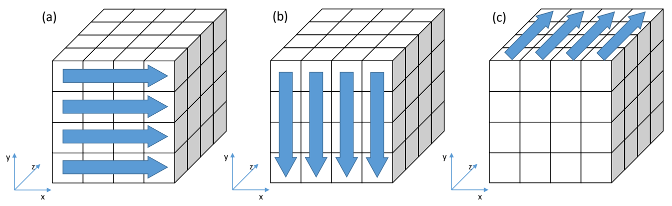

Fig. 1: Illustration of the steps involved in the 3D-FFT computation

III. GPGPU Based Parallel 3D-FFT Algorithm Design

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 [11]. CUDA C provides a simple path for users familiar with the C programming language to easily write programs to be executed by device. CUDA C extends C by allowing the programmer to define C functions, called kernels that are executed N times in parallel by N different CUDA threads, unlike the 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.

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.

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.

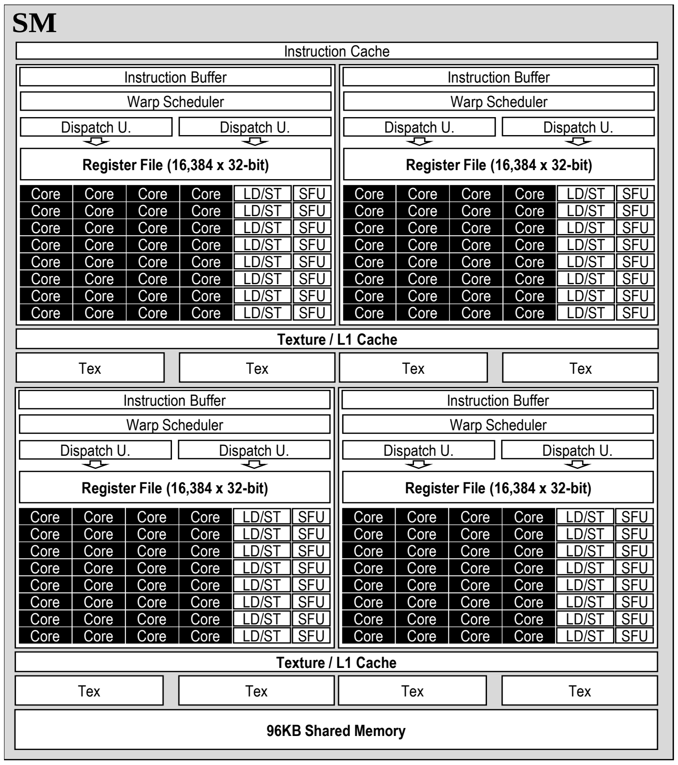

Fig. 2: Pascal GP102 Streaming Multiprocessor

B. Implementation of 3D-FFT computation on GPU

In the GPGPU based parallel computing, hardware architecture is very important while designing FFT computation algorithm to achieve the peak performance. Major key points in the algorithm design include calculation of twiddle factors, number of stages in FFT computation, batch size of the number of FFTs that will be computed in parallel, the memory architecture that keeps the intermediate values during the computation and the size of these values.

Coalesced global memory access has crucial importance for memory-bound applications on GPU. As stated in [12], a higher throughput can be achieved by ordering the data to guarantee coalesced 64bit or 128bit reads. To improve the performance of 3D-FFT, the 3D input array is represented as an 1D array of type *float2* that combines two floating point numbers for real and complex parts in one structure. When the inputs of 3D-FFT are real numbers, the complex part of input array is set as zero.

An N-point 1D-FFT process can be performed as a $logN$-stage computation module, where an 2-point 1D-FFT is calculated in each stage. To achieve the size of N, we use $N/2$ independent threads in GPU. In most cases, computing twiddle factors using *__sinf(x)* and *__cosf(x)* CUDA math library functions is faster than reading precalculated values from any memory location. So that, twiddle factors are computed on the fly in butterfly operation. The threads are communicated with each other over the shared memory. After the butterfly operation, each thread writes 2 complex values to the shared

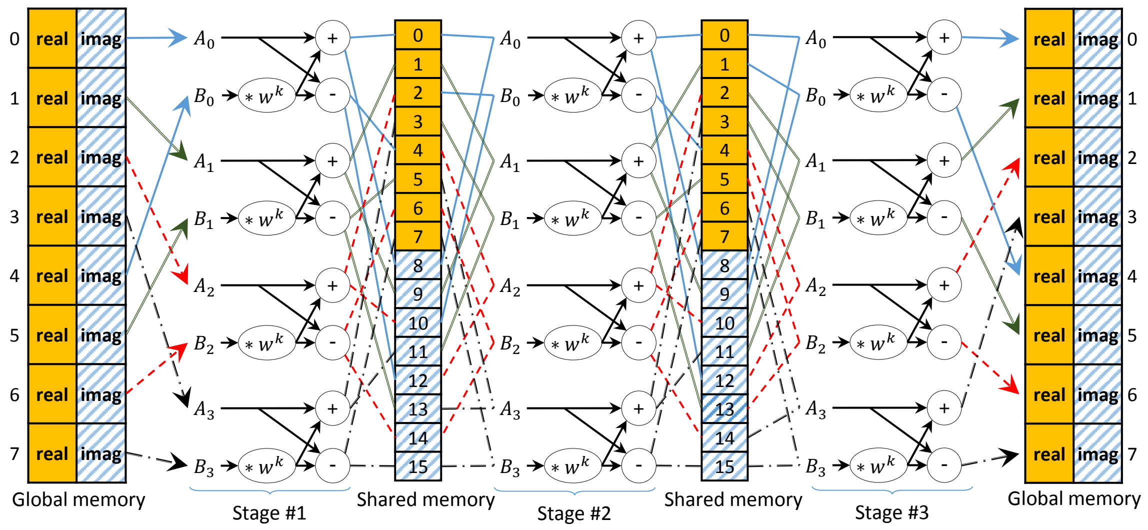

Fig. 3: 8-point FFT architecture with 3-stage (Note that; real: real part(solid), imag: imaginary part(stripe))

memory with non-contiguous locations which is calculated by thread number and current stage number. Then, for next stage, the threads read two complex values from the shared memory with contiguous locations to obtain coalesced access pattern. Unlike global memory representation, data arrays on shared memory are of type float so as to avoid bank conflict. The demonstration of 8-point 1D-FFT computation including shared memory usage with 4 independent threads is shown in Fig. 3.

During 1D-FFT computation along one axis, other lines in that axis are considered as independent. Threads can be identified using a two-dimensional thread index, forming two-dimensional block of threads, called a *thread block*. The second dimension of thread block, which defines the number of blocks, can be used to implement the other lines of one axis of 3D-FFT computation. There is a limit to the number of threads per block, since all threads of a block are expected to reside on the same processor core and must share the limited memory resources of that core. On current GPUs, a thread block may contain up to 1024 threads. However, a kernel can be executed by multiple equally-shaped thread blocks, so that the total number of threads is equal to the number of threads per block times the number of blocks. For an $N \times N \times N$ FFT computation, the number of threads per block is $N/2$ and the number of blocks is N. Therefore, the total number of threads is equal to $N \times N/2$.

N blocks are computed at the same time by the threads. Then, the threads repeat the same computation with new indexes N more times to finish all 1D-FFT computations along one axis. The threads write the results of each 1D-FFT computation to global memory so that new FFT computations can be started along next axis. For next axis, the start index is calculated by thread ids for each thread. To complete the 3D-FFT computation, all 1D-FFT computations along three axes are executed by $N \times N/2$ threads with transpose illusion which is obtained by calculating start point indexes and copying data from global memory to local memory. A pseudo-code for 3D-FFT computation described above is given in Algorithm 1.

Algorithm 1 3D-FFT computation with (N/2,N) thread block

```
1: $Index = tIdx.x + N \times tIdx.y$
2: //Along x axis
3: **for** $z = 0 \to N$ **do**
4:     $xIndex = Index + N \times N \times z$
5:     $A = input[xIndex]; B = input[xIndex + N/2]$
6:     Compute 1D FFT
7: **end for**
8: $Index = N \times tIdx.x + tIdx.y$
9: //Along y axis
10: **for** $z = 0 \to N$ **do**
11:     $xIndex = Index + N \times N \times z$
12:     $A = input[xIndex]; B = input[xIndex + N/2 \times N]$
13:     Compute 1D FFT
14: **end for**
15: $Index = N \times N \times tIdx.x + tIdx.y$
16: //Along z axis
17: **for** $z = 0 \to N$ **do**
18:     $xIndex = Index + N \times z$
19:     $A = input[xIndex]; B = input[xIndex + N/2 \times N \times N]$
20:     Compute 1D FFT
21: **end for**
```

IV. Results

The designed Cooley-Tukey based parallel 3D-FFT computation algorithm 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. We test the performance of our 3D-FFT on $N \times N \times N$ artificially generated images for each $N \in \{4, 8, 16, 32\}$.

The data transfer time between host and device memory is limited by PCI bus bandwidth. It is independent to the performance of code running time on GPU, so the transfer time is excluded from our experimental results. The correctness of our 3D-FFT outputs are verified by comparing with NVIDIA's cuFFT outputs. For a $N \times N \times N$ FFT with execution time of t seconds, its performance in GFlops is defined as

$$GFlops = \frac{5 \times N^3 \times log(N^3)}{t} \times 10^{-9} \quad (4)$$

The performance of GPU based 3D-FFT of size $N \times N \times N$ with different *batchsize* which is the number of concurrent 3D-FFT computations can be seen in Fig. 4. The performance of a batch FFT computation generally increases with the FFT size. Larger FFT problems have more parallelism, therefore improving the utilization of the computation power of GPU. The performance increase stops when more passes of global memory access is needed for some large FFT sizes.

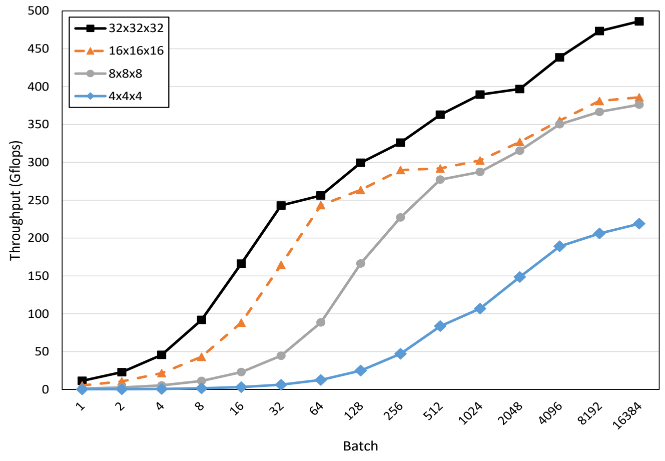

Fig. 4: Performance of 3D-FFT of size $N \times N \times N$

Our GPU based FFT algorithm can be compared with NVIDIA's cuFFT algorithm and Gu's work [8]. The cuFFT algorithm is implemented with CUDA 8.0 library and executed on NVIDIA TITAN X with same *batchsize* of our algorithm. Like our implementation, the transfer time between global and host memories for cuFFT algorithm is excluded from the measurements. As we can see from Fig. 5, we perform better than cuFFT and Gu's work. Moreover, cuFFT is a closed source algorithm. Before calling cuFFT algorithm, a plan function is executed by using the transform size, data type and batch size. The process must be waited until this plan function is finished.

V. Conclusions

This paper presents the design, methodology and implementation of GPU-based 3D-FFT algorithm for power-of-two sizes. Global memory accesses are reduced and data representation is improved to use memory coalescing. Our optimized 3D-FFT algorithm has achieved up to 486 GFlops computation throughput that is $1.14\times$ faster than the cuFFT's result. Our GPU-based 3D-FFT implementation can be used in many areas of scientic computing. As a future work, we are aiming to improve the efficiency of our GPU-based 3D-FFT algorithm and apply to a real image processing application.

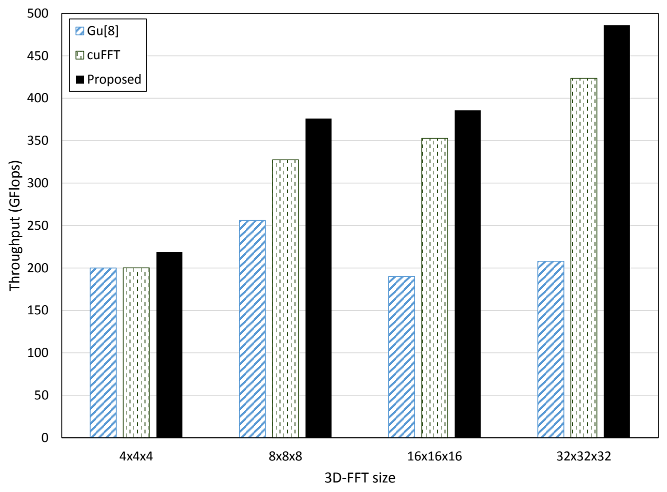

Fig. 5: Performance comparison of batched 3D-FFT implementations

Acknowledgment

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] K. Dabov, A. Foi, V. Katkovnik, and K. Egiazarian, "Image denoising by sparse 3-d transform-domain collaborative filtering," *IEEE Transactions on image processing*, vol. 16, no. 8, pp. 2080–2095, 2007.
[2] A. Buades, B. Coll, and J.-M. Morel, "A review of image denoising algorithms, with a new one," *Multiscale Modeling & Simulation*, vol. 4, no. 2, pp. 490–530, 2005.
[3] S. Matej, J. A. Fessler, and I. G. Kazantsev, "Iterative tomographic image reconstruction using fourier-based forward and back-projectors," *IEEE Transactions on medical imaging*, vol. 23, no. 4, pp. 401–412, 2004.
[4] T. R. Reed and H. Wechsler, "Segmentation of textured images and gestalt organization using spatial/spatial-frequency representations," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 12, no. 1, pp. 1–12, 1990.
[5] 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.
[6] (2017) The CUDA Programming Guide. [Online]. Available: https://developer.nvidia.com/category/zone/cuda-zone
[7] 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.
[8] L. Gu, X. Li, and J. Siegel, "An Empirically Tuned 2D and 3D FFT Library on CUDA GPU," in *Proceedings of the 24th ACM International Conference on Supercomputin*, June 2010, pp. 305–314.
[9] M. Eleftheriou, J. E. Moreira, B. G. Fitch, and R. S. Germain, "A Volumetric FFT for BlueGene/L," in *Pinkston T.M., Prasanna V.K. (eds) High Performance Computing - HiPC 2003. Lecture Notes in Computer Science*, vol. 2913, 2003, pp. 194–203.
[10] U. Sigrist, "Optimizing parallel 3D fast Fourier transformations for a cluster of IBM POWER5 SMP nodes," Ph.D. dissertation, The University of Edinburgh, Aug 2007.
[11] D. B. Kirk and W.-M. W.Hwu, *Programming Massively Parallel Processors*, 2nd ed. Morgan Kaufmann, 2012.
[12] J. Siegel, J. Ributzka, and X. Li, "CUDA Memory Optimizations for Large Data-Structures in the Gravit Simulator," in *2009 International Conference on Parallel Processing Workshops*, Sept. 2009, pp. 174–181.