PARALLEL FFT PROGRAM OPTIMIZATION ON HETEROGENEOUS COMPUTERS

by

Shuo Chen

A dissertation submitted to the Faculty of the University of Delaware in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Electrical and Computer Engineering

Fall 2015

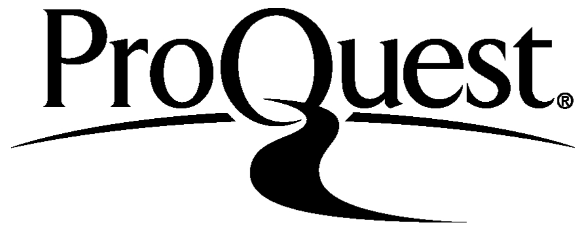
© 2015 Shuo Chen
All Rights Reserved

ProQuest Number: 10014760

All rights reserved

INFORMATION TO ALL USERS
The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript
and there are missing pages, these will be noted. Also, if material had to be removed,
a note will indicate the deletion.



ProQuest 10014760

Published by ProQuest LLC (2016). Copyright of the Dissertation is held by the Author.

All rights reserved.
This work is protected against unauthorized copying under Title 17, United States Code
Microform Edition © ProQuest LLC.

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 - 1346

PARALLEL FFT PROGRAM OPTIMIZATION ON HETEROGENEOUS COMPUTERS

by

Shuo Chen

Approved: ____

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

Approved: ____

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

Approved: ____

Ann L. Ardis, Ph.D.
Interim Vice Provost for Graduate and Professional Education

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: ______________________________

Xiaoming Li, Ph.D.
Professor in charge of dissertation

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: ______________________________

Guang R. Gao, Ph.D.
Member of dissertation committee

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: ______________________________

Chengmo Yang, Ph.D.
Member of dissertation committee

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: ______________________________

Lian-Ping Wang, Ph.D.
Member of dissertation committee

ACKNOWLEDGEMENTS

First of all, I wish to thank my adviser, Professor Xiaoming Li, for his guidance, support, encouragement and great help throughout my Ph.D. studies. His dedication to work and research, and his patience and encouragement for students will always be an enlightenment for me in my life. Without the myriad of discussions and talks with him, the works and results presented in this dissertation would never be possible.

Second, I want to thank all of my committee members for their time and useful suggestions: Professor Xiaoming Li, Professor Guang R. Gao, Professor Chengmo Yang and Professor Lian-Ping Wang.

I would also like to thank my lab colleagues, Liang Gu, Sha Li and Yuanfang Chen, who helped me a lot during the writing of this dissertation.

Last but not least, I would show my gratitude to my parents and my grandparents for all their support and care on me.

TABLE OF CONTENTS

LIST OF TABLES x
LIST OF FIGURES xi
ABSTRACT xv

Chapter

1 INTRODUCTION 1

1.1 Parallel Heterogeneous Computing 1

1.1.1 General Parallel Computing 1
1.1.2 Parallel Heterogeneous Computer System Incorporating CPU and GPGPU 2
1.1.3 CUDA Parallel Computing on GPGPU 3

1.2 General Fast Fourier Transform Heterogeneous Computing 6

1.2.1 Fast Fourier Transforms on GPGPUs 6
1.2.2 Problem Statement 9
1.2.3 Motivations 9

1.3 Sparse Fast Fourier Transform High Performance Computing 11

1.3.1 Sparse Fast Fourier Transform Algorithms 11
1.3.2 Problem Statement 12
1.3.3 Motivations 12

1.4 Dissertation Contributions 13

1.4.1 Hybrid FFTs Optimization and Parallelization 13
1.4.2 Sparse FFTs Optimization and Parallelization 15

1.5 Dissertation Organization 17

2 BACKGROUND **19**

2.1 Overview of General Fast Fourier Transform Algorithms 19

2.1.1 The General Radix-2 FFT Algorithm 19
2.1.2 The General Cooley-Tukey FFT Algorithm 20
2.1.3 Other General FFT Algorithms 21
2.1.4 I/O Tensor Representation 21

2.2 Prior Works of General Fast Fourier Transform Implementations on GPGPUs 22
2.3 Overview of Sparse Fast Fourier Transform Algorithms 23

2.3.1 Prior Works of Dense Fast Fourier Transform Algorithms . . . 23
2.3.2 Prior Studies of Sparse Fast Fourier Transform Algorithms . . 23

2.4 Objectives of Approaches 24

2.4.1 Objective of Hybrid FFT Approach 24
2.4.2 Objective of Sparse FFT Approach 25

3 RADIX DECOMPOSITION BASED HYBRID FFTS ON HETEROGENEOUS GPU-CPU COMPUTERS **26**

3.1 Overview of Our Approach 26
3.2 Radix Decomposition based Hybrid FFTs on Heterogeneous Computers 29

3.2.1 Radix Decomposition based Hybrid 2D FFT Framework . . . 29
3.2.2 Radix Decomposition based Hybrid 3D FFT Framework . . . 36

3.3 PCI Data Transfer Scheme and Load Balance 37

3.3.1 Data Transfer Scheme Through PCI Bus 37
3.3.2 Load Balancing Between GPU and CPUs 40

3.4 Performance Evaluation 43

3.4.1 Evaluation of 2D Hybrid FFT 44
3.4.2 Evaluation of 3D Hybrid FFT 45

3.4.3 Accuracy of Our Hybrid FFT . . . 46

3.5 Chapter Summary . . . 47

4 A CO-OPTIMIZED WELL-TUNED PARALLEL HYBRID GPU-CPUS FFT LIBRARY FOR LARGE FFT PROBLEMS . . 49

4.1 Overview of Our Approach . . . 49
4.2 Co-Optimized Hybrid 2D FFT Framework . . . 50

4.2.1 Load Distribution . . . 51
4.2.2 Optimizations on GPU . . . 51
4.2.3 Asynchronous Strided Data Transfer . . . 53

4.2.3.1 Data Transfer Scheme . . . 53
4.2.3.2 PCI Bandwidth Evaluation . . . 54
4.2.3.3 Comparison to CUFFT . . . 55

4.2.4 Optimizations on CPUs . . . 56
4.2.5 Cooperations of CPUs and GPU . . . 56
4.2.6 Comparison to Other Heterogeneous FFT Implementation . . 57

4.3 Co-Optimized Hybrid 3D FFT Framework . . . 57

4.3.1 Load Distribution . . . 59
4.3.2 Optimizations for GPU and Data Transfer . . . 61
4.3.3 Cooperations of CPUs and GPU . . . 62

4.4 Load Balancing between GPU and CPUs with Empirical Modeling and Tuning . . . 62

4.4.1 Load Balancing of 2D FFT . . . 63
4.4.2 Load Balancing of 3D FFT . . . 65

4.5 Performance Evaluation . . . 66

4.5.1 Performance Tuning . . . 68
4.5.2 Evaluation of 2D Hybrid FFT . . . 71
4.5.3 Evaluation of 3D Hybrid FFT . . . 76
4.5.4 Accuracy of Our Hybrid FFT . . . 78

4.6 Chapter Summary . . . 80

5 AN INPUT-ADAPTIVE ALGORITHM FOR HIGH PERFORMANCE SPARSE FAST FOURIER TRANSFORM . . 81

5.1 Overview of Sparse FFT Algorithms and Our Approach 81
5.2 Input Adaptive Sparse FFT Algorithm 85

5.2.1 General Input-Adaptive Sparse FFT Algorithm 85

5.2.1.1 Notations and Assumptions 85
5.2.1.2 Hashing Permutation of Spectrum 86
5.2.1.3 Flat Window Functions 87
5.2.1.4 Subsampled FFT 89
5.2.1.5 Reverse Hash Function for Location Recovery 89
5.2.1.6 Algorithm Description 89
5.2.1.7 The Computational Complexity 90

5.2.2 Optimized Input-Adaptive Sparse FFT Algorithm 90

5.2.2.1 Input-Adaptive Shifting 91
5.2.2.2 Optimized Algorithm 92

5.2.3 Hybrid Input-Adaptive Sparse FFT Algorithm 92
5.2.4 Real World Application 93

5.3 Experimental Evaluation 95

5.3.1 General Input-Adaptive Sparse FFT Algorithm 95
5.3.2 Optimized Input-Adaptive Sparse FFT Algorithm 97
5.3.3 Evaluation of Real World Application 98

5.4 Chapter Summary 99

6 PARALLEL INPUT-ADAPTIVE SPARSE FAST FOURIER TRANSFORM FOR STREAM PROCESSING 100

6.1 Overview of Sequential Sparse FFTs and Our Parallel Input Adaptive Sparse FFT Approach 100
6.2 Parallel Input-Adaptive Sparse FFT Algorithm 102

6.2.1 General Description of Parallel Input Adaptive Sparse FFT . . 103
6.2.2 Parallelism Exploitation and Kernel Execution 104

6.2.3 Performance Optimizations 105
6.2.4 Tuned GPU based FFT Library 106

6.3 Input Adaption Heuristics and Process 107

6.3.1 Scenario Establishment 107
6.3.2 Input Adaption for Homogeneous Signals 108
6.3.3 Input Adaption for Discontinuous Signals 110

6.4 Performance Evaluation 111

6.4.1 Input-Adaptive Sparse FFT Algorithm 112

6.4.1.1 Sequential Input-Adaptive Sparse FFT 113
6.4.1.2 Parallel Input-Adaptive Sparse FFT 114

6.4.2 Detection for Signal Discontinuity 115
6.4.3 Performance of Input Adaption Process 120
6.4.4 Precision of Our Sparse FFT 123

6.5 Chapter Summary 124

7 CONCLUSION AND FUTURE WORK 126

BIBLIOGRAPHY 128

LIST OF TABLES

3.1 Parameters for Running Time Estimation. 42

3.2 Configurations of GPU, CPU, FFTW, MKL and CUFFT. 43

4.1 Parameters for 2D FFT Running Time Estimation. 64

4.2 Parameters for 3D FFT Running Time Estimation. 65

4.3 Configurations of GPU, CPU, FFTW and MKL. 67

4.4 Valid Model Parameters in Seconds for FFTs of Size $2^{15} \times 2^{13}$ and $2^{10} \times 2^9 \times 2^9$. 71

5.1 Experimental Configurations. 95

6.1 Sub-steps Parameters of Input Adaption. 111

6.2 Configurations of GPUs and CPU. 112

6.3 Time of Input Adaption Substeps 121

LIST OF FIGURES

1.1 Comparison between General CPU and GPU (Image from Nvidia CUDA C programming guide [65]). 2

1.2 A General Heterogeneous Computer System Incorporating CPU and GPU. 3

1.3 Floating Point Operations per Second (FLOPS) for the CPU and GPU (Image from Nvidia CUDA C programming guide [65]). . . . 4

1.4 Memory Bandwidth for the CPU and GPU (Image from Nvidia CUDA C programming guide [65]). 5

1.5 CUDA Heterogeneous Programming Mode (Image from Nvidia CUDA C programming guide [65]). 7

1.6 CUDA Hardware Model (Image from Nvidia CUDA C programming guide [65]). 8

3.1 Natural Parallelism Exploitation of a Naive 2D FFT and Our Flexible Fine Grained Approach. 27

3.2 Computation of Our Out-of-card 2D FFT. 30

3.3 First Phase of the 1st-round of Our Hybrid 2D FFT Optimizations on Heterogeneous GPU and CPU. 33

3.4 Result Combination in Each Sub-stage: (a) The results reside in GPU global memory and CPU memory after the first phase of our hybrid FFT optimizations in 1st-round 2D FFT; (b) Results of the second phase of our hybrid FFT; (c) Results of the third phase; (d) Results of GPU side and CPU side are combined into host memory to produce the total output of 1st-round 2D FFT. 34

3.5 Computation of Our Out-of-card 3D FFT with 4 Passes in Each Round. 36

3.6 Concurrent Data Transfer Scheme from CPU to GPU Through PCIe Bus. . . . 39

3.7 Concurrent Data Transfer Scheme from GPU to CPU Through PCIe Bus. . . . 40

3.8 Concurrent Execution Flow of Our 2D FFT. . . . 41

3.9 Performance of Single-precision 2D Hybrid FFT. . . . 45

3.10 Performance of Double-precision 2D Hybrid FFT. . . . 46

3.11 Performance of 3D Single-precision Hybrid FFT. . . . 47

3.12 Accuracy of 2D and 3D Hybrid FFT in Single Precision. . . . 48

4.1 Overview of Co-Optimized Hybrid Large Out-of-card 2D FFT. . . . 52

4.2 PCI Bandwidth of Different Data Transfer Schemes. . . . 55

4.3 Double Precision 2D FFT Performance Tuning Comparison of Our Co-Optimized Hybrid FFT against Hybrid CUFFT/FFTW Library. 58

4.4 Overview of Co-Optimized Hybrid Large Out-of-card 3D FFT. . . . 60

4.5 Double-precision 2D FFT Performance Tuning. . . . 72

4.6 Double-precision 3D FFT Performance Tuning. . . . 72

4.7 Single-precision 2D FFT of Size from 2^{26} to 2^{29} on GTX480. . . . 74

4.8 Double-precision 2D FFT of Size from 2^{25} to 2^{28} on GTX480. . . . 74

4.9 Single-precision 2D FFT of Size from 2^{29} to 2^{30} on Tesla. . . . 75

4.10 Double-precision 2D FFT of Size from 2^{28} to 2^{29} on Tesla. . . . 75

4.11 Single-precision 3D FFT of Size from 2^{26} to 2^{29} on GTX480. . . . 76

4.12 Double-precision 3D FFT of Size from 2^{25} to 2^{28} on GTX480. . . . 77

4.13 Single-precision 3D FFT of Size from 2^{29} to 2^{30} on Tesla. . . . 77

4.14 Double-precision 3D FFT of Size from 2^{28} to 2^{29} on Tesla. 78

4.15 Accuracy of Single-precision Hybrid 2D/3D FFT. 79

4.16 Accuracy of Double-precision Hybrid 2D/3D FFT. 79

5.1 Hashing Based Permutation. 84

5.2 Application of Our Input Adaptive Sparse FFT Algorithm. 85

5.3 An Example of Dolph Chebyshev, Gaussian, Kaiser Flat Window Function with $N = 1024$. 88

5.4 Performance of Our General Sparse FFT in Sequential Case. 96

5.5 Performance of Our Optimized Sparse FFT in Sequential Case. . . 97

5.6 Performance of Our Algorithm in a Real-world Application. 98

6.1 Binning of Non-zero Fourier Coefficients. 104

6.2 Working Flow of GPU Parallelization. 106

6.3 An Application Example on Video Streams. 108

6.4 Sequential Performance vs. Signal Size. 114

6.5 Sequential Performance vs. Sparsity Parameter. 115

6.6 Parallel Performance vs. Signal Size on Three GPUs. 116

6.7 Parallel Performance vs. Sparsity Parameter on Three GPUs. . . . 116

6.8 First Partial Detection for Case-1 Discontinuity with 3 Segments and 32 Frames per Segment. 118

6.9 Partial Sampling Detection for Case-1 Discontinuity with 3 Segments and 32 Frames per Segment. 118

6.10 Partial Sampling Detection for Case-2 Discontinuity with 3 Segments and 32 Frames per Segment. 119

6.11 Partial Sampling Detection for Case-2 Discontinuity with 3 Segments and 128 Frames per Segment. 119

6.12 Performance of Input Adaption with One Segment on CPU. 122

6.13 Performance of Input Adaption with One Segment on GPUs. 122

6.14 Performance of Input Adaption with Three Segments on CPU. . . . 123

6.15 Performance of Input Adaption with Three Segments on GPUs. . . 124

ABSTRACT

Generating high performance Fast Fourier Transform (FFT) library is an important research topic for the traditional processors, CPUs, and new accelerators, like Graphics Processing Units (GPUs). It is not rare that large scientific and engineering computation, such as physics simulations, signal processing and data compression, spend majority of execution time on large size FFTs. Such FFT implementations require large amount of computing resources and memory bandwidth.

On the system side, in spite of highly influential results in prior FFT work on GPUs, the GPU performance is severely restricted by the limited memory size and the low bandwidth of data transfer through PCI channel. Additionally, current GPU based FFT implementation only uses GPU to compute, but employs CPU as a mere memory-transfer controller. The computing power of CPUs is wasted. On the algorithmic side, input signals are frequently sparse. If we know that an input is sparse, the computational complexity of FFT can be reduced. Many sparse FFT algorithms have been proposed to improve sparse FFTs efficiency. However, the existing sparse FFT implementations are confined to serial execution and are input oblivious in the sense that how the algorithms work is not affected by input characteristics.

In this dissertation, we present two high performance optimization strategies. First, we study the problems of current GPU based FFT implementations, and propose a hybrid approach for 2D and 3D FFT, which concurrently executes both multithreaded CPU and GPU in a heterogeneous computer to accelerate large FFT problems that cannot fit into GPU memory. Within the scheme, an empirical performance modeling is constructed to determine optimal load balancing between CPU and GPU, and an optimizer is proposed to exploit substantial parallelism for both GPU and CPUs and to overlap communication with computation. Second, we investigate the existing sparse

FFT algorithms and propose an input adaptive model for algorithmic parallelization. In particular, the algorithm takes advantage of the similarity between input samples to save much computation and to exploit substantial data parallelism. The solution has runtime sub-linear to the input size and gets rid of coefficient estimations dependencies, both of which improve parallelism and performance.

Chapter 1

INTRODUCTION

In this dissertation, I elaborate on two high performance optimization strategies. First, I investigate the problems of current GPU based FFT implementations. I propose a hybrid approach for multi-dimensional FFTs, which concurrently executes both CPU and GPU in a high performance heterogeneous computer to accelerate different FFT problems. The problem size can be larger than GPU memory. Within the scheme, an empirical performance modeling and tuning is constructed to determine optimal load balancing between CPU and GPU, and an optimizer is proposed to exploit substantial parallelism for both GPU and CPUs, to optimize performance for GPU and CPU, and to purposefully expose opportunities of overlapping communication with computation. Second, I investigate existing sparse FFT algorithms and propose an input adaptive algorithm for parallelization. In particular, the algorithm takes advantage of the similarity between input samples to save much computation and to exploit substantial data parallelism. The solution has runtime sub-linear to the input size and gets rid of recursive coefficient estimation of traditional sparse FFTs, both of which improve parallelism and performance.

1.1 Parallel Heterogeneous Computing

1.1.1 General Parallel Computing

The general parallel computing is to divide an entire computational problem into several subproblems such that those subproblems can be solved in parallel. It usually makes use of multiple computing resources simultaneously to parallelize the subproblems. Instructions from each part can execute concurrently with each other on different processors.

Figure 1.1: Comparison between General CPU and GPU (Image from Nvidia CUDA C programming guide [65]).

The main reason of parallel computing is to improve speed and performance, to save time and cost, to solve larger and more complex problem, and to make better use of parallel computing resources of underlying computer systems.

1.1.2 Parallel Heterogeneous Computer System Incorporating CPU and GPGPU

Heterogeneous computer for parallel computing is often called "heterogeneous computing", which refers to the system that uses more than one type of processor to compute. It typically incorporates one or more traditional processors, CPUs, as well as one or more new accelerators, like General Purpose Graphics Processing Units (GPGPUs).

Shown in figure 1.1, due to the fact that GPU devotes more transistors to data processing than CPU, GPU becomes a highly parallel, multithreaded, many-core processor with tremendous computational power and very high memory bandwidth.

In a general heterogeneous system, as shown in figure 1.2, CPU works in host side and performs as a master coordinator to partition and distribute the parallel work to GPU which works in device side. In addition, CPU is an asynchronous memory controller to operate device memory allocation and memory transfer, to invoke GPU

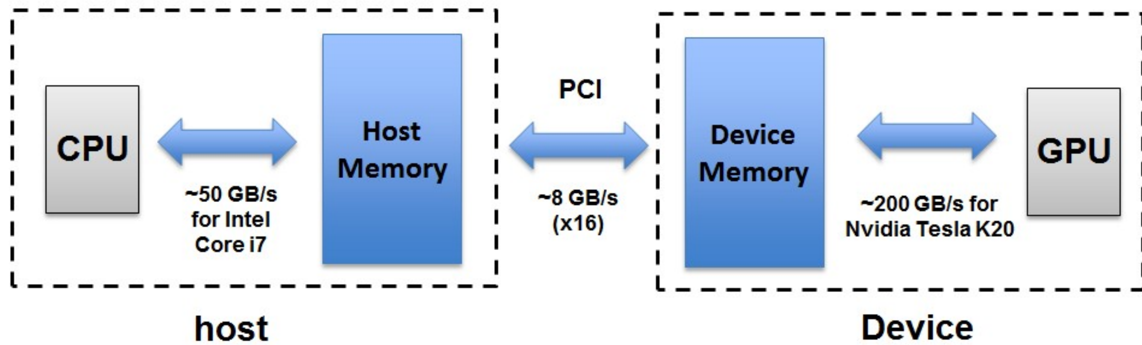


Figure 1.2: A General Heterogeneous Computer System Incorporating CPU and GPU.

kernels, and to gather the result sent back from GPU to output result of several specific parallel computations. Moreover, CPU and GPU are connected through a PCIe channel.

1.1.3 CUDA Parallel Computing on GPGPU

Due to the highly parallel architectural capacity of GPU, it supports for thousands of threads that can execute and access the device memory at the same time. Therefore, as shown in figure 1.3, GPU devotes to much higher Floating Point Operations per Second (FLOPS) at its peak performance than that of CPU. Additionally, as shown in figure 1.4, GPU also renders much higher memory bandwidth than that of CPU.

The Compute Unified Device Architecture (CUDA) [65, 58, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76] parallel programming model is one of the most successful programming models on GPGPU parallel computing. The CUDA C is designed for the programmers familiar with C language to efficiently program and implement parallel applications that can be executed and paralllelized on Nvidia GPUs. CUDA also provides the abstraction of the underlying GPU system, like thread hierarchy, memory hierarchy, and synchronization processing, which are exposed to the programmer as a set of language extension.

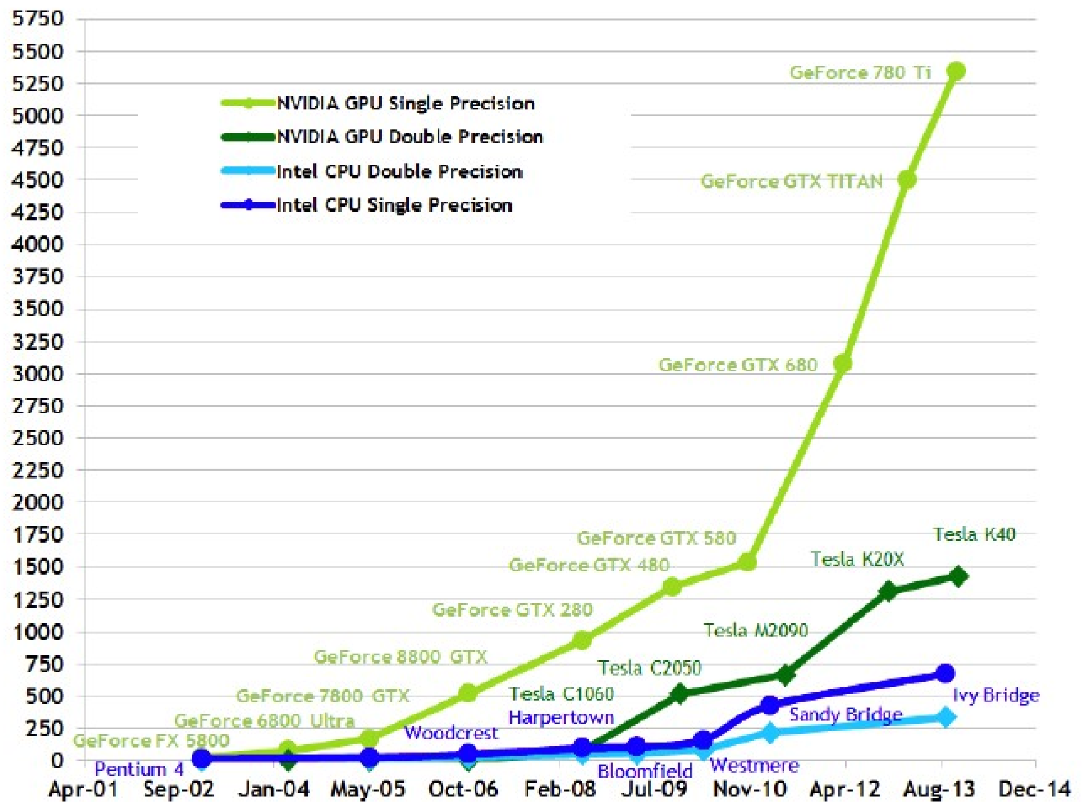


Figure 1.3: Floating Point Operations per Second (FLOPS) for the CPU and GPU (Image from Nvidia CUDA C programming guide [65]).

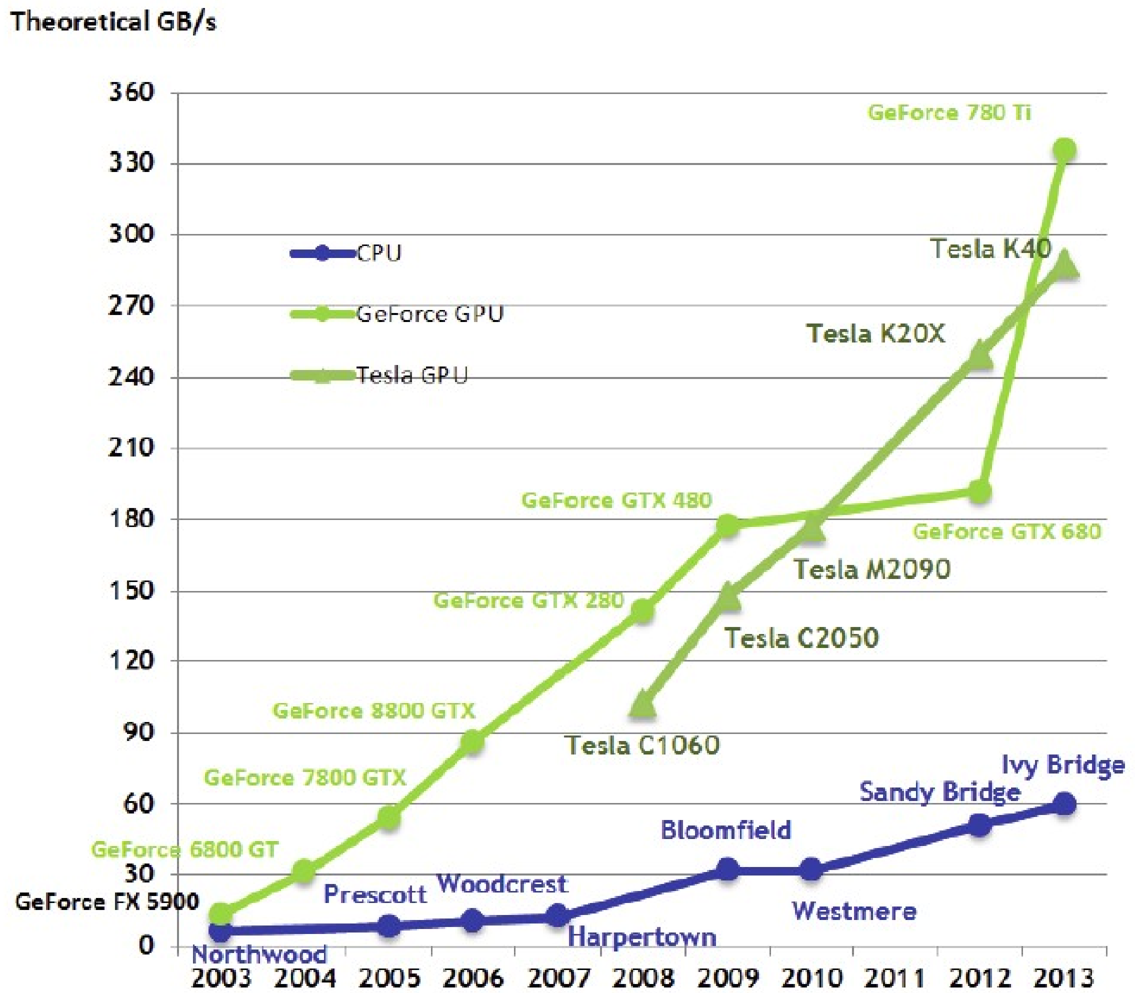


Figure 1.4: Memory Bandwidth for the CPU and GPU (Image from Nvidia CUDA C programming guide [65]).

Figure 1.5 shows a general CUDA heterogeneous programming mode. A GPU has a multi-level thread hierarchy with a grid of thread blocks and many scalar threads in each thread block. So it can support thousands of threads running at the same time. Moreover, CPU is on host side, while GPU is on device side. The CPU works as an asynchronous memory controller to asynchronously invoke GPU kernel functions and to transfer data between CPU and GPU.

As shown in figure 1.6, CUDA also has a multi-level memory hierarchy. A GPU card usually contains several Stream Multiprocessors (SMs) that can access data from the device memory (i.e. global memory) on GPU. Furthermore, in each SM, there are several CUDA cores called Stream Processors (SPs) that can share data through shared memory. Normally, a multithreaded program is divided into blocks of threads that execute independently between each other. Therefore, the GPU is able to achieve automatic scalability since a GPU with more SMs will automatically execute the program in less time than a GPU with fewer SMs.

1.2 General Fast Fourier Transform Heterogeneous Computing

1.2.1 Fast Fourier Transforms on GPGPUs

As a paradigm of heterogeneous parallel computing, 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 large-scale physics simulations, signal processing and data compression, spend majority of execution time on large size FFTs. Such FFT implementations require large amount of computing resources and memory bandwidth. Generating high performance Fast Fourier Transform library becomes an important research topic for the traditional processors, CPUs, and new accelerators, like General Purpose Graphics Processing Units (GPGPUs).

Compared with current multi-core CPUs, GPUs have been recently proved to be a more promising platform to solve FFT problems since GPUs have much more

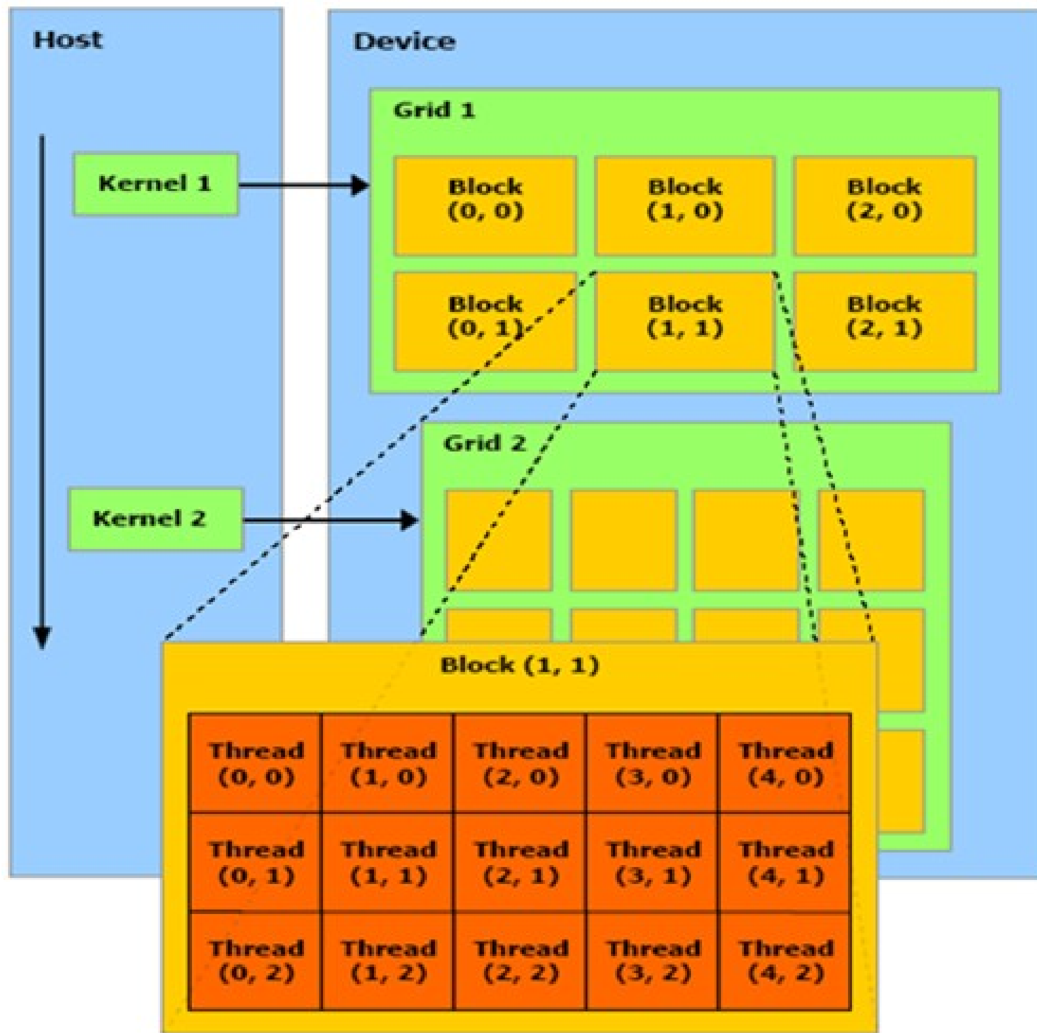


Figure 1.5: CUDA Heterogeneous Programming Mode (Image from Nvidia CUDA C programming guide [65]).

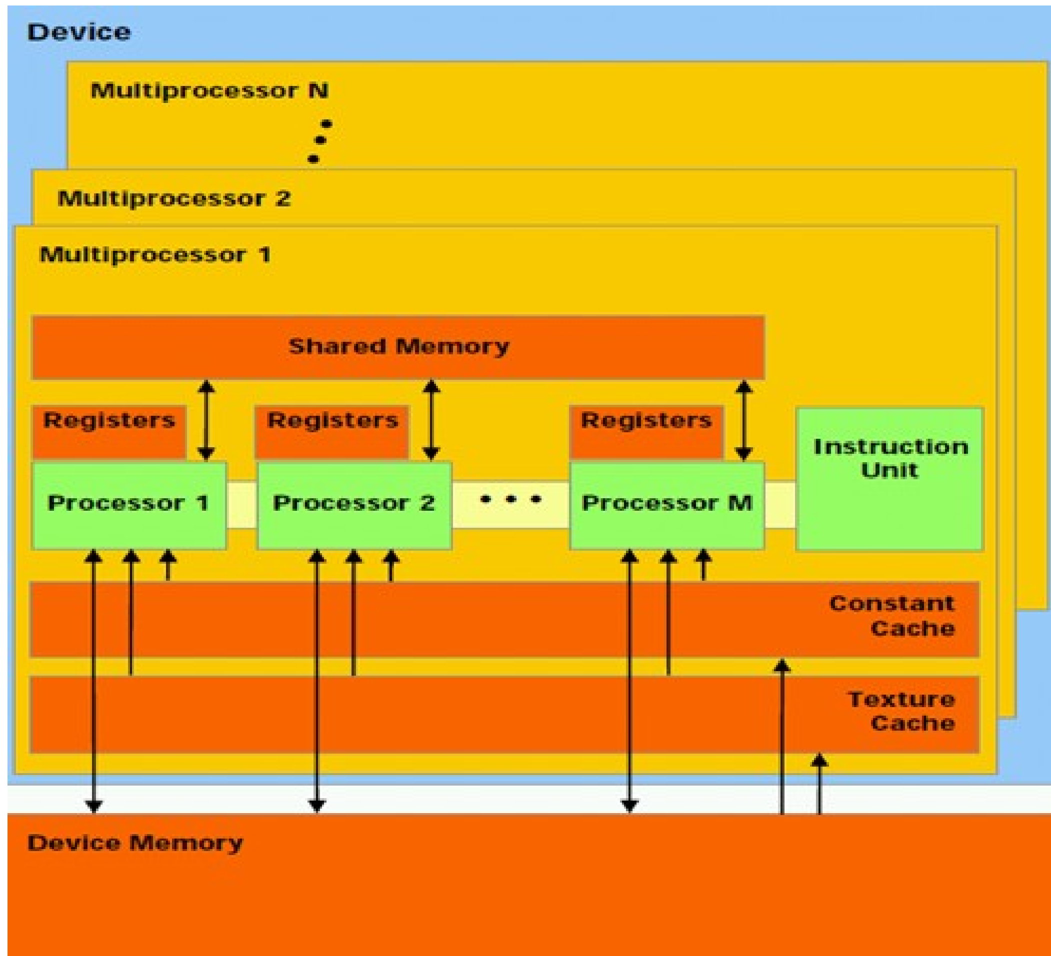


Figure 1.6: CUDA Hardware Model (Image from Nvidia CUDA C programming guide [65]).

parallel computing resources and can often achieve an order of magnitude performance improvement over CPUs on compute-intensive applications [5].

1.2.2 Problem Statement

Traditional FFT libraries have been built either on general purpose CPUs such as FFTW [17, 16, 15], SPIRAL [31, 32, 33] and Intels MKL [18, 50] or on compute-intensive GPUs, such as CUFFT [1, 57] from NVIDIA, Nukada's work on FFT [2, 3], Govindaraju's work [4, 60], Gu's FFT libraries [5, 6], Chen's FFT [7],

In spite of highly influential results in prior FFT work on GPUs, no matter the on-card FFT libraries, the out-of-card FFT libraries or the GPU-cluster based solutions, the real performance of GPUs is not significantly higher than that of current high-performance CPUs as expected, since GPU performance is severely restricted by the limited GPU memory size and the low bandwidth of data transfer between CPU and GPU through PCIe channel. In addition, in the prior FFT research on GPUs, CPU is only used as a memory or communication controller, that is, managing memory transfer requests between CPU and GPU, or between nodes. The computing power of CPUs is wasted. Furthermore, there is few that takes advantage of CPU to concurrently compute partial well-decomposed FFT with GPU. Although the existing GPU based FFT libraries 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.

1.2.3 Motivations

Therefore, the objective is to use a hybrid parallel FFT framework to concurrently execute both CPU and GPU for computing large-scale FFTs that exceed GPU memory. Incorporating CPU has several advantages: 1) Multi-core CPU is capable of computing partial work concurrently with GPU to release the pressure which is originally assigned to the GPU, and to make full use of available underlying computing

resources for performance improvement, 2) CPU helps increase data transfer bandwidth remarkably by enabling efficient utilization of PCIe bus and save much GPU memory resource since partial work is kept into local CPU memory to execute without being transferred to GPU, and 3) CPU has much larger memory size than that of GPU in general. Ogata et.al. [8] recently 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 the dissertation, 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. The key problem we solve is to engage heterogeneous computing resources and newly improved optimization strategies in the acceleration of such large FFTs.

Making FFT run concurrently on CPU and GPU comes with significant 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. 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. We cannot just optimize for CPU or just optimize for GPU. 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, and to determine the optimal load distribution among such heterogeneous resources.

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.

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-optimized and co-tuned for two different architectures.

1.3 Sparse Fast Fourier Transform High Performance Computing

1.3.1 Sparse Fast Fourier Transform Algorithms

Traditionally, the Fast Fourier Transform algorithm calculates the spectrum representation of time-domain input signals. If the input size is N, the FFT operates in $O(NlogN)$ steps. The performance of FFT algorithms is known to be determined only by input size, and not affected by the value of input. Therefore, prior FFT optimization efforts, for example the widely used FFT library, Fastest Fourier Transform in the West (FFTW), have been largely focused on improve the efficiency of FFT for various computer architectural features such as cache hierarchy, but have generally put aside the role of input characteristics in FFT performance.

So far the only feature of input value having been leveraged to improve FFT performance is input sparsity. In real world applications, input signals are frequently sparse, i.e., most of the Fourier coefficients of a signal are very small or equal to zero. If we *know* that an input is sparse, the computational complexity of FFT can be reduced.

1.3.2 Problem Statement

Sub-linear sparse Fourier algorithm was first proposed in [29], and since then, has been extensively studied in the literatures when applied to various fields [28, 21, 20, 22, 27, 19]. However, their runtimes have large exponents in the polynomials of k and $logN$, and their complex algorithmic structures restrict fast and parallel implementations.

A recent highly-influential work [24] presented an improved algorithm in the runtime of $O(k\sqrt{NlogN}logN)$ that makes it faster than FFT for the sparsity parameter k up to $O(\sqrt{N/logN})$. The follow-up work [25] proposed an algorithm with runtime $O(klogNlog(N/k))$ or even the optimal $O(klogN)$. Nevertheless, the algorithms still iterates over passes. The dependency exists between each two consecutive iterations. Hence it makes the algorithms hard to be parallelized. Additionally, just like the "dense" FFT algorithms and the earlier sparse FFT algorithms, the latest sparse FFT algorithms are oblivious to input characteristics, because input sparsity is assumed but not measured. Furthermore, the sparse FFT algorithms' design is fixed for all inputs of the same size. No part in the algorithms is adapted to other input characteristics.

1.3.3 Motivations

Here we make an interesting observation. We know that in many real-world FFT applications not only inputs are sparse, but at the same time adjacent inputs are similar. For example, in video compression, two consecutive video frames usually have almost identical sparse distribution in their spectrums, and differ only in the magnitudes of some spectrum coefficients. If the FFT on the prior input has been computed, i.e., its spectrum representation is known, and the current input has a similar sparse distribution to the prior input, can the similarity help computing the sparse FFT on the current input? To answer the question, we need to tell whether an input is similar to its predecessor, and how the knowledge about the predecessor's spectral representation can help.

Moreover, the study of sparse FFT algorithms is still immature and is under research. There is no fully parallelized and well performed version of sparse FFT. How

to break down the dependencies in the Fourier coefficients binning process, and how to exploit parallelism as much as possible to make full use of underlying parallel resource is always challenges in current sparse FFT implementations.

This dissertation answers the two main questions and propose a new high performance sublinear and parallel algorithm for sparse FFT.

1.4 Dissertation Contributions

The notable contribution of this dissertation is to research on two main high performance optimization strategies to the Fast Fourier Transform on heterogeneous computers.

1.4.1 Hybrid FFTs Optimization and Parallelization

As for the hybrid FFTs optimization and parallelization, it is the first time to come up with a hybrid implementation of FFT that concurrently executes both CPU and GPU in high performance heterogeneous computers to compute large FFTs that cannot fit into GPU memory. We have two versions of hybrid FFTs based on different optimization strategies.

First, we propose radix decomposition based hybrid FFTs on heterogeneous GPU-CPU computers. A hybrid 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 computational load concurrently. Particularly, our approach integrates radix-decomposition paradigms to tailor the extraction of computation and communication patterns for CPU and GPU, and to exploit much more hidden and implicit parallelism than other heterogeneous methods to better utilize parallel computing resources. In order to attain the best performance, work distribution between GPU and CPU is tuned to achieve an optimal ratio of loading. We developed several empirical profiling techniques for FFT problems with different characteristics on GPU and CPU, and

we develop effective heuristics to guide the entire balancing process. Our library also overlaps data transfers for increasing PCI bus bandwidth and equally importantly maintaining data and layout consistency between CPU and GPU. We evaluate the hybrid FFT from three aspects, i.e., optimal load distribution ratios, running time, and normalized Root-Mean-Square Error (RMSE). In particular, the library is compared with CPU based library FFTW and Intel MKL. On average, our FFT library of single precision on a heterogeneous computer comprised of a GeForce GTX480 GPU and an Intel Core i7 CPU is 5.8× and 4.1× faster than 1-thread SSE-enabled FFTW [17, 16, 15] library and Intel MKL [18] math library, respectively. The accuracy of single precision measured by normalized RMSE is in the range from 2.41×10^{-07} to 3.15×10^{-07}.

Second, we propose a co-optimized and well-tuned parallel hybrid GPU-CPUs FFT library for large FFT problems. In this work, a hybrid large-scale FFT partition framework is applied to divide total problem into sub-problems, to distribute sub-problems into multi-CPUs and GPU, and to parallelize the sub-problem in each side. In particular, a more flexible cooley-tukey decomposition method is applied to keep decomposing the workload to exploit parallelism as much as possible, until all threads of computing resources are fully utilized and busy working. Another important novel technique in this work is to propose a co-optimizer, which is to exploit substantial parallelism for GPU and CPUs, and to optimize the performance in each GPU and CPU based on its architectural features, such as processor throughput, memory bandwidth and I/O communication bandwidth. The co-optimizer is also utilized to enable concurrent GPU and CPUs execution, to optimize data sharing, and to provide an efficient synchronization mechanism for both GPU and CPUs to keep program consistency and to accelerate performance. Additionally, empirical performance modeling and tuning are proposed to estimate performance in each sub-step of entire program, and hence to predict and determine optimal load balancing between GPU and CPUs based on several model parameters. It replaces an exhaustive walk-through of the vast space of possible hybrid implementations of FFT on heterogeneous GPU-CPUs system with a guided empirical search. Furthermore, an effective heuristic is used to enable stream

based asynchronous executions, and to overlap communication with computation. We evaluate the performance in three high performance heterogeneous systems, including an Intel Core i7 CPU with three different Nvidia GPUs, i.e. GeForce GTX480, Tesla C2070, Tesla C2075. Overall, our hybrid FFT implementation outperforms several latest and widely used large-scale FFT implementations. For instance, our Tesla C2070 performance is 14.1× and 8.6× faster than 1-thread SSE-enabled FFTW library and Intel MKL math library, respectively. Our performance is 1.9× and 2.1× faster than 4-thread SSE-enabled FFTW and Intel MKL, with max speedups 4.6× and 2.8×, respectively. In addition, our performance has 1.3× and 3.4× speedup over Gu et.al.'s [6] and Ogata et.al.'s [8] work.

1.4.2 Sparse FFTs Optimization and Parallelization

As for the sparse FFTs optimization and parallelization, it is the first time to propose and design an input adaptive approach to improve the efficiency of existing sparse FFT algorithms. Moreover, our algorithm is the first well parallelized sparse FFT with high computation intensity, and parallelism can be exploited for heterogeneous computers to improve performance.

First, we propose an input-adaptive algorithm for high performance sparse fast fourier transform. We go over the existing traditional sparse FFT algorithms to explain the evolution from a general sparse FFT algorithm to the proposed input-adaptive sparse FFT algorithm. We then describe a general input adaptive sparse FFT algorithm which comprises of input permutation, filtering non-zero coefficients, subsampling FFT and recovery of locations and magnitudes. Subsequently, we discuss how to save the number of permutations and propose an alternatively optimized version for our sparse FFT algorithm to gain runtime improvement. Moreover, the general and the optimized versions are hybridized so that we're able to choose a specific version according to input characteristics. We also demonstrate our input adaptive approach is well applicable in real world application. Overall, our algorithm is much faster than other dense and sparse FFTs both in theory and implementation.

Second, we propose parallel input-adaptive sparse fast fourier transform for stream processing. We improve upon existing sequential sparse FFT algorithms by proposing a new parallel input-adaptive sparse FFT algorithm to break down the dependency in the recursive coefficient packaging and estimation of traditional sequential sparse FFT. It is the first time to propose a highly parallel version of sparse FFT and to exploit substantial parallelism from tradition algorithms. Particularly interesting is that, the input sparsity and the input similarity make it easier to parallelize FFT calculation. From a very high point of view, our sparse FFT algorithm applies the custom-designed sparse filters to disperse the sparse Fourier coefficients of inputs into separate bins directly in the spectrum domain. During the dispersion, the calculation on those bins are independent. Therefore it leads our sparse FFT to produce a determinatively correct output, and to be non-iterative with high arithmetic intensity as well. Substantial data parallelism is able to be exploited from our algorithm. In addition to showing how the performance of our implementation can be parallelized for GPU and multi-core CPU, we also show how our work can be well applicable, efficient and robust in the real world, such as image/video stream processing. Specifically, our input adaptive algorithm can automatically detect and take advantage of the similarity between adjacent continuous inputs to accelerate computation. When a discontinuity occurs, our discontinuity detection method can automatically detect the discontinuities inside the streams and resumes the continuous input adaptation very efficiently. Finally, we evaluate our parallel input-adaptive sparse FFT implementation on Intel Core i7 CPUs and three NVIDIA GPUs, i.e., NVIDIA GeForce GTX480, Tesla C2070 and Tesla C2075. Our parallel sparse FFT is much faster than previous FFTs both in theory and implementation. For instance of the input with size $N = 2^{24}$, our parallel implementation outperforms FFTW for k up to 2^{18}, which is an order of magnitude higher than prior sparse algorithms. Furthermore, our parallel input adaptive sparse FFT on Tesla C2075 GPU achieves up to 77.2× and 29.3× speedups over 1-thread and 4-thread FFTW, 10.7×, 6.4×, 5.2× speedups against sFFT 1.0 [24], sFFT 2.0 [24], CUFFT [1, 57], and 6.9× speedup over our sequential CPU performance, respectively.

1.5 Dissertation Organization

The organization of this dissertation is as follows.

Chapter 1 introduces the concepts of general parallel heterogeneous computing, and gives overview, problem statement and motivation of general FFT computing on heterogeneous computers, sparse FFT algorithms and computing, and provides contributions and organization of this dissertation.

Chapter 2 provides an brief introduction to the concepts and algorithms used in FFT optimization. A brief introduction to various techniques used in FFT optimization is provided.

In Chapter 3, radix decomposition based hybrid FFTs on heterogeneous GPU-CPU computers is proposed to be the first time utilizing a hybrid framework to compute multi-dimensional FFTs. This computational model leverages radix decomposition paradigms to generalize partitioning process, to tailor the extraction of computation and communication patterns for GPU and CPU, and to exploit much hidden parallelism to make better use of parallel computing resources. In order to obtain the best performance, an effective heuristics is applied to guide the entire balancing process. Our performance is faster than various high performance CPU based FFT libraries.

In Chapter 4, a co-optimized and well-tuned parallel hybrid GPU-CPUs FFT library is proposed for large problems. Similarly, a new hybrid partition framework is proposed to parallelize the work. A more flexible FFT decomposition method is applied to exploit much more parallelism, and a co-optimizer is come up with to further optimize the performance in each GPU and CPU, and to enable fully concurrent GPU-CPU execution. An empirical performance modeling and tuning is to predict and determine optimal load balancing. And, stream based asynchronous executions is to overlap communication with computation. Overall, our hybrid FFT implementation outperforms several latest and widely used large-scale FFT implementations.

In Chapter 5, an input-adaptive algorithm is proposed for high performance sparse fast fourier transform. We review the existing traditional sparse FFT algorithms to demonstrate our proposed input-adaptive sparse FFT algorithm. Our approach

comprises of input permutation, filtering non-zero coefficients, subsampling FFT and recovery of locations and magnitudes. Subsequently, we discuss how to optimize the general version of our algorithm, and how to hybridize to produce an optimal solution, so that we can choose a specific version according to input characteristics. Meanwhile, our input adaptive approach is well applicable in the real world. Overall, our algorithm outperforms than other dense and sparse FFTs both in theory and implementation.

In Chapter 6, we propose parallel input-adaptive sparse fast fourier transform for stream processing. We break down the dependency existing in traditional sequential sparse FFT such that a parallel input adaptive sparse FFT is the first time proposed to improve the performance. The input sparsity and similarity make it easier to parallelize FFT calculation. Moreover, our input adaptive algorithm can automatically detect and make use of the similarity between continuous inputs to accelerate computation. When a discontinuity occurs, our discontinuity detection method can automatically detect the discontinuities inside the streams and resumes the continuous input adaptation very efficiently. Finally, our parallel sparse FFT is much faster than previous FFTs both in sequential and parallel version.

Chapter 7 concludes the proposed approaches, methods and results introduced in this dissertation. In addition, by overviewing the current development trend of heterogeneous parallel computing and supercomputing, we discuss the future work based on our parallelization and optimization strategies.

Chapter 2

BACKGROUND

2.1 Overview of General Fast Fourier Transform Algorithms

The development of Fast Fourier Transform (FFT) algorithms [77, 78, 10, 79, 80, 81, 82, 83, 84, 9, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 101, 102] has experienced a long history. Generally, the FFT recursively decomposes a N-point Discrete Fourier Transform (DFT) into several smaller DFTs, and the divide-and-conquer approach reduces the operational complexity of a DFT from $O(N^2)$ into $O(NlogN)$ [9]. There are many FFT algorithms, or in other words, different ways to decompose DFT problems.

2.1.1 The General Radix-2 FFT Algorithm

This section is to introduce the radix-2 decimation-in-time (DIT) FFT algorithm [9]. If the size N of an input serial $x(n), n = 0, 1, ..., N-1$ is power of 2, e.g. $N = 2^r$, the Radix-2 DIT algorithm factors N as two integers, i.e. $N = 2 \cdot 2^{r-1}$ such that the N-points input is divided into two $N/2$ length sequences with respective even-numbered samples $f_1(i), i = 0, 1, ..., N/2-1$ and odd-numbered samples $f_2(i), i = 0, 1, ..., N/2-1$. If the sequences $f_1(i)$ and $f_2(i)$ are recursively split using the Radix-2 DIT, we obtain four subsequences among which two are even-numbered $f_{11}(j)$, $f_{12}(j)$, and two are odd-numbered $f_{21}(j)$, $f_{22}(j), j = 0, 1, ..., N/4-1$.

$$
\begin{aligned}
f_{11}(j) = f_1(2j) = x(4j); \quad f_{12}(j) = f_1(2j+1) = x(4j+2); \\
f_{21}(j) = f_2(2j) = x(4j+1); \quad f_{22}(j) = f_2(2j+1) = x(4j+3);
\end{aligned}
\tag{2.1}
$$

where $j = 0, 1, ..., N/4 - 1$. Therefore, the N-point DFT $T(k), k = 0, 1, ..., N-1$ can be computed using decimated sequences as follows,

$$\begin{aligned} T(k) &= \sum_{n=0}^{N-1} x(n) W_N^{nk} \\ &= [F_{11}(k) + W_{N/2}^k F_{12}(k)] + W_N^k [F_{21}(k) + W_{N/2}^k F_{22}(k)] \end{aligned} \tag{2.2}$$

where $k = 0, 1, ..., N-1$, twiddle factor $W_N^{nk} = e^{-j2\pi nk/N}$, and $F_{11}(k)$, $F_{12}(k)$, $F_{21}(k)$, $F_{22}(k)$ are the $N/4$-point DFTs in the first phase of radix-2 DIT for respective $f_{11}(j)$, $f_{12}(j)$, $f_{21}(j)$, $f_{22}(j)$. Note that $F_{11}(k+N/4) = F_{11}(k)$, $F_{12}(k+N/4) = F_{12}(k)$, $F_{21}(k+N/4) = F_{21}(k)$, $F_{22}(k+N/4) = F_{22}(k)$ due to the periodic property of the $N/4$-point DFTs, Moreover, the twiddle factor is symmetric to itself in two ways $W_{N/2}^{k+N/4} = -W_{N/2}^k$ and $W_N^{k+N/2} = -W_N^k$. Therefore, the equation 2.2 can be streamed into two computational phases to obtain the final N-point $T(k)$. We first combine the four $N/4$-point DFTs as following

$$\begin{aligned} G_1(k) = F_{11}(k) + W_{N/2}^k F_{12}(k); \quad G_1(k+N/4) = F_{12}(k) - W_{N/2}^k F_{12}(k); \\ G_2(k) = F_{21}(k) + W_{N/2}^k F_{22}(k); \quad G_2(k+N/4) = F_{22}(k) - W_{N/2}^k F_{22}(k); \end{aligned} \tag{2.3}$$

where $k = 0, 1, ..., N/4 - 1$. Then we introduce a second combination phase to merge $G_1(k)$ and $G_2(k)$ as follow into the final N-point $T(k)$. $T(k) = G_1(k) + W_N^k G_2(k)$ and $T(k+N/2) = G_1(k) - W_N^k G_2(k)$, where $k = 0, 1, ..., N/2 - 1$.

2.1.2 The General Cooley-Tukey FFT Algorithm

The general Cooley-Tukey FFT algorithm [10] is a kind of factorization FFT algorithm. Suppose we still have an input serial $x(n), n = 0, 1, ..., N-1$ of size N. Its DFT transform 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 (as row) and m in M dimension (as column), respectively. $x(l, m) = x(l \cdot M + m)$. We apply Cooley-Tukey

FFT decomposition for the original DFT

$$\text{Perform M DFTs of size L,} \quad A(p,m) = \sum_{l=0}^{L-1} x(l,m) W_L^{lp} \tag{2.4}$$

$$\text{Multiply by twiddle factors,} \quad B(p,m) = A(p,m) W_N^{pm} \tag{2.5}$$

$$\text{Perform L DFTs of size M,} \quad Y(p,q) = \sum_{m=0}^{M-1} B(p,m) W_M^{mq} \tag{2.6}$$

Therefore, $Y(k) = Y(p \cdot M + q)$. In essene, Cooley-Tukey introduces a decomposition approach that divides one dimensional computation into two. In addition, Cooley-Tukey is more general than radix-2 DIT since it is appropriate for a composite problem size, not just applicable to problem sizes that are strictly power of 2.

2.1.3 Other General FFT Algorithms

There are many other FFT algorithms [11] distinct from Radix and Cooley-Tukey methods. Prime-Factor (Good-Thomas) [23, 103, 104] 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, 105, 106, 107, 108] and Bluestein's algorithm [14, 109, 110, 111, 112] can factorize a prime-size DFT as convolution. Actually, each of the algorithms illustrated above has specific implementation.

2.1.4 I/O Tensor Representation

I/O tensor representation was introduced in FFTW [15] library, and was used to represent and implement FFT computations on computers. In my work, I extend the I/O tensor representation 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 $u = \{d(L, M, M, I, O), t^L_M d(M, 1, 1, O, O)\}$. Here u is an I/O tensor that represents a multidimensional FFT.

2.2 Prior Works of General Fast Fourier Transform Implementations on GPGPUs

Traditional FFT libraries have been built either on general purpose CPUs such as FFTW [17, 16, 15, 46, 47, 48, 49], Intels MKL [18, 50, 51, 52, 53, 54, 55, 56], SPIRAL [31, 32, 33, 34, 35, 36, 37, 38, 39] and related applications [40, 41, 42, 43, 44, 45], or on compute-intensive GPUs such as CUFFT [1, 57, 58, 67] from NVIDIA, however, there is few that takes advantage of CPU to concurrently compute partial well-decomposed FFT with GPU. Although the existing GPU based FFT libraries make use of both CPU and GPU, it only enables CPU to perform controlling GPU kernel executions and also data transfer between host and GPU.

Recently, efforts have been focused on solving in-card 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 focused 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, 59], Govindaraju's [4, 60, 61, 62] and Gu's work on 2D and 3D FFT libraries [5, 6, 63, 64] can be classified into this group.

Furthermore, Gu et.al. [6] demonstrated a GPU-based out-of-card FFT library that can solve FFT problems larger than 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, the work has been almost exclusively focused on the optimization of communication over inter-node channels.

2.3 Overview of Sparse Fast Fourier Transform Algorithms

2.3.1 Prior Works of Dense Fast Fourier Transform Algorithms

A naive discrete Fourier transform Y of a N-dimensional input series $x(n)$, $n = 0, 1, ..., N-1$ is computed as $Y(d) = \sum_{n=0}^{N-1} x(n) W_N^{nd}$, where $d = 0, 1, ..., N-1$ and N-th primitive root of unity $W_N = e^{-j2\pi/N}$. The general fast Fourier transform algorithms recursively decompose a N-dimensional DFT into several smaller DFTs [9], and reduce DFT's operational complexity from $O(N^2)$ into $O(NlogN)$. There are many FFT algorithms, or in other words, different ways to decompose DFT problems.

However until now, the general FFT algorithms that we have introduced are all *dense* FFT algorithms, which means their algorithmic time complexities are linear to the input signal size N.

2.3.2 Prior Studies of Sparse Fast Fourier Transform Algorithms

So far, the runtimes of all general FFT algorithms have been proved to be at least proportional to the size of input signal. However, if the output of a FFT is k-sparse, i.e., most of the Fourier coefficients of a signal are very small or equal to zero and only k coefficients are large, sparse Fourier transform is able to reduce the runtime to be only sublinear to the signal size N. Sublinear sparse Fourier algorithm was first proposed in [29], and since then, has been extensively studied in many application fields [28, 21, 20, 22, 27, 19]. All these sparse algorithms have runtimes faster than original FFT for sparse signals. However, their runtimes still have large exponents (larger than 3) in the polynomials of k and $logN$, and their complex algorithmic structures are hard to parallelize.

A highly influential work [24] presented an improved algorithm with the complexity of $O(k\sqrt{NlogN}logN)$ to make it faster than FFT for k up to $O(\sqrt{N/logN})$. The work in [25] followed up with an improved algorithm with runtime $O(klogNlog(N/k))$ or even the optimal $O(klogN)$. Basically, the algorithms permute input with random parameters in time domain to approximate expected permutation in spectral domain for binning the large coefficients. The probability has to be bounded to prevent large coefficients being binned into the same bucket. In addition, these algorithms iterate over passes for estimating coefficients, updating the signal and recursing on the reminder. Because dependency exists between consecutive iterations, the algorithms cannot be fully parallelized. Moreover, the selections of the permutation probability and the filter, which are crucial to the algorithms' performance, are predetermined and are oblivious to input characteristics.

2.4 Objectives of Approaches

2.4.1 Objective of Hybrid FFT Approach

The hybrid FFT research in this dissertation aims to make a hybrid implementation of FFT that concurrently executes both multithreaded CPU and GPU in a heterogeneous computer node to compute large FFT problems that cannot fit into GPU memory. The objective is to make following contributions: a hybrid large-scale FFT decomposition framework that can cooperate two FFT algorithms to extract different types of computation and communication patterns for the two different processor types; a load balancer that can assign workloads to both GPU and CPU, and determine the optimal load balancing via an effective performance modeling, which replaces an exhaustive walk-through of the vast space of possible hybrid implementations of FFT on CPU/GPU with a guided empirical search; a performance optimizer that can exploit substantial parallelism for both GPU and CPUs; and an effective heuristics which can purposefully expose opportunities of overlapping communication with computation in the process of decomposing FFT. Overall, our hybrid FFT implementation is expected to outperform other latest and widely used large-scale FFT implementations.

2.4.2 Objective of Sparse FFT Approach

For the sparse FFT research, the objective of this dissertation is to make the exploitation of the similarity between sparse input samples in stream processing to improve the efficiency of sparse FFT. Specifically, our work is expected to develop an effective heuristic to detect input similarity, and dynamically customizes the algorithm design to achieve better performance. Moreover, our algorithm is expected to be non-iterative with high computational intensity such that parallelism can be exploited for multi-CPUs and GPU to improve performance. Overall, our algorithm aims to be faster than other dense FFTs both in theory and implementation.

Chapter 3

RADIX DECOMPOSITION BASED HYBRID FFTS ON HETEROGENEOUS GPU-CPU COMPUTERS

In this chapter, we propose a radix decomposition based hybrid FFT on heterogeneous GPU-CPU computers. A hybrid 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 computational load concurrently. Particularly, our approach integrates radix-decomposition paradigms to exploit hidden and implicit parallelism to utilize parallel computing resources, and to tailor the extraction of computation and communication patterns for GPU and CPU. In order to gain the best performance, work distribution between GPU and CPU is tuned to achieve an optimal ratio of loading. We develop effective heuristics to guide the entire load balancing process. Our library also carefully manipulates data transfers for increasing PCI bus bandwidth and equally importantly maintaining consistency between GPU and CPU.

3.1 Overview of Our Approach

We use 2D FFT as an example to illustrate how our hybrid approach differs from a naive approach. The whole process of a naive 2D FFT with size $N = Y \times X$ is a two round row-column based computation where Y is the size of rows and X is size of columns. The naive approach firstly compute the Y dimensional 1D FFT for all columns along the X dimension of the 2D input array $f(y, x)$, and then compute the X dimensional 1D FFT for all the rows along Y dimension to obtain output $out(k_y, k_x) =$

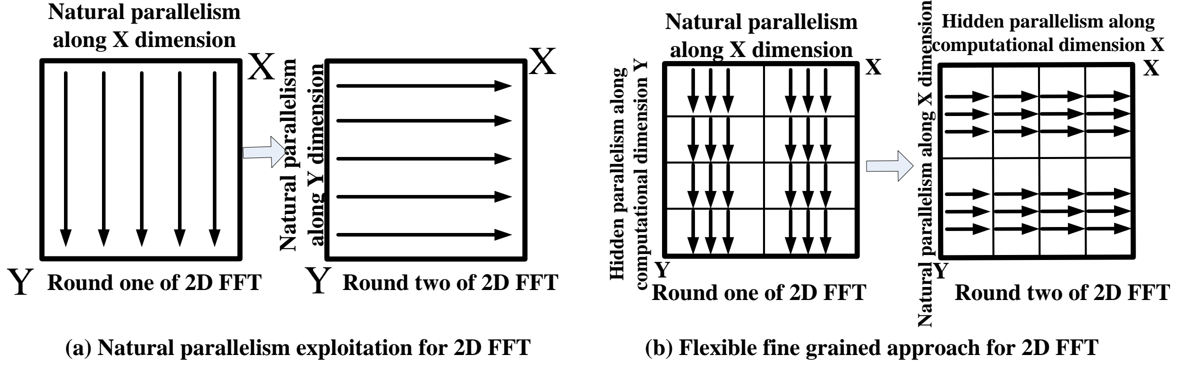


Figure 3.1: Natural Parallelism Exploitation of a Naive 2D FFT and Our Flexible Fine Grained Approach.

$\sum_{x=0}^{X-1} W_X^{xk_x}[\sum_{y=0}^{Y-1} W_Y^{yk_y} f(y,x)]$. where $x, k_x = 0, 1, ..., X-1$; $y, k_y = 0, 1, ..., Y-1$; twiddle factor $W_c^{ab} = e^{-j2\pi ab/c}$.

Apparently for the implementation of 2D FFT, natural parallelism can be explored along the columns of X dimension for each Y dimensional 1D FFT in the first round, and can be executed along Y dimension for each X dimensional 1D FFT in the second round. Figure 3.1(a) shows the parallelism exploitation of a naive 2D FFT. However, this approach can produce only coarse grained parallelism, i.e. parallelism can be found only on X dimension. The limitation is reflected in that if 2D problem size is large, concurrent 1D FFT subtasks are also large of code size and execution time, and vice versa. Therefore, in addition to the natural parallelism, which is the parallelism on the non-computational dimension in each computational round, we also want to decompose the computational dimension to exploit the hidden and implicit parallelism for each 1D FFT subtask. Although there are many prior efforts of parallelizing 2D FFT, there is few literature discussing the flexible methodology of mixing domain partition and parallelism exploitation, as it is not needed in a computing methodology setup. For the example shown in the figure 3.1(a), just as in the first round, some approaches only consider the parallelism along the X dimension, but neglect the essential hidden parallelism existing in the computing along the dimension Y. Figure 3.1(b)

highlights our approach that is proposed to exploit more parallelism on Y dimension in the first round and more parallelism on X dimension in the second round.

The key challenge in the design of a hybrid FFT decomposition is how to avoid the introduction of unnecessary complexity and redundancy during the splitting of workload. We propose to combine the Radix-2 DIT approach with the Cooley-Tukey method. The Radix-2 DIT decomposition is able to divide the computational dimension with respective even-numbered and odd-numbered sections, such that each split section can be implemented in parallel with other sections at a time, while operational complexity is reduced as well. From a high-level point of view, the hybrid decomposition works well for heterogeneous systems because CPU or GPU is capable of running its own portion of computation simultaneously without waiting for the intermediate results from one another.

Another notable contribution 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. Meanwhile, in order to attain the best parallel performance, the best balance between load overhead needs to be found. We come up with a new method to search for the optimal CPU-GPU load distributions for different problem sizes and different hegerogenities.

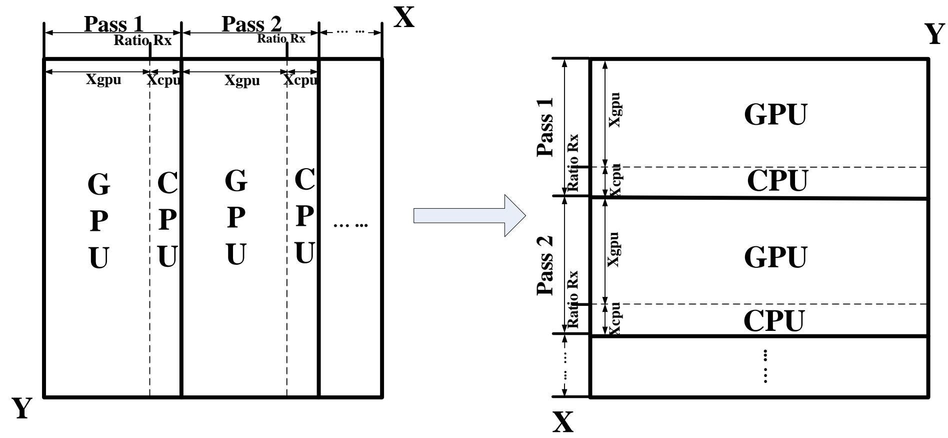
3.2 Radix Decomposition based Hybrid FFTs on Heterogeneous Computers

The hybrid FFT framework has three components: the decomposition of computation, the memory layouts at each step, and the integration of partial results from CPU and GPU to form the final FFT output. We first describe how to efficiently combine the two FFT algorithms for heterogeneous implementation. Secondly, we discuss in detail the memory layout in each computation phase, as memory layout determines the data transfers from previous steps and to next steps, and is crucial for the overall performance. Finally, we introduce how to coordinate the intermediate and final results from GPU and CPU to form the intermediate results for the next step or the final results of the whole FFT problem. For each component, we formulate the design choices as a dependent function based on architecture features, computational parameters and input configurations, such as input size, GPU and CPU load distributions and parallel subtask divisions.

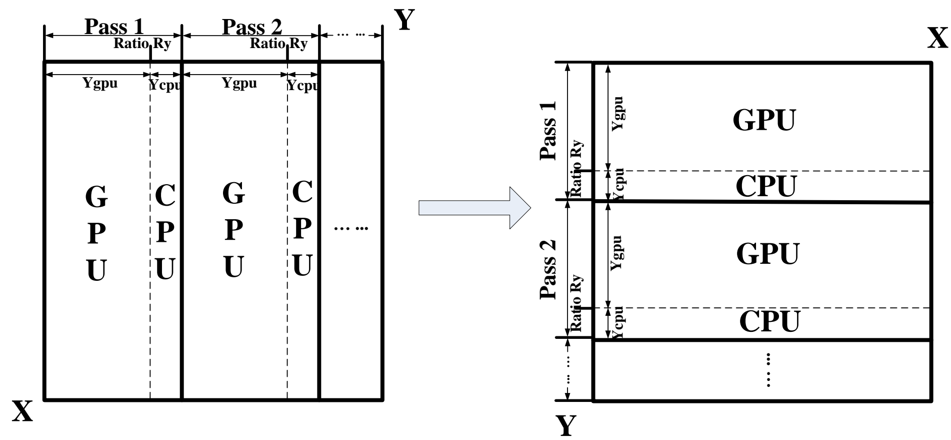
3.2.1 Radix Decomposition based Hybrid 2D FFT Framework

Our heterogeneous 2D FFT framework is assumed to solve a FFT problem with the total size $N = Y \times X$, where Y is the size of rows and X is size of columns. In 2D FFT, there are 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. Since we assume the large out-of-card 2D FFT exceeds GPU global memory, therefore, we divide the 2D problem of each round into several passes such that the subproblem of each pass assigned to GPU can fit into GPU memory and be executed with the CPU portions concurrently. Figure 3.2 shows the process of large out-of-card 2D FFT.

In each pass of the two rounds, computation load is distributed to GPU and CPU along X dimension. The work ratio of GPU to CPU for every pass is $R_X = \frac{Xgpu}{Xcpu}$ in round one and $R_Y = \frac{Ygpu}{Ycpu}$ in round two, where $Xgpu$ and $Xcpu$ are defined as the X dimensional size for GPU and CPU in the first round while $Ygpu$ and $Ycpu$



(a) Computation of the first round of 2D FFT



(b) Computation of the second round of 2D FFT

Figure 3.2: Computation of Our Out-of-card 2D FFT.

are the Y dimensional size for them in 2nd-round. The number of passes equals to $\frac{\text{X*Y*\# of bytes per element}}{\text{GPU memory in bytes}}$.

The second round of 2D FFT recursively performs the same process as the 1st-round, since the required matrix transposition can be built into the computation of the final phase of the first round. Therefore for the simplicity of description, we only use the first round computation as the example for illustration. First let us start with the FFT within GPU memory in each pass of the first round to show the technique of our hybrid approach. Such in-card FFT has in total three phases derived from the radix-2 DIT. In the first phase, instead of simply splitting the computational dimension Y into even-numbered and odd-numbered sections, we further partition those two sections once into four subtasks f_{ij} based on equation 2.1. The splitting process of the first phase in round one for respective GPU and CPU is shown as follows.

$$\begin{aligned}
&\sum_{y=0}^{Y-1} W_Y^{yk_y} f(y, 0 \leq x_g < Xgpu) \\
&= [F_{11}(k_y, x_g) + W_{Y/2}^{k_y} F_{12}(k_y, x_g)] + W_Y^{k_y} [F_{21}(k_y, x_g) + W_{Y/2}^{k_y} F_{22}(k_y, x_g)] \\
&\sum_{y=0}^{Y-1} W_Y^{yk_y} f(y, Xgpu \leq x_c < X_i) \\
&= [F_{11}(k_y, x_c) + W_{Y/2}^{k_y} F_{12}(k_y, x_c)] + W_Y^{k_y} [F_{21}(k_y, x_c) + W_{Y/2}^{k_y} F_{22}(k_y, x_c)]
\end{aligned} \tag{3.1}$$

We distribute all the four parallel partitions for four GPU concurrent kernels or four CPU multithreads to execute based on the work ratio defined before. It is worth mentioning that the partitioning is done different for CPU and GPU. Fine granularity parallelism on CPU requires individual tasks are relatively small in terms of code size, therefore, the four multithreads on CPU are scheduled to perform data parallelism instead of task parallelism. That means four threads execute simultaneously for each partitioned section along computational dimension, however, run across different partitions sequentially. Moreover, on the GPU side, each kernel supports for the execution of natural parallelism of 2D FFT in X dimension. Concurrency is achieved between GPU and CPU as CPU uses four threads for executing its own work while uses an additional thread for GPU kernels controlling. In addition, considering the remaining

large size FFT for each split section after applying the radix-2 method, we further use the Cooley-Tukey theorem to decompose large size FFT of each partitioned section into smaller subproblems. The complexity is therefore reduced while more parallelism along the computational dimension Y is exposed for better concurrency between GPU and CPU. Specifically, as the computational Y dimension is divided into four parallel subtasks, the size of each subtask v is $Y_v = Y/4 = Y_r \times Y_m$. We can then decompose the original 2D FFT into 3D form with size $X_i \times Y_m \times Y_r$. The decomposition for each split subtask F_{ij} is denoted as $F_{ij}(k_y, x) = \sum_{y=0}^{Y/4-1} f_{ij}(y = lY_m + m, x)$. As for the decomposition,

$$\begin{aligned} A(p, m, x) &= \sum_{l=0}^{Y_r-1} f_{ij}(l, m, x) W_{Y_r}^{lp}; \quad B(p, m, x) = A(p, m, x) W_{Y_r Y_m}^{pm}; \\ F_{ij}(p, q, x) &= \sum_{m=0}^{Y_m-1} B(p, m, x) W_{Y_m}^{mq}; \quad F_{ij}(k_y, x) = F_{ij}(pY_m + q, x); \end{aligned} \tag{3.2}$$

where variable x can be expressed as either X dimensional variable x_g in GPU side or x_c in CPU side. We first compute all the Y_r dimensional 1D FFTs for $Y_m \times X_i$ times across other two dimensions X and Y_m, then after multiplying corresponding twiddle factors we compute all the Y_m dimensional 1D FFTs for $Y_r \times X_i$ times across other two dimensions X and Y_r. All the Y_r and Y_m dimensional FFTs on both GPU and CPU are computed using revised 1D FFT codelets, which are compiler-generated straightline code to compute small size FFTs. The codelets perform well mainly because they are straightline, and therefore can be more effectively optimized. We revise the codelets by changing the type and precision of inputs, outputs, trigonometric functions to make them fit into our heterogeneous program. We also find that computing twiddle factors using the trigonometric functions on the fly is faster than reading precalculated values from memory. X_i

Figure 3.3 shows the work partition and distribution by illustrating the first phase of our hybrid FFT optimizations for heterogeneous GPU and CPU in 1st-round 2D FFT, where X_i is the size along X dimension for each pass i which represents one iteration of in-card FFT if FFT size is larger than GPU global memory size. In

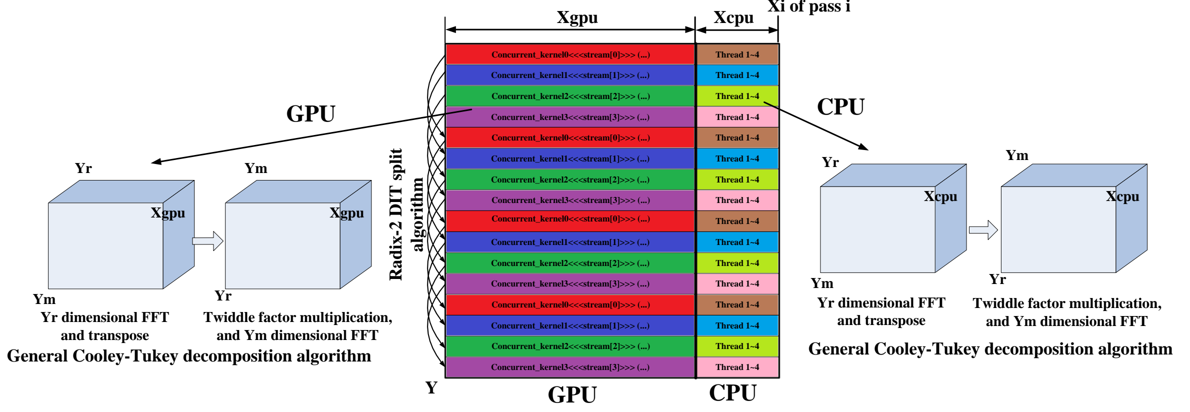


Figure 3.3: First Phase of the 1st-round of Our Hybrid 2D FFT Optimizations on Heterogeneous GPU and CPU.

figure 3.3, each chunk of one color represents a row of data belonging to GPU or CPU in pass i of the first-round 2D FFT. The group of the same color of rows denotes a partitioned subtask based on radix-2 DIT split algorithm. The partitioning along the computational Y dimension is performed on both GPU and CPU concurrently. Each subtask along Y dimension is further decomposed as Yr dimensional FFTs and Ym dimensional FFT based on the Cooley-Tukey theorem. Load distribution, therefore, can be clearly represented as $Y \times Xgpu$ for GPU and $Y \times Xcpu$ for CPU. For GPU, four concurrent kernels support for the four split parallel subtasks along Y dimension, while each kernel operates the natural parallelism in X dimension. Concurrent kernels are managed by CUDA streams. Different streams overlap their task executions with respect to one another. Each stream number in the figure 3.3 represents the execution of one concurrent kernel. On CPU, additional data parallelism is gained by further dividing each subtask into four smaller piece of code sections with equal size to each other. Four threads are created for the four parts of each subtask, while different subtasks are executed in sequential order.

Figure 3.4(a) shows the data layout residing in GPU global memory and CPU memory after the first phase of our hybrid FFT optimizations in the first round 2D FFT. For initial memory allocation, CPU only uses the original input memory, no

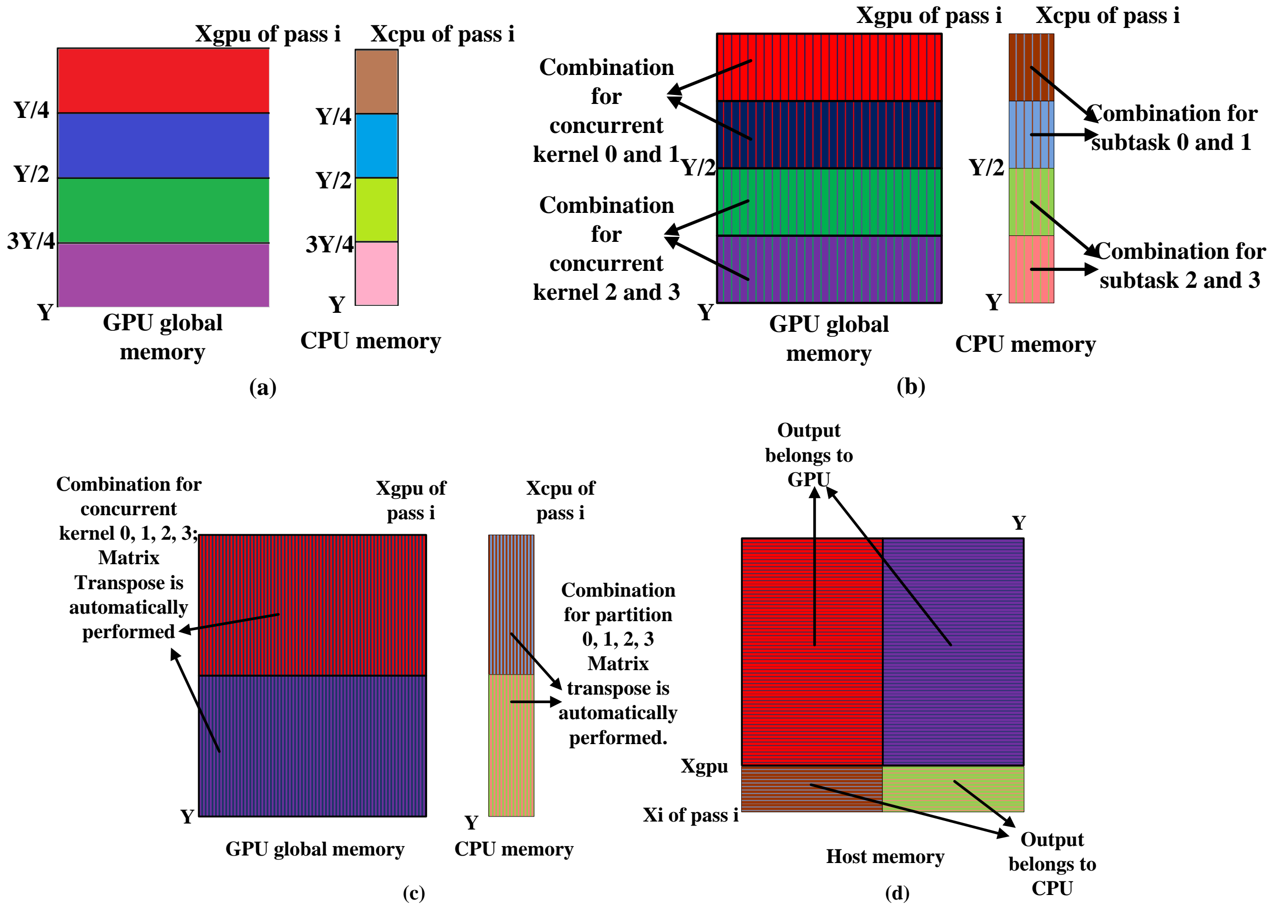


Figure 3.4: Result Combination in Each Sub-stage: (a) The results reside in GPU global memory and CPU memory after the first phase of our hybrid FFT optimizations in 1st-round 2D FFT; (b) Results of the second phase of our hybrid FFT; (c) Results of the third phase; (d) Results of GPU side and CPU side are combined into host memory to produce the total output of 1st-round 2D FFT.

other spare memory allocated at first. CPU reads the input data from the host memory directly and only one additional memory with small size $Y \times Xcpu$ is allocated for saving the temporary results after the first phase, including the output of Y_r dimensional FFT and the next in-place Y_m dimensional FFT of Cooley-Tukey decomposed approach. Each color represents the results of corresponding colored subtask in figure 3.3.

Based on equation 2.3 of the radix-2 method, the second phase needs to combine the results of all sections in the first phase of radix-2. This work is distributed to both GPU and CPU. The GPU part issues two kernels. One of the kernels combines the results of kernel 0 and 1 in first phase of radix-2, the other kernel makes combination for kernel 2 and 3 of the first phase. The CPU part uses four multithreads to firstly combine subtask 0 and 1 in the radix-2 first phase, and next all four threads combine the subtask 2 and 3. A buffer of size $Y \times Xcpu$ is allocated in CPU memory for storing the temporary results of the second phase. Figure 3.4(b) shows the results residing in GPU global memory and CPU memory after the second phase.

Furthermore, the third phase implements the final phase of radix-2 method. The outputs for kernel 0, 1, 2 and 3 on GPU are combined and matrix transpose is automatically performed to prepare for the second round of 2D FFT. Similarly, combination of subtask 0, 1, 2 and 3 on 4, as well as the matrix transposition, are done by four threads in CPU. Figure 3.4(c) shows the results residing in GPU global memory and CPU memory in the final phase.

Finally, all the data of the output in GPU memory is transferred back into the left region of host memory. The intermediate output of CPU is stored into the right side of host memory automatically without additional memory allocations and memcpy() calls. Therefore, data in GPU side and CPU side are combined into host memory to produce the total output of 1st-round 2D FFT. This work is shown in figure 3.4(d).

The second round of 2D FFT recursively can operate the same process as in the first round because as noted previously, matrix is automatically transposed at the end of the previous round.

3.2.2 Radix Decomposition based Hybrid 3D FFT Framework

In general the 3D hybrid FFT includes three rounds of computation. Each round computes 1D FFT along one dimension across the other two dimensions. Suppose the 3D input is $f(z, y, x)$, the output $out(k_z, k_y, k_x)$ is obtained from the formula $out(k_z, k_y, k_x) = \sum_{x=0}^{X-1} W_X^{xk_x} \sum_{y=0}^{Y-1} W_Y^{yk_y} [\sum_{z=0}^{Z-1} W_Z^{zk_z} f(z, y, x)]$. where $x, k_x = 0, 1, ..., X-1$; $y, k_y = 0, 1, ..., Y-1$; $z, k_z = 0, 1, ..., Z-1$;

The hybrid 3D FFT is visualized in figure 3.5. Z dimensional FFT is computed in the first round, X dimensional FFT in the second round and finally Y dimensional FFT in the third round. Each round of the 3D problem is divided into several passes where each pass is run by GPU and CPU concurrently.

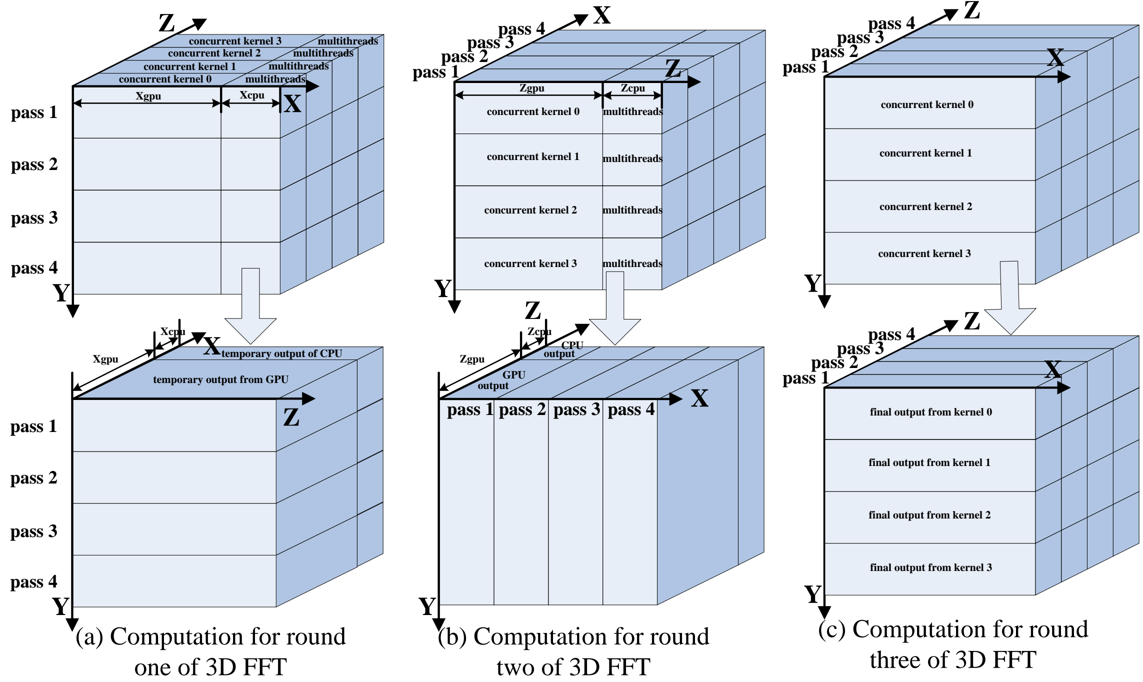


Figure 3.5: Computation of Our Out-of-card 3D FFT with 4 Passes in Each Round.

The division of passes for the Z dimensional FFT is along the Y dimension, where the number of passes equals to $\frac{\text{Z * Y * X * \# of bytes per element}}{\text{GPU memory in bytes}}$. A pass of computing Z dimensional FFT, on the other hand, distributes the computation load to GPU and CPU along X dimension. The work ratio of GPU to CPU is $R_Z = \frac{X_{gpu}}{X_{cpu}}$.

Here we suppose Z dimensional size Z is larger than the product of the other two dimensions' size which is $X \times Y$. Therefore, following the same computation model as 2D hybrid FFT, Z dimension is firstly split into 4 parallel subtasks based on radix-2 method. Furthermore, the Cooley-Tukey decomposition for Z is applied to extract different types of parallelism for GPU and CPU. Next, the division of passes for Y dimensional FFT is along X dimension with the same number of passes. In the Y dimensional round, concurrency partitioning is along Z dimension with the load ratio of GPU to CPU being $R_Y = \frac{Z_{gpu}}{Z_{cpu}}$. We still utilize radix-2 to split Y dimension into 4 concurrent subtasks run by GPU and CPU. Since the size of Y and X dimensional FFT is not too large to be further decomposed, we do not impose Cooley-Tukey theorem into the computation of these two rounds . As for the final X dimensional FFT, passes are still divided along Z dimension with the same number of divisions. Since after the second round, memory layout is continuous for the data needed for the third round, the final X dimensional FFT on GPU is computed by using CUFFT library for good performance.

3.3 PCI Data Transfer Scheme and Load Balance

Next we discuss the optimization of the PCI transfer of partial results between CPU and GPU for FFT problems whose input and output size is larger than GPU global memory. Furthermore, we develop an empirical tuning approach to find out the optimal CPU-GPU load distribution ratios for different problem sizes.

3.3.1 Data Transfer Scheme Through PCI Bus

When the size of FFT for GPU parts exceeds GPU global memory, we need to divide the original FFT input array into several smaller portions to fit into GPU memory. Suppose we have a total 2D FFT problem size equal to $N = Y \times X$ and maximum GPU memory size is $Y \times X_i$, where $X_i < X$. We need to divide total input array into several passes. The pass i transfers a subarray of size $Y \times X_i$ to fill GPU memory. Generally, in order to transfer total data of size $Y \times X_i$ into GPU, we need

use at lease Y times PCI transfers with copying a row of data of size X_i at a time. Therefore, our baseline PCI transfer scheme firstly puts the data required by GPU into a temporary memory buffer on CPU to make data layout consistent with that in GPU memory, then makes only one PCI data transfer for copying the data between CPU and GPU. Although this approach introduces a redundant step for copying original data into a temporary buffer, however, it works more effectively overall. Based on the baseline buffer technique, we improve the PCI bandwidth by reducing number of PCI bus usage while guaranteeing the consistency of input when copied into GPU. However, we also consider the tradeoff between reduced number of PCI transfers versus the entire process of data transfer. We note that there is an exploitable concurrency between the copying of the next row of data into a buffer and the copying of current row of buffers data into GPU. To decrease the number of PCI transfer, we propose a scheme that put one current block of data from original input array into a temporary buffer on CPU, then overlapping the PCI copy of current block into GPU with the copy of next block into the temporary memory. Two threads are used to performance the scheme. When the copy of current block into temporary memory finishes in thread 0, a synchronization signal is sent to another thread 1 to make it start to do PCI transfer of current block into GPU, while at the same time thread 0 start preparing the next block for the next transfer. Figure 3.6 shows the working process of our proposed data transfer scheme. Moreover, we use stream copying, i.e., cudaMemcpyAsync(stream[i]) for the PCI transfers. The number of data blocks equals to that of streams which represents the concurrency between data transfers and kernel executions on GPU. Each block is set to the same size as one split subtask of Radix-2 method mentioned in the preceding section.

In figure 3.6, each chunk of one color in host memory represents a row of data with size $Y \times Xgpu$ in pass i of the first round of 2D FFT. Each block of data with same color represents a split subtask after Radix-2 division. Every data block with same color is copied into a temporary buffer on CPU side and overlaps the previous transfer of a data block into GPU memory through PCI bus. Overlapping is also

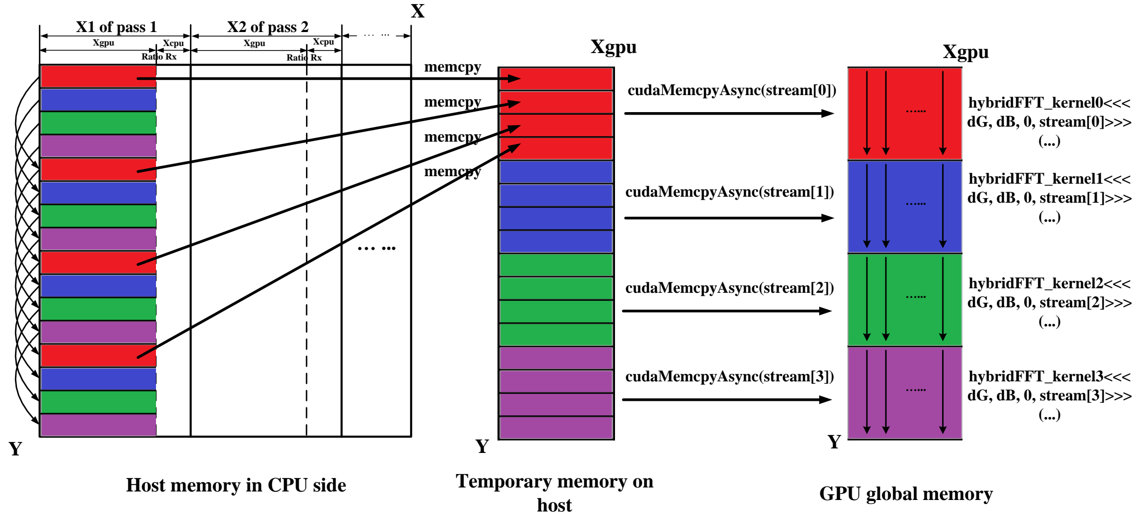


Figure 3.6: Concurrent Data Transfer Scheme from CPU to GPU Through PCIe Bus.

performed between the PCI transfer of the last block and the execution of the first GPU kernel. Particularly, to improve GPU performance across smaller memory access stride by hiding the global memory latency, we regroup the four rows of data in one split subtask into a continuous memory when copying them to buffers on CPU, then we do PCI transfer to copy the reallocated data into device.

Figure 3.7 shows the memory copy from GPU back to CPU through PCI bus. The CPU threads controlling GPU kernel launch is used to perform cudaMemcpyAsync() to transfer data blocks from GPU to CPU. Therefore, concurrent data transfers from GPU to CPU are achieved, while the kernel launch of last stream 3 also overlaps data transfer of first stream 0 from GPU to CPU. Since there is no layout consistency issue from device to host, there is no need to introduce a buffer on CPU. GPU copies data blocks directly into the host memory of total size $Y \times X$. Meanwhile, partial results in CPU side are automatically stored into the same host memory without additional memcpy().

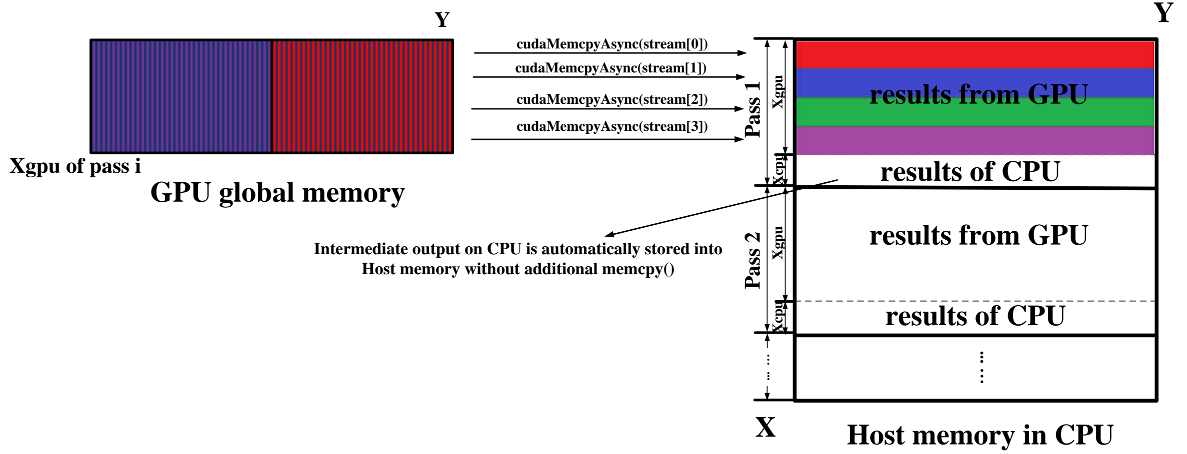


Figure 3.7: Concurrent Data Transfer Scheme from GPU to CPU Through PCIe Bus.

3.3.2 Load Balancing Between GPU and CPUs

To effectively utilize both CPU and GPU, we need to balance the distribution of work between them. Using the hybrid 2D FFT as an example, there are in total eight sub-steps for each round of 2D FFT. For each round, we need to determine a potentially different optimal distribution ratio because different rounds have different computation and communication patterns.

The general execution flow of our 2D FFT approach is summarized in figure 3.8. Our approach is to derive a performance model for each substep in the execution. The models might not be very accurate, but can be complemented with empirical tuning to guide the search for the optimal distribution ratio. Under the optimal ratio, load balancing between GPU and CPU is achieved and best performance can be modeled in the following equations. Table 3.1 lists all parameters that contribute to the performance modeling.

In GPU side, for each pass p in the 1st-round 2D FFT, the execution time is estimated as $T_G^p = \sum_{i=0}^{2} Th2db_i + Tlap_{in} + \sum_{i=1}^{3} Tg_i + Tg_{01} + Tg_{23} + Tg_{0123} + Tlap_{out} + \sum_{i=1}^{3} Td2hb_i$. For all the passes in the round h of 2D FFT, the execution time of GPU is estimated as $T_G(h) = \sum_{p=0}^{\#passes-1} T_G^p$, where $h = 1, 2$; In CPU side, the first round is different. For each pass p in the first round 2D FFT, the execution

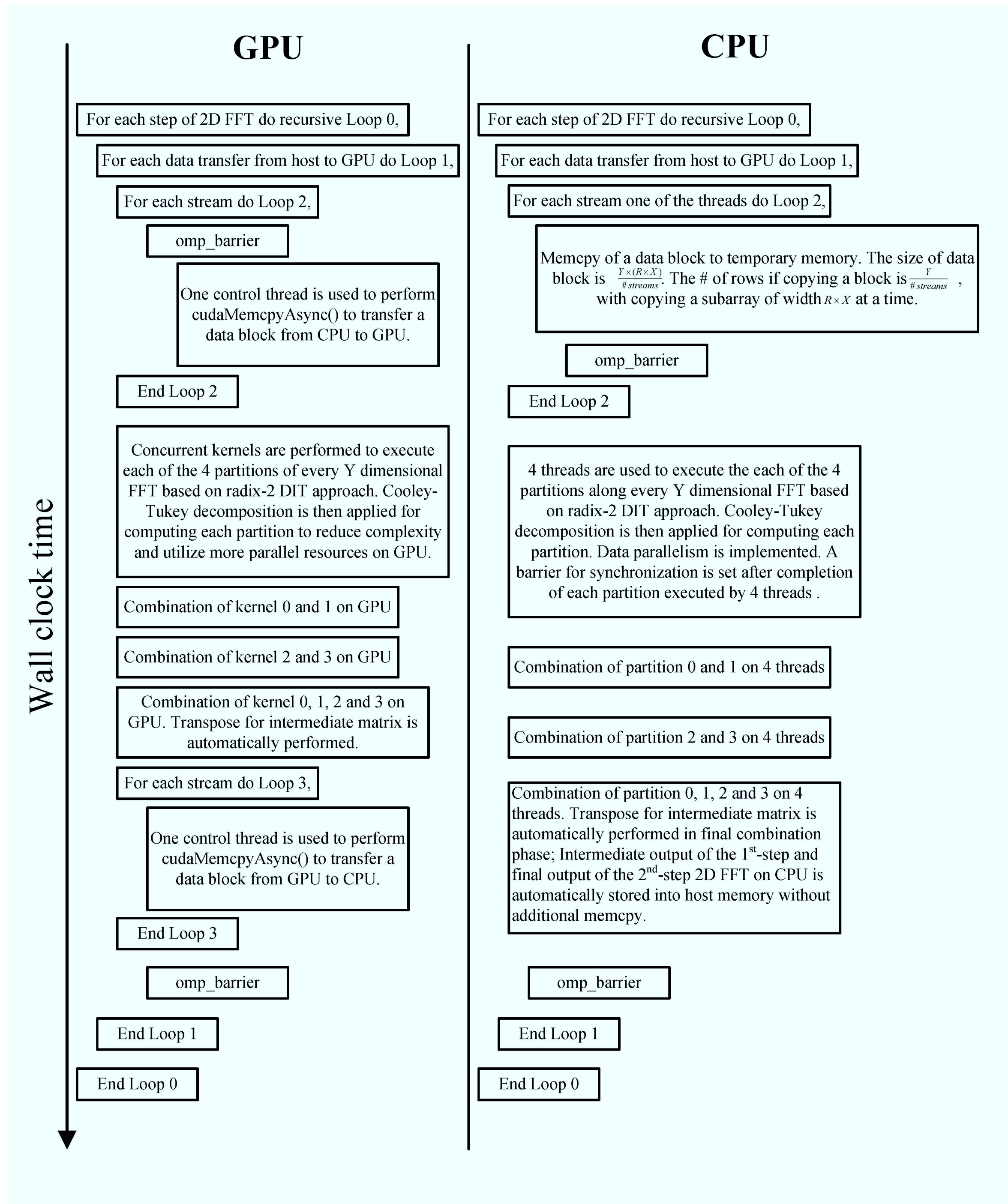


Figure 3.8: Concurrent Execution Flow of Our 2D FFT.

Table 3.1: Parameters for Running Time Estimation.

Parameters	**Description**
# of streams	# of data blocks = # of subtask of radix-2 such that each stream provides the required block of data for each subtask
$Trow$	Time for copying a row of a subarray into temporary memory
Tb_i	Time for copying each data block i into temporary memory which equals to $\frac{Y}{\#streams} \times Trow$
$Th2db_i$	Time for copying each data block i into GPU global memory through PCI bus
$Tlap_{in}$	Time for the PCI transfer of the last block into GPU overlapping with execution kernel 0 on GPU
Tg_i	Time for execution of concurrent kernel i on GPU
Tg_{ij}	Time for combination of kernel i and j on GPU
Tg_{ijkl}	Time for combination of kernel i, j, k, l on GPU and automatic matrix transpose
Tc_i	Time for computing each subtask i of radix-2 by 4 threads on CPU
Tc_{ij}	Time for combination of subtask i and j by 4 threads on CPU
Tc_{ijkl}	Time for combination of kernel i, j, k, l on CPU and automatic matrix transpose
$Tlap_{out}$	Time for the last kernel issued on GPU overlapping with transfer of data block 0 into CPU
$Td2hb_i$	Time for concurrent transfer of data block i from GPU to CPU

Table 3.2: Configurations of GPU, CPU, FFTW, MKL and CUFFT.

GPU	**Global Memory**	**FFTW**	**Nvcc & Cufft**	**PCI**
GeForce GTX480	1.5GB	3.3.2	3.2	PCIe2.0x16
CPU	**Frequency**	**Cores**	**System Memory**	**GCC & MKL**
Intel i7 920	2.66GHz	4	12GB	4.4.3 & 10.3

time is estimated as $T_C^p = \sum_{i=0}^{3} Tb_i + \sum_{i=0}^{3} Tc_i + Tc_{01} + Tc_{23} + Tc_{0123}$; For all the passes in the round h of 2D FFT, the execution time of CPU is estimated as $T_C(h) = \sum_{p=0}^{\#passes-1} T_C^p$, where $h = 1, 2$; Since two synchronizations are set after first and second rounds to guarantee the correctness, the total execution time can be modeled as the maximum of the GPU time and CPU time. The estimation is shown as $T_{\text{total}} = max\{T_G(1), T_C(1)\} + max\{T_G(2), T_C(2)\}$.

3.4 Performance Evaluation

In this section, we evaluate the hybrid 2D and 3D FFT implementation on a heterogeneous computer with a GeForce GTX480 GPU and an Intel i7 920 CPU. The GTX480 GPU has a Fermi architecture [65] which supports for 16 concurrent kernels execution. Different kernels of the same application context can execute on the GPU at the same time. The Intel i7 920 CPU has the ability to process up to eight threads simultaneously. The configurations of the GPU, CPU and FFT libraries are listed in table 3.2.

We compare our library against FFTW and Intel MKL, two of the best performing FFT implementations. We can't compare our library with other GPU-based FFT implementations because all of them require problem sizes to be smaller than the GPU memory. Our problems are larger and therefore can't fit. In FFTW, Streaming Single Instruction Multiple Data Extensions (SSE) on Intel CPU is enabled for better performance. Also FFTW uses the 'MEASURE' flag in searching since it provides much better performance than another flag 'ESTIMATE'. However the 'EXHAUSTIVE' flag is not used because for the problem sizes in this evaluation are so large that FFTW

can't finish the search under the 'EXHAUSTIVE' mode. 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 support 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. In summary, the FFTW and Intel MKL are tested with their best configurations on this computer.

All FFT problems are out-of-place with separate input and output filled with complex numbers. Furthermore, we choose the test cases from 128M points (i.e. 2^{27}) to 512M points (i.e. 2^{29}), 512M being the maximum problem size that can fit into CPU memory. The performance of a D dimensional out-of-place complex FFT is evaluated in GFLOPS defined as $GFlops = \frac{5M\sum_{d=1}^{D} log_2 N_d}{t} \times 10^{-09}$ where the total problem size is $M = N_1 \cdot N_2 \cdot ... \cdot N_D$ and t is execution time in seconds.

3.4.1 Evaluation of 2D Hybrid FFT

We evaluate performance of 2D hybrid FFT by identifying the optimal load ratios at first. For a 2D FFT, computation load is distributed to GPU and CPU along X dimension in first round and along Y dimension in second round. We perform experiments to analyze the best point of performance by tuning distribution ratio of GPU to CPU for each round. For each instance, we gradually adjust the ratio of GPU to CPU from 3 : 1 to the maximum one that our implementation can tolerate. For example, given a 2D hybrid FFT of size $N = Y \times X = 2^{14} \times 2^{14}$, the optimal performance is gained under the ratio of 127 : 1 along X dimension and 127 : 1 along Y dimension. The peak performance under the optimal ratio is 18.1 GFLOPS for the size of $N = 2^{28}$. We test performance of single-precision problem with sizes from 2^{27} to 2^{29}. Figure 3.9 shows that performance of single-precision 2D hybrid FFT in our heterogeneous system achieves 17.6 GFLOPS on average, and is 61% faster than the 4-thread FFTW and 1.43× faster than the 4-thread MKL.

We also test the 2D FFT performance in double-precision. For example, given a problem of size $N = Y \times X = 2^{15} \times 2^{12}$, the optimal performance is gained under

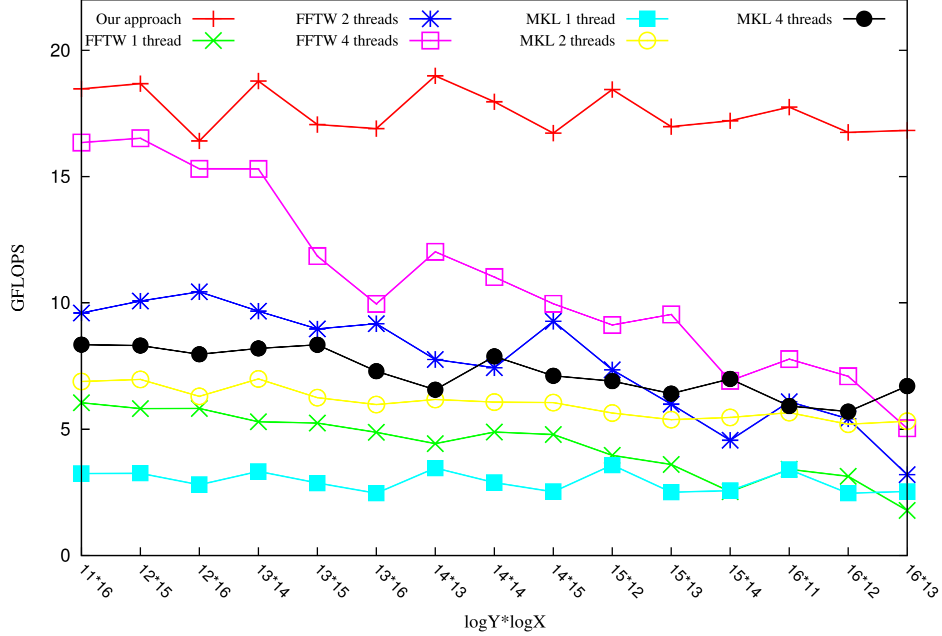


Figure 3.9: Performance of Single-precision 2D Hybrid FFT.

the ratio of 31 : 1 along X dimension and 31 : 1 along Y dimension. The peak performance under the optimal ratio is 8.1 GFLOPS for the size of $N = 2^{27}$. This ratio is smaller than that of single precision since CPU has relatively higher efficiency on double precision operations than GPU. Performance of double-precision 2D hybrid FFT is shown in figure 3.10. It achieves 8.6 GFLOPS on average and is 16% faster than 4-thread FFTW and 15% faster than 4-thread MKL.

3.4.2 Evaluation of 3D Hybrid FFT

For a 3D FFT, computation load is distributed to GPU and CPU along X dimension for the first round, and along Z dimension in the second round. The third round is computed by CUFFT since its superiority in handling 1D FFT with continuous data in memory. Similarly, we tune work ratio of GPU to CPU in each round and achieve the best point of performance under the optimal ratios. For example, optimal performance of size $Z \times Y \times X = 2^{15} \times 2^5 \times 2^7$ is gained under the ratio 63 : 1 along

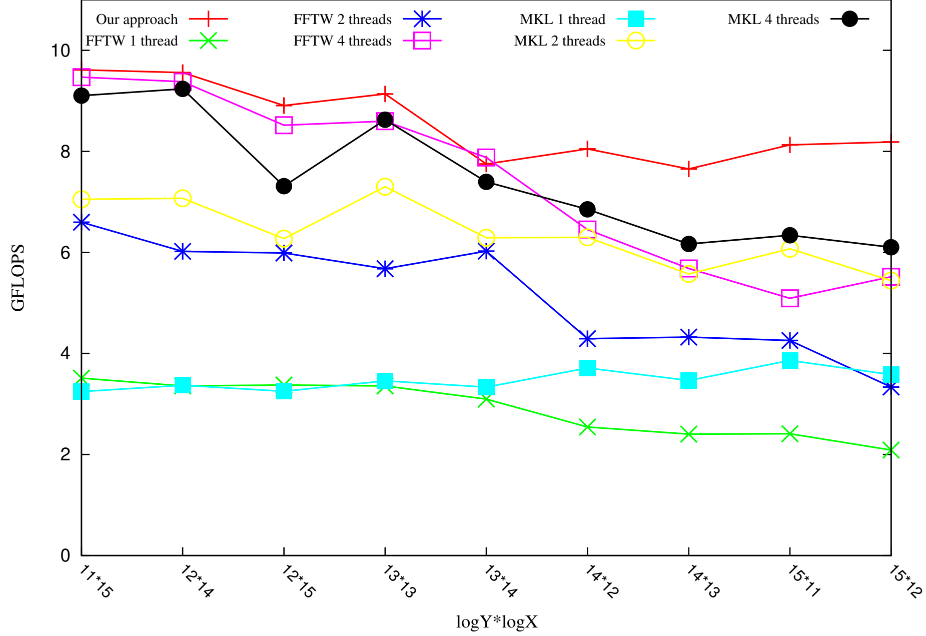


Figure 3.10: Performance of Double-precision 2D Hybrid FFT.

X dimension and 255 : 1 along large size Z dimension. The peak performance under the optimal ratio 11.8 GFLOPS for the size of $N = 2^{27}$. Figure 3.11 shows that our single precision 3D FFT library achieves on average 10.9 GFLOPS and is 21% faster than the 4-thread FFTW and 42% faster than the 4-thread MKL.

3.4.3 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 the range of [0.5, 0.5) and the difference in output is quantized as normalized RMSE over the whole data set. The normalized RMSE is an useful measurement since it evaluates the relative degree of deviations. The normalized RMSE is calculated as Normalized $RMSE = \sqrt{\frac{\sum_{i=0}^{N-1}(X_i-R_i)^2+(Y_i-S_i)^2}{2N}} / \sqrt{\frac{\sum_{i=0}^{N-1}((R_i^2+S_i^2)}{2N}}$.

The accuracy of single precision measured by normalized RMSE is shown in figure 3.12 for 2D and 3D FFT. As we could see normalized RMSE is extremely small

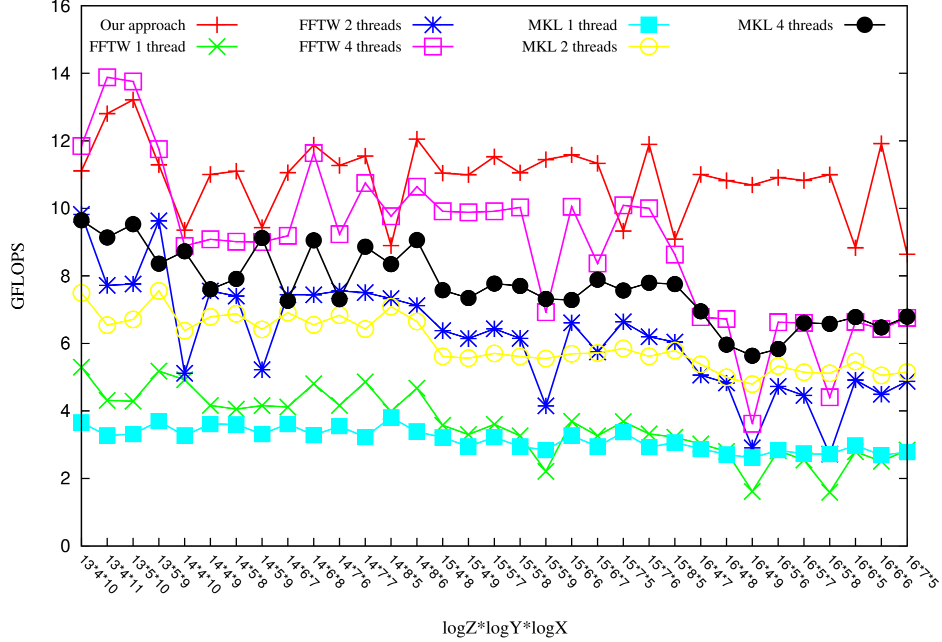


Figure 3.11: Performance of 3D Single-precision Hybrid FFT.

and is in the range from 3.9×10^{-07} to 6.5×10^{-07}. Moreover, the accuracy of double precision 2D hybrid FFT is in the range 5.9×10^{-16} to 6.4×10^{-16}.

3.5 Chapter Summary

In this chapter, we propose radix decomposition based hybrid FFTs on heterogeneous GPU-CPU computers. A hybrid framework is proposed to efficiently partition and distribute work into GPU and CPU, and concurrently uses these two different computing devices to compute multi-dimensional FFTs. Particularly, our approach integrates radix-decomposition paradigms to exploit hidden parallelism to utilize parallel computing resources, and to tailor the extraction of computation and communication patterns for GPU and CPU. In order to obtain the optimal performance, we develope a empirical profiling to guide the entire balancing process. Our library also overlaps data transfers and maintaining consistency between GPU and CPU. On average, our single precision FFT on a heterogeneous system of a GeForce GTX480 GPU and an

Figure 3.12: Accuracy of 2D and 3D Hybrid FFT in Single Precision.

Intel Core i7 CPU is 5.8× and 4.1× faster than 1-thread SSE-enabled FFTW and Intel MKL, respectively. The accuracy of single precision measured by normalized RMSE is in the range from 2.41×10^{-07} to 3.15×10^{-07}.

Chapter 4

A CO-OPTIMIZED WELL-TUNED PARALLEL HYBRID GPU-CPUS FFT LIBRARY FOR LARGE FFT PROBLEMS

In this chapter, we propose a co-optimized and well-tuned parallel hybrid GPU-CPUs FFT library for large FFT problems that exceed GPU memory size. We first review the novelty of our proposed co-optimized and well-tuned hybrid approach. Then, we elaborate on the co-optimized hybrid 2D and 3D FFT frameworks, respectively, to describe a more flexible decomposition to exploit substantial parallelism and a new co-optimizer to further deeply optimize the computation and communication inside either GPU or CPU, respectively. We subsequently apply empirical modeling and tuning, and evaluate the optimal performance.

4.1 Overview of Our Approach

For our approach, a hybrid large-scale FFT partition framework is applied to divide total problem into sub-problems, to distribute sub-problems into multi-CPUs and GPU, and to parallelize the sub-problem in each side. In particular, a more flexible cooley-tukey decomposition method is applied to decompose the workload, and to exploit parallelism as much as possible until all the computing resources are fully utilized. Another important novel technique in this work is to propose a co-optimizer, which is to exploit substantial parallelism for GPU and CPUs, and to optimize the performance in each GPU and CPU based on its architectural features, such as processor throughput, memory bandwidth and I/O communication bandwidth. The co-optimizer is also utilized to enable concurrent GPU and CPUs execution, to optimize data sharing, and to provide an efficient synchronization mechanism for both GPU and CPUs to keep

program consistency and to accelerate performance. Additionally, empirical performance modeling and tuning are proposed to estimate performance in each sub-step of entire program, and hence to predict and determine optimal load balancing between GPU and CPUs based on several model parameters. It replaces an exhaustive walk-through of the vast space of possible hybrid implementations of FFT on heterogeneous GPU-CPUs system with a guided empirical search. Furthermore, an effective heuristic is used to enable stream based asynchronous executions, and to overlap communication with computation.

4.2 Co-Optimized 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 4.1,

$$\begin{aligned} out(k_y, k_x) &= \sum_{x=0}^{X-1} W_X^{xk_x} \sum_{y=0}^{Y-1} W_Y^{yk_y} f(y,x) \\ &= \sum_{x=0}^{X-1} W_X^{xk_x} \sum_{y=0}^{Y-1} \{W_Y^{yk_y} f(y, x_{gpu}) + W_Y^{yk_y} f(y, x_{cpu})\} \end{aligned} \tag{4.1}$$

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 4.1, 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

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 4.2.

$$
\begin{aligned}
u_{gpu} =& \{d(Y_1, Y_2Xgpu, Xgpu, Igpu, Ogpu), Sync, \\
& t^{Y_1}_{Y_2} d(Y_2, Y_1X, Y_1X, O, O), d(X, 1, 1, O, O)\} \\
u_{cpu} =& \{d(Y_1, Y_2Xcpu, Xcpu, Icpu, Ocpu), Sync\}
\end{aligned}
\tag{4.2}
$$

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 $t^{Y_1}_{Y_2}$ is introduced by the Cooley-Tukey decomposition between Y_1 and Y_2 step. Figure 4.1 shows the high-level working flow of our co-optimized hybrid 2D FFT framework.

4.2.1 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.

4.2.2 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

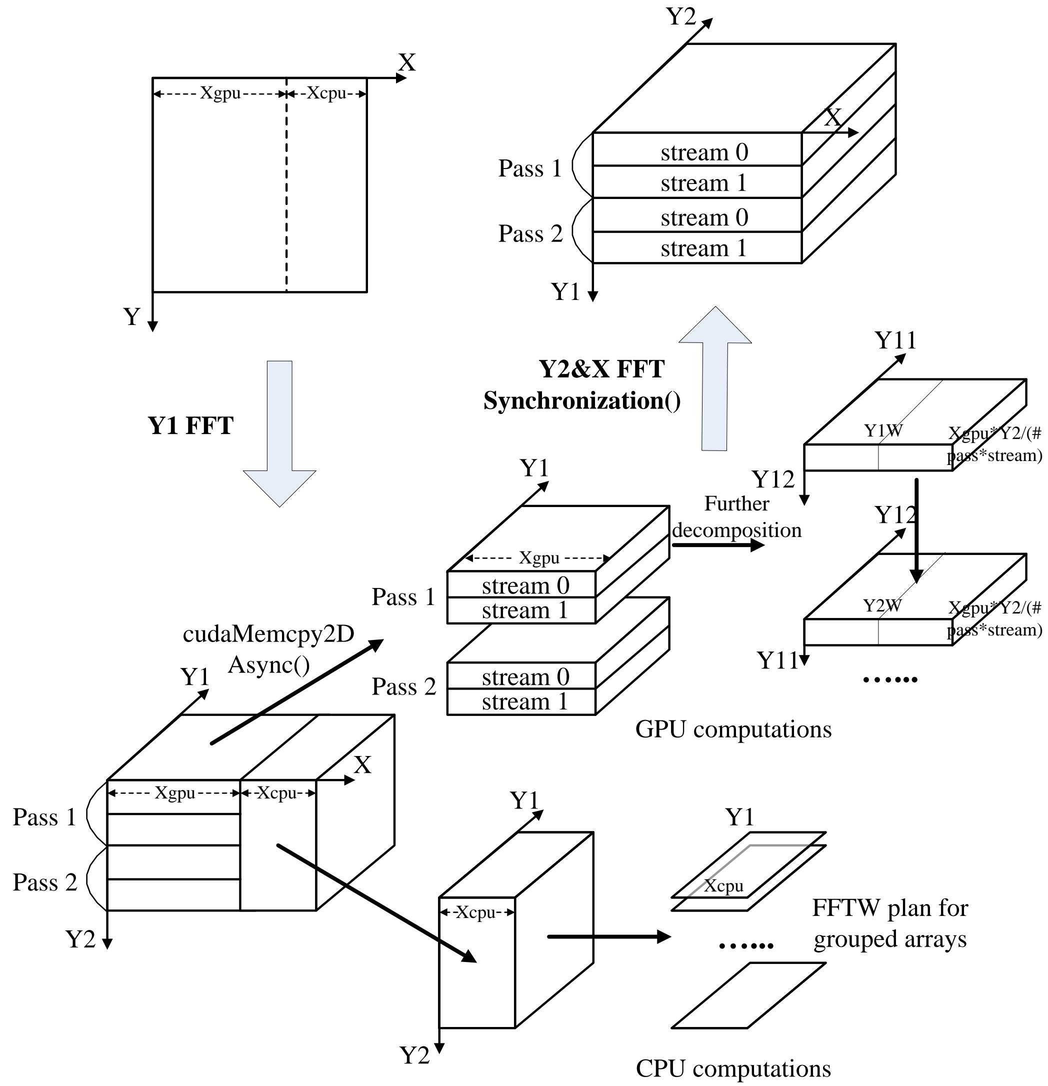


Figure 4.1: Overview of Co-Optimized Hybrid Large Out-of-card 2D FFT.

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.

4.2.3 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.

4.2.3.1 Data Transfer Scheme

In addition to the computational overhead, 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.

4.2.3.2 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 needs to use PCI bus once without allocating redundant CPU resources for Gu's buffering. Figure 4.2 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.

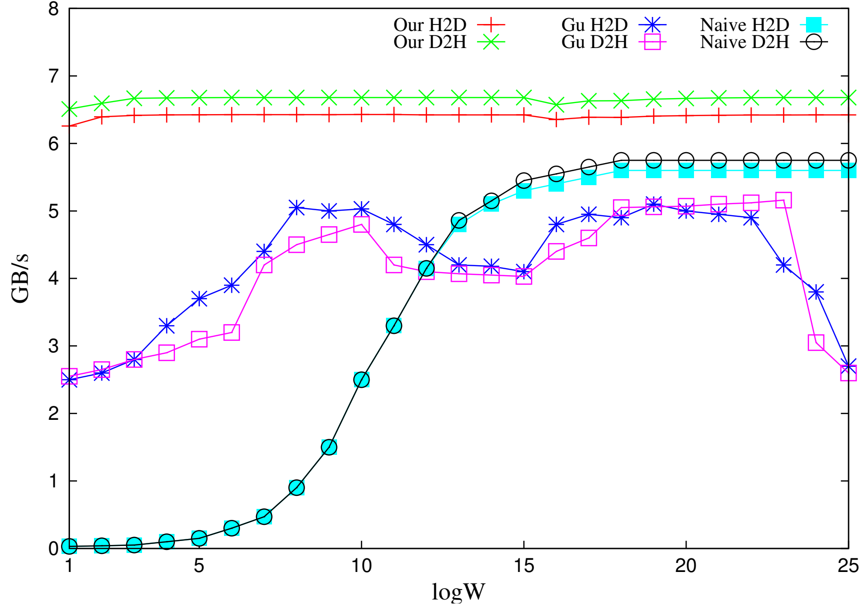


Figure 4.2: PCI Bandwidth of Different Data Transfer Schemes.

4.2.3.3 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 problem due to frequent data exchanges. Therefore achieving peak performance in heterogeneous GPU-CPUs system is a challenge. As evaluated in the result section, 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.

4.2.4 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.

4.2.5 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 sections. Independent grouped FFT computation steps are carried out in each parallel section. 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.

4.2.6 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 workload to FFTW. This heterogeneous method was first proposed in Ogata's work [8]. Computation is firstly distributed along X dimension in the 1st-round of 2D FFT and along Y dimension for the 2nd-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 4.3. 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.

4.3 Co-Optimized 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

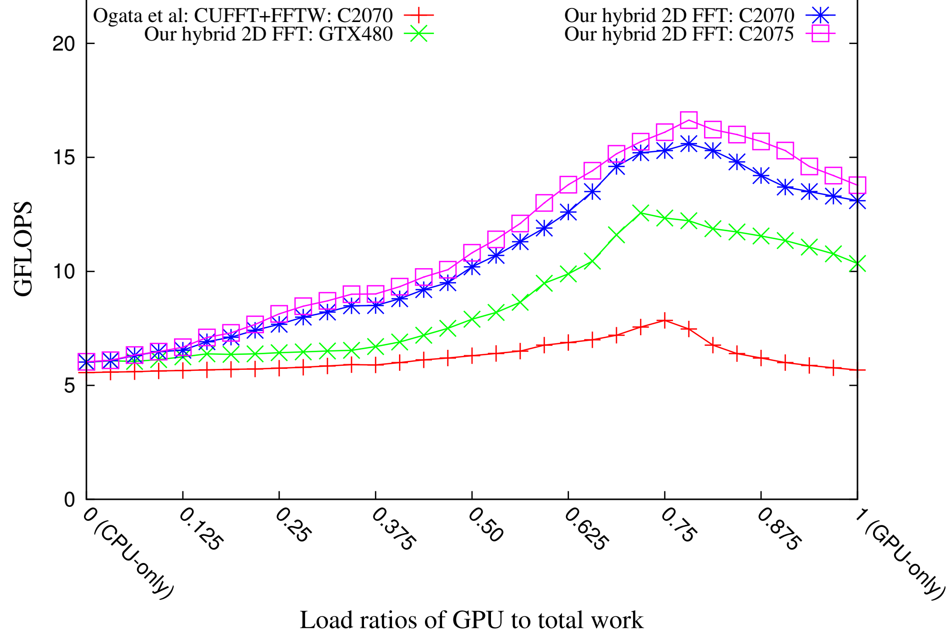


Figure 4.3: Double Precision 2D FFT Performance Tuning Comparison of Our Co-Optimized Hybrid FFT against Hybrid CUFFT/FFTW Library.

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 4.3, where $|_i$ and $|_o$ denotes respective input and output data transfers through PCI bus.

$$\begin{aligned}&\{|_i, d(Z_1, XYZ_2, XY), |_o|_i, t_{Z_2}^{Z_1} d(Z_2, XYZ_1, XYZ_1), |_o|_i,\\&d(Y_1, Y_2X, X), |_o|_i, t_{Y_2}^{Y_1} d(Y_2, Y_1X, Y_1X), |_o|_i, d(X, 1, 1), |_o\}\end{aligned} \tag{4.3}$$

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 4.4.

$$\begin{aligned}&\{|_i, d(Z_1, XYZ_2, XY), d(Y_1, Y_2X, X), t_{Y_2}^{Y_1} d(Y_2, Y_1X,\\&Y_1X), |_o|_i, t_{Z_2}^{Z_1} d(Z_2, XYZ_1, XYZ_1), d(X, 1, 1), |_o\}\end{aligned} \tag{4.4}$$

Next let's examine how the workload as represented in the equation 4.4 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 $t_{Z_2}^{Z_1}$ with Z_2 dimensional FFTs and execute X dimensional FFTs before final results are transferred back to CPU. Such GPU execution flow in formula 4.4 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 4.3, 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 4.4.

4.3.1 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 4.4, 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 4.5, where $Sync$ represents data

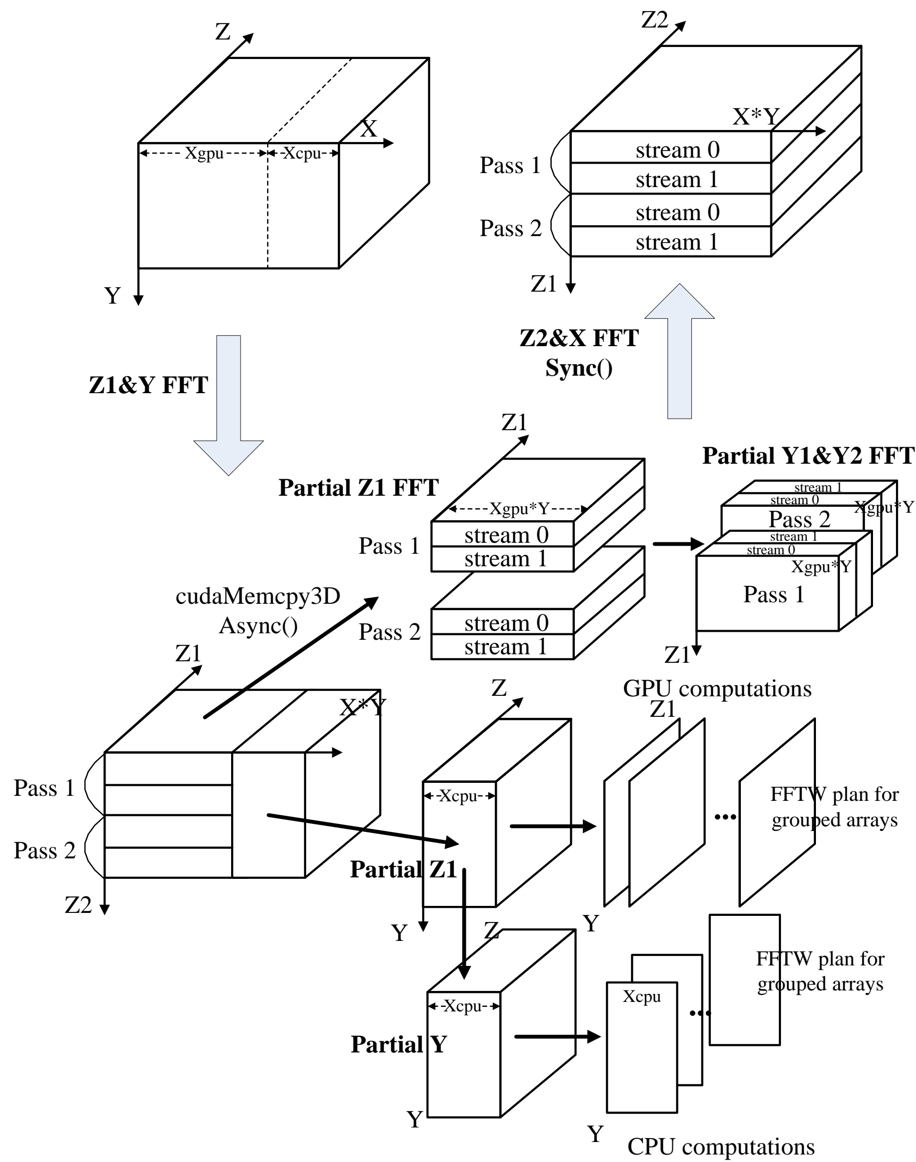


Figure 4.4: Overview of Co-Optimized Hybrid Large Out-of-card 3D FFT.

transfer and synchronization between CPU and GPU within computation.

$$\begin{aligned}
&u_{gpu} = \{|_i, d(Z_1, XgpuYZ_2, XgpuY),\\
&d(Y_1, Y_2Xgpu, Xgpu), t^{Y_1}_{Y_2}d(Y_2, Y_1Xgpu, Y_1Xgpu),\\
&Sync, t^{Z_1}_{Z_2}d(Z_2, XYZ_1, XYZ_1), d(X, 1, 1), |_o\}\\
&u_{cpu} =\\
&\{d(Z_1, XcpuYZ_2, XcpuY), d(Y, Xcpu, Xcpu), Sync\}
\end{aligned} \tag{4.5}$$

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 $R_X = \frac{Xgpu}{Xcpu}$.

4.3.2 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.

4.3.3 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.

4.4 Load Balancing between GPU and CPUs with Empirical Modeling and Tuning

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.

4.4.1 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 $Y_1W \times max(Y_{11}, Y_{12})$, 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 4.6.

$$\begin{aligned} TG_{2D} = \quad & \#passes \times max\{[Y_1 \times T_{2d\text{H2D}}(0, R_g) + \\ & T_{Y_1\text{kernel}}(0, R_g) + T_{2d\text{D2H}}(0, R_g)]; [...]; \\ & [Y_1 \times T_{2d\text{H2D}}(\text{\# streams-1}, R_g) \\ & + T_{Y_1\text{kernel}}(\text{\# streams-1}, R_g) \\ & + T_{2d\text{D2H}}(\text{\# streams-1}, R_g)]; \quad \} \end{aligned} \tag{4.6}$$

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.

4.4.2 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.

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 $Z_1W \times max(Z_{11}, Z_{12})$, 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_1W, grid size is $\frac{R_g \times Y_2}{Y_1W} \times \frac{Z}{\#\text{ passes} \times \#\text{ streams}}$
$T_{Y_2\text{kernel}}(i, R_g)$	$= T_{Y_2\text{kernel-gpu}} \times R_g$. Time of Y_2-step FFTs computation of concurrent kernel in stream i. Thread block size is Y_2W, grid size is $\frac{R_g \times Y_1}{Y_2W} \times \frac{Z}{\#\text{ passes} \times \#\text{ streams}}$,
$T_{3d\text{D2H}}(i, R_g)$	$= T_{3d\text{D2H-gpu}} \times R_g$. 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 GPU side, for hybrid $Z_1\&Y$ dimensional FFTs, the execution time is estimated as TG_{3D} shown in equation 4.7.

$$
\begin{aligned}
TG_{3D} = \quad & \#passes \times max\{[Z_1 \times T_{3d\text{H2D}}(0, R_g)+ \\
& T_{Z_1\text{kernel}}(0, R_g) + T_{Y_1\text{kernel}}(0, R_g)+ \\
& T_{Y_2\text{kernel}}(0, R_g) + T_{3d\text{D2H}}(0, R_g)]; \quad [......]; \\
& [Z_1 \times T_{3d\text{H2D}}(\#\text{ streams-1}, R_g) \\
& + T_{Z_1\text{kernel}}(\#\text{ streams-1}, R_g) \\
& + T_{Y_1\text{kernel}}(\#\text{ streams-1}, R_g) \\
& + T_{Y_2\text{kernel}}(\#\text{ streams-1}, R_g) \\
& + T_{3d\text{D2H}}(\#\text{ streams-1}, R_g)];\}
\end{aligned} \tag{4.7}
$$

On CPU side, for hybrid $Z_1\&Y$ dimensional FFTs, the execution time is estimated as TC_{3D} represented in equation 4.8.

$$
\begin{aligned}
TC_{3D} = \quad & \frac{(1 - R_g) \times X \times Z_2}{\#thds} \times T_{Z_1\text{fftw}} \\
& + \frac{Z}{\#thds} \times T_{Y\text{fftw}}(1 - R_g)
\end{aligned} \tag{4.8}
$$

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.

4.5 Performance Evaluation

In this section, we evaluate our hybrid 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. Configurations of the FFT libraries, the GPUs and the CPU with

total 24GB host memory are summarized in table 4.3, where NVCC is the compiler driver for NVIDIA CUDA GPUs.

Table 4.3: Configurations of GPU, CPU, FFTW and MKL.

GPU	**Global Memory**	**NVCC**
GeForce GTX480	1.5GB	3.2
Tesla C2070	6GB	3.2
Tesla C2075	6GB	3.2
CPU	**Frequency, # of Cores**	**FFTW & MKL**
Intel i7 920	2.66GHz, 4 cores	3.3.3 & 10.3

The performance reported here includes both computational time and data transferring time between host and device. Our library performance is compared to that of 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 pure CUFFT implementation because it requires problem size to be smaller than GPU memory, and therefore won't accept problem sizes used in this evaluation. In FFTW, Streaming Single Instruction Multiple Data Extensions (SSE) on Intel CPU 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. In summary, the FFTW and Intel MKL are tested with their best configurations on this computer. Moreover, Intel MKL provides FFTW3 wrappers for translating FFTW functions into the Intel MKL FFT functions. It uses the FFTW3 library to gain performance of Intel MKL and provides equivalent implementation for double- and single-precision functions. All the single-precision functions with prefix 'fftwf' will refer to functions with prefix 'fftw' which are double-precision functions.

Since co-optimization of computation and communication for large-scale FFT problem exceeding GPU memory size is one of the key innovations of our approach, therefore the smaller problems that fit into GPU memory are not tested at this time. We carry out three groups of performance evaluations: a) GTX480 and i7 920 CPU with 12GB host memory; b) Tesla C2070 and i7 920 CPU with 24GB host memory; c) Tesla C2075 and i7 920 CPU with 24GB host memory. Performance of our library is compared with multithreading enabled FFTW and Intel MKL.

All FFT problem sizes are larger than GPU memory. 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. The performance of a D dimensional complex FFT is evaluated in GFLOPS [5] defined as $GFlops = \frac{5M\sum_{d=1}^{D} log_2 N_d}{t} \times 10^{-09}$ where the total problem size is $M = N_1 \cdot N_2 \cdot ... \cdot N_D$ and t is execution time in seconds.

4.5.1 Performance Tuning

For both 2D and 3D FFTs, the best performance is achieved by tuning under different load distribution ratios between CPU and GPU. Therefore, we need to tune the performance with different load ratios between CPU and GPU. Our performance

modeling and empirical searching is to guide us to automatically find the optimal load balancing for different CPU/GPU combinations under different input sizes. Our approach 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.

For example, for each fixed input size of our 2D hybrid FFT library, we used two extreme profiling runs, i.e., one is run on only CPU and the other is run on only GPU. Therefore, the running time of each model parameter, $T_{2d\text{H2D-gpu}}$, $T_{Y_1\text{kernel-gpu}}$, and $T_{2d\text{D2H-gpu}}$, in either CPU-only or GPU-only case can be determined. Afterwards, the load ratio of GPU to total work is automatically varied from 0% to 100% for our performance tuning. The execution time of the model parameters under each load ratio can be calculated by associating the running time of parameters in the two extreme cases above with the varied ratios. Subsequently, instead of running the whole implementation for a large amount of times, our modeling scheme automatically works to estimate the execution time of GPU, i.e. TG_{2D}, the execution time of CPU, i.e. TC_{2D}, and the total execution time T_{total} for each ratio and each problem size. Our performance model provides a well reasonable method that associates the running time of model parameters with the varied load ratios for relatively accurate performance approximation.

Hence, our modeled performance tuning is able to be attained. In such a case, empirical searching helps find the best load ratio for each input size. The optimal ratio obtained from our performance model might not be 100% accurate, but it provided us a small and very precise region where the actual optimal ratio resides in. Therefore, it guides us to efficiently determine the actual optimal ratio in our performance evaluation instead of a walk through a vast space of all possible cases. Under actual optimal ratio, load balancing between GPU and CPU is achieved. 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.

Figure 4.5 demonstrates the effectiveness of load distribution ratio tuning on

overall FFT performance. The figure shows the actual and modeled double-precision 2D FFT performance on three different GPUs under different load ratios with problem size $2^{15} \times 2^{13}$. The x-axis is the distribution ratio from 0% to 100%. 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 and communication into two devices. the final performance of our library is also compared against the cases that run our library only on GPU or only on CPU. Table 4.4 shows the tested values of model parameters for the profiling runs of GPU-only and CPU-only case described in the load balance section above. 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 4.6 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 4.5, for GTX480, Tesla C2070 and C2075, the

Table 4.4: Valid Model Parameters in Seconds for FFTs of Size $2^{15} \times 2^{13}$ and $2^{10} \times 2^9 \times 2^9$.

Parameter	Seconds	Parameter	Seconds	Parameter	Seconds
$T_{2d\text{H2D-gpu}}$	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		
Parameter	**Seconds**	**Parameter**	**Seconds**	**Parameter**	**Seconds**
$T_{3d\text{H2D-gpu}}$	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		

optimal ratios of GPU to total work are 71.9%, 78.2% and 78.2%, respectively. The 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 4.6, 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.

4.5.2 Evaluation of 2D Hybrid FFT

The performance of 2D hybrid FFT is evaluated by identifying the optimal load ratios for each problem size at first. To assess performance with varied values of Y and X provides us a deep perspective to analyze performance tendency. In all the figures, the test points are indexed in an increasing order of Y in the problem sizes. The 2D hybrid FFT performance of all test points are reported with the best empirically found work distribution ratio of GPU to CPU.

Figure 4.7 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

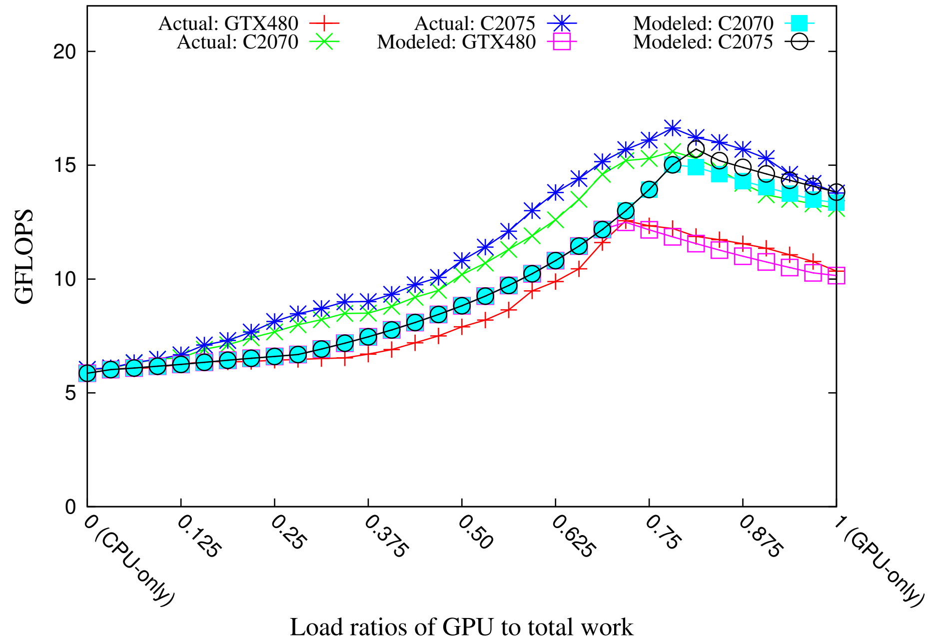


Figure 4.5: Double-precision 2D FFT Performance Tuning.

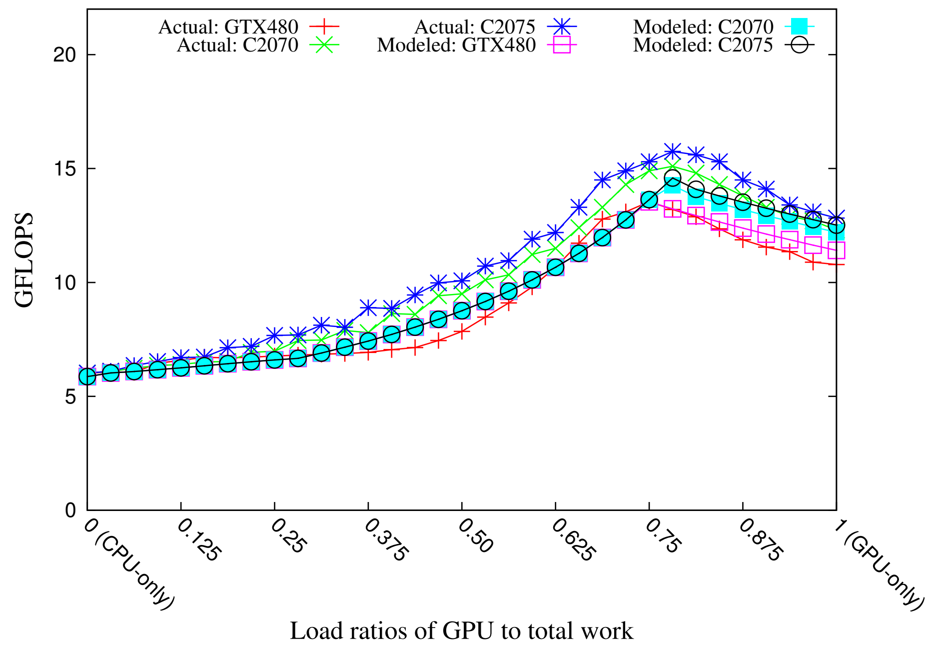


Figure 4.6: Double-precision 3D FFT Performance Tuning.

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 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 4.8. 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 4.9 and figure 4.10 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.

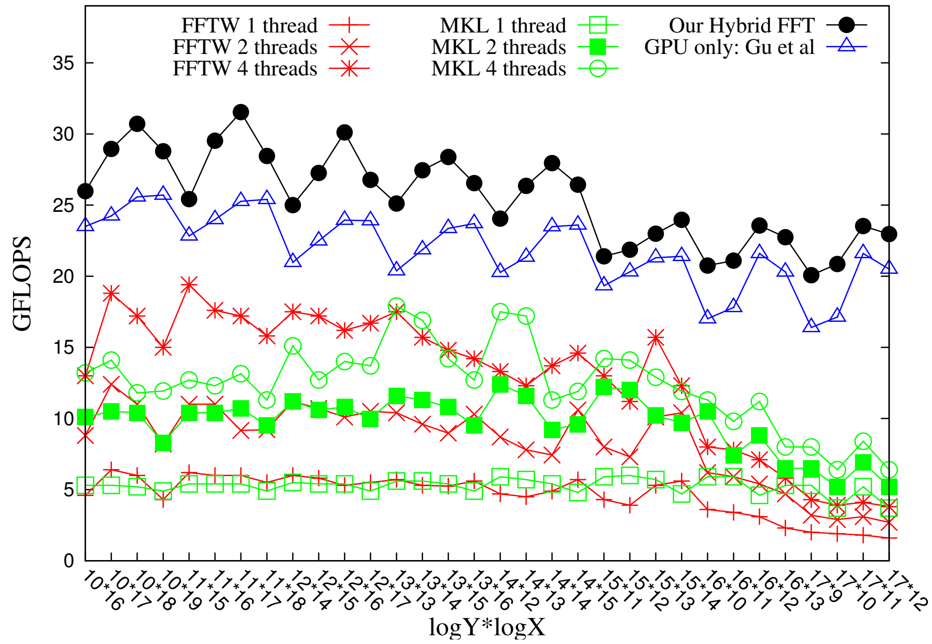


Figure 4.7: Single-precision 2D FFT of Size from 2^{26} to 2^{29} on GTX480.

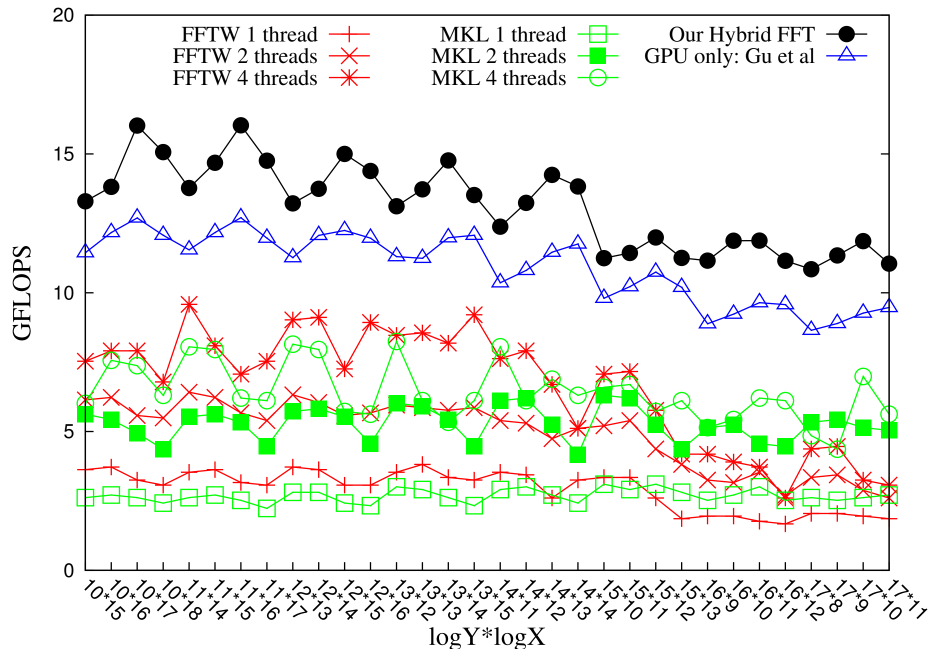


Figure 4.8: Double-precision 2D FFT of Size from 2^{25} to 2^{28} on GTX480.

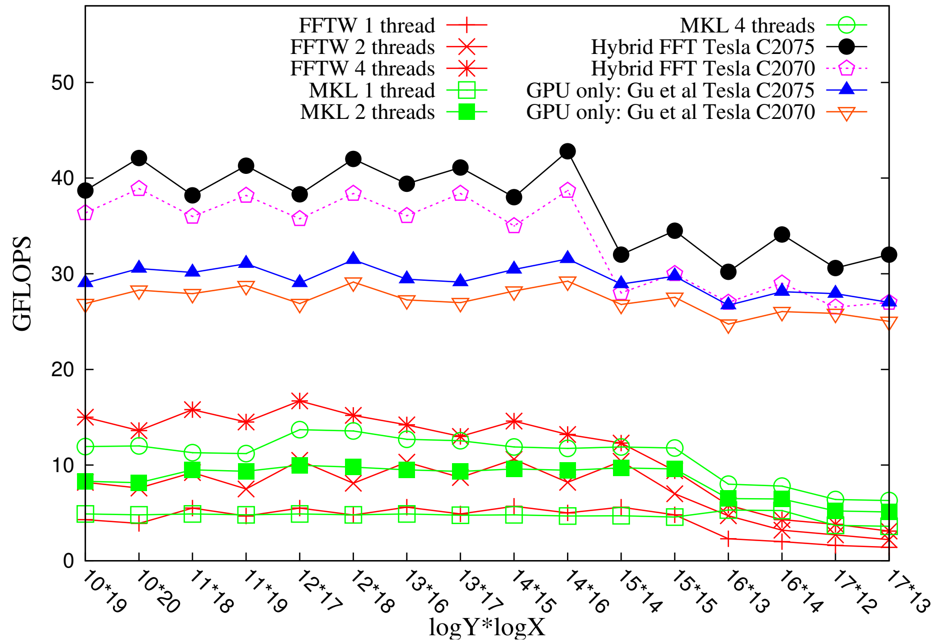


Figure 4.9: Single-precision 2D FFT of Size from 2^{29} to 2^{30} on Tesla.

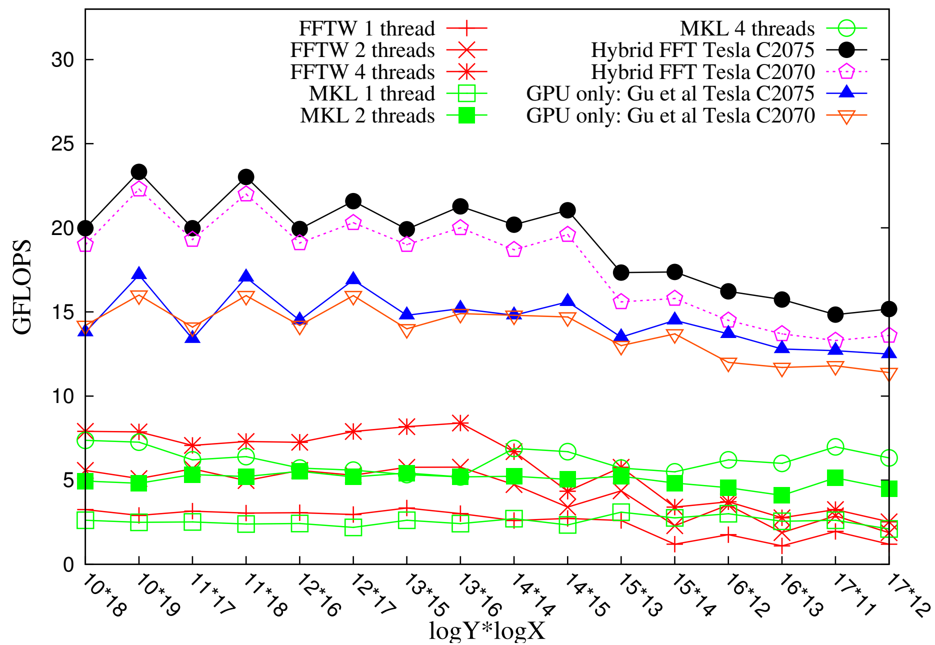


Figure 4.10: Double-precision 2D FFT of Size from 2^{28} to 2^{29} on Tesla.

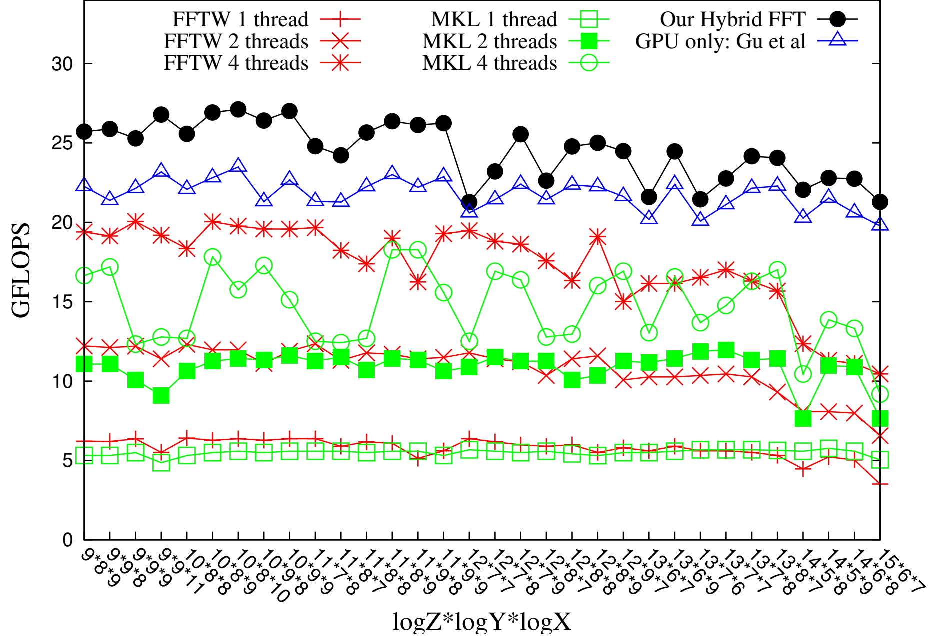


Figure 4.11: Single-precision 3D FFT of Size from 2^{26} to 2^{29} on GTX480.

4.5.3 Evaluation of 3D Hybrid FFT

Figure 4.11, 4.13 and figure 4.12, 4.14 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.

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.

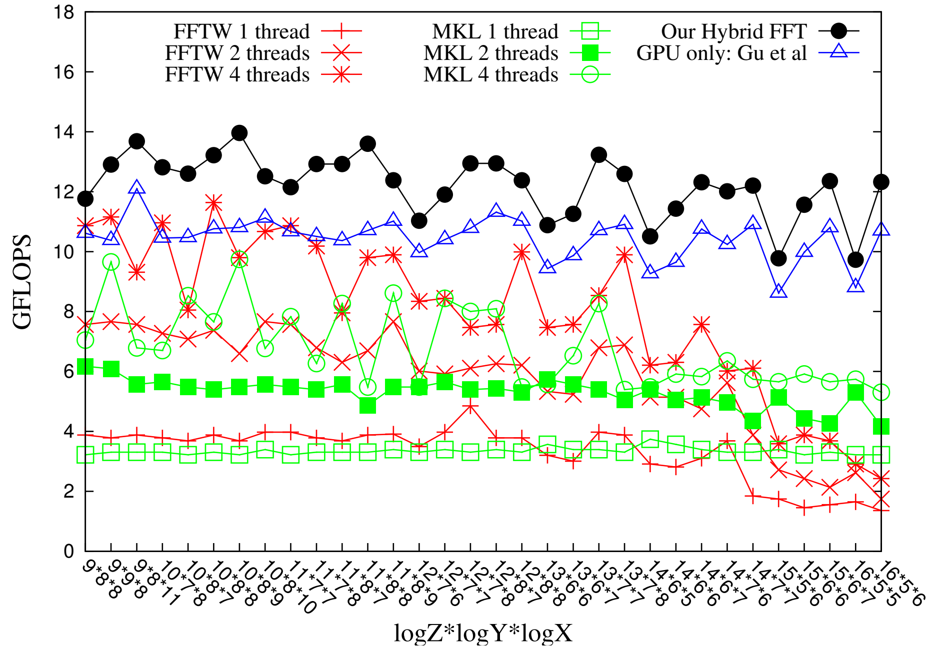


Figure 4.12: Double-precision 3D FFT of Size from 2^{25} to 2^{28} on GTX480.

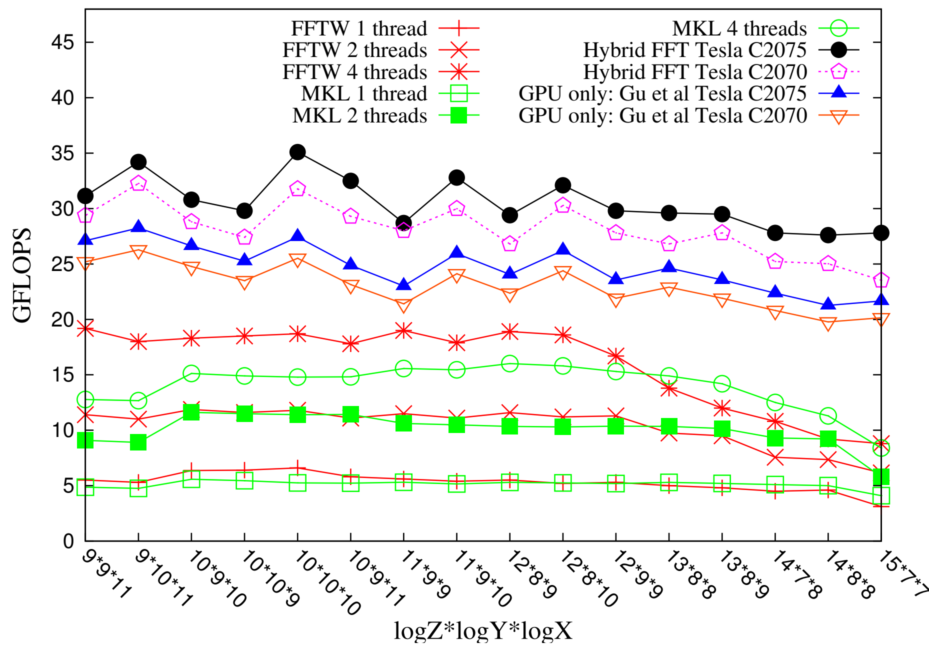


Figure 4.13: Single-precision 3D FFT of Size from 2^{29} to 2^{30} on Tesla.

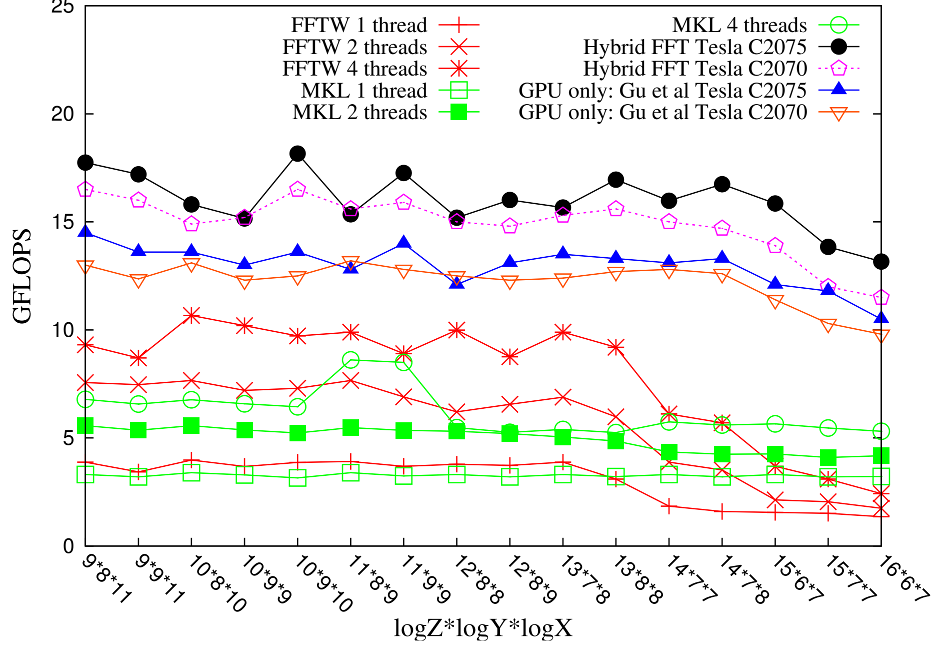


Figure 4.14: Double-precision 3D FFT of Size from 2^{28} to 2^{29} on Tesla.

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

$$\sqrt{\frac{\sum_{i=0}^{N-1}(X_i - R_i)^2 + (Y_i - S_i)^2}{2N}} \Big/ \sqrt{\frac{\sum_{i=0}^{N-1}(R_i^2 + S_i^2)}{2N}}. \tag{4.9}$$

The normalized RMSEs of single- and double-precision for both 2D and 3D FFTs are shown in figure 4.15 and figure 4.16. 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.

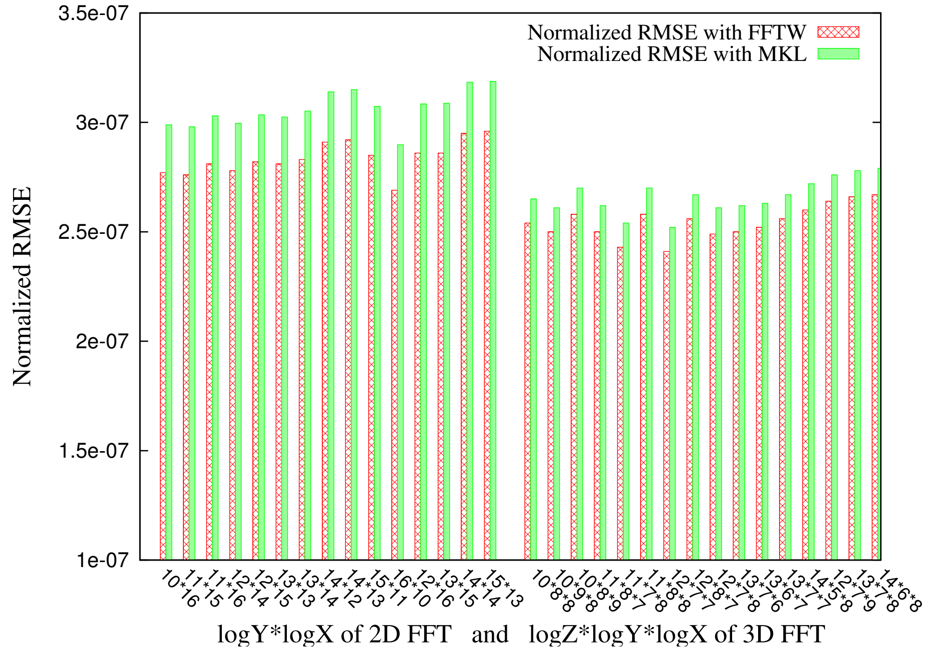


Figure 4.15: Accuracy of Single-precision Hybrid 2D/3D FFT.

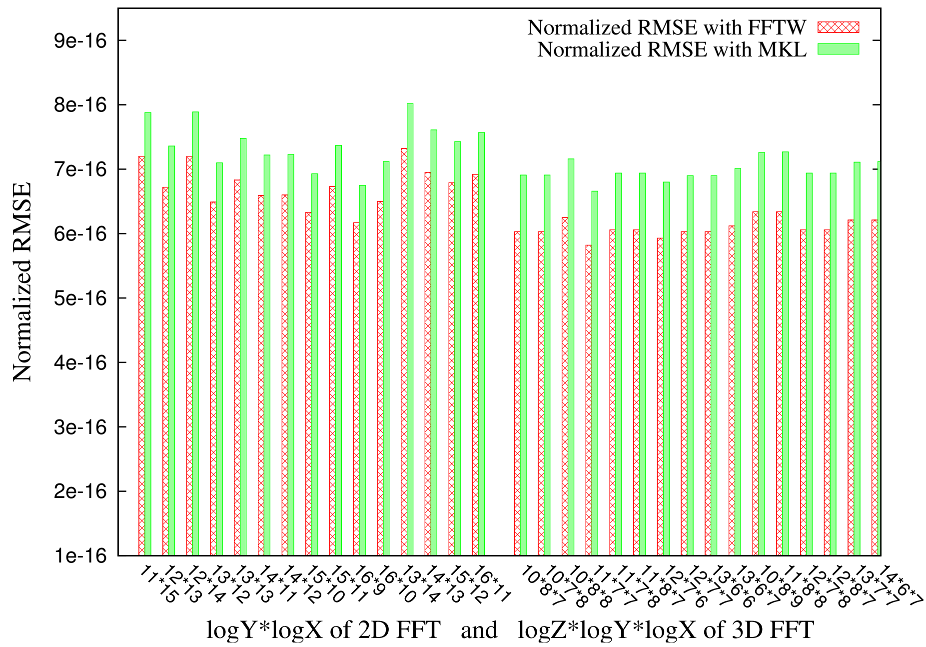


Figure 4.16: Accuracy of Double-precision Hybrid 2D/3D FFT.

4.6 Chapter Summary

In this chapter, we propose a co-optimized and well-tuned parallel hybrid GPU-CPUs FFT library for large FFT problems. A hybrid large-scale FFT partition framework is applied to do the work partition and load distribution for both GPU and CPU. In particular, a more flexible cooley-tukey decomposition method is applied to decompose the workload in each GPU and CPU, and to exploit parallelism until all the computing resources are fully utilized. Another important novel technique in this work is a co-optimizer, which is to exploit substantial parallelism, and to optimize the performance in each GPU and CPU, to enable concurrent GPU and CPUs execution, and to provide an efficient synchronization mechanism for good consistency and fast performance. Moreover, the empirical performance modeling and tuning are proposed to estimate, predict and determine optimal load balancing between GPU and CPUs. Meanwhile, an effective heuristic is used to enable stream based asynchronous executions, and to overlap communication with computation. Finally, we evaluate the performance in three high performance heterogeneous systems. Overall, our hybrid FFT implementation outperforms several latest and widely used large-scale FFT implementations. For instance, our Tesla C2070 performance is 14.1× and 8.6× faster than 1-thread SSE-enabled FFTW library and Intel MKL math library, respectively. Our performance is 1.9× and 2.1× faster than 4-thread SSE-enabled FFTW and Intel MKL, with max speedups 4.6× and 2.8×, respectively. In addition, our performance has 1.3× and 3.4× speedup over Gu et.al.'s and Ogata et.al.'s work.

Chapter 5

AN INPUT-ADAPTIVE ALGORITHM FOR HIGH PERFORMANCE SPARSE FAST FOURIER TRANSFORM

In this chapter, we go over several existing sparse FFT algorithm versions to explain the evolution from a general sparse FFT algorithm to the proposed input-adaptive sparse FFT algorithm. We first describe a general input adaptive sparse FFT algorithm which comprises of input permutation, filtering non-zero coefficients, subsampling FFT and recovery of locations and magnitudes. Subsequently, we discuss how to save the number of permutations and propose an alternatively optimized version for our sparse FFT algorithm to gain runtime improvement. Moreover, the general and the optimized versions are hybridized so that we're able to choose a specific version according to input characteristics. Finally, an example of real world application is described to illustrate our input adaptive approach.

5.1 Overview of Sparse FFT Algorithms and Our Approach

In this section we overview prior work on sparse Fourier transform, and then describe our contribution in that context.

A naive discrete Fourier transform of a N-dimensional input series $x(n)$, $n = 0, 1, ..., N-1$ is presented as $Y(d) = \sum_{n=0}^{N-1} x(n)W_N^{nd}$, where $d = 0, 1, ..., N-1$ and twiddle factor $W_N^{nd} = e^{-j2\pi nd/N}$. Fast Fourier transform algorithms recursively decompose a N-dimensional DFT into several smaller DFTs [9], and reduce DFT's operational complexity from $O(N^2)$ into $O(NlogN)$. There are many FFT algorithms, or in other words, different ways to decompose DFT problems. The Radix FFT algorithm [9] and the Cooley-Tukey FFT algorithm [10] are factorization FFT algorithms. Additionally, Prime-Factor (Good-Thomas) [23] decomposes a DFT of size $N = N_1N_2$, where N_1 and

N_2 are co-prime numbers. Twiddle factor calculation is not included in this algorithm. Moreover, Rader's algorithm [13] and Bluestein's algorithm [14] can factorize a prime-size DFT as convolution. So far, all FFT algorithms cost time at least proportional to the size of input signal. However, if the output of a DFT is k-sparse, i.e., most of the Fourier coefficients of a signal are very small or equal to zero and only k coefficients are large, the transform runtime can be reduced to only sublinear to the signal size N.

Sublinear sparse Fourier algorithm was first proposed in [29], and since then, has been extensively studied in many application fields [28, 21, 20, 22, 27, 19]. All these algorithms have runtimes faster than original FFT for sparse signals. However, their runtimes have large exponents (larger than 3) in the polynomial of k and $logN$, and their complex algorithmic structures impose restrictions on fast and parallel implementations.

A highly influential recent work [24] presented an improved algorithm with the complexity of $O(k\sqrt{NlogN}logN)$ to make it faster than FFT for k up to $O(\sqrt{N/logN})$. The work in [25] came up with an algorithm with runtime $O(klogNlog(N/k))$ or even optimal $O(klogN)$. These two approaches, however, only computed a correct sparse Fourier transform in a certain probability and therefore cannot guarantee to generate a completely accurate output. Basically, the new algorithm permutes input with random parameters in time domain to approximate expected permutation in spectral domain for subsequent binning of large coefficients. The probability has to be bounded to prevent large coefficients being binned into the same bucket. In addition, these algorithms iterate over passes of estimating coefficients, updating the signal and recursing on the reminder. Because dependency exists between consecutive iterations, the algorithm cannot be fully parallelized across different iterations. Moreover, the selection of the permuting probability, or the filter, is oblivious to input characteristics.

In this dissertation, we address these limitations by proposing a new sublinear algorithm for sparse fast Fourier transform. Our algorithm has a quite simple structure and leads to a low big-Oh constant in runtime. Our sparse FFT algorithm works in the

context that the sparse FFT is invoked on a stream of input signals, and neighboring inputs have very similar spectrum distribution including the sparsity k. The assumption is true for many real-world applications, for example, for many video/audio applications, where neighboring frames have almost identical spectral representations in the locations of large Fourier coefficients, and only differing in the coefficients' values. Our main idea is to use the output of the previous FFT, i.e., the spectral representation of the previous input, as a template to decide, for the current input signal, how to most efficiently bin large Fourier coefficients into a small number of buckets, and each bucket is aimed to have only one large coefficient whose location and magnitude can be then determined. In particular, an n-dimensional filter D that is concentrated both in time and frequency [24, 25] is utilized for binning and to ensure the runtime to be sublinear to N. What binning does is essentially to convolute a permuted input signal with the selected filter in spectral domain. During the binning, each bucket receives only the frequencies in a narrow range corresponding to the length of pass region of the filter D's spectrum, and pass regions of different buckets are disjoint. The prerequisite of having such a pass region had only one large coefficient is to make all adjacent coefficients have equal distance. The information of likely coefficient locations used in the filter tuning is derived from the sparsity template. We make use of a hash table structure to directly permute coefficients in spectral domain to achieve the expected equal distanced permutation. Figure 5.1 shows the example of our hash function based permutation in spectral domain, where f_i denotes non-zero Fourier coefficients and the numbers shown above represent locations of the coefficients.

Note that we do not permute input in time domain to approximate the equal distanced permutation with a certain probability bound, but rather directly determine the expected permutation in spectral domain. And in addition each bucket certainly bins only one large coefficient. Therefore our sparse FFT algorithm is always capable of producing a determinatively correct output. Subsequently, once each bucket bins only one large coefficient, we also need to identify its magnitudes and locations. Instead of recovering the isolated coefficients using linear phase estimation [25], we easily look up

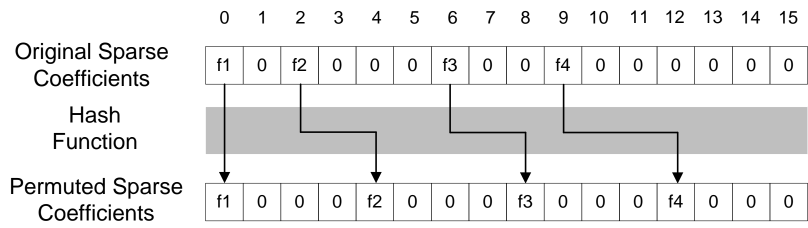


Figure 5.1: Hashing Based Permutation.

the hash table reversely to identify binned coefficients. As a result, our algorithm has the runtime at most $O(k^2 logN)$.

Additionally, if all the distances of adjacent frequencies are larger than the minimum length of filter's pass region which is obtained from empirical search, we can reduce the number of permutations and therefore further improve our algorithm to $O(klogNlog(klogN))$.

Our algorithm is evaluated empirically against FFTW, an efficient implementation of FFT with a runtime $O(NlogN)$. Overall, our algorithm is faster than FFT both in theory and implementation, and the range of sparsity parameter k that our approach can outperform dense FFT is larger than that of other sparse Fourier algorithms.

Finally, our algorithm is demonstrated to be adaptive to input characteristics. In our evaluation, we use frames from a video camera recording the movement of an object. At the beginning, we capture a video frame of that object at initial time slot T_0 and utilize our sparse FFT algorithm to generate an output. Then we can use the information of that output to help efficiently calculate sparse FFT outputs in subsequent time slots. As a result, our sparse Fourier Transform algorithm saves much time to do the image/video processing and compression. Figure 5.2 shows the example of application for our adaptive sparse FFT algorithm.

In the following sections, we will describe our methods and their applications in more detail.

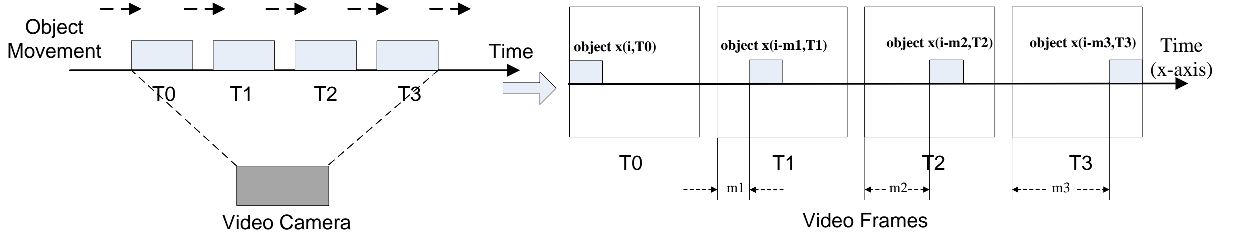


Figure 5.2: Application of Our Input Adaptive Sparse FFT Algorithm.

5.2 Input Adaptive Sparse FFT Algorithm

In this section, we use several versions of the sparse FFT algorithm to explain the evolution from a general sparse FFT algorithm to the proposed input-adaptive sparse FFT algorithm. We first describe a general input adaptive sparse FFT algorithm which comprises of input permutation, filtering non-zero coefficients, subsampling FFT and recovery of locations and magnitudes. Subsequently, we discuss how to save the number of permutations and propose an alternatively optimized version for our sparse FFT algorithm to gain runtime improvement. Moreover, general and optimized version are hybridized such that we're able to choose a specific version according to input characteristics. Finally, an example of real world application is introduced to our input adaptive approach.

5.2.1 General Input-Adaptive Sparse FFT Algorithm

5.2.1.1 Notations and Assumptions

For a time-domain input signal x with size N (assuming N is an integer power of 2), its DFT is $\hat{x}$. The sparsity parameter of input, k, is defined as the number of non-zero Fourier coefficients in $\hat{x}$. In addition, $[q]$ refers to the set of indices $\{0, ..., q-1\}$. $supp(x)$ refers to the support of vector x, i.e. the set of non-zero coordinates, and $|supp(x)|$ denotes the number of non-zero coordinates of x. Finally, our sparse FFT algorithm works by assuming the locations loc_j of non-zero Fourier coefficients can be estimated from similar prior inputs, where $j \in [k]$. The location template is

computed only once for a sequence of signal frames that are similar to each other. The Fourier template including the locations' information can be calculated from general fast Fourier transform or sparse fast Fourier transform. Therefore, our algorithm is able to compute sparse Fourier transforms for the extracted time-shifting objects within the frames that have homogeneity in the scenes and spectrums, but when we find that homogeneity is broken, our algorithm re-calculates the template and restarts the input-adaptation.

5.2.1.2 Hashing Permutation of Spectrum

The general sparse FFT algorithm starts with binning large Fourier coefficients into a small number of buckets by convoluting a permuted input signal with a well-selected filter in spectral domain. To guarantee that each bucket is to receive only one large coefficient such that its location and magnitude can be accurately estimated, we need to permute large adjacent coefficients of input spectrum to be equidistant. Knowing the possible Fourier locations loc_j and their order $j \in [k]$ from template, we make use of a hash table to map spectral coefficients into equal distanced positions.

Definition 1 *Define a hash function H: $idx = H(j) = j \times N/k$, where idx is index of permuted Fourier coefficients and $j \in [k]$.*

Next we want to determine the shifting distance s between each original location loc and its permuted position idx to be $s_j = idx_j - loc_j, j \in [k]$. Since shifting one time moves all non-zero Fourier coefficients with a constant factor, so in worst case, it will only make one Fourier coefficient be permuted into the right equidistant location. In addition, since we have total k non-zero coefficients that need to be permuted, therefore, at most k-time shiftings have to be performed to permute all the coefficients into their equal distanced positions.

Moreover, the shifting factors obtained in spectral space should be translated into correspondent operations in time domain so that they are able to take effect with input signal $x_i, i \in [N]$. In effect, shifted spectrum $\hat{x}_{loc-s}$ is equivalently represented as $x_i \omega^{si}$ in time domain, where $\omega = e^{b2\pi/N}$ is a primitive n-th root of unity and $b = \sqrt{-1}$.

Definition 2 *Define the permutation* $P_{s(j)}$ *as* $(P_{s(j)}x)_i = x_i \omega^{is(j)}$ *therefore* $\hat{P_{s(j)}x}_i = \hat{x}(loc_j - s(j))$, *where* $s(j)$ *is the factor of* j*-th shifting.*

Therefore, each time when we change the factor $s(j)$, the permutation allows us to correctly bin large coefficient at location loc_j into the bucket. The length of bucket is determined by the flat window function designed in the next section.

5.2.1.3 Flat Window Functions

In this dissertation, the method of constructing a flat window function is same as that used in the sFFT's work [24]. The concept of flat window function is derived from standard window function in digital signal processing. Since window function works as a filter to bin non-zero Fourier coefficients into a small number of buckets, the pass region of filter is expected to be as flat as possible. Therefore, our filter is constructed by having a standard window function convoluted with a box-car filter [24]. Moreover, we want the filter to have a good performance by making it to have fast attenuation in stopband.

Definition 3 *Define* $D(k, \delta, \alpha)$, *where* $k >= 1$, $\delta > 0$, $\alpha > 0$, *to be a flat window function that satisfies:*

1. $|supp(D)| = O(\frac{k}{\alpha} log(\frac{1}{\delta}))$;
2. $\hat{D}_i \in [0, 1]$ *for all* i;
3. $\hat{D}_i \in [1 - \delta, 1 + \delta]$ *for all* $|i| \leq \frac{(1-\alpha)N}{2k}$;
4. $\hat{D}_i < \delta$ *for all* $|i| \geq \frac{N}{2k}$;

In particular, flat window function acts as a filter to extract a certain set of elements of input x. Even if the filter consists of N elements, most of the elements in the filter are negligible and there are only $O(\frac{k}{\alpha} log(\frac{1}{\delta}))$ significant elements when multiplying with x in time domain. In addition, the flat window functions are precomputed in our implementation to save execution time, since their constructions are not dependent on input x but only dependent on N and k. We can lookup each value of the window function in constant time.

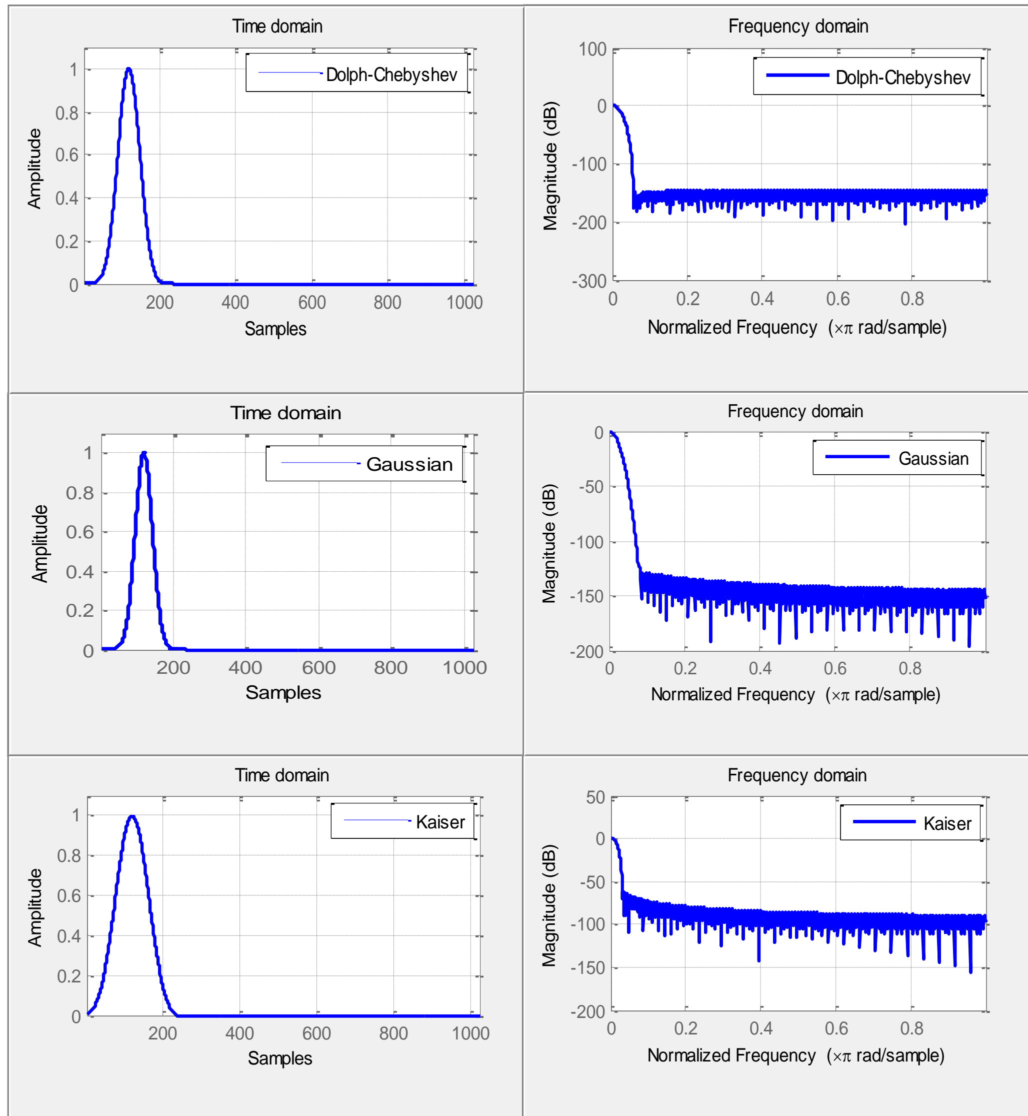


Figure 5.3: An Example of Dolph Chebyshev, Gaussian, Kaiser Flat Window Function with $N = 1024$.

Figure 5.3 shows an example of Gaussian, Kaiser and Dolph-Chebyshev flat window functions. Note that the spectrum of our filters D is nearly flat along the pass region and has an exponential tail outside it. It means that leakage from frequencies in other buckets can be negligible. By comparing the properties of the three window functions, Dolph-Chebyshev window is an optimal one for us to use due to its flat pass region as well as quick and deep attenuation in stopband.

5.2.1.4 Subsampled FFT

The coefficients binning process is to convolute input spectrum with flat window function. In actual, this convolution is instead performed in time domain by first multiplying input with filter and then computing its subsampled FFT. Suppose we have one N-dimensional complex input series x with sparsity parameter k for its Fourier coefficients, we define a subsampled FFT as $\hat{y}_i = \hat{x}_{iN/k}$ where $i \in [k]$ and N can be divisible by k. The FFT subsampling expects the locations of Fourier coefficients in spectrum domain have been equally spaced. The proof of k-dimensional subsampled FFT has been shown in [24] and the time cost is in $O(|supp(x)| + klogk)$.

5.2.1.5 Reverse Hash Function for Location Recovery

After subsampling and FFT to the permuted signal, the binned coefficients have to be reconstructed. This is done by computing the reverse hash function H_r.

Definition 4 *Define a reverse hash function* H_r: $rec = H_r(idx) = \frac{idx}{(N/k)}$, *where* idx *is index of permuted Fourier coefficients and* rec *is the order of recovered coefficients.*

Therefore, recovery of Fourier locations can be estimated as loc_{rec} by fetching the locations using the reconstructed order of frequencies.

5.2.1.6 Algorithm Description

Combining the aforementioned steps, we can describe our sparse FFT algorithm as following. Note that up to this point, we have not introduced input adaptability,

yet. Assuming we have a Fourier location template with k known Fourier locations loc and a precomputed filter D,

1. For $j = 0, 1, 2, ..., k-1$, where $j \in [k]$, compute hash indices $idx_j = H(j)$ of permuted coefficients, and determine shifting factor $s_j = idx_j - loc_j$.

2. Compute $y = D \cdot P_s(x)$, therefore $|supp(y)| = |s| \times |supp(D)| = O(|s| \frac{k}{\alpha} log(\frac{1}{\delta}))$. We set $\delta = \frac{1}{4N^2V}$, where V is the upperbound value of Fourier coefficients and $V \leq N$.

3. Compute $u_i = \sum_{l=0}^{\frac{|supp(y)|}{k}-1} y_{i+i|y|+lk}$ where $i \in [k]$.

4. Compute k-dimensional subsampled FFT $\hat{u}_i$ and make $\hat{z}_{idx} = \hat{u}_i$, where $i \in [k]$.

5. Location recovery for $\hat{z}_{idx}$ by computing reverse hash function to produce $rec = H_r(idx)$ and finally output $\hat{z}_{loc(rec)}$.

5.2.1.7 The Computational Complexity

We analyze the runtime of our general sparse FFT algorithm: Step 1 costs $O(k)$; step 2 and 3 cost $O(|s| \frac{k}{\alpha} log(\frac{1}{\delta}))$; step 4 costs $O(klogk)$ for a k-points FFT; step 5 costs $O(k)$. Therefore total running time is determined by $O(|s| \frac{k}{\alpha} log(\frac{1}{\delta}))$. It is very rarely that initial Fourier coefficients have equidistant locations, therefore $|s|$ equals to $|k|$ in general and the runtime becomes $O(\frac{k^2}{\alpha} log(\frac{1}{\delta}))$ which is asymptotic to $O(k^2 logN)$.

5.2.2 Optimized Input-Adaptive Sparse FFT Algorithm

In this section we introduce several transformations of our algorithm that improve performance and facilitate parallelization. The complexity of general adaptive sparse Fourier algorithm is asymptotic to $O(|s| \frac{k}{\alpha} log(\frac{1}{\delta}))$ if initially no adjacent Fourier coefficients are equally distanced. However, if the number of permutations can be reduced, then $|s|$ will be decreased. In fact, it is unnecessary to permute all the Fourier locations to make them equidistant between each other. Since binning the sparse Fourier coefficients is a process of convoluting permuted input spectrum with a well designed filter, so it is guaranteed that if length of filter's pass region ϵ is less than or

equal to half of the shortest distance $dist_{min}$ among all the adjacent locations of non-zero coefficients, i.e. $\epsilon <= dist_{min}/2$, then we don't need to permute all coefficients before we do a FFT. Moreover, in this way, we can get rid of aliasing distortions during the binning and each pass region essentially receives only one large coefficient. If we do not do this, aliasing error occurs and we have to permute all spectral samples.

Next we continue to apply the flat window function D to compute filtered vector $y = Dx$ and then we want to compute a FFT for y to produce final output $\hat{y}$. The form of FFT we use here is not a k-dimensional subsampled FFT described previously, since the subsampled FFT requires that locations of non-zero Fourier coefficients are permuted to be equidistant. Instead, we apply a general FFT subroutine into calculation of $\hat{y}$. The size of the FFT is dependent on the length of non-zero elements in y, which is $O(\frac{k}{\alpha}log(\frac{1}{\delta}))$ determined by non-zero region of window function D. We view the size of this FFT as a region with length $O(\frac{k}{\alpha}log(\frac{1}{\delta}))$ (i.e. $O(klogN)$) truncated from size N. Total number of such truncated region is $\frac{N}{klogN}$. In addition, since k sparse Fourier coefficients are distributed in a region consisting of N elements, we have to identify whether output of $O(\frac{k}{\alpha}log(\frac{1}{\delta}))$-dimensional FFT contains all non-zero Fourier coefficients. If not, we would like to shift the unevaluated non-zero coefficient into the truncated region. Our algorithms determines whether to do the shifting before computing FFT. Since the locations of non-zero coefficients and length of truncated region are known from template, we compare the locations with boundary of truncated region to determine the shifting factor sf.

5.2.2.1 Input-Adaptive Shifting

There are two ways to do shifting:

1. If $k <= \frac{N}{klogN}$, we shift the first unevaluated non-zero coefficient into the truncated region each time;

2. If $\frac{N}{klogN} < k$, we shift the unevaluated non-zero coefficient by a constant factor $klogN$ each time;

In the worst case, the first method performs shifting at most $O(k)$ times, while the second version takes time at most $O(\frac{N}{klogN})$. However, if all large coefficients reside in only one truncated region, we need no shifting and hence we obtain the best case. Meanwhile, the shifting sf_i to spectral coefficients, i.e. $\hat{y}_{i+sf_i}$ corresponds to time domain operation by multiplying input signal y_n with a twiddle factor, i.e. $y_n e^{-b2\pi sf_i n/N}$ where $b = \sqrt{-1}$. Therefore, the cost of shifting for one time is the length of filtered vector y, i.e. $O(klogN)$.

5.2.2.2 Optimized Algorithm

Adding the optimization heuristics and the input-adaptive shifting, the improved sparse FFT algorithm works as following:

1. Apply filter to input signal x:

Utilize a flat window function D to compute the filtered vector $y = Dx$. Time cost RT_1 is $O(\frac{k}{\alpha}log(\frac{1}{\delta}))$, i.e. $O(klogN)$.

2. Spectrum shifting: Compare k and $\frac{N}{klogN}$ to select one of the two shifting methods and then do the shifting to filtered vector y. The step-2's runtime RT_2 is $O(klogN) \leq RT_2 < O(min\{k, \frac{N}{klogN}\}\frac{k}{\alpha}log(\frac{1}{\delta}))$, i.e. $O(klogN) \leq RT_2 < O(min\{k, \frac{N}{klogN}\}klogN)$.

3. For $e \in \{1, 2, ..., min\{k, \frac{N}{klogN}\}\}$, each shifting event I_e is to compute $O(\frac{k}{\alpha}log(\frac{1}{\delta}))$-dimensional (i.e. $O(klogN)$-dimensional) FFT $\hat{z}_e$ as $\hat{z}_{e,i} = \hat{y}_i$ in current truncated region, for $i \in [O(\frac{k}{\alpha}log(\frac{1}{\delta})) = O(klogN)]$. Final output is $\hat{z}$. The step-3's runtime RT_3 is $O(klogNlog(klogN)) \leq RT_3 < O(min\{k, \frac{N}{klogN}\}klogNlog(klogN))$.

Therefore, total runtime RT of the improved sparse FFT algorithm is
$O(klogNlog(klogN)) \leq RT < O(min\{k, \frac{N}{klogN}\}klogNlog(klogN))$.

5.2.3 Hybrid Input-Adaptive Sparse FFT Algorithm

Actually, it is clear from the complexity analysis of our general and optimized sparse FFT algorithms that the two algorithm versions are best suit for different input characteristics. That is, the "optimized" version does not perform better than the

general version on all cases. We hybridize the two approaches by at runtime selecting the most appropriate version based on input characteristics.

In our optimized version of sparse FFT algorithm, it is worth mentioning that if the required length of pass region is too short, such a filter becomes hard to construct in practice. Therefore, we define a threshold $dist_{TD}$ of minimum distance $dist_{min}$. If $dist_{min} >= dist_{TD}$, then the filter can be constructed to have expected pass region. If $dist_{min} < dist_{TD}$, then our general sparse FFT has to be applied and all the Fourier locations have to be permuted to be equidistant. The threshold can be obtained by empirical search offline.

Therefore, we make the following judgment on an input to decide which algorithm version to apply for the specific input:

1. Determine shortest distance $dist_{min}$ among all adjacent locations of k large coefficients:

Initialize minimum distance $dist_{min} = 0$; For $j \in 1, 2, ..., k-1$, compute distances $dist_j = loc_j - loc_{j-1}$ between all k adjacent sparse Fourier locations loc_{j-1} and loc_j; Then if $dist_j <= dist_{min}$, update $dist_{min} = dist_j$. The runtime is $O(k)$.

2. If $dist_{min} >= dist_{TD}$, we choose to use optimized approach to save large number of permutations; If $dist_{min} < dist_{TD}$, then our general sparse FFT has to be applied and all the Fourier locations have to be permuted to be equidistant. The threshold can be obtained by empirical search in our filter design process.

This resolution assists us to create an input-aware algorithm for sparse FFT computation. The cost for the deciding process is only $O(k)$, which can be neglected compared with the runtime of either the general version or the optimized version.

5.2.4 Real World Application

In this section, we demonstrate how our input adaptive algorithm works in real world. Meanwhile, we illustrate how the Fourier location template is generated. We use a sample of video recording in real application shown in figure 5.2. The video sample uses a fix video camera to record the movement of a 2D object along x-coordinate for

a duration of time. For each time slot we obtain a 2D video frame containing the object image which can be represented as a 2D matrix $img(g, h)$ whose values stand for color digits, where $g \in [ro]$, $h \in [col]$. The number of rows and columns is ro and col, respectively. Particularly, we substitute the 2D matrix into a row-major 1D array $x_i = x(i = g * col + h) = img(g, h)$. Assuming the interval between the same object in two time-adjacent video frames is m, it can be proved that shifting to $img(g, h)$ is the same to x_i since $img(g, h - m) = x(g * col + h - m) = x_{i-m}$. Therefore, the process of video recording is modeled as a time shifting process to x_i and we want to compute its Fourier transform $\hat{x}_j$ for image/video processing and compression.

At the beginning, we capture input signal $x_{i,T0}$ in a video frame at initial time slot T_0 and calculate its Fourier transform $\hat{x}_{j,T0}$ using a dense FFT. All locations of large Fourier coefficients and their order can be obtained for $\hat{x}_i$ at T_0. Next we need to compute Fourier transform for x_{i-m_1} at time T_1. Since time-shifted x_{i-m_1} corresponds to $\hat{x}_j e^{-b2\pi m_1 j/N}$ in spectral domain, where $b = \sqrt{-1}$, hence the locations of non-zero frequencies in $\hat{x}_{j,T1}$ is same as those in $\hat{x}_{j,T0}$, but only their values differ. As a consequence, we can make use of Fourier locations gained from $x_{i,T0}$ to compute sparse (not dense) FFT for $x_{i-m_1,T1}$ at T_1 and for $x_{i-m_t,Tt}$ at remaining time slots T_t. Therefore, our sparse algorithm saves much time on dense FFTs since we only compute dense FFT once and then only calculate sparse FFTs according to input characteristics we obtained previously.

Moreover, if time shifting factor m_t is known, we can further directly multiply $\hat{x}_{j,T0}$ at initial time T_0 by $e^{-b2\pi m_t j/N}$ to efficiently attain Fourier transform $\hat{x}_{j,Tt}$ at remaining time T_t without a FFT. However, if shifting factor m_t is unknown, we cannot do this to get spectrums for $x_{i-m_t,Tt}$. This situation is feasible in real application. Suppose we use a video recorder to capture several video frames, but sometimes we don't know the time-shifted distance m_t of the two frames. Hence, we have to know m at first. The worst case is to match $x_{i,T0}$ with $x_{i-m_t,Tt}$ and determine m_t in runtime of $O(N^2)$. Nonetheless, such a process can be efficiently executed in $O(N)$ time when applying the algorithm in [30]. Therefore, if m_t is unknown, we spend time of $O(N)$

Table 5.1: Experimental Configurations.

CPU	**Frequency**	**System Memory**	**Cache**
Intel Core i7-920	2.66GHz	24GB	8192KB

on finding m_t and $O(k)$ on multiplying $e^{-b2\pi m_t j/N}$. Total runtime is $O(N+k)$. In the evaluation section, we conduct detailed evaluation to show our sparse FFT outperforms the performance of above two situations including known m_t and unknown m_t.

5.3 Experimental Evaluation

In this section we evaluate our input-adaptive sparse FFT implementation and its influence on a real-world application. We first discuss the environmental setup that we use for the evaluation and then present performance results.

The double-precision sequential implementation and the performance evaluation are conducted on a high performance computer with an Intel Core i7-920 CPU. The hardware configurations are summarized in table 5.1. For sequential version, we evaluate our general and optimized sparse FFT approaches, and compare them against three highly-influential FFT libraries: 1) FFTW 3.3.3 [15], the latest FFTW which is the most efficient implementation for computing the dense FFTs. In FFTW, Streaming Single Instruction Multiple Data Extensions (SSE) on Intel CPU is enabled for better performance. 2) sFFT 1.0 and 2.0 [24], which is one of the fastest sublinear algorithms of sparse FFT with an open source library. 3) AAFFT 0.9 [26], which is another recent sublinear algorithm with fast empirical runtime. Furthermore, all FFTW libraries we use are with two flags, i.e. ESTIMATE (a basic version marked as 'FFTW' in the plots) and MEASURE (an optimal version marked as 'FFTW OPT' in the plots).

5.3.1 General Input-Adaptive Sparse FFT Algorithm

All versions of our general sparse FFT are evaluated in two cases: First, we fix the sparsity parameter $k = 64$ and plot the execution time of our algorithm and the compared libraries for 18 different signal sizes from $N = 2^{10}$ to 2^{27}. Second, we fix

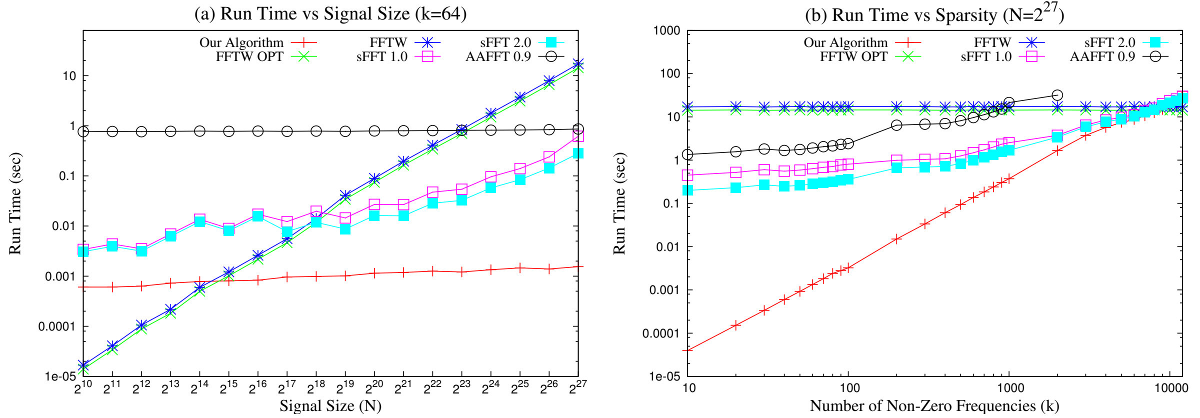


Figure 5.4: Performance of Our General Sparse FFT in Sequential Case.

the signal size to $N = 2^{27}$ and evaluate the running time under different numbers of non-zero frequencies, i.e. k.

Figure 5.4 shows our sequential sparse FFT on an Intel i7 CPU. In figure 5.4(a), we fix $k = 64$ and change N. The running time of FFTW is linear in the signal size N and sFFT 1.0/2.0 shows approximately linear in N when $N > 2^{20}$. However, our general sparse FFT appears almost constant as the signal size increases, which is a result of our sub-linear property in algorithm. AAFFT 0.9 also shows constantly over different N but its runtime performance is worse than ours and sFFT. Moreover, our approach demonstrates the fastest runtime over sFFT, FFTW and AAFFT. For $N \geq 2^{15}$, our algorithm is faster than FFTW, while sFFT and AAFFT has to reach this goal for $N \geq 2^{19}$ and $N \geq 2^{24}$, respectively. In figure 5.4(b), we fix $N = 2^{27}$ and change k. FFTW shows invariance in performance since its complexity is $O(NlogN)$ which is independent on k. Additionally, our general sparse FFT has a faster runtime than basic and optimal FFTW for k up to 11000 and 10000, respectively. However, sFFT 1.0, sFFT 2.0 and AAFFT 0.9 are faster than basic FFTW only when k is less than 8000, 9000 and 1000. Therefore, our approach extends the range of k where our performance is faster than dense FFT. Furthermore, our general algorithm performs better than other compared FFT libraries on average.

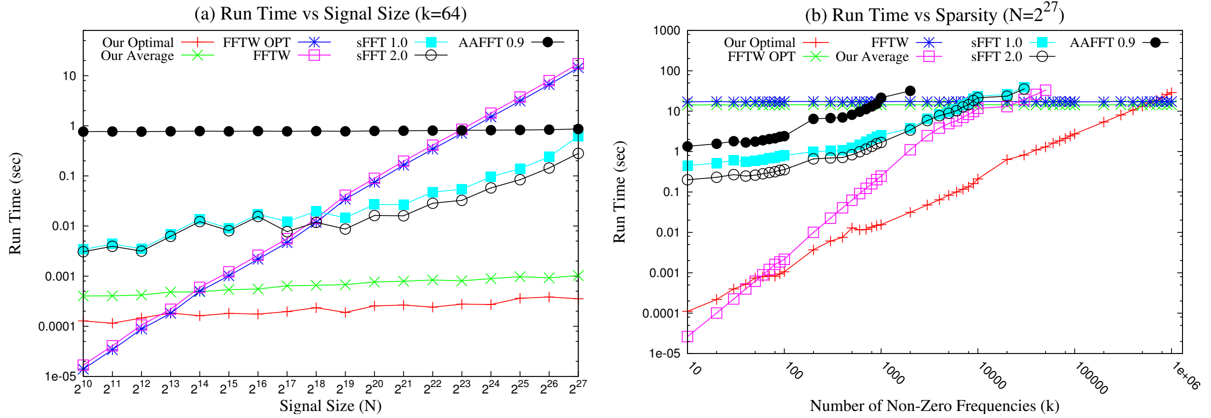


Figure 5.5: Performance of Our Optimized Sparse FFT in Sequential Case.

5.3.2 Optimized Input-Adaptive Sparse FFT Algorithm

Our optimized sparse FFT algorithm has two situations: the optimal status assumes that all large coefficients reside in only one truncated region of length $O(klogN)$ so that we need no shifting; the average case neglects this assumption but to compute for a random input over 10 runs then takes an average.

Figure 5.5 shows our optimized sparse FFT in sequential case. In figure 5.5(a), we fix $k = 64$ and change N. Our optimized approach is sub-linear due to its constant curve when N increases. In addition, the optimal and average case of our optimized algorithm is faster than FFTW when $N \geq 2^{14}$ and $N \geq 2^{15}$, respectively. However, sFFT 1.0/2.0 and AAFFT 0.9 has to achieve this purpose for $N \geq 2^{19}$ and $N \geq 2^{24}$, respectively. In figure 5.5(b), we fix $N = 2^{27}$ and change k. Our optimal and average case has a runtime faster than FFTW for k up to 1000000 and 25000, respectively. However, sFFT 1.0, sFFT 2.0 and AAFFT 0.9 are faster than basic FFTW only when k is less than 8000, 9000 and 1000. On average, our optimized algorithm performs better than other compared FFT libraries.

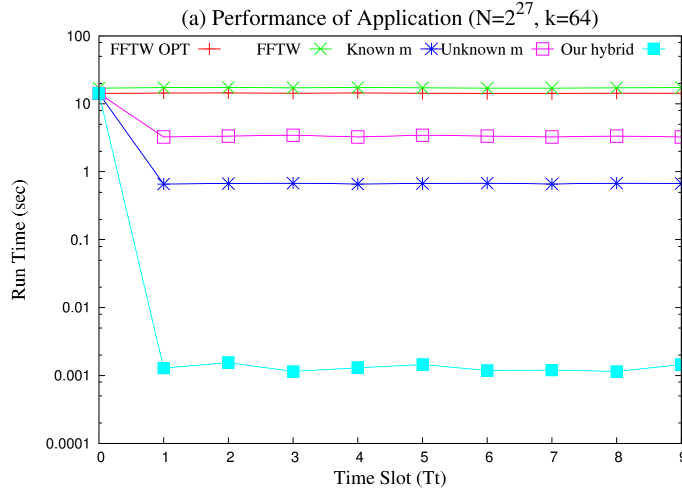


Figure 5.6: Performance of Our Algorithm in a Real-world Application.

5.3.3 Evaluation of Real World Application

We show an performance evaluation for better illustrating the real-world application of video recording in the section above. Suppose we capture total 10 video frames in 10 time slots. The displacement between the object in adjacent two time slots is set to 2^{10} points. Figure 5.6 shows the sequential performance of our hybridized sparse FFT against to the dense FFT performance, i.e. FFTW, and to the performance of two situations including known shifting factor m and unknown m. X-axis represents the time slots T_t. Input signal size is $N = 2^{27}$ points and $k = 64$. The test shows that our hybridized sparse FFT in sequential case outperform the performance of all other compared situations. It demonstrates that we can spend time to compute a dense FFT once to preprocess the Fourier location template that we need for the FFTs in remaining time. Then we can save much time by using our hybrid sparse FFT for all the subsequent input signals. Furthermore, if the number of frames is large, our sparse FFT outperforms sFFT as well as AAFFT on average in the real application.

5.4 Chapter Summary

In this chapter, we proposed an efficient input-adaptive sparse FFT algorithm that takes advantage of the similarity between sparse input samples to efficiently compute a Fourier transform in the runtime sublinear to signal size N. Specifically, our work integrates and tunes several adaptive filters to package non-zero Fourier coefficients into sparse bins which can be estimated accurately. Moreover, our algorithm is non-iterative with high computation intensity. Overall, our algorithm is faster than FFT both in theory and implementation, and the range of sparsity parameter k that our approach can outperform dense FFT is larger than that of other sparse Fourier algorithms.

Chapter 6

PARALLEL INPUT-ADAPTIVE SPARSE FAST FOURIER TRANSFORM FOR STREAM PROCESSING

In this chapter, we improve upon existing sequential sparse FFT algorithms by proposing a new parallel input adaptive sparse FFT algorithm to break down the dependency in the recursive coefficient packaging and estimation of traditional sequential sparse FFTs. It is the first time to propose a highly parallel version of sparse FFT and to exploit substantial parallelism from tradition algorithms. Additionally, due to the fact that in stream processing field, the signal inputs are frequently "sparse", i.e., most of the inputs' Fourier coefficients being zero, our parallel input adaptive sparse FFT is well applicable, efficient and robust in the input adaption process of stream processing. Specifically, our input adaptive algorithm can automatically detect and take advantage of the similarity between adjacent continuous inputs to accelerate computation. When a discontinuity occurs, our discontinuity detection method can automatically detect the discontinuities inside the streams and resumes the continuous input adaptation very efficiently.

6.1 Overview of Sequential Sparse FFTs and Our Parallel Input Adaptive Sparse FFT Approach

In this section, we go over currently exsiting sequential versions of sparse FFTs to explain the evolution from a traditional sequential sparse FFT algorithm to our proposed parallel input-adaptive sparse FFT algorithm.

Fast Fourier Transform (FFT) is frequently invoked in stream processing, e.g., calculating the spectral representation of audio/video frames, and in many cases the inputs are "sparse", i.e., most of the inputs' Fourier coefficients being zero. Compared

to the very expensive "dense" FFT computation, which requires a large amount of computing resources and memory bandwidth to compute, many "sparse" FFT algorithms have been proposed to improve FFT's efficiency when inputs are known to be sparse. To develop an efficient sparse FFT become a important trend in parallel computing and HPC field.

However, the development of current sparse FFT algorithms is immature and is still under research. The existing sparse FFT implementations [29, 28, 21, 20, 22, 27, 19], even for the nearly optimal sparse FFTs [24, 25], are limited to be sequential version algorithm. The reason is that it is hard to break down the dependency within the coefficient binning and estimation process, and hence it is difficult to exploit parallelism. There is no fully parallel version of sparse FFT now. Moreover, like their "dense" counterparts, existing sparse FFT implementations are input oblivious in the sense that how the algorithms work is not affected by the value of input. The sparse FFT computation on one frame is exactly the same as the computation on the next frame.

This chapter improves upon existing sparse FFT algorithms by proposing a new parallel algorithm to break down the dependency in the recursive coefficient packaging and estimation of traditional sequential sparse FFT. It is the first time to propose a fully parallel version of sparse FFT and to exploit substantial parallelism from tradition algorithms. Meanwhile, our proposed algorithm is able to simultaneously exploit the input sparsity and the similarity between adjacent inputs in stream processing. Additionally, our algorithm detects and takes advantage of the similarity between input samples to automatically design and customize sparse fourier template that lead to better parallelism and performance. Specifically, given a sparse signal that has only k non-zero Fourier coefficients, our algorithm is able to directly permute coefficients in spectral domain to replace the time domain approximation of tradition algorithms. So it automatically gets rid of the dependency and efficiently packages the non-zero Fourier coefficients into a small number of bins which can then be estimated accurately. Therefore, our algorithm has runtime sub-linear to the input size and exploit much data parallelism and concurrency, both of which greatly improve the performance.

Furthermore, we consider to make our sparse FFT algorithm adaptive to various input streams. For the continuous inputs, we develop an efficient heuristic to detect the similarity between the current input to its predecessor in stream processing, and when it is found to be similar, we novelly use the spectral representation of the predecessor to accelerate the sparse FFT computation on the current input. Additionally, if there is a discontinuity occurring after a segment of continuous input streams, we also propose and design a new heuristic of discontinuity detection process in our parallel sparse FFT, that can automatically detect the discontinuities inside the streams and resumes the continuous input adaptation very efficiently.

Finally, we show how the performance of our implementation can be parallelized for GPU and multi-core CPU. We evaluate our parallel input-adaptive sparse FFT implementation on Intel Core i7 CPUs and three NVIDIA GPUs, i.e., NVIDIA GeForce GTX480, Tesla C2070 and Tesla C2075. Our parallel sparse FFT algorithm is much faster than previous FFTs both in theory and implementation. For inputs with size $N = 2^{24}$, our parallel implementation outperforms FFTW for k up to 2^{18}, which is an order of magnitude higher than prior sparse algorithms. Furthermore, our parallel input adaptive sparse FFT on Tesla C2075 GPU achieves up to 77.2× and 29.3× speedups over 1-thread and 4-thread FFTW, 10.7×, 6.4×, 5.2× speedups against sFFT 1.0, sFFT 2.0, CUFFT, and 6.9× speedup over our sequential CPU performance, respectively.

6.2 Parallel Input-Adaptive Sparse FFT Algorithm

Compared with the "dense" FFT algorithms or the existing sparse FFT algorithms, our input-adaptive sparse FFT algorithm can be better parallelized. Specifically, our algorithm is non-iterative with high arithmetic intensity in most portions. The non-iterative nature exposes good coarse-grain parallelism. Moreover, data parallelism of each section can be exploited from the algorithm. In this work, we use Graphic Processing Units (GPUs) for the well-suited data parallel computations. Several architectural-oriented transformations are applied to fine-tune the algorithm for

the GPU architecture.

6.2.1 General Description of Parallel Input Adaptive Sparse FFT

Our parallel input adaptive sparse FFT is to do the parallelization based on our sequential input adaptive sparse FFT algorithm which was introduced in the above chapter 5. To help understand the role of spectral template, figure 6.1 illustrates the binning process in our algorithm. Large Fourier coefficients are binned into a small number of buckets and each bucket is designed to have only one large coefficient whose location and magnitude can be then determined. The bucket is represented by an n-dimensional filter D, that is concentrated both in time and frequency, to ensure the runtime to be sublinear to N. The filter design is described in our sequential input adaptive sparse FFT algorithm introduced in the chapter above. What binning does is essentially to convolute a permuted input signal with a well-selected filter in spectral domain. During the binning, each bucket receives only the frequencies in a narrow range corresponding to the length of filter D's pass region, and pass regions of different buckets are disjoint. The prerequisite of a pass region having only one large coefficient is to make it possible to evenly space all adjacent coefficients in spectrum later. The information of likely coefficient locations used in the filter tuning is derived from the sparsity template. Particularly, to achieve the expected equal distanced permutation, we make use of the same hash table structure of our sequential input adaptive sparse FFT algorithm introduced in the above chapter 5, to directly permute coefficients in the spectral domain.

Due to the fact that we can directly know the exact locations of non-zero Fourier coefficients (i.e. Fourier frequencies), and hence we can directly permute and package each non-zero Fourier coefficients in spectral domain, instead of making approximation from time domain.Therefore, each non-zero frequency packaging becomes independent with each other, and they can be fully parallelized. Additionally, our parallel sparse FFT is able to break down large dependencies of the traditional frequency permutation estimation process in time domain, and hence our parallel algorithm gets rid of the

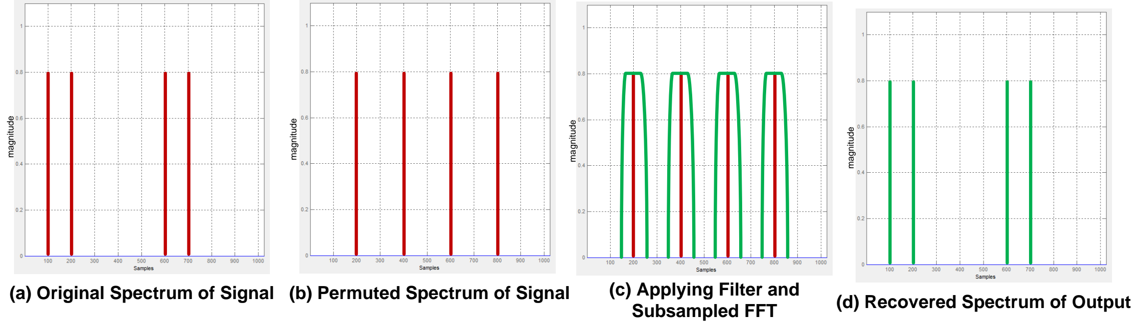


Figure 6.1: Binning of Non-zero Fourier Coefficients.

dependencies of the traditional sparse FFTs. Moreover, our algorithm does not need the packaging iterations, so our sparse FFT structure is non-iterative and has only one execution path with several sub-routines, including hashed index computation, input permuation, filtering, subsampling FFT, and coefficient recovery.

6.2.2 Parallelism Exploitation and Kernel Execution

We use the general sparse FFT implementation to demonstrate how we parallelize our input-adaptive sparse FFT algorithms. The parallelization of the optimized version is similar. Since data parallelism is a set of homogeneous tasks executed repeatedly over different data elements, we have such parallelism existing in sections of hashed index computation, filtering and input permutation, subsampling FFT, and location recovery. Therefore, to achieve high performance we construct GPU computational $kernel$ for each section.

First of all, kernel $HashFunc()$, whose number of threads is k, is responsible to compute hashed indices of permuted coefficients and to determine shift factors. The loop of size k is decomposed and each scalar thread in kernel concurrently works as each index j in the algorithm. In addition, kernel $Perm()$ with total number of threads $k^2 logN$ is used to apply filter and permutation to input. Each thread multiplies filter as well as shifting factor with input for one element. We parallelize subsampling to input in kernel $Subsample()$ with total k threads before we launch our well-tuned

FFT kernel $TunedFFT()$. Finally we obtain output from location estimation kernel $Recover()$ with k threads parallelizing the loop of algorithm.

For the parallelization of our optimized version of sparse FFT algorithm, we start to launch kernel $Filtering()$ to parallelize loop size $O(klogN)$ of applying filter to the input. Subsequently, kernel $Shifting()$ with $min\{k, \frac{N}{klogN}\}klogN$ threads is to make each thread shift one input element by a factor. For each shifting event, our tuned FFT kernel $TunedFFT()$ is launched before we gain the output.

6.2.3 Performance Optimizations

Throughout our GPU implementation to two versions algorithms, we take care of several important optimization techniques that enable GPU performance to be improved significantly.

Since GPU global memory accesses are costly, it is crucial to optimize access pattern in order to get maximum memory bandwidth. We organize memory accesses to be coalesced which indicates that threads of a half-warp (16 threads) access 16 consecutive elements at a time so that those individual accesses are grouped into a single memory access. Since in our implementation, most kernels have consecutive access patterns, therefore we enable coalesced accesses by making the size of thread block be 16×2^p where $p \geq 0$, and set grid size to $\frac{\#threads}{blocksize}$.

Moreover, data sharing between kernels can be executed efficiently by increasing data reuse inside local device memory. Host (CPU) and device (GPU) are connected through a PCIe bus that has much larger latency and smaller bandwidth than device memory. Therefore it is of great necessity to increase PCI bandwidth by reducing the number of PCI transfers and keep much more data in local device for reuse. In our implementation, we only have two transfers between CPU and GPU. The first communication is to input all precomputed data including input, Fourier locations and filter information into GPU from CPU. The second transfer is to output final sparse-Fourier results from GPU to CPU. Temporary results are kept into GPU memory and are reused between kernels without transfering back to CPU.

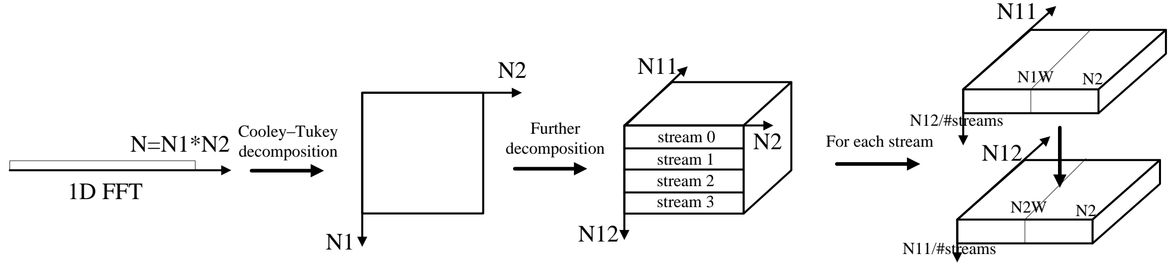


Figure 6.2: Working Flow of GPU Parallelization.

6.2.4 Tuned GPU based FFT Library

Our GPU kernel decomposes a 1D FFT of size $N = N_1 \times N_2$ into multi-dimensions N_1 and N_2. Therefore it enables the exploitation of more parallelism for parallel FFT implementation on GPU architectures. All N_1 dimensional 1D FFTs are first calculated in parallel across N_2 dimension. If the size of N_1 is still large after decomposition, we would further decompose each $N_1 = N_{11} \times N_{12}$ sized 1D FFT into two dimensional FFTs with smaller sizes N_{11} and N_{12}, respectively. On GPU, the device memory has much higher latency and lower bandwidth than the on-chip memory. Therefore, shared memory is utilized to increase device memory bandwidth. $N_1W \times N_{11} \times N_{12}$ sized shared memory needs to be allocated, where N_1W is chosen to be 16 for half-warp of threads to enable coalesced access to device memory. The number of threads in each block, for both N_{11} and N_{12}-step FFTs, is therefore $N_1W \times max(N_{11}, N_{12})$ to realize maximum data parallelism on GPU. To calculate each N_1-step 1D FFT, a size N_{11} FFT is executed to load data from global memory into shared memory for each block. Next, all threads in each block are synchronized before data in shared memory is reused by the N_{12}-step FFT and subsequently written back to global memory. Experiment tests show that such shared memory technique effectively hides global memory latency and increases data reuse, both contributing to the performance on GPU. Figure 6.2 shows the working flow of our GPU based parallelization.

6.3 Input Adaption Heuristics and Process

Our sparse FFT algorithm works efficiently in the context that the sparse FFT is invoked on a stream of input signals, and neighboring inputs have very similar spectrum distribution including the sparsity parameter k. The assumption is true for many real-world applications, for example, for many video/audio applications, where neighboring frames have almost identical spectral representations in the locations of large Fourier coefficients, and only differing in the coefficient magnitudes. Our algorithm adapts to the homogeneity in signal spectrums by utilizing the output of the previous FFT, i.e., the spectral representation of the previous input, as a template to most efficiently compute the Fourier transform for the current input signal. When the homogeneity is found to be broken, our algorithm re-calculates the template and restarts the input-adaptation. An effective heuristic is proposed in this work to detect such discontinuity in frame spectrums.

In this section, we describe our overall input adaption process. Figure 6.3 illustrates how the whole process would work in real world video stream processing. We first detail when and how Fourier location templates are generated. Then, we elaborate how our sparse FFT adapts to inputs with homogeneous and discontinuous characteristics.

6.3.1 Scenario Establishment

Assume we use a fix video camera to record the movement of a 2D object for a duration of time. Each frame of the object can be represented as a 2D matrix $img(g, h)$ whose values stand for color digits, where # of rows is ro, # of columns is col, and $g \in [ro]$, $h \in [col]$. In this work, we flatten the 2D matrix into a row-major 1D signal $x_i = x(i = g * col + h) = img(g, h)$. If the interval between the same object in two time-adjacent video frames is m in X dimension and v in Y dimension, it is clear that the shifting factor (m, v) to $img(g, h)$ is the same as x_i since $img(g - v, h - m) = x(g * col - v * col + h - m) = x_{i-v*col-m}$. Therefore, the process of video recording is

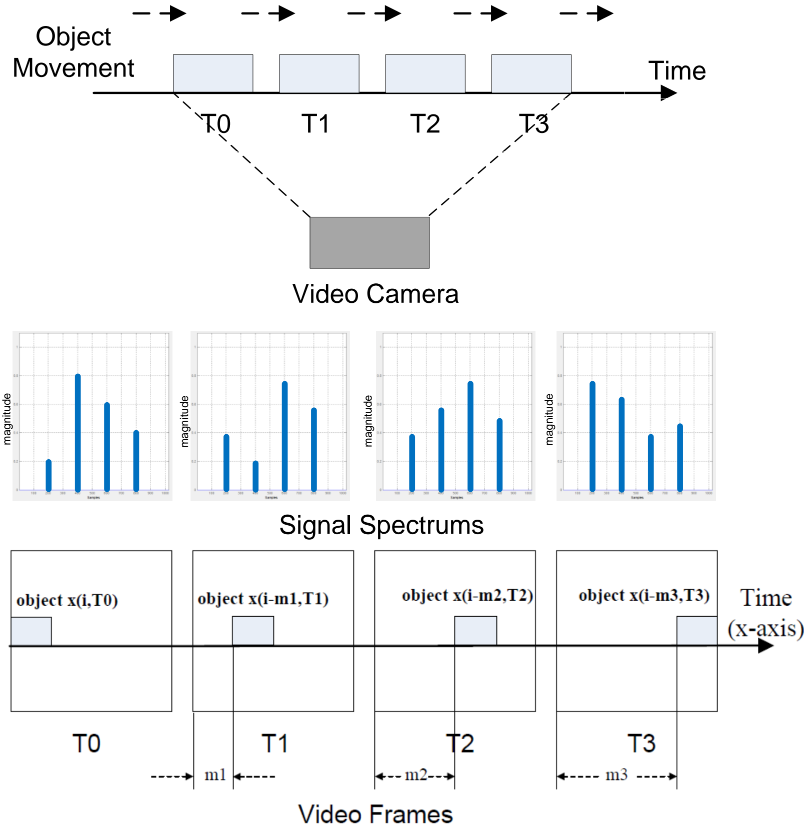


Figure 6.3: An Application Example on Video Streams.

modeled as a time shifting process to x_i, and we want to compute its Fourier transform $\hat{x}_j$.

6.3.2 Input Adaption for Homogeneous Signals

If the scene doesn't switch to another scene, i.e., the shifted object signals are homogeneous, the signals will have same amplitudes but differ in the X dimensional displacement. As a result, in the spectral domain, the neighboring frames have identical Fourier locations but differ in the coefficients.

In the beginning, the input signal $x_{i,T0}$ is captured in a video frame at the initial time slot T_0. We generate the Fourier template Tmp once by calculating $x_{i,T0}$'s Fourier transform $\hat{x}_{j,T0}$ using a dense FFT if $\hat{x}$ is not sparse or using a sparse FFT if $\hat{x}$ is

sparse. The Fourier template Tmp containing all the locations of non-zero Fourier coefficients and their order for $\hat{x}_i$ at T_0. The cost includes runtime of a full FFT, i.e. $O(FFT)$, plus the time to identify sparse Fourier locations, i.e. $O(N)$. Next, we need to compute Fourier transform for x_{i-m_1} at time T_1. Since the time-shifted x_{i-m_1} corresponds to $\hat{x}_j e^{-b2\pi m_1 j/N}$ in spectral domain, where $b = \sqrt{-1}$, hence the locations of non-zero Fourier coefficients in $\hat{x}_{j,T1}$ is same as those in $\hat{x}_{j,T0}$, but only the coefficients differ. As a consequence, the Fourier template Tmp is used to compute sparse FFT for $x_{i-m_1,T1}$ at T_1 and for $x_{i-m_t,Tt}$ in the following time slots T_t. Therefore, we only compute dense FFT once on each segment of homogeneous inputs. Except for the first input in the segment, our adaptive sparse FFT is used according to input characteristics we obtained previously.

Moreover, if time shifting factor m_t is known, we can further directly multiply $\hat{x}_{j,T0}$ at initial time T_0 by $e^{-b2\pi m_t j/N}$ to efficiently attain Fourier transform $\hat{x}_{j,Tt}$ at remaining time T_t without a FFT. However, if shifting factor m_t is unknown, we cannot do this to get spectrums for $x_{i-m_t,Tt}$. This situation is feasible in real application. Suppose we use a video recorder to capture several video frames, but sometimes we don't know the time-shifted distance m_t of the two frames. Hence, we have to know m at first. The worst case is to match $x_{i,T0}$ with $x_{i-m_t,Tt}$ and determine m_t in runtime of $O(N^2)$. Nonetheless, such a process can be efficiently executed in $O(N)$ time when applying KMP algorithm [30]. Therefore, if m_t is unknown, we spend time of $O(N)$ on finding m_t and $O(k)$ on multiplying $e^{-b2\pi m_t j/N}$. Total runtime is $O(N+k)$. The evaluation shows that our sparse FFT outperforms the performance of above two situations including known m_t and unknown m_t.

We show an performance evaluation for better illustrating the real-world application of video recording. The test described in firgure 5.6 shows that our input-adaptive FFT for homogeneous inputs outperforms the performance of all other compared situations. It demonstrates that we can spend time to compute a dense FFT once to preprocess the Fourier location template that we need for the FFTs in remaining time.

Then we can save much time by using our hybrid sparse FFT for all the subsequent input signals. Furthermore, if the number of frames is large, our sparse FFT outperforms sFFT as well as AAFFT on average in the real application. The detailed evaluation is described in the evaluation section.

6.3.3 Input Adaption for Discontinuous Signals

When the homogeneity is broken, the input-adaptation restarts by re-calculating the spectral template. There are two types of discontinuity: *Case 1*, the signal size is invariable, but its amplitudes vary; *Case 2*, both signal size and amplitudes vary.

We develop an effective heuristic to detect such discontinuity in frame spectrums. Conceptually, if the standard FFT is computed for each input and the output is compared with the output of our sparse FFT, the deviations between the two will be small in the case of homogeneity, but will become large at the discontinuity point. However, we cannot run a $O(NlogN)$-time standard FFT to detect the discontinuity, which will void all performance advantage of the sparse FFT. Instead, we use sampling. A partial size FFT is calculated as the standard, and its runtime is limited to $klogN$ so that the complexity of our library plus partial standard FFT is still kept to be strictly sublinear to N and to be smaller than the runtime of other sparse FFTs as well. We tried two sampling methods: *First-Partial Method*, simply chooses the first $klogN$ portion from the output; and *Partial Sampling Method*, samples the output by a rate of $\frac{N}{klogN}$.

Subsequently, we need to quantitively define discontinuity. We use the first-level deviation dev_i by comparing the outputs of our sparse FFT with that of sampled FFT. We further conduct a second-level deviation metric
$dev_2nd(dev_i, dev_{i-1})$ to determine the relative degree of difference between dev_i for current signal x_i and dev_{i-1} for signal x_{i-1} in the prior frame $i-1$. From our test, if discontinuity occurs at frame i, $dev_2nd(dev_i, dev_{i-1})$ at the discontinuous point will be much larger than $dev_2nd(dev_{i-m}, dev_{i-m-1})$, $1 \leq m < \#frames$, of the previously homogeneous cases, and can be accurately separated. Since we only need compute the

Table 6.1: Sub-steps Parameters of Input Adaption.

Parameters	**Functionality**
#frames	Total # of video frames per segment.
#segment	Total # of stream segments.
Full_FFT	Time of the full dense or sparse FFT to generate Fourier location template.
Partial_FFT	Time of the partial-size FFT to detect discontinuity.
T_loc	Time to find sparse Fourier locations.
IA_sFFT	Time of our input-adaptive sparse FFT.

second-level metric once, the cost for the entire detection is $O(\text{our_sparse_fft}) + klogN + O(1)$ and is asymptotic to only $O(\text{our_sparse_fft})$. Finally, after the discontinuity has been detected, our algorithm re-calculates the template and resumes the input-adaptation.

The overall performance of our input-adaptive sparse FFT algorithm can be decomposed into the time components listed in table 6.1. Suppose there are $\#segment$ segments of inputs in a stream. In each segment, the first $\#frames - 1$ frames are homogeneous and discontinuity occurs at the last frame. The execution time of our algorithm over the whole stream can be summarized as $Time = Full_FFT + T_loc + (IA_sFFT + Partial_FFT) \times (\#frames - 1) \times \#segments + (Full_FFT + T_loc + IA_sFFT) \times \#segments$.

6.4 Performance Evaluation

In this section we evaluate our input-adaptive sparse FFT implementation and its performance in a real-world-like application. All inputs are double-precision. The evaluation is conducted on three heterogeneous computer configurations. The sequential version is implemented on the Intel i7 920 CPU and the parallel implementation is tuned for three different NVIDIA GPUs, i.e. GeForce GTX480, Tesla C2070 and Tesla C2075. For both sequential and parallel versions, we evaluate our general and optimized sparse FFT approaches, and compare them against four highly-influential

Table 6.2: Configurations of GPUs and CPU.

GPU	Global Memory	NVCC	PCI
GeForce GTX480	1.5GB	3.2	PCIe2.0 x16
Tesla C2070	6GB	3.2	PCIe2.0 x16
Tesla C2075	6GB	3.2	PCIe2.0 x16
CPU	**Frequency, # of Cores**	**System Memory**	**Cache**
Intel i7 920	2.66GHz, 4 cores	24GB	8192KB

FFT libraries: 1) FFTW 3.3.3 [15], the latest FFTW which is one of the most efficient implementations of dense FFT. In FFTW, Streaming Single Instruction Multiple Data Extensions (SSE) on Intel CPU is enabled for better performance. Furthermore, we use two levels of optimizations in FFTW, i.e. ESTIMATE (a basic optimization level marked as 'FFTW' in the plots) and MEASURE (a more aggressively optimized version marked as 'FFTW OPT'). The 4-thread enabled FFTW is used in evaluation of the parallel version. 2) sFFT 1.0 and 2.0 [24], which is one of the fastest sublinear algorithms of sparse FFT. 3) AAFFT 0.9 [26], which is another recent sublinear algorithm with fast empirical runtime. 4) CUFFT 3.2, the NVIDIA CUDA FFT library for GPU-based dense FFT implementation. The reported GPU performance includes the time for both computation and data transferring between host and device. The configurations of the FFT libraries, the GPUs and the CPU have been summarized in table 6.2.

6.4.1 Input-Adaptive Sparse FFT Algorithm

We evaluate both the sequential and the parallel versions of our general sparse FFT in two cases: First, we fix the sparsity parameter $k = 64$ and plot the execution time of our library and the other libraries for 18 different signal sizes from $N = 2^{10}$ to 2^{27}. Second, we fix the signal size to $N = 2^{24}$ and evaluate the running time under different numbers of non-zero frequencies, i.e. k.

6.4.1.1 Sequential Input-Adaptive Sparse FFT

To better illustrate the performance improvement of our parallel input adaptive sparse FFT, we first reiterate the unoptimized implementation performance of our sequential input adaptive sparse FFT.

Figure 6.4 and figure 6.5 show our sequential sparse FFT on an Intel i7 CPU. The basic version of our library is labeled as flag 'General', and the average and the best case of our optimized version is marked as 'OPT-AVG' and 'OPT-BEST', respectively. Specifically, our optimized version performs best when all large coefficients reside in only one truncated region of length $O(klogN)$ so that no shifting is needed. The 'OPT-AVG" case instead runs on a random input for 10 times and then takes an average.

In figure 6.4, we fix $k = 64$ but vary N. The running time of FFTW is linear in the signal size N and sFFT 1.0/2.0 shows approximately linear in N when $N > 2^{20}$. However, our sparse FFT's performance appears almost constant as the signal size increases, which reflects the sub-linear complexity of our algorithm. In addition, AAFFT 0.9 is stable over different N but its performance is lower than ours and sFFT. Overall, our approach outperforms over sFFT, FFTW and AAFFT. Our general version, the average case and the optimal case of our optimized library become faster than FFTW with $N \geq 2^{18}$, $N \geq 2^{17}$, and $N \geq 2^{14}$, respectively, while sFFT and AAFFT achieve this goal with much larger input sizes, i.e., $N \geq 2^{19}$ and $N \geq 2^{24}$, respectively.

In figure 6.5, we fix $N = 2^{24}$ but change k. FFTW shows invariance in performance since its complexity is $O(NlogN)$ which is independent to k. Our general sparse FFT maintains its performance superiority over FFTW for k up to 3000 and 2000, respectively. Our optimal version shows a faster performance than FFTW before k reaches 100000. However, sFFT 1.0, sFFT 2.0 and AAFFT 0.9 are faster than basic FFTW only when k is less than 900, 1000 and 100. Therefore, our approach extends the range of input sparsity parameter k in which a sparse FFT outperforms a dense FFT, the range being an indicative and widely used efficiency metric when evaluating a sparse FFT algorithm. Furthermore, our library performs better than all other

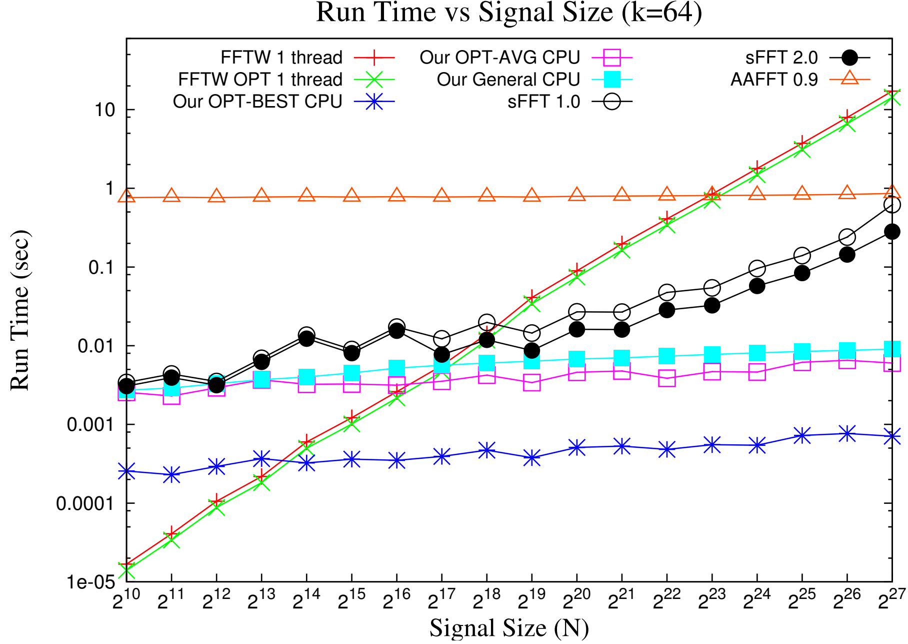


Figure 6.4: Sequential Performance vs. Signal Size.

compared FFT libraries on average.

6.4.1.2 Parallel Input-Adaptive Sparse FFT

In this section, we parallelize the unoptimized implementation of our sequential input adaptive sparse FFT, and we evaluate performance of the parallel version.

Figure 6.6 shows the parallel versions of our sparse FFT on three GPUs. Since there is no parallel version of either sFFT or AAFFT, we only compare to the 4-thread FFTW and CUFFT. In figure 6.6, we fix $k = 64$ and vary N. Both 4-thread FFTW and CUFFT are linear in the signal size N, however, our parallel performance appears constant as N increases. Our general version implementations on the three GPUs are faster than the 1-thread FFTW, the 4-thread FFTW and CUFFT when $N \geq 2^{14}$, $N \geq 2^{16}$ and $N \geq 2^{17}$. Furthermore, the optimal performance of our parallel case is faster than 1-thread FFTW, 4-thread FFTW and CUFFT when $N \geq 2^{12}$, $N \geq 2^{14}$ and $N \geq 2^{14}$.

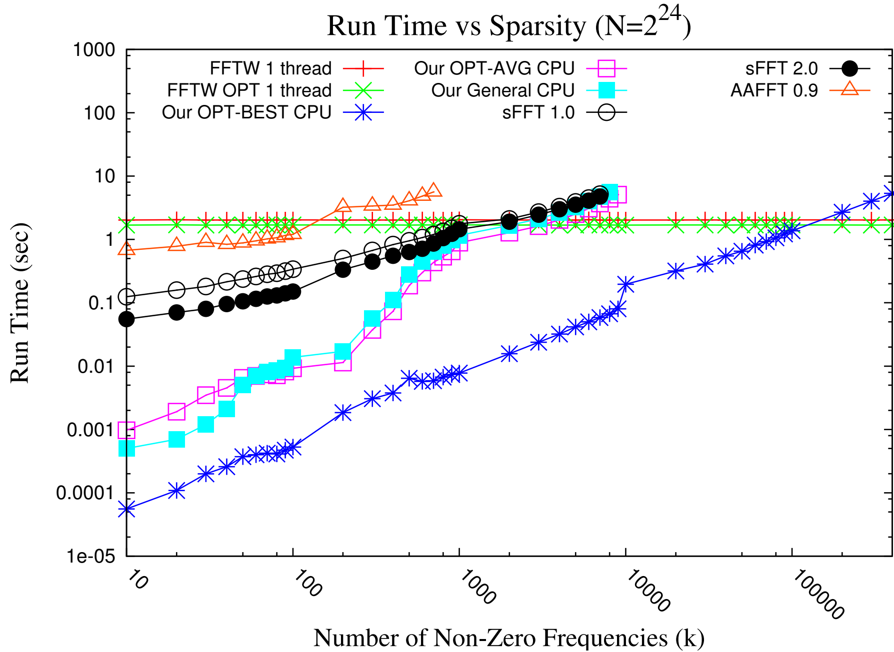


Figure 6.5: Sequential Performance vs. Sparsity Parameter.

In figure 6.7, we fix $N = 2^{24}$ and change k. Specifically, our parallel performance of basic version on GTX480, Tesla C2070 and C2075 has a runtime faster than 1-thread FFTW for k up to 30000, 40000, 50000, faster than 4-thread FFTW before k reaches to 20000, 30000, 30000, and faster than CUFFT for k less than 6000, 8000, 9000, respectively. Additionally, the optimal performance of our parallel case is better than 1-thread FFTW, 4-thread FFTW, CUFFT for k up to 500000, 100000, 40000, respectively.

6.4.2 Detection for Signal Discontinuity

In section 6.4.1, the performance of our pure sparse-FFT library is evaluated based on homogeneous input signals. When the homogeneity is broken, the heuristic introduced in the section 6.3 is used to detect when the discontinuity happens. Next, we evaluate how well our heuristic works and how much overhead it incurs.

We use test cases similar to the image streaming processing scenario described

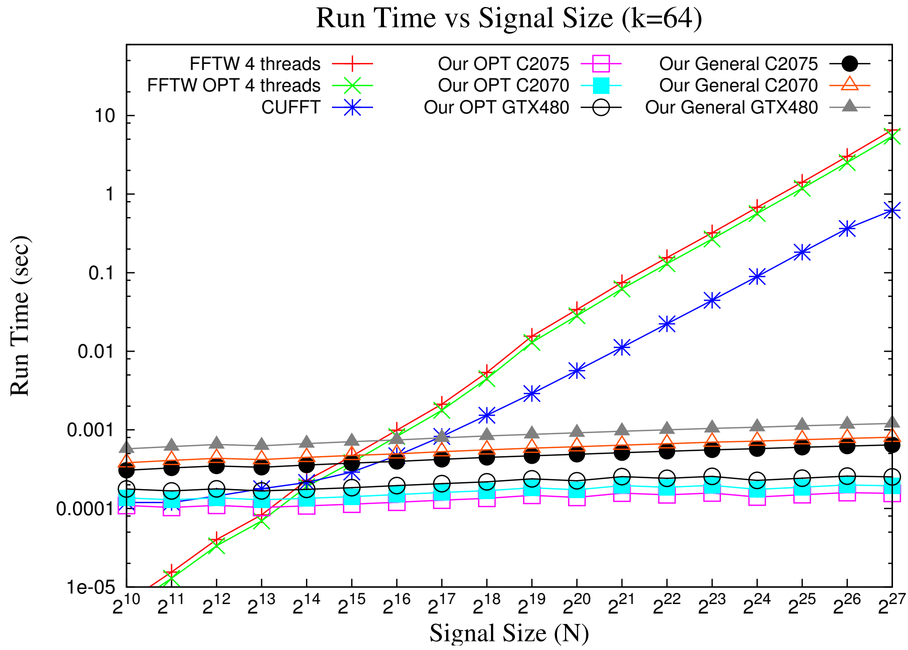


Figure 6.6: Parallel Performance vs. Signal Size on Three GPUs.

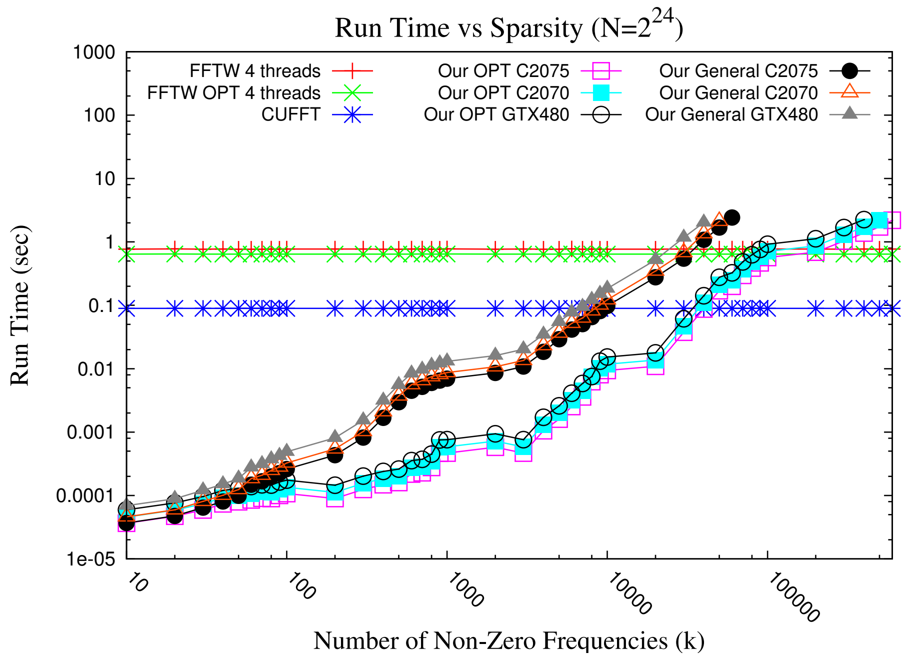


Figure 6.7: Parallel Performance vs. Sparsity Parameter on Three GPUs.

in section 6.3.3. In total 3 segments of image frames are used. In each segment, the frames are homogeneous except for the last frame at which discontinuity occurs. For the case-1 discontinuity, we keep to use the same signal size N, but re-generate signal with randomly picked amplitudes differing from the homogeneous signals. For the case-2 discontinuity, the signal size is cut down to $N/2$ and its amplitudes are randomly re-generated. The size of each image signal is $N = 2^{22}$ and the sparsity parameter $k = 64$.

Figure 6.8 and figure 6.9 show the first-partial method and the partial sampling detection method for the case-1 discontinuity. There are 3 segments and 32 frames per segment. For each homogeneous frame, it shifts by a displacement of 2^{17} points. We use the Root-Mean-Square-Error (RMSE) between the sampled FFT and our sparse FFT as the deviation metric. The RMSE is defined as $\sqrt{\frac{\sum_{i=0}^{N-1}[(F_x-f_x)^2+(F_y-f_y)^2]}{2N}}$.

As shown in figure 6.9, our library and the sampled FFT produce almost the same results for homogeneous frames with the 1st-level RMSEs in the small range of $(4.43 \times 10^1, 4.52 \times 10^1)$. However, the outputs are significantly different at the three discontinuity points with the 1st-level RMSEs being 5.35×10^1, 5.31×10^1 and 5.41×10^1, respectively. The spectrally similar signals will produce much closer RMSEs than the discontinuous cases. We can further calculate the 2nd-level RMSE as the RMSE of the 1st-level RMSEs of adjacent frames. The line in the figure represents the 2nd-level RMSE clearly shows values smaller than 2.7 for the homogeneous cases and larger than 8.5 for the discontinuous points. The two boundaries are separated by a large margin. Therefore, the discontinuity is accurately detected at frame 32, 64 and 96 by both sampling methods. Figure 6.10 shows the partial sampling detection method for case-2 discontinuity with 3 segments and 32 frames per segment. Each homogeneous frame shifts by 2^{17} points. Similarly, discontinuities are detected at frame 32, 64 and 96, and the RMSE of case-2 discontinuity is larger that of the case-1. Figure 6.11 shows the partial sampling detection for case-2 discontinuity with 3 segments and 128 frames per segment. For each homogeneous frame, it shifts by a factor of 2^{15}. The discontinuities at the frames 128, 256 and 384 are also successfully detected.

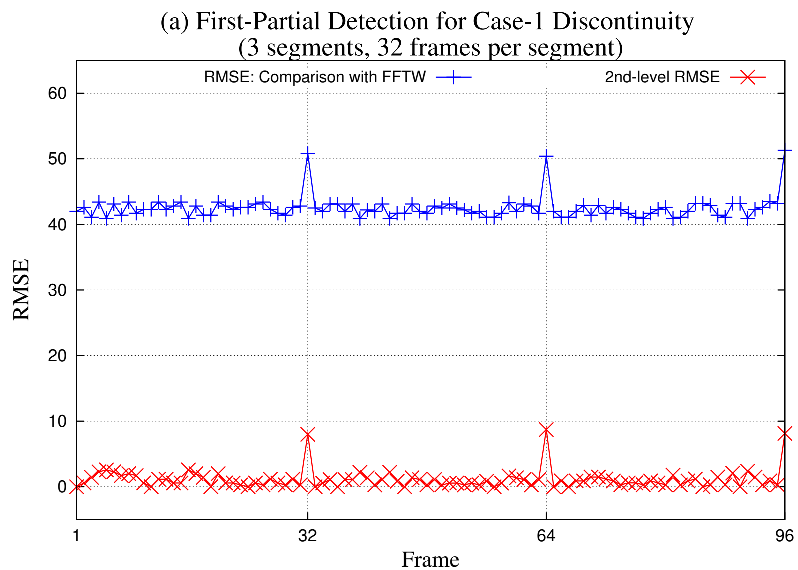


Figure 6.8: First Partial Detection for Case-1 Discontinuity with 3 Segments and 32 Frames per Segment.

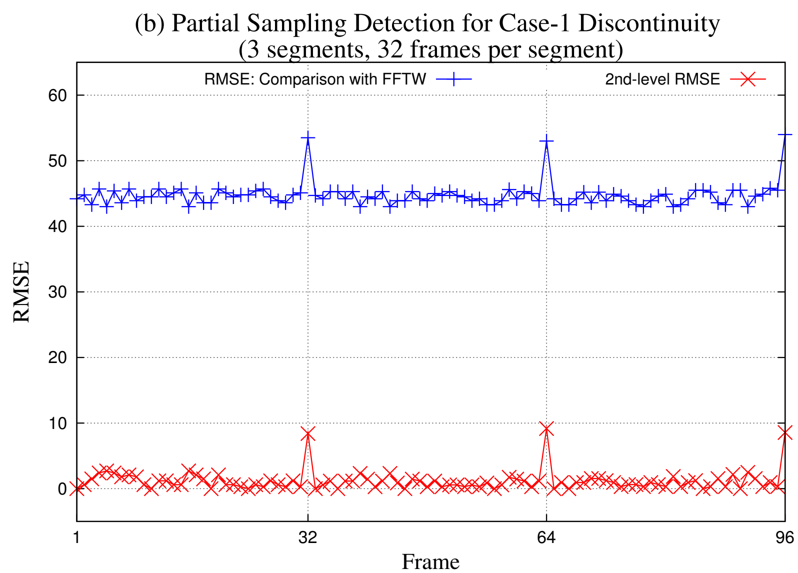


Figure 6.9: Partial Sampling Detection for Case-1 Discontinuity with 3 Segments and 32 Frames per Segment.

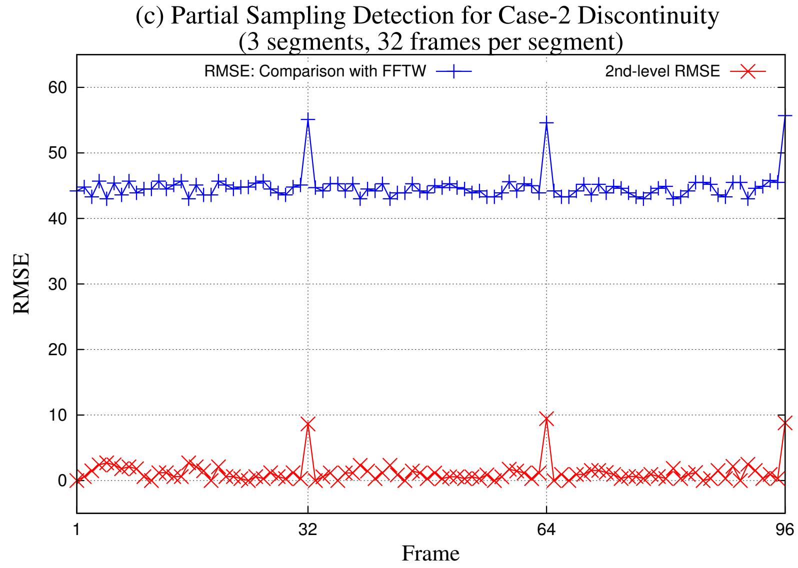


Figure 6.10: Partial Sampling Detection for Case-2 Discontinuity with 3 Segments and 32 Frames per Segment.

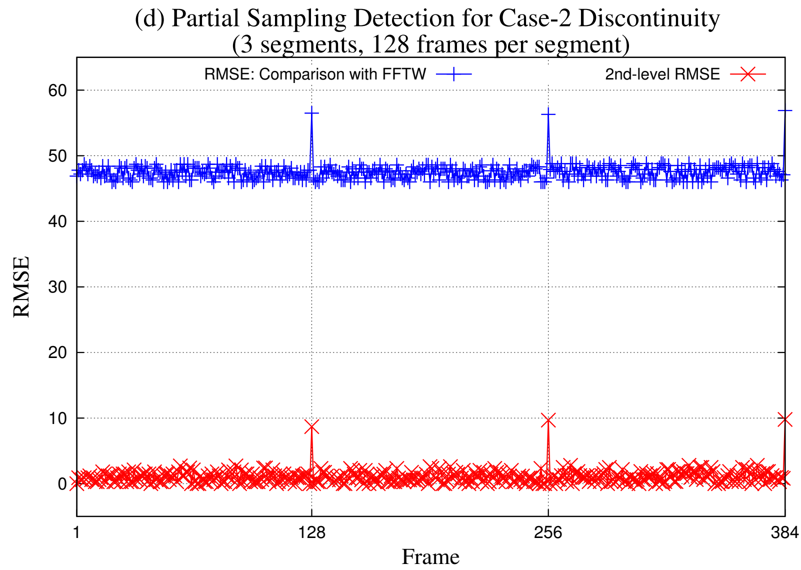


Figure 6.11: Partial Sampling Detection for Case-2 Discontinuity with 3 Segments and 128 Frames per Segment.

6.4.3 Performance of Input Adaption Process

We want to see how our algorithm works in a real-world application of the stream processing scenario established in section 6.3. Actually, our approach is to achieve efficiency in such a process: we can spend time to compute a dense FFT once to preprocess the Fourier location template that we need for the FFTs in remaining time. Then we are able to save much computational time by using our hybrid sparse FFT for all the subsequent input samples. Furthermore, if the number of frames is large, our sparse FFT outperforms other sparse FFT implementations on average in the real application.

We use the established scenario in section 6.3 to evaluate the performance of input adaption process. There are several stream segments of image frames. In each segment, all frames are homogeneous except for the last frame where discontinuity occurs. When discontinuity has been detected, we are able to regenerate Fourier template and recalculate our sparse FFT.

This section evaluates the impact of this input adaption process to the overall performance. When discontinuity is detected by the heuristic, the Fourier templates will be re-generated. The input size is $N = 2^{22}$ and the sparsity parameter $k = 64$. Table 6.3 shows the time of each sub-step in table 6.1 in the entire input adaption process of our sparse FFT. All our approaches use the hybrid subroutine. The optimal adaption, labeled as 'OPT', is to compute sFFT to generate template if input spectrum is sparse. The overall performance is measured including the overhead of the detection heuristic, the recalculation of spectrum templates when discontinuity is found, and the adaptive sparse FFT. Clearly, the more frequently the spectrum templates are calculated, the higher the overhead is, and the lower the performance advantage of our adaptive sparse FFT over the existing input oblivious algorithms. Therefore, we try to determine the break-even point by varying the number of homogeneous frames in a video segment, i.e., in each segment all frames are homogeneous except for the last frame where discontinuity occurs. According to section 6.3.3, the overall performance including our sparse FFT calculation, detection and recalculation process is presented.

Table 6.3: Time of Input Adaption Substeps

Parameter	Seconds	Parameter	Seconds
Full_FFT(FFTW_1thread)	0.347	Full_FFT(FFTW_4threads)	0.107
Partial_FFT(FFTW)	0.00002	T_loc	0.03
Full_FFT(sFFT1.0)	0.047	Full_FFT(sFFT2.0)	0.028
Full_FFT(CUFFT)	0.022	IA_sFFT(Our_CPU)	0.007
IA_sFFT(Our_GTX480)	0.001	IA_sFFT(Our_C2070)	0.0007
IA_sFFT(Our_C2075)	0.0005		

Figure 6.12 and figure 6.13 illustrates the performance of our input adaption process with one segment of frames on CPU and three GPUs, respectively. The y-axis represents the running time in seconds and x-axis denotes different number of frames per segment. In figure 6.12, when the $\#frames \geq 8$, our sequential library on Intel i7 920 CPU is faster than both 1-thread and 4-thread FFTW, while when $\#frames > 32$, our library is faster than sFFT 1.0 and 2.0. Moreover, when $\#frames = 128$, our average performance gains 26.2×, 10.1× speedup over 1-thread and 4-thread FFTW, and 3.6×, 2.3× speedup over sFFT 1.0 and 2.0, respectively. In figure 6.13, when the $\#frames \geq 8$, our parallel library on the three GPUs is faster than both 1-thread and 4-thread FFTW, while when $\#frames > 32$, our library is faster than sFFT 1.0, 2.0 and CUFFT. Furthermore, when $\#frames = 128$, our average performance on Tesla C2075 GPU achieves 55.1×, 21.1× speedup over 1-thread and 4-thread FFTW, and 7.7×, 4.6×, 3.7× speedup over sFFT 1.0, sFFT 2.0 and CUFFT, respectively. Additionally, when $\#frames = 128$, our optimal performance on Tesla C2075 obtains 6.2× speedup against that of CPU version.

Figure 6.14 and figure 6.15 illustrates the performance of our input adaption process with 3 segments of frames on CPU and GPUs, respectively. In figure 6.14, when the $\#frames \geq 8$, our sequential library on Intel i7 920 CPU is faster than both 1-thread and 4-thread FFTW, while when $\#frames \geq 32$, our library is faster than sFFT 1.0 and 2.0. Moreover, when $\#frames = 128$, our average performance gains 30.2×, 11.5× speedup over 1-thread and 4-thread FFTW, and 4.2×, 2.6× speedup over

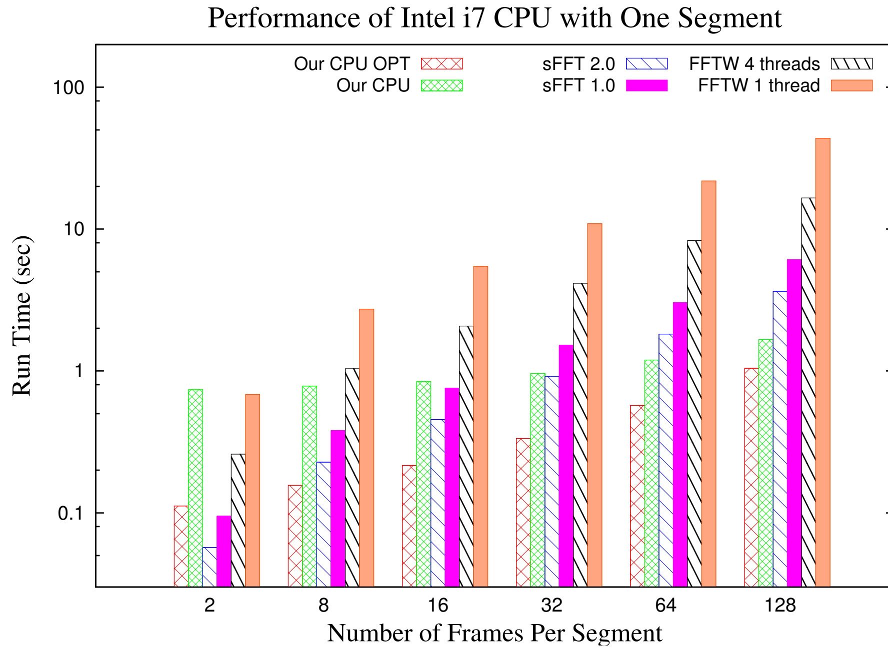


Figure 6.12: Performance of Input Adaption with One Segment on CPU.

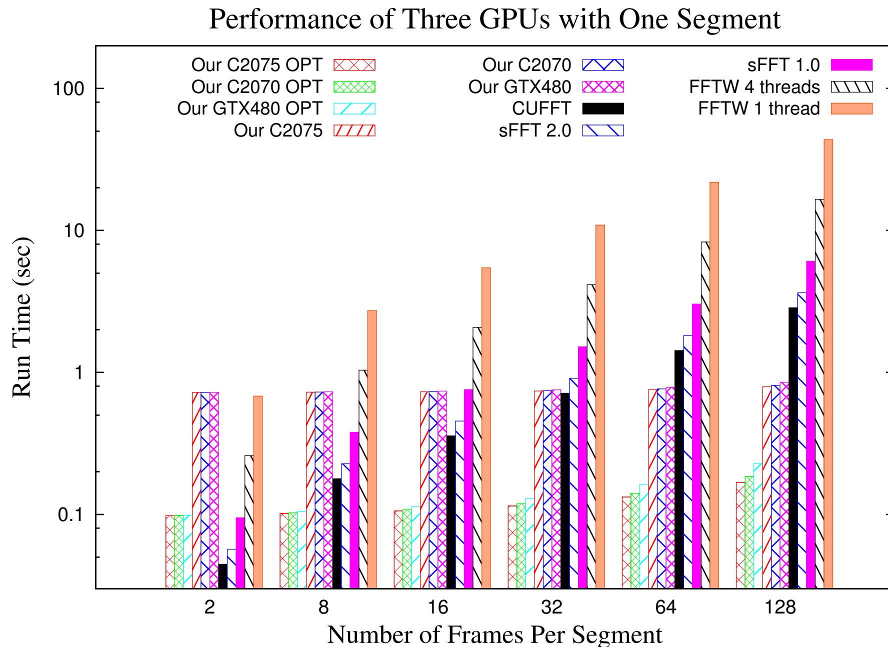


Figure 6.13: Performance of Input Adaption with One Segment on GPUs.

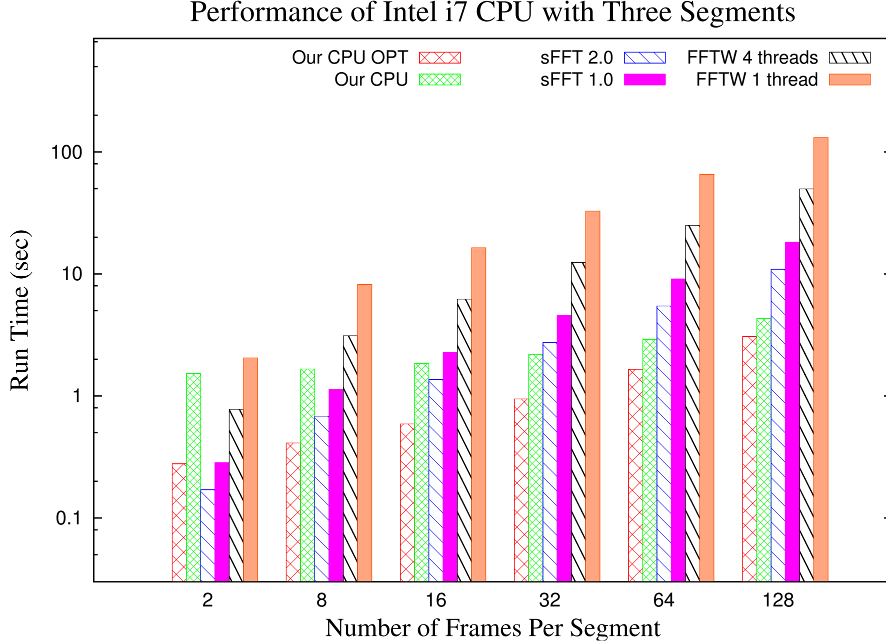


Figure 6.14: Performance of Input Adaption with Three Segments on CPU.

sFFT 1.0 and 2.0, respectively. In figure 6.15, when the $\#frames \geq 8$, our parallel library on the three GPUs is faster than both 1-thread and 4-thread FFTW, while when $\#frames > 16$, our library is faster than sFFT 1.0, 2.0 and CUFFT. Furthermore, when $\#frames = 128$, our average performance on Tesla C2075 GPU achieves 77.2×, 29.3× speedup over 1-thread and 4-thread FFTW, and 10.7×, 6.4×, 5.2× speedup over sFFT 1.0, sFFT 2.0 and CUFFT, respectively. Meanwhile, when $\#frames = 128$, our optimal performance on Tesla C2075 obtains 6.9× speedup against that of CPU version.

6.4.4 Precision of Our Sparse FFT

Note that we do not permute input in time domain to approximate the equal distanced permutation with a certain probability bound, but rather directly permute in spectral domain. In addition, each bucket certainly bins only one large coefficient. Therefore our sparse FFT algorithm is always capable of producing a determinative output.

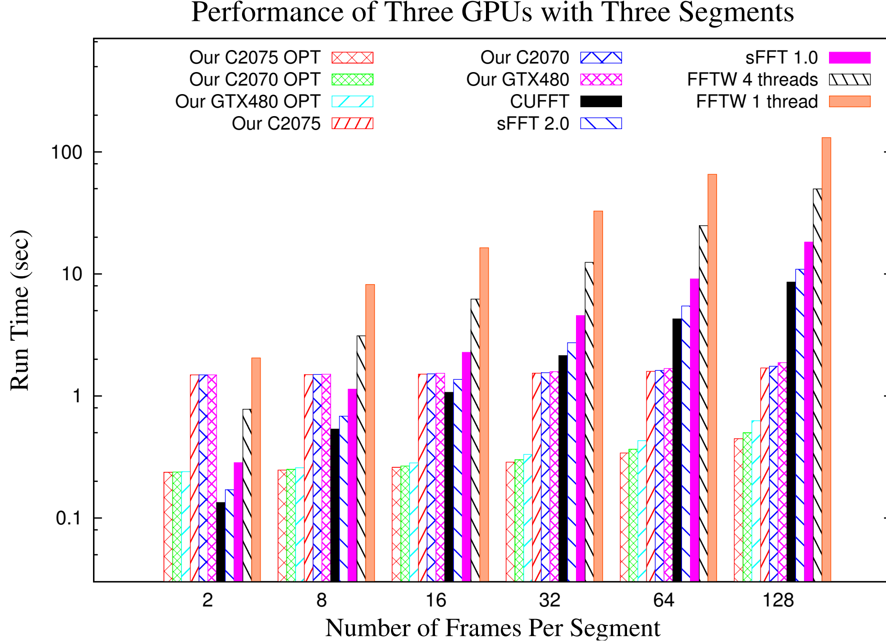


Figure 6.15: Performance of Input Adaption with Three Segments on GPUs.

The accuracy of our sparse FFT implementation is verified by comparing its complex Fourier transform (F_x, F_y) with the output (f_x, f_y) of full FFTW, which is a widely used standard FFT library, for the same double-precision input. All the sparse Fourier coefficients are assumed to be integers in the range of [-1, 1]. Again, the difference in output is quantized as RMSE which has been defined in section 6.4.2. Lower RMSE value means the two computation routines produce more similar result.

The RMSEs of different signal sizes N and sparsity parameters k are tested. With the decrease of k for each N, the RMSE decreases. Meanwhile, under the same k, when N increases, the RMSE has a relative decrease. Overall, the RMSE is small on average, and our sparse FFT can produce nearly same results as FFTW.

6.5 Chapter Summary

In this chapter, we propose a parallel input-adaptive sparse fast fourier transform for stream processing. Our approach breaks down the dependency in the recursive

coefficient packaging and estimation of traditional sequential sparse FFTs. Therefore, substantial data parallelism can be exploited for multi-core CPU and GPU to compute and to accelerate performance. In addition, our parallel input adaptive sparse FFT is well applicable in the stream processing applications. Our input adaptive algorithm can automatically detect and utilize the similarity between continuous inputs to accelerate computation. When a discontinuity occurs, our discontinuity detection method can automatically detect the discontinuities inside the streams and resumes the continuous input adaptation very efficiently. Finally, we evaluate our parallel input-adaptive sparse FFT implementation on Intel Core i7 CPUs and three NVIDIA GPUs, i.e., NVIDIA GeForce GTX480, Tesla C2070 and Tesla C2075. Our parallel sparse FFT is much faster than previous FFTs both in theory and implementation. For instance, our parallel input adaptive sparse FFT on Tesla C2075 GPU achieves up to 77.2$\times$ and 29.3$\times$ speedups over 1-thread and 4-thread FFTW, 10.7$\times$, 6.4$\times$, 5.2$\times$ speedups against sFFT 1.0, sFFT 2.0, CUFFT, and 6.9$\times$ speedup over our sequential CPU performance, respectively.

Chapter 7

CONCLUSION AND FUTURE WORK

First of all, in this dissertation, we proposed a hybrid FFT library that concurrently uses both CPU and GPU to compute large FFT problems. The library has four 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; an optimizer that exploits substantial parallelism for both GPU and CPUs; a load balancer that assigns workloads to both GPU and CPU, and determines the optimal load balancing by effective performance modeling; and a heuristic that empirically tunes the library to best tradeoff among communication, computation and their overlapping. Overall, our hybrid library outperforms several latest and widely used large-scale FFT implementations.

Secondly, to improve the existing sparse FFT algorithms, we proposed a new input-adaptive approach for the exploitation of the similarity between sparse input samples in stream processing to improve the efficiency of sparse FFT. Specifically, our work develops an effective heuristic to detect input similarity, and dynamically customizes the algorithm design to achieve better performance. In particular, we integrate adaptive filters to package non-zero Fourier coefficients into sparse bins which can be estimated accurately. Moreover, our algorithm is non-iterative with high computation intensity such that parallelism can be exploited for multi-CPUs and GPU to improve performance. Overall, our algorithm is much faster than other dense and sparse FFTs both in theory and implementation.

In the future work, the FFT algorithm parallelization and optimization for high performance computing is expected to be further implemented into some other kinds of parallel computer systems, such as CPU based cluster or GPU based cluster, in

the view of current trends of high performance computing and anticipated petascale architecture and supercomputers. Moreover, it can be further applied into some other applications or fields, such as physics simulation, multiscale modeling of turbulent clouds, climate prediction, etc. Finally, future research aims to optimize multiscale simulations and related algorithms on hybrid HPC system.

BIBLIOGRAPHY

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

[2] Akira Nukada, Yasuhiko Ogata, Toshio Endo, and Satoshi 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] Akira Nukada, and Satoshi 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] Naga K. Govindaraju, Brandon Lloyd, Yuri Dotsenko, Burton Smith, and John Manferdelli. "High performance discrete Fourier transforms on graphics processors." In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–12, 2008.

[5] Liang Gu, Xiaoming Li, and Jakob 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] Liang Gu, Xiaoming Li, and Jakob Siegel. "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] Yifeng Chen, Xiang Cui, and Hong Mei. "Large-scale FFT on GPU clusters." In *Proceedings of the 24th ACM International Conference on Supercomputing*, pages 315–324. ACM, 2010.

[8] Yasuhito Ogata, Toshio Endo, Naoya Maruyama, and Satoshi 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] Pierre 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] James W. Cooley, and John W. Tukey. "An algorithm for the machine computation of complex Fourier series." *Mathematics of Computation*, 19(90):297–301, 1965.

[11] Alan V. Oppenheim, and Ronald W. Schafer. *Discrete-Time Signal Processing*. 1999.

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

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

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

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

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

[17] Matteo Frigo, and Steven G. Johnson. "The Fastest Fourier Transform in the West." In *Proceeding of the IEEE*, 93(2):216–231, 1997.

[18] (2012) Intel Math Kernel Library. [Online]. Available: http://software.intel.com/en-us/articles/intel-mkl-103-release-notes.

[19] Adi Akavia. "Deterministic sparse fourier approximation via fooling arithmetic progressions." In *The 23rd Conference on Learning Theory*, pages 381–393, 2010.

[20] Adi Akavia, Shafi Goldwasser, and Samuel Safra. "Proving hard-core predicates using list decoding." In *The 44th Symposium on Foundations of Computer Science*, pages 146–157. IEEE, 2003.

[21] A. Gilbert, S. Guha, P. Indyk, M. Muthukrishnan, and M. Strauss. "Near-optimal sparse fourier representations via sampling." In *Proceedings on 34th Annual ACM Symposium on Theory of Computing*, pages 152–161. ACM, 2002.

[22] A. Gilbert, M. Muthukrishnan, and M. Strauss. "Improved time bounds for near-optimal space fourier representations." In *Proceedings of SPIE Wavelets XI*, 2005.

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

[24] H. Hassanieh, P. Indyk, D. Katabi, and E. Price "Simple and practical algorithm for sparse fourier transform." In *Proceedings of the 23th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1183–1194. ACM, 2012.

[25] H. Hassanieh, P. Indyk, D. Katabi, and P. E. "Nearly optimal sparse fourier transform." In *Proceedings of the 44th symposium on Theory of Computing*, pages 563–578. ACM, 2012.

[26] M. Iwen. "AAFFT (Ann Arbor Fast Fourier Transform)." http://sourceforge.net/projects/aafftannarborfa/, 2008.

[27] M. Iwen. "Combinatorial sublinear-time fourier algorithms." *Foundations of Computational Mathematics*, 10(3):303–338, 2010.

[28] Y. Mansour. "Randomized interpolation and approximation of sparse polynomials." In *The 19th International Colloquium on Automata, Languages and Programming*, pages 261–272. Springer, 1992.

[29] E. Kushilevitz, and Y. Mansour "Learning decision trees using the fourier spectrum." In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 455–464. ACM, 1991.

[30] D. Knuth, J. Morris, and V. Pratt. "Fast pattern matching in strings." *SIAM Journal on Computing*, 6(2):323–350, 1977.

[31] Jose M. F. Moura, Jeremy Johnson, Robert W. Johnson, David Padua, Viktor K. Prasanna, Markus Pschel, and Manuela Veloso. "SPIRAL: Automatic Implementation of Signal Processing Algorithms." In *Proc. High Performance Embedded Computing (HPEC)*, 2000.

[32] Jose M. F. Moura, Jeremy Johnson, Robert W. Johnson, David Padua, Viktor K. Prasanna, Markus Puschel, Bryan Singer, Manuela Veloso, and Jianxin Xiong. "Generating Platform-Adapted DSP Libraries using SPIRAL." In *Proc. High Performance Embedded Computing (HPEC)*, 2001.

[33] Markus Puschel, and Jose M. F. Moura. "SPIRAL: An Overview." In *Proc. Workshop on Optimizations for DSP and Embedded Systems (ODES)*, 2003.

[34] Markus Puschel, Bryan Singer, Jianxin Xiong, Jose M. F. Moura, Jeremy Johnson, David Padua, Manuela Veloso, and Robert W. Johnson. "SPIRAL: A Generator for Platform-Adapted Libraries of Signal Processing Algorithms." *Journal of High Performance Computing and Applications, special issue on "Automatic Performance Tuning"*, Vol. 18, No. 1, pp. 21-45, 2004.

[35] Markus Puschel, Jose M. F. Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nicholas Rizzolo. "SPIRAL: Code

Generation for DSP Transforms." In *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"'*, Vol. 93, No. 2, pp. 232-275, 2005.

[36] Thom Popovici, F. Russell, K. Wilkinson, C-K. Skylaris, P. H. J. Kelly, and Franz Franchetti. "Generating Optimized Fourier Interpolation Routines for Density Functional Theory Using SPIRAL." In *Proc. International Parallel and Distributed Processing Symposium (IPDPS)*, 2015.

[37] Franz Franchetti, Yevgen Voronenko, and Markus Puschel. "Spiral: Generating Signal Processing Kernels for New Commodity Architectures." In *Proc. EDGE Workshop*, pp. D49-D50, 2006.

[38] Franz Franchetti, Andreas Bonelli, Ekapol Chuangsuwanich, Yu-Chiang Lee, Juergen Lorenz, Thomas Peter, Hao Shen, Marek Telgarsky, Yevgen Voronenko, Markus Pschel, Jose M. F. Moura, and Christoph W. Ueberhuber. "Parallelism in Spiral." In *Proc. Workshop on Programming Models for Ubiquitous Parallelism (PMUP)*, 2006.

[39] Marek Telgarsky, James C. Hoe, and Jose M. F. Moura. "Spiral: Joint Runtime and Energy Optimization of Linear Transforms." In *Proc. International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, Vol. 3, 2006.

[40] Yevgen Voronenko, Franz Franchetti, Frederic de Mesmay, and Markus Pschel. "System Demonstration of Spiral: Generator for High-Performance Linear Transform Libraries." In *Proc. Algebraic Methodology and Software Technology (AMAST)*, 2008.

[41] Franz Franchetti, Daniel McFarlin, Frederic de Mesmay, Hao Shen, Tomasz Wiktor Wlodarczyk, Srinivas Chellappa, Marek Telgarsky, Peter A. Milder, Yevgen Voronenko, Qian Yu, James C. Hoe, Jose M. F. Moura, and Markus Puschel. "Program Generation with Spiral: Beyond Transforms." In *Proc. High Performance Embedded Computing (HPEC)*, 2008.

[42] Yevgen Voronenko, Franz Franchetti, Frederic de Mesmay, and Markus Puschel. "Generating High-Performance General Size Linear Transform Libraries Using Spiral." In *Proc. High Performance Embedded Computing (HPEC)*, 2008.

[43] Lingchuan Meng, Jeremy Johnson, Franz Franchetti, Yevgen Voronenko, Marc Moreno Maza, and Yuzhen Xie. "Spiral-Generated Modular FFT Algorithms." In *Proc. Parallel Symbolic Computation (PASCO)*, pp. 169-170, 2010.

[44] Georg Ofenbeck, Tiark Rompf, Alen Stojanov, Martin Odersky, and Markus Puschel. "Spiral in Scala: Towards the Systematic Construction of Generators for Performance Libraries." In *Proc. International Conference on Generative Programming: Concepts and Experiences (GPCE)*, pp. 125-134, 2013.

[45] Franz Franchetti, Aliaksei Sandryhaila, and Jeremy Johnson. “High Assurance SPIRAL.” In *Proc. SPIE, Proceedings of SPIE*, 2014.

[46] Steven G. Johnson, and Matteo Frigo. “A modified split-radix FFT with fewer arithmetic operations.” *IEEE Trans. Signal Processing*, 55 (1), 111119, 2007.

[47] M. Frigo, and S. G. Johnson. “FFTW: An Adaptive Software Architecture for the FFT.” In *Proc. International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, 1998.

[48] Steven G. Johnson, and Matteo Frigo. “Implementing FFTs in Practice.” *Fast Fourier Transforms*, September, 2008.

[49] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. “Cache-oblivious algorithms.” In *Proc. 40th Ann. Symp. on Foundations of Comp. Sci. (FOCS)*, 1999.

[50] Intel Corp. “Using Intel MKL Automatic Offload on Intel Xeon Phi Coprocessors.” White Paper, 2011.

[51] Todd Rosenquist, and Shane Story. “Using the Intel Math Kernel Library (Intel MKL) and Intel Compilers to Obtain Run-to-Run Numerical Reproducible Results.” *Intel Software Network*, 2012.

[52] Stephen Lewin-Berlin. “Using the Intel Math Kernel Library (Intel MKL) and Intel Compilers to Obtain Run-to-Run Numerical Reproducible Results.” *Intel Software Network*, 2009.

[53] Intel Corp. “The Flagship High-Performance Computing Math Library for Windows, Linux, and Mac OS X. Intel Math Kernel Library (Intel MKL) 10.3.” *Intel Software Network*, 2010.

[54] Todd Rosenquist. “Getting Reproducible Results with Intel MKL.” *Intel Software Network*, 2010.

[55] Martyn Corden. “Consistency of Floating-Point Results using the Intel Compiler.” *Intel Software Network*, 2010.

[56] I. Burylov, M. Chuvelev, B. Greer, G. Henry, S. Kuznetsov, B. Sabanin. “Intel Performance Libraries: Multicore-ready Software for Numeric Intensive Computation.” *Intel Technical Journal*, Vol. 11 Issue 4, November 2007.

[57] NVIDIA Corp. CUDA CUFFT Library. 2007.

[58] NVIDIA Corp. NVIDIA CUDA Compute Unified Device Architecture. 2007.

[59] Akira Nukada, and Satoshi Matsuoka. "NukadaFFT : An Auto-Tuning FFT Library for CUDA GPUs." In *NVIDIA GPU Technology Conference 2010*, Research Summit Poster, San Jose, September 2010.

[60] Yuri Dotsenko, Sara Baghsorkhi, Brandon Lloyd, and Naga Govindaraju. "Autotuning of Fast Fourier Transform on Graphics Processors." In *Proc. of ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming 2011 (PPoPP 2011)*, 2011.

[61] Brandon Lloyd, Chas Boyd, and Naga Govindaraju. "Fast Computation of General Fourier Transforms on GPUs." In *Proc. of IEEE ICME 2008 (MSR Tech. Report*, April, 2008.

[62] Naga Govindaraju, and Dinesh Manocha. "Cache-Efficient Numerical Algorithms using Graphics Hardware." *Special issue on High Performance Accelerators, Journal of Parallel Computing*, 2007.

[63] Liang Gu, and Xiaoming Li. "Performance modeling for DFT algorithms in FFTW." In *Proceeding of the 23rd International Conference on Supercomputing (ICS)*, poster paper, pp.507-508, 2009.

[64] Liang Gu, and Xiaoming Li. "DFT Performance Prediction in FFTW." In *The 22nd International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, Oct, 2009.

[65] Nvidia Corp. CUDA C Programming Guide. *CUDA Toolkit Documentation*, 2015.

[66] Jayshree Ghorpade, Jitendra Parande, Madhura Kulkarni, and Amit Bawaskar. "GPGPU Processing in CUDA Architecture." Advanced Computing: An International Journal (ACIJ), Vol.3, No.1, January, 2012.

[67] Jason Sanders, and Edward Kandrot. "CUDA by Example: An Introduction to General-Purpose GPU Programming." 2010.

[68] David Patterson. "The Top 10 Innovations in the New NVIDIA Fermi Architecture, and the Top 3 Next Challenges." Parallel Computing Research Laboratory and NVIDIA. Retrieved, October, 2013.

[69] Nvidia Corp. "NVIDIAs Next Generation CUDA Compute Architecture: Fermi." 2009.

[70] P. N. Glaskowsky. "NVIDIAs Fermi: The First Complete GPU Computing Architecture." White Paper, 2009.

[71] N. Brookwood. "NVIDIA Solves the GPU Computing Puzzle." White Paper, 2009.

[72] N. Whitehead, and A. Fit-Florea. "Precision and Performance: Floating Point and IEEE 754 Compliance for NVIDIA GPUs. *CUDA Toolkit Documentation*, 2011.

[73] Nvidia Corp. "A High-performance Area-efficient Multifunction Interpolator." In *Proc. of the 17th IEEE Symposium on Computer Arithmetic*, pp. 272279, July, 2005.

[74] R. Farber. "CUDA Application Design and Development." 2011

[75] Nvidia Corp. "Tuning CUDA applications for Fermi." *NVIDIA Application Note*, 2011.

[76] Sungpack Hong, Sang Kyun Kim, Tayo Oguntebi, and Kunle Olukotun. "Accelerating CUDA Graph Algorithms at Maximum Warp." In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming (PPoPP '11)*, 2011.

[77] Gauss, and Carl Friedrich. "Theoria interpolationis methodo nova tractata." Werke, Band 3, 265327, Knigliche Gesellschaft der Wissenschaften, Gttingen, 1866.

[78] Heideman, M. T., D. H. Johnson, and C. S. Burrus. "Gauss and the history of the fast Fourier transform." *IEEE ASSP Magazine*, 1, (4), 1421, 1984.

[79] James W. Cooley, Peter A. W. Lewis, and Peter D. Welch. "Historical notes on the fast Fourier transform." *IEEE Trans. on Audio and Electroacoustics*, 15 (2): 7679, 1967.

[80] Daniel N. Rockmore. "The FFT an algorithm the whole family can use." In *Special issue on "top ten algorithms of the century ", Comput. Sci. Eng.*, 2 (1), 60, 2000.

[81] James W. Cooley, Peter A. W. Lewis, and Peter W. Welch. "Historical notes on the fast Fourier transform." In *Proc. IEEE*, vol. 55 (no. 10), p. 16751677, 1967.

[82] G. C. Danielson, and C. Lanczos. "Some improvements in practical Fourier analysis and their application to X-ray scattering from liquids." J. Franklin Inst. 233, 365380 and 435452, 1942.

[83] S. G. Johnson, and M. Frigo. "Implementing FFTs in practice." In *Fast Fourier Transforms (C. S. Burrus, ed.)*, ch. 11, Rice University, Houston TX, September 2008.

[84] Richard C. Singleton. "On computing the fast Fourier transform." *Commun. of the ACM*, 10 (10): 647654, 1967

[85] T. Lundy, and J. Van Buskirk. “A new matrix approach to real FFTs and convolutions of length 2k.” *Computing*, 80, 23-45, 2007.

[86] S. G. Johnson, and M. Frigo. “A modified split-radix FFT with fewer arithmetic operations.” *IEEE Trans. Signal Processing*, 55 (1), 111119, 2007.

[87] W. M. Gentleman, and G. Sande. “Fast Fourier transforms – for fun and profit.” In *Proc. AFIPS*, 29, 563578, 1966.

[88] David H. Bailey. “FFTs in external or hierarchical memory.” *J. Supercomputing*, 4 (1), 2335, 1990.

[89] M. Frigo, C.E. Leiserson, H. Prokop, and S. Ramachandran “Cache-oblivious algorithms.” In *In Proceedings of the 40th IEEE Symposium on Foundations of Computer Science (FOCS 99)*, p.285-297. 1999.

[90] J. W. Cooley, P. Lewis, and P. Welch. “The Fast Fourier Transform and its Applications.” *IEEE Trans on Education*, 12, 1, 28-34, 1969.

[91] Alan H. Karp. “Bit reversal on uniprocessors.” *SIAM Review*, 38 (1): 126, 1996.

[92] Larry Carter, and Kang Su Gatlin. “Towards an optimal bit-reversal permutation program.” In *Proc. 39th Ann. Symp. on Found. of Comp. Sci. (FOCS)*, 544553, 1998.

[93] M. Rubio, P. Gmez, and K. Drouiche. “A new superfast bit reversal algorithm.” In *Intl. J. Adaptive Control and Signal Processing*, 16 (10): 703707, 2002.

[94] James W. Cooley, D.L. Favin, H.D. Helms, R.A. Kaenel, W.W. Lang, Jr. G.C. Maling, D.E. Nelson, C.M. Rader, and Peter D.Welch. “What is the fast Fourier transform?.” In *Proc. IEEE*, vol. 55, 16641674, 1967.

[95] P. N. Swarztrauber. “FFT algorithms for vector computers.” *Parallel Computing*, vol. 1, 4563, 1984.

[96] P. N. Swarztrauber. “Vectorizing the FFTs.” *Parallel Computations*, New York, 1982.

[97] M. C. Pease. “An adaptation of the fast Fourier transform for parallel processing.” *J. ACM*, 15 (2): 252264, 1968.

[98] Matteo Frigo, and Steven G. Johnson. “FFTW”. A free (GPL) C library for computing discrete Fourier transforms in one or more dimensions, of arbitrary size, using the Cooley Tukey algorithm, 1998.

[99] H. W. Johnson, and C. S. Burrus. “An in-place in-order radix-2 FFT.” In *Proc. ICASSP*, 28A.2.128A.2.4, 1984.

[100] C. Temperton. "Self-sorting in-place fast Fourier transform." *SIAM Journal on Scientific and Statistical Computing*, 12 (4): 808823, 1991.

[101] Z. Qian, C. Lu, M. An, and R. Tolimieri. "Self-sorting in-place FFT algorithm with minimum working space." *IEEE Trans. ASSP*, 52 (10): 28352836, 1994.

[102] M. Hegland. "A self-sorting in-place fast Fourier transform algorithm suitable for vector and parallel processing." *Numerische Mathematik*, 68 (4): 507547, 1994.

[103] L. H. Thomas. "Using a computer to solve problems in physics." *Applications of Digital Computers*, 1963.

[104] S. C. Chan, and K. L. Ho. "On indexing the prime-factor fast Fourier transform algorithm." *IEEE Trans. Circuits and Systems*, 38 (8): 951953, 1991.

[105] S. Chu, and C. Burrus. "A prime factor FFT algorithm using distributed arithmetic." *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 30 (2), 217227, 1982.

[106] S. Winograd. "On Computing the Discrete Fourier Transform." In *Proc. National Academy of Sciences USA*, 73(4), 10051006, 1976.

[107] S. Winograd. "On Computing the Discrete Fourier Transform." *Mathematics of Computation*, 32(141), 175199, 1978.

[108] R. Tolimieri, M. An, and C. Lu. "Algorithms for Discrete Fourier Transform and Convolution." *Springer-Verlag*, 2nd ed., 1997.

[109] Lawrence R. Rabiner, Ronald W. Schafer, and Charles M. Rader. "The chirp z-transform algorithm and its application." *Bell Syst. Tech. J.*, 48, 1249-1292, 1969.

[110] Lawrence R. Rabiner, Ronald W. Schafer, and Charles M. Rader. "The chirp z-transform algorithm." *IEEE Trans. Audio Electroacoustics*, 17 (2), 8692, 1969.

[111] D. H. Bailey, and P. N. Swarztrauber. "The fractional Fourier transform and applications." *SIAM Review*, 33, 389-404, 1991.

[112] Lawrence Rabiner. "The chirp z-transform algorithm - a lesson in serendipity." *IEEE Signal Processing Magazine*, 21, 118-119, March, 2004.