Optimized Strategies for Mapping Three-dimensional FFTs onto CUDA GPUs

[Extended Abstract]

Jing Wu
Dept. of Electrical and Computer Engineering
and Inst. for Advanced Computer Studies
University of Maryland, College Park, MD
jingwu@umiacs.umd.edu

Joseph JaJa
Dept. of Electrical and Computer Engineering
and Inst. for Advanced Computer Studies
University of Maryland, College Park, MD
joseph@umiacs.umd.edu

ABSTRACT

We address in this paper the problem of mapping three-dimensional Fast Fourier Transforms (FFTs) onto the recent, highly multithreaded CUDA Graphics Processing Units (GPUs) and present some of the fastest known algorithms for a wide range of 3-D FFTs on the NVIDIA Tesla and Fermi architectures. We exploit the high-degree of multithreading offered by the CUDA environment while carefully managing the multiple levels of the memory hierarchy in such a way that: (i) **all** global memory accesses are coalesced into 128-byte device memory transactions issued in such a way as to optimize effects related to partition camping [19], locality [22], and associativity. and (ii) all computations are carried out on the registers with effective data movement involved in shared memory transposition. In particular, the number of global memory accesses to the entire 3-D dataset is minimized and the FFT computations along the X dimension are almost completely overlapped with global memory data transfers needed to compute the FFTs along the Y or Z dimensions. We were able to achieve performance between 135 GFlops and 172 GFlops on the Tesla architecture (Tesla C1060 and GTX280) and between 192 GFlops and 290 GFlops on the Fermi architecture (Tesla C2050 and GTX480). The bandwidths achieved by our algorithms reach over 90 GB/s for the GTX280 and around 140 GB/s for the GTX480.

Categories and Subject Descriptors

D.1.3 [**Concurrent Programming**]: Parallel Programming

General Terms

Algorithm, Performance

Keywords

Fast Fourier Transform, GPU, Multi-threaded Algorithms, Scientific Computing

1. INTRODUCTION

The Fourier Transform and its discrete version, the Discrete Fourier Transform (DFT), constitute some of the most fundamental tools used throughout science and engineering. The introduction of the Cooley-Tukey [1] Fast Fourier Transform (FFT) algorithm is considered to be a breakthrough that has led to a number of very efficient methods for computing the DFT. These methods have enabled the widespread use of the FFT algorithm by both practitioners and researchers in a wide range of science and engineering applications such as computational fluid dynamics and digital signal processing. Since its introduction, considerable efforts have been devoted to map the FFT computation onto various specialized and general purpose parallel architectures, as they emerged over the years, so as to enable computational scientists to handle larger and larger scale applications. However, three-dimensional FFTs remain a significant computational challenge whenever the transform size is very large, which is often the case for large scale applications.

In this paper we address the problem of mapping power-of-two size three-dimensional FFTs onto the recent multithreaded GPUs and present some of the fastest known algorithms on four recent NVIDIA platforms. GPUs offer an extremely attractive choice because of their high performance and low cost, and are now widely available as co-processors on CPUs. For example, the Tesla C1060 contains 240 streaming processors (SPs), and can deliver in principle peak performance of up to 1 TFLOPS and peak bandwidth of up to 102GB/s to the 4GB device memory, at a cost of a few hundred dollars. Moreover, the CUDA parallel programming model [16][15] has played a critical role in ensuring the success of the recent GPUs due to the relative simplicity of the programming model, and its highly multithreaded and data parallel framework. Substantial performance gains have recently been reported on a wide range of scientific computations on GPUs using CUDA.

The main contributions of this paper are:

- A new approach for mapping power-of-two sizes 3D FFT computation onto CUDA GPUs which overlaps the FFT computation along the X dimension with data transfers required for the FFT computations along the Y and Z dimensions by adjusting the granularity of consecutive issue of device memory transactions and streaming multiprocessor (SM) executions of resident blocks.
- A minimal number of global data accesses such that all memory accesses are coalesced into 128-byte memory transactions, coupled with highly multi-threaded operations executed directly on the registers.
- The execution plan is tuned to optimize global memory accesses based on new strategies to deal with partition

camping, row access locality, and associativity.

- A general strategy to avoid shared memory bank conflicts in data transposition.
- Extensive tests that illustrate superior performance, both in terms of numbers of floating-point operations per second and global memory bandwidth achieved, on four different platforms.

The rest of the paper is organized as follow. Section 2 introduces recent GPU architectures and the CUDA programming model, with a particular emphasis on the memory model since our three-dimensional FFT computations are bandwidth-bound. This will be followed in Section 3 with an overview of recent related work using CUDA algorithms. In the following section, we will describe our overall strategy and the techniques used in implementing our algorithms. In Section 5, we will report a summary of the experimental results achieved by our algorithms on four recent NVIDIA platforms, and we conclude in Section 6.

2. CUDA OVERVIEW

Recent GPUs using the CUDA programming model have attracted considerable interest in the high-performance computing community due to their extremely high peak performance, low cost, and the relative simplicity of the programming model. Moreover these many-core processors tend to achieve much better performance to power ratios than the corresponding multicore CPUs while at the same time scaling to thousands of cores on a single card. The CUDA programming model uses multi-threading and data parallelism to exploit the many-core architectures of the recent NVIDIA GPUs, thereby achieving orders of magnitude better performance compared to multicore CPUs, especially on scientific applications. In this section, we start by giving an overview of such architectures, focusing on the four platforms used in our tests, followed by a summary of the main features of the CUDA programming model. We pay a particular attention to the memory model since this will play a central role in our algorithms.

The basic architecture of the recent NVIDIA GPUs consists of a set of Streaming Multiprocessors (SMs), each of which containing up to 32 Streaming Processors (SPs or cores) executing in a SIMD fashion; a large number of registers; and a small shared memory organized into multiple banks. Threads running on the same SM can share data and synchronize, limited by the available resources (number of registers and size of the shared memory) on the SM. Each GPU has small constant and texture caches (typically around 64KB). All the SMs have access to a very high bandwidth Global Memory; such a bandwidth is achieved only when simultaneous accesses are coalesced into contiguous 16-word lines. However the latency to access the global memory is around 400-800 cycles, which is quite high. A summary of the parameters of the four platforms we use in this paper is given in Table 1.

The CUDA programming model envisions phases of computations running on a host CPU and a massively data parallel GPU acting as a co-processor. The GPU executes data parallel functions called kernels using thousands of threads. Each GPU phase is defined by a grid consisting of all the threads that execute some kernel function. Each grid consists of a number of thread blocks such that all the threads in a thread block are assigned to the same SM. Several thread blocks can be executed on the same SM, but this will limit the number of threads per thread block since they all have to compete for the resources (registers and shared memory) available on the SM. Programmers need to optimize the use of shared memory and registers among the thread blocks executing on the same SM, if any.

Each SM schedules the execution of its threads into warps, each of which consists of 32 parallel threads. For the Tesla architecture (16 banks), a shared memory request for a warp is issued in two memory requests, one for each half-warp with a speed of two clock cycles. On the other hand, for the Fermi architecture (32 banks), a shared memory request for a warp is issued in one memory request with a speed of two clock cycles. When all the operands of the warps are available in the shared memory, the SM issues a single instruction for the 16 threads in a half-warp. The cores within an SM will be fully utilized as long as operands in the shared memory reside in different banks of the shared memory (or access the same location from a bank). If a warp stalls, the SM switches to another warp resident in the same SM.

Optimizing performance of multithreaded computations on CUDA requires careful consideration of global memory accesses (as few as possible and should be coalesced into multiple of contiguous 16-word lines); shared memory accesses (threads in a warp should access different banks); and partitioning of thread blocks among SMs; in addition to carefully designing highly data parallel implementations for all the kernels involved in the computation. In particular, threads in a half-warp which access contiguous words in the global memory are grouped together into a single coalesced global memory access thereby achieving the best possible throughput. Otherwise CUDA uses the minimum number of coalesced global memory accesses to cover the region touched by the half warp.

3. THREE-DIMENSIONAL FFTS AND RELATED GPU ALGORITHMS

We introduce in this section the basic FFT algorithms indicating those that will be used in the rest of the paper, and review the related GPU algorithms that have recently appeared in the literature.

3.1 FFT Algorithms

The one-dimensional discrete Fourier transform of n complex numbers represented by an array X is the complex vector represented by the array Y defined by:

$$Y[k] = \sum_{j=0}^{n-1} X[j]\omega_n^{jk} \quad (1)$$

where $0 \leq k < n$, and $\omega_n = e^{\frac{-2\pi\sqrt{-1}}{n}}$ the n^{th} root of unity. Various Fast Fourier Transform (FFT) algorithms have been proposed since the early 1960's, each of which has computational complexity of $O(n \log n)$. The most famous FFT algorithm is the Cooley-Tukey algorithm that uses a divide-and-conquer strategy to decompose a large size DFT into smaller size DFT's and compute these DFT's recursively. More specifically, let $n = n_1 n_2$ and let $j = j_1 n_2 + j_2$ and

[1]The shared memory size of the two Fermi devices is the default size

[2]Our TESLA C2050 card is not the typical card, and has around 2.62GB of device memory.

Table 1: Basic Parameters of our Four Platforms

	# of SMs	# of SPs per SM	# of Registers	Shared Mem.	Global Mem.	Peak Mem. BW	Clock Freq.
Tesla C1060	30	8	16K	16KB	4GB	102GB/s	1296MHz
GTX280	30	8	16K	16KB	1GB	141.7GB/s	1296MHz
Tesla C2050[1]	14	32	32K	48KB[2]	3GB	144GB/s	1147MHz
GTX480	15	32	32K	48KB[2]	1.5GB	177.4GB/s	1401MHz

$k = k_1 + k_2 n_1$ for $0 \leq j, k < n$ with $0 \leq j_1, k_1 < n_1$, and $0 \leq j_2, k_2 < n_2$. Then Eq (1) can be re-written as:

$$Y[k_1 + k_2 n_1] = \sum_{j_2=0}^{n_2-1} \left[\left(\sum_{j_1=0}^{n_1-1} X[j_1 n_2 + j_2] \omega_{n_1}^{j_1 k_1} \right) \omega_n^{j_2 k_1} \right] \omega_{n_2}^{j_2 k_2} \quad (2)$$

Eq (2) expresses the DFT computation as a sequence of three steps. The first step consists of n_2 DFT's each of size n_1, called radix-n_1 DFT, and the second step consists of a set of twiddle factor multiplications (multiplications by $\omega_n^{j_2 k_1}$). Finally, the third step consists of n_1 DFTs each of size n_2, called radix-n_2 DFT.

The Cooley-Tukey algorithm can be implemented in a number of ways depending on the recursive structure and the input/output order. Two important variations based on the recursive structure are the so-called the *decimation in time (DIT)* and the *decimation in frequency (DIF)* algorithms. The *DIT* algorithm uses n_2 as the initial radix, and recursively decomposes the DFTs of size n_1; while the *DIF* algorithm uses n_1 as the initial radix, and recursively decomposes the DFTs of size n_2. In this paper, we will focus on the *DIF* algorithm.

We note the two variations regarding the input and output orderings, namely in-order and bit-reversed order. Assuming that all the steps are carried out in-place, an examination of Eq(2) indicates that, after the first step, the output array becomes $XA[k_1 n_2 + j_2]$, and after the twiddle factor multiplication step, the output array is $XB[k_1 n_2 + j_2]$, while after the 3^{rd} step, the output array becomes of the form $XC[k_1 n_2 + k_2]$. A quick comparison against the DFT output array $Y[k_1 + k_2 n_1]$ implies that if both the radix-n_1 and radix-n_2 DFTs are in order, we would need a transposition of the intermediate output array after the 2^{nd} step so that the output is in order. However, if both the radix-n_1 and radix-n_2 DFTs are computed in bit-reversed order (namely, direct butterfly execution), and no transposition is done after the 2^{nd} step, we would generate a size n DFT with bit-reversed order output. In this paper, we will use the in-order input, bit-reversed order output since the corresponding in-place computation will allow us to better exploit the characteristics of the global memory. However, our algorithm can be converted to the in-order input, in-order output version accordingly.

A multi-dimensional DFT can be defined recursively as a set of DFTs along each of the dimensions of a multi-dimensional array. In particular, the 3D DFT of a 3D array of size $I \times J \times K$ is defined as follows:

$$Y[i, j, k] = \sum_{n=0}^{K-1} \sum_{m=0}^{J-1} \sum_{l=0}^{I-1} X[l, m, n] \omega_I^{il} \omega_J^{jm} \omega_K^{kn} \quad (3)$$

For each element in the DFT array, it is a summation of all the input elements multiplied by a specific coefficient determined by the input and output indices. Clearly, the order of the dimensions can be arbitrary, and the computational can be carried out in any order of the dimensions. Applying the Cooley-Tukey algorithm along each dimension, we can compute the 3D FFT on N elements in $O(N \log N)$ complexity.

3.2 GPU Related Algorithms

During the past few years, a number of efficient multi-dimensional FFT implementations on GPUs have been reported (for example, [9, 18, 6, 13, 12]). Here we summarize the results that are most relevant to our work.

Since 2006 NVIDIA has provided a CUFFT library[17], supporting 1D, 2D and 3D FFTs, and whose performance has shown continuous improvements over successive versions. In particular, a significant improvement appears in the later versions tailored for the Fermi architecture (devices with compute capability 2.x). However, the new library does not seem to achieve similar performance improvements for the Tesla architecture (devices with compute capability 1.x); moreover, the improvements on the Fermi architecture seem to depend significantly on the particular size of the transform.

The work of Govindaraju et al. [5] [6] has mainly targeted the one-dimensional case, showing a factor of 2-4 performance improvement relative to CUFFT version 1.1. Their library utilizes different algorithms for different transform sizes: 1) a global memory algorithm for larger FFTs with higher radixes; 2) a shared memory algorithm for smaller FFTs; 3) a hierarchical algorithm that makes use of both the shared memory and the global memory for larger FFTs with small radixes; and 4) mixed-radix FFTs for sizes with smaller prime factors and Bluestein's algorithm implementation for larger prime factors.

The work of Nukada et al. [13, 12, 11] has focused on the computation of the 3D FFT. Their algorithm distinguishes the X dimension transform from the Y and the Z dimension transform: the X dimension transform is performed using shared memory within a single kernel; the Y and the Z dimension transforms are computed using the multirow FFT algorithm [20], [10], [4], [8]. Their latest paper published in 2009 [12] introduces their auto-tuning library, showing performance that is 2.6-8.0 times higher than CUFFT library 2.1 for a number of transform sizes. Recently, they published a library called *Nukada FFT library* online [11] without explaining whether any new techniques beyond their earlier work were used. The performance of this library shows some improvements for the Tesla architecture but only modest performance is achieved on the Fermi architecture.

Finally, Gu et al. [7] proposed an empirically tuned 2D and 3D FFT library for power-of-two sizes using the in-order Cooley-Tukey FFT algorithm, achieving performance that is superior to the previously published results on a number of transform sizes. Their method is based on an extension of an IO tensor representation for multi-dimensional FFT and the use of codelets, both of which were originally developed for the well-known FFTW algorithm[3]. The authors point out possibilities for grouping or interleaving of different dimensions or computation steps to improve global memory per-

formance as well as overall performance. Their framework starts by generating a large number of possible FFT implementation strategies, followed by empirically going through these strategies until an optimized one is found. No details were given about the search strategy or the corresponding overhead.

4. OUR OVERALL STRATEGY AND CORE TECHNIQUES

Our work is based on the *DIF* version of the original Cooley-Tukey algorithm with in-order input and bit-reversed order output. A key feature of this algorithm is the "in-place" computation for all stages of the computation, which we will exploit to use memory access patterns that achieve good memory bandwidth. Our scheme targets large size 3D FFT such that no dimension is smaller than 128 as long as the input data can fit in the device memory. Every data element is assumed to be a complex number such that each of the real and imaginary parts is a single-precision floating point number, and hence each complex number is represented by 8 consecutive bytes. Our implementations are tuned to both the Tesla and Fermi architectures, which turn out to require slightly different implementations but with the same overall approach.

4.1 Representation of the 3D FFT Decomposition

As noted before, the Cooley-Tukey algorithm to compute the DFT of $n = n_1 \times n_2$ elements consists of three steps, the first of which involves n_2 radix-n_1 DFTs, followed by twiddle factor multiplications, and ending with n_1 radix-n_2 DFTs. Since we are dealing with 3D data, we need to specify the decomposition for computing the DFT along each dimension, as well as the data sets used for each radix computation. We will represent such a decomposition by making use of the tensor representation originally introduced in FFTW.

We first note that the data elements of a 3-D array will be stored in the device memory along the X dimension first, then the Y dimension followed by the Z dimension. Consider for example an array of size $256 \times 256 \times 256$. The entries of each vector along the X dimension will appear as a contiguous block of 256 complex numbers, while the entries of a vector along the Y dimension will have a stride of 256 between any consecutive entries of the vector. Along the Z dimension, consecutive entries will be 256×256 entries apart on the device memory. The FFT computation along each dimension will be specified by a number of FFTs each with a possibly different radix and each operating on the data along the dimension using a stride relative to *that dimension.* The actual global memory stride can easily be computed from such a specification. More specifically, a decomposition say $n = n_1 \times n_2$ (that is, radix-n_1 followed by radix-n_2) along the X dimension will be represented as follows:

- $X(n_2, n_1, n_2, n, tw)$
- $X(n_1, n_2, 1, n_2, no{-}tw)$

The above representation should be interpreted as follows. We start by performing n_2 FFTs each of radix n_1 on data along the X dimension with stride n_2, and hence these FFTs encompass n entries, followed by twiddle factor multiplications (which in our case are computed on the fly using fast intrinsic sine/cosine functions provided by CUDA). Then n_1 FFTs, each of radix n_2 is computed on the data along the X dimension with a stride of 1, and hence each FFT encompasses n_2 contiguous elements. We can extend the same representation to a decomposition with more factors such as $n = n_1 \times n_2 \times n_3$. Assuming that n is the size of the X dimension, this decomposition can be represented as:

- $X(n_2 n_3, n_1, n_2 n_3, n, tw)$
- $X(n_1 n_3, n_2, n_3, n_2 n_3, tw)$
- $X(n_1 n_2, n_3, 1, n_3, no{-}tw)$

The use of dimension name (X in the above equation) is necessary since we will be interleaving the radix computations between the different dimensions. Let's consider for a simple example the case of $256 \times 256 \times 256$ where the decomposition along the X dimension is given by $256 = 16 \times 4 \times 4$, while the decompositions along the Y and Z dimensions are identical $256 = 16 \times 16$. One (extremely inefficient) way to compute the corresponding 3D FFT can be represented as follows:

- $X(16, 16, 16, 256, tw) X(64, 4, 4, 16, tw)$ $X(64, 4, 1, 4, no{-}tw)$
- $Y(16, 16, 16, 256, tw) Y(16, 16, 1, 16, no{-}tw)$
- $Z(16, 16, 16, 256, tw) Y(16, 16, 1, 16, no{-}tw)$

4.2 CUDA Architecture Constraints

In this section we outline our main strategies to map the FFT computation on the Tesla and Fermi architectures so as to optimize the use of the available resources (both computation and memory resources) while managing the constraints imposed by these architectures.

4.2.1 Managing the CUDA Memory Hierarchy

The CUDA memory hierarchy consists of a global memory accessible by all the streaming processors, coupled with a shared memory and a set of registers on each of the SMs. Given that the FFT computation involves operations along each of the dimensions over a large 3D dataset stored in global memory, we have to pay a particular attention to the memory hierarchy while trying to execute a highly multi-threaded computation.

Given the typical size of our FFT computations, all the input, intermediate, and output data have to be held in the global memory, which has the largest access latency (400-800 cycles) in the memory hierarchy. Global memory accesses are carried out as 32-byte, 64-byte, or 128-byte device memory transactions. To achieve high bandwidth, global memory accesses must be coalesced - that is, global memory loads and stores by a half thread warp must be contiguous so as to result in a very few (one if possible) memory transactions. Since each complex number in our computation is represented by 8 bytes, aligned consecutive memory access of threads of a half-warp satisfies the largest 128-byte memory transaction size. In fact, global memory accesses issued by the threads in a warp will be executed as two 128-byte device memory transactions on either architecture thereby achieving a very good memory bandwidth. Unlike previously published GPU FFT algorithms, we always ensure coalesced 128-byte global/device memory transactions in addition to exploiting spatial and temporal locality to optimize effective device memory bandwidth. In particular, we exploit low-level device memory system hardware features to approach the theoretical device memory bandwidth. Device memory partition [19] and memory locality [22] are two important issues for a very good bandwidth. For example,

the device memory of GTX280 has 8 partitions and hence active warps should avoid issuing transactions that touch only a subset of them (so-called partition camping). Row access locality of device memory [22] is also preferred for high memory bandwidth, which can be interrupted by both algorithm restrictions and memory access streams issued by active warps. Note that the performance bottleneck of a relatively optimized radix FFT kernel is still the effective global/device memory throughput and hence we focus on memory optimization.

Compared to the global memory, the shared memory is much faster. The size of the shared memory per SM is 16KB for compute capability 1.3 (GTX280 and Tesla C1060) and 48KB (the default size) for compute capability 2.0 (GTX480 and Tesla C2050). Note that the shared memory size of the Fermi architecture can be configured between 16 KB and 48 KB. Each shared memory is divided into equal-sized memory modules (banks) so as to enable concurrent access. For the Tesla architecture, the bank count is 16 (half-warp) and for the Fermi architecture, the bank count is 32 (warp). The shared memory access is most efficient when bank conflicts are avoided, and hence we developed a general bank conflict free data transposition strategy. We observe that the L1 cache available on the Fermi architecture does not seem to significantly speed-up our FFT implementations while the L2 cache plays an important role.

Registers represent the fastest level of the memory hierarchy and are allocated to live threads; the peak arithmetic throughput can only be achieved by using registers rather than the shared memory[21]. The total number of 32-bit registers available is 16KB for compute capability 1.3 and 32 K for compute capability 2.0. We note that a thread is allocated at most 128 registers for compute capability 1.3 and 64 registers for compute capability 2.0 even though the compute capability 2.0 SM has more registers overall. The number of registers available and the maximum number of registers that can be allocated to a thread will have a direct impact on the radix decomposition adopted for each size. In particular, the maximum number of registers that can be allocated to a single thread on the Tesla architecture allows us to compute a radix-32 FFT using only the registers, which cannot be done on the Fermi architecture. For the latter architecture, we have to use more than a single thread to compute a radix-32 FFT. *In our implementation, an FFT of any radix along X, Y or Z dimension is computed directly on the registers, with the FFT computations along the X dimension almost completely overlapped with global memory data transfers needed to compute the FFTs along the Y or the Z dimension. This constitutes a major feature of our algorithms which distinguishes it from other published algorithms.*

4.2.2 Managing CUDA Threads

Note that CUDA programs rely on thread parallelism to hide memory and arithmetic latencies. However, relying only on increasing thread parallelism to optimize performance is not necessarily a good strategy because of the limits on several hardware resources such as number of registers and size of shared memory. Based on our experience, 64 threads per block on the Tesla architecture and 128 threads per block on the Fermi architecture seem to achieve the best balanced performance. In addition, we try to overlap global data movement and small radix computations along the X dimension to alleviate the latency dependency with the relatively small thread block parallelism. Our strategy is to make each thread compute a relatively small size FFT directly and use more threads to compute a single radix FFT if necessary. We will explain this process further later.

4.3 Overall Strategy

In our implementation, each kernel loads and stores the entire 3D data once from and into the global memory during which FFTs of certain radix sizes are carried out along possibly two dimensions concurrently. In general, we attempt to overlap a small radix FFT computation along the X dimension with data movement from the global memory needed for FFT computations along other dimensions. The mathematical properties of the Cooley-Tukey algorithm provide a rich set of possibilities for decomposing and re-ordering the overall computation so as to exploit the main characteristics of either the Tesla or Fermi architecture.

We start by stating an immediate implication of the mathematical formulation of the Cooley-Tukey FFT algorithm related to the ordering of the FFT subcomputations.

- Given a decomposition of the FFT along each dimension into a series of small-radix FFTs, each of which to be called an *FFT sub-computation*, we can arbitrarily inter-mix the FFT sub-computations of different dimensions as long as the relative ordering of the FFT sub-computations along each dimension is preserved.

This property was also observed by Gu et al. [7] .

We are now in a position to provide the main features of our strategy.

- The FFTs along the Y and the Z dimension are computed through separate kernels (typically two kernels for each dimension) while the FFT sub-computations along the X dimension are inserted into the kernels corresponding to the Y and Z dimensions. Occasionally, the FFT sub-computations along the Y and the Z dimension may be combined in the same kernel for improved performance on the Tesla architecture.
- The kernels to execute the FFT sub-computations along the Y and Z dimensions achieve high-bandwidth global memory accesses through the coalesced access of chunks of contiguous 128-bytes (16 elements) along the X dimension and through tuning the memory transactions issue sequence for device memory locality optimization. The corresponding radix FFTs are computed directly on registers.
- The FFT sub-computations along the X dimension are computed during the execution of the kernels for the Y and Z dimension FFT computations through the use of the shared memory to transpose data across the registers while avoiding bank conflicts.
- Within each kernel, the data loading (from global memory or shared memory rearrangement) and the FFT sub-computations are organized in such a way that the dependency between the data supply and the computations is optimized to match the execution pipeline.

The implementation of this strategy consists of three main steps. The first amounts to decomposing appropriately each of the Y and Z dimension size into a product of radixes (typically two) each of which is handled by a kernel. The second

step involves a decomposition of the X dimension, taking into consideration the decompositions along the Y and Z dimensions. At this step, we need to figure how to insert each of the corresponding FFT sub-computations along X into one of the Y or Z kernels so as to achieve high memory bandwidth and overlapped computation and data movement. Finally, we have to determine the workload of each thread and allocate the appropriate number of threads to each FFT radix computation. We will next describe the strategy to carry out each of these steps using the case of $256 \times 256 \times 256$ on the Tesla architecture.

4.3.1 Y and Z Dimension Decomposition

Two main factors seem to play a dominant role in determining the best decomposition for each of the Y and Z dimensions. Given that each Y or Z FFT sub-computation will access memory in a coalesced manner along the X dimension, the available resources have to be able to support a batch of 16×2^k Y and Z FFT sub-computations in the X dimension in parallel, for some non-negative integer k. The second factor is to try to achieve a load balance between different kernels while ensuring overall effective global memory access by the kernels.

The first factor puts an upper bound on the size of the radix that can be used on a given architecture, and the second implies almost balanced decomposition for each of the Y and Z dimensions whenever such a decomposition is needed.

Consider our running example of an input of size 256×256×256. Since we won't be able to accommodate 16 FFT(256) on a single SM of Tesla (which is usually the case for large size Y/Z dimension transform), each of the Y and Z dimensions has to be decomposed into a product of radixes. A balanced decomposition suggests that we use $256 = 16 \times 16$ for each of the Y and Z dimensions, implying the following four FFT sub-computations along the Y and Z dimensions:

- $\{Y(16, 16, 16, 256, tw)\}$
- $\{Y(16, 16, 1, 16, no-tw)\}$
- $\{Z(16, 16, 16, 256, tw)\}$
- $\{Z(16, 16, 1, 16, no-tw)\}$

Braces are used to indicate the boundaries of each kernel. We will next describe how to insert the FFT sub-computations along X into these kernels in such a way that their executions will be almost completely overlapped with the coalesced memory accesses for the above kernels.

4.3.2 X Dimension Decomposition

As we move data from the global memory in a coalesced fashion to carry out the FFT sub-computations along Y and Z, we organize each of the X dimension transforms into smaller-radix FFTs that can be incorporated into the kernels executing the Y and Z FFT sub-computations. Therefore the data movement should be organized so that each of the FFT sub-computations along X can be carried out by the same thread block executing the kernels. However, our Cooley-Tukey algorithm (*DIF* version) requires larger strides in early stages and smaller strides in later stages while the coalesced global memory access requires consecutive accesses to contiguous 128×2^k bytes of data. We resolve this tension between these requirements by using a number of small contiguous chunks with some stride in the X dimension for the earlier stages while using a large contiguous chunk for the later stages. Loading the data through the use

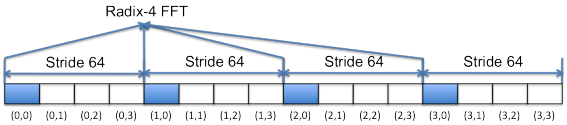


Figure 1: X Dimension Element Partition

of multiple small chunks (each chunk is of size 128 bytes) will incur a certain performance degradation, which depends on the size of the strides. In general, the FFT along X dimension is decomposed into three or four small-radix FFTs such as radix-2, radix-4, or radix-8 FFTs.

Consider again our running example of 256×256×256 data size whose FFT has to be computed on a Tesla GPU. We decompose the X dimension as $256 = 4 \times 8 \times 8$ and hence each such FFT can be computed as the sequence:

- $X(64, 4, 64, 256, tw)$
- $X(32, 8, 8, 64, tw)$
- $X(32, 8, 1, 8, no-tw)$

Suppose we want to insert the first FFT sub-computation into a Y kernel, which implies 64 sets of radix-4, stride 64 computation with associated twiddle factors for each row of 256 elements. For the Tesla architecture, we use a 64-thread block to load 64 elements for the X dimension in one row and exchange elements using block synchronization. To accommodate the computation and performance requirement, 256 elements in a row are partitioned into 4×4 sub-groups each of size 16 denoted from (0, 0), (0, 1) up to (3, 3) accordingly. This partition imposes a stride-64 (Figure 1) between elements of the same sub-group index from (0, x), (1, x), (2, x) and (3, x). Then 4 blocks of 64 threads consisting of 16 half-warps will be responsible for the 16 sub-groups and 4 half-warps from the same thread block will access the corresponding sub-group (0, x), (1, x), (2, x) and (3, x), (x can be 0, 1, 2, 3 for 4 blocks). Note sub-group data chunks are each of size 128-byte, namely the maximum coalesced device memory transaction size. Finally our overall algorithm for computing FFT($256 \times 256 \times 256$) can be summarized by the following representation in which each kernel is enclosed between braces.)

- $\{Y(16, 16, 16, 256, tw)\}$
- $\{X(64, 4, 64, 256, tw), Y(16, 16, 1, 16, no-tw)\}$
- $\{X(32, 8, 8, 64, tw), Z(16, 16, 16, 156, tw)\}$
- $\{X(32, 8, 1, 8, no-tw), Z(16, 16, 1, 16, no-tw)\}$

We will later provide the details about how the various small-radix FFTs are allocated to the thread blocks. Since each of the last three kernels contains FFT sub-computations along two distinct dimensions, the intermediate data needs to be appropriately transposed through the shared memory so that the corresponding FFT sub-computations can be carried out effectively. This is explained next.

4.3.3 Bank Conflict Free Shared Memory Transposition

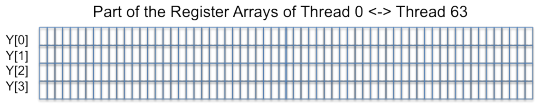
The FFT sub-computations along the Y and Z dimensions are always carried out directly on registers. To compute a small-radix FFT along the X dimension, we have to use the shared memory to transpose the data and move it back into registers before completing the sub-computations, after which we have to transpose back the elements into the registers as in the original layout for further processing.

To make efficient use of the shared memory, bank conflicts have to be avoided, although occasionally, trading bank conflicts for smaller shared memory usage can actually result in

better performance. This will occur in some kernels on the Fermi architecture.

Additional requirements on the shared memory transposition include balanced workload and avoiding warp divergence among the threads in a thread block.

The word size of each bank is 32-bit, the same size of a register and half the size of a complex number. To avoid bank conflicts, we separate the transposition of the real parts and the imaginary parts and add padding as necessary. We only consider the real parts for now; the imaginary parts are handled in a similar way. The transpose operation is carried out more or less the same way on both the Tesla and the Fermi architectures. At the beginning, the elements held in the registers are transferred into the shared memory and then loaded back in a transposed fashion into the registers. After the X dimension radix computation, a reverse transpose is conducted through the shared memory to restore the original layout of the data.



(a)

Shared Memory Array of Size 16x16

[0, 0~15]	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
[0, 16~31]	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
[0, 32~47]	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
[0, 48~63]	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
[1, 0~15]	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
[1, 16~31]	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
[1, 32~47]	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
[1, 48~63]	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
[2, 0~15]	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
[2, 16~31]	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
[2, 32~47]	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
[2, 48~63]	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
[3, 0~15]	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
[3, 16~31]	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
[3, 32~47]	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
[3, 48~63]	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63

(b)

Shared Memory Array of Size 16x16

[0, 0~15]	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
[0, 16~31]	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
[0, 32~47]	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
[0, 48~63]	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
[1, 0~15]	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
[1, 16~31]	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
[1, 32~47]	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
[1, 48~63]	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
[2, 0~15]	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
[2, 16~31]	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
[2, 32~47]	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
[2, 48~63]	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
[3, 0~15]	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
[3, 16~31]	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
[3, 32~47]	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
[3, 48~63]	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63

(c)

Figure 2: Shared Memory Transposition: (a) Register Arrays of 64 threads; (b) Store Elements from Registers to the Shared Memory Array; (c) Load Elements from the Shared Memory to Registers.

Continuing with our $256 \times 256 \times 256$ example and focusing on the second kernel above, we use 64 threads to load a 64×16 sub-array along the $X \times Y$ dimensions such that each half-warp loads four 128-byte chunks along the X dimension each time, for a total of 16 times load, ending up with each thread holds 16 Y dimensional elements with stride 16 in the end. Note that to ensure full utilization of the threads and maintain balanced workloads, each thread will have to compute 4 sets of radix-4 FFT along the X dimension and one set of radix-16 FFT along the Y dimension in four execution loops; namely, each time 4 rows of 64 elements are transposed. The data layout in the registers is illustrated in (Figure 2a) where the column corresponding to thread i represents the data held in the registers allocated to that thread. Our goal is to"transpose" this initial data layout so that it is stored into the shared memory as illustrated in (Figure 2b) and is loaded from the shared memory as illustrated in (Figure 2c).

Bank conflicts occur when multiple threads try to access different words from the same bank. The Tesla architecture has 16 banks and in this transposition scenario, bank conflicts do not occur. In other cases, we may have to use padding. Consider for example the case when we have to perform $X(8, 8, 8, 64, tw)$, namely, the workload of one block from the 4 blocks computing one row of $X(32, 8, 8, 64, tw)$. In this case, we need an 8×64 shared memory to transpose so that each thread will have its 8 elements required by the radix-8 FFT along the X dimension. This time, upon loading, every 8 consecutive threads will load 8 times of 8 consecutive elements from each 64-element row. Since 64 is a multiple of the number of banks (16), the number of banks used in the first row will need to be shifted in the second row to avoid threads in two consecutive rows trying to access the same bank. Namely, we need to pad 8 elements per 64-element row in this step and hence the resulting shared memory is of size $8 \times (64 + 8)$.

In general, the key idea is to stagger the banks from row to row so that bank conflicts are avoided. By using this strategy, it is clear that we will always be able to avoid bank conflicts.

4.4 Execution Plans

Once we have decided on the sequence of kernels to be executed, we have to allocate the operations to threads, which have to be organized into thread blocks and grid blocks.

It turns out that we use more or less fixed-size thread blocks for each of the Tesla and Fermi architectures. More specifically, we typically use 64 threads per block on the Tesla and 128 threads per block on the Fermi. We assign operations to thread blocks in such a way as to optimize the device memory throughput with respect to the partition camping problem and the row locality issue. We use a 2D representation $\{x_{size}, y_{size}\}$ for each block. The x_{size} is used to represent the number of threads along the X dimension, for each fixed value of X. The y_{size} is used to represent the number of threads used to compute the radix-FFT sub-computations along either the Y or the Z dimension. Therefore the total number of threads in a block is $x_{size} \times y_{size}$. Clearly the x_{size} threads are allocated to handle the X-dimension FFT sub-computations as well as transposition.

The organization of the grid of thread blocks is managed as a 2D array [x, y]. The x dimension of the array corresponds to the number of blocks used to cover the X dimension of the input data. For example, if the X dimension FFT size is 256 and the number of the threads in a block is 64, then we should have 4 blocks for the X dimension. The y dimension of the grid corresponds to the number of blocks

in Y and Z dimension. For our running example, the first kernel of the Y dimension needs 16 of 256/16 blocks to cover the data plane corresponding to a single Z coordinate value. To cover the entire data set, we need 16×256 blocks. We may change to a more balanced execution declaration (i.e. 4×16 as the x vector and 256 as the y vector) to avoid the CUDA grid size declaration limit. The thread blocks are executed in the order of their block IDs, so the block ID assignment should be tuned to optimize the device memory throughput, mainly for locality. For the Y dimension sub-steps, assigning block ID according to memory layout {x, y, z} is in general quite good.

We end this section by stating a couple of optimization techniques that may need to be applied to achieve optimized device memory throughput.

- *In-place or out-of-place execution.* Our algorithm is an in-place algorithm (reading and writing with the same stride), which helps to manipulate the memory access pattern. However occasionally, we may want to exploit out-of-place execution order (options) for global memory accesses locality possibility. Out-of-place execution for Y and Z dimension involves transposition in Y/Z dimension between sub-steps of the same dimension transform (which is merely a different stride access of device memory among rows (X dimension) of 256 elements). Such transposition can be done together during the storing into and the loading from the global memory step and results in global memory access with a balanced stride among kernels for the same dimension FFT computation. Whether it is actually adopted needs to be tuned with specific data sizes. Take size 256 FFT in the Y dimension for example. It is decomposed into 16x16. An in-order execution will consist of (i)16 sets of radix-16 with input and output stride 16 with twiddle, (ii) 16 sets of radix-16 with input and output stride 1. However, an out-of-order execution will consist of two 16 sets of radix-16 with input stride 16 and output stride 1, in addition to the twiddle multiplications. Note that we only tune this execution order for Y and Z dimension for the overall global memory latency while the properties of the memory access in the X dimension are all preserved.
- *Intermediate memory for smaller memory stride in Z dimension transform.* Based on the algorithm, the strides of the Z dimension are much larger than the Y dimension; if Z dimension transform is computed in more than one kernel, strictly implemented from the algorithm will yield relatively large global memory latency. This optimization attempts to make use of device memory transaction locality. We believe such an approach will provide more opportunities to achieve better device memory bandwidth throughput.
- *Ordering of Y and Z dimensions.* We always compute the Y dimension transform before the Z dimension. Inserting the X dimension sub-steps will involve stride-coalesced global memory access. Inserting such an X dimension access stride into the Y dimension kernels is much smaller than that the corresponding Z dimension kernels. Also, sub-steps of the same dimension matter when the sizes are not the same. For example, if we decompose Y dimension FFT size 128 into 16x8, which radix to compute first matters because this results in different memory strides. This probably arises from different pipeline granularities of continuous device memory transaction issues and computation workload of the same thread.

5. EXPERIMENTAL RESULTS

The performance of our 3D FFT scheme is evaluated on four NVIDIA GPU cards: two Tesla architecture cards with compute capability 1.3 (GTX280 and Tesla C1060), and two Fermi architecture cards with compute capability 2.0 (GTX480 and Tesla C2050). Hence for each architecture we have two cards with similar execution units but different memory bandwidths. Specifically, the GTX280 and the Tesla C1060 have the same number of identical streaming multiprocessors with respectively 141GB/s and 102GB/s peak device memory bandwidths. For the other two variations of the Fermi architecture, the peak device memory bandwidths are respectively 144GB/s (Tesla C2050) and 177GB/s (GTX480). Hence, for actual memory-bounded implementations of 3D FFT, we expect to see a performance increase when we move from Tesla C1060 to GTX280 or from Tesla C2050 to GTX480. Details about the various processors are shown in Table 1.

In our tests, the size of each dimension of the 3D FFT is a power of two and all of our implementations have been carefully compared to the output produced by CUFFT for correctness.

We capture two performance measures: the number of GFlops and the global memory bandwidth by our implementations. More precisely, if the execution time of our 3D FFT on data of size $NX \times NY \times NZ$ is t seconds, then its GFlops is measured using the standard formula:

$$GFlops = \frac{5 \cdot NX \cdot NY \cdot NZ \cdot [\log_2 (NX \cdot NY \cdot NZ)] \cdot 10^{-9}}{t}$$

Regarding the effective global memory bandwidth achieved, we use the formula:

$$BW = \frac{8 \cdot NX \cdot NY \cdot NZ \cdot \#\_of\_accesses \cdot 10^{-9}}{t}$$

where the #_of_accesses is the total number of global memory accesses (loading or storing). Each of our tests (our algorithm and other libraries as available) is run 5 times after which the arithmetic mean of the total runtime is used to compute the performance measures introduced above.

5.1 Performance Evaluation on the Tesla Architecture

Figure 3 illustrates the performance of our algorithm on the Tesla C1060 card compared to the best previous algorithms, and Figure 4 illustrates the corresponding performance on the GTX280. For each case, we try to increase the 3D data size up to the maximum possible that can fit into the global memory of the device. We run the tests using our algorithm, the CUFFT library, and the Nukada Library [11]. For Gu's performance on GTX280, we extracted the numbers from their paper [7]. The detailed decomposition, grouping and ordering schemes used for our implementations are given in the appendix. It is clear that our strategy achieves significantly better performance than the previous known schemes.

In our implementations, we used the same programs for the Tesla architecture, except for the data size 256×128×128.

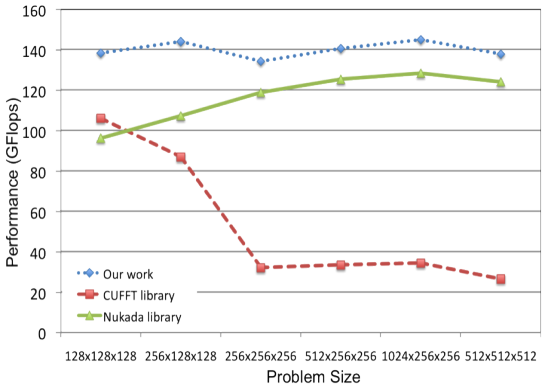


Figure 3: Performance on Tesla C1060

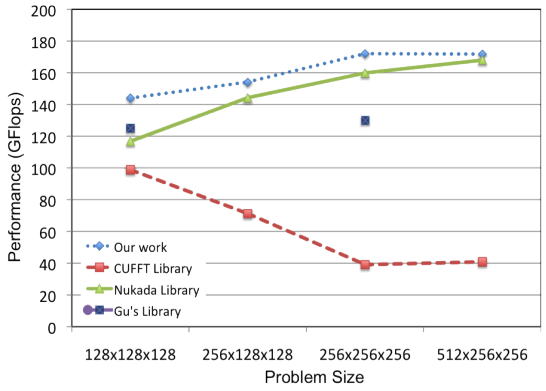


Figure 4: Performance on GTX280

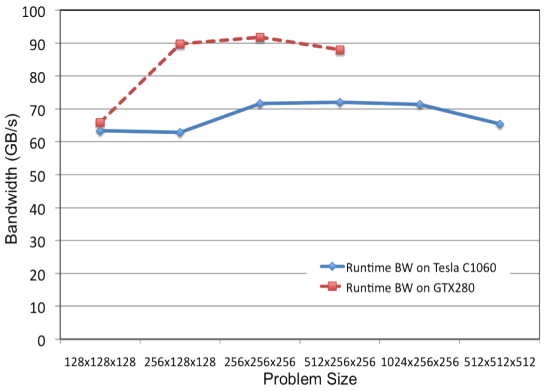


Figure 5: Actual Bandwidth on Tesla Devices

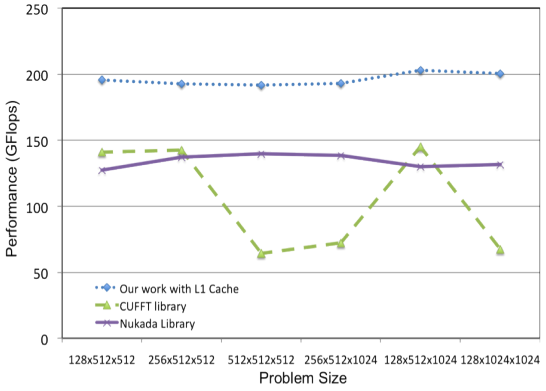


Figure 6: Performance on Tesla C2050

We slightly re-tuned the $256 \times 128 \times 128$ directly on the GTX280. As mentioned earlier, we expect better performance on the GTX280 since the theoretical bandwidth increases from 102GB/s to 141.7GB/s. The performance for the original code on data of size $256 \times 128 \times 128$ is respectively 144 GFlops and 140 GFlops on the Tesla C1060 and the GTX280. In the initial code, we decompose each of the Y and Z dimension transforms into 32×4 and 4×32 and combine the radix-4 sub-steps from the two dimensions into one kernel, inserting the X dimension transforms into kernels. This results into a relatively significant computation workload for each kernel, including large radix FFTs and transpositions. Such workload allocation is favored by the Tesla C1060 since the overhead of the device memory latency is much more significant (around 30%) than that of the GTX280. The code for 128×128×128 is the same because of its competitive performance on both cards; the computation overhead is not as significant as that of $256 \times 128 \times 128$ since the X dimension size is smaller.

Figure 5 and Table 2 show the actual bandwidth utilization of our implementations. As we can see from the figure, the actual device memory bandwidth of Tesla C1060 is usually lower than that of the GTX280 except for the computation-bound data size ($128 \times 128 \times 128$).

5.2 Performance Evaluation on the Fermi Architectures

[1]"NA" indicates cases of memory size usage larger than the global memory capacity.

Figures 6 and 7 illustrate the performance of our algorithms on the Tesla C2050 and the GTX480, compared to the best known 3D FFT algorithms on these platforms. Figure 8 illustrates the actual global memory bandwidth achieved on the two Fermi devices. The numbers reported were obtained by running our algorithms, the CUFFT library, and the Nukada library [11], on the same size 3D datasets.

Similarly, we use the same code, initially tuned on Tesla C2050, and evaluate the performance on both cards. Hence we are able to achieve around 200 GFlops on the C2050 and above 260 GFlops on the GTX480. We note the possibility of using caching on Fermi by setting the compilation flag on L1 and L2 cache. According to [14] all accesses to GPU DRAM go through L2, including CPU-GPU memory copies. For Fermi devices, global memory accesses are cached: the compilation flag -dlcm is used to determine if it can be cached in both L1 and L2 (the default setting) (dlcm=ca) or in L2 only (dlcm=cg). We evaluate the performance difference of caching effects on the two cards and is shown in Table 3 and Table 4. The evaluation indicates the L1 cache does not help much.

Table 5 shows the actual bandwidth utilization of our implementations with both L1 and L2 cache and just with L2 cache. As expected, the actual device memory bandwidth achieved on the Tesla C2050 is lower than that of the GTX480.

Table 2: Bandwidth achieved on the Tesla architecture cards

Data size	BW on Tesla C1060 (GB/s)	BW on GTX280 (GB/s)
128x128x128	63.27	65.85
256x128x128	62.86	89.70
256x256x256	71.64	91.76
512x256x256	72.02	87.92
1024x256x256	71.39	NA[1]
512x512x512	65.37	NA[1]

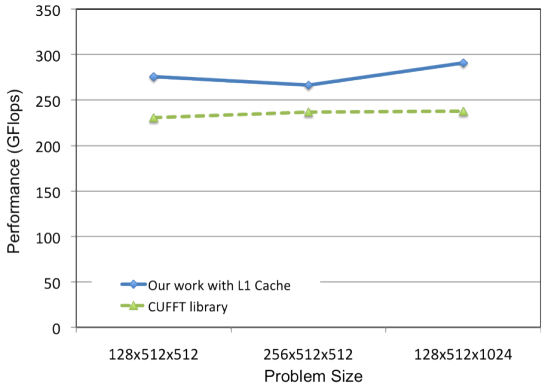


Figure 7: Performance on GTX480

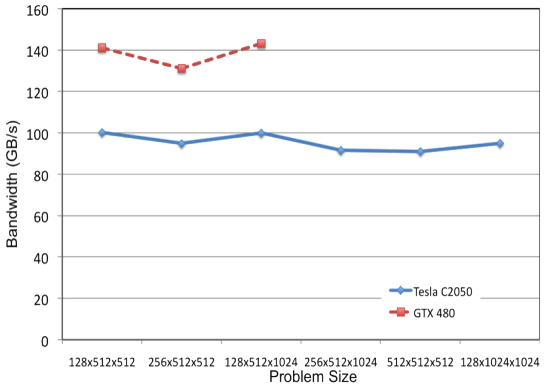


Figure 8: Actual Bandwidth on Fermi Devices

Table 3: Cache Effects of Perf. on Tesla C2050 (GFlops)

Data size	Tesla C2050 with L1+L2 Cache	Tesla C2050 with L2 Cache only
128x512x512	195.70	195.42
256x512x512	192.69	200.97
128x512x1024	202.97	203.04
512x512x512	191.70	195.04
256x512x1024	193.07	191.04
128x1024x1024	200.37	201.60

6. CONCLUSIONS

We presented in this paper a new approach to map multidimensional FFTs onto GPUs which seems to lead to extremely fast implementations for a wide number of data sizes across the Tesla and Fermi architectures. Our approach is carefully tailored to exploit the highly multithreaded environment in such a way as to almost completely overlap the FFT computations along the X dimension with the data transfers needed for the FFT computations along the other two dimensions. Moreover we minimize the number of global memory accesses while ensuring that each global memory access is a coalesced 128-byte memory transaction and optimizing the effects of related to partition camping, locality, and associativity. Our approach can easily be applied to 2D and 4D FFT computations to generate fast implementations on GPUs.

7. ACKNOWLEDGMENTS

This work was partially supported by an NSF PetaApps award, grant OCI0904920, the NVIDIA Research Excellence Center at the University of Maryland, and by an NSF Research Infrastructure Award, grant number CNS 0403313. We also thank the anonymous reviewers for their comments, noting that one of them pointed out reference [2] to us, which reports on mapping FFT algorithms on GPUs done concurrently with ours.

8. REFERENCES

[1] J. Cooley and J. Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, 19(90):297–301, 1965.

[2] Y. Dotsenko, S. Baghsorkhi, B. Lloyd, and N. Govindaraju. Auto-tuning of fast fourier transform on graphics processors. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pages 257–266, New York, NY, USA, 2011. ACM.

[3] M. Frigo, Steven, and G. Johnson. The design and implementation of fftw3. In *Proceedings of the IEEE*, pages 216–231, 2005.

[4] S. Goedecker. Rotating a three-dimensional array in an optimal position for vector processing: case study for a three-dimensional fast fourier transform. *Computer Physics Communications*, 76:294–300, Aug. 1993.

[5] N. K. Govindaraju, S. Larsen, J. Gray, and D. Manocha. A memory model for scientific algorithms on graphics processors. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.

[6] N. K. Govindaraju, B. Lloyd, Y. Dotsenko, B. Smith, and J. Manferdelli. High performance discrete fourier transforms on graphics processors. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 2:1–2:12, Piscataway, NJ, USA, 2008. IEEE Press.

[7] 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 Supercomputing*, ICS '10, pages 305–314, New York, NY, USA, 2010. ACM.

[8] D. G. Korn and J. J. Lambiotte. Computing the Fast Fourier Transform on a Vector Computer. *Mathematics of Computation*, 33:977–992, 1979.

[9] K. Moreland and E. Angel. The fft on a gpu. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on*

Table 4: Cache Effects of Perf. on GTX480 (GFlops)

Data size	GTX480 with L1+L2 Cache	GTX480 with L2 Cache only
128x512x512	275.58	284.88
256x512x512	266.42	280.05
128x512x1024	290.83	289.53

Table 5: Actual Bandwidth of Fermi Devices (GB/s)

Data size	C2050 L1+L2	C2050 L2	GTX480 L1+L2	GTX480 L2
128x512x512	100.20	100.06	141.10	145.86
256x512x512	94.86	98.94	131.14	137.87
128x512x1024	99.92	99.96	143.18	142.53
512x512x512	91.01	92.47	NA	NA
256x512x1024	91.53	90.57	NA	NA
128x1024x1024	94.99	95.58	NA	NA

Graphics hardware, HWWS '03, pages 112–119, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

[10] P. N. and Swarztrauber. Fft algorithms for vector computers. *Parallel Computing*, 1(1):45 – 63, 1984.

[11] Nukada. Nukada FFT Library website. http://matsu-www.is.titech.ac.jp/ nukada/nufft/, 2011.

[12] 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*, SC '09, pages 30:1–30:10, New York, NY, USA, 2009. ACM.

[13] 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*, SC '08, pages 5:1–5:11, Piscataway, NJ, USA, 2008. IEEE Press.

[14] NVIDIA Corporation. CUDA and Fermi Update, 2010.

[15] NVIDIA Corporation. NVIDIA CUDA C programming best practices guide, 2011.

[16] NVIDIA Corporation. NVIDIA CUDA C programming guide, 2011.

[17] NVIDIA Corporation. NVIDIA CUDA cufft library, 2011.

[18] Y. Ogata, T. Endo, N. Maruyama, and S. Matsuoka. An efficient, model-based cpu-gpu heterogeneous fft library. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1 –10, april 2008.

[19] Ruetsh, Greg and Micikevicius, Paulius. Optimizing Matrix Transpose in CUDA, 2011.

[20] C. Van Loan. *Computational frameworks for the fast Fourier transform*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1992.

[21] V. Volkov. Better Performance at Lower Occupancy, 2010.

[22] G. L. Yuan, A. Bakhoda, and T. M. Aamodt. Complexity effective memory access scheduling for many-core accelerator architectures. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 34–44, New York, NY, USA, 2009. ACM.

APPENDIX

A. EXECUTION PLANS

A.1 Tesla Architecture Plans

A.1.1 Tesla 256×256×256

- $\{Y(16, 16, 16, 256, tw)\}$
- $\{X(64, 4, 64, 256, tw), Y(16, 16, 1, 16, no{-}tw)\}$
- $\{X(32, 8, 8, 64, tw), Z(16, 16, 16, 256, tw)\}$
- $\{X(32, 8, 1, 8, no{-}tw), Z(16, 16, 1, 16, no{-}tw)\}$

A.1.2 Tesla 512×256×256

- $\{X(128, 4, 128, 512, tw), Y(16, 16, 16, 256, tw)\}$
- $\{X(256, 2, 64, 128, tw), Y(16, 16, 1, 16, no{-}tw)\}$
- $\{X(64, 8, 8, 64, tw), Z(16, 16, 16, 256, tw)\}$
- $\{X(64, 8, 1, 8, no{-}tw), Z(16, 16, 1, 16, no{-}tw)\}$

A.1.3 Tesla 1024×256×256

- $\{X(256, 4, 256, 1024, tw), Y(16, 16, 16, 256, tw)\}$
- $\{X(256, 4, 64, 256, tw), Y(16, 16, 1, 16, no{-}tw)\}$
- $\{X(128, 8, 8, 64, tw), Z(16, 16, 16, 256, tw)\}$
- $\{X(128, 8, 1, 8, no{-}tw), Z(16, 16, 1, 16, no{-}tw)\}$

A.1.4 Tesla 128×128×128

- $\{X(32, 4, 32, 128, tw), Y(4, 32, 4, 128, tw)\}$
- $\{X(16, 8, 4, 32, tw), Y(32, 4, 1, 4, no{-}tw), Z(32, 4, 32, 128, tw)\}$
- $\{X(32, 4, 1, 4, no{-}tw), Z(4, 32, 1, 32, no{-}tw)\}$

A.1.5 Tesla 256×128×128

- $\{X(64, 4, 64, 256, tw), Y(4, 32, 4, 128, tw)\}$
- $\{X(32, 8, 8, 64, tw), Y(32, 4, 1, 4, no{-}tw), Z(32, 4, 32, 128, tw)\}$
- $\{X(32, 8, 1, 8, no{-}tw), Z(4, 32, 1, 32, no{-}tw)\}$

A.1.6 Tesla 512×512×512

- $\{X(128, 4, 128, 512, tw), Y(32, 16, 32, 512, tw)\}$
- $\{X(128, 4, 32, 128, tw), Y(16, 32, 1, 32, no{-}tw)\}$
- $\{X(64, 8, 4, 32, tw), Z(32, 16, 32, 512, tw)\}$
- $\{X(128, 4, 1, 4, no{-}tw), Z(16, 32, 1, 32, no{-}tw)\}$

A.2 Fermi Architecture Plans

A.2.1 Fermi 128×512×1024

- $\{X(64, 2, 64, 128, tw), Y(16, 32, 16, 512, tw)\}$
- $\{X(16, 8, 8, 64, tw), Y(32, 16, 1, 16, no{-}tw)\}$
- $\{Z(32, 32, 32, 1024, tw)\}$
- $\{X(16, 8, 1, 8, no{-}tw), Z(32, 32, 1, 32, no{-}tw)\}$

A.2.2 Fermi 128×1024×1024

- $\{X(64, 2, 64, 128, tw), Y(32, 32, 32, 1024, tw)\}$
- $\{X(16, 8, 8, 64, tw), Y(32, 32, 1, 32, no{-}tw)\}$
- $\{Z(32, 32, 32, 1024, tw)\}$
- $\{X(16, 8, 1, 8, no{-}tw), Z(32, 32, 1, 32, no{-}tw)\}$

A.2.3 Fermi 128×512×512

- $\{X(64, 2, 64, 128, tw), Y(16, 32, 16, 512, tw)\}$
- $\{X(16, 8, 8, 64, tw), Y(32, 16, 1, 16, no{-}tw)\}$
- $\{X(16, 8, 1, 8, no{-}tw), Z(32, 16, 32, 512, tw)\}$
- $\{Z(16, 32, 1, 32, no{-}tw)\}$

A.2.4 Fermi 256×512×512

- $\{X(64, 4, 64, 256, tw), Y(32, 16, 32, 512, tw)\}$
- $\{Y(16, 32, 1, 32, no{-}tw)\}$
- $\{X(32, 8, 8, 64, tw), Z(32, 16, 32, 512, tw)\}$
- $\{X(32, 8, 1, 8, no{-}tw), Z(16, 32, 1, 32, no{-}tw)\}$

A.2.5 Fermi 256×512×1024

- $\{X(64, 4, 64, 256, tw), Y(32, 16, 32, 512, tw)\}$
- $\{X(32, 8, 8, 64, tw), Y(16, 32, 1, 32, no{-}tw)\}$
- $\{X(32, 8, 1, 8, no{-}tw), Z(32, 32, 32, 512, tw)\}$
- $\{Z(32, 32, 1, 32, no{-}tw)\}$

A.2.6 Fermi 512×512×512

- $\{X(64, 8, 64, 512, tw), Y(32, 16, 32, 512, tw)\}$
- $\{X(64, 8, 8, 64, tw), Y(16, 32, 1, 32, no{-}tw)\}$
- $\{Z(32, 16, 32, 512, tw)\}$
- $\{X(64, 8, 1, 8, no{-}tw), Z(16, 32, 1, 32, no{-}tw)\}$