Using GPUs to Compute Large Out-of-card FFTs

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

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

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

ABSTRACT

The optimization of Fast Fourier Transfer (FFT) problems that can fit into GPU memory has been studied extensively. Such on-card FFT libraries like CUFFT can generally achieve much better performance than their counterparts on a CPU, as the data transfer between CPU and GPU is usually not counted in their performance. This high performance, however, is limited by the GPU memory size. When the FFT problem size increases, the data transfer between system and GPU memory can comprise a substantial part of the overall execution time. Therefore, optimizations for FFT problems that outgrow the GPU memory can not bypass the tuning of data transfer between CPU and GPU. However, no prior study has attacked this problem. This paper is the first effort of using GPUs to efficiently compute large FFTs in the CPU memory of a single compute node.

In this paper, the performance of the PCI bus during the transfer of a batch of FFT subarrays is studied and a blocked buffer algorithm is proposed to improve the effective bandwidth. More importantly, several FFT decomposition algorithms are proposed so as to increase the data locality, further improve the PCI bus efficiency and balance computation between kernels. By integrating the above two methods, we demonstrate an out-of-card FFT optimization strategy and develop an FFT library that efficiently computes large 1D, 2D and 3D FFTs that can not fit into the GPU's memory. On three of the latest GPUs, our large FFT library achieves much better double precision performance than two of the most efficient CPU based libraries, FFTW and Intel MKL. On average, our large FFTs on a single GeForce GTX480 are 46% faster than FFTW and 57% faster than MKL with multiple threads running on a four-core Intel i7 CPU. The speedup on a Tesla C2070 is $1.93\times$ and $2.11\times$ over FFTW and MKL. A peak performance of 21GFLOPS is achieved for a 2D FFT of size 2048×65536 on C2070 with double precision.

Categories and Subject Descriptors

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICS'11, May 31–June 4, 2011, Tucson, Arizona, USA.
Copyright 2011 ACM 978-1-4503-0102-2/11/05...$10.00.

General Terms

Algorithms, Design, Performance

Keywords

FFT, DFT, Library, GPU, CUDA

1. INTRODUCTION

FFT is the fast algorithms to compute Discrete Fourier Transform (DFT), which transfers an input series from time or space domain to frequency domain (Inverse DFT does the opposite). FFT reduces the complexity of a DFT from $O(N^2)$, N being the size of input series, to O(Nlog(N)) by recursively adopting a divide-and-conquer approach. FFT is an important tool in spectral analysis, signal processing, data compression and many other fields. Meanwhile, it is frequently the most time-consuming part of a program. This is particularly true for a large sized FFT due to its heavy demand in memory bandwidth and computational resources.

In order to compute an FFT more efficiently, many FFT libraries have been built on both, general purpose CPUs and computation accelerators such as GPUs. Examples of FFT libraries on CPUs include FFTW [1], SPIRAL [2, 3] and Intel's MKL [4], etc. In our tests, FFTW and MKL typically achieve about 10GFLOPS in double precision on an Intel i7 CPU with multi-threading and vectorization enabled. On the GPU side, best-performing FFT libraries include CUFFT [5], a vendor-provided implementation, and several other research FFT libraries, [6, 7, 8]. Benefiting from GPU's high bandwidth of their off-chip memory and abundant ALUs, these FFT libraries usually out-perform their counterparts on CPUs by a large margin. For example, CUFFT can achieve more than 50GFLOPS in double precision on a high end GPU, Tesla C2070.

The impressive performance of the current GPU-based FFT libraries has a prerequisite, however, that is *all* the input and output data of the computation must reside in the GPU's off-chip memory. Before those libraries are called, all the input data of an FFT has to be transferred from the system memory to the GPU by the user. After the library is called, the output data of the FFT needs to be transferred back as well. This prerequisite leads to two implications. First of all, those libraries cannot handle very large FFT problems, which happen to be required in many applications such as large-scale physics simulations. The maximum problem size for those libraries is limited by the size of the GPU memory. The latest NVIDIA's GeForce GPU, GTX480, has 1.5GB global memory which can only hold a 3D out-of-place FFT of size 256^3 with double precision. Second, the performance advantage of those GPU FFT libraries over their CPU counterparts will be discounted when the data transfer is counted in. The CPU-GPU data channel, in most cases the PCI bus, has larger latency and smaller bandwidth than

the GPU memory. For example, PCIe 2.0 with 16 lanes has only a theoretical peak bandwidth of 8GB/s, only 10% of the 80GB/s bandwidth of the GPU off-chip memory.

Computing large FFTs that cannot fit into the GPU memory needs to transfer the FFT data back and forth over PCI bus at least $\frac{2\times problem_size}{GPU_memory_size}$ times. This is because each data point of an FFT's output mathematically depends on *all* the input data. Therefore, when part of the input is copied to the GPU memory, only intermediate results can be calculated. All the intermediate results on the CPU need at least another round of GPU computations to get the final FFT output. In particular, the same argument applies to not only 1D but 2D and 3D FFTs as well. Another challenge of the problem is that the effective bandwidth of the CPU-GPU data channel is sensitive to the layouts of the data array. Without optimization, a naive implementation of data transfer needs to transfer the FFT data in many non-contiguous chunks in the CPU memory and each PCI transfer can be small. A large number of small transfers is particularly inefficient for the CPU-GPU data channel.

Previous works have studied FFTs on external or hierarchical memory. Bailey [9] proposed a two-round algorithm to compute 1D FFT on a hierarchical memory system including a solid state disk and main memory. However, streaming technique that could overlap the memory communication with FFT computation was not introduced in that work. Moreover, the data transfer between the disk and the main memory is unoptimized in the previous work, while it could be further optimized by a blocked buffer on the disk as suggested later in this paper.

There are cluster based works that can compute large sized FFTs [10, 11]. Particularly, Chen et.al reported the implementation of a large 3D FFT on a 16-node GPU cluster [12]. That work used CUFFT as their solution of in-node FFT and optimized the 3D FFT on a particular cluster architecture. Most cluster based FFT implementations are limited to 3D FFTs, which have abundant natural parallelism. Hence, high performance can be easily achieved with more compute nodes. Most important, the effective PCI bus bandwidth during the transfer of FFTs, a challenge that cannot be bypassed for in-node FFT implementation, is not well studied nor optimized.

This paper is the first effort to address these unique challenges in the implementation of large out-of-card FFTs on a single GPU. The paper makes two main contributions: (1) We propose a Cooley-Tukey algorithm based decomposition framework that co-optimizes both CPU-GPU data transfer and balance of on-GPU computation for 1D, 2D, and 3D FFTs. (2) We develop a blocked buffer technique for 1D FFTs to achieve a high effective bandwidth on the CPU-GPU data channel. Moreover, this technique may be applied to more general data transfer problems as well.

2. OVERVIEW AND BACKGROUND

This work targets FFT problems whose input and output data is larger than the GPU memory and therefore is allocated in system memory. A key difference between our work and other on-card GPU libraries is that these libraries only need to optimize the computation on GPU but our library needs to optimize the data transfer over the PCI bus as well. The data transfer for an on-card GPU FFT library is quite straightforward and is done by the library users. the whole input data allocated on the CPU memory is copied into the GPU in one pass. After the library finishes the FFT computation, the output is transferred back to the host, again, in one pass. However, in this work, the data transfer itself needs to be optimized along with the on-card FFT computation. The interface of our library is on the CPU side just like FFTW or MKL. Specifically, this work deals with double precision complex 1D, 2D and 3D FFTs with power-of-two sizes, which are the most commonly used FFT problems. The input and output arrays of an FFT are allocated as unpagable CPU memory, also called pinned memory, to maximize PCI bus transfer bandwidth.

In this paper, FFT problems and the algorithm we use are denoted in an extended I/O tensor format [8, 13]. We use this concise representation to specify an FFT of any dimension. Optimization algorithms are described as transformations in this representation space as well. In short, an I/O dimension d_1 is defined as $d_1 = d(n, i, o, I, O)$, where n is the FFT size, i and o are the input and output strides and I and O are the addresses of the input and output arrays. One I/O dimension represents the FFT problem on one dimension, and a sequence of I/O dimensions compose an I/O tensor $t = \{d_1, d_2, ..., d_p\}$ which can neatly represent a multi-dimensional FFT. The two pointers, I and O, specify where to store the data (on CPU memory, GPU's global or shared memory) and whether the computation is in-place or out-of-place. However, in this work they are not shown because it is evident that during a PCI transfer one of the pointers is on CPU memory and the other is on GPU memory. Moreover, we can tell that an I/O dimension needs to be out-of-place if the input stride is not equal to the output stride, which suggests there is a transposition. Otherwise, it does not matter whether the I/O dimension is in-place or out-of-place. Without showing the I/O pointers, a 2D FFT of size Y $\times$ X can be represented as $t = \{d(Y, X, X), d(X, 1, 1)\}$ in tensor format.

FFT transforms used in our method are based on the Cooley-Tukey algorithm [14] which decomposes a single $r\times m$ sized FFT into three steps. First, compute r number of FFTs of size m. Second, transpose r with m and multiply a constant matrix called twiddle factors on the intermediate results. Finally, compute m number of FFTs of size r. The Cooley-Tukey algorithm can be precisely represented in tensor form in equation (1), called Decimation In Time(DIT), or equation (2), called Decimation In Frequency(DIF), depending on where the transposition is performed.

$$\{d(rm, i, o)\} = \{d(m, ir, o)t_r^m d(r, mo, mo)\} \quad (1)$$

$$\{d(rm, i, o)\} = \{d(r, im, im)t_m^r d(m, i, ro)\} \quad (2)$$

Here t_r^m represents multiplication of twiddle factors with size $m \times r$. In the real computation, this part will be combined with the adjacent computation steps. These two tensor representations are the basic components of the Cooley-Tukey algorithm and they can be recursively applied in all four direct FFT parts in equation (1) and (2). Different combinations of how to apply the decomposition will derive a DIT, a DIF or hybrid algorithm.

After decomposing a large sized FFT using the Cooley-Tukey algorithm, the smaller sub-FFTs can usually fit into the GPU memory. A batch of such FFTs, possibly with a stride, is transferred to the GPU and is computed using our own FFT kernels, sometimes along with the twiddle factor multiplication. In some cases, NVIDIA's CUFFT is used to solve small 1D and 2D FFTs with a stride equal to one, i.e., the data is contiguous in memory, when CUFFT is faster than our own kernel. Our FFT kernels are optimized using the tensor representation and the Cooley-Tukey algorithm. Specifically, a FFT is recursively decomposed until it is small enough to be directly solved efficiently. Typically, the sizes are 4 to 16 on our tested GPUs.

We have therefore two levels of decomposition: the decomposition of large FFT problems into subproblems that can fit into a GPU and the further decomposition of those on-card FFTs to optimize the kernel computation. A particular way of performing these two levels of decomposition will deliver a different implementation of the target FFT problem. The result of the decomposition will be denoted as a sequence of FFTs in tensor format, which represents all

necessary FFT computation, transposition and twiddle factor multiplication. Please note that a sequence of I/O dimensions implicitly determines the number of FFTs for each I/O dimension. For example, there are $\frac{n_1 \cdot n_2 \cdot \ldots \cdot n_j}{n_i}$ number of FFTs of size n_i for the I/O dimension $d(n_i, i, o)$ in a sequence of j I/O dimensions.

We follow the same approach as is shown in paper [8] for on-GPU optimization. To summarize, specially revised codelets [15], [16] , which are compiler generated C programs to solve small FFTs in FFTW, are used to compute FFTs on a GPU. High dimensional FFTs are computed on that dimension without being transposed to a lower dimension first. Codelets within one FFT dimension or across multiple dimensions are grouped into the fewest number of kernels. Each kernel has one pass of global and multiple passes of shared memory accesses. Therefore, the overall global memory accesses are minimized. Moreover, 16 or 32 threads are coalesced into a single memory transfer when accessing adjacent data so that a higher global memory bandwidth is achieved.

The new architectural features in the latest Fermi GPUs [17] also affect optimization decisions. Compared with older GPUs, Fermi introduces a larger shared memory size(48KB vs 16KB), more banks in shared memory(32 banks vs 16 banks) and more coalescing threads(32 threads vs 16 threads). Those new features are incorporated into the optimization through the parametrization of the algorithm. Fermi's newly added L1 and L2 cache, is not really helpful to FFT because of its highly regular and non-repeating data access pattern. Concurrent kernel execution is not applicable either because the kernels corresponding to a sequence of I/O dimensions (or codelets) have data dependency among each other, and therefore can not be executed concurrently.

Overall, we use FFT decomposition algorithms based on the I/O tensor framework to maximize the data transfer PCI bandwidth and balance computation kernels to have better overlap with communication. As we will show later, the transfer of subarrays over the PCI bus can have an order of magnitude difference in effective bandwidth depending on the width of the subarray. With our proposed FFT algorithms, a close to optimum PCI bus bandwidth is achieved and computation kernels between rounds are of comparable size. The last key component in the optimization of large FFTs is a blocked buffer algorithm which is applicable to a wide class of data transfer problems. Particularly, this algorithm is used to increase the effective PCI bus bandwidth of 1D FFTs when other FFT related optimizations are not applicable.

3. PCI TRANSFER OF SUBARRAYS

In this section, we illustrate the general scheme of how to move partial data of a large FFT between CPU and GPU memory. Several methods are proposed to improve the performance of such data movement over a communication channel such as the PCI bus.

When the FFT is larger than the GPU memory size, only a portion of the whole data can be transferred each time. If we have a large high dimensional FFT or a 1D FFT divided by the Cooley-Tukey algorithm, a batch of the smaller FFTs will take a block of subarrays within the original large array as input. Since these subarrays have the same length and a constant stride between each other, we call them *regular subarrays*. Figure 1 shows C regular subarrays of length W and stride X between each other in a large array of size $C \times X$. Note that the large array is contiguous on the X dimension, and the regular subarrays as a whole are not contiguous in system memory. But they need to be copied to a single contiguous array in GPU memory, as is shown on the right side of Figure 1.

This large array in CPU memory can actually be a 1D, 2D or 3D array but is just shown in a 2D point of view. Assume the total size

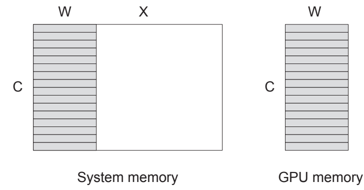

Figure 1: regular subarrays within a large array

of the large FFT is $C \times X$. For a large 1D FFT, the regular subarrays contain a part of the small FFTs in one computation step of the Cooley-Tukey algorithm, i.e. W FFTs of length C. If the large FFT is a 2D problem of size $Y \times X$, then the regular subarrays comprise part of the Y dimensional FFTs, where $Y = C$. In order to copy the whole chunk of subarrays to or back from GPU memory, there need to be C number of *cudaMemcpyAsync()* function calls and each call copies a subarray of width W. As we want to use as much GPU memory as possible to reduce passes of subarray transfers, $C \times W$ is usually chosen to be the largest value allowed on the GPU memory. For example, if the FFT problem has an input and an output arrays with complex double precision data type, the maximum power-of-two size of $C \times W$ is 32M on a NVIDIA's GTX480 with 1.5GB global memory,

The choice of how a large FFT problem is divided has a great impact on the data transfer performance. In other words, even if we know the maximal value of $C \times W$, the different choices of C or W can lead to almost one magnitude difference in transfer time. As we fix the $C \times W = 32M$, for example, and change W from 8 to 32M (C changes accordingly), the effective PCI bandwidth is illustrated in Figure 2. The curves 'H2D' (host to device, i.e., CPU

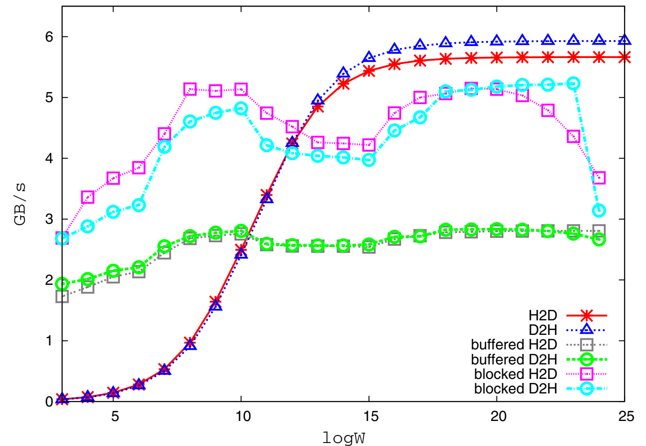

Figure 2: PCI bandwidth test of subarray transfer

memory to GPU memory) and 'D2H' (device to host) in figure 2 show the overall effective bandwidth of the transfer with different W. The value of W is shown in log_2 scale and $16 \times W$ is the actual width in Bytes. These two curves show that PCI bus keeps a fairly high bandwidth (about 6GB/s out of 8GB/s theoretical peak) with large W and a small numbers of *cudaMemcpyAsync()* calls. The PCI bandwidth, however, quickly decreases by an order of magnitude when W decreases to be smaller than 2^{13} to 2^{14}. We also consider whether the stride between two adjacent subarrays, i.e. X value, will affect the transfer performance or not. However, test results show that it does not have much influence on the effective bandwidth of the PCI bus.

In order to increase the effective PCI bandwidth of the subarray

transfer for small W values, we introduce a buffer on CPU memory. The buffer has the same total size of the regular subarrays. All the data on the subarrays is transferred to this buffer first and then the whole buffer is copied to GPU using a single PCI transfer. Device to host transfer is exactly the opposite process. The benefit of this approach is that the rearrangement of the subarrays to a continuous chunk of memory using *memcpy()* can lead to lower total overhead than transferring subarrays directly using additional *cudaMemcpyAsync()* calls. The host to device and device to host bandwidth with a buffer on system memory is shown as the curves 'buffered H2D' and 'buffered D2H' in figure 2. This method generally achieves a bandwidth of 2GB/s to 3GB/s and can improve the performance of the direct transfer method when W is smaller than 2^{11}.

To further increase the bandwidth, the preparation of the buffer can be done concurrently with the the transfer of the buffer over the PCI bus. The key issue is the synchronization between the two steps. A further optimization we propose is that the buffer on system memory can be divided into blocks as is shown in figure 3 and we can overlap the preparing time of a block with the PCI transfer time of another block. More specifically, when *memcpy()* is

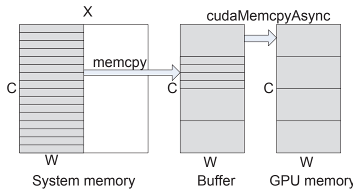

Figure 3: PCI transfer of subarrays with blocked buffer

used to transfer a block of data from the large array to the buffer, *cudaMemcpyAsync()* can be used to transfer the previous block to GPU at the same time. Additional threads are needed for the device to host subarray transfer. First, *cudaMemcpyAsync()* is called to move a block of data from the GPU memory to the buffer. Then after calling the cudaStreamSynchronize() function, a signal is sent by the first thread and the second thread will handle the data movement of the block from the buffer to the large global array. The curves 'blocked H2D' and 'blocked D2H' in figure 2 show the effective bandwidth of transferring subarrays using a blocked buffer. The optimum number of blocks for each (C,W) pair is found by an empirical search. As a result of the overlap between the two steps of data movement, this method can greatly increase the PCI bandwidth to about 5GB/s when W is smaller than 2^{12}.

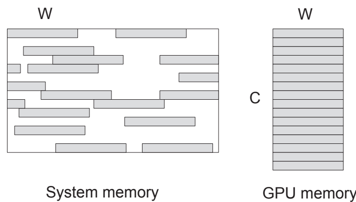

Figure 4: More general case of subarray transfer

So far, we have presented multiple methods to increase the effective PCI bus bandwidth of transferring regular subarrays. The proposed blocked buffer algorithm can be applied to communication channels other than PCI bus and much more general random-stride subarray transfer problems as is shown in figure 4. For regular subarrays with W larger than 2^{12}, we can improve the PCI bus bandwidth to more than $3GB/s$ without using any FFT related optimization. We will shown in the following sections that we can further improve the PCI bandwidth of FFT problems to almost optimum by co-optimizing the Cooley-Tukey algorithm based decomposition and the parameter tuning of the PCI transfer step.

4. LARGE 1D FFT

First, we will discuss the computation of 1D FFTs with size larger than GPU memory. 1D FFT has no natural parallelism that can be extracted from simple problem division. However, the Cooley-Tukey decomposition algorithm can be applied on a large 1D FFT to get smaller sized 1D sub-problems that can fit into GPU memory and be computed on card. As is discussed before, at least two rounds of partial computation is needed for a large sized FFT. The simplest way to compute a large sized 1D FFT is to decompose the original problem once and compute the sub-problems with a two-round algorithm.

4.1 Two-Round 1D FFT algorithm

A 1D FFT of size $X = X_1 \times X_2$, represented as $d(X, 1, 1)$ in the I/O tensor format, is divided with the DIT Cooley-Tukey algorithm represented in equation (1) by choosing $m = X_1$ and $r = X_2$. Then, we have the following decomposition equation, $d(X_1, X_2, 1)t_{X_2}^{X_1}d(X_2, X_1, X_1)$. Figure 5 shows the overview of our two-round large 1D FFT algorithm. The computation of a

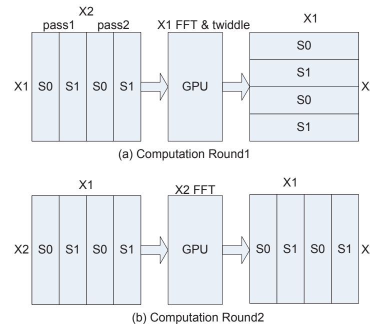

Figure 5: Two-round computation for large 1D FFT

large 1D FFT is divided into two rounds. Each round includes several passes of GPU computation and the number of passes is $\frac{X \times data_size}{GPU_memory_size}$.

Each pass of GPU computation can use multiple streams to overlap the computation with communication. For example, figure 5 shows the case of using two streams, S0 and S1, in each round. The relation between the stream size, the number of streams and the number of passes is shown in equation (3).

$$stream_size \times (\#streams) \times (\#passes) \leq GPU_memory_size \quad (3)$$

We use empirical search to find the optimum number of streams and the stream size.

For the host to device data transfer, round one needs $C = X_1$ number of PCI transfers and a width of $W = \frac{X_2}{(\#passes) \times (\#streams)}$ for each transfer. For example, if $X_1 = X_2 = 2^{13}$, W will be smaller than 2^{11}. The blocked buffer method can be used to perform the host to device subarray transfer and about $5GB/s$ PCI bus bandwidth is achieved according to the results in figure 2.

On the GPU, FFTs of size X_1 are further decomposed by the Cooley-Tukey and are computed using a sequence of codelets. The last codelet is rewritten to include twiddle factors of size $X_1 \times X_2$. The transposition of X_1 to the low dimension is performed on GPU instead of on CPU because of GPU's higher bandwidth. Moreover, this transposition is incorporated in the computation of the codelets with the help of shared memory, so no explicit transposition is needed. Finally, the output of round one is copied back to the CPU in a single PCI transfer. Close to 6GB/s PCI bus bandwidth is achieved in this step.

In round two of the computation, a host to device and a device to host PCI subarray transfer with $C = X_2$ and $W = \frac{X_1}{(\#passes)\times(\#streams)}$ is performed. Similar to the host to device transfer in round one, about 5GB/s PCI bandwidth can be achieved. The number of passes and streams in round two need not to be the same as for round one. No transposition is needed on the GPU in this round.

In order to keep a high PCI bandwidth of all three transfers, we want to keep both the W value large and the C value small. Therefore, X_2 is chosen to be equal or slightly larger than X_1. Another reason to choose equal or close X_1 and X_2 values is that the two computation kernels in the two rounds will be balanced. This helps to overlap the computation time with the PCI transfer time.

4.2 Three-Round 1D FFT algorithm

When the problem size X increases, X_1 and X_2 in the above two-round algorithm will increase, and more subarrays with smaller size need to be transferred in one pass. The decrease of width of subarray will hurt bandwidth significantly in both rounds. In this case, we propose another alternative algorithm using three rounds of PCI bus transfer with higher bandwidth for each round. If $X = X_1 \times X_2 \times X_3$, applying the DIT Cooley-Tukey algorithm (1) twice gives equation (4), which suggests a three-round 1D FFT algorithm.

$$\{d(X_1, X_2X_3, 1)t^{X_1}_{X_2}d(X_2, X_1, X_1) \\ t^{X_1X_2}_{X_3}d(X_3, X_1X_2, X_1X_2)\} \quad (4)$$

There are other valid decomposition schemes that can be derived from recursively applying DIT or DIF algorithm on different parts of the previous equation.

The three-round computation of equation (4) is illustrated in figure 6. Different data transfer strategies are needed for the

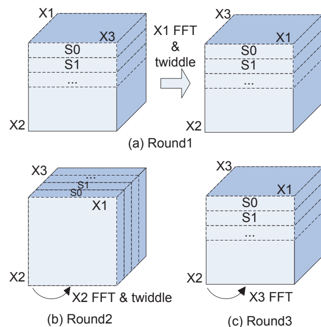

Figure 6: Three-round computation for large 1D FFT

three rounds of computation. In round one, reading from system memory needs $C = X_1$ PCI transfers, each with a width of $W = \frac{X_3X_2}{(\#passes)\times(\#streams)}$. Writing backs in round one and both accesses in round three have $C = X_3$ and $W = \frac{X_1X_2}{(\#passes)\times(\#streams)}$. To keep W large and C small, X_2 is chosen to be larger than both X_1 and X_3. Arrays that need to be transferred in round two occupy a contiguous chunk of system memory. Only one PCI transfer is needed for round two, so it already has the best bandwidth over the PCI bus.

By adding one more round of GPU computation, the bandwidth of each round is improved. However, this three-round 1D FFT algorithm is only beneficial for very large 1D FFTs where a two-round algorithm has low PCI bandwidth. Thus the overhead of adding one round of PCI communication is justified. In our case, due to the limitation of our system memory, the largest 1D FFT we can test is 256M and a two-round algorithm still has a good PCI transfer bandwidth for this size. So this three-round 1D FFT algorithm should be used for even larger 1D FFTs.

5. LARGE 2D FFT

5.1 Naive 2D FFT algorithm

For a large 2D FFT of total size $N = Y \times X$ where $16N$ is larger than the GPU's global memory size, there are naturally 2 rounds of computation by definition, i.e. Y dimensional FFTs and X dimensional FFTs. One can easily takes advantage of this natural parallelism and compute a batch of these smaller sub-FFTs on the GPU if these Y dimension sub-problems or X dimensional sub-problems can fit into the GPU memory. The data layout in system memory of this straightforward algorithm is shown in figure 7. Y dimensional

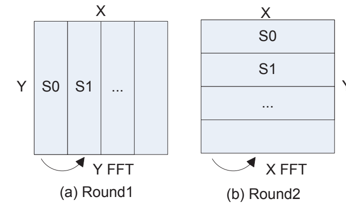

Figure 7: Naive algorithm for large 2D FFT

FFTs are computed in round one and X dimensional FFTs in round two. Similarly, each round is divided into several passes and each pass has a couple of streams. Round two has no PCI bandwidth issue because each stream needs only one PCI transfer between host and device. In round one, there are $C = Y$ number of reads from and writes to subarrays on system memory. Each access over PCI has a width of $W = \frac{X}{(\#passes)\times(\#streams)}$. According to the PCI performance curves in figure 2, when X is large and Y is small, the data transfer pattern has a good PCI bandwidth. However, this naive algorithm may lose up to 50% of the peak PCI performance when Y is large. Moreover, the amount of GPU computation between two rounds is also unbalanced when Y is much larger than X. Therefore, this naive algorithm is only used when Y is small.

5.2 Y decomposition 2D FFT algorithm

In order to increase the effective PCI bus bandwidth of round one in the naive 2D FFT algorithm, we apply the Cooley-Tukey algorithm once on the Y dimensional FFTs. The original 2D FFT in tensor form, $\{d(Y, X, X), d(X, 1, 1)\}$, becomes equation (5) for $Y = Y_1 \times Y_2$.

$$\{d(Y_1, Y_2X, X)t^{Y_1}_{Y_2}d(Y_2, Y_1X, YX)d(X, 1, 1)\} \quad (5)$$

FFTs of size Y_1 and Y_2 are further decomposed on the GPU using the Cooley-Tukey and are eventually computed by codelets. In particular, we rewrite the last codelet of the Y_1 codelet sequence

so that the twiddle factor computation between Y_1 and Y_2 can be computed within that codelet. X dimensional FFTs are computed using the CUFFT library due to its good on-GPU performance for 1D FFTs.

Figure 8 visualizes this decomposition in equation (5). FFTs of

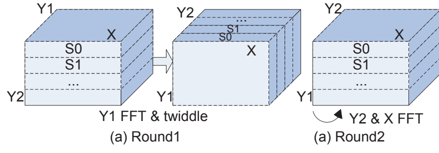

Figure 8: Y decomposition algorithm for large 2D FFT

size Y_1 is computed and a transposition between Y_1 and Y_2 is performed on GPU in round one of the algorithm. The device to host copy of round one needs a single PCI transfer (C=1) and thus has the best PCI bandwidth. The host to device copy in round one and the two-way communications in round two need C=Y_1 and C=Y_2 number of PCI transfers. All of them have smaller C and better bandwidth than the naive algorithm. To achieve high bandwidth for both rounds, both Y_1 and Y_2 need to be small. Round one computes Y_1 sized FFTs and round two computes Y_2 and X sized FFTs. We want the amount of computation in two rounds to be similar so that they can be better hidden by the PCI communication. The best choice depends on the size of the 2D FFT and is found empirically.

Suppose there is a 2D FFT of size $Y = X = 8192$ with $\#passes = \#streams = 2$, $Y_1 = 128$ and $Y_2 = 64$. Round one in the naive algorithm has a copy width of $W = 2^{11}$ for each PCI transfer, and roughly 4.7GB/s host to device and 4.2GB/s device to host PCI bus bandwidth can be achieved if the blocked buffer algorithm is used. Without a blocked buffer, only 3.4GB/s and 3.3GB/s are available for direct PCI transfers. The Y decomposition 2D FFT algorithm, however, has a copy width of $W = 2^{17}$ for the host to device transfers in round one and a bandwidth of 5.6GB/s is achieved by a direct PCI transfer. Round two of the algorithm has a copy width of $W = 2^{18}$ for each PCI transfer and can achieve 5.6GB/s and 5.9GB/s for each direction. In other words, close to theoretical peak bandwidth is achieved in all two rounds of PCI transfers by using this Y decomposition 2D FFT algorithm.

A special case where the above 2D FFT algorithms will not work is when the X dimension of a 2D FFT is larger than the GPU memory size. In this case, a naive algorithm needs two rounds PCI transfer and GPU computation just for the X dimensional FFTs and another round for the Y dimensional FFTs. The solution we propose is to decompose the X dimensional FFT using our large 1D FFT algorithm and combine the Y dimensional FFT into one of the two rounds of X dimensional computation. Therefore, still only two rounds of PCI transfer are needed.

6. LARGE 3D FFT

6.1 Naive 3D FFT algorithm

Similar to the 2D FFT case, the natural parallelism in 3D FFTs of size $X \times Y \times Z$ can be translated into a straightforward way to compute large sized 3D FFTs on a GPU if $X \times Z$ and $X \times Y$ can fit into the GPU memory. The data layout in system memory of the naive two round algorithm is shown in figure 9. Z dimensional FFTs are computed in the first round and Y dimensional FFTs in the second round. X dimensional FFTs can be combined into either round depending on which one has less computation. Again the on-GPU Z and Y dimensional FFTs are further decomposed

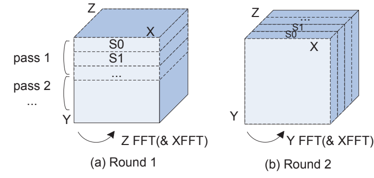

Figure 9: Naive algorithm for large 3D FFT

using the Cooley-Tukey algorithm and are computed by our own kernels. X dimensional FFTs are computed by calling CUFFT for better performance. Each stream in round two accesses a continuous chunk of data on the host memory and therefore has optimum PCI bandwidth during data transfer. However, round one has $C = Z$ number of PCI transfers and each transfer has a data width $W = \frac{XY}{(\#passes)\times(\#streams)}$. For large Z and small $X \times Y$, this round of PCI transfer will have a low effective PCI bus bandwidth according to figure 2. We need to further decompose the Z dimension to avoid this scenario.

6.2 Z decomposition 3D FFT algorithm

Similar to the Y decomposition 2D FFT algorithm, we apply the Cooley-Tukey algorithm once on Z dimensional FFTs of a 3D FFT when Z is large. For $Z = Z_1 \times Z_2$, a direct 3D FFT represented as $\{d(Z, XY, XY)d(Y, X, X)d(x, 1, 1)\}$ is transformed into the tensor format as is shown in equation (6) after applying a DIT algorithm on Z.

$$\{d(Z_1, XYZ_2, XY)t^{Z_1}_{Z_2}d(Z_2, XYZ_1, XYZ_1) \\ d(Y, X, X)d(X, 1, 1)\} \quad (6)$$

The system memory layout of this Z decomposition algorithm for 3D FFT with large Z is illustrated in figure 10. The PCI bandwidth

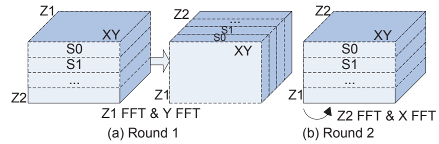

Figure 10: Z decomposition algorithm for large 3D FFT

analysis of this algorithm is similar to that of the Y decomposition 2D FFT algorithm except that Y becomes Z and X becomes $X \times Y$ in this case. A close to optimum PCI bandwidth is achieved for all four transfers between the host and device if Z_1 and Z_2 are chosen to be close.

Strictly following equation (6), only FFTs of size Z_1 will be computed in round one and all Z_2, Y and X kernels will be in round two. The computation will be extremely unbalanced between the two rounds. Particularly, the computation time in round two will be difficult to be hidden in the communication time. Instead, we apply an optimization proposed in paper [8] to rearrange the computation order. The general reorder rule is that the order of these I/O dimensions can be adjusted as long as FFTs of size Z_1 appears before FFTs of size Z_2. Therefore, we can exchange the I/O dimension $d(Z_2, XYZ_1, XYZ_1)$ and $d(Y, X, X)$ and still have the correct output. The result of this reordered 3D FFT is shown in formula (7).

$$\{d(Z_1, XYZ_2, XY)t^{Z_1}_{Z_2}d(Y, X, X) \\ d(Z_2, XYZ_1, XYZ_1)d(X, 1, 1)\} \quad (7)$$

Now, Y dimensional FFT is computed in the first round of computation and X dimensional in the second round as is shown in figure 10. Moreover, size Y FFTs and size X FFTs can be exchanged as well in order to best balance the computation between two rounds.

There are two corner cases of large 3D FFTs that can not be handled by the above two algorithms. One is that $X \times Y$ is larger than the GPU's memory size (but X is still smaller). The other one is that X alone is larger than the GPU memory size. In the first case, the Y dimensional FFT can be decomposed and, in the second case, X can be decomposed using the Cooley-Tukey.A two-round decomposition algorithm can be derived using the I/O tensor representation to achieve a good bandwidth over PCI. These two-round decomposition schemes are similar to our large 1D FFT and the large 2D FFT algorithms and therefore are not further discussed here.

7. EVALUATION

In this section, we present the performance of our large 1D, 2D and 3D FFT implementations on three NVIDIA GPUs, i.e. GeForce GTX480, Tesla C1060 and Tesla C2070. Of the three GPUs, the GTX480 and the C2070 are based on the latest GF400/Fermi architecture while the C1060 is based on the slightly older GT200 architecture. Details of the three NVIDIA GPUs and the related configurations of their host systems are listed in table 1.

The host machine of the GTX480 and C2070 has a high-end Intel i7 920 CPU. The performance of FFTW and MKL on this CPU is compared with that of our FFT library on the two Fermi-based GPUs. The host CPU for the C1060 is an Intel server CPU Xeon E5405. The configuration of the CPUs and the CPU based FFT libraries are listed in table 2. In FFTW, the support of the Single Instruction Multiple Data (SIMD) extension is enabled to take advantage of the vector instructions (SSE2) on the Intel CPUs. FFTW's *patient level*, choices of search method and search space size, is set as 'MEASURE'. This method takes around 30 minutes on searching for large sized FFTs in our test but it provides much better performance than a lower patient level, 'ESTIMATE'. Since all host CPUs are multi-core processors, multithreading is enabled and the performance with different number of threads is shown for both FFTW and MKL. In short, the performance of FFTW and MKL is configured to be their best on our test systems.

Currently this work handles 1D, 2D and 3D FFTs with power-of-two sizes, and other sizes can be easily included by adding codelets of different sizes. The FFT computation is performed out-of-place, which preserves the input array. The input and output are complex number and can be in either double precision or single precision. Only double precision results are shown due to its higher accuracy, which is valuable particularly for large FFT sizes. The correctness of our FFT's output is checked by a comparison with FFTW and MKL on the same input.

The range of the FFT size in our test is determined by both the size of the system memory and the size of the GPU memory. The smallest test case is the smallest FFT problem that cannot fit into GPU memory or computed by CUFFT, and the largest test case is the largest power-of-two FFT that can fit into pinned system memory. For a D dimensional out-of-place complex FFT with double precision, the total number of elements $M = N_1 \cdot N_2 \cdot \ldots \cdot N_D$ should be equal or larger than 32M for GTX480, 64M for C1060 and 128M for C2070 and should be equal or smaller than 256M due to the available CPU memory on our system. In fact, our algorithm is able to handle much larger FFTs if given more system memory. Finally, as a convention, the performance of FFT is reported in GFLOPS defined in equation (8) where t is the execution time in seconds.

$$GFlops = \frac{5M\sum_{d=1}^{D} log_2 N_d}{t} * 10^{-9} \quad (8)$$

7.1 Performance of Large 1D FFTs

Figure 11 shows the performance of 1D FFTs of size 32M to 256M in log_2 scale on GTX480 and C2070. FFTW and Intel MKL's performance on the host system with an Intel i7 920 CPU is listed as a comparison. FFTW shows good scalability when the number of threads is within 4. FFTW with 8 threads has similar performance to 4 threads on this 4-core CPU, and both achieve around 6 to 7.5GFLOPS. MKL has similar scalability but performs significantly worse than FFTW on sizes of 128M and 256M. Our large 1D FFT on the GTX480 achieves 11GFLOPS, on average, and is 62% faster than FFTW and $2.4\times$ faster than MKL with 4 threads. Compared with a single thread on a CPU, our GPU FFT is $4.5\times$ faster than FFTW and $3.3\times$ faster than MKL. Our peak performance of 15.5GFLOPS is achieved on the size of 32M. This problem size can actually fit into the GPU but CUFFT is somehow unable to compute it. In this particular case, our approach needs only one round of PCI transfer although a global synchronization in between two rounds of computation remains necessary.

A Tesla C2070, on the other hand, supports duplex communication on the PCIe bus, which theoretically has bi-directional bandwidth of 16GB/s. However, its 1D FFT performance is only 10% to 15% faster than that of GTX480. A possible reason is that duplex PCI transfers cannot properly overlap with a memory-bound kernel like FFT, because host to device transfer, device to host transfer and kernel accesses on global memory may result in a bottleneck in global memory bandwidth or its memory control unit. For the two largest 1D FFTs, the C2070 achieves 11GFLOPs and is $1.7\times$ times faster than FFTW and $3.55\times$ faster than MKL with 4 threads. Compared with a single CPU thread, the C2070 is $4.5-4.8\times$ faster.

Figure 12 shows the performance of 1D FFTs of size 64M to 256M on a Tesla C1060. Intel MKL crashes when performing a 1D FFT of size 256M and therefore this point is not shown. On the Intel Xeon E5405, FFTW with 4 threads achieves 3.6GFLOPS and MKL achieves 3.0GFLOPS for large 1D FFTs on average. Our library on the C1060 achieves 4.2GFLOPS, which is 16.7% faster than the 4-thread FFTW but is much worse than our library on Fermi GPUs. This is because the C1060 has much fewer double precision ALUs compared with the Fermi cards. The double-precision performance of pre-Fermi GPUs is about only 10% of the single-precision performance.

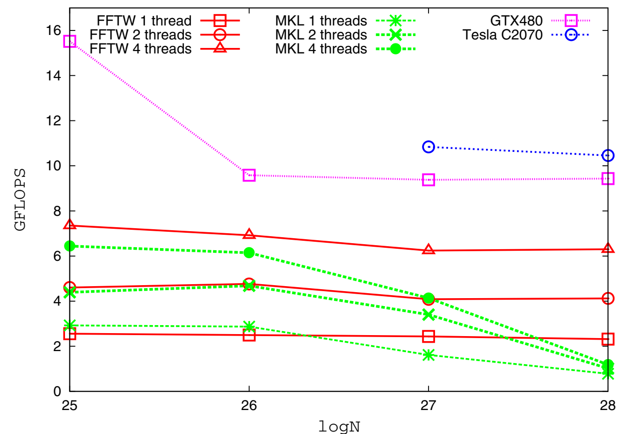

Figure 11: 1D FFT on GTX480 and Tesla C2070

GPU	Global Memory	CUDA Driver	Nvcc & Cufft	PCI	System Memory
GeForce GTX480	1.5GB	260.19.21	3.2	PCIe2.0 x16	12GB
Tesla C2070	6GB	260.19.21	3.2	PCIe2.0 x16	12GB
Tesla C1060	4GB	256.53	3.1	PCIe2.0 x16	9GB

Table 1: Configuration of GPUs

CPU	Frequency	Cores	GCC	FFTW	MKL	System Memory
i7 920	2.66GHz	4	4.4.3	3.2.2	10.2.6	12GB
Xeon E5405	2.00GHz	4	4.1.2	3.2.2	10.2.6	9GB

Table 2: Configuration CPUs and FFT libraries

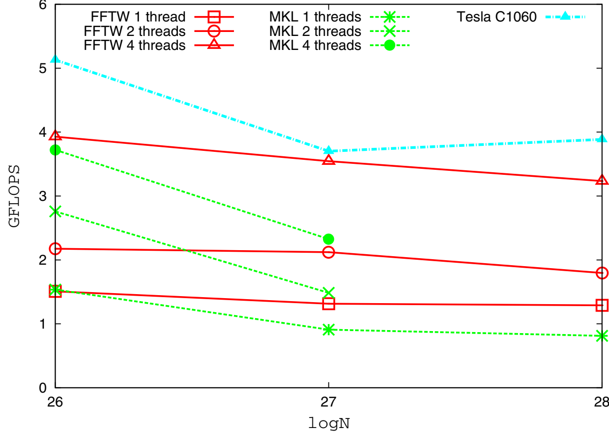

Figure 12: 1D FFT on Tesla C1060

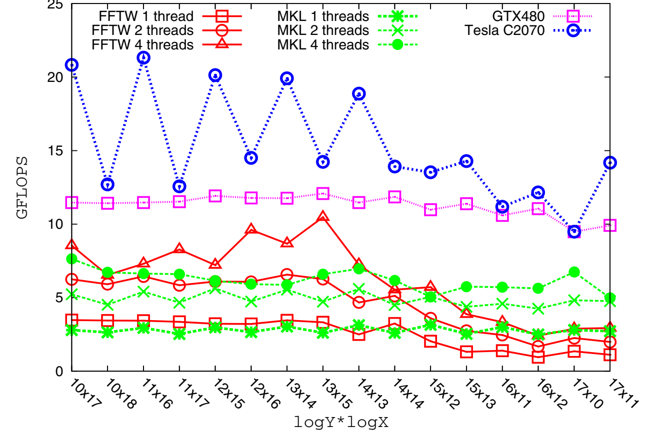

Figure 14: 2D FFT $Y \times X \geq 128M$ on GTX480 and Tesla C2070

Moreover, C1060's best PCI bandwidth of a single array transfer (when C=1 and W=64M) is only 5.6GB/s for host to device and 4.3GB/s for device to host transfers, which is much worse than that of Fermi GPUs. As a result, our blocked buffer algorithm achieves lower PCI bus bandwidth on the C1060 as well.

7.2 Performance of Large 2D FFTs

This section shows the performance of 2D FFTs with size $Y \times X$, where $32M \leq Y \times X \leq 256M$. Due to a large number of test points, the results on GTX480 are divided into two figures. Figure 13 shows the 2D FFTs where $Y \times X \leq 64M$ and figure 14 shows those $Y \times X \geq 128M$. Different $Y \times X$ sizes are indexed in an increasing order of Y. Similar to 1D, FFTW shows a good scalability up to 4 threads in both figures. FFTW with 8 threads has almost the same performance as 4 threads, and therefore is not shown. As we can see, FFTW's performance decreases dramatically from 10GFLOPS to less than 3GFLOPS with the increase of Y due to the loss of data locality in the Y dimensional computation. However, MKL with 2 threads and 4 threads suffers less from the loss of Y dimensional locality on this Intel CPU and performs better than FFTW.

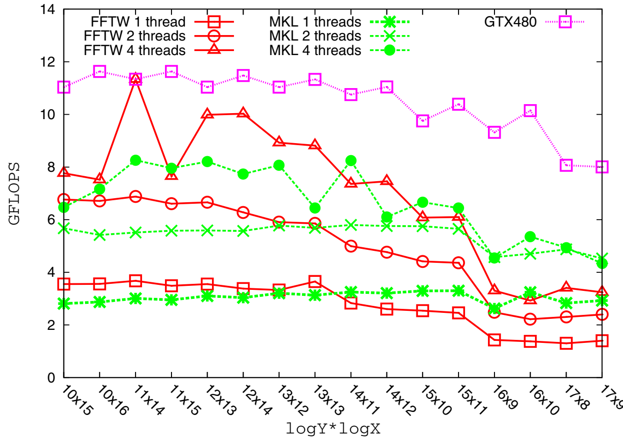

Figure 13: 2D FFT $Y \times X \leq 64M$ on GTX480

Because we divide the Y computation into two rounds, our 2D FFTs on a GTX480 has a much smaller performance decrease with the increase of Y than FFTW. We achieve an average of 10.9GFLOPS, which is 64% and 69% faster than FFTW and MKL with 4 threads. Since CUFFT on the C2070 can compute 2D FFTs with size $Y \times X \leq 64M$, our FFT results on C2070 are only shown in figure 14. The fluctuation in figure 14 is because 2D FFTs of size $Y \times X = 128M$ can fit into the 6GB global memory on the C2070 but somehow cannot be handled by CUFFT. For these sizes, we only need the host to device transfer of round one and the device to host transfer of round two in our algorithm, and can eliminate the PCI transfer of intermediate results. The best 2D FFT performance on the C2070 is 21GFLOPS for a size of 1024×65536. On average, the C2070 achieves 15.2GFLOPS and is 2.3× faster than FFTW and 2.4× faster than MKL with 4 threads on an Intel i7. C2070 has four times more double precision ALU than GTX480, however its speedup over GTX480 is not big for a communication bound problem like FFT.

Figure 15 shows our large 2D FFT test results on the Tesla C1060 and the performance of FFTW and MKL on the Intel Xeon machine. FFTW's performance decreases with the increase of the Y dimension but MKL is much better for these sizes. Our 2D FFT achieves 7.88GFLOPS and is 2.66× faster than FFTW and 2.13× faster than MKL with 4 threads.

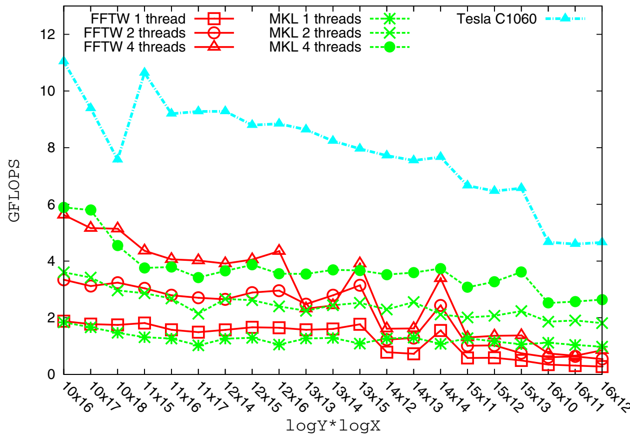

Figure 15: 2D FFT $Y \times X$ on C1060

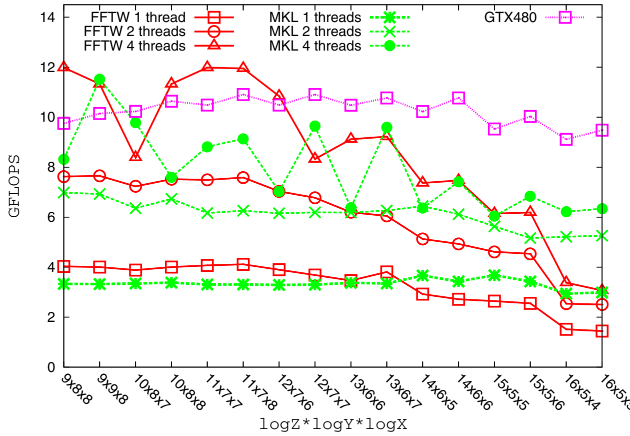

Figure 16: 3D FFT $Z \times Y \times X \leq 64M$ on GTX480

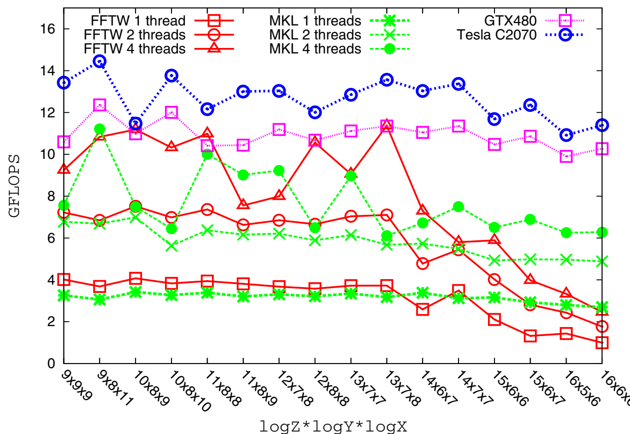

Figure 17: 3D FFT $Z \times Y \times X \leq 128M$ on GTX480 and Tesla C2070

7.3 Performance of Large 3D FFTs

Figures16 and 17 illustrate the performance comparison of our large FFT library with FFTW and MKL for 3D FFTs of size $N = Z \times Y \times X$, where $32M \leq N \leq 256M$. FFTW with 4 threads performs slightly better than our GPU library for small Z values, but its performance decreases quickly as Z increases. MKL's 3D FFTs with 4 threads have similar performance with FFTW and is better than FFTW for large Z. Our high dimensional decomposition algorithm shows a big advantage when the highest dimension of the FFT is larger than 2048. FFTW and MKL quickly lose data

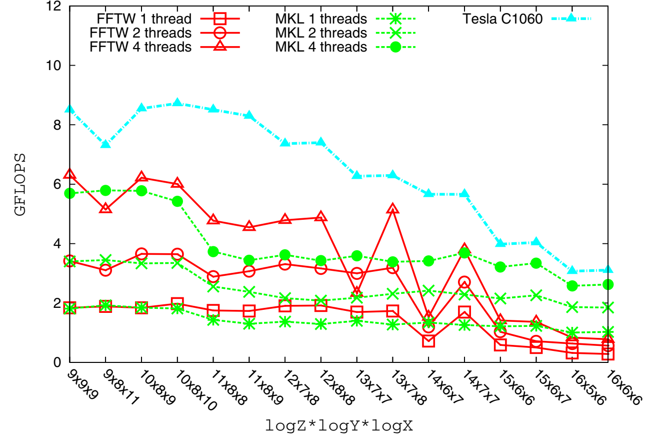

Figure 18: 3D FFT $Z \times Y \times X \leq 64M$ on C1060

locality for these sizes and therefore perform much worse than our GPU library. Overall, our library has an average of 10.6GFLOPS on the GTX480 and is 27% faster than FFTW and 52% faster than MKL. The C2070, on the other hand, achieves 12.7GFLOPS for large 3D FFTs which can be attributed to its duplex PCI communication ability.

The performance of FFTW and MKL on the Xeon E5405 and our 3D FFTs on the Tesla C1060 is shown in figure 18. Compared with the Fermi GPU cards, 3D FFTs on the C1060 have an obvious performance decrease with the increase of the Z dimension. Our Z decomposition algorithm decreases the number of PCI transfers from Z to Z_1 and Z_2. However, with the increase of Z, the number of PCI transfers still increases slowly. The effective PCI bus bandwidth for the C1060 card decreases faster than for Fermi based GPUs. This leads to an obvious decrease in performance for 3D FFTs on the C1060 when Z is large. Overall, the C1060 still achieves 6.4GFLOPS, on average, and is 72% faster than FFTW with 4 threads and and 64% faster than MKL' best performance.

Overall, our GPU based large FFT library is faster than multithreaded FFTW and MKL on almost all 1D, 2D and 3D FFT sizes. The performance of the Tesla C2070 is slightly better than the GeForce GTX480 and both Fermi cards are much faster than the GT200 based Tesla C1060. Across 1D, 2D and 3D sizes, the C1060 has an average performance of 7GFLOPS, a GTX480 achieves 11GFLOPS and the C2070 achieves 14GFLOPS. As a comparison, both FFTW and MKL achieve only 3.5GFLOPS on the Xeon and about 7GFLOPS on i7. We also tested our library with single precision data. Because FFT is a communication bounded problem, both the performance of our large FFT library and CPU based libraries have 50% to 100% speedup over their double precision results. The speedup of our GPU FFT library over the CPU libraries is similar to the double precision case. We do not show these single precision results here simply due to limited space.

7.4 Precision of Our Large FFTs

The correctness of our large GPU FFT library is verified by a comparison with FFTW and MKL. All three libraries are tested with the same double precision pseudo-random input data in the range of $[-0.5, 0.5)$ and the difference in output is quantized as root mean square error (RMSE) over the whole data set. Let us assume there are two complex FFT output arrays of length N, (x_i, y_i) and (X_i, Y_i). The RMSE between these two output arrays is calculated as shown in equation (9)

$$RMSE = \sqrt{\frac{\sum_{i=1}^{N} ((x_i - X_i)^2 + (y_i - Y_i)^2)}{N}} \tag{9}$$

The RMSE is a widely used metric to measure the relative accuracy of two computations. Lower RMSE value means the two computation routines produce more similar result. The RMSE results of the 1D, 2D and 3D FFTs between our GPU FFT and FFTW are shown as the solid bars in figure 19. Overall, the RMSE is extremely small and is in the range of $(1.4\times10^{-12}, 3.6\times10^{-12})$. With the increase of FFT size, the RMSE increase almost linearly. The RMSE between MKL and our GPU FFT library, shown as the shadowed bars in figure19, is of the same order of magnitude as the FFTW results.

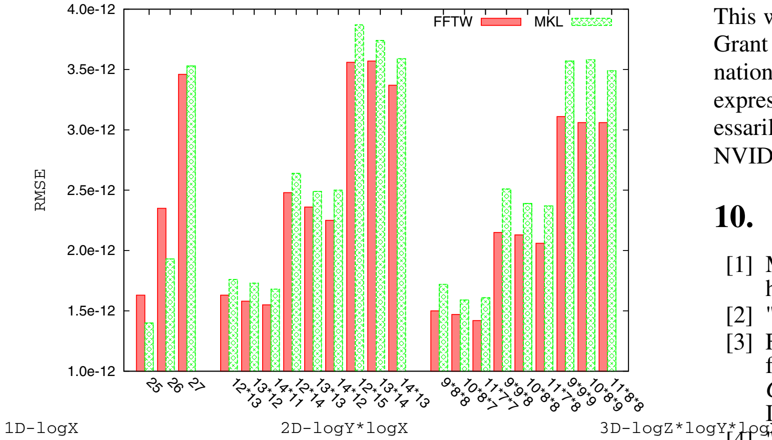

Figure 19: Precision of 1D, 2D and 3D FFTs on GTX480

8. CONCLUSION AND FUTURE WORKS

This is the first work that enables GPUs to efficiently compute FFTs of such large sizes on a single compute node. So far, CUFFT and many other CUDA based FFT libraries have mostly been focusing on FFT's on-card performance. There are at least two limitations of the FFT performance they have achieved. First, their FFT size is limited by the GPU on-card memory. Second, a user has to transfer the FFT's input and output data between CPU and GPU memory via PCI bus. When the data transfer over PCI bus is counted in, this transfer time can easily eliminate the majority of the GPU's on-card performance advantage. This work is the first step in attacking the above two realistic problems in the case of FFT and has achieved success in the comparison with CPU based FFT libraries.

In this paper, we propose a computation framework for GPUs to efficiently compute large 1D, 2D and 3D FFTs that do not fit into GPU memory. Unlike on-card FFT problems, whose performance depends on the speed of the GPU memory and the number of ALUs, the dominant performance factor for such large FFTs is the PCI bus bandwidth and the balance of computation. Specifically, we propose a couple of FFT related decomposition algorithms and a general blocked buffer algorithm to maximize the effective PCI bus bandwidth and best hide the computation time. The FFT computation kernels on GPUs are generated following an on-card FFT computation work proposed in paper [8]. For a high-end GPU, NVIDIA's GTX480, our algorithms achieve a speedup of 1.62× for 1D, 1.65× for 2D and 1.28× for 3D over FFTW with multithreading enabled on an Intel i7 CPU. Even higher speedup is achieved on a Tesla C2070, i.e. 1.5×, 2.3× and 1.53×, over FFTW.

This work can be easily extended to higher dimensional FFTs, FFTs with non power-of-two sizes and multi-GPU architectures. The idea behind our decomposition algorithm is to maximize the data locality of arrays to be transferred over the PCI bus, through both algorithmic manipulation and implementation optimization, so that a better bandwidth can be achieved. This FFT decomposition framework and the algorithms proposed in this paper may also be applied to FFTs in other hierarchical memory or communication bound problems other than FFT, where high performance can also be achieved by maximizing the memory or network bandwidth.

9. ACKNOWLEDGMENTS

We thank the anonymous reviewers for their helpful suggestions. This work is supported by the National Science Foundation under Grant No. 0746034 and 0904534 and NVIDIA's equipment donation. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation or NVIDIA.

10. REFERENCES

[1] M. Frigo and SG Johnson. "The FFTW web page" http://www.fftw.org, 2010.

[2] "The SPIRAL Project" http://www.spiral.net, 2010.

[3] F. Franchetti and Y. etc. Voronenko. FFT program generation for shared memory: SMP and multicore. In *SC 2006 Conference, Proceedings of the ACM/IEEE*, pages 51–51. IEEE, 2006.

[4] "Intel Math Kernel Library" http://software.intel.com/en-us/articles/intel-mkl/, 2010.

[5] "NVIDIA CUFFT Library" http://developer.nvidia.com/object/cuda_3_2_downloads.html, 2010.

[6] N.K. Govindaraju and B. etc. Lloyd. High performance discrete Fourier transforms on graphics processors. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–12. IEEE Press, 2008.

[7] 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.

[8] L. Gu, X. Li, and J. Siegel. An empirically tuned 2D and 3D FFT library on CUDA GPU. In *Proceedings of the 24th ACM International Conference on Supercomputing*, pages 305–314. ACM, 2010.

[9] D.H. Bailey. FFTs in external of hierarchical memory. In *Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, pages 234–242. ACM, 1989.

[10] A. Gupta and V. Kumar. The scalability of FFT on parallel computers. *IEEE Transactions on Parallel and Distributed Systems*, pages 922–932, 1993.

[11] D. Takahashi and Y. Kanada. High-performance radix-2, 3 and 5 parallel 1-D complex FFT algorithms for distributed-memory parallel computers. *The Journal of Supercomputing*, 15(2):207–228, 2000.

[12] Y. Chen and X. etc. Cui. Large-scale FFT on GPU clusters. In *Proceedings of the 24th ACM International Conference on Supercomputing*, pages 315–324. ACM, 2010.

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

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

[15] M. Frigo and SG Johnson. The Fastest Fourier Transform in the West. 1997.

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

[17] Whitepaper NVIDIA's next generation cuda compute architecture: Fermi, 2009.