A HYBRID GPU/CPU FFT LIBRARY FOR LARGE FFT PROBLEMS

by

Shuo Chen

A thesis submitted to the Faculty of the University of Delaware in partial fulfillment of the requirements for the degree of Master of Science in Electrical and Computer Engineering

Spring 2013

A HYBRID GPU/CPU FFT LIBRARY FOR LARGE FFT PROBLEMS

by

Shuo Chen

Approved: _____

Xiaoming Li, Ph.D.
Professor in charge of thesis on behalf of the Advisory Committee

Approved: _____

Kenneth E. Barner, Ph.D.
Chair of the Department of Electrical and Computer Engineering

Approved: _____

Babatunde A. Ogunnaike, Ph.D.
Interim Dean of the College of Engineering

Approved: _____

James G. Richards, Ph.D.
Vice Provost for Graduate and Professional Education

ACKNOWLEDGMENTS

First of all, I wish to thank my adviser, Dr. Xiaoming Li, for his guidance, encouragement and great help. Without his strong spport, this work could not be possible. I would also like to thank my lab colleagues, Sha Li and Yuanfang Chen, who helped me a lot during the writing of this thesis. Last but not least, I would show my gratitude to my parents for all their support and care on me.

TABLE OF CONTENTS

LIST OF TABLES . . . . . **vi**
LIST OF FIGURES . . . . . **vii**
ABSTRACT . . . . . **viii**

Chapter

1 INTRODUCTION **1**

1.1 Background of Fast Fourier Transforms on GPGPUs 1
1.2 Challenges of Hybrid FFTs on Heterogeneous GPU/CPU Systems . . 2
1.3 Solutions Description 3

2 OVERVIEW OF FFT ALGORITHM **5**

2.1 Overview of General FFT Algorithm 5

2.1.1 Cooley Tukey Algorithm 5
2.1.2 Other FFT Algorithms 5
2.1.3 I/O Tensor Representation 6

2.2 Overview of Our Hybrid FFT Algorithm 6

3 OUR HYBRID GPU/CPU FFT LIBRARY **10**

3.1 Hybrid 2D FFT Framework 10

3.1.1 Load Distribution 12
3.1.2 Optimizations on GPU 12
3.1.3 Asynchronous Strided Data Transfer 13

3.1.3.1 Data Transfer Scheme 13
3.1.3.2 PCI Bandwidth Evaluation 14

3.1.3.3 Comparison to CUFFT 14

3.1.4 Optimizations on CPUs 15
3.1.5 Cooperations of CPUs and GPU 16
3.1.6 Comparison to other heterogeneous FFT implementation . . . 16

3.2 Hybrid 3D FFT Framework 18

3.2.1 Load Distribution 20
3.2.2 Optimizations for GPU and Data Transfer 20
3.2.3 Cooperations of CPUs and GPU 21

4 LOAD BALANCING BETWEEN GPU AND CPU 23

4.1 Load Balancing of 2D FFT 24
4.2 Load Balancing of 3D FFT 25

5 PERFORMANCE EVALUATION 27

5.1 Performance Tuning with Load Distribution Ratios 28
5.2 Evaluation for 2D Hybrid FFT 30
5.3 Evaluation for 3D Hybrid FFT 32
5.4 Accuracy of Our Hybrid FFT 35

6 CONCLUSION AND FUTURE WORK 39

BIBLIOGRAPHY 40

LIST OF TABLES

4.1 Parameters for 2D FFT Running Time Estimation. 24

4.2 Parameters for 3D FFT Running Time Estimation. 26

5.1 Configurations of GPU, CPU, FFTW and MKL. 27

5.2 Valid Model Parameters in Seconds for FFTs of Size $2^{15} \times 2^{13}$ and $2^{10} \times 2^9 \times 2^9$. 30

LIST OF FIGURES

3.1 Overview of Hybrid Large Out-of-card 2D FFT. 11

3.2 PCI Bandwidth of Different Data Transfer Schemes. 15

3.3 Double Precision 2D FFT Performance Tuning Comparison of Our Hybird Approach against the Hybird CUFFT/FFTW Library. . . . 17

3.4 Overview of Hybrid Large Out-of-card 3D FFT. 19

5.1 Double-precision 2D FFT Performance Tuning. 31

5.2 Double-precision 3D FFT Performance Tuning. 31

5.3 Single-precision 2D FFT of Size from 2^{26} to 2^{29} on GTX480. 33

5.4 Double-precision 2D FFT of Size from 2^{25} to 2^{28} on GTX480. . . . 33

5.5 Single-precision 2D FFT of Size from 2^{29} to 2^{30} on Tesla. 34

5.6 Double-precision 2D FFT of Size from 2^{28} to 2^{29} on Tesla. 34

5.7 Single-precision 3D FFT of Size from 2^{26} to 2^{29} on GTX480. 35

5.8 Double-precision 3D FFT of Size from 2^{25} to 2^{28} on GTX480. . . . 36

5.9 Single-precision 3D FFT of Size from 2^{29} to 2^{30} on Tesla. 36

5.10 Double-precision 3D FFT of Size from 2^{28} to 2^{29} on Tesla. 37

5.11 Accuracy of Single-precision Hybrid 2D/3D FFT. 38

5.12 Accuracy of Double-precision Hybrid 2D/3D FFT. 38

ABSTRACT

Graphic Processing Units (GPU) has been proved to be a promising platform to accelerate large size Fast Fourier Transform (FFT) computation. However, current GPU-based FFT implementation only uses GPU to compute, but employs CPU as a mere memory-transfer controller. The computation power in today's high-performance CPU is wasted. In this project, a hybrid optimization framework is proposed to use both CPU and GPU in heterogeneous CPU-GPU systems to compute large scale 2D and 3D FFTs that exceed GPU memory. This work introduces a flexible partitioning scheme that makes it possible to decompose FFT for two computing devices with hugely different performance characteristics. The partitioning scheme enables concurrent execution of FFT sub-problems on CPU and GPU. Additionally, our approach integrates several FFT decomposition paradigms to tailor the extraction of computation and communication patterns for CPU and GPU, and in the process exploits more hidden parallelism than other heterogeneous methods. In addition, our work automatically adapts to different hardware configurations by tuning for architecture features and the work distribution between GPU and CPU. Several empirical profiling techniques are proposed to characterize the communication and computation of FFT problems on GPU and CPU, and we develop effective heuristics to guide the entire empirical tuning process. Our library also overlaps data transfers to achieve higher bandwidth over PCI bus and equally importantly maintains data and layout consistency between CPU and GPU. We evaluate our hybrid FFT library from three aspects, i.e., optimal load distribution ratios, running time, and precision of result. In particular, the library is compared with CPU based libraries FFTW and Intel MKL, as well as a GPU based library on three GPUs, i.e., NVIDIA GeForce GTX480, Tesla C2070 and Tesla C2075. On average, our large FFT library is 121% and 145% faster than

the 4-thread SSE-enabled FFTW and the 4-thread SSE-enabled Intel MKL, with max speedups 4.61 and 2.81, respectively.

Chapter 1

INTRODUCTION

1.1 Background of Fast Fourier Transforms on GPGPUs

Fast Fourier Transform (FFT) is one of the most widely used numerical algorithms in science and engineering domains. It is not rare that large scientific and engineering computation such as fluid dynamics simulations spend majority of execution time on large size FFTs. Recently the Graphical Processing Units (GPUs) have been proved to be a promising platform to solve large FFT problems. At first, efforts have been focused on solving FFT problems whose sizes can fit into the device memory of GPU. It means that only two simple data transfers are needed in the solving of one FFT problem, one copying all the source data from CPU memory to GPU memory using the PCI bus, and the other copying all the results back. Since the data transfer does not have much to optimize, the prior works focus on the decomposition of FFT problems for the two-level organization of processing cores on GPU and the efficient usage of GPU on-device memory hierarchy. Libraries such as CUFFT from NVIDIA [1], Nukada's work on 3D FFT [2, 3], and Govindaraju's [4] and Gu's work on 2D and 3D FFT [5] can be classified into this group. Recently, Gu et.al. [6] demonstrated a GPU-based FFT library that can solve FFT problems larger than the GPU device memory. Since one data transfer cannot move all data between CPU and GPU, multiple data transfers are needed. Gu et.al. proposed a joint optimization paradigm that co-optimizes the communication and the computation phases of FFT, and an empirical searching method to find the best tradeoff between the two factors. For even larger FFT problems, Chen et.al. presented a GPU cluster based FFT implementation [7]. However, since computation contributes only a trivial part to the overall

execution time, the work has been almost exclusively focused on the optimization of communication over inter-node channels.

The prior FFT work on GPU, no matter the on-card FFT libraries, the out-of-card FFT libraries or the GPU-cluster based solutions, all involve CPU in the loop. However, CPU is only used as a memory or communication controller, that is, executing the memory transfer requests between CPU memory and GPU memory, or between nodes. The computing power of CPU is wasted. Ogata et.al. [8] attempted to divide the computation to both CPU and GPU, though targeting at problems whose sizes can fit into the GPU memory. The small problem assumption makes the optimization of data communication between CPU and GPU trivial because all data can be copied to GPU in one data copying, which largely avoids the challenges of co-optimizing both computation and communication between two different types of devices. In this work, we present a hybrid FFT library that engages both CPU and GPU in the solving of large FFT problems that can not fit into the GPU memory.

1.2 Challenges of Hybrid FFTs on Heterogeneous GPU/CPU Systems

Making FFT run concurrently on CPU and GPU come with significantly challenges. First of all, CPU and GPU are two computer devices with totally different performance characteristics. Even though FFT can be decomposed in many different ways, not a single method can arbitrarily divide a problem into subtasks with two different performance patterns. The fundamental difficulty lies in the mathematical properties of FFT. In FFT, a simple change to the division of computation will lead to global effects on the data transfers, because ultimately any single point in the output of a FFT problem is mathematically dependent on *all* input points. In other words, we cannot just optimize for CPU or just optimize for GPU. While a part of computation might be very efficient on CPU, the need to move data from previous GPU computation parts, or the need to send data to the following GPU computation parts, might be expensive and could totally diminish all advantages gained from the optimization of the current step. In simpler words, the first problem we need to solve is to divide a

FFT workload between two types of computing devices that are connected by a slow communication channel.

The second challenge is the magnitude of the vast space of possible hybrid implementations for one FFT problem. In addition to the large number of possible algorithmic transformations, as outlined in the first challenge, CPU and GPU architectural features also need to be considered in the search. Reconciling CPU and GPU architectures is hard because they simply like different styles of computation/communication mix. For example, FFT might be decomposed into same computation sub-tasks but with different memory transfer needs. CPU and GPU will respond very differently to the subtle difference in memory transfer requests. Parameters such as the stride of memory accesses, or the coalescing of memory requests determine whether CPU or GPU is more appropriate for a sub-step of FFT. Moreover, the decision of workload assignment needs to be put into a search space that consists of many different ways of decomposition and different ways of data transfer. In particular, computation and communication can be efficiently overlapped, an important performance booster, only if the data dependency between the CPU parts and the GPU parts is appropriately arranged. In other words, even if we already find the best algorithm for a FFT problem, i.e., the best division of computation, the implementation of the algorithm still needs to be co-tuned for two different architectures.

1.3 Solutions Description

This project for the first time proposes a hybrid implementation of FFT that can take advantage of both CPU and GPU in a heterogeneous computer node in solving large FFT problems. The work makes three main contributions: (1) a hybrid large-scale FFT decomposition framework that combines the Radix algorithm and the Cooly-Turkey algorithm to enable the extraction and the tailoring of different workload and data transfer patterns appropriate for the two different computing devices, (2) an empirical performance modeling for FFT sub-steps on CPU and GPU, which estimates performance based on several key parameters, and replaces an exhaustive walk-through

of the vast space of possible hybrid implementations of FFT on CPU/GPU with a guided empirical search, and (3) key heuristics to purposefully expose opportunities of overlapping communication with computation in the process of decomposing FFT, and heuristics to find the best tradeoff among computation time, data transfer cost, and their overlapping.

Chapter 2

OVERVIEW OF FFT ALGORITHM

2.1 Overview of General FFT Algorithm

FFT algorithms recursively decompose a N-point DFT into several smaller DFTs [9], and the divide-and-conquer approach reduces the operational complexity of a Discrete Fourier Transform (DFT) from $O(N^2)$ into $O(NlogN)$. There are many FFT algorithms, or in other words, different ways to decompose DFT problems.

2.1.1 Cooley Tukey Algorithm

Our hybrid FFT library is based on the general Cooley-Tukey factorization FFT algorithm [10]. In this section we briefly introduce the FFT algorithms and overview how they are incorporated into our hybrid approach. The DFT transform of an input series $x(n), n = 0, 1, ..., N-1$ of size N is presented as $Y(k) = \sum_{n=0}^{N-1} x(n)W_N^{nk}$. We can map the one dimensional input into two dimensions indexed by l in L dimension and m in M dimension, respectively. The Cooley-Tukey FFT decomposes the original DFT into three sub-steps: (1) Perform M DFTs of size L, $A(p,m) = \sum_{l=0}^{L-1} x(l,m)W_L^{lp}$; (2) Multiply twiddle factors, $B(p,m) = A(p,m)W_N^{pm}$; and (3) Perform L DFTs of size M, $Y(p,q) = \sum_{m=0}^{M-1} B(p,m)W_M^{mq}$. Therefore, $Y(k) = Y(pM+q)$.

In essence, Cooley-Tukey introduces a decomposition approach that divides one dimensional computation into two. Moreover, the Radix algorithm is a special case of the Cooley-Tukey algorithm for power-of-two FFT problems.

2.1.2 Other FFT Algorithms

There are many other FFT algorithms [11] distinct from CooleyTukey method. Prime-Factor(Good-Thomas) [12] decomposes a DFT of size $N = N1 \times N2$, where $N1$

and $N2$ are co-prime numbers. Twiddle factor calculation is not included in this algorithm. In addition, Rader's algorithm [13] and Bluestein's algorithm [14] can factorize a prime-size DFT as convolution. Actually, each of the algorithms illustrated above has specific implementation.

2.1.3 I/O Tensor Representation

In this project, we extend the I/O tensor representation introduced in FFTW [15] to represent the Cooley-Tukey algorithmic transformation of hybrid FFTs. An I/O tensor $d(C, Si, So, I, O)$ denotes FFTs along a data dimension where C is the size of one dimensional FFT, Si and So represent the stride of input and output, and I and O are the addresses of input and output array. t_M^L represents multiplication of twiddle factors with size $L \times M$. The I/O tensor representation captures the two most important factors that determine FFT's performance, i.e., data access patterns and computation load. As an example, the Cooley-Tukey FFT decomposition can be precisely denoted as an extended I/O tensor representation $u = \{d(L, M, M, I, O), t_M^L d(M, 1, 1, O, O)\}$. Here u is an I/O tensor that represents a multidimensional FFT.

2.2 Overview of Our Hybrid FFT Algorithm

A notable contribution in our work is the achievement of an adaptive library for 2D FFT that automatically achieves optimal performance using available heterogeneous GPUs-CPUs resources. Traditional FFT libraries have been built either on general purpose CPUs such as FFTW, SPIRAL and Intels MKL or on compute-intensive GPUs such as CUFFT, however, there is few that takes advantage of CPU to concurrently compute partial well-decomposed FFT with GPU. Although CUFFT makes use of both CPU and GPU, it only enables CPU to perform controlling GPU kernel executions and also data transfer between host and GPU. Comparing with current FFT libraries, we fully employ both CPU and GPU computational resources for heterogeneous HPC.

In this project, a hybrid optimization framework is proposed to use both CPU and GPU in heterogeneous CPU-GPU systems to compute large scale 2D and 3D FFTs that exceed GPU memory. This computational model generalizes a partitioning scheme that efficiently distributes work to two different computing devices to make them execute FFT computation load concurrently. Generally, our hybrid FFT approach is guided by Radix algorithm and Cooley-Tukey decomposition algorithm. The I/O tensor representation is modified to specify the partial work in CPU and GPU and the synchronization process is identified in implementation to maintain the accuracy of results.

In addition to the work load partitions, the optimizations in heterogeneous system are intensively discussed. In GPU side, two levels of FFT decomposition paradigms are studied. In general, the decomposition of large out-of-card FFT problem is performed at first to generate subproblems that can fit into a GPU. We propose an scheme for efficient data transfers between CPU and GPU through a PCI express bus. Particularly, since the data portion required to be contiguous in GPU memory has no contiguous locality in host memory, a technique is proposed to handle the strided memory copy between GPU and CPU to keep high PCI bandwidth. To further optimize performance, we discuss the factors that restrict high PCI bandwidth based on well-designed tests and propose a method can overlap different partial data transfers to achieve optimal performance.

Furthermore, the deeper level decomposition is applied into optimization for kernel computation of the on-card FFTs. The on-GPU computation is based on the specially revised codelets which are compiler generated C programs to solve small FFTs in FFTW. Codelets are able to realize the strided data accesses to save the execution time of matrix transpose. Moreover, codelets that are used to compute multiple dimensional FFTs are grouped into the fewest number of kernels, and each kernel will have minimum number of accesses to global memory with multiple accesses to shared memory (48KB in GTX480) to hide memory latency dramatically. 32 threads are coalesced into a single memory access so that a higher global memory bandwidth is

achieved. In addition, we also make use of asynchronous streams to manage concurrent kernel executions and the overlapping between kernels and data transfers.

As for optimizations in CPU side, FFTWs advanced interface is used to transform a group of complex arrays at a time. Operations on non-contiguous data is well performed so that much execution time of transposition is saved. Improved performance is obtained if we parallelize the grouped FFT computations using multiple concurrent threads in CPU side. An additional thread is made use of to control GPU computation so that load balancing between CPU and GPU can be gained.

To achieve overall good performance, we leave specific part of computation in our FFT library into GPU for computation. If we expect to incorporate CPU to work for it, more data transfers between CPU and GPU have to be processed for data merge through a slow PCI bus even if GPU load can be somehow released. We test the performance of such all-hybrid computation which incorporates CPU and GPU to handle all the work. It is better than that of pure GPU computation, but not as optimal as our partially hybrid implementation since we need to tradeoff co-optimizing computation with communication.

Finally, in order to attain the best parallel performance, the best balance between load overhead needs to be found. We combine both performance modeling and empirical searching at build time to determine the optimal CPU-GPU load distributions for different problem sizes and different heterogenities. Our performance modeling is to split the total execution in either GPU or CPU into several sub-steps, analyze the heterogeneous execution flow, and derive a performance model for each sub-step. Therefore, the modeled performance tuning under varying load ratio can be attained and the best ratio for each input size is able to be found by empirical searching. The optimal ratio obtained from our performance model might not be very accurate, but it provided us a small and very precise region where the actual optimal ratio indeed resides in. It enables to guide us to efficiently determine the actual optimal ratio in our performance evaluation instead of a walk-through of the vast space of all possible

cases. Although the performance modeling and tuning is done at build time, the overhead is negligible as it only takes in the order of microseconds to evaluate our models. The implementation result shows that work balance is obtained under the optimal load ratio between CPU and GPU.

Chapter 3

OUR HYBRID GPU/CPU FFT LIBRARY

3.1 Hybrid 2D FFT Framework

Our heterogeneous 2D FFT framework solves FFT problems that are larger than GPU memory. Suppose that the problem size is $N = Y \times X$, where Y is the number of rows and X is number of columns. Generally 2D FFT involves two rounds of computation, i.e. Y dimensional 1D FFT for all columns along X dimension and then X dimensional 1D FFT for all the rows along Y dimension. A 2D FFT for an 2D input $f(y, x)$ of size N is defined in equation 3.1,

$$\begin{aligned} out(k_y, k_x) &= \sum_{x=0}^{X-1} W_X^{xk_x} \sum_{y=0}^{Y-1} W_Y^{yk_y} f(y, x) \\ &= \sum_{x=0}^{X-1} W_X^{xk_x} \sum_{y=0}^{Y-1} \{W_Y^{yk_y} f(y, x_{gpu}) + W_Y^{yk_y} f(y, x_{cpu})\} \end{aligned} \tag{3.1}$$

where $x, k_x = 0, 1, ..., X_{gpu}, ..., X-1$; $y, k_y = 0, 1, ..., Y-1$; $x_{gpu} = 0, 1, ..., X_{gpu}-1$; $x_{cpu} = X_{gpu}, ..., X-1$; twiddle factor $W_c^{ab} = e^{-j2\pi ab/c}$. From equation 3.1, the work load of 2D FFT in the first round can be distributed into GPU and CPU, respectively. The work ratio of GPU to CPU in round one is denoted as $R_X = \frac{Xgpu}{Xcpu}$ where $Xgpu$ and $Xcpu$ are the X dimensional sizes for the GPU and CPU parts in the first round. The total 2D FFT can be represented as $u_{2d} = \{d(Y, X, X, I, O), d(X, 1, 1, O, O)\}$ in an extended I/O tensor format. When the 1D FFTs in each round of 2D problem are large, we further apply Y dimensional Cooley-Tukey decomposition into the large 2D FFT to reduce computational complexity and exploit more parallelism as well. In our

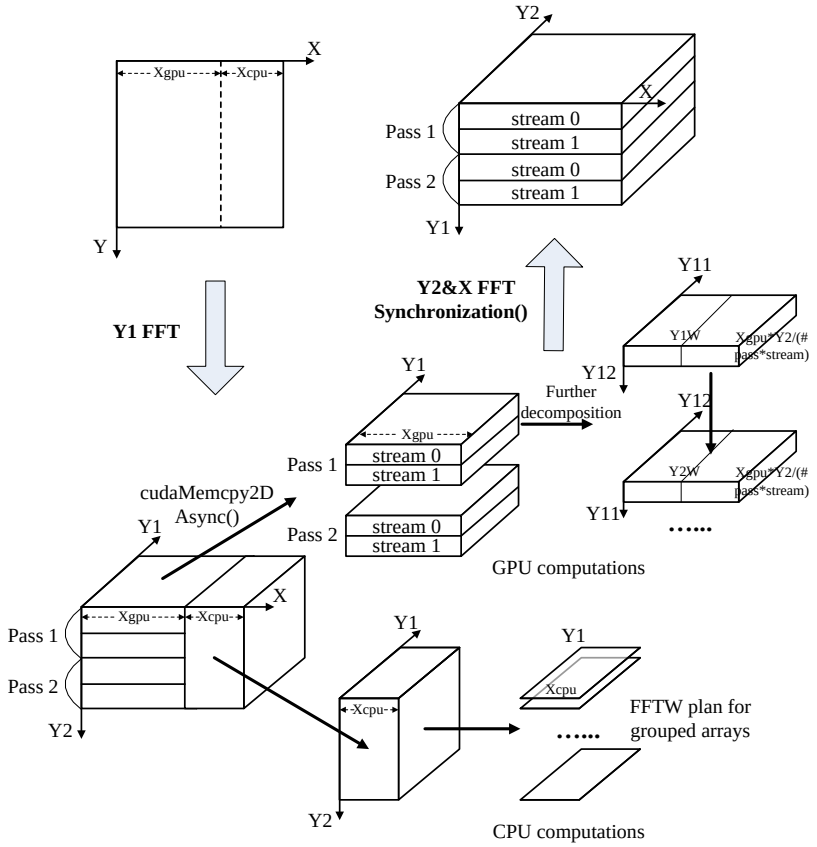

Figure 3.1: Overview of Hybrid Large Out-of-card 2D FFT.

hybrid 2D FFT framework, the tensor representation of work distribution on GPU, i.e. u_{gpu}, and on CPU, i.e. u_{cpu}, are transformed as equation 3.2.

$$\begin{aligned} u_{gpu} =& \{d(Y_1, Y_2 X gpu, X gpu, I gpu, O gpu), Sync, \\ & t_{Y_2}^{Y_1} d(Y_2, Y_1 X, Y_1 X, O, O), d(X, 1, 1, O, O)\} \\ u_{cpu} =& \{d(Y_1, Y_2 X cpu, X cpu, I cpu, O cpu), Sync\} \end{aligned} \tag{3.2}$$

where $Y = Y_1 \times Y_2$, $Sync$ denotes data transfer and synchronization between CPU and GPU within computation. As a result, three dimensional computations, i.e. Y_1, Y_2, X in order, need to be executed. Also note that a twiddle factor computation $t_{Y_2}^{Y_1}$ is introduced by the Cooley-Tukey decomposition between Y_1 and Y_2 step. Figure 3.1 shows the high-level working flow of our hybrid 2D FFT framework.

3.1.1 Load Distribution

For Y_1 dimensional computation, work load is distributed between GPU and CPU. Work ratio of GPU to CPU is again denoted as $R_X = \frac{Xgpu}{Xcpu}$, but can take values different from that of the first round. On GPU side, a portion of Y_1 dimensional FFTs need to be firstly computed. The size of 2D FFT problem on GPU may exceed that of GPU global memory. In this case, we divide the 2D FFT of GPU part into several passes such that the sub-problem of each pass can fit into GPU memory and be executed with the CPU portions concurrently. The number of passes equals to $\frac{\text{Xgpu*Y*\# of bytes per element}}{\text{GPU memory in bytes}}$. Each pass of GPU computation takes advantage of multiple streams to overlap computation and communication. The optimal number of streams can be determined from our empirical search.

3.1.2 Optimizations on GPU

On GPU, all the Y_1 dimensional 1D FFTs are calculated by codelets, i.e., highly-efficient straightline code segments that solve small FFT problems. The codelet provides automatic matrix transposing within FFT substeps such that much transposition time can be saved. The concept of codelet was first introduced in FFTW, though its codelet generator only generates CPU code. In our work, we extends FFTW codelet generator to generate GPU-based codelets. In addition, if size Y_1 is still large, we would further decompose $Y_1 = Y_{11} \times Y_{12}$ sized 1D FFT into two dimensional FFTs with smaller sizes Y_{11} and Y_{12}, respectively. Since device memory is of much higher latency and lower bandwidth than on-chip memory, shared memory on GPU is utilized for the decomposed FFTs to increase device memory bandwidth dramatically. For example, NVIDIA GTX480 GPU has 48KB shared memory which can store 6K complex single-precision data or 3K complex double-precision data in maximum. $Y_1W \times Y_{11} \times Y_{12}$ sized shared memory needs to be allocated, where Y_1W is chosen to 16 for half-warp of threads in GTX480 to enable coalesced access to device memory. The number of threads in each block, for both Y_{11} and Y_{12}-step sub-FFTs, is therefore $Y_1W \times max(Y_{11}, Y_{12})$. To calculate each Y_1-step 1D FFT, a size Y_{11} codelet is first executed to load data from

global memory into shared memory for each block. Next, all threads in a block are synchronized to finish its work before data in shared memory is reused by the Y_{12}-step codelet and subsequently written back to global memory. Experiment tests show that such shared memory technique effectively hides the latency of global memory and increases data reuse, both contributing to performance on GPU.

3.1.3 Asynchronous Strided Data Transfer

An efficient data transfer scheme has been exclusively studied in this section. A technique is proposed to process strided memory copy between GPU and CPU and to maintain high PCI bandwidth. Furthermore, we discuss the restrictions to PCI bandwidth based on well-designed tests and propose a method that overlaps different partial data transfers with kernel executions for performance improvement.

3.1.3.1 Data Transfer Scheme

Another performance hurdle is strided memory transfers between CPU and GPU. Since we separate the load between CPU and GPU, the portion of input data required to be continuous in GPU memory is not contiguous in host memory. For each pass of our 2D FFT, if we use simple CUDA memory copy operations to transfer the total $\frac{Xgpu \times Y_1 \times Y_2}{\#\text{ of passes} \times \#\text{ of streams}}$ sized data into GPU, we need to utilize PCI bus $Y_1 \times \frac{Y_2}{\#\text{ of passes} \times \#\text{ of streams}}$ times because we transfer $Xgpu$ sized data each time. Clearly, the high PCI transfer overhead will kill all potential performance gain. CudaMemcpy2DAsync() can make only one call to transfer a strided 2D memory area of size $\frac{Xgpu \times Y_2}{\#\text{ of passes} \times \#\text{ of streams}}$ into GPU at a time. Therefore, the total number of PCI transfers is reduced to only Y_1. Moreover, such data transfer optimization supports for CUDA asynchronous concurrent execution. Different data transfers managed by different streams can be executed concurrently and can be overlapped with different streamed GPU kernel executions.

To best use cudaMemcpy2DAsync() in hybrid FFTs, when copying the GPU output back to CPU, it is used to copy multiple 2D strided arrays. Each streamed

PCI transfer at this time could copy $\frac{Xgpu \times Y_2 \times Y_1}{\#\text{ of passes} \times \#\text{ of streams}}$ sized data. After the Y_1-step FFTs, all the streams on GPU side are synchronized, and a subsequent barrier is set to synchronize GPU with CPU.

3.1.3.2 PCI Bandwidth Evaluation

Particularly, our asynchronous strided transfer scheme achieves more efficient bandwidth than that of PCI transmission approach proposed in Gu's out-of-card FFT work [6] since we do not need to waste additional CPU resource to prepare buffer and therefore we get rid of overhead caused by buffering method in Gu's work.

To demonstrate the improvement of our PCI bandwidth, we used the same subarray test as Gu's work [6], where there are C *regular subarrays* of length W each. There is a stride $X - W$ between every two regular subarrays in a *large array* of size $C \times X$. The regular subarrays of a fixed size $C \times W = 32M$ need to be transfer into GPU memory from system memory through PCI bus. Note that the large array is contiguous in system memory but regular subarrays are not contiguous. Naive CUDA memory-copy operation is required to occupy PCI bus C times to transfer only one regular subarray at a time, however, our 2D strided transfer approach only needs to use PCI bus once without allocating redundant CPU resources for Gu's buffering. Figure 3.2 shows the improvement of our PCI bandwidth over Gu's work. Overall, PCI bandwidth of our 2D hybrid implementation can achieve 6.5 GB/s on average comparing to only 4.2 GB/s of Gu's work and 3.4 GB/s of naive PCI transfer.

3.1.3.3 Comparison to CUFFT

Theoretically, Tesla C2070 can sustain over 200 GFLOPS on a single precision FFTs that fits in GPU memory using CUFFT. However, this performance excludes the time spent on transferring input to GPU and transferring result back, which users need to do for every CUFFT call. If that time is included, also on Tesla C2070, CUFFT only delivers 41 GFLOPS. Our library is evaluated with all time included. Unlike N-body simulations or matrix multiplications, FFT is largely a memory-bound

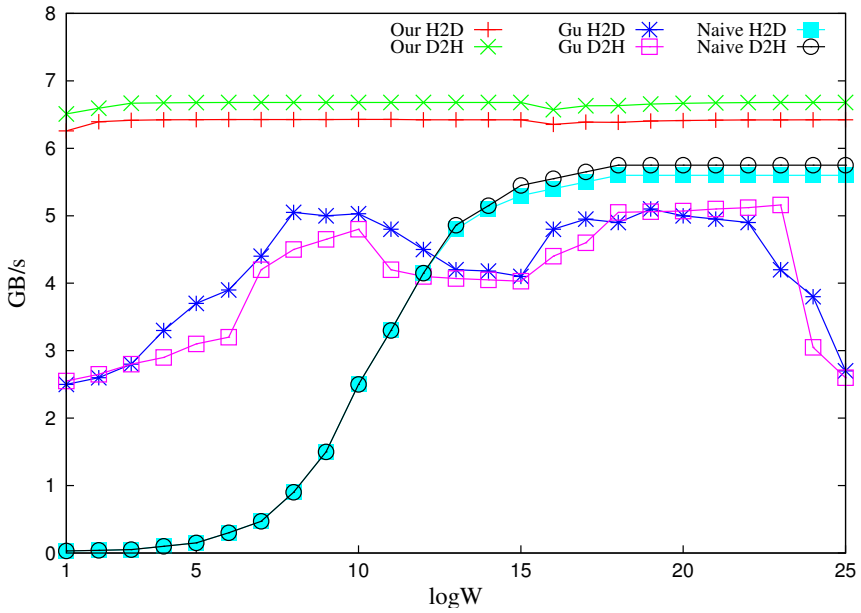

Figure 3.2: PCI Bandwidth of Different Data Transfer Schemes.

problem due to frequent data exchanges. Therefore achieving peak performance in heterogeneous CPU-GPU system is a challenge. As evaluated in section 5, our peak single-precision performance achieves 44 GFLOPS, and more importantly, problem sizes for our implementation are at least twice larger than the largest problem CUFFT can handle. The co-optimization of computation and communication for such problems is the key innovation of our work. Note that the complexity of FFT being O(NlogN), it is harder to solve larger FFTs efficiently.

3.1.4 Optimizations on CPUs

For Y_1 dimensional computation on CPU, Y_1 sized 1D FFTs are required to calculate for $Xcpu \times Y_2$ times. For each present Y_1 dimensional 1D FFT, data accesses have a stride of $Xcpu \times Y_2$. In addition, each 1D FFT needs to do a strided transpose. Both strided memory accesses and strided transpose are very expensive on CPU. Instead, we group the transformation of multiple complex arrays into a concurrent group operation and allow it to operate on non-contiguous (strided) data. Therefore, we need

no input or output transposition and save much execution time. We set the number of arrays—$Xcpu$—to be the maximum of what a FFTW group plan could execute at a time. For each grouped array, the plan computes size Y_1 1D FFT across a stride of $Y_2 \times X$ for input and X for output. We need to execute such kind of plan for totally Y_2 times.

3.1.5 Cooperations of CPUs and GPU

To coordinate GPU and CPU in hybrid FFT, we parallelize the workload in CPU side, which essentially is a loop of size Y_2, into 4 concurrent subsections. Independent grouped FFT computation steps are carried out in each parallel subsection. Workload of GPU including data transfers and kernel executions is parallelized with CPU computations. Afterwards, jobs on GPU driven by different streams are synchronized before the task synchronization between GPU and CPUs. There is no matrix transposition on either GPU or CPU since computations in either side is re-organized to naturally subsume the strided transposition.

The subsequent calculation of twiddle factor multiplication $t_{Y_2}^{Y_1}$ and Y_2 & X dimensional FFTs is left for GPU. For $Y_2 = Y_{21} \times Y_{22}$ dimensional FFTs, Cooley-Tukey decomposition is again applied since relative large size of Y_2 would hurt the performance of codelet based GPU computing. Similarly, shared memory is taken into account for reusing data between the decomposed Y_{21}-step FFTs and the subsequent Y_{22}-step FFTs. For the last X-step, i.e., 1D contiguous FFT sub-problems, CUFFT library is used because it provides good performance for row-major contiguous 1D FFTs. Instead of using ordinary CUFFT plan, we make use of stream-enabled CUFFT plan such that all Y_2 and X dimensional computations plus both PCI transfers of Y_2's input and X's output become stream-based asynchronous executions.

3.1.6 Comparison to other heterogeneous FFT implementation

In addition, we also compare our hybrid FFT library against a naive hybrid 2D FFT implementation comprising of assigning GPU workload to CUFFT and CPU

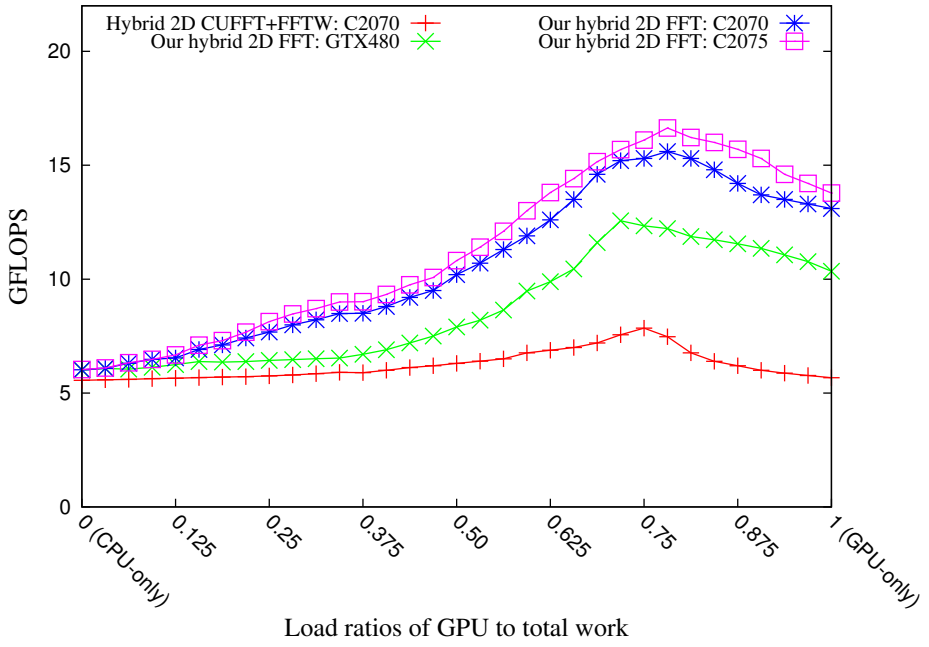

Figure 3.3: Double Precision 2D FFT Performance Tuning Comparison of Our Hybird Approach against the Hybird CUFFT/FFTW Library.

workload to FFTW. This heterogeneous method was first proposed in Ogata's work [8]. Computation is firstly distributed along X dimension in the 1^{st}-round of 2D FFT and along Y dimension for the 2^{nd}-round. In GPU side, sub-problems is further divided into several passes to facilitate data transfer between GPU and CPU. Matrix transpose, CUFFT and data transfer are processed in asynchronous manner. In CPU side, instead of ordinary FFTW along with transpose, FFTW advanced interface is utilized to handle strided data more efficiently. The purpose of this comparison is to see how our optimization technique improves over a naive hybrid CUFFT/FFTW solution. In the experiment, we vary the CPU/GPU work ratio from 0% to 100% for the naive solution and show its double precision performance curve of size $2^{15} \times 2^{13}$ on C2070 in figure 3.3. The best performance for the naive hybrid FFT is only 7.7 GFLOPS which is far below that of our hybrid version. The main reason is the lacking of co-optimization in the naive solution.

3.2 Hybrid 3D FFT Framework

General 3D FFT requires three rounds of computation. Each round computes 1D FFT along one dimension across the other two dimensions. Suppose the 3D input has sizes (Z, Y, X), the 3D FFT can be represented in tensor form as $u_{3d} = \{d(Z, XY, XY, I, O), d(Y, X, X, O, O), d(X, 1, 1, O, O)\}$.

To describe how our hybrid 3D FFT works, we start with a simple hypothetical scenario where all the work is assigned to GPU, and then continue to reveal how computation is extracted from this GPU-only hypothetical case and is assigned to CPU. Suppose that $Z = Z_1 \times Z_2$ and $Y = Y_1 \times Y_2$, the u_{3d} can be transformed as in formula 3.3, where $|_i$ and $|_o$ denotes respective input and output data transfers through PCI bus.

$$\begin{aligned}&\{|_i, d(Z_1, XYZ_2, XY), |_o|_i, t^{Z_1}_{Z_2} d(Z_2, XYZ_1, XYZ_1), |_o|_i,\\ &d(Y_1, Y_2X, X), |_o|_i, t^{Y_1}_{Y_2} d(Y_2, Y_1X, Y_1X), |_o|_i, d(X, 1, 1), |_o\}\end{aligned} \tag{3.3}$$

The problem with this initial formula is that the workload in the formula cannot be well balanced between two computing devices. To balance computations between CPU and GPU, and to enable asynchronous communications, the computations sub-steps need to be reordered [6]. The reordered computations are summarized in equation 3.4.

$$\begin{aligned}&\{|_i, d(Z_1, XYZ_2, XY), d(Y_1, Y_2X, X), t^{Y_1}_{Y_2} d(Y_2, Y_1X,\\ &Y_1X), |_o|_i, t^{Z_1}_{Z_2} d(Z_2, XYZ_1, XYZ_1), d(X, 1, 1), |_o\}\end{aligned} \tag{3.4}$$

Next let's examine how the workload as represented in the equation 3.4 can be distributed to CPU and GPU. We start our discussion with a simple hypothetical scenario where all the work is assigned to GPU. In this case only two rounds of computation are required to execute total 3D FFT. The first round is to input data into GPU for several passes, to calculate Z_1, Y_1, Y_2 dimensional FFTs in order for each pass, and to output intermediate results of Y_2 into CPU. The second round is to transfer the temporary results into GPU, to calculate twiddle factor $t^{Z_1}_{Z_2}$ with Z_2 dimensional FFTs and execute X dimensional FFTs before final results are transferred back to CPU. Such

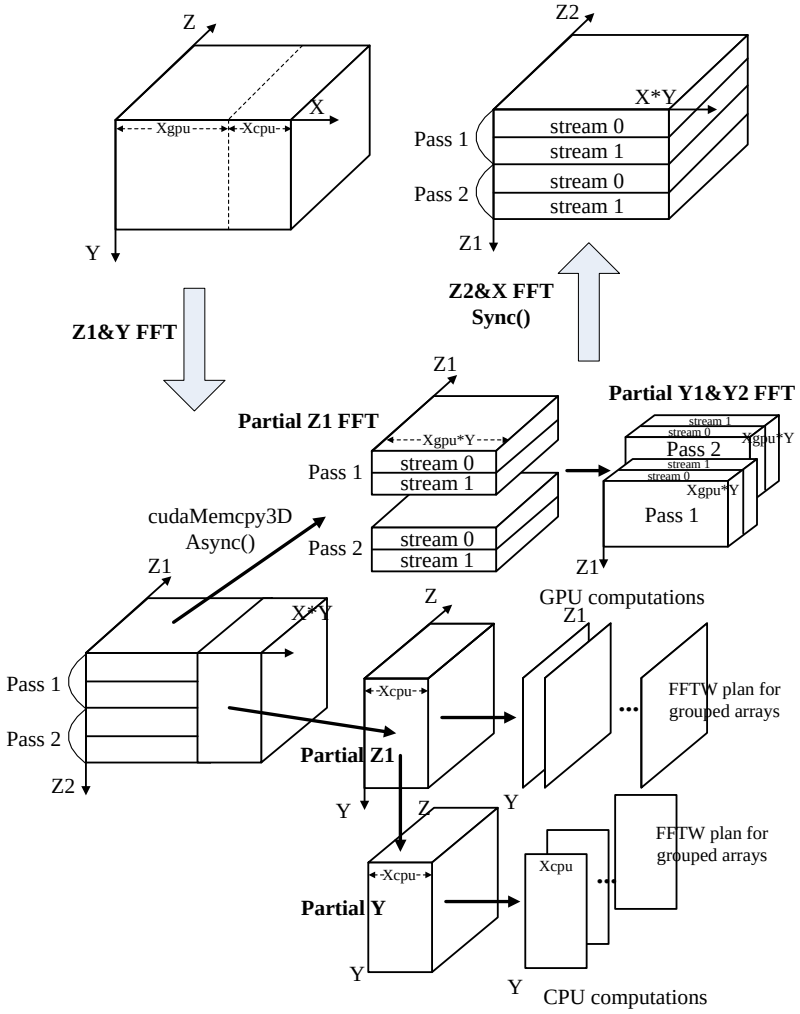

Figure 3.4: Overview of Hybrid Large Out-of-card 3D FFT.

GPU execution flow in formula 3.4 achieves great performance improvement because it saves the number of data transfers between CPU and GPU by $6 \times \#$ passes times in total, while in the original setup in formula 3.3, the total number of PCI transfers is $10 \times \#$ passes since each portion of that tensor u_{3d} requires to invoke PCI bus transfers $2 \times \#$ passes times for both input and output data between CPU and GPU. The hybrid 3D FFT framework is illustrated in figure 3.4.

3.2.1 Load Distribution

We have shown in the hybrid 2D FFT framework that if we want to achieve actual high performance from heterogeneous implementation, we need to get rid of frequent uses of PCI bus transfers. For the pure GPU implementation in formula 3.4, in addition to the initial input transfer and final output transfer through PCI bus, we only need two extra PCI transfers including copying output of Y_2 dimensional FFTs from GPU back to CPU, and copying the input of subsequent Z_2 dimensional FFTs from CPU into GPU. Therefore if we want to employ CPU for computing along with GPU, we need to arrange the data exchange between CPU and GPU to occur between Y_2 and Z_2 dimensional FFTs to reduce the total number of PCI bus transfers. Otherwise, more PCI transfers will be invoked to merge the partial results of CPU and GPU between two sub-steps of FFTs. Therefore, CPU is used to compute Z_1, Y_1 and Y_2 dimensional FFTs, and the subsequent calculation of Z_2 and X dimensional FFTs would be left for GPU to finish. In summary, the heterogeneous 3D FFT tensor on GPU u_{gpu} and the tensor on CPU u_{cpu} are represented as formula 3.5, where $Sync$ represents data transfer and synchronization between CPU and GPU within computation.

$$
\begin{aligned}
&u_{gpu} = \{|_i, d(Z_1, XgpuYZ_2, XgpuY),\\
&d(Y_1, Y_2Xgpu, Xgpu), t^{Y_1}_{Y_2} d(Y_2, Y_1Xgpu, Y_1Xgpu),\\
&Sync, t^{Z_1}_{Z_2} d(Z_2, XYZ_1, XYZ_1), d(X, 1, 1), |_o\}\\
&u_{cpu} =\\
&\{d(Z_1, XcpuYZ_2, XcpuY), d(Y, Xcpu, Xcpu), Sync\}
\end{aligned}
\tag{3.5}
$$

For the first round computation composed of Z_1, Y_1 and Y_2 dimensional 1D FFTs, the work load is distributed to GPU and CPU along X dimension. Work ratio of GPU to CPU is $R_X = \frac{Xgpu}{Xcpu}$.

3.2.2 Optimizations for GPU and Data Transfer

On GPU side, a portion of data in the total FFT problem needs to be transferred from CPU initially. Since the size of FFTs assigned to GPU is larger than that of

GPU global memory, the total Z_1 dimensional FFTs of GPU part are split into several passes such that subtask of each pass can fit into GPU memory. The number of passes equals to $\frac{\text{Xgpu*Y*Z*\# of bytes per element}}{\text{GPU memory in bytes}}$. Each pass of GPU computation still makes use of multiple streams to overlap computation and communication.

For better utilization of shared memory, $Z_1 = Z_{11} \times Z_{12}$ sized 1D FFTs are decomposed into two dimensional FFTs with smaller size Z_{11} and Z_{12}. Shared memory is allocated with size $Z_1W \times Z_{11} \times Z_{12}$, where Z_1W enables coalesced access to shared memory. The number of threads in each block is set to be $Z_1W \times max(Z_{11}, Z_{12})$ which matches the natural parallelism for both size Z_{11} and Z_{12} 1D FFTs within their respective shared memory. The computation for decomposed FFTs within shared memory is the same as the Y_1 decomposed FFTs in our 2D hybrid FFT framework, and therefore is not further discussed here.

Since the portion of input data that is required to be continuous in GPU memory is not contiguous in host memory, We use cudaMemcpy3DAsync() on GPU to transfer a strided 3D memory area of size $\frac{Xgpu \times Y \times Z_2}{\text{\# of passes} \times \text{\# of streams}}$ into GPU at a time. Therefore, the total number of PCI transfer is reduced to Z_1. Moreover, different data transfers managed by different streams can be executed concurrently and overlapped with different streamed GPU kernels. For the copying of output from GPU back to CPU, we still need the 3D strided memory copy. Each streamed PCI transfer at this time could copy $\frac{Xgpu \times Y \times Z_2 \times Z_1}{\text{\# of passes} \times \text{\# of streams}}$ sized data. After each stream finishes calculating Z_1 dimensional FFTs, it will continue to compute Y_1 and Y_2 dimensional FFTs without waiting for other streams. All Z_1, Y_1 and Y_2 dimensional computations plus both Z_1's input and Y_2's output PCI transfers are asynchronous executions, and the only synchronization needed is after the Y_2-step.

3.2.3 Cooperations of CPUs and GPU

Similar to the 2D hybrid FFT, we execute the size Z_1 FFTs in groups. For each grouped array, the plan computes size Z_1 1D FFTs across a stride of $X \times Y \times Z_2$ for input and $Xcpu * Y$ for output. The total number of executions of such plans is $Xcpu \times Z_2$.

For the following Y_1 and Y_2 dimensional FFTs, we only need to calculate size Y FFTs instead on CPU. The number of grouped array is $Xcpu$. The plan computes size Y 1D FFTs across a stride of $Xcpu$ for input and X for output. The total number of executions of such plans is Z. Moreover, all the grouped FFT tasks, in total $Xcpu \times Z_2$ plus Z, are distributed to 4 concurrent threads. The work of GPU is executed in a control thread concurrently with CPU computations. There is only an invocation to cudaThreadSynchronize() to synchronize all the streams on GPU. A subsequent barrier is set to synchronize the work of GPU with CPUs.

Chapter 4

LOAD BALANCING BETWEEN GPU AND CPU

The 2D and 3D hybrid FFT frameworks layout the basic schemes of workload distribution between CPU and GPU. However, there are parameters whose values need to be tuned for the optimal load balancing for different CPU/GPU combinations. In this work, we combine both performance modeling and empirical searching to finish the last mile towards the optimal load balancing. The empirical tuning is done at build time.

Our approach is to split the total execution in either GPU or CPU into several primitive sub-steps, analyze the heterogeneous execution flow, and derive a performance model for each primitives. The model parameters provide estimated execution time that is parameterized with the load ratio of GPU to total work. For each problem size, we calibrate the models with two profiling runs, one on CPU and GPU each, to determine the values of model parameters in different distribution ratios.Afterwards, using those parameters, we can automatically estimate, rather than really measuring, the total execution time of our implementation under varying ratios. We further use dynamic-programming to find the optimal implementation for different problems using the primitives as building blocks. However, the estimated performance might not be completely precise. Therefore, we don't purely rely on the aforementioned performance estimation but only use it to provide a small region of potentially good choices, for which we empirically measure their performance and choose the best one. Therefore, we avoid a walk-through of the vast space of all possible combinations of primitives.

Although the modeling is done at build time, the overhead is negligible as it only takes in the order of microseconds to evaluate our models. Experimental results of performance tuning and validation will be described in more detail in section 5.1.

4.1 Load Balancing of 2D FFT

Using the hybrid 2D FFT as an example, suppose that the total problem size is $Y_1 \times Y_2 \times X$. The load ratio of GPU to total work is set to R_g along X dimension, therefore the ratio of CPU to the total is $1-R_g$. The execution time of the whole process can be modeled as 8 parameters, which are summarized in table 4.1. We used two runs, one on GPU and CPU each, to determine $T_{2d\text{H2D-gpu}}$, $T_{Y_1\text{kernel-gpu}}$, and $T_{2d\text{D2H-gpu}}$ as execution time of corresponding table 4.1's parameters in GPU-only case, and to determine $T_{Y_1\text{fftw-cpu}}$ as execution time of $T_{Y_1\text{fftw}}(1-R_g)$ in CPU-only case. Therefore, each parameter value in table 4.1 can be modeled with different distribution ratios.

Table 4.1: Parameters for 2D FFT Running Time Estimation.

Parameters	Description
# passes	Total # of passes. Subproblem of each pass fits into GPU memory.
# streams	Total # of streams that enables asynchronous kernel executions and transfers.
# thds	# of threads of CPU.
$T_{2d\text{H2D}}(i, R_g)$	$= T_{2d\text{H2D-gpu}} \times R_g$. Time of copying a 2D strided array of size $\frac{R_g \times X \times Y_2}{\#\text{ passes} \times \#\text{ streams}}$ from host to device in stream i.
$T_{Y_1\text{kernel}}(i, R_g)$	$= T_{Y_1\text{kernel-gpu}} \times R_g$. Time of Y_1-step FFTs computation of concurrent kernel in stream i. Thread block size is $Y_1 W \times max(Y_{11}, Y_{12})$, grid size is $\frac{R_g \times X \times Y_2}{\#\text{ passes} \times \#\text{ streams}}$.
$T_{2d\text{D2H}}(i, R_g)$	$= T_{2d\text{D2H-gpu}} \times R_g$. Time of copying a 2D strided array of size $\frac{R_g \times X \times Y}{\#\text{ passes} \times \#\text{ streams}}$ from device to host in stream i.
$T_{Y_1\text{fftw}}(1-R_g)$	$= T_{Y_1\text{fftw-cpu}} \times (1-R_g)$. Time of Y_1-step FFTs on advanced FFTW plan for grouped array of size $(1-R_g) \times X$ in CPU. Total number of plans is Y_2.
$T_{Y_2\&X}$	Time of subsequent calculation of Y_2 and X dimensional FFTs.

On GPU side, for hybrid Y_1 dimensional FFTs, the execution time is estimated as TG_{2D} shown in equation 4.1.

$$\begin{aligned} TG_{2D} = \quad & \#passes \times max\{[Y_1 \times T_{2d\text{H2D}}(0, R_g) + \\ & T_{Y_1\text{kernel}}(0, R_g) + T_{2d\text{D2H}}(0, R_g)]; [...]; \\ & [Y_1 \times T_{2d\text{H2D}}(\#\text{ streams-1}, R_g) \\ & + T_{Y_1\text{kernel}}(\#\text{ streams-1}, R_g) \\ & + T_{2d\text{D2H}}(\#\text{ streams-1}, R_g)]; \quad \} \end{aligned} \tag{4.1}$$

On CPU side, for hybrid Y_1 dimensional FFTs, the execution time is estimated as $TC_{2D} = \frac{Y_2}{\#thds} \times T_{Y_1\text{fftw}}(1 - R_g)$.

Since synchronization is set after Y_1-step FFT on both GPU and CPU side to guarantee the correctness of results, the execution time of hybrid Y_1 dimensional FFT can be modeled as the maximum of the GPU time and CPU time, i.e., $T_{Y_1} = max\{TG_{2D}, TC_{2D}\}$. And the total time estimation will be consequently calculated as $T_{\text{total}} = max\{TG_{2D}, TC_{2D}\} + T_{Y_2\&X}$. Afterwards, empirical searching is employed to find the parameter values that can make TG_{2D} equal to TC_{2D}, as well as the sub-steps along other dimensions, which indicates the optimal load balancing.

4.2 Load Balancing of 3D FFT

The load balancing in the hybrid 3D FFT framework is similar to that of the 2D cases. Suppose that the total problem size is $Z_1 \times Z_2 \times Y_1 \times Y_2 \times X$. The load ratio of GPU to total work is denoted as R_g along X dimension and ratio of CPU to total problem is $1 - R_g$. Performance parameters for the sub-steps in 3D hybrid FFT are summarized in table 4.2. Two profiling runs still help determine $T_{3d\text{H2D-gpu}}$, $T_{Z_1\text{kernel-gpu}}$, $T_{Y_1\text{kernel-gpu}}$, $T_{Y_2\text{kernel-gpu}}$, $T_{3d\text{D2H-gpu}}$, and $T_{Z_1\text{fftw-cpu}}$, $T_{Y\text{fftw-cpu}}$ as execution time in respective GPU-only and CPU-only case for the parameters in table 4.2.

On GPU side, for hybrid $Z_1\&Y$ dimensional FFTs, the execution time is estimated as TG_{3D} shown in equation 4.2.

$$
\begin{aligned}
TG_{3D} = \quad & \#passes \times max\{[Z_1 \times T_{3d\text{H2D}}(0, R_g) + \\
& T_{Z_1\text{kernel}}(0, R_g) + T_{Y_1\text{kernel}}(0, R_g) + \\
& T_{Y_2\text{kernel}}(0, R_g) + T_{3d\text{D2H}}(0, R_g)]; \quad [.....]; \\
& [Z_1 \times T_{3d\text{H2D}}(\#\text{ streams-1}, R_g) \\
& + T_{Z_1\text{kernel}}(\#\text{ streams-1}, R_g) \\
& + T_{Y_1\text{kernel}}(\#\text{ streams-1}, R_g) \\
& + T_{Y_2\text{kernel}}(\#\text{ streams-1}, R_g) \\
& + T_{3d\text{D2H}}(\#\text{ streams-1}, R_g)]; \}
\end{aligned}
\tag{4.2}
$$

Table 4.2: Parameters for 3D FFT Running Time Estimation.

Parameters	Description
$T_{3d\text{H2D}}(i, R_g)$	$= T_{3d\text{H2D-gpu}} \times R_g$. Time of copying a 3D strided array of size $\frac{R_g \times Y \times Z_2}{\#\text{ passes} \times \#\text{ streams}}$ from host to device in stream i.
$T_{Z_1\text{kernel}}(i, R_g)$	$= T_{Z_1\text{kernel-gpu}} \times R_g$. Time of Z_1-step FFTs computation of concurrent kernel in stream i. Thread block size is $Z_1 W \times max(Z_{11}, Z_{12})$, grid size is $\frac{R_g \times Y \times Z_2}{\#\text{ passes} \times \#\text{ streams}}$.
$T_{Y_1\text{kernel}}(i, R_g)$	$= T_{Y_1\text{kernel-gpu}} \times R_g$. Time of Y_1-step FFTs computation of concurrent kernel in stream i. Thread block size is $Y_1 W$, grid size is $\frac{R_g \times Y_2}{Y_1 W} \times \frac{Z}{\#\text{ passes} \times \#\text{ streams}}$
$T_{Y_2\text{kernel}}(i, R_g)$	$= T_{Y_2\text{kernel-gpu}} \times R_g$. Time of Y_2-step FFTs computation of concurrent kernel in stream i. Thread block size is $Y_2 W$, grid size is $\frac{R_g \times Y_1}{Y_2 W} \times \frac{Z}{\#\text{ passes} \times \#\text{ streams}}$,
$T_{3d\text{D2H}}(i, R_g)$	$= T_{3d\text{D2H-gpu}} \times R_g$. Time of copying a 3D contiguous array of size $\frac{R_g \times Y \times Z_1 \times Z_2}{\#\text{ passes} \times \#\text{ streams}}$ from device to host in stream i.
$T_{Z_1\text{fftw}}$	$= T_{Z_1\text{fftw-cpu}} \times (1 - R_g)$. Time of Z_1-step FFTs on advanced FFTW plan for grouped array of size Y in CPU. Total # of plans is $(1 - R_g) \times X \times Z_2$.
$T_{Y\text{fftw}}(1 - R_g)$	$= T_{Y\text{fftw-cpu}} \times (1 - R_g)$. Time of Y-step FFTs on advanced FFTW plan for grouped array of size $(1 - R_g) \times X$ in CPU. Total # of plans is Z.
$T_{Z_2 \& X}$	Time of subsequent calculation of Z_2 and X dimensional FFTs.

On CPU side, for hybrid $Z_1\&Y$ dimensional FFTs, the execution time is estimated as TC_{3D} represented in equation 4.3.

$$TC_{3D} = \frac{(1 - R_g) \times X \times Z_2}{\#thds} \times T_{Z_1\text{fftw}} + \frac{Z}{\#thds} \times T_{Y\text{fftw}}(1 - R_g) \tag{4.3}$$

Similarly, since a synchronization is set after $Z_1\&Y$-step FFT on both GPU and CPU side, the execution time of hybrid $Z_1\&Y$ dimensional FFT can be modeled as the maximum of the GPU time and CPU time, i.e., $T_{Z_1\&Y} = max\{TG_{3D}, TC_{3D}\}$. The total time estimation is calculated as $T_{\text{total}} = max\{TG_{3D}, TC_{3D}\} + T_{Z_2\&X}$. Empirical searching techniques similar to 2D cases are used to balance the substeps, as well as those along other dimensions.

Chapter 5

PERFORMANCE EVALUATION

In this section, we evaluate the hybrid 2D and 3D FFT implementation on three heterogeneous computer configurations. A single model of CPU, Intel i7 920, is coupled with three different NVIDIA GPUs, i.e. GeForce GTX480, Tesla C2070 and Tesla C2075 in the three experiments. The configurations of the GPUs, CPU and FFT libraries are summarized in table 5.1.

We compare our library in both single- and double-precisions against FFTW and Intel MKL, two of the best performing FFT implementations on CPU. Moreover, our hybrid FFT library is compared with Gu's out-of-card FFT work [6], a highly efficient GPU-based FFT library and the only one that we know can handle the problems sizes larger than GPU memory. The whole design of this performance evaluation is to let us see how much performance improvement can be achieved by using both CPU and GPU in computation, against the best-performing GPU-only or CPU-only FFT implementations. Please note that we can't compare our library with other GPU-based FFT implementations because all of them require problem sizes to be smaller than GPU memory, and therefore are unable to handle the problem sizes used in this evaluation. In FFTW, Streaming Single Instruction Multiple Data Extensions (SSE) on Intel CPU

Table 5.1: Configurations of GPU, CPU, FFTW and MKL.

GPU	**Memory**	**Compute Capability**	**NVCC & CUFFT**
GeForce GTX480	1.5GB	2.0	3.2
Tesla C2070	6GB	2.0	3.2
Tesla C2075	6GB	2.0	3.2
CPU	**Frequency**	**Cores**	**FFTW & MKL**
Intel i7 920	2.66GHz	4	3.3.2 & 10.3

is enabled for better performance. Also FFTW results are got with the 'MEASURE' flag, the second most extensive performance tuning mode. The 'EXHAUSTIVE' flag in FFTW, which represents the most extensive searching and tuning, is not used because the problem sizes in this evaluation are so large that FFTW can't finish its search under the 'EXHAUSTIVE' mode. For example, we tried running FFTW in 'EXHAUSTIVE' mode for a 2^{28} FFT problem, but found FFTW couldn't finish the search in 3 days. In addition, Intel MKL automatically enables SSE at run time. Both FFTW and MKL are chosen to run with four threads. Even though the i7 CPU supports 8 hyperthreads, the 8-thread FFTW and MKL didn't show performance advantage over, actually in some cases were slower than, the 4-thread versions.

All FFT problems are out-of-place with separate input and output with initial inputs filled by random numbers. For double-precision implementation on GTX480, we choose the test cases from 32M points (i.e. 2^{25}) to 256M points (i.e. 2^{28}). 32M-point FFT is twice the maximal problem size that GTX480 memory can accommodate and 256M-point FFT is the maximum problem size that can fit into host memory. For single precision tests on GTX480, the sizes are from 64M points (i.e. 2^{26}) to 512M points (i.e. 2^{29}). Similarly, for Tesla C2070/C2075, test cases are from 256M points (i.e. 2^{28}) to 512M points (i.e. 2^{29}) for double precision implementation and from 512M points (i.e. 2^{29}) to 1024M points (i.e. 2^{30}) for single precision test. The performance of a D dimensional out-of-place complex FFT is evaluated in GFLOPS defined as

$$GFlops = \frac{5M \sum_{d=1}^{D} log_2 N_d}{t} \times 10^{-09} \tag{5.1}$$

where the total problem size is $M = N_1 \cdot N_2 \cdot ... \cdot N_D$ and t is execution time in seconds.

5.1 Performance Tuning with Load Distribution Ratios

For both 2D and 3D FFTs, our performance modeling and empirical searching find the optimal ratio and best performance for different input sizes. To demonstrate the effect and accuracy of load distribution ratio tuning on overall FFT performance, we vary distribution ratio from 0% to 100%.

Figure 5.1 shows the actual and modeled double-precision 2D FFT performance on three different GPUs under different load ratios with problem size $2^{15} \times 2^{13}$. In particular, 0% represents running our hybrid FFT library only on CPU and 100% represents running only on GPU. The two extreme cases will help demonstrating the intrinsic overhead incurred by splitting computation/communication into two devices. Table 5.2 shows the tested values of model parameters for the profiling runs of GPU-only and CPU-only case described in section 4.1 and 4.2. With this preparation, parameters in table 4.1 and 4.2 can be effectively modeled to determine the overall performance of our implementation in GFLOPS. As shown in the figure, the estimated optimal ratio is 100%, 100% and 96% as closed as the actual one measured on GTX480, Tesla C2070 and C2075, respectively. Moreover, the modeled optimal and average performance is 99%, 96%, 95%, and 98%, 93%, 91%, as closed as the actual one measured on the three GPUs, respectively.

Figure 5.2 shows the actual and modeled double-precision 3D FFT performance with size $2^{10} \times 2^9 \times 2^9$. The estimated optimal ratio is successfully identified comparing to the actual one measured on the three GPUs. Moreover, the modeled optimal and average performance is 100%, 95%, 93%, and 98%, 94%, 91%, as closed as the actual one.

The modeling error is mainly caused by the overlapping between kernel computation and data communication, although the asynchronous scheme has been considered comprehensively when designing our model parameters. The secondary reason of error is due to the caching on CPU which causes the runtime performance slightly different from the estimated for different ratios. However, from the accuracy test of modeling described above, our estimation is still effective when determining optimal ratios and the best performance.

In addition, speedups of our optimal performance over GPU-only and CPU-only case are also tested. As shown in figure 5.1, for GTX480, Tesla C2070 and C2075, the optimal ratios of GPU to total work are 71.9%, 78.2% and 78.2%, respectively. The

Table 5.2: Valid Model Parameters in Seconds for FFTs of Size $2^{15} \times 2^{13}$ and $2^{10} \times 2^9 \times 2^9$.

Parameter	Time	Parameter	Time	Parameter	Time
$T_{2d\text{H2D-gpu}}$	0.003	$T_{Y_1\text{kernel-gpu}}$	0.041	$T_{2d\text{D2H-gpu}}$	0.042
$T_{Y_1\text{fftw-cpu}}$	0.195	$T_{Y_2\&X}$	1.137		
$T_{3d\text{H2D-gpu}}$	0.024	$T_{Z_1\text{kernel-gpu}}$	0.014	$T_{Y_1\text{kernel-gpu}}$	0.028
$T_{Y_2\text{kernel-gpu}}$	0.1	$T_{3d\text{D2H-gpu}}$	0.02	$T_{Z_1\text{fftw-cpu}}$	0.0008
$T_{Y\text{fftw-cpu}}$	0.01	$T_{Z_2\&X}$	1.221		

best-balanced performance is 21.4%, 19.1% and 20.7% faster than GPU-only performance, and is 1.09×, 1.59× and 1.76× faster than CPU-only cases. Moreover, for the three GPUs shown in figure 5.2, the optimal ratios of GPU to the total are 75.0%, 78.2% and 78.2%. The best-balanced performance is 25.6%, 22.8% and 23.1% faster than GPU-only performance, and is 1.25×, 1.51× and 1.62× faster than CPU-only case.

Not shown in this figure, but the single-precision performance tuning with ratios has similar curve as that of the double-precision case. Also the optimal ratio of GPU to CPU in single-precision version is larger than that of double precision since GPU has relatively higher performance on single precision operations than CPU.

5.2 Evaluation for 2D Hybrid FFT

We evaluate various 2D FFT problems on the three heterogeneous configurations. The 2D hybrid FFT performance of all test points are reported with the empirically found work distribution ratio of GPU to CPU. In all the figures, the test points are indexed in an increasing order of Y in the problem sizes. Figure 5.3 shows our single-precision 2D FFT performance on Geforce GTX480 with problem sizes from 2^{26} to 2^{29}. On average, our single-precision 2D hybrid FFT on GTX480 achieves 25.5 GFLOPS. Our optimally-distributed performance is 16% faster than Gu's pure GPU version, and is also 95% faster than the 4-thread FFTW and 1.06× faster than the 4-thread MKL. In particular, even if we run our hybrid FFT only on GPU, it is still faster than Gu's work, a high-performance GPU-based FFT implementation, mainly

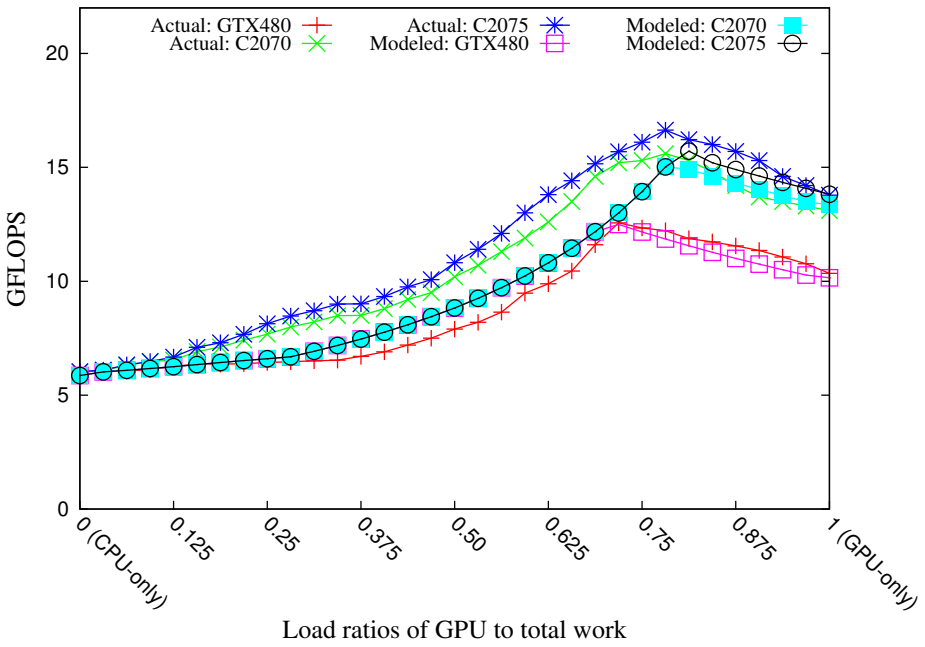

Figure 5.1: Double-precision 2D FFT Performance Tuning.

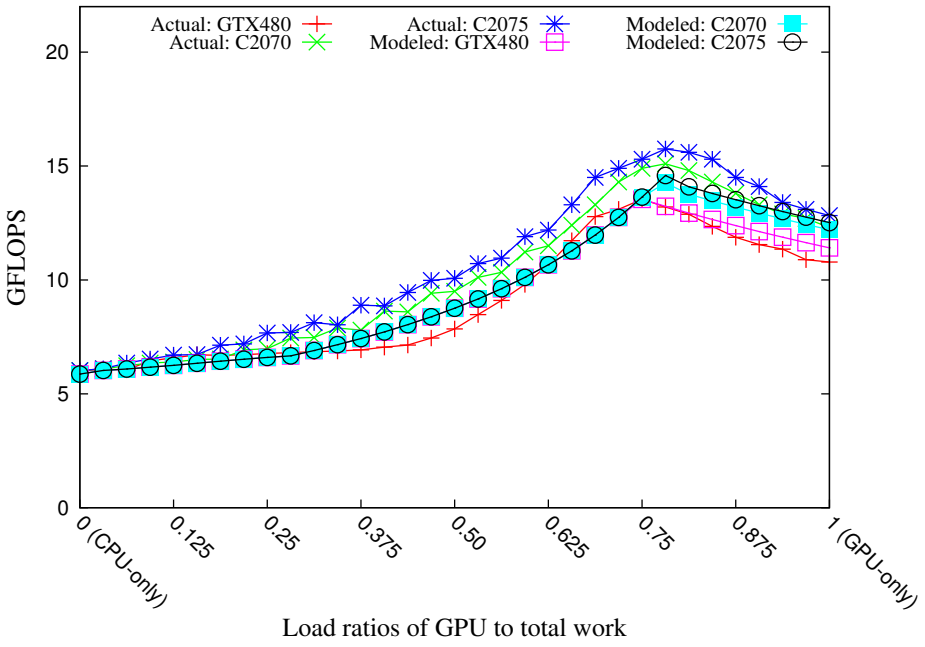

Figure 5.2: Double-precision 3D FFT Performance Tuning.

attributing to the asynchronous transfer schemes in our hybrid algorithm.

Furthermore, we also test 2D hybrid FFT performance in double-precision as shown in figure 5.4. Our double-precision 2D hybrid FFT on GTX480 achieves 13.1 GFLOPS. Moreover, our optimal performance is 20% faster than Gu's pure GPU implementation, and is 98% faster than the 4-thread FFTW and 1.04× faster than the 4-thread MKL.

Additionally, figure 5.5 and figure 5.6 show our large 2D FFT results on the Tesla C2070/C2075 with even larger problem sizes in single and double precision. On average, our single-precision 2D hybrid FFT achieves 37.2 GFLOPS on Tesla C2075 and 33.7 GFLOPS on Tesla C2070, which represent speedups of 26% and 24% over Gu's pure GPU implementation, 2.23× and 1.93× over the 4-thread FFTW, and 2.41× and 2.09× over the 4-thread MKL, respectively.

For double precision, the performance is 19.1 GFLOPS and 17.8 GFLOPS on Tesla C2075 and C2070, which represent 29% and 28% speedups over Gu's pure GPU implementation, 2.08× and 1.87× speedups over the 4-thread FFTW and 2.24× and 2.02× speedups over the 4-thread MKL.

Not shown in figures, but the overall performance of our hybrid 2D FFT is 2.22× and 22% faster than CPU-only and GPU-only case.

Particularly notable is that as Y increases, the performance of both FFTW and MKL decreases rapidly because the data locality loses rapidly along the Y dimensional computation when Y increases. On the contrary, our hybrid FFT demonstrates a much more stable performance.

5.3 Evaluation for 3D Hybrid FFT

Figure 5.7, 5.9 and figure 5.8, 5.10 show the performance of our single- and double-precision 3D hybrid FFT on GTX480 and Tesla C2075/C2070. On average our library achieves 18.4 GFLOPS on GTX480, 23.2 GFLOPS on C2075 and 21.5 GFLOPS on C2070.

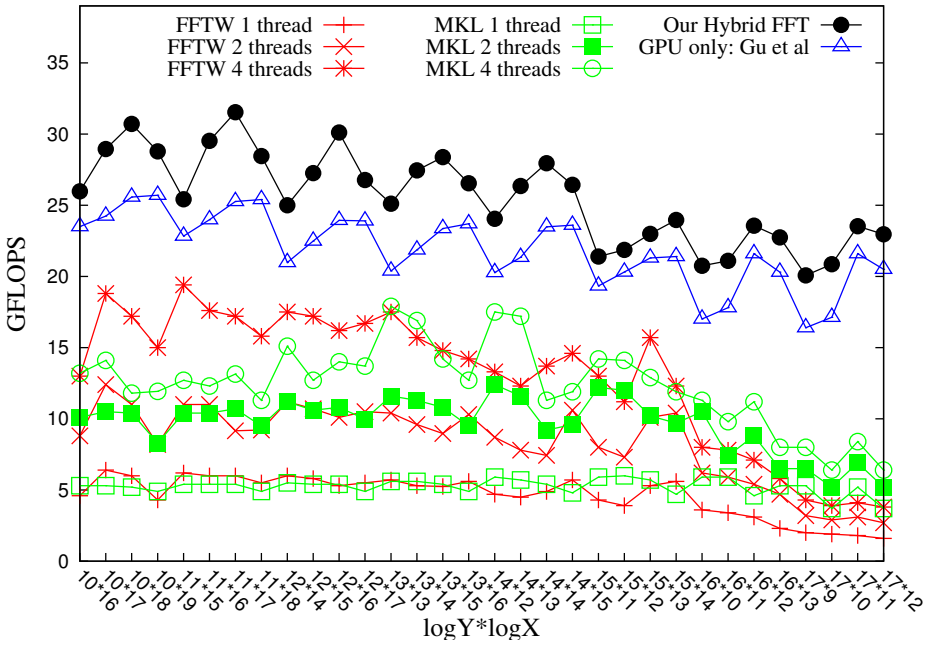

Figure 5.3: Single-precision 2D FFT of Size from 2^{26} to 2^{29} on GTX480.

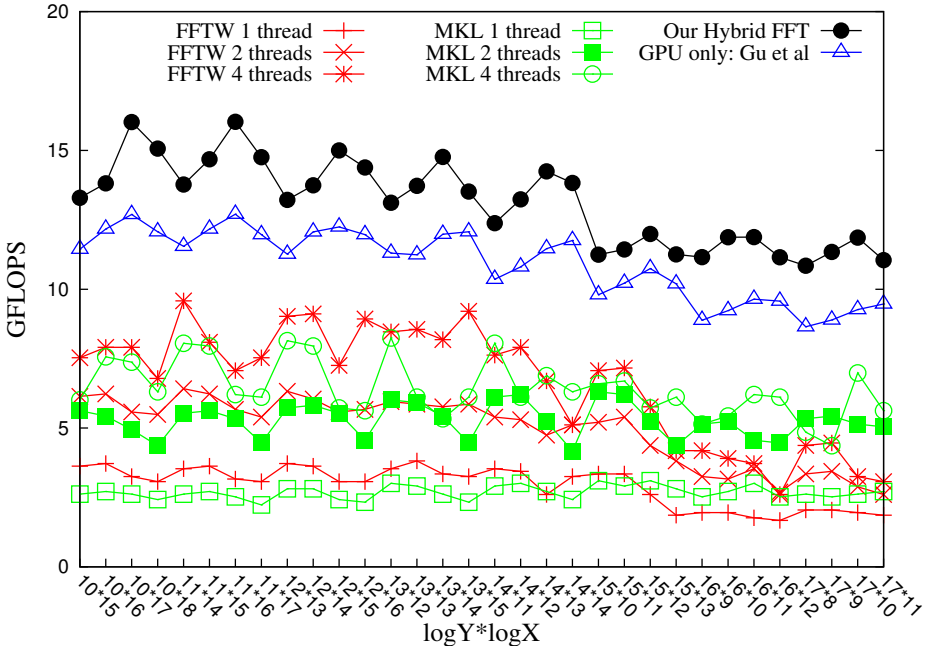

Figure 5.4: Double-precision 2D FFT of Size from 2^{25} to 2^{28} on GTX480.

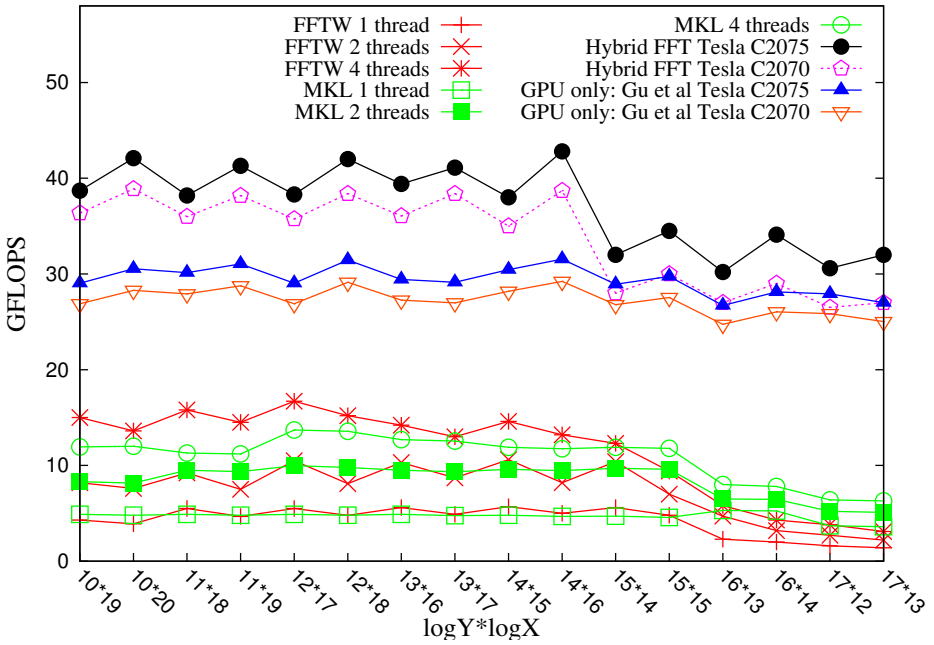

Figure 5.5: Single-precision 2D FFT of Size from 2^{29} to 2^{30} on Tesla.

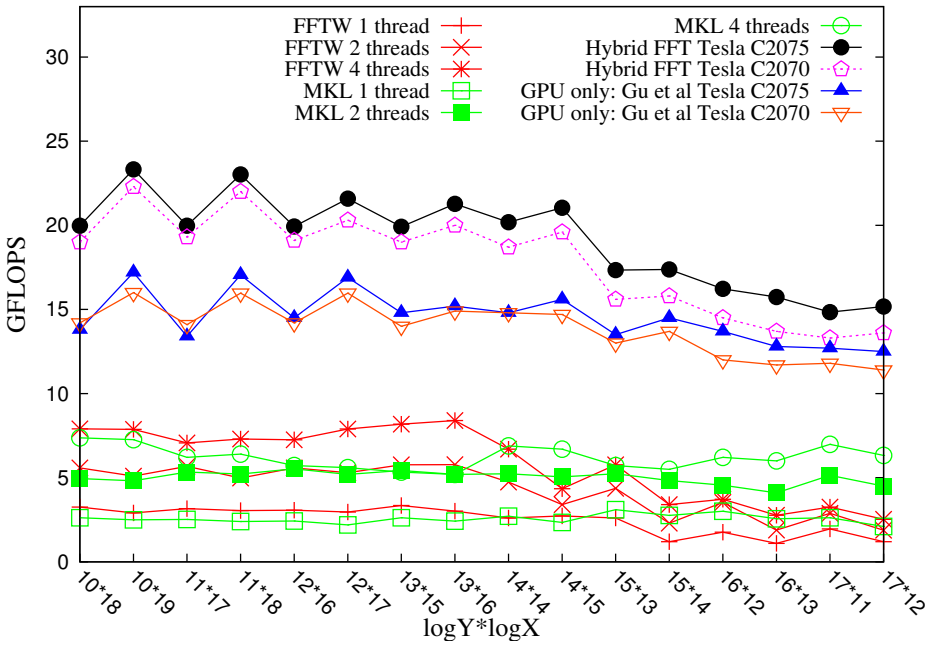

Figure 5.6: Double-precision 2D FFT of Size from 2^{28} to 2^{29} on Tesla.

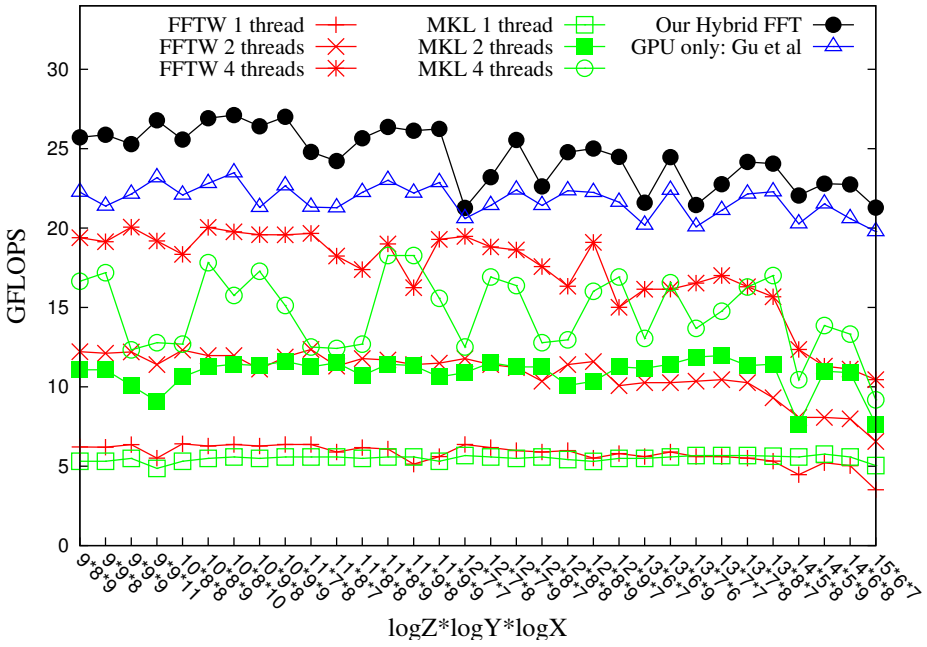

Figure 5.7: Single-precision 3D FFT of Size from 2^{26} to 2^{29} on GTX480.

On average, our hybrid 3D FFT library is 19.5% faster than Gu's GPU only FFT implementation, 74.2% faster than the 4-thread FFTW and 1.09× faster than MKL. Not shown in figures, but our 3D performance is 2.02× and 18% faster than CPU-only and GPU-only case. Similar to their 2D performance, FFTW's and MKL's 3D performance decrease quickly as Z increases due to the loss of data locality though MKL generally performs better than FFTW for large Zs. Our hybrid library generally maintains its good performance for the same large Z cases.

5.4 Accuracy of Our Hybrid FFT

The correctness of our hybrid FFT library is verified against FFTW and MKL. All three libraries are tested with the same single-precision input data randomly chosen from -0.5 to 0.5 and the difference in output is quantified as normalized RMSE over the whole data set. The normalized RMSE evaluates the relative degree of deviations

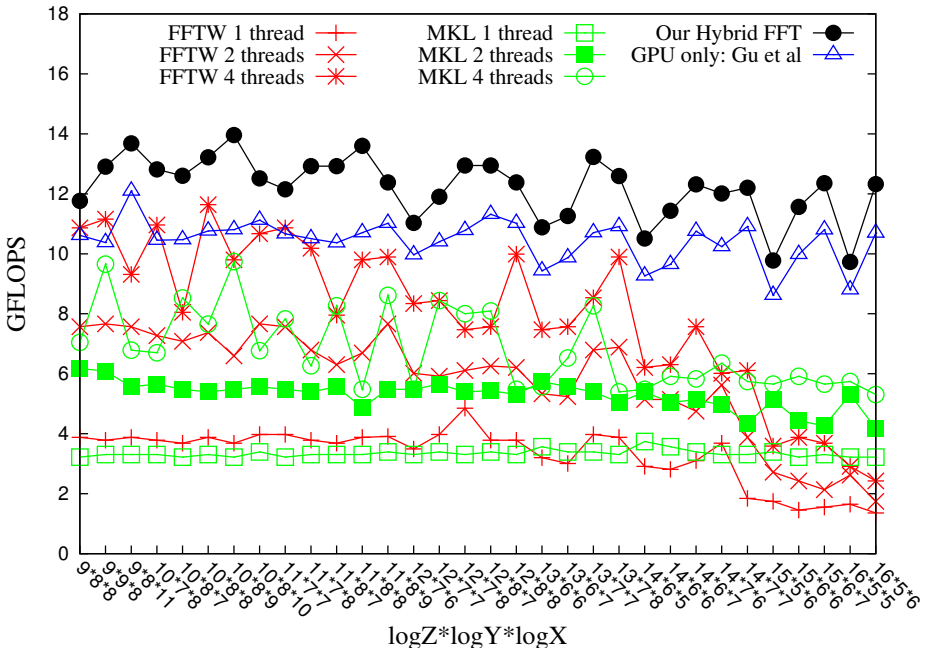

Figure 5.8: Double-precision 3D FFT of Size from 2^{25} to 2^{28} on GTX480.

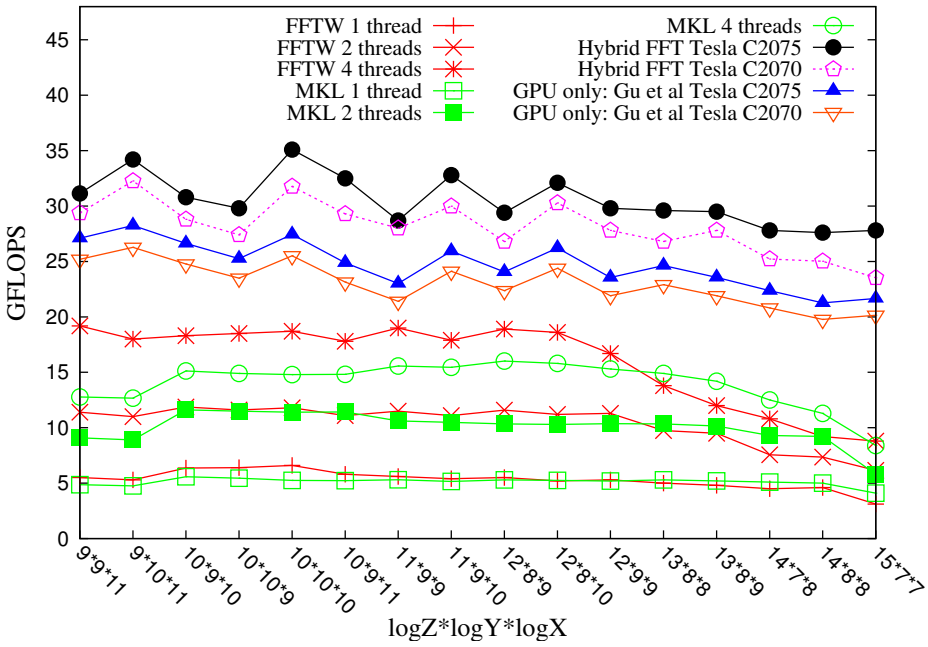

Figure 5.9: Single-precision 3D FFT of Size from 2^{29} to 2^{30} on Tesla.

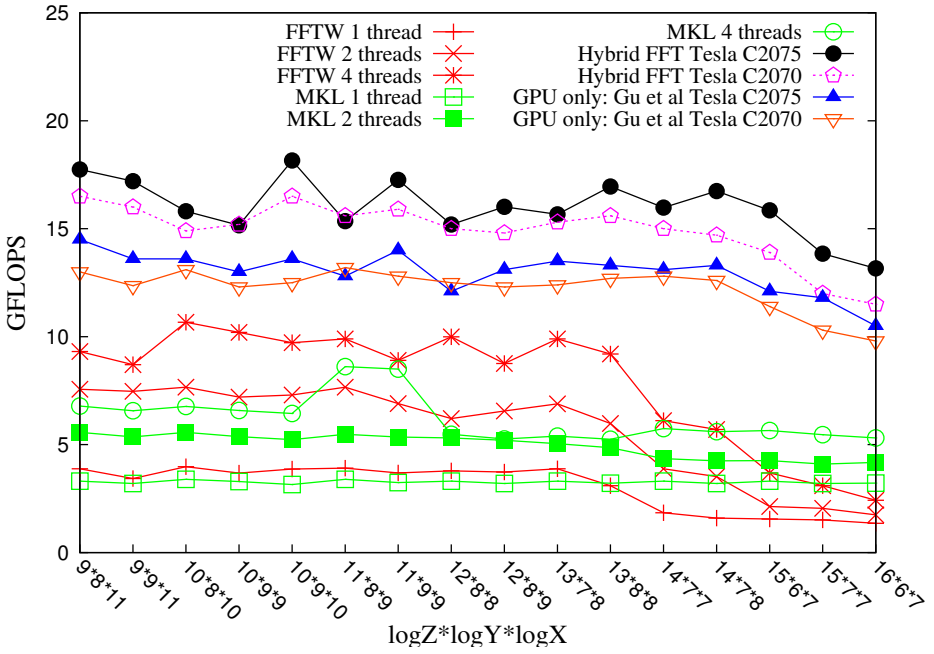

Figure 5.10: Double-precision 3D FFT of Size from 2^{28} to 2^{29} on Tesla.

and is a wildly used metric for numeric accuracy. The normalized RMSE is defined as

$$\sqrt{\frac{\sum_{i=0}^{N-1}(X_i - R_i)^2 + (Y_i - S_i)^2}{2N}} \Big/ \sqrt{\frac{\sum_{i=0}^{N-1}(R_i^2 + S_i^2)}{2N}}. \tag{5.2}$$

The normalized RMSEs of single- and double-precision for both 2D and 3D FFTs are shown in Figure 5.11 and figure 5.12. As we can see the normalized RMSE is extremely small and is in the range from 2.41×10^{-07} to 3.18×10^{-07} for single precision and 5.82×10^{-16} to 8.02×10^{-16} for double precision. In other words, our hybrid FFT library produces almost the same results as FFTW and MKL.

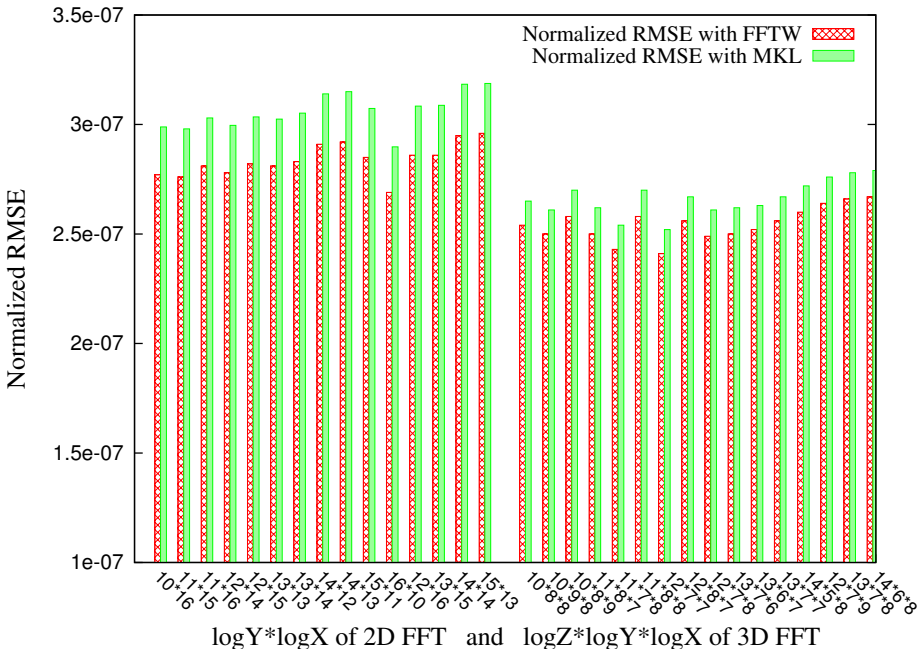

Figure 5.11: Accuracy of Single-precision Hybrid 2D/3D FFT.

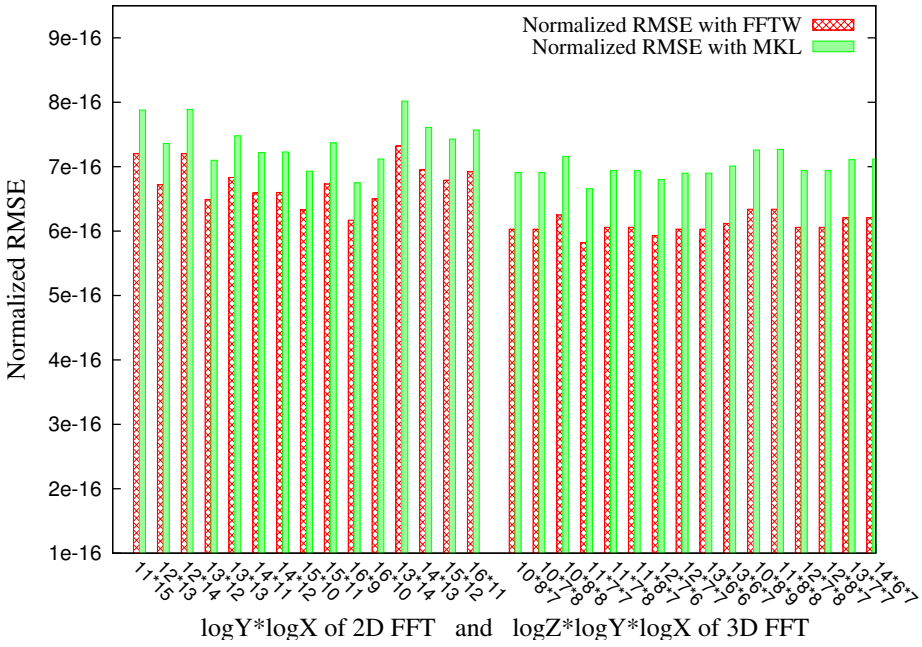

Figure 5.12: Accuracy of Double-precision Hybrid 2D/3D FFT.

Chapter 6

CONCLUSION AND FUTURE WORK

In this project, we proposed a hybrid FFT library that concurrently uses both CPU and GPU to compute large FFT problems. The library has three key components: a decomposition paradigm that mixes two FFT algorithms to extract different types of computation and communication patterns for the two different processor types; a load balancer that assigns workloads according the computation capability of CPU and GPU; and an optimizer that empirically tune the library to find the best tradeoff among communication, computation and the overlapping between the previous two factors. Overall, our hybrid library outperforms two best performing FFT implementations by 121% and 145%, respectively.

Our future work would be able to implement our hybrid approach in the platform possessing more GPU/CPU resources. Specifically, we will extend our library to exploit the parallelism in a cluster with multi-GPUs and multi-CPUs. Our hybrid FFT approach and performance model would still be capable of working on such advanced heterogeneous computing systems. Additionally, some other flexible applications, such as sparse FFT algorithm and modeling of turbulent cloud dynamics, could be further implemented very efficiently using our hybrid method.

BIBLIOGRAPHY

[1] "NVIDIA CUFFT Library" http://developer.nvidia.com.

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

[3] 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.

[4] N. 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*, pages 1–12, 2008.

[5] 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. ISBN 978-1-4503-0018-6.

[6] L. Gu, J. Siegel, and X. Li. Using gpus to compute large out-of-card ffts. In *Proceedings of the international conference on Supercomputing*, ICS '11, pages 255–264, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0102-2.

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

[8] 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.

[9] P. Duhamel and M. Vetterli. Fast fourier transforms: a tutorial review and a state of the art. *Signal Process.*, 19(4):259–299, Apr. 1990. ISSN 0165-1684.

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

[11] AV Oppenheim and RW et al. Schafer. Discrete-Time Signal Processing. 1999.

[12] IJ Good. The interaction algorithm and practical Fourier analysis. *Journal of the Royal Statistical Society. Series B (Methodological)*, 20(2):361–372, 1958.

[13] CM Rader. Discrete Fourier transforms when the number of data samples is prime. *Proceedings of the IEEE*, 56(6):1107–1108, 1968.

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

[15] M. Frigo and S. G. Johnson. The design and implementation of fftw3. *Proceeding of the IEEE*, 93(2):216–231, 2005.