An Empirically Tuned 2D and 3D FFT Library on CUDA GPU

Liang Gu
Department of ECE
University of Delaware
Newark, DE, USA
lianggu@udel.edu

Xiaoming Li
Department of ECE
University of Delaware
Newark, DE, USA
xli@ece.udel.edu

Jakob Siegel
Department of ECE
University of Delaware
Newark, DE, USA
jakob@udel.edu

ABSTRACT

In this paper, a Cooley-Tukey algorithm based multi-dimensional FFT computation framework on GPU is proposed. This framework generalizes the decomposition of multi-dimensional FFT on GPUs using an I/O tensor representation, and therefore provides a systematic description of possible FFT implementations on GPUs. The framework is geared to the efficiency of multi-dimensional FFT on GPU architectures. In particular, no global transposition among dimensions is performed and some previously unnoticed grouping and commutability of multiple dimensions are highlighted in order to reduce the number of computational kernels and minimize the number of global memory accesses. Important architectural factors and constraints of CUDA, such as coalesced access, bank conflicts and register pressure are also considered in this framework. Moreover, we adapt codelets, a straight-line style FFT implementation originally developed in FFTW, into our framework and prove that they are highly efficient on GPUs.

A 2D and 3D FFT library, currently supporting power-of-two sizes, is implemented on this framework and empirically-tuned results are compared with CUFFT and other recent publications on three NVIDIA GPUs. On a high-end NVIDIA GPU, GeForce GTX280, our 2D implementation is 2.8× faster than CUFFT and 1.6× faster than the best previously published results on average. Our 3D FFT implementation achieves 22.7× speed up over CUFFT on average. Furthermore both implementations show better precision than CUFFT. This library and its framework are potentially extensible to more general FFT problem sizes and other parallel architectures as well.

Categories and Subject Descriptors

G.4 [**Mathematical Software**]: Parallel and Vector Implementation

General Terms

Algorithms, Design, Performance

Keywords

2D FFT, 3D FFT, Library Generation, Empirical Tuning, GPU, CUDA

ICS'10, June 2–4, 2010, Tsukuba, Ibaraki, Japan.

1. INTRODUCTION

Graphics processing units(GPUs) have recently become one of the most popular platforms for high performance computing(HPC). With a large number of scalar processors and abundant bandwidth, GPUs have shown amazing computation power. NVIDIA's current generation of GT200 series GPUs can achieve a theoretical peak performance of $933GFlops$ and a theoretical memory bandwidth of 141.6Gbyte/s [1]. Intel's Larrabee recently demonstrated 1TFlops peak performance in SGEMM $4K \times 4K$ calculation [2]. The next generation NVIDIA GPU, Fermi, will have as many as 512 cores and greater potential in double precision computation. Compared with current multi-core CPUs, GPUs have many more parallel computing resources and can often achieve an order of magnitude performance improvement over CPUs on compute-intensive applications.

Discrete Fourier Transform(DFT) is one of the most important digital signal processing techniques. It is widely used in spectral analysis, image processing, data compression and many other fields. Its recursive algorithms, Fast Fourier Transforms(FFTs), reduce DFT's complexity from $O(N^2)$ to $O(Nlog(N))$. On traditional CPUs, a few FFT libraries such as FFTW [3] and SPIRAL [4] have been able to generate highly efficient implementations of DFT. Furthermore, recent work [5] shows that if enabled with single instruction multiple data(SIMD) extensions, these libraries can achieve more than 10GFlops on a multi-core CPU. With much more concurrent threads and much larger bandwidth, GPUs have become an important platform to implement high performance FFT. Unlike N-body simulations or matrix multiplications, FFT is largely a memory-bound problem due to frequent data exchanges. Therefore achieving peak performance on a GPU is a challenge. Before the advent of CUDA, there had been some FFT implementations using the OpenGL graphics API [6, 7]. However, the performance of these implementations is limited by lack of shared memory to assist fast data exchange.

The only publicly available FFT library on CUDA is CUFFT [8] provided by NVIDIA. It supports 1D, 2D and 3D transforms of complex and real-valued single precision data and 1D batched execution(multiple DFTs of the same sizes). Govindaraju etc. [9] and Nukada etc. [10, 11] have shown separately that CUFFT is far from optimized on high-end GPUs. These works have achieved performance improvements of $2\times - 4\times$ on 1D, 2D and $5\times - 8\times$ on 3D over CUFFT implementation. Both projects choose to implement Stockham radix-R FFT algorithm [12] to avoid bit-reverse in the original Cooley-Tukey algorithm. Govindaraju's work adopts three computation strategies. A shared memory algorithm is

used whenever the FFT problem can fit into shared memory. Otherwise a global memory FFT or a hierarchical FFT algorithm is used. Large prime size DFT is handled by Bluestein algorithm [13]. In order to achieve good performance, a couple of standard optimization techniques are applied in a hand-code level.

For multi-dimensional FFT in [9], FFTs along the a higher dimension is transposed to the lowest dimension before computation to increase data locality. To reduce the cost of a separate pass of transposition, it can be combined with one step of FFT computation. Nukada etc. have demonstrated in [10, 11] that computing FFT on a high dimension in-place without transposition can achieve better performance for 3D FFTs. However, this work has not presented a more general decomposition scheme of multi-dimensional FFT and their result is limited to two 3D FFT sizes.

In this paper, we propose a general computation framework of multi-dimensional FFTs based on Cooley-Tukey algorithm and a set of highly efficient straight-line code called codelet that is generated for small size DFTs. This framework covers a large number of possible DFT implementations, and it takes special performance characteristics of GPUs into consideration. More specifically, this framework spans a search space by decomposing FFT on each dimension, and grouping or exchanging FFT steps among computation kernels. Empirical search is then used to find a good implementation within the search space. Our implementation of 2D and 3D FFTs using this framework outperforms all currently released results on a high-end GPU, GTX280.

2. OVERVIEW OF CUDA AND MULTI-DIMENSIONAL FFT

2.1 Overview of NVIDIA CUDA

To make the computation power of modern NVIDIA GPUs available for general purpose applications, NVIDIA introduced compute unified device architecture(CUDA) [8] which consists of a special CUDA driver, an API in the form of a C language extension, the nvcc compiler-driver tool chain and the actual GPU hardware. The API allows programmers to write programs that utilize GPU's processing power without the need to use graphics APIs such as OpenGL or DirectX to access the device resources. A CUDA GPU is most easily described as a collection of Multiprocessors(MPs). The number of MPs varies from one device model to another. All the MPs have access to a global memory space which is the only resource on the device that can be read/written by all MPs. Each MP consists of 8 Streaming Processor Units (SPUs) that are driven by a single instruction unit. Furthermore all the threads that are running in parallel on a MP share computation resources on the MP. Those resources limit the number of threads that can be handled by at MP at the same time.

A kernel that is executed on the device will run in a Single Instruction Multiple Thread(SIMT) fashion. This is comparable to a SIMD execution, except that the threads are free to diverge. This diverging is handled by either serializing the different execution paths or by predication. The threads are organized in a grid of thread blocks, where all the threads of one block are always executed by the same MP. Communication and synchronization is limited to the threads inside the same block.

The hardware as well as the programming model put many constrains on what kind of application can be efficiently implemented on a GPU. The cost of communication and synchronization between threads, the constraint of shared hardware resources and the SIMT model further limit the possible complexity of a single kernel. Some candidates of CUDA implementation are obviously parallel algorithms(e.g. N-body simulation or ray tracing). However, many other problems still can greatly benefit from utilizing the GPUs processing power at least for their computational intense sections.

2.2 Representation of Multi-dimensional DFT

Multi-dimensional DFT can be represented in many different forms. In this work, we adopt an extended I/O tensor representation which is originated from FFTW [14]. This tensor representation can neatly expose many optimization opportunities and make the data movement in multi-dimensional DFT explicit, which is particularly important for the optimization on a GPU. This representation assumes constant-stride access of input and output data, which is true for both multi-dimensional DFT and the Cooley-Tukey algorithm. First, an I/O dimension d_1 is defined as $d_1 = d(n, i, o, I, O)$, where n is the DFT size, i and o are input and output strides, I and O are the pointers pointing to the starting addresses of input and output arrays. The d represents direct(normal) DFT while a t represents twiddle DFT as we will explain later. A couple of I/O dimensions compose an I/O tensor $t = \{d_1, d_2, ..., d_p\}$ which represents a multi-dimensional DFT. As in this paper, a tensor of nested loops, as is described in FFTW, is not used and a DFT problem is fully described in a single I/O tensor. A DFT on an input array of X points, $a_0...a_{X-1}$, is defined in equation (1).

$$A_k = \sum_{x=0}^{X-1} a_x e^{-\frac{2\pi i}{X}kx} \tag{1}$$

If this DFT is to be transformed from input address I to output address O and the data is be loaded and stored continuously, it can be compactly represented as $\{d(X, 1, 1, I, O)\}$ in I/O tensor form.

A multi-dimensional DFT is defined recursively following Equation (1) along each dimension of a multi-dimensional array. For example, a 3D DFT of transform length X, Y, Z is defined in equation (2).

$$\begin{aligned} A_{k_x,k_y,k_z} &= \sum_{x=0}^{X-1}\left(e^{-\frac{2\pi i}{X}k_x x}\sum_{y=0}^{Y-1}\left(e^{-\frac{2\pi i}{Y}k_y y}\right.\right. \\ &\left.\left.\sum_{z=0}^{Z-1}(e^{-\frac{2\pi i}{Z}k_z z}a_{x,y,z})\right)\right) \end{aligned} \tag{2}$$

If the 3D array is stored continuously along X dimension, this 3D transform can be compactly denoted as $\{d(Z, XY, XY, I, O), d(Y, X, X, O, I), d(X, 1, 1, I, O)\}$ in I/O tensor format. The tensor notation of multi-dimensional DFT not only specifies the sequence of 1D transforms to perform but also tells that these transforms are performed out-of-place each time, i.e. first I to O, then O to I, finally I to O. Following the multi-dimensional DFT definition or this tensor notation, a *row-column* algorithm computes XY 1D DFTs of size Z along dimension Z first, then similar batched 1D DFTs along Y and X are performed. Since the summations in (2) are interchangeable, the computation along different dimensions can be done in any order. Directly following the definition, a d dimensional DFT of total data size $N = N_1N_2N_3...N_d$ has a complexity of $O(N^{\frac{d+1}{d}})$. If FFT algorithms are used for each 1D transformation , the overall complexity decreases to $O(Nlog(N))$.

3. A MULTI-DIMENSIONAL FFT FRAMEWORK ON CUDA

3.1 Cooley-Tukey and Codelet in Tensor Form

We base our multi-dimensional FFT work on the Cooley-Tukey algorithm, which decomposes a large DFT problem to smaller ones recursively and solves sub-problems directly when they become small enough. The other key component of our framework are *codelets*, straight-line code that implements small size DFTs. Codelets are generated by a special purpose compiler *genfft* [15] and are originally used in FFTW. *Genfft* selects a FFT algorithm that is most likely to be efficient for a particular DFT size, creates a directed acyclic graph(DAG), simplifies the DAG, schedules instructions and un-parses to C. In the simplification step, several common compiler techniques and DFT specific transforms, such as making constants positive and DAG transposition, are applied. The instructions in a codelet is further scheduled with an algorithm to asymptotically minimized register spills. As a result, codelets have equal or fewer number of operations than the most sophisticated hand-written FFT implementation [15] and frequently have less register pressure. These straight-line style codelets have been proven to be highly-efficient on CPUs within the framework of FFTW [14]. However, no prior work on GPUs has implemented DFT using codelets as far as we know. In this work, we revised the codelets which are designed for the optimization on CPUs and successfully integrated them into our work on GPUs. Each codelet handles a small DFT and there is no data exchange between threads. After eliminating the loops in the codelets, they become completely branchless which is perfect for an execution on GPUs. Register pressure of codelets is typically much smaller than hand-written programs. For example, 51 or 52 registers are used for each thread in [10] for a 16-point FFT implementation on CUDA. A codelet computing the same FFT uses only 40 registers per thread. After carefully spilling 7 variables into the shared memory, the register usage drops to 32, which increases the occupancy from 25% to 50%.

There are primarily two kinds of codelets namely, direct and twiddle codelets. A Cooley-Tukey algorithm decomposes the DFT $d(rm, i, o, I, O)$ by computing r DFTs of size m, multiplying twiddle factors and computing m DFTs of size r. There are two implementation scenarios of Cooley-Tukey, decimation in time(DIT) and decimation in frequency (DIF). DIT recurses on the first batch of DFTs and DIF recurses on the second batch. Both cases are applicable in our multi-dimensional FFT framework. Take DIT as an example, a loop of DFTs $d(m, ri, o, I, O)$ are computed first, then twiddle factor multiplication and the second half loops of DFTs $d(r, mo, mo, O, O)$ are combined and solved using a twiddle codelet. The DIT algorithm recurses on the first part until DFT size is small enough to be solved by a direct codelet. Instead of treating the DFT loop as a second tensor as is in FFTW, we view it as an increase in the dimensions of the DFT problem. In our new notation, the above DIT algorithm is represented in equation (3),

$$\{d(rm, i, o, I, O)\} = \{d(m, ir, o, I, O), t^m_r(r, mo, mo, O, O)\} \quad (3)$$

where t^m_r is the twiddle codelet of size r including twiddle factors of m by r. With this notation, we overcome the inability to represent twiddle factor multiplication in FFTW and clearly specify the data movement in each step of the Cooley-Tukey algorithm. The capability to explicitly represent data movement in FFT is crucial to the success of our framework because the performance of FFT is to largely bounded by the memory accesses overhead on CUDA. As is shown in equation (3), r out-of-place DFTs of size m with transposition is performed first. Then m DFTs of size r with m by r twiddle factors multiplication is performed in place. In the case of DIF, the twiddle factors are multiplied to the output of the first batch of DFTs and these two steps are fused into one twiddle codelet. In our tensor form, DIF can be computed as is shown in equation (4).

$$\{d(rm, i, o, I, O)\} = \{t^r_m(r, im, im, I, I), d(m, i, ro, I, O)\} \quad (4)$$

In FFTW, twiddle codelets are in-place and transposition is always performed in direct DFTs. We have revised twiddle codelets so that they can perform transposition as well. Thus, the same problem can be solved with two more methods other than equation (3) and (4) where the transpose is performed out-of-place in the twiddle codelets.

3.2 Important Architecture Factors and Constraints of CUDA in the Framework

It is important that our multi-dimensional FFT framework adapts to CUDA architectural features: the global memory access, the shared memory usage, and the register pressure. This section briefly describes the characteristics of the three architectural factors and generally how they impact the performance of DFTs. In the next subsection, we detail the interaction of the factors with the transformation rules in our framework.

In CUDA, the latency of accessing global memory is much larger than that of shared memory. However large amount of input, output and intermediate results must reside on global memory because it is the only writable memory that is large enough to hold those data. If threads of a half-warp access continuous words of size $1, 2, 4$ or 8 Bytes, the individual reads are grouped together into a single coalesced access, which can be much more efficient than individual memory accesses. CUDA 1.2 or higher version uses minimal number of coalesced accesses to cover the region touched by the half-warp. In a bandwidth-bound application like FFT, coalesced access to global memory is among the top priorities in optimization. In our tensor representation, an I/O dimension $d(d, i, o, I, O)$ denotes one step of computation from I to O on global memory in a multi-dimensional FFT problem. If $i = 16 * 2^j$ where $j >= 0$, $o = 16 * 2^k$ where $k >= 0$ and I, O are aligned to 128Byte, this I/O dimension ensures 128Byte coalesced access.

When the above coalescing condition is not satisfied but the overall accessed region is a continuous block, shared memory can be used to assist coalesced access. For example, a 2D FFT is denoted as $\{d(y, x, x, I, O)d(x, 1, 1, O, O)\}$ and a direct computation of X dimension violates coalescing condition. In this case, we can copy the data to shared memory first, then perform the computation along X in shared memory before copying back. This process can be denoted as equation (5) if $x = 16 * 2^j, j >= 0$ and $xy < 2048$.

$$\{d(y, x, x, I, O)d(x, 1, 1, O, O)\} = \{d(y, x, x, I, S), \{d(x, 1, 1, S, S) + cp(y, x, x, S, O)\}\} \quad (5)$$

Here, S stands for an address on shared memory and cp stands for data copy. The size of shared memory on CUDA is currently 16KByte and that can hold fewer than 2048 complex numbers.

Another constraint of CUDA implementation is the bank conflict on shared memory. As described in the NVIDIA CUDA Programming Guide [16], bank conflict is eliminated

if strides between adjacent threads are odd and each thread accesses 32bit word at one time. Thus data type should be changed from *float2* in global memory to *float* in shared memory. For the above problem, one padding should be added at the end of X axis in shared memory if y is even. For our multi-dimensional framework, a more complicated higher dimensional padding scheme in shared memory is used and will be described in the next section.

In a multi-dimensional DFT of total size $N = d_1 d_2 ... d_n$, the total number of threads needed to compute one I/O dimension $d(d_i, i, o, I, O)$ is N/d_i. A CUDA block can hold at most 512 threads. Further limitations depends on the usage of register per thread and $16KByte$ shared memory per block. Table 1 shows the number of stack variables reported in codelets and register usage per thread of some codelets we use. The number of register usage is an average because

direct codelet	n1_2	n1_4	n1_8	n1_16	n1_32	n1_64
variables in code	5	13	36	82	136	202
register usage	6	12	18	40	60	40
twiddle codelet	t1_2	t1_4	t1_8	t1_16	t1_32	t1_64
variables in code	11	31	61	97	135	228
register usage	12	18	30	57	58	60

Table 1: Register usage per thread of some codelets

nvcc may use different numbers of registers between compilations. As we can see, the total number of variables grows rapidly with the increase of codelet size, but register usage grows much slower. As a result, all these codelets can run with at least 25% occupancy on GTX280 if shared memory usage is limited. Moreover, if the register usage is slightly more than 16 or 32 and free space on shared memory allows, we sometimes can double the occupancy by spilling a couple of carefully chosen variables in codelets to shared memory.

3.3 Transformations of Multi-dimensional DFT in Tensor Form

An implementation of a multi-dimensional DFT includes a series of transformation steps of the original problem. We solve a multi-dimensional DFT by a sequence of CUDA kernel functions. Each kernel implements one or several steps in the transform series and each kernel typically loads and stores the multi-dimensional array once. There are a large number of valid kernel sequences for a specific DFT. The problem is how to define a meaningful subset of the search space on CUDA and find the best performing sequence within it. As is shown in [9, 10], multi-dimensional DFT is largely a bandwidth-bound problem for CUDA. On one hand, the number of CUDA kernels(i.e. passes of global memory data accesses) should be minimized. On the other hand, each kernel should have abundant parallelism and the overall computation complexity should be kept low. We address the this problem and the optimization problem from a novel angle. The key point of our framework is a group of transforms of multi-dimensional DFTs in extended tensor form. These tensors have clear implication with regard to the interaction with the GPU architectural factors and provides guideline in performance tuning. Overall, these transforms span a huge enough search space of DFT implementations and the optimal solution within this search space can be found by empirical search.

A. Cooley-Tukey Decomposition of Multi-dimensional DFTs.

We will primarily use the DIT Cooley-Tukey algorithm on 2D DFT to describe our decomposition transforms. DIF and higher dimensional cases are similar. The row-column computation of a 2D DFT is represented as $\{d(y, x, x, I, O), d(x, 1, 1, O, O)\}$. As is stated earlier, Y dimension DFTs naturally have coalesced global memory access if $x = 16 * 2^j$ and $j >= 0$. X dimension DFT can be computed within shared memory and coalesced access of global memory is achieved. However, such a direct solution is inefficient or even impossible for large size DFTs. There are primarily three reasons to decompose a DFT into smaller ones. First, decomposition reduces codelet length and overall complexity. Second, it generates more parallelism so that more CUDA resources can be used. For instance, directly solving the 2D DFT of size 256×256 using codelet *n1_256* will allow only 256 threads and each thread needs to execute a codelet of about 7400 lines long. Finally, 1D DFT problems of size equal to or larger than 2048 will not fit into shared memory before decomposition.

If $y = y_1 y_2$ and $x = x_1 x_2$, we can apply a DIT Cooley-Tukey algorithm, as is represented in equation (3), to both dimensions of the 2D DFT. The resulted tensor expression is given in formula (6).

$$\{d(y_1, xy_2, x, I, O), t^{y_1}_{y_2}(y_2, xy_1, xy_1, O, O), \\ d(x_1, x_2, x_2, O, S), t^{x_1}_{x_2}(x_2, 1, x_1, S, O)\} \quad (6)$$

If the DFT size is large enough, we can recursively apply Cooley-Tukey algorithm to decompose the DFT. However, this process is not always beneficial when the codelet is too small due to the increase of global memory accesses and synchronization overheads. For instance, substituting the codelet *n1_16* with two codelets *n1_4* and *t1_4* using shared memory results in a performance decrease for 2D FFT of size 256×256. This decrease occurs probably because the latency of global memory load and store can not be hidden well with the presence of a synchronization between *n1_4* and *t1_4*. From our tuning experience, a codelet of size 8 to 32 is usually small enough to stop this decomposition process.

B. Grouping of Codelets.

In a sequence of codelets computing a multi-dimensional DFT, each codelet can form an independent kernel that loads from and stores to global memory. On the other hand, some codelets can be grouped into one kernel with intermediate results stored in shared memory. The benefit of grouping codelets into fewer kernels is obvious, i.e. fewer passes of global memory accesses. Nevertheless, certain limits exist to prevent grouping many codelets into a single kernel. Grouped kernels use shared memory to store intermediate results and there is only 16KByte of it on current NVIDIA GPU. Suppose the size of the last codelet along X dimension is x_i, where $x_i >= 16$, and the product of the sizes of the codelets need to be grouped is N. If the codelet sequence includes codelet x_i, the group naturally contains continuous data along X and N is limited by 2048 due to shared memory size. If the sequence does not include x_i, 16 continuous data along X axis need to be grouped so that coalescing access is achieved. In this case, $16N < 2048$, i.e. $N < 128$, is the constraint.

What many other multi-dimensional DFT implementations did not notice is that different dimensions or computation steps of different dimensions can be grouped into one computation kernel. For example, 2D DFT of size 128×128 can be expressed in formula (6), where $y_1 = x_1 = 16$ and $y_2 = x_2 = 8$. According to the grouping constraints mentioned above, the two X codelets can be easily grouped into

one kernel. But Y codelets has to be computed separately resulting in three passes of global memory access. However, codelet y_2 can be combined into the X kernel after the rewriting as is shown in formula (7). The last three codelets can be computed efficiently within one kernel.

$$\{d(y_1, xy_2, x, I, O), t^{y_1}_{y_2}(y_2, xy_1, xy_1, O, S), d(x_1, x_2, x_2, S, S), t^{x_1}_{x_2}(x_2, 1, x_1, S, O)\} \quad (7)$$

Exploiting this possibility of grouping kernels is of crucial importance to the performance of many multi-dimensional DFTs. In the case of 2D FFT of size 128×128, the total number of global memory accesses reduces by one-third and performance increases by more than 22% after eliminating one kernel.

For each computation step, a codelets is often computed in-place and the least number of data transposition is achieved in this framework. In the above example (7), two codelets along X and Y dimension are in-place and the other two are out-of-place. If $y = y_1y_2y_3$, we can apply equation (3) two times and the resulted codelet sequence along Y dimension is shown in formula (8).

$$\{d(y_1, xy_2y_3, x, I, O), t^{y_1}_{y_2}(y_2, xy_1, xy_1, O, O), t^{y_1y_2}_{y_3}(y_3, xy_1y_2, xy_1y_2, O, O)\} \quad (8)$$

Unlike the *Hierarchical FFT* described in [9] where each decomposition step associates with a data transposition, the above formula shows that our method needs only one transposition step for each dimension despite of how many codelets are used. Such a scheme applies to higher dimensions or X dimension, because (z, xy, xy, I, O) and (x_1, x_2, x_2, O, S) are essentially the same kind of problem as (y, x, x, I, O). If there are more than two transpositions in the I/O dimensions you want to group together, the computation on shared memory has to be out-of-place. An in-place computation kernel on shared memory handles twice the size of DFT compared with an out-of-place kernel.

C. Commutability and Partial Order.

An important property of multi-dimensional DFT is its commutability among dimensions. Because the summations in equation (2) is commutable, different orders of computing 1D DFT along all dimensions are mathematically equivalent. After applying Cooley-Tukey algorithm and the multi-dimensional DFT is expressed in tensor form, certain partial commutability exists. Part of the codelets sequence of one dimension can be moved into the codelets sequence of another dimension. However, the partial order among codelets of one dimension should be preserved. An example of such a partial order exchange of formula (7) is shown in formula (9).

$$\{d(y_1, xy_2, x, I, O), d(x_1, x_2, x_2, O, O), t^{y_1}_{y_2}(y_2, xy_1, xy_1, O, S), t^{x_1}_{x_2}(x_2, 1, x_1, S, O)\} \quad (9)$$

However formula (10), on the other hand, is an invalid exchange because the partial order of codelets x_1 and x_2 within X dimension is destroyed.

$$\{d(y_1, xy_2, x, I, O), t^{x_1}_{x_2}(x_2, 1, x_1, O, S), t^{y_1}_{y_2}(y_2, xy_1, xy_1, O, S), d(x_1, x_2, x_2, S, S)\} \quad (10)$$

This partial commutability is given without a strict mathematical proof but its validity is obvious and has been verified via our implementation.

Performing partial order exchange sometimes provides more choices in grouping codelets into kernels. Taking 2D DFT of size 16×16 as an example, a direct row-column algorithm $\{d(16, 16, 16, I, S), d(16, 1, 1, S, O)\}$ has only 16 threads for global memory access and non-coalesced writes to O. If we use DIT Cooley-Tukey algorithm once on Y dimension and exchange codelet order across dimensions, the following implementation (11) is available.

$$\{d(16, 16, 16, I, S), (16, 1, 1, S, O)\} = \{d(4, 64, 64, I, S), d(16, 1, 1, S, S), t^4_4(4, 16, 16, S, O)\} \quad (11)$$

Thus, only one kernel is used to solve this 2D FFT. It allows 64 threads to access global memory in 128Byte coalescing and the padding on shared memory is also easy as we will discuss later. Such a one kernel implementation enabled by partial commutability across dimensions increases the performance by about 100% and achieves 197GFlops on GTX280 for batched execution.

4. IMPLEMENTATION OF 2D AND 3D FFT LIBRARY ON CUDA

In this section, we describe the most important implementation and optimization issues that are considered when we implement and tune a 2D and 3D FFT library for CUDA based on the the multi-dimensional DFT framework introduced in the last section. Our library currently supports 2D and 3D DFTs of power-of-two sizes on NVIDIA GPU and provides batched execution.

4.1 Efficient Use of Global and Shared Memory on CUDA

Coalesced global memory access is of crucial importance for memory-bound applications on CUDA. Even for the latest compute models, where the memory access is organized in segments and strict coalescing is not needed, the grouping of the global memory accesses results in a better performance. As is shown in [17], by ordering the data to guarantee coalesced 64bit or 128bit reads, a higher throughput can be achieved for all used compute models. We arranged the data on global memory as arrays of type *float2*, combining two single precision floating point elements for real and complex parts in one structure. By using this layout all the needed operands for one warp can be read in a single 128Byte segment (compute model ≥ 1.3) or as 16 coalesced 64bit reads (compute model < 1.3). More transactions and memory latency for other warps can be hidden in the follow up computations without having to wait on additional memory operations. Our experiments show an implementation using *float2* on global memory is about 13% faster than using arrays of *float*.

Data arrays on shared memory, on the other hand, are of type *float* so as to avoid bank conflict. Codelets typically have several revised versions with different input and output data types so that it can be used on or between global and shared memory. Unlike 3D FFT library [10, 11], twiddle factors in our 2D and 3D FFT library are computed on the fly. Thus, codelets are revised so that they compute the twiddle factors instead of reading them from memory. In most cases, computing twiddle factors using __sin(x) and __cos(x) math library functions is faster than reading precalculated values from any memory location. The throughput of these math functions is 1 operation per clock and the accuracy of the overall implementation is better than CUFFT as we will show later.

Avoiding or minimizing bank conflicts on shared memory is also important to the performance of our FFT library. Our implementation typically has a sequence of thread access patterns(up to three) on the same data set and minimal bank conflicts in each access step is desired. For a 2D array of size $x \times y$ on shared memory, threads along the Y axis with data stride x are always bank-conflict-free. If threads along the X axis with stride 1 also need to access the 2D array

and x is even, one padding is added at then end of each X array. Otherwise, no padding is necessary if x is odd [16]. It is not always possible to achieve a bank-conflict-free layout for a 3D data array of size $x \times y \times z$ on shared memory that has threads accessing along all three dimensions. However, two special cases are frequently encountered, and we develop efficient padding schemes for both of them so that bank conflicts are eliminated. The first case is when $x = 16$, where only one padding is needed at the end of each X array similar to 2D even-x case is needed to eliminate bank conflicts for each dimension. If $y = 16$ and x is even, a 2D padding is illustrated in figure 1. Subfigure(a), (b) and (c) show thread blocks accessing along Z, Y and X axes. With this padding, at most 2-way bank conflict occurs when threads access elements along the Z axis. The second case is $y = 16$ and x is odd, no padding is needed at the end of each X array compared with the above case and and no bank conflict exists along any dimension.

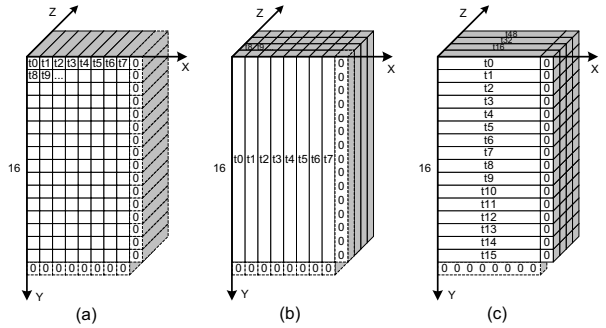

Figure 1: 3D array padding on shared memory when y==16. "0" denotes padding and t_i represents the elements accessed by thread i

4.2 Optimization Tradeoffs

Generally, fewer computation kernels or passes of global memory access are crucial for a better performance. However it is not always obvious whether splitting or grouping should be performed in kernels and it is important to find a good tradeoff. Table 2 shows the passes of global memory access and overall bandwidth achieved for 2D FFT on GTX280. On average, we achieve 74.8GB/s bandwidth on a GTX280 which has a theoretical bandwidth of 141.6GB/s.

N	16	32	64	128	256	512	1024	2048	4096
Passes	1	2	2	2	3	3	3	5	5
GBps	78	87	65	58	87	74	63	70	79

Table 2: Passes of global memory accesses and effective bandwidth(GBps) of 2D FFTs $N \times N$ on GTX280

We are not always using the least possible number of kernels and passes of global memory access because they will force us to use either large codelets or multiple passes of small codelets in shared memory. Large codelets reduce the total number of threads, i.e. parallelism, and waste the computation power within GPU. A codelet of size 256 has more than 7000 lines of code and executing it in a single thread takes much more time than distributing it to, for example, 16 threads. On the other hand, it is not always a good idea to use small codelets and the overhead of synchronization and transposition has to be justified. Frequently, a kernel computing one middle size codelet can perfectly hide the latency of global memory access without using shared memory. Decomposing it into two smaller codelets adds the overheads of a synchronization and shared memory accesses, which prevent a good hiding of latency and can result in much worse performance. Computing multiple codelets within shared memory is also limited by the shared memory size and bank conflict issues of multi-dimensional arrays. If there are more than two transpositions involved, the computation has to be out-of-place in shared memory, which further reduces the shared memory capacity. One last constraint comes from the static assignment of registers to thread blocks during compilation. Total register usage is estimated by multiplying the total number of threads and the largest number of registers used by individual threads. Such an pessimistic estimation of register pressure frequently decreases the occupancy of different codelets when they are grouped together into shared memory.

Our 2D and 3D FFT library supports batched FFT implementation, which has more parallelism than a single run and achieves better performance. The largest batch size is limited by 1 GByte of global memory. Furthermore, at least 32 threads are needed to efficiently utilize global memory bandwidth. For relatively large FFT sizes, there are enough threads within each block. However, 2D FFTs less than 16×16 and 3D FFTs less than 8×8 can only achieve 64Byte and 32Byte coalesced access respectively. Therefore, We batch 2 to 8 identical FFT problems into a single block to get more parallelism and each block has at least 32 threads. The assumption of a batch size being no less than 8 for small DFTs is reasonable. For batched 2D and 3D FFT of size 2, data is copied between global memory and shared memory in a separate step, where 128Byte coalescing is achieved.

5. EVALUATION

We evaluate the performance of our 2D and 3D FFT library for power-of-two sizes on three NVIDIA GPU cards, GeForce GTX280, Quadro FX5600 and GeForce 9500GT, which represent the three generations of NVIDIA's CUDA architecture that have been released so far. Their configurations are listed in table 3. The data copy time between host and device is limited by PCI bus bandwidth. It is independent to the performance of code running on GPU, and therefore is excluded from our measurements. The measured runtime of each FFT is normalized to be around 1 second by repeated execution and the minimum is reported out of four such measurements. The correctness of our FFT output is verified by comparing with FFTW double precision implementation. Our implementation supports both in-place and out-of-place FFT computation if the total data size is less than 2048. Otherwise, the computation has to be out-of-place. No planning or pre-calculation is needed for this FFT library and the twiddle factors are computed on the fly. We support batched execution for both 2D and 3D FFTs and maximum batch size is limited by 1GBytes of global memory. For a D dimensional FFT with a total data size of $M = N_1 N_2 ... N_D$ and with execution time of t seconds, its performance is reported in GFlops defined in equation (12).

$$GFlops = \frac{5M \sum_{d=1}^{D} log_2 N_d}{t} * 10^{-9} \quad (12)$$

5.1 Performance of 2D and 3D FFTs on GeForce GTX280

Figure 2 shows the performance of 2D single and batched FFT of size $N \times N$ on GeForce GTX280. Subfigure (a) compares the GFlops achieved by our implementation, CUFFT and Govindaraju's work [9]. Normalized runtime against our 2D FFT library is shown in subfigure (b) to highlight the runtime difference for small transform sizes. As we can see,

GPU	Compute capability	Multi-processors	Bandwidth (GBps)	Driver	Nvcc & Cufft	Gcc
GeForce GTX280	1.3	30	141.7	185.18.08	2.2	4.3.2
Quadro FX5600	1.0	16	76.8	190.18	2.3	4.1.2-46
GeForce 9500GT	1.1	4	25.6	190.18	2.3	4.1.2

Table 3: Configuration of tested GPUs

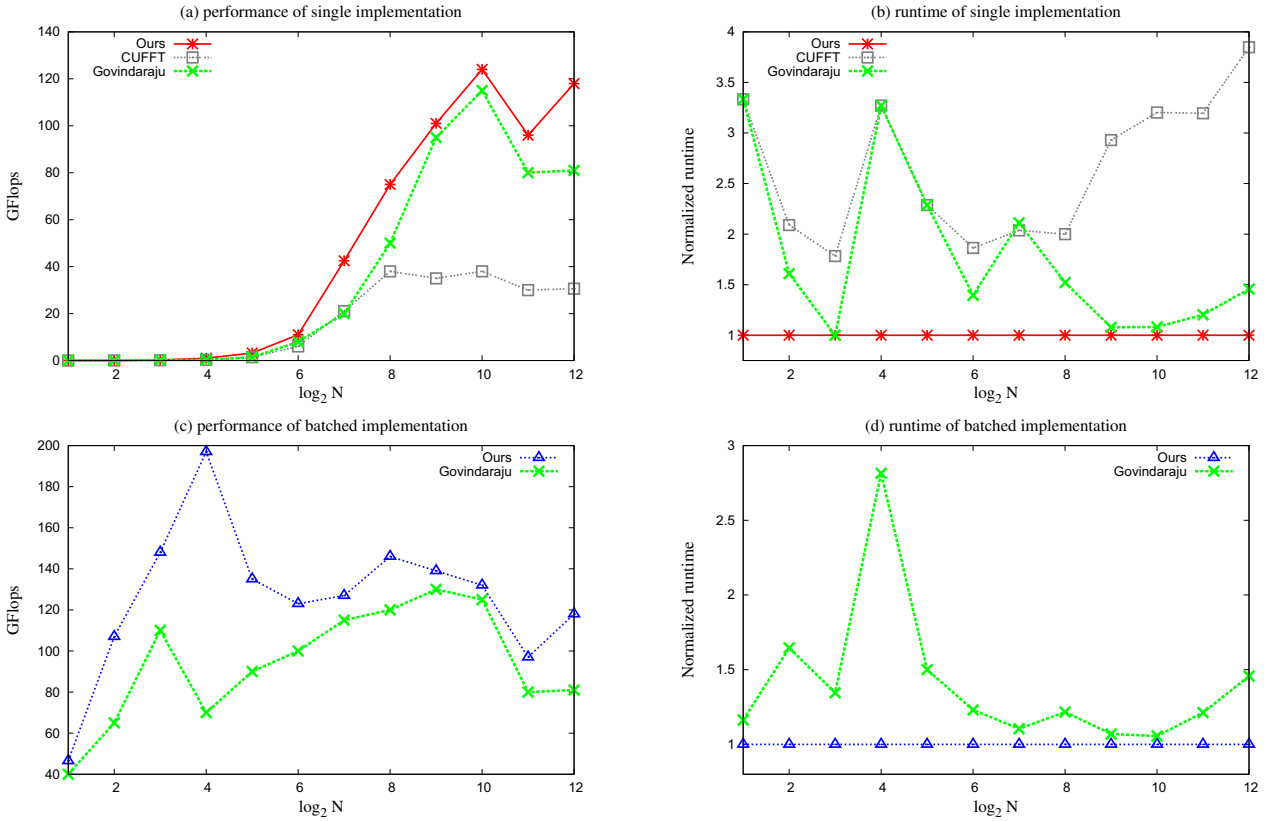

Figure 2: Performance of 2D FFT of size $N \times N$ on GTX280

we perform better than CUFFT and Govindaraju's work on all power-of-two $N \times N$ sizes that can fit into global memory. We achieve 2.65× speedup over CUFFT and 1.78× over Govindaraju's results on average. Performance of batched 2D FFTs of size $N \times N$ is shown in the two subfigures (c) and (d). Batch size is $2^{24}/N^2$ for a size $N \times N$ FFT, the maximum supported on CUDA. CUFFT does not provide batched implementation of 2D FFT, and therefore is not listed. Our batched implementation is faster than Govindaraju's results on all sizes. Particularly, batched 2D FFT of size of 16×16 achieves $197GFlops$, 2.8× faster than Govindaraju's result. On average, we have a speed up of 1.4 times.

The performance of a single FFT execution 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. The performance of both single and batched execution presents irregular shapes. The reason for the irregular shapes is that this library is hand-tuned and not all optimization techniques are applied to every FFT size. Some techniques such as the grouping and exchange of codelets are only beneficial for some particular DFT sizes, i.e., when they can indeed decrease the number of kernels or passes of global memory access. Therefore, the speedups for different DFT sizes are not even.

Figure 3 shows the performance comparison of 3D FFT among CUFFT, Nukada's work [11] and our implementation. Subfigure (a) compares the GFlops achieved and subfigures (b) shows the normalized execution time against our single execution. Similarly, batch size is $2^{24}/N^3$ for a size $N \times N \times N$ FFT. A single run of our 3D FFT implementation is up to 48.8 times and, on average, 17.4 times faster than CUFFT. Our batched 3D FFTs achieve $163.9GFlops$ on average and reach peak performance of $259GFlops$ on size 8×8×8. Nukada's work [11] reports only two $N \times N \times N$ 3D DFT results. For size 256×256×256, our implementation achieves the same performance, and we are about $20GFlops$ faster than their result on size 128×128×128. Our library demonstrates higher speedup over CUFFT on 3D than 2D FFT. This is because 3D FFT exposes more chances for codelet grouping or exchange techniques so as to reduce or assist memory access. CUFFT probably uses transposition three times for 3D FFT and two times for 2D FFT and our library uses none for either cases. The overhead of transposition can be a huge overhead for 3D FFT.

Figure 4 presents the performance of 2D FFTs of size $N_1 \times N_2$ and 3D FFTs of size $N_1 \times N_2 \times N_3$ on GTX280. We compare the GFlops achieved by CUFFT, our single and batched implementation for some randomly chosen sizes. Our single FFT implementation is faster than CUFFT by 3× for 2D and 28× for 3D FFTs for these test sizes. Our

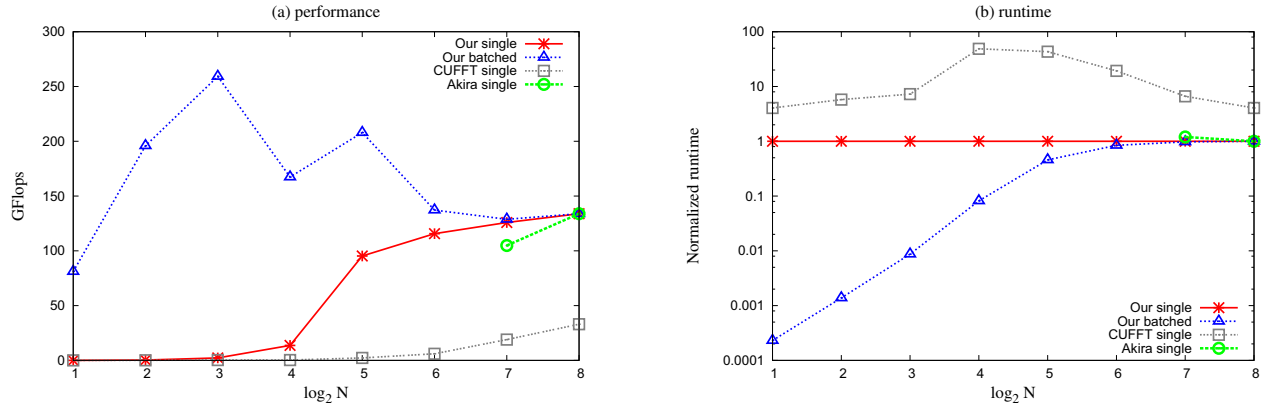

Figure 3: Performance of 3D FFT of size $N \times N \times N$ on GTX280

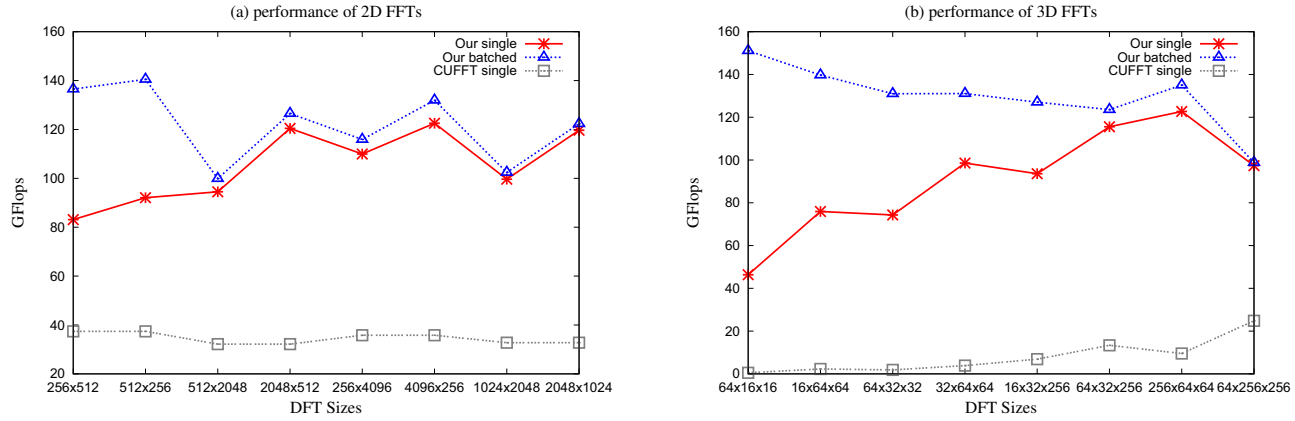

Figure 4: Performance of 2D FFTs of size $N_1 \times N_2$ and 3D FFTs of size $N_1 \times N_2 \times N_3$ on GTX280

batched implementation achieves $122GFlops$ for 2D and $130GFlops$ for 3D FFTs on average. For FFTs with the same total data size, CUFFT often has similar performance because it transposes each dimension to the lowest one and computes 1D FFT there. However, our library exploits the size difference in these FFTs and therefore performs different on different sizes. For example, we achieve better performance for 2D FFT of size 2048×512 than for 512×2048 because FFTs along X axis of the former can fit into shared memory but that of the other one can not.

5.2 Performance of 2D and 3D FFT on Quadro FX5600 and GeForce 9500GT

Figure 5 and figure 6 show the performance of 2D and 3D FFTs on Quadro FX5600 and GeForce 9500GT GPUs. Compared with a GTX280, these two graphic cards have fewer registers, active warps and smaller global memory bandwidth. The Quadro FX5600 has 16 multiprocessors and the GeForce 9500GT has only 4. CUFFT performs relatively better on these older generation GPUs, especially for large 2D and 3D DFT sizes. For 2D FFTs of sizes 1024×1024, 2048×2048 and 4096×4096, CUFFT is about 29% faster than our implementation on FX5600 and 19% faster on 9500GT. However, our implementation is faster than CUFFT on all other 2D and 3D sizes on both GPUs. More specifically for our single execution, we achieve an average speedup of $2.46\times$ for 2D FFT and $11.5\times$ for 3D FFT over CUFFT on a FX5600. Batched 2D and 3D FFTs have an average performance of $60GFlops$ and $71GFlops$. For 9500GT, our library is faster than CUFFT by $1.97\times$ for 2D and $7.56\times$ for 3D FFTs. Batched 2D and 3D FFTs achieve $16GFlops$ and $17GFlops$ on average.

The better performance of CUFFT on large 2D FFT sizes might result from data locality being of greater important on low-end GPUs than high-end ones. Therefore the cost of computing high dimensional FFTs in place increases as DFT sizes grows for our implementation. Though we do not have the source code of CUFFT but CUFFT probably uses transposition and only computes FFTs on the lowest dimension. And this is justified for large FFT sizes on low-end cards .

5.3 Precision evaluation

As a benefit of less computational complexity of the codelet, our FFT library has better precision than CUFFT. The root mean squared error(RMSE) of our FFT library and CUFFT is compared in figure 7. Input of these FFTs are unified pseudo-random numbers in the range of $[-0.5, 0.5)$. RMSE is computed by comparing FFT outputs on CUDA with FFTW double precision outputs. Absolute values of the RMSE are in the range of 10^{-8} to 10^{-3}. In figure 7, the normalized RMSE shows our precision is about $17 - 19\%$ better than CUFFT's on average. Besides the limitation of single precision numbers, another major source of error is the low-precision CUDA trigonometric functions that we use, i.e. *__sin* and *__cos*. Better precision is achieved in another version of our library where pre-calculated twiddle factors are loaded to the device. But this decreases the overall performance at times.

In summary, even though our current implementation is

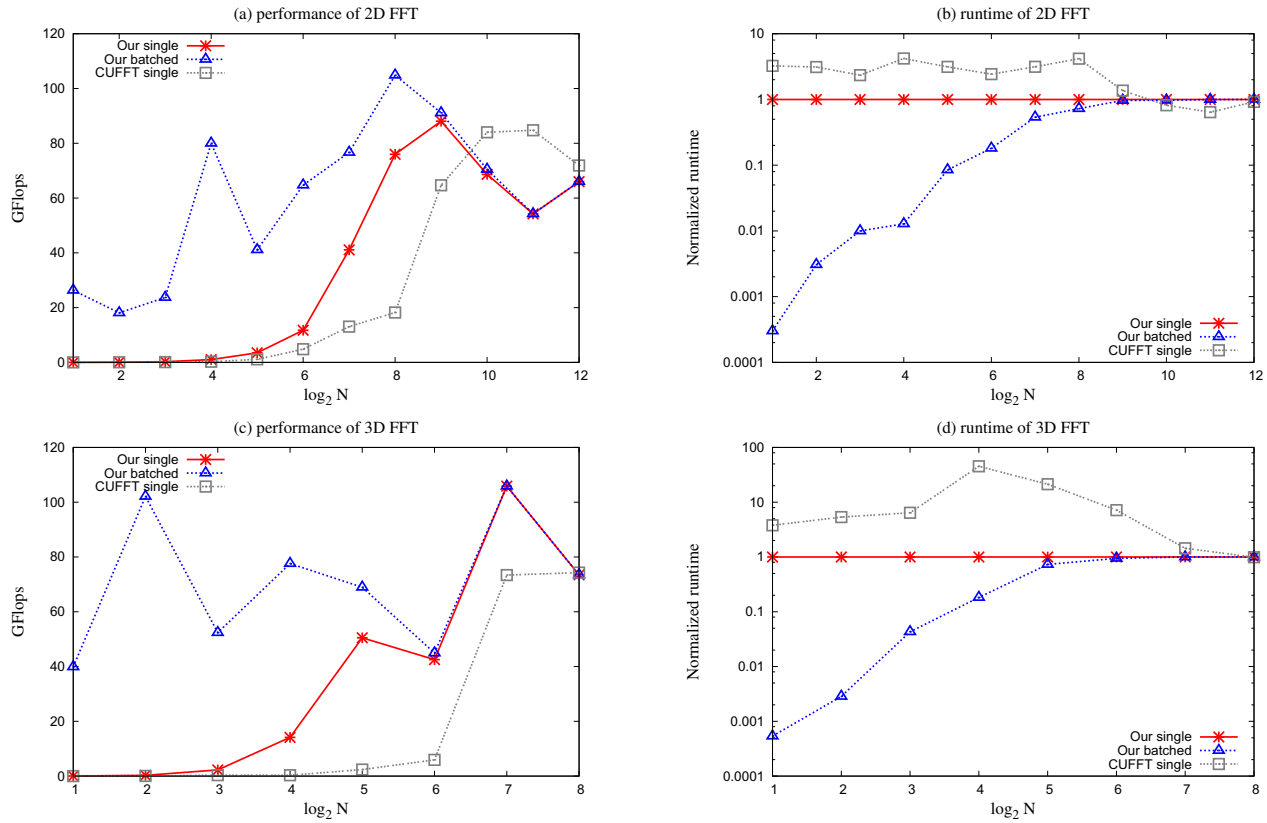

Figure 5: Performance of 2D and 3D FFT of size $N \times N$ and $N \times N \times N$ on FX5600

not tuned with fully automated empirical search, it already shows a good potential of our multi-dimensional FFT framework. Furthermore, better performance may be achievable by exhaustive search in the search space we described in this paper.

6. CONCLUSION

A Cooley-Tukey algorithm and codelet based multi-dimensional FFT computation framework for GPU is proposed in this paper. Some important architecture factors and constraints of the GPU are mapped to an extended I/O tensor format. Higher dimensional FFTs are computed in-place without being transposed to lower dimensions. Furthermore, grouping and commutability among dimensions are highlighted to reduce passes of global memory access and assist bank-conflict-free access to shared memory. A 2D and 3D FFT library using this framework is implemented on NVIDIA CUDA. On a currently high-end GPU, GTX280, our 2D library is 1.6× faster than the best previously published results and 2.8× faster than CUFFT on average. our 3D FFT library, on the other hand, is 22.7× faster than CUFFT. Experiments on older GPUs show our work achieves about $2\times - 12\times$ speedup over NVIDIA's CUFFT. Furthermore, our implementation shows better precision than CUFFT as well. The main contribution of this paper is the computation framework of multi-dimensional FFTs that is based on decomposition, grouping and commutability. Even though this search space is only partially explored in our 2D and 3D library on CUDA by hand-tuning, our test results have shown the success of this proposed multi-dimensional FFT framework.

Further performance improvements are expected if auto-tuning is used in this empirical search process. This work can be extended to non-power-of-two and higher multi-dimensional FFTs. Other parallel systems than GPUs, especially those that use shared memory, can also be the target platforms of this computation framework.

7. REFERENCES

[1] NVIDIA CUDA C programming best practices guide, July 2009.

[2] Justin Rattner. Supercomputing 2009 keynote, 2009.

[3] M. Frigo and SG Johnson. "the fftw web page." http://www.fftw.org, 2009.

[4] Markus Püschel and José M. F. Moura etc. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):232– 275, 2005.

[5] Franz Franchetti and Markus Püschel etc. Discrete Fourier transform on multicore. *IEEE Signal Processing Magazine, special issue on "Signal Processing on Platforms with Multiple Cores"*, 26(6):90–102, 2009.

[6] K. Moreland and E. Angel. The FFT on a GPU. In *Proceedings of the ACM SIGGRAPH EUROGRAPHICS conference on Graphics hardware*, pages 112–119. Eurographics Association Aire-la-Ville, Switzerland, Switzerland, 2003.

[7] Naga K. Govindaraju and Scott etc. Larsen. A memory model for scientific algorithms on graphics processors. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 89, New York, NY, USA, 2006. ACM.

[8] "Nvidia CUDA Zone" http://www.nvidia.com/object/cuda_home.

[9] N.K. Govindaraju, B. Lloyd, Y. Dotsenko, B. Smith,

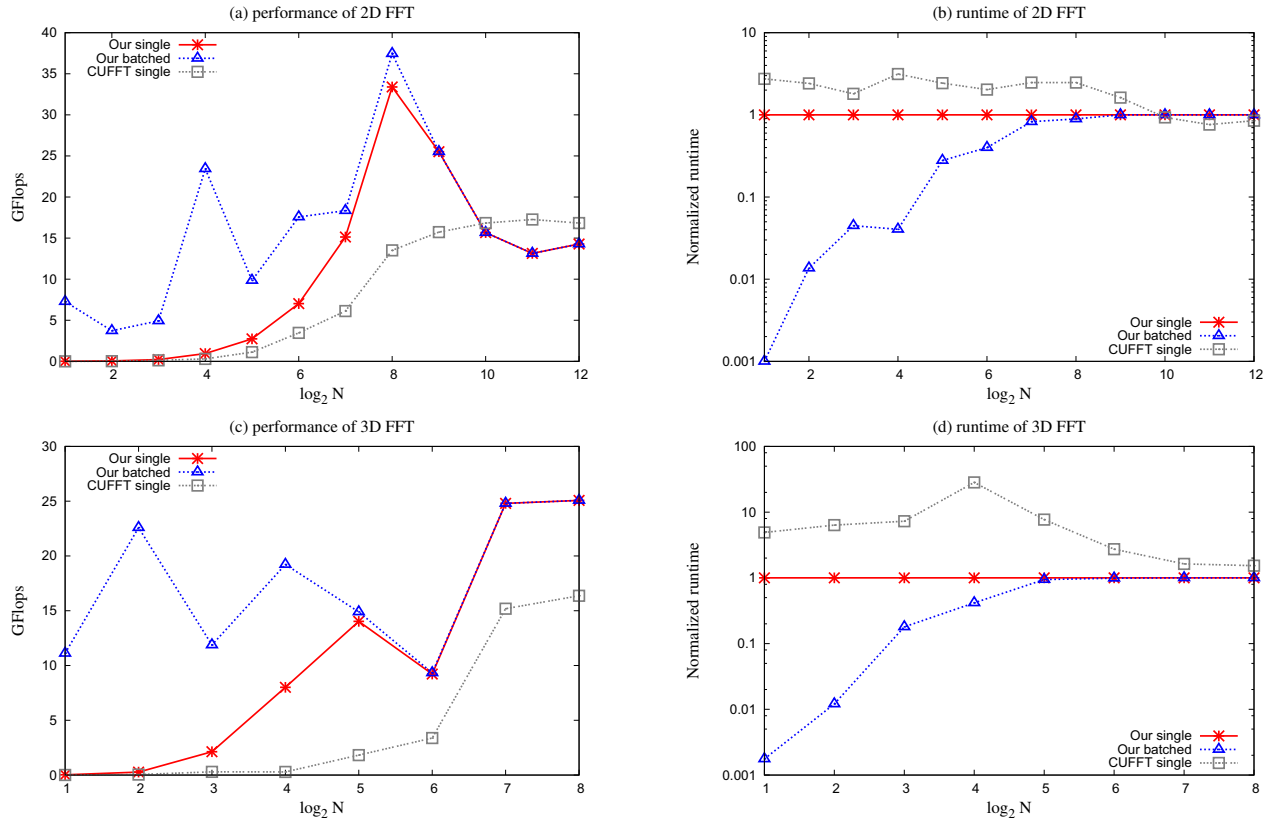

Figure 6: Performance of 2D and 3D FFT of size $N \times N$ and $N \times N \times N$ on 9500GT

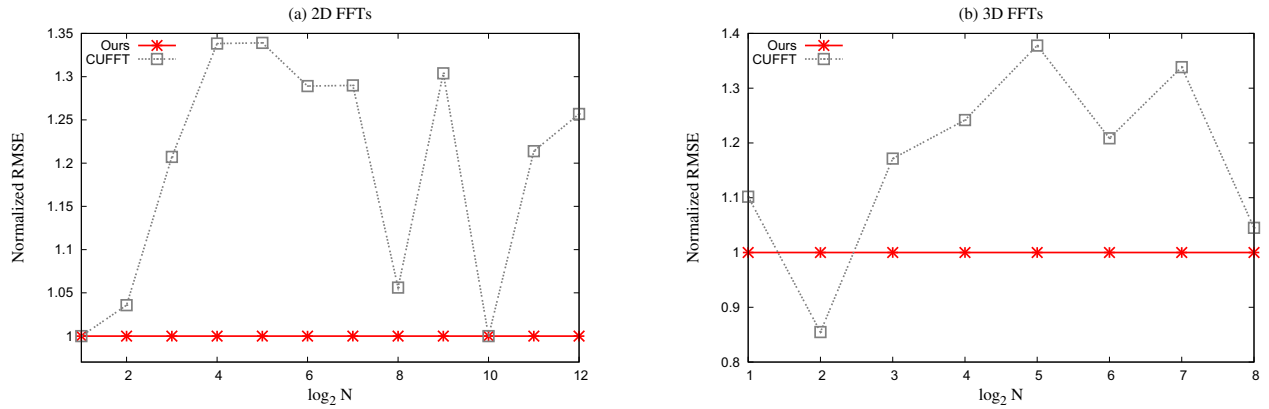

Figure 7: RMSE of 2D FFTs of size $N \times N$ and 3D FFTs of size $N \times N \times N$ on GTX280

and J. Manferdelli. High performance discrete Fourier transforms on graphics processors. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–12. IEEE Press, 2008.

[10] A. Nukada, Y. Ogata, T. Endo, and S. Matsuoka. Bandwidth intensive 3-D FFT kernel for GPUs using CUDA. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–11. IEEE Press, 2008.

[11] A. Nukada and S. Matsuoka. Auto-tuning 3-D FFT library for CUDA GPUs. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–10. ACM, 2009.

[12] C. Van Loan. *Computational frameworks for the fast Fourier transform*. Society for Industrial Mathematics, 1992.

[13] L. Bluestein. A linear filtering approach to the computation of discrete Fourier transform. *Audio and Electroacoustics, IEEE Transactions on*, 18(4):451–455, 1970.

[14] Matteo Frigo and Steven G. Johnson. The design and implementation of fftw3. *Proceeding of the IEEE*, 93(2):216–231, February 2005.

[15] M. Frigo. A fast Fourier transform compiler. *ACM SIGPLAN Notices*, 34(5):169–180, 1999.

[16] NVIDIA CUDA programming guide, Augst 2009.

[17] Jakob Siegel, Juergen Ributzka, and Xiaoming Li. CUDA Memory Optimizations for Large Data-Structures in the Gravit Simulator. In *Proceedings of The 38th International Conference on Parallel Processing*, 2009.