HIGH-PERFORMANCE SPARSE FOURIER TRANSFORM ON PARALLEL ARCHITECTURES

A Dissertation

Presented to

the Faculty of the Department of Computer Science

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy

By

Cheng Wang

May 2016

HIGH-PERFORMANCE SPARSE FOURIER TRANSFORM ON PARALLEL ARCHITECTURES

Cheng Wang

APPROVED:

Dr. Barbara Chapman, Chairman
Dept. of Computer Science

Dr. Omprakash Gnawali
Dept. of Computer Science

Dr. Shishir Shah
Dept. of Computer Science

Dr. Weidong Shi
Dept. of Computer Science

Dr. Elebeoba E. May
Dept. of Biomedical Engineering

Dean, College of Natural Sciences and Mathematics

Acknowledgements

The study for Ph.D. in the past five years is intense and full of challenges. I could not have achieved anything without help from many people. I am grateful to my advisor, Dr. Barbara Chapman, for her invaluable guidance, support, and encouragement throughout this lengthy process. Her devotement and enthusiasm to the community motivated me in the Ph.D. pursuit. I would like to thank also my other committee members, Dr. Omprakash Gnawali, Dr. Shishir Shah, Dr. Weidong Shi, and Dr. Elebeoba E. May, who all took the time to review my work and offer their valued feedback.

I would like to express my sincere gratitude to my mentor, Dr. Sunita Chandrasekaran. I have been extremely fortunate to work with Dr. Chandrasekaran during the past five years. Without her constant support, guidance, patience, and persistence, this Ph.D. work would not have been possible.

As a student in the HPCTools research group, I had the privilege to work with and receive guidance from some senior members. Dr. Abid Malik, Dr. Dounia Khaldi, and Tony Curtis all spent considerable time in reviewing and offering appreciated suggestions for the content and scope of my dissertation. I am also grateful to have worked in such a great research environment, with people whom I consider good friends. I closely worked with Rengan Xu on developing an OpenACC compiler validation testsuite. I have had the great pleasure to work with Suyang Zhu and Peng Sun on developing a portable runtime system for supporting the programming on embedded systems. To all these students, and many more with whom I have engaged in valuable discussions over the years, in particular during the weekly group

meetings, thank you.

This work was supported by funding from Shell E&P. I would like thank Detlef Hohl for his generous support and insights on this Ph.D. work. I am very grateful to Mauricio Araya-Polo for his guidance for the project and during my summer internship at Shell. Huge thanks to compiler experts, Elana Granston and Eric Stotzer, who have been greatly supportive of the work on the Texas Instruments KeyStone II platform. I would like to extend a special word of thanks to Piotr Indyk and his research group at MIT who pioneered the sparse FFT work. And many thanks for their helpful discussions. I also had the opportunity to work in the Data Infrastructure Team at Facebook Inc. for a summer internship under the supervision of Sergey Edunov. I would like to extend my heartfelt thanks to all of them.

Last, but certainly not least, I would like to thank my family for all their love, encouragement, and understanding. I wish to thank my parents and my parents-in-law supporting me in all my pursuits. I wish to express my special gratitude to my wife, Danni Li, for her eternal love, understanding and believing she gives to me in these days and nights, especially during the tough times. I could never have done this without my family's unconditional support. All I have done is dedicated to them.

HIGH-PERFORMANCE SPARSE FOURIER TRANSFORM ON PARALLEL ARCHITECTURES

An Abstract of a Dissertation

Presented to

the Faculty of the Department of Computer Science

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy

By

Cheng Wang

May 2016

Abstract

Fast Fourier Transform (FFT) is one of the most important numerical algorithms widely used in numerous scientific and engineering computations. With the emergence of big data problems, however, in which the size of the processed data can easily exceed terabytes, it is challenging to acquire, process and store a sufficient amount of data to compute the FFT in the first place. The recently developed *sparse* FFT (sFFT) algorithm provides a solution to this problem. The sFFT can compute a compressed Fourier transform by using only a small subset of the input data, thus achieves significant performance improvement.

Modern homogeneous and heterogeneous multicore and manycore architectures are now part of the mainstream computing scene and can offer impressive performance for many applications. The computations that arise in sFFT lend it naturally to efficient parallel implementations. In this dissertation, we present efficient parallel implementations of the sFFT algorithm on three state-of-the-art parallel architectures, namely multicore CPUs, GPUs and a heterogeneous multicore embedded system. While the increase in the number of cores and memory bandwidth on modern architectures provide an opportunity to improve the performance through sophisticated parallel algorithm design, the sFFT is inherently complex, and numerous challenges need to address to deliver the optimal performance. In this dissertation, various parallelization and performance optimization techniques are proposed and implemented. Our parallel sFFT is more than 5x and 20x faster than the sequential sFFT on multicore CPUs and GPUs, respectively. Compared to full-size FFT libraries, the parallel sFFT achieves more than 9x speedup on multicore CPUs and 12x speedup on GPUs for a broad range of signal spectra.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Discrete Fourier Transform (DFT) is one of the most fundamental methods used in a wide variety of disciplines including audio, communication, wave simulations and cryptography. The most popular approach for computing the DFT is the Fast Fourier Transform (FFT) algorithm. Invented in the 1960s, FFT is the fastest known method which computes the DFT of an arbitrary signal of size n from/to time (space) domain to/from the frequency (wavenumber) domain, in $O(nlogn)$ time. FFT has been universal importance in scientific and engineering applications for a long time and was considered to be one of the top 10 most influential and important algorithms in the 20th century [21].

With the emergence of big data problems, however, in which the size of the

processed data can easily exceed gigabyte or even terabytes, it is challenging to acquire, process and store a sufficient amount of data to compute the FFT in the first place. For instance, in medical imaging, it is highly desirable to reduce the time that a patient spends in an MRI machine.

On the other hand, any algorithms for computing the DFT must take time at least proportional to its output size, which is $O(n)$, irrespective of the structure and *sparsity* of the data in the transformed domain. However, in many applications, the output of the FFT is *sparse*, i.e., most of the Fourier coefficients of a signal in the frequency domain are negligibly small or equal to zero with only a few dominant frequencies. Many applications of interest, e.g., audio, image, and video data, seismic signals, biomedical signals, financial information, social graph data, cognitive radio applications and many more massive data sets are sparse. In this case, the FFT is sub-optimal because $O(nlogn)$ operations on n input data points lead to only k number of significant outputs, where $k \ll n$, and $n - k$ zero/negligible small coefficients for a k-sparse output signal. It motivates the need for an algorithm that computes the FFT in sub-linear time, i.e., in an amount of time that is considerably smaller than the size of the processed data, and that uses only a subset of the input data.

The *Sparse* Fourier Transform (short for *sFFT*)[1] provides a precise solution to address this problem [42, 43]. As opposed to the FFT whose execution time is proportional to the size of the signal, n, the sFFT can use only a considerably small subset of the input data to compute the compressed FFT for the only small number

[1]For rest of the dissertation, we use the shorter form, *sFFT*, to refer the *sparse* FFT

of k large coefficients. Since sFFT runs in time proportional to the signal sparsity k, where $k \ll n$, it achieves significant performance improvements. Specifically, the sFFT employs special signal processing filters, notably the Gaussian and Dolph-Chebyshev filters [61], to sample the input signals and bin them into a small set of buckets; each bucket contains potentially only one large Fourier coefficient, of which the location and magnitude can then be determined.

Besides the algorithmic improvements, parallel architectures such as multicore CPU and GPUs have played a significant role in the practical implementations of many scientific and engineering applications. The computations that arise when big data problems are emerging lead it a nature path to improve the performance of sFFT through efficient parallel implementations.

Furthermore, the original Cooley/Tukey-FFT and similar algorithms have been well studied and implemented by the community and vendors. Such parallel FFT implementations include FFTW [29], Intel Math Kernel Library (MKL) [3] and Nvidia's cuFFT [7]. Those implementations exploit modern parallel computer architectures and are carefully designed to deliver the highest possible performance. Due to the novelty of sFFT, to the best of our knowledge, no such high-performance parallel implementation ever existed.

The goal of the dissertation work aims to make a solid contribution to developing high-performance parallel versions of sFFT for a variety of modern parallel architectures. The parallel implementations should be able to overcome the challenges and beat the performance of state-of-the-art full-size FFT implementations.

1.2 Challenges

Although the increase in the number of cores and memory bandwidth on modern parallel architectures present an opportunity to improve the performance through sophisticated parallel algorithm design, there are numerous challenges that need to addressed to deliver the optimal performance. In this section, we will discuss the key challenges in parallelization and performance optimization for sFFT.

1.2.1 Parallelism

Due to the novelty of sFFT, there is no much effort on parallelizing the algorithm. To the best of the knowledge, we are the first such effort that develops well-optimized parallel sFFT implementations.

Not surprisingly, though, the majority of functions in sFFT is not straightforward to parallelize. The challenges come with loop-carried dependence inherently exists in many pieces of sFFT, which is one of the major obstacles preventing an efficient parallel implementation. Furthermore, the huge need for thread synchronization in sFFT is another primary concern regarding the parallel efficiency. Since multiple threads might be accessing the shared resources simultaneously, they have to be carefully protected through thread synchronization mechanism (e.g., mutex locks, critical sections). Locks need to be avoided whenever possible and critical sections should be minimized otherwise they can be a potentially large performance bottleneck.

1.2.2 Porting sFFT to Diverse Architectures

In this dissertation work, we aim to develop parallel sFFT implementations for three different parallel architectures: multicore CPUs, GPUs, and a heterogeneous multicore embedded system.

Modern multicore CPUs (e.g., Intel Sandy Bridge typically with 8 to 16 cores) are optimally designed for workloads needing for low-latency. That is, modern CPUs strongly favor lower latency of operations with clock cycles in the nanoseconds, and we need to build techniques which can exploit these low latencies very well. GPUs, on the other hand, are throughput-oriented architecture. They work best on the problem sets which can be ideally solved by using massive fine-grained parallelism, using thousands or even tens of thousands of threads. Graphics processing is one such area with massive computational requirements, but where each of the tasks is relatively small, and often a set of operations are performed on data in the form of a pipeline. The throughput of this pipeline is more important than the latency of the individual operations. Heterogeneous embedded systems work differently from multicore CPUs and GPUs in the sense that they typically come with two or more different types of processors (e.g., ARMs + DSPs) in a single chip. Workloads are therefore partitioned into multiple smaller tasks which are scheduled running on different processors.

Although today's architectural trends clearly favor increasing parallelism, and scalability is the central objective in designing efficient parallel algorithms, an optimal

implementation requires the in-depth knowledge of the underlying processor's architecture, choosing a suitable parallel programming model and exploiting architecture-specific optimization techniques. That means different architectures may lean toward diverse parallelization and performance optimization techniques. For instance, efficient GPU programming typically requires careful scheduling of data movement between host CPUs and GPUs, and manipulating a kernel function to exploit the fine-grained massive parallelism by mapping threads to thousands of GPU cores. These are usually not the case for shared-memory multicore CPU architectures.

Therefore, the challenge comes from the fact that the parallelization and performance optimization technique which is best suitable for one architecture may be entirely unacceptable while moving to another. In Chapter 5 and 6, we will show some techniques which are suitable for multicore CPUs with 8 to 16 cores then become entirely problematic when we port the sFFT to GPUs with thousands number of cores.

1.2.3 Dynamic Irregular Memory Access Pattern

It is well-known that caches play a critical role in modern computer systems. Achieving high performance on such systems requires tailoring the memory reference patterns of applications to exploit the machine's memory hierarchy. The primary software approaches for improving the effective cache utilization in a program is increasing *temporal* and *spatial* locality through program transformation. In modern computer architectures, processor always moves blocks of contiguous data elements

into the cache, so whenever a program references a single array element, the entire enclosing block will be moved to the cache. Accordingly, *temporal locality* refers to reuse the same data element in the cache within a relatively short time duration before the cache block is evicted, eliminating the need for repeated access to the main memory. *Spatial locality*, on the other hand, occurs when memory locations mapped to the same cache block is reused before the block is evicted. The combination of temporal and spatial locality can minimize the traffic of transferring cache blocks across the machine's memory hierarchies.

For the regular memory access pattern which typically has stride-one array references, i.e., `A[i]`, the memory accesses are therefore consecutive. This regular memory access pattern allows the program to exploit the spatial locality, i.e., to reuse each cache blocks multiple times before it gets evicted from the cache. More importantly, *static* code analysis techniques allow the compiler to improve better temporal and spatial locality at compile time. For instance, the compiler transformation techniques such as loop blocking [17, 28, 30, 56, 86] and software prefetching [67, 79] have significantly improved the memory hierarchy utilization for regular applications.

The sFFT, unfortunately, falls into the other category: *irregular* applications. Such kind of irregular applications heavily utilizes *sparse* data structures, which are accessed through indirect array reference pattern in the form of `A[B[i]]`. The issue of irregular applications is that indirect array accesses often result in irregular memory reference pattern that exhibits poor temporal and spatial locality and consequently can lead to poor performance. In fact, only 5-10% of the processor's peak performance is typically achieved from irregular applications [9]. Poor data locality

is becoming even more of a performance challenge for multicore architectures where shared memory results in cores competing for memory bandwidth.

Even worse, the memory access pattern in irregular applications is usually *dynamic*: it remains unknown until runtime and often varies across the computation dynamically. It makes conventional compiler transformation techniques such as cache blocking [17, 56, 86] and loop reordering schemes [30, 63] hard to apply.

Alternatively, *runtime* transformation techniques[2] have been developed to exploit the data locality for irregular applications. The main idea behind the runtime transformation is to change computation order and data layout at runtime so that code is transformed to access the reordered data with potentially better temporal and spatial locality. These techniques can be categorized into two groups: data reordering and computation reordering. The former relocates the elements of data such that elements accessed closely in time are placed closing in memory space; the latter, on the other hand, changes the order in which data elements are referenced so that iterations accessing the same or adjacent elements are referenced consecutively in time.

The challenges inherent in employing runtime transformation techniques come with the complexities in finding out an optimal data layout, while at the same time, the need to amortize the overhead of data and computation reordering at runtime. Therefore, it must assume that the benefits of improved data locality will outweigh the cost of data and computation transformation. The work in this dissertation aims

[2] *Runtime* transformation means to transform the data layout and computation order at runtime

to explore an efficient and practical runtime transformation strategy that can effectively improve the data locality for sFFT with minimum transformation overhead. Chapter 8 will discuss the details of the technique.

In summary, it is a non-trivial task for developing high-performance parallel sFFT algorithms on diverse architectures. These challenges include minimizing the need for global synchronization that may impede the parallelism, a thorough understanding of the underlying processor architecture in order to choose the best suitable optimization techniques, and exploring a runtime transformation technique to exploit the data locality for irregular memory access patterns. The parallelization and optimization techniques used in this dissertation work address the challenges mentioned above for various architectures.

1.3 Objective and Contributions

The goal of the dissertation work aims to make a solid contribution to developing high-performance parallel sFFT implementations for a variety of modern parallel computer architectures. The parallel implementations should be able to overcome the challenges mentioned above and beat the performance of state-of-the-art full-size parallel FFT implementations.

Sequential sFFT implementation and performance improvement. Our first contribution in this dissertation work is we develop an optimized sequential implementation of the sFFT as the starting point for our later on parallel sFFT

implementations. Our sequential implementation is based on a prototype implementation obtained from the MIT's sFFT project website [41]. However, we largely reimplement the sFFT by carefully choosing the compact data structures for the ease of parallelization and performance optimization. We show that our optimized implementation achieves more than 1.5 times performance improvement than the original MIT's sFFT implementation.

Parallel sFFT implementations on diverse architectures. Our major contribution in this dissertation work is we develop parallel versions of the sFFT for three state-of-the-art multicore and massively parallel architectures. To the best of our knowledge, our work is among the first few efforts on developing parallel sFFT for multicore and accelerator-based architectures, and ours achieves the best performance among all the few published work [48].

Performance optimizations. As discussed above, the sFFT is inherently difficult to parallelize. In this dissertation work, we propose multiple optimization techniques that can effectively address the parallelization and performance challenges in parallelizing the sFFT. These optimization techniques can be primarily categorized into two groups based on the level of implementation: high-level and low-level approaches. High-level optimization techniques are architecture-independent and can easily migrate from one architecture to another. Low-level optimizations, on the other hand, mostly rely on the architecture-specific features, thus are hard to be portable to various architectures. In our dissertation work, we propose both high-level and low-level performance optimization techniques to deliver the best possible performance.

Results. We evaluate the performance of parallel sFFT with the sequential implementation. We show that the parallel sFFT significantly improves the performance of sFFT by the factor of 5x and 20x on multicore CPUs and GPUs, respectively. The promising result is due to our sophisticated parallel algorithm design and multi-levels of performance optimization techniques we exploit. Moreover, we also compare the performance of the parallel sFFT with the major full-size parallel FFT implementations. The experimental results show that our parallel sFFT implementations achieve more than 9x speedup on multicore CPUs and 12x speedup on GPUs for a broad range of single spectra. It offers a promising opportunity to replace the FFT routines by the sFFT in a vast majority of scientific and engineering applications and expect a significant performance improvement.

Pioneering in using OpenMP accelerator model. Last but not least, we port the parallel sFFT to a heterogeneous multicore embedded system by using OpenMP 4.0 accelerator model. OpenMP is a high-level programming model for shared-memory parallel programming. OpenMP 4.0 extends its execution model to support heterogeneous accelerator-based architectures. Due to the novelty of OpenMP 4.0 accelerator model and for the lack of mature compiler support, we are among the first effort to port sophisticated real-world applications to heterogeneous multicore embedded systems by using OpenMP 4.0. In this dissertation work, we will also report the first-hand experience in using OpenMP 4.0 and the lessons we learned toward high-level programming model for heterogeneous embedded systems.

1.4 Dissertation Organization

The rest of the dissertation is organized as follows. In Chapter 2, we first briefly survey the existing full-size FFT implementations. Then we review the current development of sFFT for the past of recent years. Finally, since sFFT is one of the irregular applications which pose unique challenges in exploiting data locality, we survey the most influential techniques on data and computation transformation techniques that can improve the data locality of irregular applications at runtime.

In Chapter 3, we introduce the sFFT from the numerical perspective. We break down the sFFT into major functional stages and present the algorithm step by step. We hope it can help the reader to understand better the parallel sFFT algorithm we are going to present later. Furthermore, we give an overview of some applications of the sFFT algorithm that emerged over the past few years.

Chapter 4 presents our sequential implementation of the sFFT. Our sequential sFFT implementation is based on the MIT's original C++ implementation but we largely reimplement using C. We will discuss the design choices and implementation details as well. Then, we evaluate the performance of our sFFT implementation with the MIT's original implementation. We show that our improved implementation makes over 2x speedup over the MIT's original implementation. Further, we also evaluate the sFFT against FFTW, one of the most widely used ordinary FFT libraries. We show that our sFFT implementation is considerably faster than FFTW for a broad range of signal size n and sparsity k. Finally, we study the time distribution of the major functional stages of sFFT and figure out the most time-consuming

step of the algorithm. We show that irregular memory access pattern is the primary performance bottleneck for sFFT.

Chapter 5 to Chapter 7 shows our major effort on the parallel sFFT for three divergent parallel architectures: multicore CPUs, GPUs, and a heterogeneous multicore system. We present the key challenges in parallelizing the sFFT at scale and the approaches to address the challenges. We also present our approaches on improving the data locality for the diverse architectures, respectively. We evaluate the performance of the parallel sFFT implementations with the major parallel full-size FFT libraries. The experimental results are promising regarding execution time, cache miss rate, average memory access latency, and parallel speedup.

In Chapter 8, we introduce a heuristic to improve further the data locality for sFFT. Since sFFT is one of the *dynamic* irregular applications, a runtime data and computation transformation technique is essential to further improve its performance. In Chapter 8, we propose an online transformation algorithm that reorders the data layout at runtime. We first analyze the inherent complexity in finding out an optimal data layout in general and reveal that designing a new data layout transformation algorithm could be reduced to a classic time-space tradeoff. Based on the insight, we present a new data transformation algorithm which complements the prior work with respective strengths. We apply the technique on sFFT and show the average 30% performance improvement.

Finally, Chapter 9 concludes the dissertation with directions of future work.

Chapter 2

Related Work

2.1 FFT Implementations

The original Cooley/Tukey-FFT and similar algorithms have been implemented by vendors and open-source communities such that they are well optimized for specific platforms. Those libraries include cuFFT [7] for NVIDIA's GPUs, AMD Core Math Libraries (ACML) [6] for AMD's APUs, and Intel Math Kernel Library (MKL) [3] for Intel processors. FFTW [29] is another widely used open-source FFT library portable to multiple x86-based architectures.

Due to the memory-bound nature of the FFT algorithm, its performance heavily depends on the design of memory subsystem and how well it can be exploited. There is some research working on FFT optimizations for various computer architectures [19, 33, 69].

2.2 Recent Development in Sparse FFT

The first *sublinear* sparse Fourier algorithm was proposed in [55]. Over the past few years, the topic has been extensively studied, from the algorithmic [42, 43, 52, 53], implementation [76, 80] and application [38] perspectives. These developments include the first deterministic algorithms that make no errors [11, 52, 53], as well as algorithms that, given a signal with k-sparse spectrum, compute the non-zero coefficients in time $O(klogN)$ [42] or even $O(klogk)$ [31, 58, 71, 75].

A recent breakthrough research from MIT [43] presented an improved algorithm and reduced the time complexity of sFFT to $O(logn\sqrt{nklogn})$. MIT's sFFT is faster than standard FFT for the sparsity k up to $(n/logn)$. The new algorithm employs a specific filter which achieves the runtime asymptotically faster than its prior studies. Therefore more applications with "denser" signal spectrum could also achieve performance gains from the sFFT algorithm.

There exists few sFFT implementations [41, 48, 76]. However, they are either only a sequential prototype implementation or an implementation that is barely optimized. To our best of knowledge, we are the first effort of developing parallel sFFT implementations on state-of-the-art parallel architectures, and we achieve the best performance so far.

2.3 Runtime Data and Computation Transformation

In this section, we survey the existing runtime transformation approaches for irregular applications. In general, prior transformation techniques can be categorized into two groups: computation reordering and data reordering.

Computation reordering reorders the iterations of the central loop enclosing the irregular references so that iterations accessing the same or adjacent data elements are consecutive in time. Figure 2.1 shows a simplified example of computation reordering. As is shown in the Figure, the original iteration sequence (1,2,3,4) has been transformed into the new order of (3,1,2,4). Thus, the array reference pattern becomes regular. Techniques for determining the suitable iteration order include lexicographical sort [24], bucket sort [66], z-sort [35], and so on. The main objective for computation reordering is to improve the temporal locality.

Data reordering relocates the data layout in the array such that the elements accessed closely in time are placed closing in memory space. As is shown in Figure 2.1, the original data layout (1,2,3,4) has been reordered as (2,3,1,4). After data reordering, iteration i accesses the data element i. Therefore, the irregular accesses have been eliminated. Data reordering improves the spatial locality. Because determining the optimal data layout at memory is a NP-hard problem in its general form [72], researchers have proposed various heuristics-based algorithms, including consecutive packing (CPACK) [25], Reverse Cuthill-McKee (RCM) [60], recursive coordinate bisection (RCB) [15], multi-level graph partitioning (METIS) [54], and

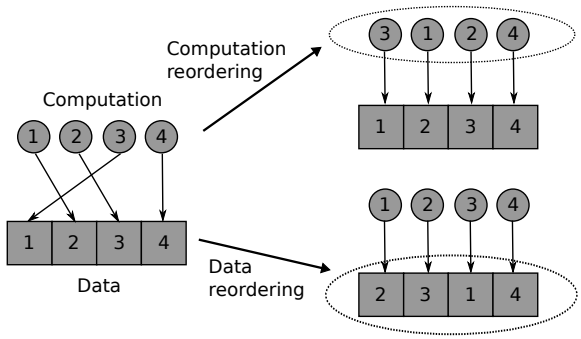


Figure 2.1: A simplified example of data and computation reordering

hierarchical clustering algorithm (GPART) [35]. Previous studies also found that in some cases, the combination of the data and computation reordering yield better results than each alone [25,35]. The rest of the chapter will review the prior approaches on runtime data and computation reordering. We will discuss the advantages and limitations of each approach.

2.3.1 Runtime Data Reordering Techniques

Consecutive Packing (CPACK)

The consecutive packing (CPACK) algorithm is a canonical runtime data reordering approach proposed by Ding and Kennedy [25]. The main idea behind the CPACK is that placement of data elements in memory in the order they are accessed should improve spatial locality. Therefore, CPACK employs a *first-touch policy* such that data will be moved into consecutive locations in the order the computation first accesses them. Implementing this first touch policy requires a single pass over the data access sequence.

One of the advantages of the CPACK algorithm is it has very low time complexity since it traverses the index array and the object array once and only once. Therefore, the time complexity of CPACK algorithm is linear to the size of index array, which is $O(n)$. It needs to note that, however, the first-touch policy is a greedy strategy so it does not consider the affinity of the data [27], i.e., how frequently two elements are accessed together. For instance, given a data access pattern `x y w x z u x z`, where the cache line size is 2, CPACK will simply pack `x` and `y` into the same memory block since they are firstly touched together. However, it is clearly to see that packing `x` and `z` should yield better locality since they would improve the locality for the two accesses to `z`, compared to the original placement which can only improve the data locality for the access to `y`.

Hierarchical Clustering Algorithm (GPART)

Hierarchical clustering algorithm (GPART) is another widely used data layout transformation algorithm proposed by Han *et. al* [35]. The main idea of the GPART bases on the observation that scientific codes use two or more distinct irregular references within a loop iteration. In this case, GPART optimizes the data layout by abstracting the underlying mapping relationship between computation and the data layout as a graph; computations can be viewed as edges connecting array elements represented with nodes. This abstraction results in a graph view out of data access patterns. As a result, data locality optimizations can then be mapped to a graph partitioning problem. Partitioning the graph and putting nodes in a partition close in memory naturally improves spatial and temporal locality [12]. Finding an optimal graph

partition is an NP-hard problem, and several heuristics are therefore proposed [12].

GPART first sorts the nodes in descending order based on the degrees, i.e., the number of incident edges. It then visits the nodes one-by-one and arbitrarily chooses one node among the neighboring nodes and group them together. However, similar to the limitation of the CPACK, GPART arbitrarily chooses one of the neighboring nodes to group together, it can easily miss a better layout for some data elements. Furthermore, GPART has higher time complexity than CPACK due to the operations of sorting and graph partitioning. Nevertheless, GPART benefits for the scenario that multiple irregular references in a loop iteration.

Reverse Cuthill-McKee (RCM)

Alike the GPART, Reverse Cuthill-McKee (RCM) is a graph-based approach to reordering the data layout for irregular computations. Rather than the GPART, which sorts the nodes based on the degrees, RCM simply uses reverse breath-first search (BFS) to reorder data. Thus, RCM has lower transformation overhead.

Space Filling Curve (SFC)

Similar to GPART and RCM, Space-Filling Curve (SFC) [64] also employs geometric coordinate information to reorder data layout in memory. Instead of using a graph, SFC relies on space-filling curves which are continuous, non-smooth curves that pass through each point in a finite k-dimensional space. Examples include Morton and

Hilbert curves. Since the interactions tend to be local, reordering data using space-filling curves reduces the distance in the memory between two geometrically nearby points in the memory, thus, yields better locality.

Recursive Coordinate Bisection (RCB)

One major limitation of the SFC algorithm is it works well only when data is uniformly distributed in memory. When data is unevenly distributed, an alternative approach is recursive coordinate bisection (RCB) [15]. The key idea of the RCB is it recursively splits each dimension into two (equally) partitions by finding the median of data coordinates in that dimension. The process is recursively repeated with alternative dimension. After the partition, data within a partition are stored consecutively thus the locality is improved.

Multilevel Graph Partitioning (METIS)

A major limitation of the SFC and RCB is that geometric coordinates information is needed for each node. This information may not be available for some applications. In comparison, graph partitioning algorithms such as METIS [54] can be applied naturally based on the data reference pattern embedded in loops.

Discussion

In summary, METIS and RCB have significantly greater time complexity than CPACK and GPART, and they are quite expensive when used for cache optimizations. Some

early studies indicate that CPACK and GPART outperform METIS and RCB in most cases [35, 87]. Furthermore, it is important to note that RCB and METIS are canonical algorithms in graph partitioning and load balancing. They play an significant role in distributed-memory programming where an optimal data partitioning and load balancing could minimize the inter-node communications. In such field, greater processing time is acceptable yet not the case for cache optimizations. Consequently, METIS (RCB) and the objectives of this dissertation each addresses different aspects of data locality: the former crosses nodes and aims to minimize communication in distributed-memory space, while the later crosses cores in a node and seeks to reduce cache misses in shared-memory space.

2.3.2 Runtime Computation Reordering Techniques

Bucket Sort

If each iteration performs only one irregular access (e.g., Figure 2.1), sorting the loop iteration by the indexes of access data may result in a better temporal locality. For instance, for the index array of `a b c a`, sorting the array in the ascending order, i.e., `a a b c` can reorder the computation such that accesses the two `a` consecutively and the temporal locality is therefore improved. Bucket sort with cache-sized buckets is commonly used [66] to reduce the cost on sorting. That is because bucket sorting can achieve nearly linear time complexity given the input data (index array) is uniformly distributed. Also, bucket sort is usually more cache-friendly compared with other sorting algorithms if the buckets can fit into the cache.

Lexicographical Sort (LEXSORT)

For loop iterations with multiple irregular accesses, applying the bucket sort is non-trivial. Instead, lexicographical sorting (LEXSORT) can be used to sort computations with multiple irregular access patterns [24]. The idea of LEXSORT is to sort the index array in the lexicographical order (*a.k.a* dictionary order). Figure 2.2 shows an example of computation reordering using lexicographical sort. The circle represents the loop iteration order while the square represents the array of data elements that the loop iteration accesses. The alphabet letters from `a` to `e` denotes the location of each data element (We use alphabet letters instead of numbers is to distinguish from the index array). As is shown in the figure, each loop iteration has two irregular accesses. For instance, iteration `1` accesses to data elements in locations `b` and `c`. After the LEXSORT, the loop iteration is reordered as the lexicographical order; thus, the temporal locality can be substantially improved. For example, if we assume that each cache block can hold only two elements, and there is only one block in the cache at a time (unit-line cache). In the example of Figure 2.2, there are five cache misses in the original data reference pattern while the number of cache misses is reduced to 3 after the lexicographical sort.

2.3.3 Hybrid Approach

As discussed above, data and computation reordering can effectively improve the data locality when they are used separately. Recent studies also show that the combination of the computation and data reordering will yield better results in most

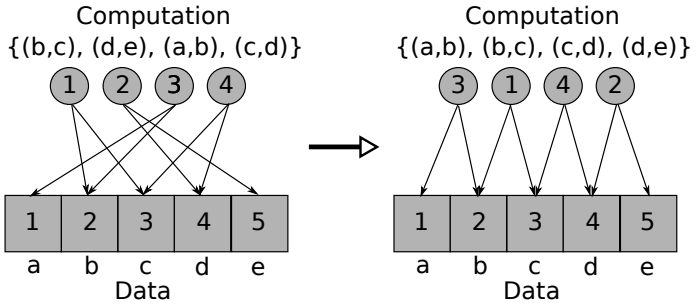


Figure 2.2: Computation reorder through lexicographical sort

cases [25, 35]. Ding and Kennedy [25] demonstrated that applying computation reordering after data reordering further reduces the cache miss rate from 7.48% to 0.25%, a factor of 30, on a 2K cache with 16-molecule cache lines.

2.4 Summary

In summary, this section briefly overviews the existing work in FFT implementations. Also, we survey the recent development in sFFT. Furthermore, since sFFT is one of the irregular applications, we review the major approaches on improving the data locality through data and computation transformation techniques at runtime.

Since it is an NP-complete problem to find an optimal data layout with minimum cache misses in general, it is virtually infeasible to keep searching for a general, practical algorithm that can yield the best memory layout, either with or without hardware extensions, through data reordering, computation transformation, or their combinations – the three methods that have been pursued by most existing work.

This section we reviewed key algorithms used for runtime data and computation

reordering. Most of these prior approaches are based on a vague but well practiced rule: arranging data elements which are accessed together in a short period into nearby locations in memory increases the likelihood that cache blocks will be reused multiple times before they get evicted from the cache. Although the prior studies show promising results for improving the locality of irregular applications, much fewer efforts has been spent on understanding the fundamental characteristics of data access patterns exhibited by programs and interactions with a specific data and computation reordering approach. As a result, they either do not provide any performance guarantee or are effective to only some limited scenarios.

In this dissertation work, we fundamentally study the inherent complexity of the problem. We then reveal that the essence of designing a runtime transformation algorithm can be reduced to a canonical complexity of time-space tradeoff. Based on that insight, we propose a novel data reordering algorithm in Chapter 8 which can circumvent the limitations of the previous methods and improve the data locality for irregular applications such as sFFT.

Chapter 3

Sparse Fourier Transform – An Overview

3.1 Overview

In this chapter, we briefly introduce the Sparse FFT (sFFT) from the view of a numerical algorithm. We break down the sFFT into several functional stages and present the algorithm in a step-by-step approach. We believe that this will help the reader to understand better the parallel algorithm in later chapters. We do not cover the theoretical analysis and proofs of the algorithm in this dissertation. The reader is referred to the original paper in [43]. However, we will present the implementation details and an empirical analysis of the performance of the algorithm in Chapter 4.

3.2 Sparse FFT

Discrete Fourier Transform (DFT) is a fundamental numerical algorithm used in a wide variety of disciplines including audio, communication, wave simulations and cryptography to name a few. It has been of universal importance in scientific and engineering applications for a long time. Fast Fourier Transform (FFT) [21] is the fastest known approach that computes the DFT of an arbitrary n-length signal from/to time (space) to/from the frequency (wavenumber) domain, with a computational complexity of $O(nlogn)$.

With the emergence of big data problems, in which the size of the processed data can easily exceed gigabyte or even terabytes, it is challenging to acquire, process and store a sufficient amount of data to compute the FFT in the first place. On the other hand, any algorithm for computing the FFT must take at least $O(nlogn)$ time to its input data size, irrespective of the structure and sparsity of the data in the transformed domain. In many applications, however, the output of the FFT is *sparse*, i.e., most of the Fourier coefficients in the transformed domain are negligibly small or close to zero while only a few of them are significant. In this case, the full-size FFT is sub-optimal because $O(nlogn)$ operations on n input data points lead to only a few number of k non-zero/significant outputs, where $k \ll n$, while the rest of $n - k$ coefficients are zero/negligible small.

Many applications of interest, e.g., audio, video, medical images [14, 77], GPS signals [36, 39], seismic data, biomedical signals [78], financial data, social graphs, cognitive radio and many more massive data sets can fall into the sparse Fourier

spectrum. For example, a typical 8x8 block in a video frame has on average 7 non-negligible coefficients (i.e., 89% of the coefficients are negligible) [18]. Images and audio data are equally sparse. This sparsity provides the rationale underlying compression schemes such as MPEG and JPEG [43]. This motivates the need for an algorithm that can compute the Fourier transform in *sub-linear* time, i.e., in an amount of time that is considerably smaller than the size of the data, and that use only a subset of the input data.

The recently developed *Sparse* Fourier Transform algorithm (sFFT) [43] provides a precise solution to address this problem. Unlike the conventional FFT, which computes the entire input data with size n, the sFFT, on the other hand, can use only a considerably small subset of the input data to compute a compressed FFT for only the number of k largest output coefficients, thus achieves substantial performance improvements. Specifically, the sFFT employs special signal processing filters, notably the Gaussian and Dolph-Chebyshev filters, to sample the input signals and bin them into a small set of buckets. Since the signal is sparse in the frequency domain, each bucket is likely to contain only one large Fourier coefficient, of which the location and magnitude can then be precisely determined. The sFFT achieves a runtime of $O(logn\sqrt{nklogn})$, which is faster than FFT for k up to $O(n/logn)$.

3.3 Computational Stages in Sparse FFT

In this section, we outline the basic components and techniques used in the sFFT algorithm. We partition the algorithm into several major steps by reducing it to

several sub-problems.

3.3.1 Notation

For an input signal $x \in \mathbb{C}^n$ with size n, its Fourier spectrum is denoted by $\hat{x}$. The signal sparsity k, is defined as the number of non-zero Fourier coefficients in $\hat{x}$. G is the flat window function, while $\hat{G}$ denotes its spectrum in the frequency domain.

3.3.2 Stage 1: Random Spectrum Permutation

The sFFT starts with binning large Fourier coefficients into a small number of buckets by convoluting the permuted input signal with a well-designed filter (will be discussed in Subsection 3.3.3). The first challenge comes with how to deal with spectra in which two large coefficients are too close to each other, and thus cannot be easily isolated via binning. To guarantee that each bucket receives only one large Fourier coefficient, which can then be accurately located (to find its position) and estimated (to find its value), the sFFT algorithm randomly permutes the input signal so that the adjacent Fourier coefficients in the frequency domain are evenly separated. In addition, the distance between the original location and permuted location should be largely enough so that adjacent coefficients are not be binned into the same bucket.

The sFFT employs a hashing-based spectrum permutation method to address issue. Specifically, it defines a "hash" function that maps indices of the original signal spectrum to the permuted locations so that the original locations can then be recovered at the end of the algorithm.

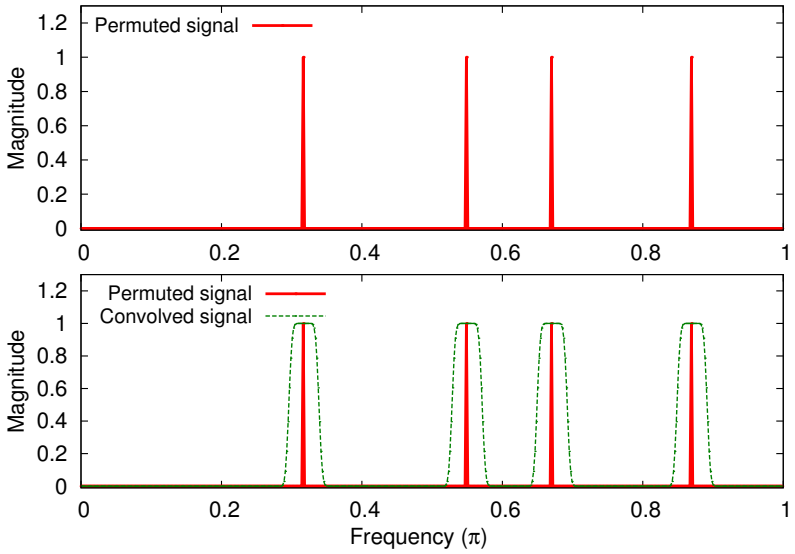


Figure 3.1: Random signal spectrum permutation and filtering with flat window functions. This example has the parameter $k = 4$, i.e., we select the top 4 largest samples

Definition 1 *Define the spectra permutation $P_{\sigma,\tau}$ such that, given an n-dimensional input signal x, an random integer σ that is invertible mod n, and an offset integer $\tau \in [n]$, $(P_{\sigma,\tau}x)_i = x_{\sigma i+\tau}$. Then $(\widehat{P_{\sigma,\tau}x})_{\sigma i} = \hat{x}_i\omega^{-\tau i}$.*

According to the definition 1, permuted signal in the time domain with a shifting factor of τ will lead to phase changes in the frequency domain. It helps in separating the spectra and bin coefficients to the correct buckets. Figure 3.1 (top) shows the permuted signal in which significant frequencies are well separated.

3.3.3 Stage 2: Flat Window Function

For the sFFT algorithm to be sublinear in time, only partials of the input signals can be used to compute the FFT. It is achieved by grouping subsets of the Fourier

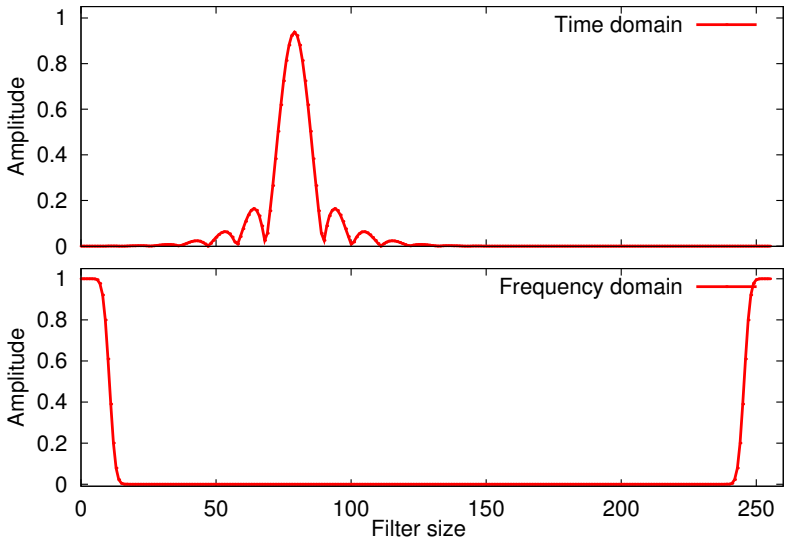


Figure 3.2: An example of flat window function in time (top) and frequency (bottom) domain. This example has the window size $n = 256$

coefficients into a small number of buckets. Since each bucket contains only a single frequency (i.e., frequencies are isolated via the spectrum permutation as was discussed in Subsection 3.3.2), then each bucket can recover the frequency separately. It leads to the sample complexity and the execution time is proportional to the number of buckets, which is lower bounded by the signal sparsity k.

Merely sampling a handful of data points out of the input signal, however, is impossible because it will lead to spectral leakage, That is, the discontinuities introduced by splicing the signal will appear as sharp components spread out in the frequency domain. To minimize this effect, sFFT employs a flat window function working as a filter to "smooth" the process. Specifically, the sampled signal will convolve with the filter and bin into one of a small number of buckets, as is shown in Figure 3.1 at the bottom. The sFFT employs a Gaussian and Dolpstartsh-Chebyshev

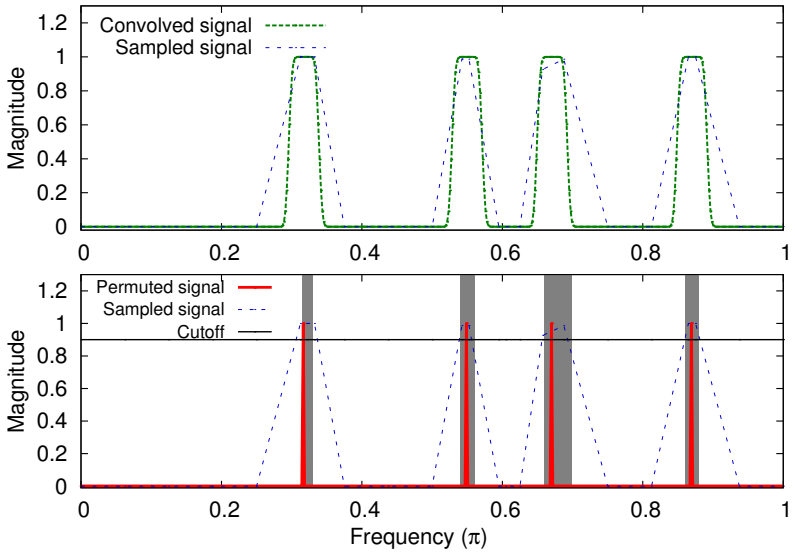


Figure 3.3: Subsampled FFT (top) and Cutoff function (bottom). This example has the parameter $k = 4$, i.e., we select the top 4 largest samples

filter G, depicted in Figure 3.2. The reason is based on the observation that the Gaussian and Dolpstartsh-Chebyshev filter concentrates both in time and frequency domain, i.e., G is close to zero except at a small number of *pass* regions, and its Fourier transform $\hat{G}$ is negligible except for a fraction of coefficients with an exponential tail outside of it. It guarantees the minimal leakage from frequencies to other buckets.

3.3.4 Stage 3: Subsampled FFT

In the signal permutation and filtering stages, the permuted signal gets the rate of B sampled and binned into the number of B buckets. It permits to transform just a fraction of the entire signal. As a result, instead of computing the N-dimensional FFT, it can compute the subsampled B-dimensional FFT, where B is the number of

buckets, in $O(BlogB)$ time. As is shown in Figure 3.3 (top), the algorithm computes the B-dimensional subsampled FFT as an outline of the full-size convoluted signal.

3.3.5 Stage 4: Cutoff

After Stage 3, we get the number of B buckets at frequency domain. Each bucket contains at most one potential large coefficient. In the k-sparse signal spectrum where $k \ll B$, it is still highly likely that many of the buckets are close to zero. Furthermore, the algorithm guarantees that each large coefficient has a low probability of being missed if we select the top $O(k)$ samples. Therefore, in this step the size of the data to be processed is further reduced by selecting only the top k coefficients of maximum magnitude. It can be done effectively through a quick selection algorithm which can select the top k largest elements from a set of B buckets.

3.3.6 Stage 5: Reverse Hash Function for Location Recovery

After removing non-significant coefficients in Stage 4, the selected coefficients have to be reconstructed, by finding the original locations in the frequency domain and estimating the magnitudes.

Stages 1 to 4 define a *hash function* $h_\sigma : [n] \rightarrow [B]$ that maps size of n input signals to the number of B buckets. In order to find the original locations in the frequency domain, this hash function has to be reversed by removing the phase changes due to the permutation stage. This is done by computing the reverse hash function h_r.

Algorithm 1 Outer loop of sFFT

1: **Input:** signal $x \in \mathbb{C}^n$ with size n, signal sparsity k
2: **Output:** A k-sparse vector $\hat{x}$

1. Run a number of L *location loops*, returning L sets of coordinates $I_1, \cdots, I_L$.
2. Count the number s_i of occurrences of each found coordinate i, which is $s_i = |\{r | i \in I_r\}|$.
3. Only keep the coefficients which occurred in at least twice in the location loops. $I' = \{i \in I_1 \cup \cdots \cup I_L | s_i > L/2\}$.
4. Run a number of L *estimation loops* on I', returning L sets of frequency coefficients $\widehat{x^r_{I'}}$.
5. Estimate each frequency coefficient x_i as $\widehat{x}_i = median\{x^r_i | r \in \{1, \cdots, L\}\}$.

Stages 1 to 5 run for a number of $L = O(logn)$ *location loops* with different permutation parameters σ and τ, and return the L sets of locations of candidate coefficients $I_1, \cdots, I_L$. For each output of the location inner loop I_i, it counts the number s_i of occurrences of each found coefficient i, that is $s_i = |\{r | i \in I_r\}|$, and only keep the coefficients which occurred in at least twice in the location loops. $I' = \{i \in I_1 \cup \cdots \cup I_L | s_i > L/2\}$.

3.3.7 Stage 6: Magnitude Estimation

In this stage, given the sets of locations I' and frequencies $\widehat{x^r_{I'}}$ from location loops, it estimates each frequency coefficient $\widehat{x}_i$ as $\widehat{x}_i = median\{x^r_i | r \in \{1, \cdots, L\}\}$. The median is taken in real and imaginary components separately.

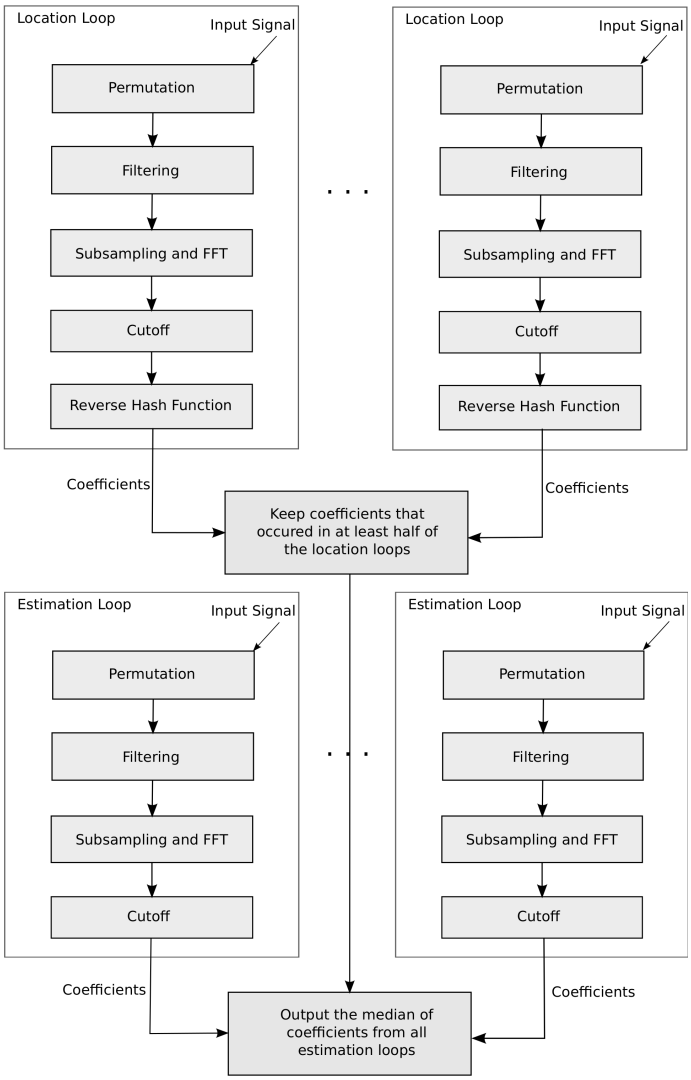


Figure 3.4: A simplified flow diagram of sFFT outer loop

3.3.8 Outer Loop

The sFFT is an iterative-based algorithm running on the number of $L = O(logn)$ outer loops. Each iteration of the outer loop executes the all the stages (we called it as the *inner loop*) as was discussed above. By running multiple iterations of the inner loop, the sFFT algorithm is guaranteed to have a probability to find out the locations and recover the magnitude of the large Fourier coefficients. Algorithm 1 shows an iteration of the outer loop. A simplified flow diagram is shown in Figure 3.4.

3.4 Sparse FFT 2.0

sFFT 2.0 is very similar to the original sFFT (referred as sFFT 1.0). The only difference is that sFFT 2.0 applies a heuristic in the location recovery stage (described in Subsection 3.3.6) and is able to find out the large Fourier coefficients quickly. The idea of the heuristic is to apply a special filter, namely *Mansour filter* [62], to restrict the locations of the large coefficients. Let M be the size of the *Mansour filter*, it does the following during a pre-processing stage:

1. Choose $\tau \in [n] uniformly at random.$
2. For $i \in [M]$, set $y_i = x_{i(n/M)+\tau}$.
3. Compute $\hat{y}$ as the FFT of y.
4. Output the coefficients of maximum magnitude in $\hat{y}$.

Remember that the flat window function is mainly used to sample the input signal and minimize the effect of spectral leakage. The major advantage of Mansour filter is it has no spectral leakage at all. Since the error rate is reduced, it needs fewer iterations to get the desired result thus the execution time is reduced. [1]

3.5 Sparse FFT Applications

In this section, we give an overview of some of the data-intensive applications of the sparse FFT algorithms and techniques that emerged over the last few years. These applications involve, for example, GPS receivers, cognitive radios, and, more generally, any analog signal that we wish to digitize. It is these applications that we focus on as they highlight the role of sparse FFT algorithms in the signal processing of big data.

Spectrum sensing. The goal of a spectrum sensing algorithm is to scan the available spectrum and identify the "occupied" frequency slots. In many applications, this task needs to be done quickly since the spectrum changes dynamically. Unfortunately, scanning a GHz-wide spectrum is a highly power-consuming operation. To reduce the power and acquisition time, The approach present in [37] uses the sparse FFT to compute the frequency representation of a sparse signal without sampling it at full bandwidth. One observation made from the paper is that the signal spectrum is *sparsely* occupied at frequency domain. As a result, the sFFT is able to compute the Fourier transform of a sparse signal faster than the FFT,

[1] Due to the performance improvement in sFFT 2.0, unless stated otherwise, we will refer sFFT as sFFT 2.0 for the rest of chapters in this dissertation.

reducing the time and energy cost on baseband processing.

GPS synchronization GPS is one of the most widely used wireless systems. A GPS receiver has to lock on the satellite signals to calculate its position. The process of locking on the satellites is quite costly and requires hundreds of millions of hardware multiplications, leading to high power consumption. The fastest known algorithm for this problem is based on the Fourier transform and has a complexity of $O(nlogn)$, where n is the number of signal samples.

The paper [40] uses sparse FFT techniques to speed-up the process. The improvement is based on the following observation: since the output of the inverse FFT contains a single peak corresponding to the correct shift, the inverse step can be implemented using the sparse FFT. In fact, since $k = 1$, the algorithm is particularly simple and relies on a simple aliasing filter. Furthermore, since the sparse inverse FFT algorithm uses only some of the samples of the product, it suffices to compute only those samples. This reduces the cost of the forward step as well. The experiments on real signals show that the new algorithm reduces the amount of computations by a factor of more than 2.2.

3.6 Summary

In this chapter, we show an overview of the major functional stages in the sFFT algorithm from the numerical algorithm point of view. We hope this will help reader to better understand our parallel sFFT algorithm discussed later. Moreover, we survey multiple data-intensive applications with heavy need of Fourier Transform.

The applications show performance improvement by using the sFFT.

In next chapter, we will present our optimized sequential implementation of the sFFT algorithm, which will be served as a starting point for the further parallel implementations. The performance evaluations will also be discussed in next chapter.

Chapter 4

Sequential Implementation and Performance Evaluation

4.1 Overview

In this chapter, we present our sequential implementation of the sFFT, which serves as a starting point for the parallel implementations. We largely reimplement the algorithm based on the prototype implementation from the MIT project website [41][1]. Some design choices and implementation details will be discussed in Section 4.2.

Second, we evaluate the performance of the UH sFFT implementation in Section 4.3. We demonstrate that our optimized sFFT implementation is more than 2x faster than the MIT's original implementation. Further, we also evaluate the

[1]For the following of this section, we use the term"MIT sFFT", refers to the MIT's original prototype implementation, and use "UH sFFT" denotes our optimized implementation.

performance of our sFFT implementation against FFTW, one of the most widely used standard FFT libraries [29]. We show that sFFT is faster than FFTW for a considerable range of signal size n and sparsity k, without the loss of numerical accuracy.

In Section 4.4, we profile the major functional stages in sFFT and point out the most time-consuming stages of the algorithm. Then, we perform performance analysis through theoretical and experimental approaches. The experimental result shows that the irregular memory access pattern is the root cause of the performance bottleneck in sFFT. The details of the results will be discussed in Section 4.5.

4.2 Sequential Implementation

In this section, we present our sequential implementation of the sFFT algorithm. Our sequential implementation is based on a prototype implementation we obtained from the MIT's project website [41]. The MIT's implementation is developed by using C++ and utilized the Standard Template Library (STL) for simplicity. However, we largely reimplement the sFFT for the following reasons.

First of all, the MIT's implementation is mainly for proof-of-concept purpose. It is sequential only and does not take any consideration of the parallelism inherently in sFFT. Some data structures used in the original implementation are not thread-safe, and certain code structures carry on loop-carried dependence. It makes parallelizing the algorithm directly from the MIT's implementation non-trivial.

Table 4.1: Major data structure and interfaces changed from the MIT's implementation

C++	**C**	**Description**
std::map$\langle int, complex \rangle$	struct$\{int, complex\}$	Stores the key-value pair of the location and value of the non-zero Fourier Coefficients
std::pair$\langle int, int \rangle$	struct$\{int, int\}$	Stores the original and permuted location of the signal spectrum
std::vector$\langle T, Alloc \rangle$	Arrays	Container represents arrays that can change in size. Calculated the needed size before allocating the array
std::sort$(\cdots)$	qsort$(\cdots)$	Quick sort
std::upper_bound(first, last, val)	Manually implemented based on binary search	Returns an iterator pointing to the first element in the range [first,last) which compares greater than val
std:nth_element(first, nth, last)	Used a wrapper function	Rearranges the elements in the range [first,last), in such a way that the element at the nth position is the element that would be in that position in a sorted sequence.
std::map::count$\langle k \rangle$	Manually implemented based on binary search	Searches the container for elements with a key equivalent to k and returns the number of matches
complex	struct$\{real, real\}$	Decouple the complex type to a struct of real and imaginary part

Second, the MIT's implementation does not take any advantage of modern computer architectures. So it does not deliver the highest possible performance. On the other hand, many other high-performance standard FFT libraries such as FFTW [29] and cuFFT [7] are highly optimized to exploit the modern computer architectures. Thus, it is of crucial importance to develop a high-performance sFFT implementation.

For all the reasons above, we largely reimplement the sFFT algorithm using C

instead of C++ based on two major observations. First of all, compared to the standard FFT, the sFFT suffers from the low compute-to-memory ratio, as well as indirect and irregular memory access patterns. Achieving higher performance on modern computer architecture requires choosing a compact data structure which can best exploit the memory hierarchy. However, the original MIT's implementation largely utilizes C++ standard collection-based data structures such as containers. While those data structures provide more flexibility and simplicity, it comes at the cost of space overhead. Moreover, it creates additional levels of indirection, which suffers from inefficient cache utilization. For instance, in the MIT's implementation, it uses the `std::map` to store the pair of location and value of the estimated coordinates in the frequency domain. The `std::map` is usually implemented as red-black trees which have average access time $O(logN)$ (N is the number of key-value pairs in the tree). In our implementation, however, we choose to use flat and compact data structures such as where all data is in primitive type and stored sequentially in memory. It leads to better cache utilization and data locality.

Second, a simpler code structure in C is more compiler-friendly. That is, it makes the compiler easier to exploit its code optimization techniques, which is particularly important for achieving high performance on modern massively parallel architectures such as GPGPUs and Intel Xeon Phi. Table 4.1 shows the major data structure and functional interface changes from the MIT's implementation. Other code optimization techniques will be discussed in Chapter 5.

4.3 Performance Evaluation

In this section, we evaluate the performance of three compared algorithms: 1)MIT sFFT implementation, 2)UH sFFT implementation and 3) standard FFTW (full-size standard FFT library). We will demonstrate that the UH sFFT implementation is more than 1.7x faster than the MIT's original implementation. Furthermore, we will show that sFFT is faster than FFTW for a considerable range of signal size n and sparsity k.

4.3.1 Experimental Setup

Table 4.2 shows the experimental setup for the performance evaluation. We run the experiments on an Intel Sandy Bridge architecture, which contains six cores per socket running at the frequency 2.5 GHz with two sockets in total. The memory sub-system has three levels of cache: the L1 and L2 cache has the size of 32 KB and 256 KB each, and is private to each core, respectively. The L3 cache, with the size of 15 MB, is shared by all the six cores per socket. The DDR memory on the test machine has the total size of 65 GB, largely enough to fit the large data sets into memory.

The compiler version used to build the sFFT is gcc-4.8.2 with optimization level `-O3` enabled. The FFTW version is 3.3.4 [29], configured with multi-threading supported. Furthermore, we use the Valgrind 3.8.1 [68] to collect the cache performance data. The details will be discussed in Section 4.5. We run the experiments at least for five times and calculate the average value.

Table 4.2: CPU test-bench

Processor	**Intel(R) Xeon(R) CPU E5-2640 (Sandy Bridge)**
Core per Socket	6
Core Frequency	2.50 GHz
L1 Cache	6 x 32 KB D/I, 8-way, 64 B line size
L2 Cache	6 x 256 KB, 8-way, 64 B line size
L3 Cache	15 MB, 20-way, 64 B line size
TLB Size	4 KB per page
Memory Size	65 GB in total

4.3.2 Experimental Results – Double Precision

In this subsection, we evaluate the performance of our optimized sFFT implementation (i.e., UH sFFT) with the MIT's original version. For completeness, we also compare against the FFTW, one of the most commonly used standard FFT library with a runtime complexity of $O(nlogn)$. In this subsection, all the data sets are double-precision floating point numbers. We will evaluate the results of single-precision numbers in next subsection.

The test signals are generated in the same approach as for in the MIT's original implementation [41]. Specifically, k frequencies are selected uniformly at random from the size of n input signal and normalized a magnitude of 1. The rest frequencies are set to zero with additive white Gaussian noise. In all experiments, the parameters of sFFT are chosen so that the average L1 error is no more than the order of `1e-7` per non-zero frequency. Therefore, the error rate of sFFT is acceptably small. We follow the documentation of the MIT's original implementation for choosing the error mentioned above [41].

Execution Time vs. Signal Size N

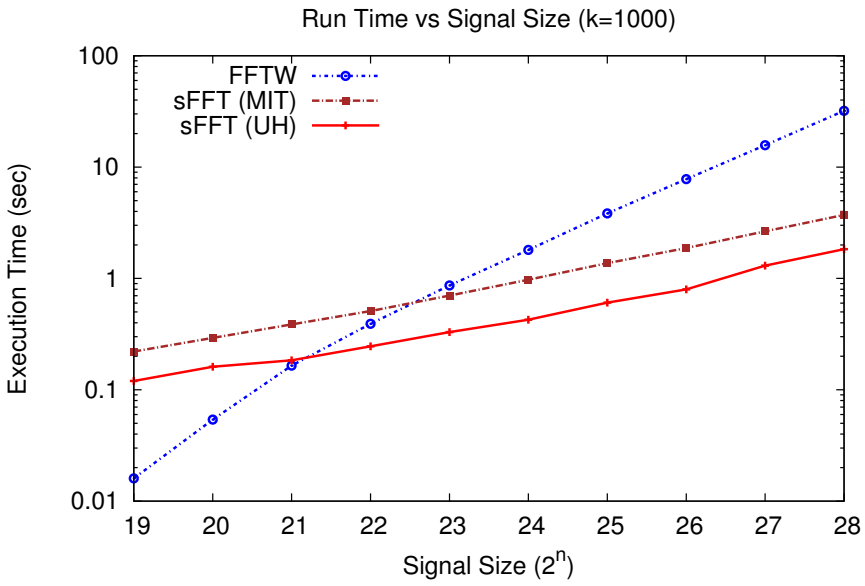
In this experiment, we fix the signal sparsity $k = 1000$ and report the execution time of the three compared algorithms for signal sizes n ranging from 2^{19} to 2^{28}. Figure 4.1(a) plots the average execution time of UH sFFT, MIT sFFT and FFTW.

As expected, Figure 4.1(a) shows that the UH sFFT is constantly faster than MIT sFFT implementation. Table 4.3(a) shows speedup of UH sFFT over the MIT's implementation. It can be seen from the table that the average performance improvement over 2x.

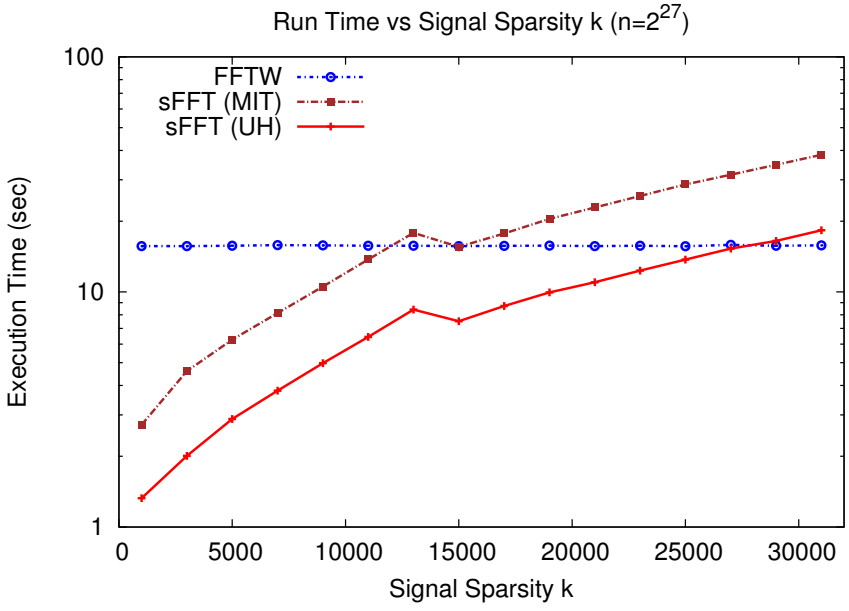
Compared to FFTW, Figure 4.1(a) shows that the execution time of sFFT and FFTW are approximately linear in the log scale. However, the slope of the line of sFFT is less than the slope of FFTW, which is a result of the sublinear runtime of sFFT. As a result, the performance gap between sFFT and FFTW diverges greatly with the increase of signal size n. This gives more credit to sFFT for large input data sets.

Figure 4.1(a) also shows that the UH sFFT becomes faster than FFTW when the signal size is equal or greater than 2^{21} (i.e., 2,097,152) at recovering the exact 1000 non-zero large coefficients. Compared to UH sFFT, the cross point for MIT sFFT is at round 2^{23} (8,388,608), requiring 4x greater sized signal to beat over the performance of FFTW.

Table 4.3(a) (and Figure 4.2(a)) also shows the speedup of sFFT over FFTW at $k = 1000$ and n ranges from 2^{19} to 2^{28}. It can be seen from the table that UH sFFT is faster than FFTW from 0.13x to 17.5x with the increase of the signal size n, while



(a) Execution time vs. signal size ($k = 1000$)



(b) Execution time vs. signal sparsity k ($n = 2^{27}$)

Figure 4.1: Performance evaluation of UH's sFFT implementation vs. MIT's implementation and FFTW

Table 4.3: Result of UH sFFT vs. MIT sFFT vs. FFTW ($k = 1000$)

(a) Fix $k = 1000$, vary n from 2^{19} to 2^{28}

Signal size 2^n	**UH sFFT (sec)**	**MIT sFFT (sec)**	**FFTW (sec)**	**Speedup UH vs. MIT sFFT**	**Speedup UH sFFT vs. FFTW**	**Speedup MIT sFFT vs. FFTW**
19	0.12	0.22	0.02	1.84	0.13	0.07
20	0.16	0.29	0.05	1.82	0.34	0.18
21	0.18	0.39	0.17	2.10	0.90	0.43
22	0.25	0.51	0.39	2.07	1.59	0.77
23	0.33	0.70	0.87	2.13	2.62	1.23
24	0.43	0.97	1.80	2.29	4.23	1.85
25	0.61	1.37	3.84	2.26	6.33	2.80
26	0.80	1.87	7.79	2.35	9.76	4.16
27	1.30	2.65	15.71	2.03	12.07	5.94
28	1.83	3.73	32.05	2.03	17.50	8.60

(b) Fix $n = 2^{27}$, vary signal sparsity k from 1000 to 31000

Signal sparsity k	**UH sFFT (sec)**	**MIT sFFT (sec)**	**FFTW (sec)**	**Speedup UH vs. MIT sFFT**	**Speedup UH sFFT vs. FFTW**	**Speedup MIT sFFT vs. FFTW**
1000	1.33	2.72	15.67	2.05	11.81	5.76
3000	2.01	4.61	15.68	2.30	7.81	3.40
5000	2.88	6.27	15.72	2.17	5.45	2.51
7000	3.81	8.13	15.82	2.14	4.16	1.95
9000	4.98	10.54	15.78	2.12	3.17	1.50
11000	6.44	13.75	15.72	2.14	2.44	1.14
13000	8.41	17.85	15.73	2.12	1.87	0.88
15000	7.51	15.55	15.67	2.07	2.09	**1.01**
17000	8.71	17.72	15.70	2.03	1.80	0.89
19000	9.96	20.47	15.75	2.05	1.58	0.77
21000	11.00	22.87	15.66	2.08	1.42	0.68
23000	12.32	25.60	15.73	2.08	1.28	0.61
25000	13.74	28.67	15.64	2.09	1.14	0.55
27000	15.30	31.55	15.87	2.06	**1.04**	0.50
29000	16.50	34.77	15.72	2.11	0.95	0.45
31000	18.30	38.34	15.78	2.10	0.86	0.41

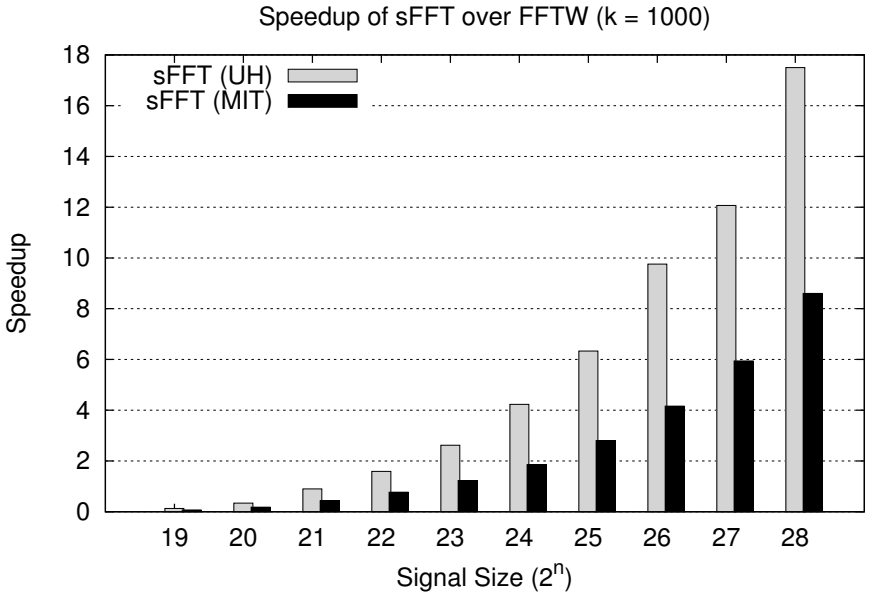
MIT sFFT is faster than FFTW from 0.07x to 8.6x. As a result, we can conclude that UH sFFT implementation makes 2x performance improvement over the original MIT's sFFT implementation.

Execution Time vs. Signal Sparsity k

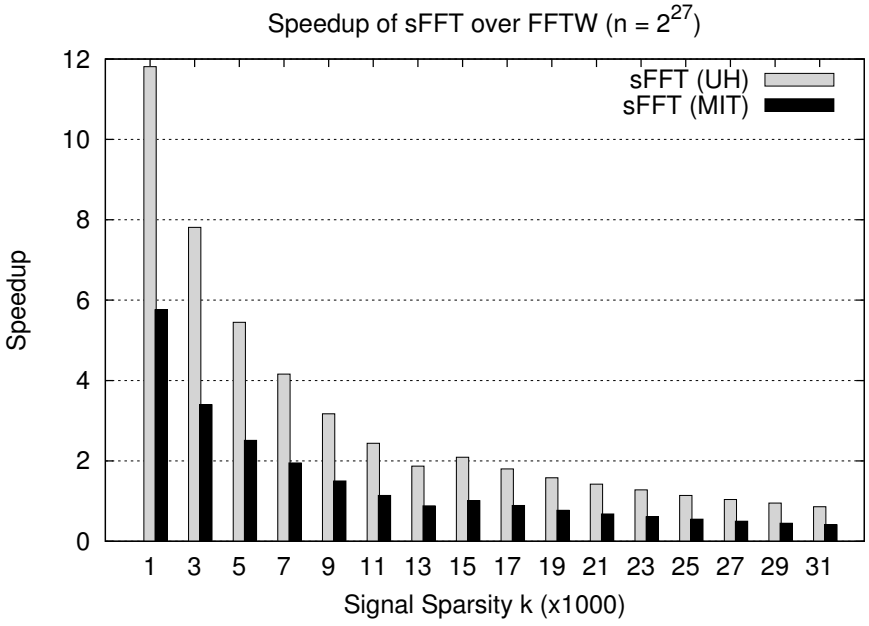
In this experiment, we fix the signal size to $n = 2^{27}$ (i.e., 134,217,728) and evaluate the execution time of sFFT vs. the number of non-zero frequencies k. We range the k from 1000, 3000, to as dense as $31,000$. Figure 4.1(b) illustrates the average execution time of the compared algorithms.

From the Figure 4.1(b), we can see that MIT sFFT is faster than FFTW for k up to 15,000. The UH sFFT, on the other hand, is faster than FFTW for k is around 30,000, doubling the band of the signal spectrum for sFFT being faster than FFTW. It means more applications with "denser" signal spectrum which was slower than FFTW by using the MIT's sFFT implementation now can beat over the FFTW by using our optimized sFFT implementation. Besides, the result is consistent with the theoretical analysis that the crossing value for sFFT to be faster than full-size FFT is around $\sqrt{n}$. Finally, FFTW has a runtime of $O(nlogn)$, irrelevant of the signal sparsity k, as can be seen in Figure 4.1(b) as well. Thus, as the sparsity of the signal decreases (i.e., k increases), FFTW eventually becomes faster than sFFT. Nevertheless, the results still show that UH sFFT implementation extends of applications for which sparse approximation of FFT is practical.

The speedup result is shown in Table 4.3(b). As is shown in the table, the



(a) Speedup of sFFT over FFTW ($k = 1000$)



(b) Speedup of sFFT over FFTW ($n = 2^{27}$)

Figure 4.2: Performance evaluation sFFT vs. FFTW

speedup of UH sFFT implementation is constantly over 2x than the MIT's sFFT implementation for various signal sparsity k.

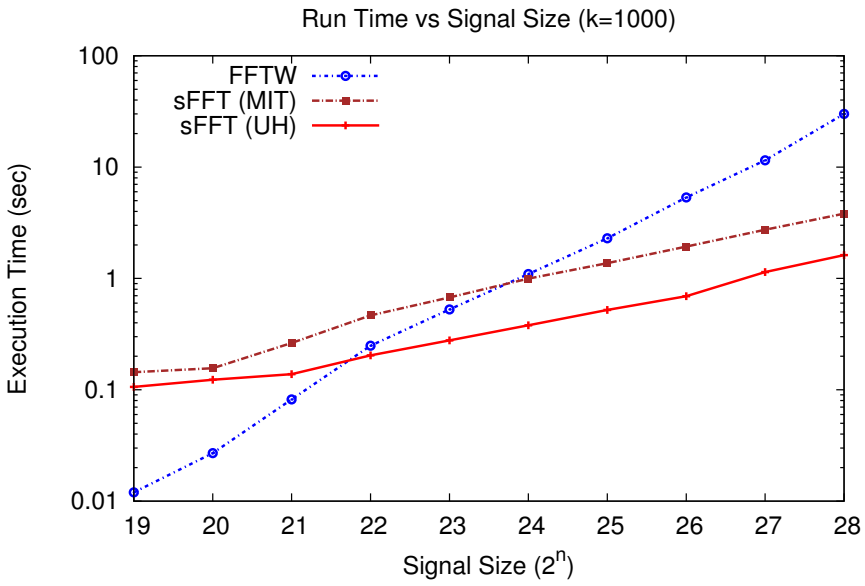
4.3.3 Experimental Results – Single Precision

In this subsection, we evaluate the performance of sFFT for input data sets are single-precision floating point numbers.

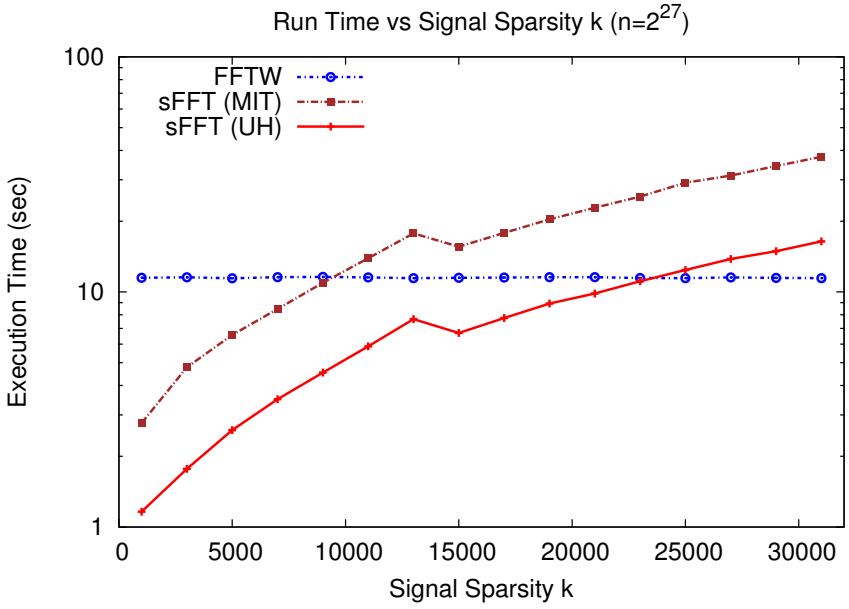
Execution Time vs. Signal Size N

Similar to the experiments for double-precision numbers, in this experiment, we fix the signal sparsity $k = 1000$ and report the execution time of the three compared algorithms for signal sizes n ranging from 2^{19} to 2^{28}.

Figure 4.3(a) plots the average execution time of UH sFFT, MIT sFFT and FFTW on input data of single-precision float numbers. We can observe from the figure that the UH sFFT is still constantly faster than MIT sFFT implementation. Table 4.4(a) shows speedup of UH sFFT over the MIT's implementation. It shows the average speedup of UH sFFT is over 2x than the MIT's sFFT implementation. The result is also consistent with the double-precision numbers, which shows the 2x performance improvement of UH sFFT as well. Note that from Figure 4.3(a), the UH sFFT reduces the execution time so that the signal size n is decreased from 2^{24} to 2^{22} in order to be faster than the FFTW. The results on single-precision are consistent with the double-precision numbers.



(a) Execution time vs. signal size ($k = 1000$)



(b) Execution time vs. signal sparsity k ($n = 2^{27}$)

Figure 4.3: Performance evaluation of UH's sFFT implementation vs. MIT's implementation and FFTW (single precision)

Table 4.4: Result of UH sFFT vs. MIT sFFT vs. FFTW ($k = 1000$), single precision

(a) Fix $k = 1000$, vary n from 2^{19} to 2^{28}

Signal size 2^n	**UH sFFT (sec)**	**MIT sFFT (sec)**	**FFTW (sec)**	**Speedup UH vs. MIT sFFT**	**Speedup UH sFFT vs. FFTW**	**Speedup MIT sFFT vs. FFTW**
19	0.11	0.14	0.01	1.36	0.11	0.08
20	0.12	0.16	0.03	1.27	0.22	0.17
21	0.14	0.26	0.08	1.91	0.59	0.31
22	0.20	0.47	0.25	2.28	1.22	0.53
23	0.28	0.68	0.53	2.43	1.90	0.78
24	0.38	1.00	1.09	2.62	2.88	1.10
25	0.52	1.37	2.29	2.63	4.39	1.67
26	0.69	1.94	5.34	2.80	7.72	2.76
27	1.14	2.74	11.50	2.40	10.07	4.20
28	1.62	3.82	30.05	2.36	18.54	7.87

(b) Fix $n = 2^{27}$, vary signal sparsity k from 1000 to 31000

Signal sparsity k	**UH sFFT (sec)**	**MIT sFFT (sec)**	**FFTW (sec)**	**Speedup UH vs. MIT sFFT**	**Speedup UH sFFT vs. FFTW**	**Speedup MIT sFFT vs. FFTW**
1000	1.16	2.77	11.48	2.38	9.88	4.15
3000	1.77	4.78	11.54	2.71	6.53	2.41
5000	2.59	6.56	11.44	2.54	4.43	1.74
7000	3.50	8.46	11.56	2.42	3.30	1.37
9000	4.54	10.95	11.59	2.41	2.55	1.06
11000	5.87	13.91	11.53	2.37	1.97	0.83
13000	7.66	17.75	11.45	2.32	1.49	0.65
15000	6.69	15.58	11.49	2.33	1.72	0.74
17000	7.75	17.83	11.52	2.30	1.49	0.65
19000	8.93	20.37	11.56	2.28	1.30	0.57
21000	9.84	22.83	11.56	2.32	1.17	0.51
23000	11.10	25.46	11.46	2.29	1.03	0.45
25000	12.40	29.15	11.45	2.35	0.92	0.39
27000	13.81	31.26	11.53	2.26	0.83	0.37
29000	14.91	34.32	11.48	2.30	0.77	0.33
31000	16.41	37.58	11.45	2.29	0.70	0.30

Execution Time vs. Signal Sparsity k

Comparable to experiments for double-precision inputs, we fix the signal size to $n = 2^{27}$ (i.e., 134,217,728) and evaluate the execution time of sFFT vs. the number of non-zero frequencies k in this experiment. We range the k from 1000, 3000, to as dense as $31,000$. Figure 4.3(b) illustrates the average execution time of the compared algorithms.

From the Figure 4.3(b), we can see that MIT sFFT is faster than FFTW for k up to 9,000. The UH sFFT, on the other hand, is faster than FFTW for k is up to 23,000. It stretches the signal spectrum by 2.5x. The speedup result is shown in Table 4.4(b). As is shown in the table, the speedup of UH sFFT implementation is over 2x than the MIT's sFFT implementation for various signal sparsity k. Note that this result on single-precision inputs is still consistent with the double-precision results.

4.3.4 Numerical Accuracy

In this subsection, we check the sFFT's promising performance is not at the expense of reducing the numerical accuracy[2]. To achieve this, we compute the $L1$ error rate which is, essentially to compute the Root-Mean-Square-Error (RMSE) of the output results. Specifically, we compute the difference between each output data point of

[2]The UH sFFT implementation guarantees the same order of numerical accuracy as the MIT's original sFFT implementation. Here we evaluate the numerical accuracy of UH sFFT implementation against full-size standard FFT implementation (i.e., FFTW)

sFFT, denoted $\hat{x}_i$, with that of FFTW, namely $\hat{y}_i$, and calculate the RMSE as follows:

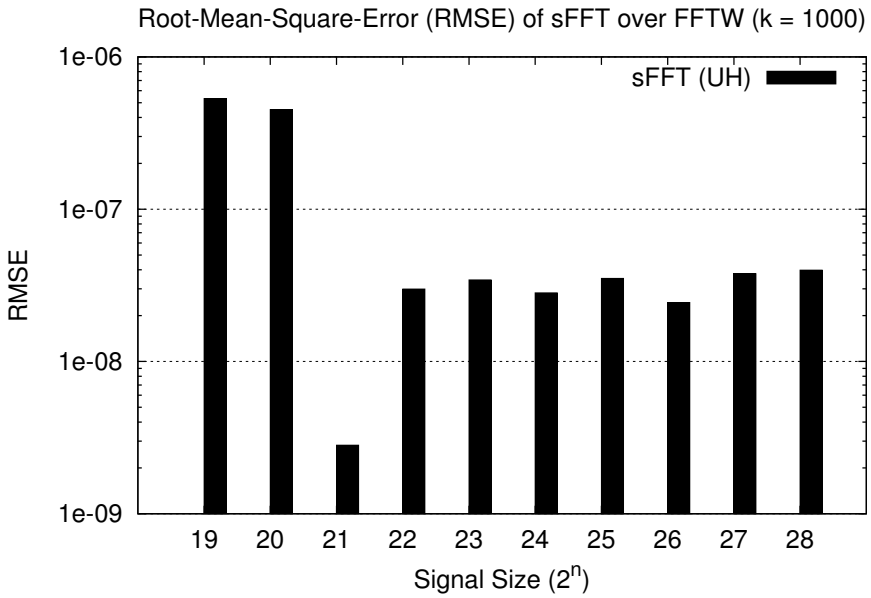
$$RMSE = \frac{1}{k} \sum_{0<i<n} |\hat{x}_i - \hat{y}_i| \tag{4.1}$$

Figure 4.4 shows the RMSE comparison with various signal size n and signal sparsity k. Figure 4.4(a) plots the RMSE with signal size n from 2^{19} to 2^{28} while fix the sparsity $k = 1000$. It can be seen from the figure that, overall, the error rate is negligibly small. The average RMSE is under $1.22e-07$ per large frequency, with the median value $3.47e-08$. Figure 4.4(b) shows the RMSE for signal size $n = 2^{27}$ while varying the signal sparsity k from 1000 to 31,000. It can be seen from the figure that the average RMSE value is $7.09e-07$, and the median value is $4.80E-07$. The error rate is acceptably small, so we can conclude that the reduced execution time of sFFT is not at the cost of its numerical accuracy. It is also interesting to see from the Figure 4.4(b) that the RMSE increases slightly with the signal sparsity k. That is because it is harder to recover the large frequencies when the signal becomes "denser". Nevertheless, the overall RMSE is still in the order of $1e-7$, which is slight.

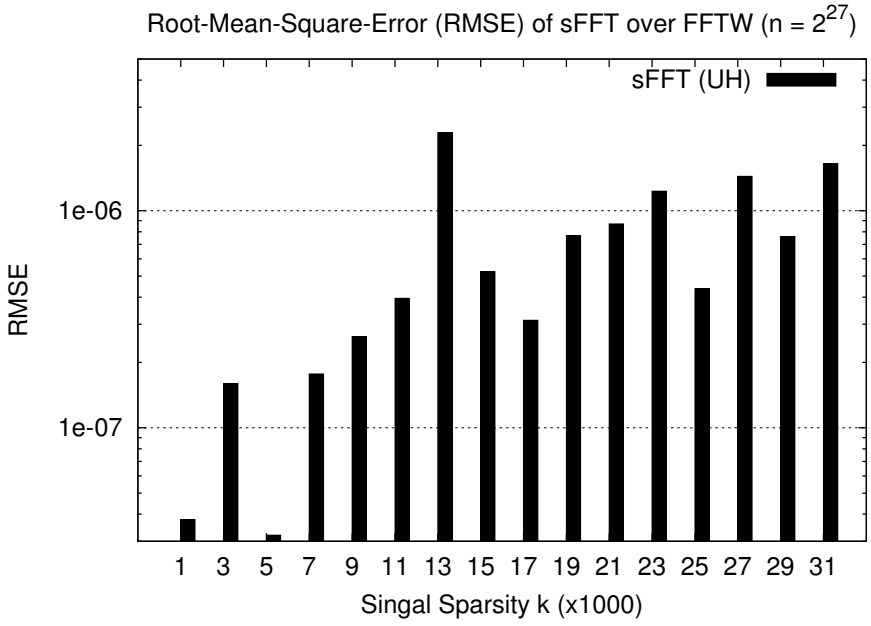
4.4 Time Distribution of Major Stages in Sparse FFT

In this section, we profile the sequential sFFT implementation and study the time distribution of the major stages of the sFFT as we discussed in Chapter 3.

Figure 4.5(a) plots the time distribution by varying signal size n from 2^{18} (i.e.,

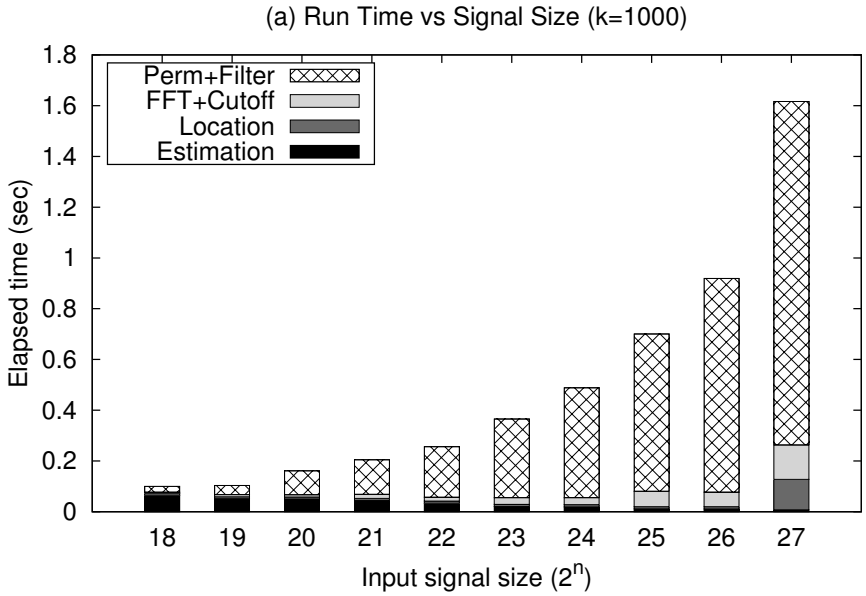


(a) RMSE of sFFT over FFTW ($k = 1000$)

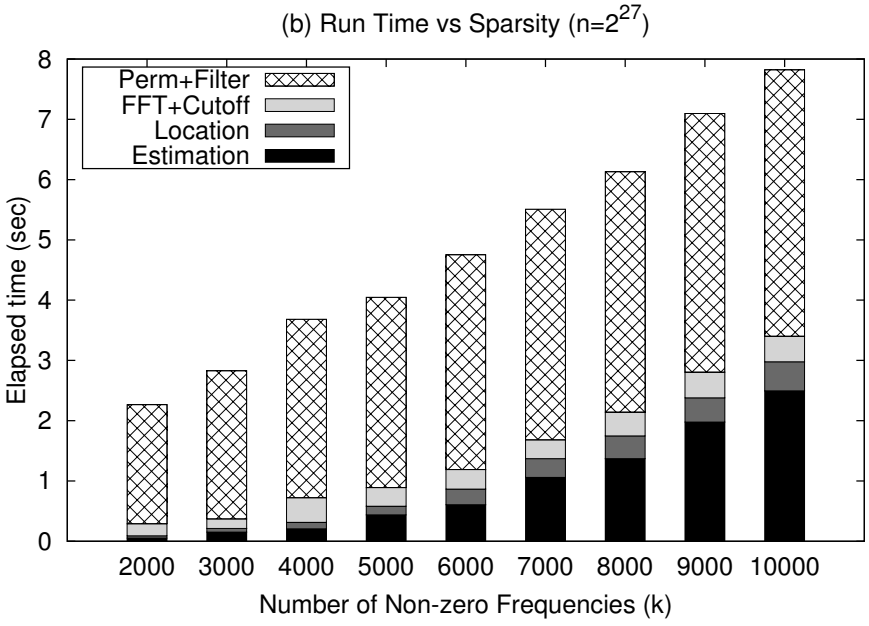


(b) RMSE of sFFT over FFTW ($n = 2^{27}$)

Figure 4.4: Numerical accuracy of sFFT over FFTW



(a) Execution time vs. signal size ($k = 1000$)



(b) Execution time vs. signal sparsity ($n = 2^{27}$)

Figure 4.5: Profiling results for main steps of sFFT

262,144 points) to 2^{27} (i.e., 134,217,728 points) with signal sparsity k fixed to 1000. As can be seen from the figure, the time taken on the stages of permutation and filtering (Stage 1 and 2, denoted as *perm+filter* in figure[3] increases quickly with the signal size n, soon dominate the entire execution time. It is interesting to note that the time taken by the *estimation loop* (Stage 6) decreases when the signal size n increases. This is because the size of the *estimation loop* is determined by the *relative* signal sparsity, i.e., the percentage of non-zero frequencies k out of input signal size n. In Figure 4.5(a), since we fix the signal sparsity k and increase the signal size n, the relative sparsity decreases instead. That is why the *estimation* time goes down with the increase of the signal size n.

Figure 4.5(b) plots the time distribution of sFFT when the signal sparsity k increases with fixed signal size n. In the experiment, we vary the k from 2000 to 10,000 while fix the $n = 2^{27}$. As expected, the *perm+filter* constantly dominates the overall execution time. Also, the time of the *estimation* stage increases with signal sparsity k for the reason of increasing the *relative* sparsity as well.

In summary, since the *perm+filter* stage is the most time-consuming part of the sFFT algorithm, we will mainly outline optimizing this function when we present our parallel algorithms. Nevertheless, we still parallelize the entire sFFT algorithm instead of just the *perm+filter* stage. The major purpose is to avoid the data transfer overhead due to bulk volume PCIe data transfers between host CPUs and accelerators.

[3]We will use the term *perm+filter* refer to Stage 1 and 2 for the rest of the dissertation.)

4.5 Study of Irregular Memory Access Pattern in Sparse FFT

In this section, we study the *perm+filter* stage, the most time-consuming function in sFFT as we discussed in the last section. We will perform a theoretical analysis of the cache miss rate, as well as conduct the experiments to measure the actual cache miss rate. We show that irregular memory access pattern is the principal performance bottleneck for sFFT we need to primary address in order to achieve good performance. Irregular memory access pattern exhibits poor spatial and temporal locality, which consequently, results in poor performance and parallel scalability.

4.5.1 Cache Miss Rate Analysis

In this subsection, we analyze the theoretical bound of the cache misses caused by the irregular memory accesses.

Figure 4.6 presents a simplified code snippet (irrelevant code logistics has been removed) of the *perm+filter* stage of the sFFT. We can see from the code that three arrays are used in the inner loop: `buckets[], signal[]` and `filter[]`. They denote (in the context of sFFT) size of B buckets, the input signal and the filter, respectively.

From the locality point of view, the array `buckets[]` has spatial locality, as the data reference pattern is unit-strided. Furthermore, it may have temporal locality if its size can fit into the cache That is because accessing to the `buckets` follows a

```
1 /* Perm + Filter stage in sFFT */
2 int idx = init_val
3 do t = 1, time
4   do i = 1, num
5     idx = (idx + ai) % num
6     buckets[i % B] += signal[idx] * filter[i]
7   end do
8 end do
```

Figure 4.6: The simplified *perm+filter* stage in sFFT

round-robin pattern (via modular operation). The array `signal[]`, virtually has no either spatial and temporal locality since its access pattern is irregular. It is worth noting that the `signal[]` array has cyclic reuse pattern, but the reuse distance is too far away that a cache block may have already been flushed out of the cache even though it is reused in near future. As a result, although the same cache block might be reused later, it still needs to load the block from the memory. So it indicates that the accesses to the `signal[]` array may not have spatial locality. The `filter[]` array, on the other hand, has perfect spatial locality. Consequently, we hypothesize that the irregular memory access pattern in sFFT mainly comes from accessing the array `signal[]`, the input signal.

Based on this insight, we analyze the cache misses by estimating the cache miss rate for the *perm+filter* stage. For simplicity, we assume that each access to the `filter` and `bucket` array always hits in the cache since the memory access pattern is consecutive. In real systems, regular access patterns may still cause few cache misses due to the capacity miss. We will measure this effect later in this section. In addition, we make an assumption that each access to the array `signal` will end up with a cache miss. Our theoretical cache misses analysis is based on these two

```
1  /* Complex arithmetic in sFFT */
2  do t = 1, time
3    do i = 1, num
4      idx = (i * ai) % num
5      bucket[i % B].re += filter[i].re * signal[idx].re -
6                          filter[i].im * signal[idx].im
7
8      bucket[i % B].im += filter[i].re * signal[idx].im +
9                          filter[i].im * signal[idx].re
10   end do
11 end do
```

Figure 4.7: Complex arithmetic decomposition of sFFT

assumptions. We will justify the hypothesis by experiments presented in next section.

Further, note that arrays in the code shown in Figure 4.6 use `complex` data types which can be decomposed into a pair of *real* and *imaginary* parts. Consequently, the complex number arithmetics in the code of *perm+filter* can be expressed in Figure 4.7. It can be seen from the figure that each iteration of the inner loop has a total number of 10 load operations. Since we assume that only irregular loads can cause cache misses, accessing to the `signal[idx].re` is the only cache miss out of the 10 memory accesses. Accessing to `signal[idx].im`, on the other hand, can always hit in the cache. That is because the data layout for a complex number is in the format of `real[0], imaginary[0], real[1], imaginary[1], ...`, in our sFFT implementation. Given the data is well aligned, once we load a cache block containing the `real` part of a complex number, it is highly likely that the `imaginary` part will reside in the same cache block. Therefore, we can calculate that the theoretical cache miss rate for the *perm+filter* is 10% because of 1 out of 10 memory loads in inner each loop iteration will end with a miss.

4.5.2 Experimental Evaluation

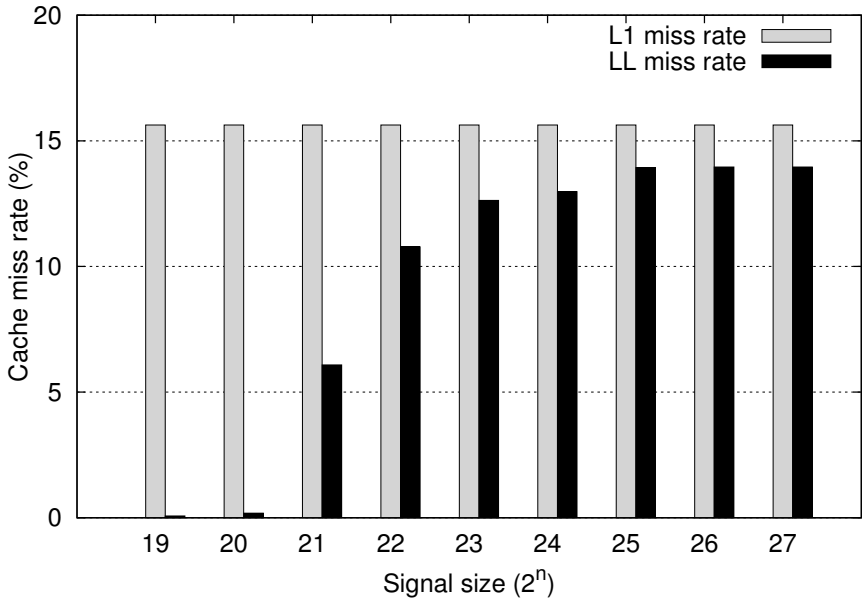
In this subsection, we measure the L1 and LL (Last Level) cache miss rate by varying the size n of the signal. Since an LL cache miss usually has greater performance penalty than an L1 miss, we highlight more on the LL cache miss rate.

L1 Cache Miss Rate

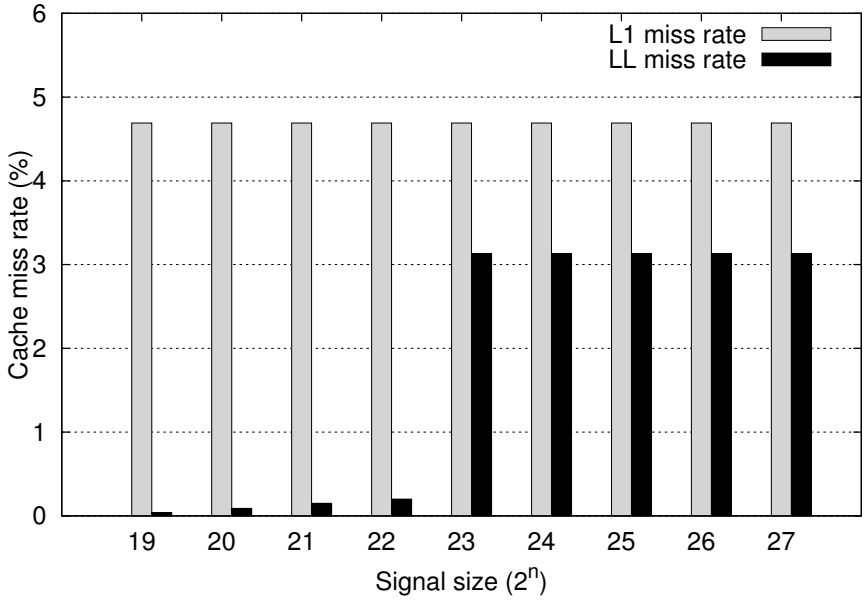
In this section, we evaluate the L1 cache miss rate of the *perm+filter* stage in sFFT for various data sizes n from 2^{19} to 2^{27}. The signal sparsity k is fixed to 1000. Figure 4.8(a) shows the results. We can observe from the figure that the L1 cache miss rate is constantly 15.63%. It is because even the smallest input data size (i.e., 2^{19}, 512 KB) cannot fit into the L1 cache (32 KB). The measured L1 cache miss rate is consistent with the theoretical result (10%), indicating that each irregular access to the `signal[idx]` array leads to a cache miss at the L1 data cache.

Interesting, though, the 15.63% L1 cache miss rate is still is higher than the theoretical value, which is 10%. It is because the theoretical analysis assumes zero cache miss for the regular memory accesses for `filter` and `bucket` arrays. In real computations, however, even the regular memory accesses may still cause the cache misses, in particular between loading different cache lines.

To justify the hypothesis, we conduct another experiment which is nearly the same as the previous experiment on measuring the L1/LL cache miss rate, but changing array reference pattern from irregular to regular, i.e., from `signal[idx]` to `signal[i]`. The purpose is to measure the cache miss rate for regular memory access pattern,



(a) L1/LL cache miss rate of the *perm+filter* stage in sFFT



(b) L1/LL cache miss rate for *regular* memory access

Figure 4.8: L1/LL cache miss rate for irregular and regular memory accesses in sFFT

and to estimate the percentage of regular miss rate in the overall cache misses rate shown in Figure 4.8(a).

Figure 4.8(b) shows the L1/LL cache miss rate for regular memory access pattern. From the figure, we can see that L1 cache miss rate is constantly at 4.69%. The number reveals one interesting fact that the regular access pattern takes account for around 5% of cache miss rate out of the total of 15%. Subtracting 5% from the 15% cache miss rate, we could obtain the 10% cache miss rate for irregular access pattern, which is the same as of the theoretical analysis.

LL Cache Miss Rate

Figure 4.8 also shows the last-level (LL) cache miss rate. From the Figure 4.8(a), we can see that the LL cache miss rate grows with the signal size n. It begins with 0.07% with data size of 512 KB ($n = 2^{19}$) and constantly ends with 13.96% with data size of 128 MB ($n = 2^{27}$). Since the LL cache size is 15 MB, the cache miss rates are as expected that firstly fits into the cache then gradually exceeds the capacity of the cache.

Furthermore, from Figure 4.8(b) we can also observe that the LL cache miss rate for regular memory access pattern begin0 with nearly zero and end up with constantly 3.13%. Minus 3.13% from the 13.96%, we can get that the LL cache miss rate for irregular access pattern is around 10%, which is consistent with the theoretical analysis. The result also indicates that the LL cache miss rate for irregularly accessing to `signal[idx]` is nearly 100% for large signal size.

4.6 Summary

In this chapter, we present our optimized implementation of the sFFT algorithm. We discussed multiple design choices in order to deliver the best possible performance. Compared to the original MIT sFFT implementation, our implementation achieves more than 2x performance improvement. The performance improvement allows more applications to fit into the spectrum of sFFT. In addition, we also evaluate the performance with FFTW, one of the most widely used full-size FFT libraries. The experimental results show that the sFFT is as much as 17x faster than the FFTW, and the performance gap still increases quickly with even larger input signals. Further, we also show that the promising performance of sFFT is not at the cost of reduced numerical accuracy.

After that, we study the time distribution of the major functional stages in sFFT. We show that the *perm+filter* stage is the most time-consuming stage in sFFT. Finally, we analyze the *perm+filter* stage by using both theoretical and experimental approaches. The result indicates that the irregular memory access pattern, which caused nearly 100% of L1 and LL cache misses, is the root cause of the poor performance for the *perm+filter* stage. These two observations provide us a strong motivation that exploiting data locality of the sFFT is critical to improving the performance as well as the parallel scalability.

In next few chapters, we will present the parallel sFFT implementations for a variety of state-of-the-art parallel architectures. We will discuss our major approaches in improving the data locality. We will also show that the enhancement of the

data locality will significantly improve the performance and the parallel scalability of sFFT.

Chapter 5

PsFFT: Parallel Sparse FFT on Multicore CPUs

5.1 Overview

In this chapter, we present our *Parallel* sFFT implementation, namely *PsFFT*, on multicore CPUs[1]. Since sFFT is a fairly new numerical algorithm, and FFT is of key importance to a considerable amount of scientific and engineering applications, it is a nature path to enhance the performance of sFFT through parallel computing techniques on state-of-the-art parallel architectures[2].

Since *PsFFT* is implemented by using OpenMP, a *de facto* programming model for shared-memory parallel programming, we first briefly introduce the OpenMP in

[1]For the rest of the dissertation, we will use the term *PsFFT* to refer our parallel sFFT implementation on multicore CPUs.

[2]The majority of contents in this chapter are based on our prior publication in [80].

Section 5.2.

Second, we will discuss the challenges in parallelizing sFFT in Section 5.3. Section 5.4 presents the PsFFT algorithm in a step-by-step approach. Some techniques to address the challenges in the parallelization and to improve the data locality will be discussed in Section 5.5.

Section 5.6 will evaluate the performance of the PsFFT algorithm with: 1) the sequential implementation of sFFT as was discussed in Chapter 4, and 2) the parallel FFTW. We will show that the PsFFT achieves a considerable amount of speedup compared to the sFFT and FFTW. We will also demonstrate that the data locality is improved by employing the locality optimization techniques. Finally, we will conclude and discuss some open questions we need to address in future.

5.2 Parallel Programming with OpenMP

5.2.1 Overview

OpenMP is a *de facto* programming model for shared-memory parallel programming [8]. It provides a simple interface including a set of compiler directives, library routines and environment variables, which can be used to create a parallel region, define work sharing, manage the data environment, and specify synchronizations. Currently, OpenMP mainly supports programming languages in C, C++, and Fortran.

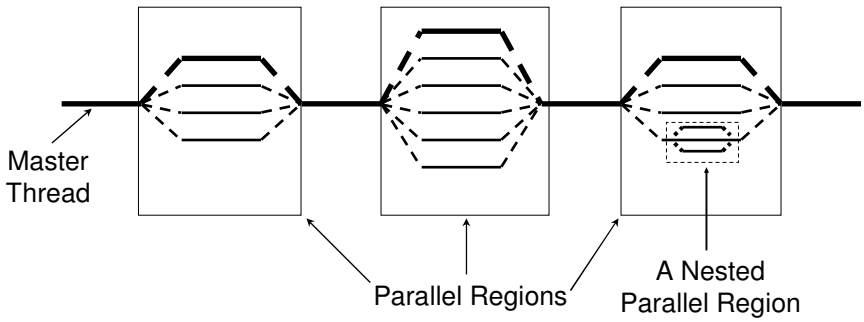


Figure 5.1: Fork-join model of OpenMP

OpenMP adopts a simple fork-join model (presented in Figure 5.1). An OpenMP program begins with an execution using a single thread, called the *master* thread. The master thread spawns teams of threads in response to OpenMP directives, which perform work in parallel. OpenMP also allows the threads to spawn another level of teams of threads inside a parallel region, which is call nested parallelism. Parallelism is thus added incrementally; the serial program evolves into a parallel one. OpenMP directives are inserted at key locations in the source code. These directives take the form of comments in Fortran and `#pragmas` in C and C++. Most OpenMP directives apply to structured blocks, which are blocks of code with one entry point at the top and one exit point at the bottom. The compiler interprets the directives and creates the necessary code to parallelize the indicated code blocks.

5.2.2 Major Contents

Major OpenMP directives enable the program to create a team of threads to execute a specified region of code in parallel (`omp parallel`), the sharing out of work

in a loop or in a set of code segments (work-sharing constructs such as loop construct `omp do` in Fortran (or `omp for` in C/C++), sections construct `omp sections`), data environment management (`private` and `shared`), and thread synchronization (`barrier`, `critical` and `atomic`). User-level runtime routines allow users to detect the parallel context(`omp_in_parallel()`), check and adjust the number of executing threads(`omp_get_num_threads()` and `omp_set_num_threads()`) and use locks (`omp_set_lock()`). Environment variables may also be used to adjust runtime behavior of OpenMP applications particularly by setting defaults for the current execution. For example, it is possible to set the default number of threads to execute a parallel region(`OMP_NUM_THREADS`) and the default scheduling policy (`OMP_SCHEDULE`) etc.

For the loop construct (`omp do` in Fortran or `omp for` in C/C++), a `schedule` clause with several variants is provided to specify how iterations of the loop are assigned (scheduled) among threads in order to achieve load balancing. `schedule(static)` evenly divides the iteration space into several chunks and each thread is assigned at most one chunk. `schedule(static,chunk_size)` indicates that the iterations are divided into chunks of the specified size and the chunks are assigned to threads in a round-robin fashion. The way of assigning chunks to threads is fixed for `static` scheduling once the iteration number and thread number are known. For `dynamic` scheduling, the assignment dynamically happens at runtime. Each thread grabs one chunk of the loop iterations at a time to execute and requests another one when it finishes the current chunk until no more chunks remain. `guided` scheduling is similar to the `dynamic` scheduling except that the chunk size is proportional to the

number of the remaining iterations divided by the number of threads until reaching to the specified chunk size. Finally, the choice of the scheduling policies mentioned above can be deferred until runtime using an environment variable `OMP_SCHEDULE` if `schedule(runtime)` is used.

5.2.3 An Example using OpenMP

Figure 5.2 shows a code example in which a loop calculating the value of PI is parallelized using OpenMP. In this case, a work-sharing construct `omp for` is combined with the parallel construct `omp parallel` to indicate that the loop body is going to be executed in parallel and the work of the entire iteration space is shared by multiple threads. A data environment clause `private` is needed to keep a private copy of the temporary variable `x` used for each thread, preventing possible race condition otherwise because the default data attribute for all variables is `shared` in OpenMP. The partial sums of the loop iterations assigned to multiple threads are summarized into a final one using the `reduction` clause using `addition` operation. In summary, it is critical for users to judge if a loop is parallelizable before adding OpenMP directives. They also need to make sure the data environment attributes for each variable is implicitly or explicitly specified as desired.

5.2.4 Advantages of OpenMP

As can be seen from the Figure 5.2, one advantage of using OpenMP is it is a high-level parallel programming model which can largely simplify the programming

```
1  int computePI (void) {
2    int i;
3    double x, pi, sum = 0.0;
4    double step;
5
6    long num_steps = 10000000;
7    step = 1.0 / (double) num_steps;
8
9    #pragma omp parallel for reduction (+:sum) private (x)
10   for (i = 1; i <= num_steps; i++) {
11     x = (i - 0.5) * step;
12     sum = sum + 4.0 / (1.0 + x * x);
13   }
14
15   pi = step * sum;
16   printf ("step:%e sum:%f PI=%.20f\n", step, sum, pi);
17   return 0;
18 }
```

Figure 5.2: PI calculation using OpenMP

efforts on parallel architectures. Compared with other multi-threading programming paradigms such as POSIX Thread [59], an OpenMP programmer does not need to specify all the details of managing and coordinating threads. Therefore, the programmer can focus more on the strategies on the application level and lets the compiler handle the low-level details. It greatly shortens the software development cycles.

Second, since OpenMP is a high-level programming model, the OpenMP programmer inserts some parallel directives and does not change the control flow of the corresponding sequential program, which dramatically reduces the efforts of adapting a previously sequential program to parallel form. It largely reserves the structure of its corresponding sequential program thus facilitates the understanding and maintenance of the program. The nice feature of high-level in OpenMP also provides

sufficient code portability across different parallel architectures whenever possible. That is, an OpenMP program usually does not depend on architecture-specific features thus, it does not need to be largely re-factored when ported to another type of architecture.

Last but not least, OpenMP has been well implemented in a large number of mainstream compilers from various vendors and open source communities including GNU, LLVM, Intel etc. [5]. The implementations are maturely enough to be used in real production code with considerable performance guarantee [81].

In summary, the success of OpenMP relies on several advantages: easy of use, incremental parallelism, uniform code for both sequential and parallel versions, portability and performance guarantee. These are the main reasons we decide to use OpenMP as a parallel programming model to implement the sFFT algorithm. In next section, we will start with discussing the challenges in parallelizing the sFFT algorithm. Then we present our parallel sFFT algorithm and discuss the approaches we address the challenge.

5.3 Challenges

Challenge 1: Loop-carried dependence

The first challenge to parallelize the sFFT is a loop-carried dependence in the *perm+filter* stage. Figure 5.3 illustrates a simplified code snippet for the *perm+filter* stage in sFFT. As can be seen from the figure, the loop-carried dependence occurs

```
1 /* Perm + Filter stage in sFFT */
2 int idx = init_val
3 do t = 1, time
4   do i = 1, num
5     idx = (idx + ai) % num
6     buckets[i % B] += signal[idx] * filter[i]
7   end do
8 end do
```

Figure 5.3: The simplified *perm+filter* stage in sFFT

when updating the `idx` variable in the inner loop(line 5). The `idx` determines which signal point to sample for each iteration. In section 5.4, we will present a simple *index mapping* technique which can effectively remove the loop-carried dependence for this stage.

Challenge 2: Race condition

As is shown in the figure, the sFFT starts with sampling the input signals by convoluting with the filter, and binning into one of the buckets. Simply partitioning the inner loop such that each thread gets one chunk of the loop iteration is impossible because it is highly likely that signals on different threads are binned into the same bucket, leading to the *race condition* issue in parallel computing domain. For instance, in Figure 5.3, signal points in iteration `i = 0, B, 2B, ...` will go to the same `bucket[0]` due to the modular operations by the size of `B`. If the iterations are on different threads which will update the same bucket simultaneously, the *race condition* problem will incur.

A trial approach to address this issue is to use a set of *mutex locks* when updating the buckets. Specifically, when a thread updates a bucket, it first tries to acquire

the lock for the bucket. If successes, the thread will lock the bucket to prevent other threads updating the same bucket simultaneously. After the thread finishes the update, it will release the lock then other threads might repeat the same process to acquire the lock. The major limitation of this approach is that locks will serialize the writing process, so there is virtually no parallelization at all. In addition, this approach is costly in space since each bucket needs a separate *mutex lock*. It is very inefficient in space when the data size becomes too large.

Another possible approach is called *local reduction*. That is, each thread maintains a local copy of buckets. Each thread first updates its local buckets. At a final stage, all threads merge its local buckets into a global one. This approach can largely address the parallelization bottleneck issue except for the last stage. The major limitation of this approach is the space overhead due to the local copy of the buckets for each thread. Also, the serialization in the final stage may cause the performance bottleneck as well.

To address the challenge of the *parallel reduction* problem, we present a *bucket tiling* approach in Section 5.4. Our approach can effectively address the *parallel reduction* problem for sFFT without the need of serialization nor the cost of extra space.

Challenge 3: Dynamic irregular memory access pattern

Dynamic irregular memory access pattern involved in the *perm+filter* stage of sFFT when it randomly samples the signals and bins into the buckets. As is shown in

Figure 5.3, the sFFT samples the data entries in the `signal` array, multiplies with the `filter` and accumulates the results into the `buckets`. As can be observed from the figure, accesses to the `signal[index]` are irregular. Furthermore, the value of strides factor (i.e., `ai`) remains unknown until runtime and it will get updated at each iteration of the time-step loop.

As was studied in Section 4.5, the irregular memory access pattern causes not only poor cache utilization but also poses great challenges in efficiently parallelizing the sFFT. The challenges come from two folds. First of all, the irregular memory accesses make the even data partitioning across different threads non-trivial. A sophisticated partition algorithm such as graph partitioning is required to minimize the demanding communication and traffic in memory and balance the workload across threads. The second challenge incurred by high memory traffic in irregular applications is even more fundamental: multicore architectures with shared memory results in cores competing for the memory bandwidth, which substantially limits the application's scalability. In Section 5.5, we will present data locality optimization techniques which can enhance the spatial and temporal locality, thus improve the performance and parallel scalability.

5.4 PsFFT: Parallel Sparse FFT

In this section, we present our parallel sFFT implementation, namely *PsFFT*, on multicore CPUs. According to the profiling results discussed in Section 4.4, the *perm+filter* stage (Stage 1&2) is the most time-consuming function in sFFT, we will

spend most of the space on discussing the approaches to parallelizing and optimizing for this stage. Nevertheless, we still present the parallel approaches for the rest of the stages, which will be relatively straightforward.

5.4.1 Stage 1 & 2: Random Spectrum Permutation and Filtering

The sFFT starts with convolving the permuted input signal with a well-designed filter and binning them into a small number of buckets. Figure 5.3 shows the simplified code snippet of the *perm+filter* stage. Noted that the iterations in the loop are `filter_size` rather than signal size `n` because the tails of the filter are almost zero, so it is unnecessary to compute zero-valued points.

Index Mapping

Among the several challenges to parallelize this stage, the first challenge comes with a loop-carried dependence while updating the variable `idx` that prevents the loop from being parallelized. The `idx` determines and stores the stride distance while permuting the input signal which depends on its previous value.

To address this challenge, we use an *index mapping* approach that can simply remove the loop-carried dependence issue. The idea is based on an observation from Figure 5.3. From the figure, we can see that the values of the `idx` among iterations follow the pattern such that: `init_val, (ai + init_val) % n, (2 * ai + init_val) % n` for loop iteration `i = 0, 1, 2`. Therefore, the idea of the

```
1  int index = init_val;
2  for(int i=0; i < filter_size; i++) {
3    ...
4    // Original index calculation
5    index = (index + ai) % n;
6
7
8    // After index mapping
9    index = (i * ai + init_val) % n;
10   ...
11 }
```

Figure 5.4: Index mapping used in Stage 1&2: random spectrum permutation and filtering

index mapping is to directly map the value of `index` to the loop iterator `i` without relying on its prior value, as is shown in Figure 5.4. As a result, the complexity of obtaining the value of `index[i]` only relies on the loop iterator `i` and can be therefore parallelized.

Bucket Tiling

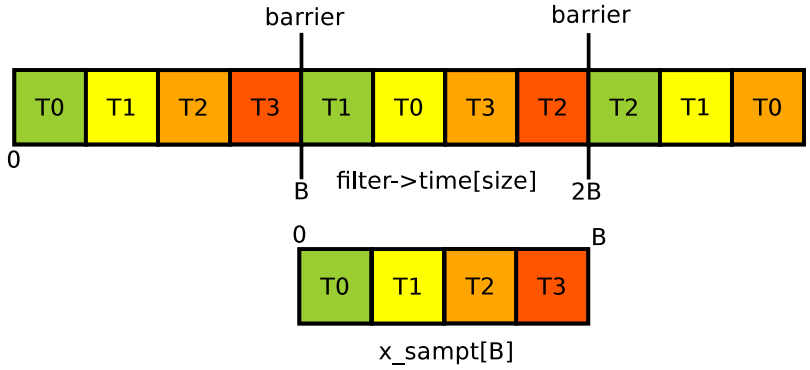
The second challenge is the *race condition* problem, as was discussed in Section 5.3. We address this issue by proposing a *bucket tiling* approach. The idea is similar to a well-known parallel programming paradigm namely *Bulk synchronous parallel* (BSP) [1]. That is, we tile a loop iteration into sets of collision-free iterations. In sFFT, a two layered iteration space is created where the inner layer is collision-free and suitable for a data-parallel mechanism.

Specifically, we partition the buckets into some groups, and each thread will only perform the *perm+filter* stage of the signals belonging to its group. Thus, there will be no collision in a bucket. The signal mapping to be permuted and filtered with the

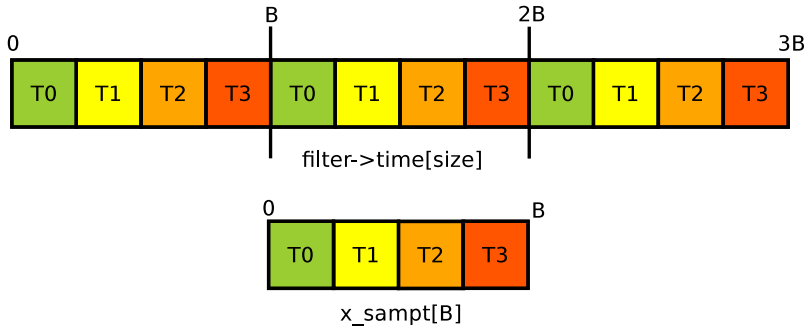
corresponding buckets must be determined beforehand. As is shown in Figure 5.3, the process of hashing signals into buckets follows a round-robin fashion. Thus, the reduction collision only occurs between outer loop iterations. For instance, when loop iteration `i` iterates through `0` to `B - 1`, each permuted signal will be binned into adjacent buckets accordingly. The collision occurs only when `i` equals to `0, B, 2B,` ... when the signal are accumulated in the same bucket.

Figure 5.5(a) shows the basic bucket tiling approach. Data with a color affiliated in the loop will be mapped to buckets with the same color. Updating the buckets (`x_sampt[B]`) entails two steps. In the first step, each thread reads data chunks from the first B elements of the filter and updates the corresponding buckets. In the second step, threads have to wait until all of them have finished updating the buckets before moving onto the next round. In this approach, there will be no hash collision within each round of updating the buckets. It performs better than the traditional *local reduction* approach because each thread only needs to handle one chunk of the data instead of the whole set. There is also no need to synchronize the global array at the end.

A barrier-free bucket tiling. In the basic bucket tiling approach in Figure 5.5(a), a global barrier is required between each round of the outer loop. It is because data chunks of the filter mapped into the same buckets can be hashed to different threads for each round depending on the data layout in memory. For instance, in Figure 5.5(a) the data in green will be affiliated with thread 0 in the first round while with thread 1 in the second round and thread 2 in the third round. A data race hazard between each round exists and global synchronization is needed to



(a) A basic bucket tiling approach



(b) *Barrier-free* bucket tiling

Figure 5.5: bucket tiling

ensure that all threads finish updating the buckets before moving to the next round.

However, the global synchronization impedes the complete loop to be fully parallelized. It also causes significant overhead when the number of threads increases. To address this problem, we introduce an enhanced *barrier-free* bucket tiling approach, as shown in Figure 5.5(b). Unlike the basic approach where the data is only affiliated within an inner loop iteration; the enhanced design will ensure the data be affiliated between each round as well. That is, the data chunks hashing into the same bucket will always affiliate to the same thread. For instance, in Figure 5.5(b), the

```
1  void inner_loop_perm_filter(real_t *d_origx, int n,
2                              real_t *d_filter,
3                              int filter_size
4                              real_t *d_x_sampt,
5                              int B,
6                              int ai) {
7    int i, j;
8    int n_2 = n * 2;
9
10   // Number of rounds to accumulate
11   // the permuted samples into buckets
12   int round = filter_size / B;
13
14   // Buckets are initialized to 0
15   #pragma omp parallel for
16   for (i = 0; i < B; i ++) {
17     int i_2 = i * 2;
18     d_x_sampt[i_2] = 0.;
19     d_x_sampt[i_2 + 1] = 0.;
20   }
21
22   // Main body of the loop
23   #pragma omp parallel
24   for (i = 0; i < round; i++) {
25     int off1 = i * B * 2;
26     #pragma omp for nowait
27     for (j = 0; j < B; j++) {
28       unsigned off2 = off1 + 2 * j;
29       unsigned index = off2 * (unsigned) ai % n_2;
30
31       // Calculate the real and imaginary part of
32       // complex arithmetics respectively
33       d_x_sampt[2 * j] += REAL(index, off2, 0);
34       d_x_sampt[2 * j + 1] += IMAG(index, off2, 0);
35     }
36   }
37 }
```

Figure 5.6: Stage 1&2: Parallel Permutation and Filtering using OpenMP

data chunks in green in the `filter` will be hashed into the same buckets of `x_sampt` maintaining the same color. These data chunks are only affiliated with thread 0 in each round. As a result, the thread that finishes one round will directly move to the next round, asynchronously without having to wait for other threads to be finished. The global barrier between each round is therefore eliminated.

Figure 5.6 shows the code snippet of the parallel *perm+filter* stage after applying the *index mapping* and *bucket tiling* techniques. Compared to the sequential *perm+filter* code shown in Figure 5.3, the original loop has been partitioned into two nested loops which can be executed in parallel.

The main advantage of using the *bucket tiling* approach is each thread only updates its own chunks of buckets (i.e., `d_x_sampt[]` in the code) so there is no need to utilize *mutex locks* to prevent the concurrent update nor allocate private memory space to perform the local reduction. Therefore, the *bucket tiling* exploits the maximum potential parallelism without extra space cost.

5.4.2 Stage 3: B-dimensional FFT

After binning the spectra into a smaller number of B buckets, a B-dimensional FFT is performed. In PsFFT, we simply employ the parallel FFTW [29] to perform this function. Furthermore, since all *inner loops* will be repeated for iterations *outer loops* times, of each calculates the same dimension of FFTW, we set up the *FFTW plan* [2] only once. The FFTW plan creates some essential parameters, e.g., twiddle factors used for calculating the FFT, which only depends on the input signal size. Since for

```
1  void 1d_fft(int outer_loops,
2              complex_t **x_sampt,
3              int B) {
4    int i;
5
6    // Setup FFTW plan
7    fftw_plan p = fftw_plan_dft_1d(B, // Dimension
8                     x_sampt[0],      // Input
9                     x_sampt[0],      // Output, in-place
10                    FFTW_FORWARD,    // FFT or iFFT
11                    FFTW_ESTIMATE);  // Planner Flag
12
13   #pragma omp parallel for
14   for (i = 0; i < outer_loops; i++) {
15     // FFTW for calculating B-dimensional FFT
16     fftw_execute_dft(p, x_sampt[i], x_sampt[i]);
17   }
18
19   // Destroy the FFTW plan
20   fftw_destroy_plan(p);
21 }
```

Figure 5.7: Stage 3: B-dimensional FFT using OpenMP

sFFT, the input size for Stage 3 is always the number of buckets (i.e., B), we batch the FFTW by creating the FFTW plan only at the first time it calls the FFTW and then reuse it for the rest of iterations. This approach can avoid repeatedly create and destroy the same FFTW plan each time when PsFFT performs the B-dimensional FFT and thus save a lot of time.

Figure 5.7 shows the code snippet of Stage 3, i.e., B-dimensional FFT. As was discussed, the FFTW plan is created only once and reused by all the rest of the iterations in the *outer loop*. In addition, we use the `FFTW_ESTIMATE` planner flags to create the FFTW plan quickly.

5.4.3 Stage 4: Cutoff

After Stage 3, the B-dimensional FFT, there are B buckets containing potentially large Fourier coefficients at frequency domain. Since the spectra are sparse in frequency, still very few of them (i.e., k, where $k \ll B$) are significant. The objective of this stage is to select the top k largest coefficients out of B in the magnitude and store their locations.

Therefore, this problem can be reduced to find out the top k largest objects in a B-dimensional array. There are many classic algorithms to perform this function. The most straight-forward approach is to sort the B-dimensional and select the top k largest elements. However, this method is bounded by the sorting algorithm, which is typical $O(klogk)$ in time. A better algorithm is to utilize a *min* binary heap. Specifically, we build a min heap with the size of k by inserting the first k elements of the array into the heap. Then, for each element from $k+1$ to B, we compare it with the root of the heap. If the element is greater than the root, then replace the element as the new root and heapify the binary heap. Else, ignore the element since it cannot be the top k elements. Finally, the heap has k largest elements as we need. Overall, the time complexity of this approach is bounded by $O(Blogk)$. The space complexity of this algorithm is $O(k)$. In our implementation of PsFFT, we use the *Quick Select* algorithm, since it delivers on average *linear* runtime (i.e., $O(B)$) without extra space cost. The details of the algorithm can be found at [45].

Figure 5.8 shows the parallel cutoff function. In PsFFT, we decide to parallelize the *outer loop* instead of the *quick select* algorithm itself for two reasons. First of all,

```
1  void cutoff(int outer_loops,
2              int **J, int num,
3              complex_t **x_sampt,
4              real_t **samples,
5              int B) {
6    int i,j;
7
8    #pragma omp parallel for
9    for (i = 0; i < outer_loops; i++) {
10
11     // Transform the complex number,
12     // from a + bi to complex absolue value a^2 + b^2
13     for(j = 0; j < B; j++) {
14       samples[i][j] = cabs2(x_sampt[i][j]);
15     }
16
17     // Quick select algorithm
18     find_largest_indices(samples[i], // input
19                          B,          // input size
20                          J[i],       // output
21                          num);       // output size
22   }
23 }
```

Figure 5.8: Stage 4: Parallel cutoff function

according to the profiling results discussed in Section 4.4, the *B-dimensional FFT + Cutoff* (i.e., Stage 3&4) together takes less than 10% of the overall sFFT execution time. Thus, it will not make much difference to parallelize the *inner loop* of this stage, i.e., the quick select algorithm. Second, it is non-trivial to parallelize the quick select algorithm due to the strong dependency nature in the algorithm. Therefore, it is a nature choice to exploit a coarse-grained parallelism (i.e., to parallelize the outer loop) instead of a fine-grained parallelism (i.e., the inner loop).

5.4.4 Stage 5: Recover Hash Function for Location Recovery

The Stages from 1 to 4 define a hash function mapping each coefficient into one of the buckets. The "real" locations of large coefficients need to be reversed by eliminating the phase changes caused by spectrum permutation and filtering.

To parallelize this stage, each thread independently computes the reverse hash function to recover the location of the potential large coefficients. As described in Section 3.2, Stages 1 to 5 iterates for a number of *location loops* times, so for each inner loop (determining the location), we increment the occurrences(score) of the location recovered. Once the score of the frequency equals to the threshold, we consider this location of the coefficient to be large and add it to `hits`.

5.4.5 Stage 6: Magnitude Estimation

The purpose of this step is that given the locations of large coefficients we need to reconstruct the magnitudes. Since the inner loops are repeated for a number of L loops, given a specific location $r \in I'$, we can get a set of magnitudes $\{\hat{x}_i^r | i \in L\}$. So the magnitude $\hat{x^r}$ is computed as the median of the candidate magnitudes for all the L location loops, i.e., median($\{\hat{x}_i^r | i \in L\}$).

To parallelize this stage, given the number of `num_hits`, which is the number of potential large coefficients obtained from the previous stage, each thread computes the reconstruction of the magnitude for a given location in parallel.

5.5 Data Locality Optimization

As studied in Chapter 4.5, sFFT is a memory-bounded algorithm with poor spatial and temporal locality, it is of key importance to exploit the available memory bandwidth. As an early effort, we explore the blocking techniques for sFFT due to two reasons.

First of all, blocking computations via loop nest restructuring has been used successfully to improve memory hierarchy utilization in regular applications; yet it is not thoroughly understood how blocking works on irregular applications. Second, not all memory references are irregular in sFFT. As is shown in Figure 5.3, data accesses to the array `filter` and `bucket` are still regular. Consequently, loop blocking techniques still make effects for this *complex* irregular application. Furthermore, keeping data in cache or registers for the regular computations, therefore, reduce the memory traffic pressure and can save more memory bandwidth for irregular memory accesses.

In our blocking implemented for sFFT, the data is divided into several cache line sized blocks and operations are carried on this block in order to avoid repeatedly fetching data from main memory. We evaluate the effects of blocking techniques later in Section 5.6.

5.6 Performance Evaluation

In this section, we evaluate the performance of the PsFFT on an Intel Sandy Bridge architecture, the same as we used to evaluate the sequential implementation of sFFT in Chapter 3. Table 4.2 shows the experimental setup. The compiler version used to build the sFFT is gcc-4.8.2 with optimization level -O3. In addition, the cache data is collected using Valgrind 3.8.1 [68]. The FFTW version we used is 3.3.4. Similar to Chapter 4, we evaluate both double-precision and single-precision floating point numbers as the input data.

5.6.1 Choosing the Best Block Size

In the first experiment, we explore different block sizes and choose the size of blocks which delivers the best performance for the following experiments. Table 5.1 shows the execution time of the *perm+filter* stage for various block sizes ranging from 32 to 4096. As we can see from the table, for a smaller number of threads, different block size does not make too much difference. For the number of threads being larger (e.g., 6 threads), an over-large block can greatly increase the execution time. For this reason, we choose the block size to be 32 for the rest of the experiments.

Table 5.1: Execution time (sec) of different block sizes for PsFFT ($N = 2^{22}$, $k = 1000$)

no. of threads	**1**	**2**	**3**	**4**	**5**	**6**
block size = 32	0.22	0.12	0.09	0.07	0.08	0.06
block size = 64	0.22	0.12	0.09	0.08	0.08	0.07
block size = 128	0.22	0.12	0.09	0.09	0.06	0.06
block size = 256	0.22	0.12	0.09	0.08	0.07	0.07
block size = 512	0.22	0.12	0.09	0.08	0.08	0.07
block size = 1024	0.22	0.12	0.09	0.08	0.08	0.09
block size = 2048	0.22	0.12	0.12	0.09	0.08	0.08
block size = 4096	0.22	0.12	0.12	0.13	0.16	0.15

5.6.2 Experimental Results – Double Precision

In this subsection, we evaluate the performance of PsFFT for the double-precision floating point numbers. We compare the performance of PsFFT with 1) UH sequential sFFT implementation (i.e., UH sFFT), 2) MIT original sequential sFFT implementation (i.e., MIT sFFT) and 3) parallel FFTW.

Execution Time vs. Signal Size N

In this experiment, we fix the signal sparsity $k = 1000$ and report the execution time of the compared algorithms for signal sizes n ranging from 2^{19} to 2^{28}.

Figure 5.9(a) plots the average execution time of the compared algorithms. From the figure, we can see that the MIT sFFT is the slowest while the UH sFFT implementation reduces the execution time by 2x. The PsFFT, on the other hand, significantly reduces the execution time compared to the sequential UH sFFT and MIT sFFT implementation.

Table 5.2(a) shows the speedup of PsFFT over the sequential version of sFFT. It can be seen from the table that, compared to UH sFFT, the PsFFT achieves 2.9x speedup on average. The PsFFT achieves the average speedup of 4.95x compared to the MIT sFFT implementation.

Compared to parallel FFTW, Figure 5.9(a) shows that the execution time of PsFFT and FFTW are approximately linear in the log scale. However, the slope of the line of PsFFT is less than the slope of FFTW, which is a result of the sublinear runtime of sFFT. Furthermore, the PsFFT becomes faster than FFTW when the signal size is around 2^{22} (more than 4 million data points) at recovering the exact 1000 non-zero large coefficients. Compared to the sequential implementations of sFFT where the cross point is 2^{26} and 2^{24} for MIT and UH sFFT implementation, respectively, the PsFFT greatly reduces the signal size by 16x and 4x in order to be faster than FFTW.

Figure 5.10(a) (also shown in Table 5.3(a)) shows the speedup of PsFFT over parallel FFTW at $k = 1000$. It can be seen from the figure that PsFFT is faster than FFTW from 0.27x to 9.24x with the increase of the signal size n. It is interesting to note that the gap of execution time between PsFFT and FFTW diverges fast with the signal size n due to the sublinear execution time of sFFT. This gives more credits to the PsFFT for large input signals.

Figure 5.11(a) also compares the speedup of PsFFT and FFTW on 6 threads. From the figure, we can see that the average speedup of PsFFT is 3x out of 6 threads. So the parallel efficiency is 50%. Compared to PsFFT, the speedup of parallel FFTW is slightly better for most of the signal sizes. The average speedup for FFTW is 3.14x

on 6 threads.

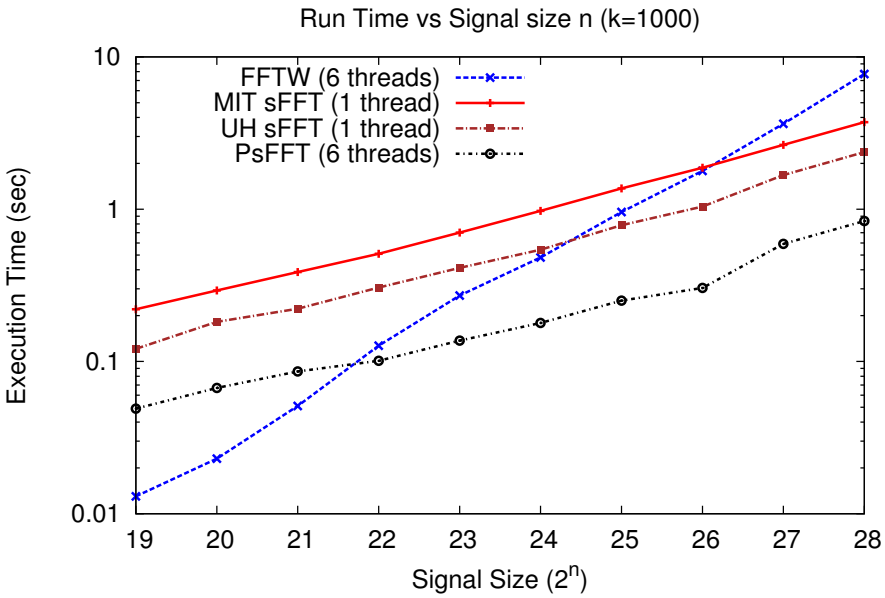
Execution Time vs. Signal Sparsity k

In this experiment, we fix the signal size to $n = 2^{27}$ (i.e., 134,217,728) and evaluate the execution time of PsFFT vs. the number of non-zero frequencies k. We range the k from 1000, 3000, to as dense as $31,000$. Figure 5.9(b) illustrates the average execution time of the compared algorithms. We can see from the figure that the PsFFT significantly reduces the execution time compared with MIT sFFT and UH sFFT implementations.

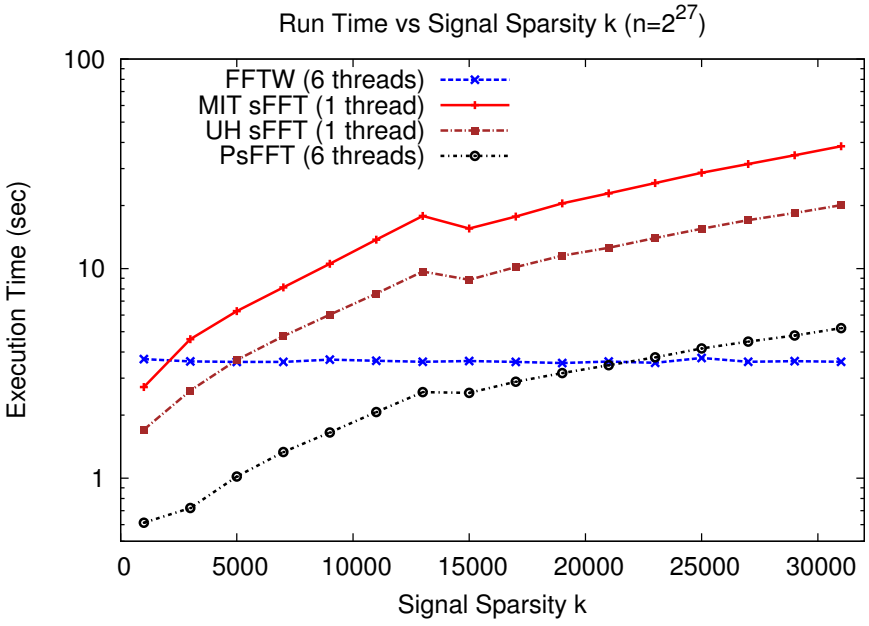
Table 5.2(b) shows the speedup of PsFFT over the sequential version of sFFT when we fix the signal size n and vary the signal sparsity k. It can be seen from the table that, compared to UH sFFT, the PsFFT achieves 3.62x speedup on average. The PsFFT achieves the average speedup of 6.48x compared to the MIT sFFT implementation.

Compared to parallel FFTW, Figure 5.9(b) shows that the PsFFT is faster than FFTW for k up to 23000. Compared to the sequential MIT and UH sFFT implementation where the cross point is at $k = 2000$ and $k = 5000$, the PsFFT greatly expands the signal sparsity k by 5x, indicating more applications with broader signal bands can fit into the PsFFT.

Figure 5.10(b) (also shown in Table 5.3(b)) shows the speedup of PsFFT over parallel FFTW at $k = 1000$. As is shown in the figure, the PsFFT is faster than FFTW by 6.03x at $k = 1000$ and the gap reduces with larger sparsity k. Finally



(a) Execution time vs. signal size ($k = 1000$)



(b) Execution time vs. signal sparsity k ($n = 2^{27}$)

Figure 5.9: Execution time of PsFFT vs. sFFT vs. FFTW

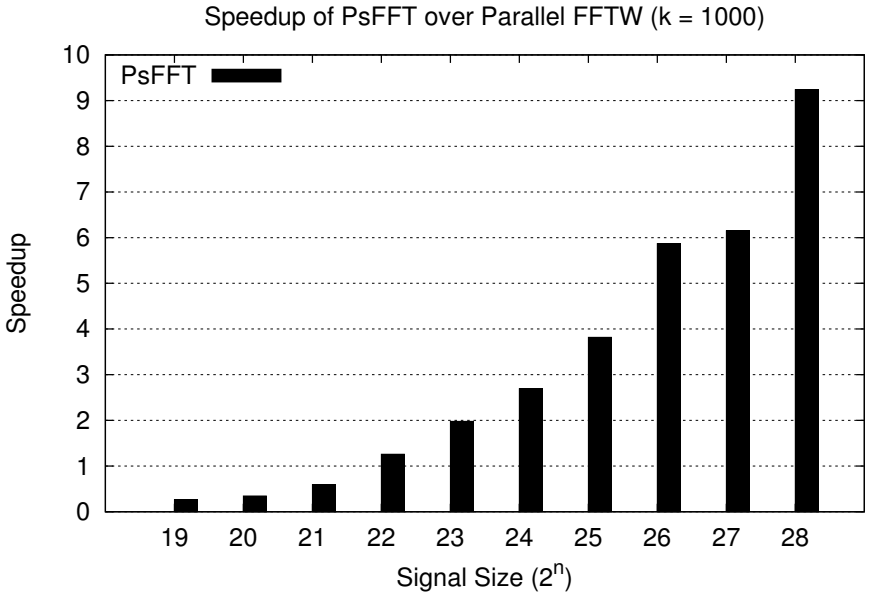
Table 5.2: Speedup of PsFFT (6 threads) over sFFT (1 thread)

(a) Fix $k = 1000$, varying n from 2^{19} to 2^{28}

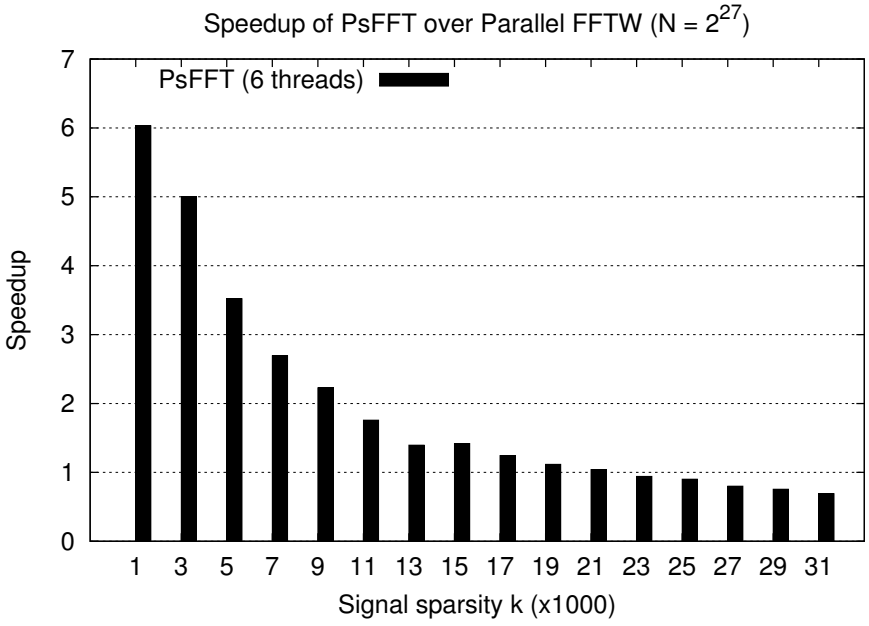
Signal size 2^n	**PsFFT (sec)**	**UH sFFT (sec)**	**MIT sFFT (sec)**	**Speedup PsFFT vs. UH sFFT**	**Speedup PsFFT vs. MIT sFFT**
19	0.05	0.12	0.22	2.47	4.50
20	0.07	0.18	0.29	2.72	4.37
21	0.09	0.22	0.39	2.58	4.49
22	0.10	0.31	0.51	3.03	5.05
23	0.14	0.41	0.70	3.01	5.12
24	0.18	0.54	0.97	3.03	5.44
25	0.25	0.78	1.37	3.12	5.46
26	0.30	1.04	1.87	3.42	6.16
27	0.59	1.67	2.65	2.83	4.47
28	0.84	2.37	3.73	2.84	4.46
			average	**2.90**	**4.95**

(b) Fix $n = 2^{27}$, varying k from 1000 to 31000

Signal sparsity k	**PsFFT (sec)**	**UH sFFT (sec)**	**MIT sFFT (sec)**	**Speedup PsFFT vs. UH sFFT**	**Speedup PsFFT vs. MIT sFFT**
1000	0.61	1.70	2.72	2.78	4.44
3000	0.72	2.62	4.61	3.63	6.39
5000	1.02	3.67	6.27	3.60	6.15
7000	1.33	4.76	8.13	3.57	6.10
9000	1.65	6.03	10.54	3.65	6.39
11000	2.07	7.60	13.75	3.68	6.65
13000	2.58	9.68	17.85	3.76	6.93
15000	2.55	8.85	15.55	3.47	6.09
17000	2.89	10.17	17.72	3.53	6.14
19000	3.18	11.53	20.47	3.63	6.45
21000	3.46	12.57	22.87	3.63	6.61
23000	3.78	14.00	25.60	3.71	6.78
25000	4.16	15.49	28.67	3.72	6.89
27000	4.49	17.04	31.55	3.80	7.03
29000	4.79	18.41	34.77	3.84	7.26
31000	5.20	20.09	38.34	3.86	7.37
			average	**3.62**	**6.48**



(a) Speedup of PsFFT over FFTW($k = 1000$)



(b) Speedup of PsFFT over FFTW ($n = 2^{27}$)

Figure 5.10: Speedup of PsFFT over parallel FFTW

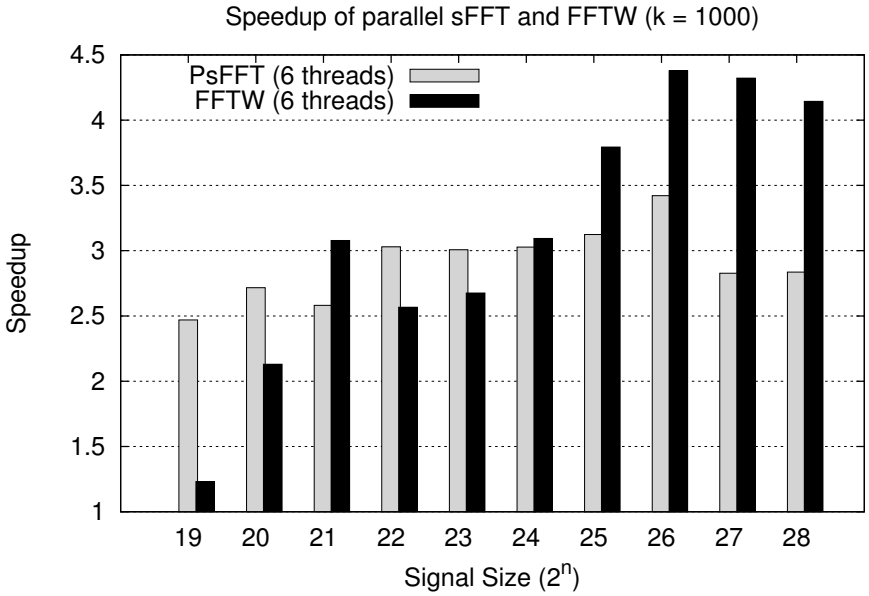
Table 5.3: Speedup of PsFFT over parallel FFTW (6 threads)

(a) Fix $k = 1000$, vary n from 2^{19} to 2^{28}

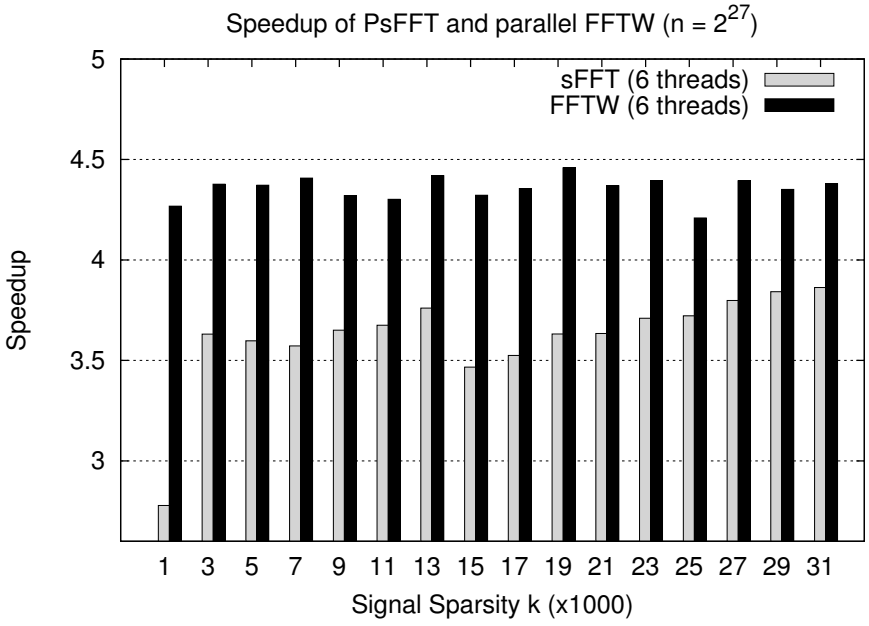
Signal size 2^n	**PsFFT (sec)**	**FFTW (sec)**	**Speedup of PsFFT vs. FFTW**
19	0.05	0.01	0.27
20	0.07	0.02	0.34
21	0.09	0.05	0.59
22	0.10	0.13	1.26
23	0.14	0.27	1.98
24	0.18	0.48	2.69
25	0.25	0.96	3.82
26	0.30	1.78	5.87
27	0.59	3.64	6.15
28	0.84	7.72	9.24

(b) Fix $n = 2^{27}$, vary signal sparsity k from 1000 to 31000

Signal sparsity k	**PsFFT (sec)**	**FFTW (sec)**	**Speedup of PsFFT vs. FFTW**
1000	0.61	3.70	6.03
3000	0.72	3.61	5.00
5000	1.02	3.59	3.52
7000	1.33	3.59	2.69
9000	1.65	3.68	2.23
11000	2.07	3.63	1.76
13000	2.58	3.59	1.40
15000	2.55	3.62	1.42
17000	2.89	3.59	1.24
19000	3.18	3.54	1.12
21000	3.46	3.60	1.04
23000	3.78	3.55	0.94
25000	4.16	3.75	0.90
27000	4.49	3.59	0.80
29000	4.79	3.62	0.76
31000	5.20	3.59	0.69



(a) Speedup of PsFFT and FFTW ($k = 1000$)



(b) Speedup of PsFFT and FFTW ($n = 2^{27}$)

Figure 5.11: Speedup of PsFFT and FFTW (6 threads)

PsFFT becomes slower than FFTW when signal spectra become dense.

Figure 5.11(b) plots the speedup for various signal sparsity k with fixed signal size n. From the figure, we can see that the speedup of PsFFT is 3.61x on average, while for FFTW, the average speedup is 4.35x. While the parallel speedup of PsFFT is slightly lower than FFTW, it is still way faster than FFTW for a wide range of signal size n and sparsity k.

5.6.3 Experimental Results – Signal Precision

In this subsection, we evaluate the performance of PsFFT for input data sets of single-precision floating point numbers.

Execution Time vs. Signal Size N

In this experiment, we fix the signal sparsity $k = 1000$ and report the execution time of the compared algorithms for signal sizes n ranging from 2^{19} to 2^{28}.

Figure 5.12(a) plots the average execution time of the compared algorithms. From the figure, we can see that the result is consistent with the double-precision inputs. That is, the MIT sFFT is the slowest while the UH sFFT implementation reduces the execution time by 2x. The PsFFT, on the other hand, significantly reduces the execution time compared to the sequential UH sFFT and MIT sFFT implementation.

Table 5.4(a) gives the speedup of PsFFT over the sequential version of sFFT. It can be seen from the table that, compared to UH sFFT, the PsFFT achieves 2.9x

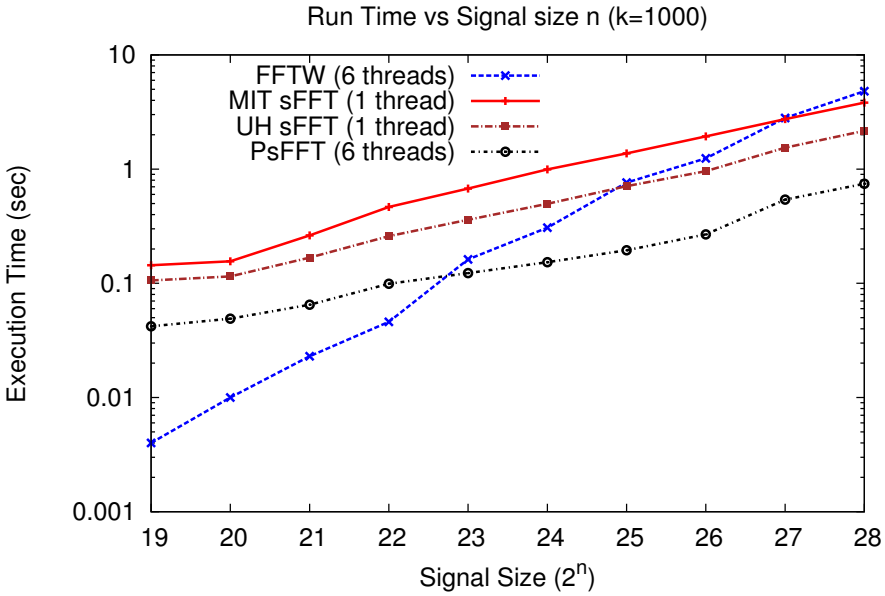
speedup on average. The PsFFT achieves the average speedup of 5.18x compared to the MIT sFFT implementation. The experimental result is relatively the same as the double-precision results.

Compared to parallel FFTW, Figure 5.12(a) shows that the execution time of PsFFT and FFTW are approximately linear in the log scale. However, the slope of the line of PsFFT is less than the slope of FFTW, which is a result of the sublinear runtime of sFFT. Furthermore, the PsFFT becomes faster than FFTW when the signal size is around 2^{23} (i.e., 8,388,608) at recovering the exact 1000 non-zero large coefficients. Compared to the sequential implementations of sFFT where the cross point is 2^{27} and 2^{25} for MIT and UH sFFT implementation, respectively, the PsFFT greatly reduces the signal size by 16x and 4x in order to be faster than FFTW.

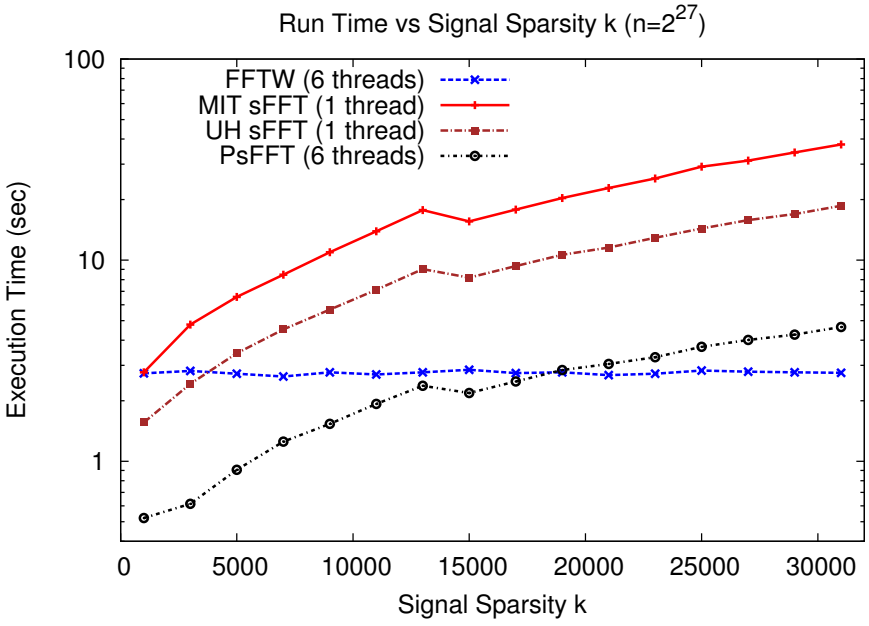
Table 5.5(a) shows the speedup of PsFFT over parallel FFTW at $k = 1000$. It can be seen from the figure that PsFFT is faster than FFTW from 0.1x to 6.45x with the increase of the signal size n. It is interesting to note that the gap of execution time between PsFFT and FFTW diverges fast with the signal size n due to the sublinear execution time of sFFT. This gives more credits to the PsFFT for large input signals.

Execution Time vs. Signal Sparsity k

In this experiment, we fix the signal size to $n = 2^{27}$ (i.e., 134,217,728) and evaluate the execution time of PsFFT vs. the number of non-zero frequencies k. We range the k from 1000, 3000, to as dense as $31,000$. Figure 5.12(b) illustrates the average



(a) Execution time vs. signal size ($k = 1000$)



(b) Execution time vs. signal sparsity k ($n = 2^{27}$)

Figure 5.12: Execution time of PsFFT vs. sFFT vs. FFTW (single precision)

Table 5.4: Speedup of PsFFT (6 threads) over sFFT (1 thread), single precision

(a) Fix $k = 1000$, varying n from 2^{19} to 2^{28}

Signal size 2^n	**PsFFT (sec)**	**UH sFFT (sec)**	**MIT sFFT (sec)**	**Speedup PsFFT vs. UH sFFT**	**Speedup PsFFT vs. MIT sFFT**
19	0.04	0.11	0.14	2.52	3.43
20	0.05	0.12	0.16	2.35	3.18
21	0.07	0.17	0.26	2.58	4.05
22	0.10	0.26	0.47	2.62	4.71
23	0.12	0.36	0.68	2.93	5.49
24	0.15	0.50	1.00	3.25	6.50
25	0.20	0.71	1.37	3.64	7.04
26	0.27	0.96	1.94	3.58	7.22
27	0.54	1.54	2.74	2.85	5.06
28	0.75	2.17	3.82	2.91	5.12
			average	**2.92**	**5.18**

(b) Fix $n = 2^{27}$, varying k from 1000 to 31000

Signal sparsity k	**PsFFT (sec)**	**UH sFFT (sec)**	**MIT sFFT (sec)**	**Speedup PsFFT vs. UH sFFT**	**Speedup PsFFT vs. MIT sFFT**
1000	0.52	1.56	2.77	2.99	5.31
3000	0.61	2.424	4.783	3.95	7.79
5000	0.91	3.444	6.561	3.80	7.24
7000	1.25	4.523	8.462	3.61	6.76
9000	1.53	5.677	10.949	3.70	7.14
11000	1.93	7.116	13.906	3.69	7.22
13000	2.37	9.025	17.745	3.80	7.47
15000	2.18	8.185	15.576	3.75	7.13
17000	2.49	9.349	17.834	3.75	7.15
19000	2.84	10.64	20.368	3.75	7.17
21000	3.05	11.545	22.827	3.79	7.49
23000	3.29	12.916	25.457	3.92	7.74
25000	3.71	14.362	29.145	3.87	7.86
27000	4.01	15.811	31.263	3.94	7.80
29000	4.26	16.95	34.324	3.98	8.05
31000	4.65	18.633	37.577	4.01	8.08
			average	**3.77**	**7.34**

Table 5.5: Speedup of PsFFT over parallel FFTW (6 threads), single precision

(a) Fix $k = 1000$, vary n from 2^{19} to 2^{28}

Signal size 2^n	**PsFFT (sec)**	**FFTW (sec)**	**Speedup of PsFFT vs. FFTW**
19	0.04	0.00	0.10
20	0.05	0.01	0.20
21	0.07	0.02	0.35
22	0.10	0.05	0.46
23	0.12	0.16	1.32
24	0.15	0.31	2.01
25	0.20	0.76	3.90
26	0.27	1.24	4.64
27	0.54	2.80	5.18
28	0.75	4.81	6.45

(b) Fix $n = 2^{27}$, vary signal sparsity k from 1000 to 31000

Signal sparsity k	**PsFFT (sec)**	**FFTW (sec)**	**Speedup of PsFFT vs. FFTW**
1000	0.52	2.74	5.24
3000	0.61	2.81	4.58
5000	0.91	2.73	3.01
7000	1.25	2.64	2.11
9000	1.53	2.77	1.81
11000	1.93	2.70	1.40
13000	2.37	2.77	1.17
15000	2.18	2.85	1.30
17000	2.49	2.75	1.10
19000	2.84	2.77	0.97
21000	3.05	2.68	0.88
23000	3.29	2.73	0.83
25000	3.71	2.83	0.76
27000	4.01	2.79	0.70
29000	4.26	2.77	0.65
31000	4.65	2.75	0.59

execution time of the compared algorithms. We can see from the figure that the PsFFT significantly reduces the execution time compared with MIT sFFT and UH sFFT implementations.

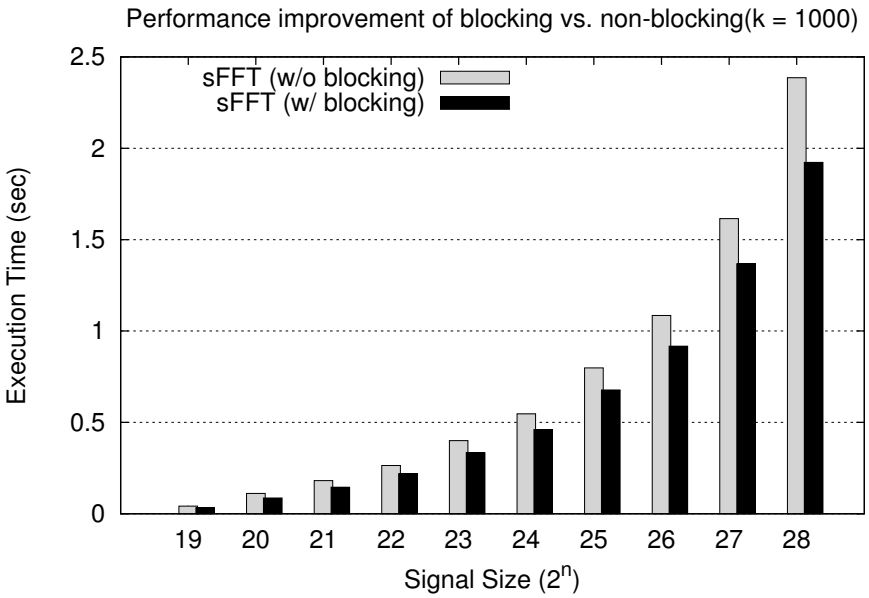
Table 5.4(b) shows the speedup of PsFFT over the sequential version of sFFT when we fix the signal size n and vary the signal sparsity k. It can be seen from the table that, compared to UH sFFT, the PsFFT achieves 3.77x speedup on average. The PsFFT achieves the average speedup of 7.34x compared to the MIT sFFT implementation.

Compared to parallel FFTW, Figure 5.12(b) shows that the PsFFT is faster than FFTW for k up to 19000. Compared to the sequential MIT and UH sFFT implementation where the cross point is at $k = 1000$ and $k = 3000$, the PsFFT greatly expands the signal sparsity, indicating more applications with broader signal bands can fit into the PsFFT.

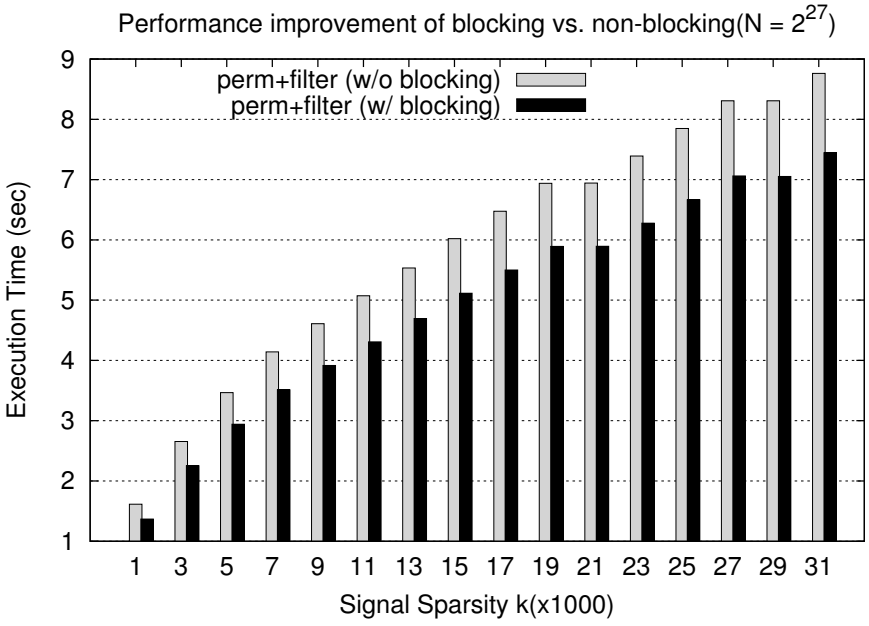
Table 5.5(b) shows the speedup of PsFFT over parallel FFTW at $k = 1000$. As is shown in the figure, the PsFFT is faster than FFTW by 5.2x at $k = 1000$ and the gap reduces with larger sparsity k. Finally PsFFT becomes slower than FFTW when signal spectra become dense.

5.6.4 Effects of Data Locality Optimization

In this subsection, we evaluate the performance improvements by employing the blocking technique. Figure 5.13 shows the execution time of the *perm+filter* before and after using the blocking technique. It can be seen from the figure that, the



(a) Execution time of blocking vs. non-blocking in PsFFT ($k = 1000$)



(b) Execution time of blocking vs. non-blocking in PsFFT ($n = 2^{27}$)

Figure 5.13: Execution time of blocking vs. non-blocking in PsFFT

Table 5.6: Cache Miss Rate on blocking vs. non-blocking

	blocking	**non-blocking**
L1 Miss Rate	11.35%	15.63%
LL Miss Rate	11.29%	15.54%

blocking reduces the average execution time by 20%.

Furthermore, we also measure the cache miss rate before and after applying the blocking optimization. Table 5.6 shows the L1 and LL cache miss rate results. As is shown in the table, the blocking optimization reduces the L1 and LL cache miss rates from 15.63% to 11.35%, a 38% cache performance improvement. The result justifies the effectiveness of the blocking optimization.

5.7 Summary

In this chapter, we present our parallel sFFT implementation, namely PsFFT, on multicore CPUs. We start with discussions on the challenges in paralleling the sFFT at scale. After that, we introduce the PsFFT algorithm and present how we address these challenges. We evaluate the performance of the PsFFT on an Intel Sandy Bridge architecture with 6 cores in one socket. The experimental results show that PsFFT achieves more than 3x speedup compared to our UH sequential sFFT implementation, and more than 5x speedup if compares with the MIT's original sFFT implementation, on 6 threads. Compared to the parallel version of FFTW, the PsFFT reduces the execution time by up to 9x for a considerable wide range of signal sizes and sparsity. Note that the performance gap is expected to be greater for

larger signals. It indicates a variety of applications with wide signal bands can fit into the sFFT and get the performance improvement by using the sFFT.

In next chapter, we will move to another parallel architecture, namely GPUs. GPUs become widely used as a high-performance computing platform due to its impressive computing capacity and the affordable cost. However, it is a non-trivial task to port the sFFT to GPUs because of the huge divergence in architectures compared to multicore CPUs. In next chapter, we will present the GPUs challenges and discuss the approaches we parallelize the sFFT on GPUs.

Chapter 6

cusFFT: A CUDA-based Sparse FFT on Accelerator-based Architectures

6.1 Overview

Parallel processors such as GPUs have played a significant role in the practical implementation of many numerical applications. The computations that arise in these applications lend themselves naturally to efficient parallel implementations. In this chapter, we present a high-performance parallel algorithm for computing sFFT on GPUs, namely cu*s*FFT, using CUDA programming language [1]. Before we present

[1]The majority of contents in this chapter are based on our prior publication in [20]

the cu*s*FFT algorithm, we will briefly review the salient details of NVIDIA's state-of-the-art GPU architecture and the CUDA parallel programming model in Section 6.2.

Although the increase in the number of cores and memory bandwidth on modern GPUs present an opportunity to improve the performance through sophisticated parallel algorithm design, numerous challenges that need to be addressed to deliver optimal performance. These challenges include evenly partitioning the workload to maximize hardware utilization, minimizing the need for global synchronization that may impede the parallelism in the fine-grained GPU computing, reducing the number of redundant operations caused by the parallelization, and coalescing access to the global memory whenever possible. The parallelization and optimization techniques used in the cu*s*FFT algorithm meet the challenges mentioned above for GPU architectures. Section 6.3 discuss some challenges in order to efficiently port sFFT to GPUs.

6.2 Introduction to GPUs and CUDA

Before discussing the design and implementation of our cu*s*FFT algorithm on GPUs, we very briefly review the salient details of NVIDIA's current GPU architecture and the CUDA parallel programming model. GPU is being increasingly adopted as a general-purpose computing platform to accelerate a vast majority of scientific and engineering applications.

While different NVIDIA GPUs offer various hardware configurations, in this paper, we focus on state-of-the-art Kepler GK110 architecture [4] as a testbed. A

full Kepler GK110 configuration consists of an array of 15 Streaming Multiprocessors(SM), each of which features 192 single-precision CUDA cores, and each core has fully pipelined floating-point and integer arithmetic logic units. Each SM could access up to 65536 registers. Thread management, including thread creation, scheduling, and synchronization are managed entirely by the hardware, so essentially the overhead is minimum. To efficiently manage the large number of threads on the hardware, the SM schedules threads in groups of 32 parallel threads called a *warp*. In the Kepler GK110 architecture, each SM features four warp schedulers, each with dual instruction dispatch units, allowing four warps to be issued and executed concurrently.

In the memory subsystem, each SM has 64 KB of on-chip memory which can be configured as shared memory/L1 cache, or both. In addition to L1 cache and shared memory, Kepler also introduces a 48 KB cache for data that is known to be read-only for the duration of the function. Use of the read-only path is beneficial because it takes the load footprint off of the shared/L1 cache path. In addition, the read-only data cache provides higher tag bandwidth. Use of the read-only path can be managed automatically by the compiler or explicitly by the programmer by using `__ldg()` intrinsic. The GK110 GPUs also equip a DDR memory up to 6 GB that can be addressable by all the SMs.

From the programmer's perspective, a CUDA program invokes parallel functions called *kernels* that execute across many parallel threads. A group of threads is organized as a *block*, and an array of blocks are organized as a *grid*. A block is a group of concurrent threads that can interact with each other through synchronization and

per-block shared memory space private to that block. When invoking a kernel, the programmer specifies both the number of blocks and the number of threads per block to be created when launching the kernel. So we can essentially map the CUDA's hierarchy of threads to the hierarchy of processors on a GPU. A GPU executes on one or more kernel grids; an SM executes one or more blocks; and CUDA cores and other execution units in the SM execute thread instructions.

6.3 GPU Challenges

To achieve maximal performance on the GPU platform, in many cases, it requires a deeper understanding of the memory hierarchy and the execution model of the hardware. For instance, it is very important to follow the right memory access pattern to the global memory failing which performance can be affected. To achieve good performance, all threads of a warp should read/write global memory in a coalesced way, i.e., the k-th thread accesses the k-th word in a cache line. Non-coalesced memory access (meaning that it strides across memory lines in the global memory) could lead to more memory transactions than necessary. Because a global memory transaction incurs hundreds of cycles of latency, non-coalesced memory access could significantly degrade the effective throughput of GPUs. An additional challenge is to find an effective way to partition the workload evenly among the hundreds or even thousands of CUDA cores. Parallelism if too fine-grained can result in insufficient balance of work per thread, on the other hand, if a thread has too much workload, this may over-pressure the registers per core and incur more register spilling behaviors.

It is very difficult to design an effective sFFT algorithm that can achieve a high level of parallelism at the same time maximize utilization on the GPU. Parallelizing the algorithm is even more challenging due to loop-carried dependencies in the most time-consuming kernel. The algorithm being heavily memory-bound leads to the relatively small amount of workload per thread, this is yet another performance barrier. For the rest of the chapter, we will highlight potential solutions to these major challenges.

6.4 GPU Sparse FFT

In this section, we present the cu*s*FFT algorithm, a parallel algorithm for computing the sparse FFT on GPUs. We will still adopt a step-by-step approach by partitioning the sFFT into multiple major stages, and discuss the parallel approach respectively. In this section, we will mainly outline the most straight-forward techniques to parallelize the sFFT. Some optimizations will be discussed in Section 6.5.

6.4.1 Stage 1&2: Random Spectrum Permutation and Filtering

The sFFT starts with convolving the permuted input signal with a well-designed filter and binning them into a small number of buckets. Algorithm 2 shows the code snippet of the inner loop in the sequential implementation. Noted that the iterations in the loop is `filter_size` rather than signal size `n` because the tails of the filter is

Algorithm 2 Pseudo code for serial permutation and filtering

```
1: Input: signal[n], filter[filter_size]
2: Output: buckets[B]
3: procedure PermFilter(signal, filter, buckets, B, n,
   filter_size)
4:     Initialize: index ← init_val
5:     while gcd(a, n) ≠ 1 do                          ▷ a,n are co-prime
6:         a ← random() mod n
7:     end while
8:     ai ← mod_inverse(a)                             ▷ ai is modular inverse of a
9:     for i ← 1, B do
10:        buckets[i] ← 0                              ▷ initialize buckets
11:    end for
12:                          ▷ Main body of spectrum permutation and filtering
13:    for i ← 1, filter_size do
14:        buckets[i mod B] += signal[index] × filter[i]
15:        index ← (index + ai) mod n
16:    end for
17: end procedure
```

almost zero, so it is not needed to convolve the signal points to a zero-sized filter.

There are multiple challenges need to be addressed in order to efficiently port the sFFT to GPUs. Some of the challenges are independent to specific architectures. For instance, *index mapping* technique is used to break the loop-carried dependence which is an essential step to port sFFT to any parallel architectures. Since we have already discussed these techniques when we parallelized the sFFT to multicore CPUs (i.e., PsFFT in Chapter 5), this section we mainly highlight on the techniques specific to massively parallel architectures such as GPUs.

GPU Bucket Tiling

Similar to PsFFT, there is a *hash collision* issue caused by binning the permuted signal into a reduced set of buckets. Collisions arise when multiple threads try to update the same bucket simultaneously. For instance, as is shown in Algorithm 2, signal on the loop iterations `i = 0, B, 2B, ...`, are binned into the same bucket, leading to the hash collision.

Compared to multicore CPU architecture, a GPU typically has thousands or even tens of thousands number of cores. It is a unique challenge to address the "hash collision" problem since there would be as many as hundreds numbers of threads updating the same bucket concurrently, leading to serialize the thread execution. As was introduced in Section 6.2, GPU threads are grouped and executed in a lock-step way in a unit of wrap (32 threads). Each time four wraps of threads will be issued and executed concurrently. Therefore, thread serialization will seriously impede the GPU performance.

There are several major approaches to address the *hash collision* issue. One commonly used approach is to use histogram on GPUs. Specifically, each thread creates its own private copy of sub-histogram and performs the local reduction, which is collision-free. At the final stage, all threads need to combine all the local private copy of the histograms into a global final histogram by using atomic operations.

Although this approach might be a general solution to many applications, unfortunately, it has two major drawbacks for the sFFT problem. First of all, the

per-thread approach (i.e., each thread maintains a local histogram) requires a significant amount of memory replication in the size of $B \times N$ bins, where B is the number of bins and N is the number of threads. For GPUs with thousands of threads, it is quite obvious that this approach can be extremely expensive in terms of memory space.

Further, to minimize the memory access latency, most of the implementations in the *per-thread* approaches [32, 73] utilize on-chip GPU shared memory to store the private sub-histograms. Unfortunately, the size of the shared memory is limited to 48KB even if the state-of-the-art Nvidia Tesla K20x GPUs is considered. This significantly limits the number of bins to 256 or even smaller for a conventional histogram computation [73]. This is a major limitation for sFFT which has much larger number of buckets, $B = \sqrt{nk/logn}$. For instance, for even a fairly small signal with size of size of $n = 2^{18}$ and $k = 1000$, B could be as large as 3,816. The number of buckets B grows quickly with the size of the input signal and sparsity k.

For a typical GPU architecture with size of 48 KB shared memory, suppose each element in the histogram has *complex double* type (i.e., 16 byte), a histogram with number of 3816 elements would need at least $3816 \times 16/1024 = 59.6KB$ memory space, which is greater than the size of the shared memory. This indicates that the shared memory could not even hold a single sub-histogram for even relatively a small input signal size.

Some of the histogram computation approaches in [32, 73] use *per-warp* or *per-block* replication instead of the *per-thread* approach, i.e, privatize the sub-histograms per each warp or per thread block. Since a warp usually consists of 32 threads and a

thread block size could reach to as large as 512 for mainstream GPUs, this approach could significantly relieve shared memory pressure. However, the main drawback of this approach is that it still needs atomic operations to update the sub-histograms within a warp or thread block. The usage of atomic operations can be a major bottleneck to performance. In addition, the atomic merging of sub-histograms into a global one at the final stage can become another bottleneck of the performance.

To address this issue, we propose a GPU *bucket tiling* approach. The idea is based on an observation with respect to histogram computation, which accesses the input data array sequentially (or follow a certain pattern), but updates the bucket array in a data-dependent (i.e., random) way. The issue with sFFT, however, is just as opposite: accessing the input signal is random while updating the buckets is predictable, as shown in Algorithm 2. Specifically, the number of rounds to access the buckets, `filter_size`, can be divided into `filter_size/B` rounds. The `filter_size` is divisible by `B`, since both of them are powers of 2. As a result, there is no collision within a round, i.e., *intra*-round, and collision occurs only across rounds, i.e., *inter*-round. For instance, updating the buckets when i equals to `(0,B)` is collision-free and collision only occurs when `i = 0, B, 2B, etc..`

Based on this observation, we propose a GPU *bucket tiling* approach. The basic idea is to partition the original loop iteration into two nested loops, where the outer loop is collision-free and suitable to be mapped to each CUDA thread and the inner loop is processed by one thread. Algorithm 3 shows the pseudo code with loop partition employed for perm+filter step. The outer loop, which has the size of the buckets, is parallelizable and mapped to each CUDA thread, each thread executes

Algorithm 3 Pseudo code for GPU permutation and filtering

```
1: Input: signal[n], filter[filter_size], buckets[B], ai
2: Output: buckets[B]
3: procedure PERMFILTER(signal, filter, buckets,
   B, n, filter_size, ai)
4:     rounds ← filter_size/B
5:     for all tid ← 1, B in parallel do
6:         Initialize myBucket ← 0
7:         for j ← 1, rounds do
8:             off ← idx + B × j
9:             index ← off × ai mod n
10:            myBucket+ = signal[index] × filter[off]
11:        end for
12:        buckets[tid] ← myBucket
13:    end for
14: end procedure
```

the reduction operation for one bucket, independently.

Compared to the conventionally GPU histogram reduction approaches, the bucket tiling approach has mainly three advantages: (a) Bucket replication is not required. Each CUDA thread is responsible for one bucket. (b) Does not require the sub-histograms to merge to a single histogram, as a result we avoid using atomic operations (c) Strikes a balance between fine-grained and coarse-grained parallelism, since the size of the inner loop is very small, each thread does a small amount of work.

6.4.2 Stage 3: B-dimensional cuFFT

After binning the spectra into a smaller number of B buckets, the B-dimensional FFT is performed. In our GPU algorithm, we simply employ the cuFFT [7] to perform this function. Furthermore, since this step will be repeated for the number

Algorithm 4 Pseudo code for sort & select

```
1: Input: buckets[B], k
2: Output: buckets[k], J[k]          ▷ Location and value of the top-k largest elements
3: procedure SORTSELECT(buckets, J, B, k)
4:     for all tid ← 1, B in parallel do
5:         J[tid] = tid                                              ▷ Store the index
6:     end for
7:     ReverseSortByValue(buckets[B], J[B])
8:     Select(buckets[k], J[k])
9: end procedure
```

of `outer_loops` times, of each calculates the same dimension of cuFFT, we use the batched mode of cuFFT and we compute cuFFT only once. By sharing the twiddle factors, the batched cuFFT combines the number of `outer_loops` transforms into one function call, which is much faster than repeatedly calling the cuFFT function.

6.4.3 Stage 4: Cutoff

After we apply the B-dimensional cuFFT, there are buckets containing potentially large Fourier coefficients in the frequency domain. Since the spectra is sparse in frequency, very few of them are potentially large. The objective of the cutoff function is to select the top k largest number of coefficients in the magnitude and store their locations.

In the baseline implementation, we apply the sort&select method, as is shown in Algorithm 4. Specifically, we first sort the B buckets in a decreasing order and store the locations of values of the top k largest elements. Here we use the sorting algorithm to perform the cutoff function for several reasons. First of all, sorting is a key building block of many algorithms. It has received a large amount of attention

in both sequential and parallel implementations [22, 23, 85]. For GPU algorithms, the studies in [57, 74, 84] suggest an increase in performance using the right sorting algorithms.

In this dissertation, we use Nvidia's Thrust library [10] to implement the sort&select algorithm. Thrust is a CUDA-based library performing the GPU-accelerated sort, scan, transform, and reduction operations. Using Thrust has several benefits: Thrust is Nvidia's official library integrated with state-of-the-art CUDA API. As a result, our implementation depending on the Thrust could achieve better robustness, performance and maintains, compared to relying on any external third-party libraries. Nevertheless, sorting is sub-optimal especially for $k \ll n$. We discuss an alternate technique, in the next section, to fulfill the cutoff function.

6.4.4 Stage 5: Reverse Hash Functions for Location Recovery

The Stages from 1 to 4 define a hash function that maps each coefficient into one of the buckets. The "real" locations of large coefficients need to be reversed by eliminating the phase changes caused by spectrum permutation and filtering. Algorithm 5 shows the pseudo code for the parallel location recovery on GPUs. Specifically, we launch a number of k threads and each thread independently computes the reverse hash function to recover the location of the potential large coefficients. As described in Chapter3, stages 1 to 5 repeats a number of *location loops* times, so for each inner loop (determining the location), we increment the occurrences(score) of the location

Algorithm 5 Pseudo code for GPU location recovery

```
1: Input: J[k], score[n], a, B
2: Output: hits[num_hits]
3: procedure LOCRECOVERY(J, hits, score, k,
   num_hits, a, n, B)
4:     for all tid ← 1, k in parallel do
5:         my_J ← J[tid]
6:         low ← (ceil((my_J − 0.5) × n/B) + n) mod n
7:         high ← (ceil((my_J + 0.5) × n/B) + n) mod n
8:         loc ← (low × a) mod n
9:         for j ← low, high do
10:            atomicAdd(score[loc], 1)
11:            if score[loc] == threshold then
12:                atomicAdd(num_hits, 1)
13:                hits[num_hits] ← loc
14:            end if
15:            loc ← (loc + a) mod n
16:        end for
17:    end for
18: end procedure
```

recovered. Once the score of the frequency equals to the threshold, we consider this location of the coefficient to be large and add it to `hits`.

6.4.5 Stage 6: Magnitude Reconstruction

The purpose of this step is that given the locations of large coefficients we need to reconstruct the magnitudes. Since the inner loops are repeated for a number of L loops, given a specific location $r \in I'$, we can get a set of magnitudes $\{\hat{x}_i^r | i \in L\}$. So the magnitude $\hat{x^r}$ is computed as the median of the candidate magnitudes for all the L location loops, i.e., median$(\{\hat{x}_i^r | i \in L\})$. To parallelize this step, we launch the number of threads equal to `num_hits`, which is the number of potential large coefficients obtained from the previous step. Each thread computes the reconstruction of

Algorithm 6 Pseudo code for GPU magnitude reconstruction

```
1: Input: hits[num_hits], buckets[B], ai, filter[filter_size]
2: Output: loc[num_hits], val[num_hits]
3: procedure MagRecon()
4:     for all tid ← 1, num_hits in parallel do
5:         my_hits ← hits[tid]
6:         n_div_B ← n/B
7:         for j ← 1, outer_loops do
8:             permuted_loc ← ai[j] × my_hits mod n
9:             hashed_to ← permuted_loc/n_div_B
10:            dist ← permuted_loc mod n_div_B
11:            if dist > n_div_B/2 then
12:                hashed_to ← (hashed_to + 1) mod B
13:                dist ← dist − n_div_B
14:            end if
15:            dist ← (n − dist) mod n
16:            mag[j] ← buckets[j][hashed_to]/filter_freq[dist]
17:        end for
18:        sort(mag, outer_loops)
19:        median ← (outer_loops − 1)/2
20:        loc[tid] ← my_hits
21:        val[tid] ← mag[median]
22:    end for
23: end procedure
```

the magnitude for a given location independently.

6.5 Optimizations

In Section 5.4, we had presented some basic techniques to parallelize sFFT for GPUs. In this section, we present some performance optimization techniques for sFFT achieving better performance on GPUs.

6.5.1 Fast K-selection Algorithm

As was presented in Section 6.4, stage 4 of the algorithm applied a cutoff function to select the k largest elements from a set of B buckets. In Section 6.4, we employed a basic *sort&select* algorithm which basically sort the entire array and select the k largest elements from the sorted set. At that time we decided to use the *sort&select* algorithm mainly because sorting is well supported on GPU architecture for which we can expect an acceptable performance, robustness and ease of maintains in sFFT.

The approach is inefficient when the input data increases, thus, sorting the entire list becomes too expensive. That is, $BlogB$ operations with a typical sorting algorithm only leads to k useful data points, where $k \ll B$. Therefore, one could expect to accomplish this task in a time proportional to the data size, i.e., at the linear time.

Further, beyond sFFT, there also exists numerous applications requiring the same use case where only the k largest values in a list are retained while the remaining entries are ignored or set to zero [16, 46]. Linear-time fast selection algorithms have received a large amount of attention in sequential and parallel implementations [23, 44]. For GPU algorithms, Alabi *et. al* [13] proposed a *BucketSelect* algorithm. Similar to bucket sorting, the proposed algorithm works well only when the data is uniformly distributed, i.e., the number of elements assigned to each bucket is roughly equal. For the sFFT algorithm, unfortunately, only very few of the buckets are large while the rest of them are almost empty.

In this dissertation, we propose a fast selection algorithm which is simple but

Algorithm 7 Pseudo code for GPU fast k-selection algorithm

```
1: Input: buckets[B], k
2: Output: J[k]
3: procedure FastSelect(buckets, J, B, k)
4:     Initialize: count ← 0
5:     for all tid ← 1, B in parallel do
6:         key ← tid
7:         value ← buckets[tid]
8:         if value ≥ thresh then
9:             myCount ← atomicAdd(count, 1)
10:            J[myCount − 1] ← key
11:        end if
12:    end for
13: end procedure
```

effective in sFFT. As shown in Algorithm 7, we assign a number of B threads and each thread processes one element in the buckets. If the value in the buckets is greater than the threshold, the element is chosen and the index is stored. It is noted that the choice of the threshold is important for the algorithm. If it is too small, many small coefficients will be picked up and falsely treated as "large". On the other hand, if the threshold is too large, some useful large coefficients will be lost. In this dissertation work, we choose the threshold based on a key observation: only the k out of B buckets are large while rest of them are very small with similar amplitudes. Consequently, the value of the threshold is chosen to be in the same order of the "small" noise coefficients, of which the value could be empirically obtained from past experience. Most of the time, this approach might yield slightly more than the number of k elements. Nevertheless, this minor *false positive* will not affect the correctness of the algorithm but only add slight extra time on computing the extra "large" coefficients for the rest of the stages.

6.5.2 Asynchronous Data Layout Transformation

Note that in the first two steps of the sFFT (perm+filter), the algorithm permutes the input signals and bins them into a smaller number of buckets. As shown in Algorithm 3 (line 10), since the `index` is calculated as `index = (i * ai ) % n`, the data reference pattern to the input signal (`signal[index]`) is largely strided. Such irregular memory reference access pattern leads to non-coalesced global memory accesses on GPUs, which creates memory traffic and is a significant bottleneck for achieving good performance.

Prior work [47, 88] relies on static compiler-based techniques that detect the irregularities and reorders the computations at compile time. In sFFT, since the *ai* is randomly generated, the irregular access pattern is *dynamic*, i.e., it will remain unknown until run time and even change during computation. Such dynamic nature severely limits the compiler techniques to be effectively adopted for the sFFT problem. Other conventional approaches relying on shared memory technique may achieve substantial performance improvement. Those techniques are very effective for block-based matrix multiplication as well as matrix transpose problems. In sFFT, however, since the data reference stride is too large, simply loading a block of the signal will easily stride out of the shared memory size. A number of prior studies [26,65] also explore runtime data transformation techniques that reorder the data layout at runtime. However, the issue here is that this approach can create overhead due to data transformation. problem of this approach is the overhead of data layout transformation will be added to the critical path of the application.

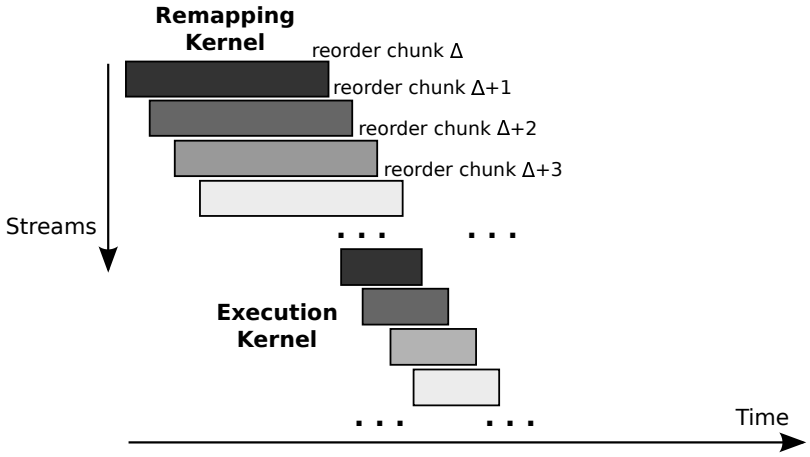


Figure 6.1: Asynchronous data layout transformation

To coalesce the memory access, we propose an *asynchronous* data layout transformation technique that reorders the data on-the-fly. To achieve this, the original non-coalesced computation kernel is split into two kernels: one performs the data layout transformation while the other one accesses the ordered data. In order to hide the overhead of data layout transformation, we take advantage of CUDA concurrent kernel executions where multiple kernels execute concurrently on different CUDA streams. Figure 6.1 shows an example illustrating the use of asynchronous data layout transformation technique to hide the overhead. The *remapping kernel* creates a chunk of new array, A', containing the coalesced data. The new order is created based on a desirable mapping technique between threads and data locations, i.e., for loop iteration i, $A'[i] = A[index]$, where A is the original input signal. The second kernel, *execution kernel*, computes the original program but directly accesses the reordered data A'. The chunk size is empirically chosen to be the bucket size B. So after reordering a chunk of B-size data, the second kernel launches a number of B threads that computes the B elements in a batch. Note that there is no data

dependence between two remapping kernels (so does execution kernel), so remapping the data chunk δ and $\delta + i$ can happen concurrently. The maximum depth of the concurrency depends on the CUDA compute capability that is different for different GPU architecture. For GK110 used in this paper, the maximum number of concurrently executed kernels is 32.

6.6 Performance Evaluation

We evaluate the performance of cu$s$FFT in this section. We compare the performance with:

- cuFFF, the fastest implementation for computing the ordinary full-sized FFT with the runtime of $O(nlog(n))$ on NVIDIA GPUs.
- We also evaluate against both of our implementations: the baseline implementation as was described in Section 6.4 and the performance improvements of optimizations discussed in Section 6.5.
- For completeness, we also compare with OpenMP version of sFFT (PsFFT) on multicore CPUs as was described in Chapter5, UH sequential sFFT implementation and MIT sFFT implementation.
- We also evaluate the performance of cusFFT with the parallel FFTW, one of the most widely used ordinary full-sized FFT library for multicore CPUs.

Table 6.1: GPU test-bench

GPU Type	**Nvidia Tesla K20x**
CUDA capability	3.5
No. of CUDA cores	2688 cores / 14 SMs
Processor clock	732 MHz
Shared memory size	64 KB
Global memory size	6 GB
Memory bandwidth	250 GB/s

6.6.1 Experimental Setup

Table 6.1 shows the experimental configurations for this evaluation. We evaluated the cu*s*FFT on NVIDIA Kepler K20x GPU. For the experiments on CPUs, we still use the same platform as PsFFT (i.e., Table 4.2). The CUDA compiler is NVIDIA's `nvcc` compiler with version 5.5. The FFTW library used is the version 3.4 with multi-threading configured.

6.6.2 Experimental Results – Double Precision

In this subsection, we evaluate the performance of cu*s*FFT for the input data as double-precision floating point numbers.

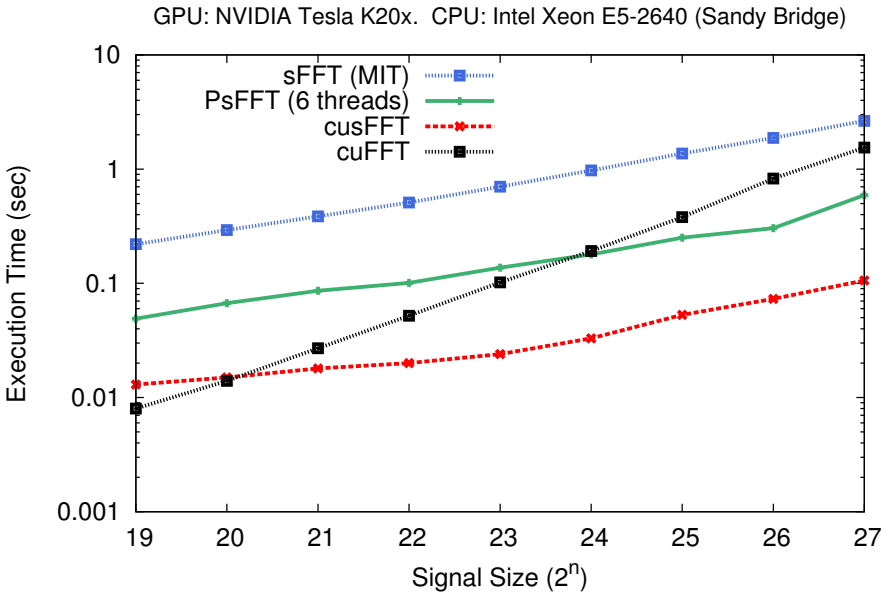
Execution Time vs. Signal Size n

In Figure 6.2(a), we have fixed the sparsity parameter $k = 1000$. We report the runtime of the algorithms compared for different signal sizes n ranging from 2^{19} to 2^{27}. The execution time is also shown in Table 6.2. In Figure 6.2(a), we plot the average execution time for cu*s*FFT, Parallel sFFT (PsFFT), the MIT's original sFFT

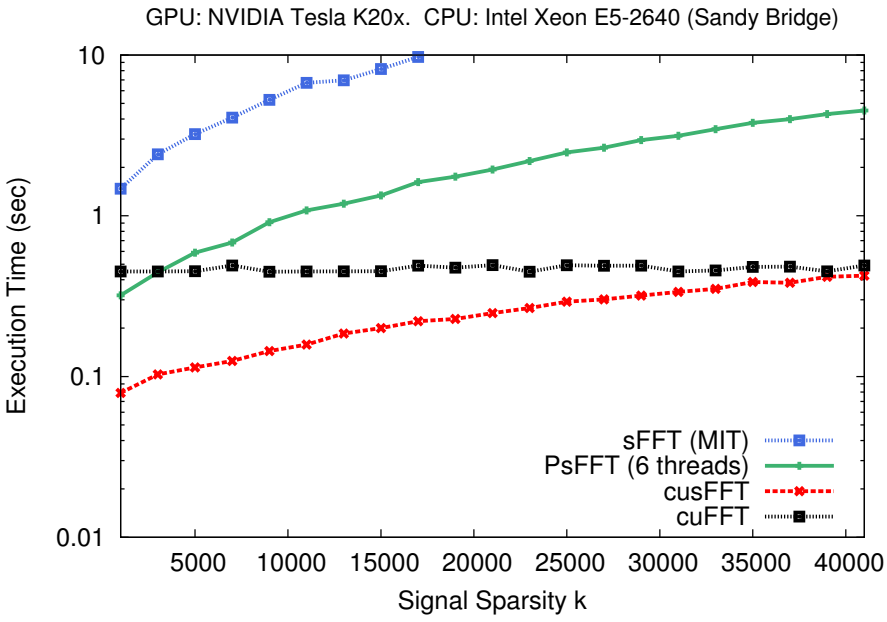
implementation and cuFFT. As expected, the cu$s$FFT performs the best, followed by the PsFFT and the MIT sequential sFFT. The figure also shows that the original MIT sFFT implementation is always slower than cuFFT. For the PsFFT, the figure shows that it is faster than cuFFT for signal size $n > 2^{24}$ in recovering the exact 1000 non-zero coefficients. The cusFFT, on the other hand, reduces the cross point as $n = 2^{20}$ to be faster than cuFFT. Figure 6.2(a) also shows that the execution time of cu$s$FFT and cuFFT are approximately linear in the log scale. However, the slope of the line for cu$s$FFT is less than the slope for cuFFT, which is a result of cu$s$FFTs sub-linear runtime. The gap becomes greater when the data size increases, meaning that speedup of cu$s$FFT will be more than cuFFT.

Table 6.4 shows the speedup of cu$s$FFT compared to the other versions of sFFT, namely PsFFT and MIT sFFT. From the table, we can see that the average speedup of cu$s$FFT over PsFFT is 4.28x on 6 threads. Compared to the MIT's original sFFT implementation, the average performance improvement is over 21x.

Compared to full-size standard FFT implementations, Table 6.6 shows the speedup of cu$s$FFT over cuFFT and parallel FFTW. As can be seen from the table, we increase the signal size n from 2^{19} to 2^{27} by fixing the signal sparsity $k = 1000$. Compared to the cuFFT, cu$s$FFT is faster by the factor of 0.5x to 11.15x. The gap is increasing with the size of input signals. Compared to the parallel FFTW, cu$s$FFT outperforms by the factor from 0.93x to 26.25x.



(a) Execution time vs. signal size ($k = 1000$)



(b) Execution time vs. signal sparsity k ($n = 2^{27}$)

Figure 6.2: Performance evaluation of cu*s*FFT with PsFFT, MIT sFFT and cuFFT

Table 6.2: Execution time (in second) of cusFFT ($k = 1000$)

Signal size 2^n	**cusFFT**	**PsFFT (6 threads)**	**MIT sFFT (1 thread)**	**cuFFT**	**FFTW (6 threads)**
19	0.01	0.05	0.22	0.01	0.01
20	0.02	0.07	0.29	0.01	0.02
21	0.02	0.09	0.39	0.03	0.05
22	0.02	0.10	0.51	0.06	0.13
23	0.03	0.14	0.70	0.11	0.27
24	0.04	0.18	0.97	0.23	0.48
25	0.06	0.25	1.37	0.45	0.96
26	0.08	0.30	1.87	0.98	1.78
27	0.14	0.59	2.65	1.55	3.64

Execution Time vs. Signal Sparsity k

In this experiment, we fix the signal size n to be $n = 2^{25}$ and change the signal sparsity k from 1000 to 41,000. Figure 6.2(b) shows the average execution time of the compared algorithms while the number of non-zero frequencies k ranges from 1000 to 41000. The same data points are shown in Table 6.3. Since cuFFT has a runtime of $O(nlogn)$, they are independent of the number of non-zero frequencies k, as can be seen in the figure.

Figure 6.2(b) also shows that the MIT's sFFT implementation is always much slower than the cuFFT. The PsFFT reduces the execution time such that it is faster than cuFFT only when the signal sparsity $k < 3000$ while for most of the data points shown in the figure, PsFFT is slower than cuFFT. The cu$s$FFT, on the other hand, is significantly shower than cuFFT until the signal sparsity k becomes as large as 41000.

Table 6.5 shows the speedup of cu$s$FFT compared to other versions of the sFFT

Table 6.3: Execution time (in second) of cusFFT ($n = 2^{25}$)

Signal sparsity k	**cusFFT**	**PsFFT (6 threads)**	**MIT sFFT(1 thread)**	**cuFFT**	**FFTW (6 threads)**
1000	0.08	0.32	1.47	0.67	0.93
3000	0.10	0.45	2.41	0.61	0.95
5000	0.11	0.59	3.22	0.61	0.92
7000	0.13	0.68	4.08	0.61	0.93
9000	0.14	0.91	5.27	0.61	0.93
11000	0.16	1.08	6.72	0.61	0.96
13000	0.19	1.19	6.97	0.61	0.95
15000	0.20	1.34	8.19	0.67	0.96
17000	0.22	1.62	9.74	0.64	0.97
19000	0.23	1.75	11.64	0.67	0.93
21000	0.25	1.94	13.19	0.61	0.93
23000	0.27	2.19	15.43	0.67	0.96
25000	0.29	2.48	17.53	0.64	0.94
27000	0.30	2.65	19.98	0.61	0.95
29000	0.32	2.96	21.75	0.66	0.97
31000	0.34	3.14	24.96	0.67	0.95
33000	0.35	3.46	28.88	0.61	0.95
35000	0.39	3.79	30.24	0.61	0.96
37000	0.38	3.99	34.80	0.64	0.94
39000	0.42	4.30	36.18	0.67	0.97
41000	0.42	4.52	39.15	0.67	0.94

Table 6.4: Speedup of cu*s*FFT vs. PsFFT (6 threads) and MIT sFFT (1 thread) ($k = 1000$)

Signal size 2^n	**cusFFT (sec)**	**PsFFT (sec)**	**MIT sFFT (sec)**	**Speedup cu*s*FFT vs. PsFFT**	**Speedup cu*s*FFT vs. MIT sFFT**
19	0.01	0.05	0.22	3.50	15.75
20	0.02	0.07	0.29	3.53	15.40
21	0.02	0.09	0.39	4.30	19.31
22	0.02	0.10	0.51	4.39	22.16
23	0.03	0.14	0.70	5.07	25.99
24	0.04	0.18	0.97	5.11	27.83
25	0.06	0.25	1.37	4.56	24.92
26	0.08	0.30	1.87	3.75	23.13
27	0.14	0.59	2.65	4.27	19.08
			average	**4.28**	**21.51**
			median	**4.30**	**22.16**

implementations including PsFFT and MIT's original sFFT. We can see from the table that cu*s*FFT is faster than PsFFT by the factor of 7.7x. Compared to the original MIT sFFT sequential implementation, it achieves over 55x performance improvement.

Table 6.7 shows the speedup of cu*s*FFT over the full-size FFT implementations. As can be seen from the table, the execution time of cu*s*FFT increases with the signal sparsity k while the execution time of fulls-size FFT is irrelevant of the signal sparsity. The table shows that the cu*s*FFT is largely faster than cuFFT by the factor from 8.47x to 1.58x. Compared to FFTW on 6 CPU threads, the cu*s*FFT is faster from 11.72x to 2.22x. The experimental results indicate that the cu*s*FFT largely outperforms the full-size standard FFT for a wide range of signal spectrum.

Table 6.5: Speedup of cu*s*FFT vs. PsFFT (6 threads) and MIT sFFT (1 thread) ($n = 2^{25}$)

Signal sparsity k	**cusFFT (sec)**	**PsFFT (sec)**	**MIT sFFT (sec)**	**Speedup cu*s*FFT vs. PsFFT**	**Speedup cu*s*FFT vs. MIT sFFT**
1000	0.08	0.32	1.47	4.05	18.65
3000	0.10	0.45	2.41	4.32	23.44
5000	0.11	0.59	3.22	5.18	28.24
7000	0.13	0.68	4.08	5.46	32.66
9000	0.14	0.91	5.27	6.33	36.56
11000	0.16	1.08	6.72	6.84	42.52
13000	0.19	1.19	6.97	6.43	37.66
15000	0.20	1.34	8.19	6.70	40.95
17000	0.22	1.62	9.74	7.33	44.07
19000	0.23	1.75	11.64	7.68	51.05
21000	0.25	1.94	13.19	7.83	53.17
23000	0.27	2.19	15.43	8.21	57.79
25000	0.29	2.48	17.53	8.49	60.04
27000	0.30	2.65	19.98	8.78	66.17
29000	0.32	2.96	21.75	9.28	68.17
31000	0.34	3.14	24.96	9.36	74.29
33000	0.35	3.46	28.88	9.85	82.28
35000	0.39	3.79	30.24	9.79	78.14
37000	0.38	3.99	34.80	10.42	90.87
39000	0.42	4.30	36.18	10.30	86.75
41000	0.42	4.52	39.15	10.66	92.32
			average	**7.77**	**55.51**
			median	**7.83**	**53.17**

Table 6.6: Speedup of cu*s*FFT vs. cuFFT vs. FFTW (6 threads) ($k = 1000$)

Signal size 2^n	**cusFFT (sec)**	**cuFFT (sec)**	**FFTW (sec)**	**Speedup cu*s*FFT vs. cuFFT**	**Speedup of cu*s*FFT vs. FFTW**
19	0.01	0.01	0.01	0.50	0.93
20	0.02	0.01	0.02	0.68	1.21
21	0.02	0.03	0.05	1.45	2.55
22	0.02	0.06	0.13	2.57	5.52
23	0.03	0.11	0.27	4.19	10.04
24	0.04	0.23	0.48	6.46	13.77
25	0.06	0.45	0.96	8.22	17.42
26	0.08	0.98	1.78	12.10	22.02
27	0.14	1.55	3.64	11.15	26.25

6.6.3 Experimental Results – Single Precision

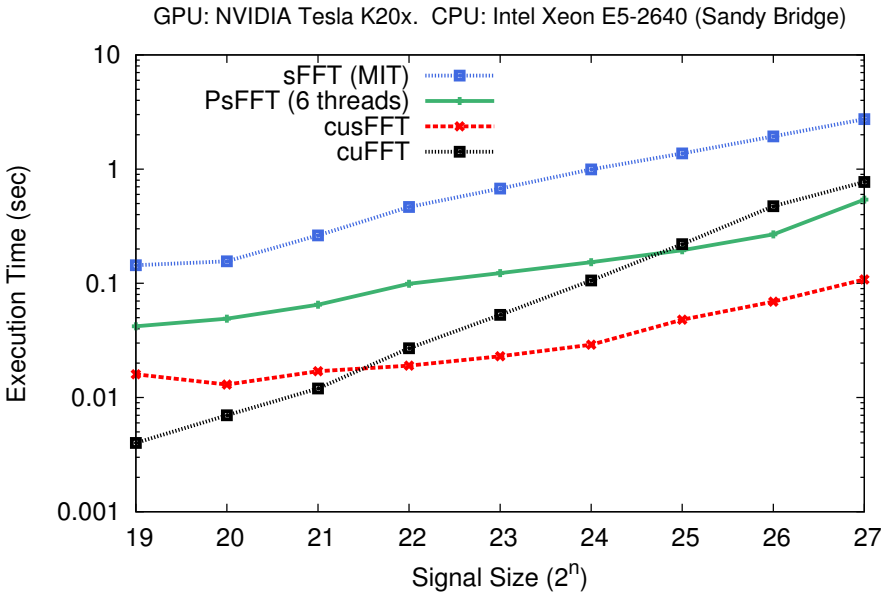
In this subsection, we evaluate the performance of cu*s*FFT for the input data as single-precision floating point numbers.

Execution Time vs. Signal Size n

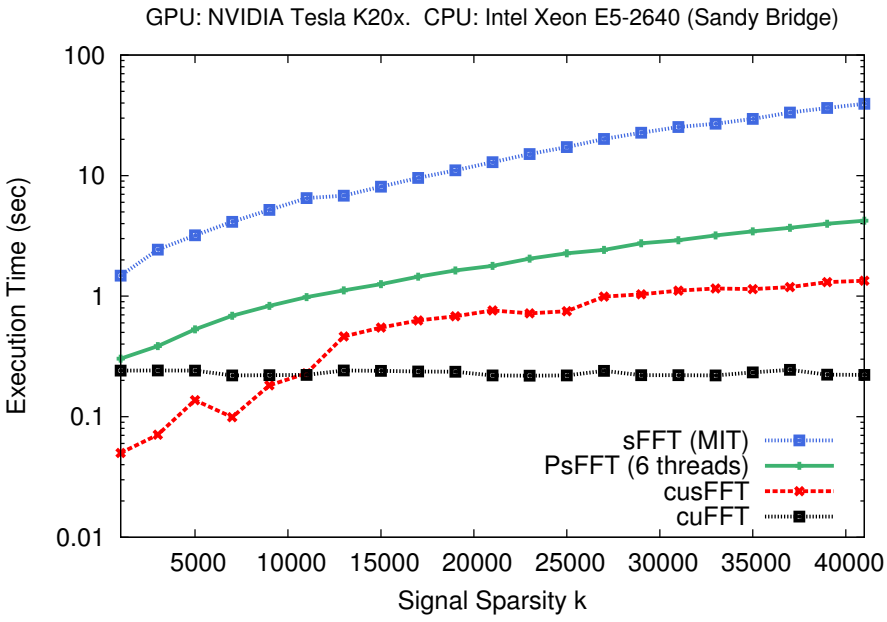
Figure 6.3(a) shows the execution time of the compared algorithms when the signal size ranges from 2^{19} to 2^{27} and fix the signal sparsity $k = 1000$. Similar to the double-precision results, the MIT sFFT performs the worst, always slower than the cuFFT. The PsFFT is slower than the cuFFT until $n = 2^{25}$. The cu*s*FFT reduces the cross point to be less than $n = 2^{22}$ in order to beat cuFFT. The data points of the execution time are also shown in Table 6.8.

Table 6.7: Speedup of cu$s$FFT vs. cuFFT vs. FFTW (6 threads) ($n = 2^{25}$)

Signal sparsity k	**cusFFT (sec)**	**cuFFT (sec)**	**FFTW (sec)**	**Speedup cu$s$FFT vs. cuFFT**	**Speedup cu$s$FFT vs. FFTW**
1000	0.08	0.67	0.926	8.47	11.72
3000	0.10	0.61	0.953	5.92	9.25
5000	0.11	0.61	0.924	5.39	8.11
7000	0.13	0.61	0.931	4.90	7.45
9000	0.14	0.61	0.929	4.24	6.45
11000	0.16	0.61	0.959	3.85	6.07
13000	0.19	0.61	0.948	3.30	5.12
15000	0.20	0.67	0.964	3.34	4.82
17000	0.22	0.64	0.971	2.89	4.39
19000	0.23	0.67	0.929	2.96	4.07
21000	0.25	0.61	0.927	2.46	3.74
23000	0.27	0.67	0.965	2.51	3.61
25000	0.29	0.64	0.937	2.19	3.21
27000	0.30	0.61	0.948	2.02	3.14
29000	0.32	0.66	0.972	2.07	3.05
31000	0.34	0.67	0.945	2.00	2.81
33000	0.35	0.61	0.951	1.75	2.71
35000	0.39	0.61	0.961	1.59	2.48
37000	0.38	0.64	0.94	1.66	2.45
39000	0.42	0.67	0.968	1.60	2.32
41000	0.42	0.67	0.941	1.58	2.22



(a) Execution time vs. signal size ($k = 1000$)



(b) Execution time vs. signal sparsity k ($n = 2^{27}$)

Figure 6.3: Execution time of cu$s$FFT, single precision

Table 6.8: Execution time (in second) of cusFFT ($k = 1000$), single precision

Signal size 2^n	**cusFFT**	**PsFFT (6 threads)**	**MIT sFFT (1 thread)**	**cuFFT**	**FFTW (6 threads)**
19	0.02	0.04	0.14	0.01	0.01
20	0.01	0.05	0.16	0.01	0.01
21	0.02	0.07	0.26	0.01	0.02
22	0.02	0.10	0.47	0.03	0.05
23	0.02	0.12	0.68	0.05	0.16
24	0.03	0.15	1.00	0.11	0.31
25	0.05	0.20	1.37	0.22	0.76
26	0.07	0.27	1.94	0.47	1.24
27	0.11	0.54	2.74	0.77	2.80

Table 6.9: Execution time (in second) of cusFFT ($n = 2^{25}$), single precision

Signal sparsity k	**cusFFT**	**PsFFT(6 threads)**	**MIT sFFT(1 thread)**	**cuFFT**	**FFTW (6 threads)**
1000	0.05	0.30	1.48	0.24	0.75
3000	0.07	0.39	2.43	0.24	0.77
5000	0.14	0.53	3.20	0.24	0.78
7000	0.10	0.69	4.13	0.22	0.78
9000	0.18	0.83	5.19	0.22	0.76
11000	0.23	0.98	6.50	0.22	0.74
13000	0.46	1.12	6.81	0.24	0.74
15000	0.55	1.26	8.10	0.24	0.72
17000	0.63	1.45	9.58	0.24	0.74
19000	0.68	1.63	11.07	0.24	0.74
21000	0.76	1.78	12.92	0.22	0.73
23000	0.72	2.05	15.10	0.22	0.75
25000	0.43	2.27	17.25	0.22	0.77
27000	0.99	2.42	20.12	0.24	0.75
29000	1.04	2.74	22.71	0.22	0.73
31000	1.11	2.91	25.27	0.22	0.75
33000	1.16	3.19	26.92	0.22	0.76
35000	1.15	3.45	29.51	0.23	0.75
37000	1.19	3.69	33.37	0.24	0.76
39000	1.31	3.99	36.41	0.22	0.74
41000	1.34	4.22	39.38	0.22	0.78

Execution Time vs. Signal Sparsity k

In this experiment, we evaluate the performance of cu$s$FFT by fixing the signal size to be 2^{25} and varying the signal sparsity k from 1000 to 41000. Figure 6.3 shows the execution time of the compared algorithms. It can be seen from the figure that the execution time of the MIT's sFFT and PsFFT are always greater than the cuFFT. The cu$s$FFT on the other hand, reduces the cross point to $k = 10000$. Note that in the work of developing the cu$s$FFT, we mainly focus on the double-precision inputs for the consideration of the numerical accuracy. We did not exploit much optimizations on single-precision numerical operations.

6.7 Summary

In this chapter, we present an effective parallel algorithm for computing sparse FFT on GPUs, namely cu$s$FFT. We report the bottlenecks in the algorithm that impedes the performance. We explore several suitable and optimized solutions to tackle these issues and demonstrate that the proposed cu$s$FFT algorithm is over 10x faster than cuFFT for large data size, and over 26x compared to the parallel FFTW on multicore CPUs. Moreover, compared to the sequential and parallel version of the sFFT implementations, the cu$s$FFT improves the performance by the factor by 4.3x and 21.5x, respectively, due to the power of GPU computing. The promising results indicate that cu$s$FFT is able to replace the full-size standard FFT routines widely used in a vast of scientific and engineering applications and is expected to gain the significant performance improvement.

Chapter 7

Parallel Sparse FFT on Heterogeneous Multicore Embedded Systems

7.1 Overview

In this chapter, we port the sFFT to a Texas Instruments (TI) KeyStone II platform, a high-performance heterogeneous System-on-Chip (SoC). As the architecture is relatively novel, we first give an overview of the architecture, from the perspective of CPU and memory subsystems in Section 7.2. After that, we present the major challenges in programming heterogeneous multicore embedded system in Section 7.3, and present the OpenMP accelerator model as a high-level programming model to address the challenge in Section 7.4.

We measure the memory access latency of the system's memory hierarchy in Section 7.5. Memory access latency is an important performance metrics as it can be used to measure the effectiveness of the data locality optimizations, and include the cache hits and misses as well as delays in buses and memory controllers.

In Section 7.6, we present the approaches we port the sFFT to the Texas Instruments KeyStone II ARM+DSP heterogeneous architecture. We also discuss the key techniques in exploiting locality of sFFT in Section 7.7. Finally, we show how these techniques can effectively improve the locality of the sFFT on the platform in Section 7.8.

7.2 Architecture Overview

The Texas Instruments (TI) Keystone II architecture integrates an octa-core C66X DSP with a quad-core ARM Cortex A15 RISC processor in a non-cache coherent shared memory environment. Figure 7.1 shows the block diagram of the device. The Cortex-A15 quad-cores are fully cache-coherent, although as on the C6678 the DSP cores do not maintain cache coherency. External memory bandwidth is doubled with dual DDR3 controllers. An additional Hyperlink interface is also included.

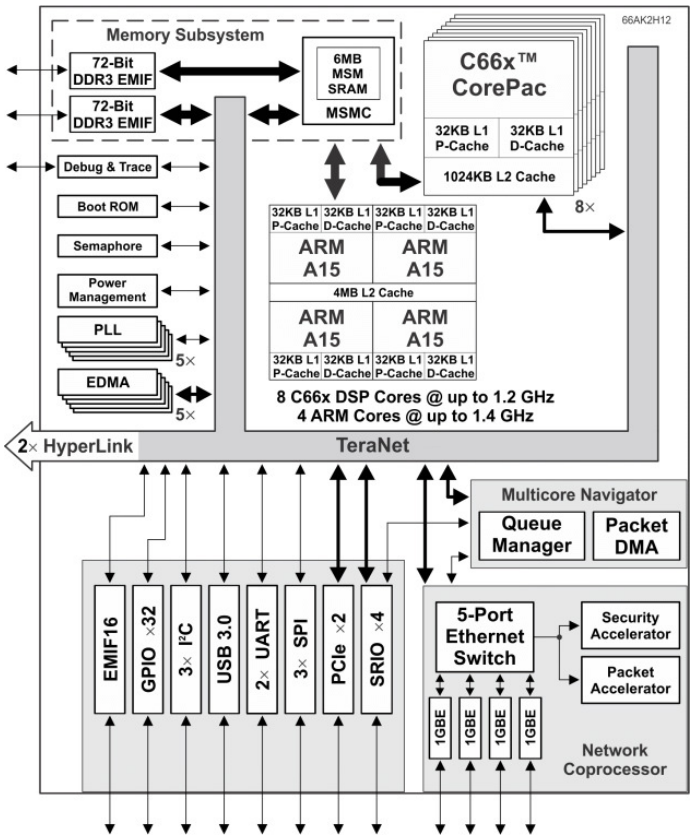


Figure 7.1: Texas Instruments KeyStone II SoC block diagram

7.2.1 C66x DSP Core

The main compute core inside the Keystone II architecture is the C66x DSP from Texas Instruments, which is based on a Very Long Instruction Word (VLIW) architecture. The core has two data paths, each capable of executing four instructions per cycle on four functional units named M, D, L and S. The M unit primarily performs multiplication operations, the D unit performs load/store and address calculations, and the L and S units perform addition and logical operations. Overall the two data

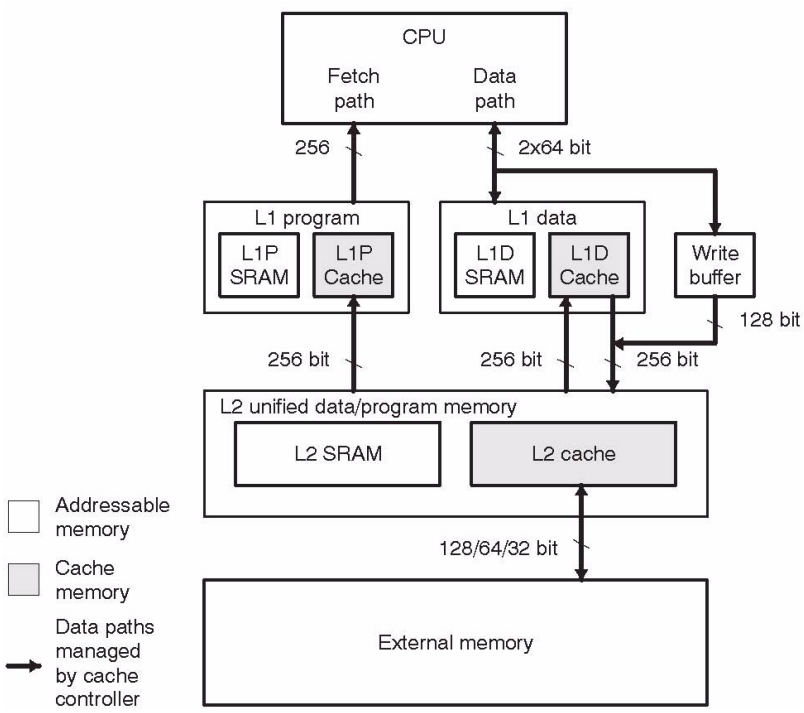


Figure 7.2: C66x cache memory block diagram

paths appear as an 8-way VLIW machine capable of executing up to eight instructions in each cycle. The instruction set also includes Single Instruction Multiple Data (SIMD) instructions allowing vector processing on up to 128-bit vectors. For example, the M unit can perform four single precision multiplies per cycles whereas each L and S unit can each perform two single precision additions per cycle. Together the two data-paths can issue 16 single FLOP per cycle. The double precision capability is about one-fourth of single precision FLOPs.

7.2.2 C66x DSP Memory Subsystem

The memory system is a Non-Uniform Memory Architecture (NUMA) [70]. A C66x subsystem can access different memory regions, with accesses to memories that are physically closer to a processor being faster. The memory regions (Figure 7.2) are as follows:

- Level-1 program (L1P) and data (L1D): 32KB, 1-cycle access time, configurable as mapped sRAM, cache, or a combination of mapped and cached.
- Local-L2: 1MB, configurable as mapped sRAM, cache, or a combination of mapped and cached, and shared between the L1D and L1P caches.
- Shared-L2: 6MB, shared memory on-chip, can be only configurable as sRAM.
- DDR: multiple 2GB of off-chip memory.

7.3 Challenges in Programming on Heterogeneous Multicore Embedded Systems

Multicore embedded systems usually consist of embedded CPUs, sensors and accelerators to provide high-performance but low-power solutions. Although these embedded systems offer high hardware capabilities, two challenges have started to emerge that require innovative approaches [82, 83].

The first challenge comes that the programming models and standards have not

kept pace with the increasing number/type of cores in a SoC. The limited availability of multicore software programming models and standards pose a challenge for their full adoption. Programmers typically have to write low-level codes, schedule task units, and manage synchronization explicitly between cores. As the hardware complexity is rapidly growing, it is nearly impossible to expect programmers to handle manually all the low-level details. This is not only time-consuming but an error-prone approach.

The second challenge is the software portability. The state-of-the-art for programming embedded systems includes proprietary vendor-specific software development toolchains which are tightly coupled with specific platforms. As a result, software developers have to significantly restructure the code if they want to port it to other platforms, while at the same time, ensure the performance. It leads to a less-productive and error-prone software development process that is unacceptable for the fast-growing complexity of embedded hardware. If the multicore embedded industry is to adopt quickly multicore embedded devices, one of the key factors to consider is to move from proprietary solutions to open standards.

7.4 OpenMP Accelerator Model

To address the programming challenges, the Texas Instruments KeyStone II SoC supports a subset of OpenMP 4.0 accelerator model. OpenMP is a high-level programming model for shared-memory parallel programming. OpenMP 4.0 extends its execution model to support heterogeneous accelerator-based architectures. The

OpenMP accelerator model assumes that a computation node has a host device connected with one or more accelerators as target devices. It uses a host-centric model in which a host device "offloads" code regions and data to accelerators for execution, specified using the `target` construct. This construct causes the data and the executable to be offloaded to the accelerators. So far the KeyStone II mainly supports the following OpenMP 4.0 features:

- **#pragma omp target**: specifies the region of code that should be offloaded for execution onto the target device.
- **#pragma omp declare target**: specifies functions mapped to a device.
- **#pragma omp target data**: creates a device data environment by mapping host buffers to the target for the extent of the associated region.
- **#pragma omp target update**: used to synchronize host or device buffers within a target data region as required.

7.4.1 A Code Example of OpenMP Accelerator Model

Figure 7.3 shows an code example of the classic "vector-add" operations using OpenMP 4.0 on the TI KeyStone II. In OpenMP 4.0, The `target` construct is used to specify the region of code that should be offloaded for execution onto the target device. The `map` clauses on the target construct specify data movement from host to device before execution of the offloaded region, and device to host after execution of the offloaded region.

```
1  int size = 1024;
2  float a[size];
3  float b[size];
4  float c[size];
5
6  #pragma omp target map(to:a[0:size],b[0:size],size) \
7                     map(from: c[0:size])
8  {
9      int i;
10     #pragma omp parallel for
11     for (i = 0; i < size; i++) {
12         c[i] = a[i] + b[i];
13     }
14 }
```

Figure 7.3: Vector-add using OpenMP 4.0

As is shown in the figure, the arrays `a, b, c` with the size of 1024 initially reside in host (ARM Linux) memory. Upon encountering a `target` construct, space is allocated in device memory for arrays `a[0:size], b[0:size]` and `c[0:size]`. Any variables annotated `'to'` are copied from host memory to device memory. The target region is executed on the device (DSPs). Note that the `#pragma omp parallel for` is used to distribute iterations of the for loop across the 8 DSP cores. Any variables annotated `'from'` are copied from device memory to host memory.

7.4.2 Texas Instruments-Specific Extensions to OpenMP 4.0

So far we have seen that OpenMP 4.0 provides a very simple approach for programming heterogeneous platforms. Programmers just use the `target` construct associated with the `map` clauses to offload the code and data to/from the host and target devices. However, there are still some extra efforts a programmer needs to handle to exploit the better performance. This section we introduce some additional

language features that Texas Instruments extended as a complement to the OpenMP 4.0 specification.

Reducing the Offload Overhead

As is shown in Figure 7.3, arrays which were initially allocated on host memory need to be transferred to the device memory on DSPs. Data synchronization between the host and target device can be a significant source of overhead. On the TI KeyStone II, although the host and target device share internal and external physical memory, the target device does not have a memory management unit (MMU); and there is no hardware cache coherency between the target and host device.

As a result, the host accesses shared memory using virtual addresses and the target accesses the shared memory using physical addresses. Moreover, host device variables can span multiple non-contiguous pages in Linux virtual memory whereas the target device operates on contiguous physical memory. When mapping variables from the Linux process space, the variables must be copied into contiguous memory for target operation. This copy is inefficient, especially for large variables. For the code example shown in Figure 7.3, the arrays `a[0:size]` and `b[0:size]` are firstly copied from host-only virtual memory to a contiguous memory space which is addressable by both of the host and target devices. At the end of the target region, the array `c[0:size]` will be copied back to the host-only memory region. As we can see, this data transfer scheme does not take advantage of the nature of the "shared-memory" on the platform.

```
1  /* Allocate buffer in device memory */
2  float* a = (float*) __malloc_ddr(size);
3  float* b = (float*) __malloc_ddr(size);
4  float* c = (float*) __malloc_ddr(size);
5
6  /* Initialize buffer on the host */
7  for (int i = 0; i < size; i++)
8  {
9      a[i] = 1.0;
10     b[i] = 1.0;
11     c[i] = 0.0;
12 }
13
14 /**
15  * Map to is a cache write-back operation on host.
16  * Map from is a cache invalidate operation on host.
17  * No copy performed on map to and from.
18  */
19 #pragma omp target map(to:a[0:size],b[0:size],size) \
20                    map(from: c[0:size])
21 {
22     int i;
23     #pragma omp parallel for
24     for (i = 0; i < size; i++) {
25         c[i] = a[i] + b[i];
26     }
27 }
28
29 /* Free buffer */
30 free_ddr(a);
31 free_ddr(b);
32 free_ddr(c);
```

Figure 7.4: Vector-add using TI-specific extensions for contiguous memory allocation

To eliminate this copy, TI provides a set of special purpose dynamic memory allocation APIs, namely `__malloc_ddr()`, `__malloc_msmc()`, `__free_ddr()`, and `__free_msmc()`, for dynamic memory allocation and free on the shared DDR and MSMC memory, respectively. The physical memory associated with this heap is contiguous and is mapped to a contiguous chunk of virtual memory on the host. If any host variables allocated via this API are mapped into target regions, the map clauses translate to cache management operations on the host, significantly reducing the overhead. Figure 7.4 shows the "vector-add" example using the TI extensions for physically contiguous memory allocation.

Local Data

For most of the general-purpose CPU programmers, most of the time they do not need to worry about the data placement in memory. They simply call the function `malloc()` to allocate the memory and the compiler and hardware will handle the cache and data locality for them. For embedded systems such as TI DSPs, on the other hand, programmers always have to manage the data placement manually in order to achieve the optimal performance. For example, for the code in Figure 7.4, the arrays are initially allocated in DDR memory. But what if we need to allocate them on local memory, such as L2 scratchpad memory?

To support this requirement, an additional `local` clause has been added by TI, which maps a variable to the L2 scratchpad memory. Variables allocated using the `local` clause have an undefined initial value on entry to the target region and any updates to the variable in the target region cannot be reflected back to the host.

```
1  /* Allocate a 1KB scratch buffer */
2  char* scratch_buffer = malloc(1024);
3
4  /**
5    * a[] is copied to the device
6    * scratch_buffer[] is allocated in L2 SRAM,
7    * scratch_buffer[] is not copied to the device
8    */
9  #pragma omp target map(tofrom: a[0:size]) \
10                    map(local: scratch_buffer[0:1024])
11 {
12   // Perform operations on buffer a[] in DDR using
13   // the L2 SRAM scratch_buffer
14   operate_on(a, scratch_buffer);
15 }
16 /**
17  * a[] is copied back to the Host,
18  * scratch_buffer[] is not copied back to host
19  */
```

Figure 7.5: A code example using the *local* extension.

Mapping host variables to target scratchpad memory provides significant performance improvements in many cases. On the TI KeyStone II, each DSP core's 1MB L2 memory can be configured as 128K cache, 768K scratchpad available to user programs and 128K reserved by the runtime. The 768K scratchpad is accessible via the local map type. Figure 7.5 shows a code example using the `local` clause which allocates a local buffer on the L2 scratchpad memory.

7.5 Memory Access Latency Measurements

The average data access latency in memory hierarchies is a critical metric which is used to measure the effectiveness of data locality optimizations. In this section, we measure the average memory access latency for both load and write operations.

The experimental results provide an important performance metric in evaluating the locality optimizations for sFFT.

7.5.1 Experimental Setup

The TI DSP compiler version used for the experiments is version 7.6.0. To turn the cache on/off, we utilize the TI Chip Support Library (CSL) [49]. The CSL provides a set of APIs used for configuring and controlling the DSP on-chip peripherals. The CSL is compatible with various TI C6000 series DSPs, which shortens the development time by providing standardization and portability.

7.5.2 EDMA Bandwidth Measurement

The EDMA engine was used to manage the data transfer, as it usually delivers higher transfer speed. So EDMA is a preferred way to move a large bulk of data between different levels of memory hierarchy. Since the EDMA transfers the data in an asynchronous way, an "EDMA wait" function is used to synchronize the events, which leads to waiting on a "barrier" until the EDMA finishes the data transfer.

To measure the EDMA bandwidth, we developed a micro-benchmark which moves contiguous chunks of data between different levels of memory and measured the memory throughput. The chunk size is chosen as 512 KB.

Table 7.1 shows the results. We measure the single channel EMDA bandwidth in Gbyte per second, one core at a time. As is shown in the table, the bandwidth

Table 7.1: Single channel EDMA bandwidth measured in GB/s

DSP Core	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
ddr → ddr	3.3	3.3	3.3	3.3	3.3	3.3	3.3	3.3
ddr → msmc	6.0	6.0	6.0	6.0	5.9	5.9	11.7	11.7
ddr → l2	6.0	6.0	6.0	6.0	5.9	5.9	6.1	6.1
msmc → ddr	5.3	5.3	5.3	5.3	5.0	5.0	7.2	7.2
msmc → msmc	6.2	6.2	6.2	6.2	6.2	6.2	12.2	12.2
msmc → l2	6.2	6.2	6.2	6.2	6.2	6.2	6.2	6.2
l2 → ddr	5.3	5.3	5.3	5.3	5.0	5.0	5.9	5.9
l2 → msmc	6.2	6.2	6.2	6.2	6.2	6.2	6.2	6.2
l2 → l2	5.8	5.8	5.8	5.8	6.2	6.2	5.8	5.8

between DDR, MSMC, and L2 memory is measured. Each measurement performs on a single DSP core from core 0 to core 7. The data shows an empirical peak bandwidth we can achieve using EDMA to transfer a block of contiguous data on a single DSP core. Also, note that core 6 and 7 deliver a higher bandwidth. That is because the DSP core (0-5) is assigned to an EDMA channel controller from 1 to 3. Core 6 and 7 use the EDMA controller 4, which gives higher bandwidth.

7.5.3 Memory Copy Bandwidth Measurement

Typically, there are two approaches of moving a block of data between different levels of the memory hierarchy: asynchronous and synchronous methods. The asynchronous approach transfers the data asynchronously to the CPU calculations. So the CPU does not need to stall and wait for the data to be ready. Therefore, it does not cost the CPU cycles to load and store the data. As we described in the last subsection, the asynchronous data transfer is performed by using EDMA.

In this subsection, we measure the synchronous data movement bandwidth. In

Table 7.2: Single core memory copy bandwidth measured in GB/s

DSP Core	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
ddr → ddr	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
ddr → msmc	1.5	1.5	1.5	1.5	1.5	1.5	1.5	1.5
ddr → l2	1.6	1.6	1.6	1.6	1.6	1.6	1.6	1.6
msmc → ddr	1.7	1.7	1.7	1.7	1.7	1.7	1.7	1.7
msmc → msmc	2.5	2.5	2.5	2.5	2.5	2.5	2.5	2.5
msmc → l2	2.5	2.5	2.5	2.5	2.5	2.5	2.5	2.5
l2 → ddr	1.7	1.7	1.7	1.7	1.7	1.7	1.7	1.7
l2 → msmc	2.6	2.6	2.6	2.6	2.6	2.6	2.6	2.6
l2 → l2	3.1	3.1	3.1	3.1	3.1	3.1	3.1	3.1

this approach, the CPU gets involved into the data transfer to load/store each data element. To achieve this, we also developed a micro-benchmark by employ the `memcpy()` function to copy the data. Since we configure both the L2 and MSMC memory as the sRAM, the data is not cacheable. Table 7.2 shows the experimental results.

Compared to the EDMA data transfer rate shown in Table 7.1, the memory bandwidth on synchronous movement is much lower. That is because the CPUs need to get involved into the data movement.

7.5.4 Memory Access Latency on L1 Cache

L1 Load Test

A micro-benchmark is developed to test the memory access latency on the L1 cache. Figure 7.6 shows the kernel of the code for the L1 load test. Note that in Figure 7.6, the variable `c` is intentionally defined out of the function to avoid the dead

Table 7.3: L1 load test results (in CPU cycles)

Hardware prefetching	**ON**	**OFF**
L1 load (warm cache)	1.13	1.13

coding issue caused by the TI compiler. Also, we use the compiler hint `#pragma MUST_ITERATE(1)` immediately before the loop to instruct the compiler not to check the zero-trip loop to avoid the additional overhead on branch.

```
1 #pragma MUST_ITERATE(1)
2 #pragma UNROLL(1)
3 for (int i = 0; i < sz; i++) {
4   c += src[i];
5 }
```

Figure 7.6: L1 load test

Moreover, we use another compiler hint, `#pragma UNROLL(1)` to tell the compiler not to unroll the loop. So in this case, each loop iteration takes only one CPU cycle, which is called "single-cycle loop". Note that the loop has both load and add operations, but it still takes 1 CPU cycle because of the software-pipelined loops. That is, the accumulation of the `src[i]` will execute with loading the next element `src[i+1]` simultaneously. Also, the array `src` is pre-loaded into the L1 cache, so the L1 is warm. We test the L1 load performance for the hardware prefetching is both turned on and off, respectively.

Table 7.3 shows the results of the L1 load test. It can be seen from the table that loading from L1 takes around 1 CPU cycle, which is as expected. We can also observe from the table that hardware prefetch did not make any difference on the result because the cache is already warm. Noted that the number is a little larger

Table 7.4: L1 store test results (in CPU cycles)

	Hard prefetching ON/OFF	
L1 Store	**Cold cache**	**Warm cache**
	1.17	1.06

than one because the loop is not perfectly software-pipelined, i.e., the load and add operations may not be entirely overlapped.

L1 Store Test

In this subsection, we test the CPU cycles for storing data into the L1 cache, while the L1 is cold and warm, respectively. To measure the L1 'cold' cache, we first turn off the L1 cache. Then preload data into L2 and turn the L1 on again. So now L2 is 'warm' but L1 is still 'cold'. Then we run the L1 'cold' store test.

According to the DSP Cache Guide [51], the L1 cache is read-allocate only. So in the code for the L1 store test, once there is a cache miss, it will store the data into the L2 cache via the *write buffer* (see the Figure 7.2). In the 'warm' store test, since data has been pre-loaded to the L1 cache before, there will be no cache misses and all data is stored at the L1D cache.

Table 7.4 shows the results for L1 store test. When the cache is cold, according to the DSP cache guide, L1D is a read allocate cache, meaning that a cache line is allocated on a read miss only. For a write miss, the data is written to the lower-level memory through a write buffer, bypassing the L1D cache. Therefore, it indicates that the data is stored in L2 cache via the write buffer. In other words, the 'cold' store test tests the CPU cycles for storing a cache line into the L2 cache via the write

buffer. The 'warm' cache test is easier to understand because it tests when the store hits at L1D. From the results, we can see that the CPU cycles taking on 'warm' store are less than the 'cold' store. That is because in the 'cold' store test, data has to go to the L2 cache and a new line needs to allocate in the L2. For the warm test, on the other hand, no new cache line allocation is needed. Furthermore, noted that the prefetch did not make any difference for both cold and warm test because L1 is read-allocate only, meaning it will not allocate any new cache line on L1 in the store test.

7.5.5 Memory Access Latency on L2, MSMC and DDR Memory

L2 Load Test

We develop a micro-benchmark for the L2 load and store test. It tests when data is initially in DDR and MSMC, respectively. Table 7.5 shows the results. Note that in the table, 1/1 annotates that one data entry loaded and the stride is 1. On/On/DDR means the L1 cache is turned on, the L2 is on, and data is initially in the DDR memory.

From the Table 7.5, we could see that the column of Off/On/DDR is the case for testing the memory access latency on L2. That is because L1 is turned off while L2 is on, and the data entry is pre-loaded into the L2, which is warm. It can be seen from the table that the memory access latency for L2 load test is 9 CPU cycles. Noted that in Table 7.5 that when `N = 2` and the stride is 16 and 32, the results are

Table 7.5: L2, MSMC and DDR load test results in CPU cycles

N/Stride, L1/L2/DDR or MSMC	**On / On / DDR**	**Off / On / DDR**	**Off / On / MSMC**	**Off / Off / DDR**	**Off / Off / MSMC**
1/1	1	9	24	100	24
1/16	1	9	24	100	24
1/32	1	9	24	100	24
2/1	1	9	24	100	24
2/16	1	6	13	60.5	13
2/32	1	6.5	13	60.5	13

Table 7.6: Strided load terst

N/Stride, L1/L2/DDR or MSMC	**Cold / Warm / DDR**	**Cold / Warm / MSMC**	**Cold / Cold / DDR**
2/1	13.5	15.5	61.5
2/16	16.5	23	68
2/32	16.5	23	68

reduced to around half. That is because the instructions of two parallel loads are issued in parallel.

We also experiment on the strided load test, of which the results are shown in Table 7.6. The experiments show results when the L1 and L2 are turned on, while L2 is warm and cold, respectively. The first and second column in the table are tests when L2 is warm. To achieve this, we first turned off L1D, pre-loaded data into L2, and turned L1D on again. The third column showed when L2 is cold and data is in DDR. To do this, we invalidate both of the L1 and L2 in each test.

MSMC Load Test

According to the Table 7.5, MSMC load test is for N/stride equals to 1/1, 1/16, 1/32, L1 is off while L2 is on, data in MSMC. That is because L1 is turned off and MSMC is not cached through L2 by default. From the table, the CPU cycles on MSMC load test is 24 cycles.

Similarly, we tested for n/stride equals to 1/1, 1/16, and 1/32, both L1 and L2 are off, but data in MSMC (Column 6). It also tested the CPU cycles loading from MSMC. The results should be the same as L1 is off and data in MSMC (column 4). It is because MSMC is not L2 cacheable by default. And the results confirm the hypothesis which equals to 24 cycles.

For n/stride equals to 2/1, 2/16, 2/32, L1 is off while L2 is on, data in MSMC. We noted CPU cycles while loading from MSMC but loading two words in parallel. For n/stride equals to the 2/1, the result is 24, which is the same as n/stride equals to 1/1.

DDR Load Test

To test the CPU cycles on loading data from DDR memory, we turn both L1 and L2 off and keep the data in DDR initially, as is shown in Table 7.5 (Column 5). From the table, we could see that the CPU cycles on DDR load test are 100.

Table 7.7: CPU store test on L2, MSMC and DDR

L2		MSMC		DDR	
Cold	**Warm**	**Cold**	**Warm**	**Cold**	**Warm**
1.77	1.00	1.00	1.00	1.00	1.00

L2, MSMC and DDR Store Test

Table 7.7 shows the results of the CPU store test. For all the tests, stride equals to 1 and n = 3200 to compensate the loop overhead. For L2 and MSMC tests, L1D is turned off while L2 is invalidated before each test. For DDR test, both L1D and L2 are turned off.

From the result, store to L2 takes 1.77 cycles on the cold cache and 1.00 on the warm cache due to the effects of the write buffer. When L2 is cold, for write requests to DDR misses on L1D (which is turned off), it is passed on to L2 through the write buffer. If L2 detects a miss for this address, the corresponding L2 cache line fetches from external memory. So the L2 test measured the write buffer cycles, which is closed to 1 cycle. The result is consistent with L1 store test when the cache is cold. Furthermore, the cold cache takes more CPU cycles than the warm cache because of the additional cycles on fetching a line from DDR memory.

For the store test on to MSMC memory: The result for both warm and cold cache is 1 cycle, which is close to L1D results. It is because the write buffer can only bypass L1D and write results to L2. Also, MSMC is not L2-cacheable by default. As a result, when L1 is turned off, the CPU cycles should be equal to writing data to MSMC, not to L1D.

Table 7.8: Interference of write-on-read

Regular (PF ON)		Regular (PF OFF)		Irregular	
L1 SRAM	**DDR**	**L1 SRAM**	**DDR**	**L1 SRAM**	**DDR**
10.4	12.0	5.3	6.1	143.4	145.7

For the store test to DDR memory. Since the L1D and L2 are disabled, the results should be equal to storing in DDR. Again, the write buffer should not matter since now L1 and L2 are disabled.

In summary, the results in Table 7.7 indicate that all write operations take only 1 CPU cycle. That is because the C66x CorePac has improved write merging and optimized burst sizes that reduce the stalls to external memory. It can also be seen from the table that when there is a write miss on L1 cache, it takes 1.17 cycles, which is slightly longer than others. It is because L1D is the *read-allocate* only, meaning that a cache line is allocated only when a read miss occurs. On a write miss, the data is written to the lower-level memory through a *write buffer*, bypassing L1D cache (see Figure 7.2). The write buffer consists of 4 entries. On C66x devices, each entry is 128-bits wide [51].

7.5.6 Interference of Write-on-Read

In the sFFT inner loop, the next load of `orgix[index]` may not get started until the last write of `x_sampt[i]` instruction finishes. To investigate this, the array `x_sampt[]` is put into both L1 SRAM and DDR, respectively. We put the array into L1 SRAM since this will cause the next load to never stall. Table 7.8 shows the results. From the result we can see that putting `x_sampt` into L1 SRAM saves around 2 cycles per

loop iteration, indicating store will not stall the load.

7.6 Parallel Sparse FFT on TI KeyStone II

In this section, we present our approach to port the parallel sFFT on the Texas Instruments KsyStone II SoC. According to the time distribution we measured in Section 4.4, the stage *perm+filter* dominates the overall execution time. Therefore, we only port the *perm+filter* stage onto DSPs while remaining the rest of functional stages on the ARMs.

Figure 7.7 shows the code snippet of the *perm+filter* step on theDSPs. We can see from the code that we use the OpenMP 4.0 `target` construct and `map` clause to port this stage onto DSPs. Specifically, the code region inside of the `target` construct will execute on DSPs. The `map` clause transfers the data between ARMs and DSPs. The `parallel` region inside of the `target` is precisely the same as the PsFFT on mulcore CPUs, but executes on the DSPs. It indicates the major advantage of using OpenMP that programmers can largely preserve the code structure among different possible architectures. Note that the code in Figure 7.7 is a naive version without any in-depth performance optimizations. Section 7.7 will discuss multiple performance optimization techniques for the sFFT on DSPs.

```
1  void inner_loop_perm_filter_dsp(complex_t *origx, int n,
2                                  complex_t *filter,
3                                  int filter_size,
4                                  complex_t *x_sampt, int B,
5                                  int ai)
6  {
7    #pragma omp target map(to: origx[0:n], \
8                               filter[0:filter_size], \
9                               n, B, ai, filter_size) \
10                        map(from: x_sampt[0:B])
11   {
12     int i, j;
13     int round = filter_size / B;
14
15     // Buckets initialized to 0
16     #pragma omp parallel for
17     for (i = 0; i < B; i++) {
18       x_sampt[i].re = 0.;
19       x_sampt[i].im = 0.;
20     }
21
22
23     // Main body of the sFFT permutation and filter
24     for (i = 0; i < round; i++) {
25       int off = i * B;
26       #pragma omp parallel for
27       for (j = 0; j < B; j++) {
28         int off2 = off + j;
29         unsigned index = (unsigned) (off2 * ai) % n;
30         cmpy_acc(filter[off2],
31                  origx[index],
32                  x_sampt[off2 % B],
33                  x_sampt[off2 % B]);
34       }
35     }
36   }
37 }
```

Figure 7.7: The *perm+filter* stage on DSPs using OpenMP 4.0

7.7 Performance Optimizations

In this section, we present a set of performance optimization techniques on the TI KeyStone II platform. Those optimization techniques aim to reduce the data transfers between the ARMs and DSPs, improve multiple levels of parallelism, and enhance the data locality for sFFT.

7.7.1 Optimization of the data transfer between the ARM and DSP

As introduced in Secion 7.4.2 about the TI specific extensions to the OpenMP 4.0, the main DDR memory on the TI KeyStone II is physically shared between the host ARM CPUs and the target DSP accelerators. Logically, however, it is partitioned into two regions, one for ARM Linux and the other is a physically contiguous memory shared by the ARMs and DSPs. As a result, if one allocates a memory region on the ARM using e.g., `malloc()`, that memory will be allocated on the heap memory on the ARMs and can be only addressable by the ARMs. To make this data region accessible by both the ARMs and DSPs, we can use the OpenMP 4.0 `map()` clause to copy the data from the heap to the shared contiguous memory region. The data copy between the ARMs and DSPs memory is, unfortunately, very expensive.

In order to address this issue, we use the TI-specific extensions to allocate and free contiguous segments of memory that can be accessed by both of the ARMs and DSPs cores. Specifically, we use the `__malloc_ddr()` and `__free_ddr()` routines to

allocate and free the shared memory.

7.7.2 Hardware Prefetching

Hardware prefetching is an important performance improvement feature of todays processors. It has been shown to improve performance by 10-30% without requiring any programmer effort. Prefetching is the concept of fetching data closer to the processor before it is needed. The intent is that the data will be readily available when the processor needs it. Without prefetching, the processor will have to stall and wait for the data.

Hardware may improve the performance of regular applications as it can reduce the number of capacity misses in a cache. For irregular applications, however, hardware prefetching may saturate memory bandwidth as the irregular memory access pattern is dynamic thus hardly predictable. Hence, we turn off the feature of the hardware prefetching on the TI KeyStone II for the sFFT.

To evaluate the impacts of the hardware prefetching, we develop a benchmark based on the "vector add" operations, i.e., adding the elements of two arrays, `a[i]` and `b[i]`, and store the results back to the array `c[i]`. This way, accessing each element of the arrays is in sequential order and regular. We expect the performance decrease if we turn off the hardware prefetching.

Table 7.9 shows the results of the vector addition when the hardware prefetching is on and off, respectively. As expected, the execution time of "VecAdd" is increased from 0.05 sec to 0.07 sec after the hardware prefetching is turned off.

Table 7.9: Effects of hardware prefetching (in second)

	Hardware prefetching ON		**Hardware prefetching OFF**	
	VecAdd	**Strided VecAdd**	**VecAdd**	**Strided VecAdd**
Core 0	0.05	0.70	0.07	0.67
Core 1	0.05	0.70	0.07	0.68
Core 2	0.05	0.70	0.07	0.67
Core 3	0.05	0.70	0.07	0.67
Core 4	0.05	0.70	0.07	0.67
Core 5	0.05	0.70	0.07	0.68
Core 6	0.05	0.70	0.07	0.67
Core 7	0.05	0.70	0.07	0.67

Furthermore, we developed another benchmark named "Strided VecAdd" in the form of `c[i] = a[index] + b[i]`, where the array `a` is accessed at random as the `index` is generated randomly at runtime. The purpose is to intentionally "interfere" the hardware prefetcher and make each prefetching a wrong prediction.

Table 7.9 shows the performance ("Strided VecAdd"). From the table, quite suprisingly, it does not make a significant difference when the prefetching is on and off. That is because TI memory controller is "smart" enough to keep track of the memory access pattern and the hardware prefetching will be kicked in only when it observes a stream of contiguous memory reference. Consequently, in this example of strided vector addition, the hardware prefetcher is not triggered so there is no much performance difference. From the table, the performance is slightly better when the prefetching is turned off. That is because the extra overhead on memory controller is not added while the hardware prefetching is turned off. For the rest of the experiments, we leave the hardware prefetching on as it does not make much difference to the performance.

Table 7.10: Clock cycles per loop iteration for various block sizes in L1 SRAM

Block size	**16**	**32**	**64**	**128**	**256**	**512**	**1024**
L1 OFF	1465.1	776.1	429.2	255.8	170.5	127.0	104.4
L1 ON	98.3	90.0	89.2	87.9	87.5	87.1	87.1

7.7.3 Packing Optimizations

The purpose of the packing optimization is to improve the locality for irregular memory accesses in the sFFT. The basic idea is to split the original irregular memory accesses into two steps. In the first step, it "packs" a block of data by gathering it from DDR and stores it into the on-chip local memory (L1/L2 sRAM). Then, in the second step, the original sFFT inner loop executes but loads the "packed" data. The benefits of the packing optimization highly depend on many factors, including the block size, memory hierarchy to store the block and whether the cache is turned off during the packing process.

We conducted several experiments with different configurations to find out the optimal combination of block size and its location. Table 7.10 shows the clock cycles per loop iteration with various block sizes; a block is stored in the L1 SRAM. We also measure the results when the L1 is turned on and off, respectively, during the packing process.

We can observe from the table that the clock cycles per loop iteration are much smaller if the cache is turned on during the packing process. The primary purpose of turning off the cache during the packing process is to avoid the extra cache misses. However, the result shows that turning cache off incurs a substantial overhead.

Table 7.11: Clock cycles per loop iteration for the block in L2 SRAM

Block size	**16**	**32**	**64**	**128**	**256**
L1/L2 OFF	1471.6	781.5	433.0	259.6	175.2
L1/L2 ON	106.1	91.3	90.6	91.5	94.3
Block size	**512**	**1024**	**2048**	**4096**	**8192**
L1/L2 OFF	131.8	108.7	97.6	92.1	89.2
L1/L2 ON	94.3	93.1	91.5	91.4	91.4

Table 7.11 shows the results when the local block is in L2 SRAM. In can be seen from the table that the clock cycles per loop iteration are slightly greater than the block in L1 SRAM. That is because the L2 SRAM has larger memory access latency. Nevertheless, the L2 can store more blocks since the size of L2 is far greater than L1. As a result, when the block size is equal to 8192, the cycles when the cache is turned off is 89.2 whereas it is 91.4 when the cache is ON. This shows the benefits when the cache is turned off.

7.7.4 Blocking Optimization

The blocking optimization is based on the packing optimization discussed previously, which intends to address the following issues:

- Cache capacity (i.e., working set size with respect to cache),
- cache conflicts should be virtually eliminated,
- cold start misses (via packing loop),
- and false sharing in case of parallelization.

There are several expected benefits of this optimization strategy. First of all, global network traffic is significantly reduced. When this loop is parallelized, at most one thread will read or write array data that map to the same cache line.

Secondly, each memory location in DDR is read at most once. If L1 SRAM is data is cacheable in L2, then virtually all cache lines will be written back to memory exactly once. Otherwise, multiple instructions will be needed to write out a single cache line, but this will be the minimum supported by the system.

Thirdly, each thread finishes reading/writing a given cache line before accessing another cache line from the same array. Fourthly, footprint in the cache is slight (and independent of problem size).

Last but not least, the blocking optimization reduces of memory access latency (cold start misses) to `origx` which accounts for the bulk of memory system latency. This is accomplished by fetching two distinct cache lines in parallel which is the only way to hide misses to arrays with large non-power of two strides. This was accomplished by "coaxing" the compiler without needing to resort to assembly code. The compiler does not automatically attempt this.

This parallelization of memory loads was intentionally not done for `origx[]` and `x_sampt[]`. At last measurement, due to high spatial and temporal locality, loads of these arrays were taking too few cycles on average to justify extra CPU cycles to optimize for this. It would also add additional complexity to the code which could not be justified in the writing of this dissertation.

However, there are several expected drawbacks for this optimization strategy.

Table 7.12: Clock cycles per loop iteration different optimization strategies

Original	**Optimized**	**Blocking**
241.4	154.9	91.7

First of all, compared to the original version, it no longer exploits hardware prefetching of the `origx` array because cache lines of the `origx` are not accessed consecutively. This loss is expected to be at most a few cycles/per result (result = original loop iteration). Secondly, it incurs CPU cycle overhead compared to other serial versions of about 3-4 cycles/result (result = original loop iteration). Both of these drawbacks are supposed to be minor compared to the expected benefits of best-possible, scalable, parallel performance.

Table 7.12 shows results for the "original" version i.e., the naive version without applying any specific optimization. The optimized version uses software pipelining, elimination of loop-carried dependency, balancing resources and enabling SIMD instruction. The blocking version is what we have discussed above.

7.7.5 Miscellaneous Optimizations

Exploiting Software-Pipelining

Many application routines are loop-centric, including irregular applications. In the sFFT, the inner loop takes more than 70% of the total execution time [80]. Identifying the most time-consuming loop and speeding the looped code is the key to the overall performance improvements. A technique called *software pipelining* [34] contributes the biggest boost to improving looped code performance.

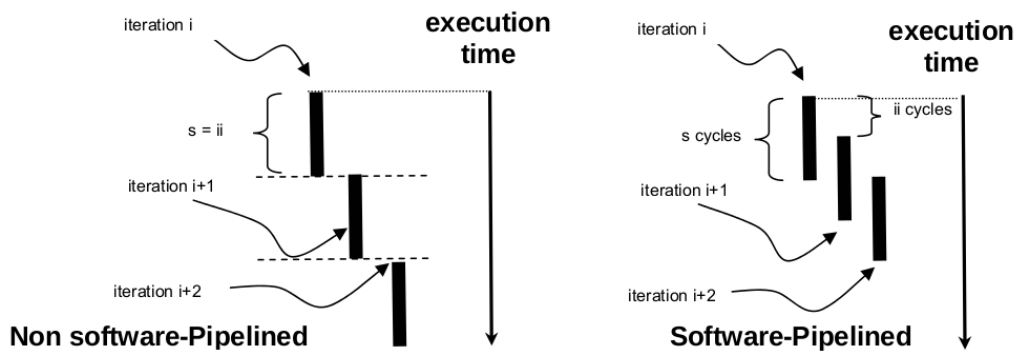


Figure 7.8: Software pipelined loops [34]

Figure 7.8 shows the software pipelined loops. Without software pipelining, loops are scheduled so that loop iteration `i` completes before iteration `i+1` begins. Software pipelining allows iterations to be overlapped. Thus, as long as correctness can be preserved, iteration `i+1` can start before iteration `i` finishes. It leads to a much higher utilization of the machines resources than might be achieved from non-software-pipelined scheduling techniques.

As is shown in the Figure 7.8, in a software-pipelined loop, even though a single loop iteration might take `s` cycles to complete, a new iteration is initiated every `ii` cycles. In an efficient software-pipelined loop, where `ii < s`, `ii` is called the *initiation interval*; it is the number of cycles between starting iteration `i` and iteration `i + 1`. `ii` is equivalent to the cycle count for the software-pipelined loop body. `s` is the number of cycles for the first iteration to complete, or equivalently, the length of a single scheduled iteration of the software-pipelined loop. Consequently, we can see that software pipelining can potentially improve the performance as multiple iterations of the loop can execute in parallel. To make the best of the performance, the pipeline should be utilized as full as possible.

Eliminating Loop-Carried Dependency

The main goal in loop optimization is to minimize the iteration interval (`ii`). The iteration interval is the number of cycles it takes the CPU to complete one iteration of the parallel representation of the loop code. The overall cycle count of the software pipelined loop can be approximated with `ii * number_of_iterations`. The value of `ii` is bounded below by two factors: the loop-carried dependency bound and the partitioned resource bound, determined by the compiler.

The loop-carried dependency bound is defined as the distance of the largest loop carry path, and a loop carry path occurs when one iteration of a loop produces a value that is used in a future iteration. The loop-carried dependency bound can be visualized with a dependency graph, which is a graphical representation of the data dependency between each instruction. Figure 7.9 shows the simplified, corresponding dependency graph (i.e., branch related code removed). The nodes in the graph are the instructions executed in a loop iteration. The edges (arrows between node pairs) denote the ordering constraints. The edges are annotated with the number of cycles needed between the source and destination instructions. In most cases, results are written to registers at the end of the cycle in which the instruction executes and available on the following cycle.

In this graph, there are two critical cycles, each with length 7. To reduce the loop-carried dependency bound, the largest cycle in the graph must be shortened or eliminated. This can be accomplished by eliminating one of the edges in the cycle. To do so, one must understand the origin of the edges.

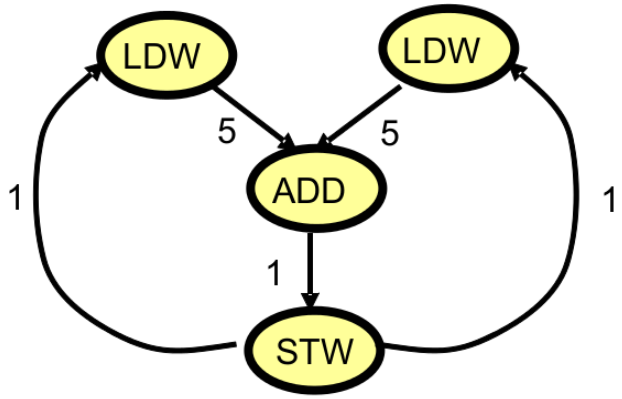


Figure 7.9: Loop-Carried dependency cycles

To the compiler, the address of the data pointed to load and store are unknown at compile time; it has to prepare for all possible outcomes during compilation. In the worst case where memory access patterns are dynamically irregular, as the dependency graph shows, load (`i+1`) depends on store (`i`), which in turn depends on add (`i`), which depends on load (`i`). In other words, iteration (`i+1`) is dependent on iteration (`i`), and it cannot be started until 7 cycles after iteration (`i`) is started. Therefore, the loop carried dependency bound for the loop is 7. However, if the load and store will never point to the same memory buffer, there should not be any loop-carried dependency. In this case, the *restrict* keyword can be used to inform the compiler that the pointers in a given function will never point to the same memory buffer [89]. This can avoid the unnecessary high loop-carried dependency that the compiler might conclude.

Exploiting SIMD Instructions

The TI C66x DSP improved the SIMD capability (each instruction can process multiple data in parallel) combined with the natural instruction level parallelism of C6000 architecture (e.g., execution of up to 8 instructions per cycle) results in a very high level of instruction-level parallelism. C66x ISAs can now execute instructions that operate on 128-bit vectors [50].

Two of the most frequently used SIMD instructions are the LDDW and STDW instructions, which perform aligned 64-bit load and 64-bit store, respectively. Assuming all the data used are 16-bit data (short), it would be ideal if the LDDW and STDW instructions can operate on 4 elements every time. If the data is declared as 32-bit type (int, float), LDDW and STDW can operate on 2 elements every time.

Exploiting the SIMD instructions is an essential optimization for irregular applications. It is simply because loading/storing multiple words in a single instruction rather than of multiple short loads/store instructions can significantly reduce the memory traffic and efficiently improve the memory bandwidth.

Figure 7.10 shows the sFFT inner loop optimized by using SIMD instructions. The `_amemd8(void *ptr)` allows aligned loads and stores of 8 bytes to memory. Note that when performing SIMD operations on C66x, the `__x128_t` type is often used, which is a container type for storing 128-bit of data.

```
1  /** sFFT complex number multiplication
2   * using SIMD instructions
3   * c1 = c2 + c3 * c4
4   */
5  #define cmpy_acc(c1, c2, c3, c4)
6  {
7    lddw_re_im(&c1, c1_re, c1_im);
8    lddw_re_im(&c2, c2_re, c2_im);
9    lddw_re_im(&c3, c3_re, c3_im);
10
11   c4_re = c1_re * c2_re - c1_im * c2_im + c3_re;
12   c4_im = c1_re * c2_im + c1_im * c2_re + c3_im;
13
14   stdw_re_im(&c4, c4_re, c4_im);
15 }
16
17 /** 128 bit types are double-word aligned.
18  *  Use aligned loads
19  */
20 #define lddw_re_im(b2, re, im)
21 {
22   (re) = _amemd8((void *) (&((b2).re)));
23   (im) = _amemd8((void *) (&((b2).im)));
24 }
25
26 /** 128 bit types are double-word aligned.
27  *  Use aligned stores
28  */
29 #define stdw_re_im(b2, re, im)
30 {
31   _amemd8((void *) (&(b2.re))) = re;
32   _amemd8((void *) (&(b2.im))) = im;
33 }
```

Figure 7.10: SIMD instructions in sFFT inner loop

Loop Unrolling and Resource Balancing

As introduced in the architectural overview, the C6x DSP architecture is partitioned into two nearly symmetric halves (A-side and B-side) with limited connectivity between the two. Each side contains half the registers and four functional units, noted by the types of instructions that execute on multiplication, load and store, shifts, branches, logical and arithmetic operations. Consequently, it is a key to evenly partition the computations between the two halves and balance the available resources. For example, if 8 multiply instructions need to be completed every iteration, each `.M` unit (multiplication) needs to complete 4 multiply instructions per iteration (since there are two `.M` units). Therefore, the next iteration, regardless of whether software pipelining is used or not, will not be able to start until at least 4 cycles later.

In irregular applications, the irregular memory accesses can easily cause the unbalanced resource partition, which significantly burdens the traffic of the cross path between the two halves. If the resource partition information indicates an unbalanced partition between the A side and B side functional units, additional techniques, such as loop unrolling, can be used to balance the usage of the two halves of the core.

Figure 7.11 shows the concept of unrolling the loop for better (more balanced) utilization of the critical `.D` unit (for load and store) resource (the situation would be analogous for the critical `.T` address path). On the left side, four loop iterations are shown, as indicated by the double arrows, producing eight results in four cycles. One `.D` unit is unused every other cycle. The right side shows performance after unrolling the loop by 2x. Both `.D` units are therefore executing useful instructions

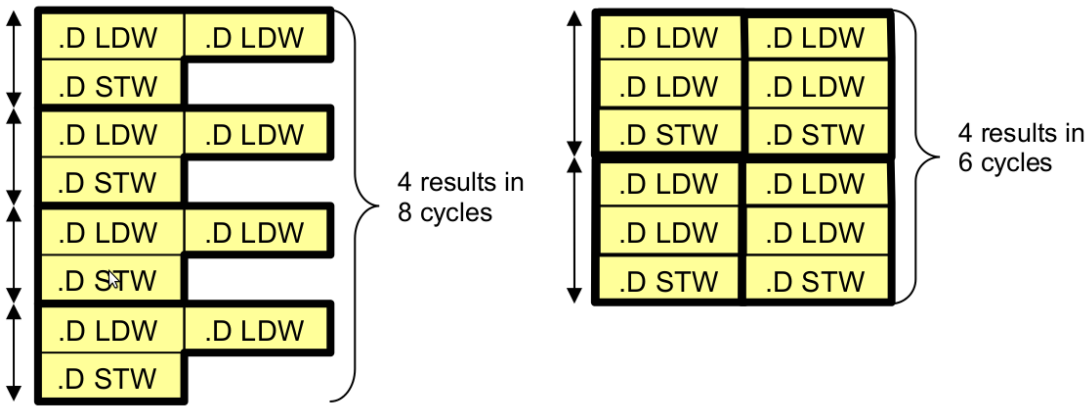


Figure 7.11: Loop unrolling and resource balancing.

in every cycle and the CPU cycles are further reduced to 6 to produce the 4 results.

Unrolling the loop manually, however, is usually tedious and error-prone, even for quite simple loops. For the TI's compiler, one can pass hints to the compiler via directives and instruct the compiler to unroll the loop automatically. This is done by using the `MUST_ITERATE` pragma: `MUST_ITERATE(lower_bound, upper_bound, factor)`. It specifies to the compiler certain properties of the loop count. The `lower_bound` defines the minimum possible total iterations of the loop, the `upper_bound` defines the maximum possible total iterations of the loop, and the factor tells the compiler that the total iteration is always an integer multiple of this number. The `lower_bound` and `upper_bound` must be divisible and at least as big as the factor. Any of the three parameters can be omitted if unknown, but if any of the value is known, it is recommended to provide this information to the compiler. For instance, `#pragma MUST_ITERATE(1,,2)` instructs the compiler that the loop trip count is at least 1, and is a factor of 2. So the compiler is most likely to unroll the loop by the factor of 2.

7.8 Performance Evaluation

In this section, we evaluate the performance of the sFFT on TI KeyStone II platform. Currently, we developed five versions of the sFFT inner loop (i.e., *perm+filter* stage) on DSPs as below:

- DSP Version 1 – Original baseline version without any optimization
- DSP Version 2 – Optimized version after applying the above mentioned low-level optimizations (without SIMD optimization)
- DSP Version 3 – Sequential blocking version with SIMD optimization
- DSP Version 4 – Parallel blocking version with SIMD optimization
- DSP Version 5 – Porting Intel parallel blocking version (As described in Chapter 5) to DSPs

For the performance evaluation purpose, we also developed three different versions of the sFFT inner loop for the ARM architecture.

- ARM Version 1 – Original baseline version without any optimization
- ARM Version 2 – Intel parallel blocking version described in Chapter 5
- ARM Version 3 – Parallel blocking version for ARM. A newly developed version. Took DSP Version 4 as the code base

Note that the Intel code (i.e., PsFFT) can be directly ported to the ARM architecture without any modifications to the sFFT source code.

Table 7.13: sFFT time distribution on ARMs ($k = 1000$)

Signal size (2^n)	**18**	**19**	**20**	**21**	**22**	**23**
Original MIT sFFT C++ version						
perm+filter	0.177	0.448	0.784	1.277	1.966	3.021
overall	0.756	0.941	1.406	1.761	2.463	3.521
percentage	23.3%	47.6%	55.8%	72.5%	79.8%	85.8%
Sequential baseline C version						
perm+filter	0.167	0.481	0.914	1.479	2.196	3.445
overall	0.422	0.692	1.164	1.700	2.445	3.711
percentage	39.6%	69.5%	78.5%	87.0%	89.8%	92.8%
Parallel baseline C version (4 threads)						
perm+filter	0.1855	0.4812	0.9131	1.4843	2.2004	3.4387
overall	0.2732	0.5791	1.0103	1.5791	2.3081	3.5636
percentage	67.9%	83.1%	90.4%	94.0%	95.3%	96.5%

7.8.1 sFFT Time Distribution

As a very first experiment, we executed the baseline version of the sFFT on the ARM and measured the execution time. The objective is to justify the time distribution on the stage of the *perm+filter* domains the overall execution time.

Table 7.13 shows 3 groups of data on ARM for original MIT C++, our C baseline sequential implementation (described in Chapter 4), and our C baseline parallel implementation (perm+filter is still sequential while the rest stages are in parallel). The purpose is to justify that the perm+filter stage takes most of the execution time. From the table, the third group shows that in the parallel baseline version, perm+filter stage takes most of the time. The stages other than the perm+filter are parallelized on the ARM. This serves as the baseline version for the sFFT ARM + DSP code. Noted that perm+filter is the stage that is executed on the DSP while the rest of them are executed on ARM, either sequential or in parallel.

Table 7.14: Execution time (sec) of DSP Version 1

Signal size 2^n	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**	**cycles**
perm + filter	0.67	0.93	1.28	1.73	2.42	3.43	4.57	6.39	**252**

Table 7.15: Execution time (sec) of DSP Version 2

Signal size 2^n	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**	**cycles**
perm + filter	0.40	0.59	0.84	1.12	1.57	2.24	2.98	4.16	**165**

For the results shown in rest of this chapter, otherwise specified, the execution time refers to the time on *perm+filter* only.

7.8.2 Execution Time of DSP Version 1

Table 7.14 shows the execution time (perm+filter) of DSP version 1, the naive DSP version without any architecture-specific optimization. From the results, we could see that the average CPU cycles per loop iteration is 252.

7.8.3 Execution Time of DSP Version 2

Table 7.15 shows the execution time (perm+filter) of DSP version 2, the optimized DSP version after applying the CPU-level optimizations. It can be seen from the results that the average cycles per loop iteration have been reduced from 252 to 165, 1.5 times performance improvements.

Table 7.16: Execution time (sec) of different batch sizes for DSP Version 4 ($n = 2^{24}$, block size = 16)

no. of threads	**1**	**2**	**4**	**8**
batchsz = 1	1.88	1.31	0.71	0.41
batchsz = 2	1.84	1.29	0.69	0.41
batchsz = 4	1.80	1.29	0.68	0.41
batchsz = 8	1.85	1.28	0.69	0.42
batchsz = 16	1.90	1.28	0.68	0.41

7.8.4 Exploring Different Batch Sizes for DSP Version 4

Since the performance of DSP version 4, parallel blocking version, potentially lies on the different batch size and block size. In this subsection, we explore the performance impact on different batch sizes.

Table 7.16 shows the execution time of different batch sizes for the DSP version 4, by varying the number of threads. From the table, we can see that the batch size does not have a significant impact on performance. Consequently, we fix the batch size to 1 to reduce the loop overhead. In addition, we change the `#pragma MUST_ITERATE(1)` to `#pragma MUST_ITERATE(1, BATCHSZ)` to allow the compiler to better handle the pipelined code.

7.8.5 Exploring Different Block Sizes for DSP Version 4

In this subsection, we measure the execution time of the DSP version 4 by varying the blocking size. The result is shown in Table 7.17.

It can be seen from the table that the performance is improved with the increase of the block size. For the block size equals to 64, which is the largest possible size

Table 7.17: Execution time (sec) of different block sizes for DSP Version 4 ($n = 2^{23}$, batch size = 1)

no. of threads	**1**	**2**	**4**	**8**
blocksz = 1	1.40	1.03	0.54	0.33
blocksz = 4	1.32	0.81	0.45	0.29
blocksz = 16	1.27	0.75	0.43	0.27
blocksz = 32	1.25	0.73	0.41	0.27
blocksz = 64	1.23	0.74	0.41	0.26

fitting into the L1 SRAM, it delivers best performance. Therefore, we choose the block size as 64 for future performance measurements.

7.8.6 Execution Time of DSP Version 4

In this subsection, we measure the execution time of DSP Version 4, the parallel blocking version. According to the measurements above, we set the batch size to be 1 and block size 64, for the best performance. It is worth to note that we did not explicitly measure the performance of DSP Version 3. It is because DSP Version 3 is essentially the same as Version 4 for the single thread. So the results of DSP Version 4 on 1 thread represents the results of DSP Version 3. Table 7.18 shows the results.

7.8.7 Execution Time of DSP Version 5

The DSP Version 5 is adapted from the old Intel blocking version (corresponding to Intel Version 2), which was ported to the TI KeyStone II platform for the performance comparison purpose. Table 7.19 shows the results. The block size is 16.

It can be seen from the table that TI version 5 has poor scalability. In particular,

Table 7.18: Execution time (sec) of DSP Version 4 (batchsz = 1, blocksz = 64)

no. of threads	**1**	**2**	**4**	**8**
$\mathbf{n = 2^{18}}$	0.24	0.16	0.1001	0.0755
$\mathbf{n = 2^{19}}$	0.35	0.22	0.13	0.10
$\mathbf{n = 2^{20}}$	0.45	0.27	0.16	0.11
$\mathbf{n = 2^{21}}$	0.64	0.39	0.21	0.14
$\mathbf{n = 2^{22}}$	0.92	0.50	0.28	0.19
$\mathbf{n = 2^{23}}$	1.19	0.72	0.41	0.26
$\mathbf{n = 2^{24}}$	1.62	0.94	0.52	0.34
$\mathbf{n = 2^{25}}$	2.43	1.33	0.75	0.48
speedup($\mathbf{n = 2^{25}}$)	1.00	1.82	3.24	5.03

Table 7.19: Execution time (sec) of DSP Version 5

no. of threads	**1**	**2**	**4**	**8**
$\mathbf{n = 2^{18}}$	0.54	0.33	0.19	0.15
$\mathbf{n = 2^{19}}$	0.76	0.44	0.25	0.22
$\mathbf{n = 2^{20}}$	1.05	0.62	0.34	0.29
$\mathbf{n = 2^{21}}$	1.39	0.82	0.46	0.42
$\mathbf{n = 2^{22}}$	1.97	1.14	0.66	0.57
$\mathbf{n = 2^{23}}$	2.76	1.61	0.89	0.87
$\mathbf{n = 2^{24}}$	3.73	2.17	1.20	1.15
$\mathbf{n = 2^{25}}$	5.12	2.98	1.62	1.48
speedup($\mathbf{n = 2^{25}}$)	1.00	1.72	3.15	3.46

the performance improvement is only marginal from 4 cores to 8. The result indicates that irregular memory references, which result in high memory traffic is the principal performance bottleneck. The results also justify the hypothesis in version 4; the packing algorithm can substantially reduce the memory traffic and bandwidth burden.

Table 7.20: Execution time (sec) of different block sizes for ARM Version 3 ($n = 2^{23}$, batch size = 1)

no. of threads	**1**	**2**	**4**
blocksz = 1	1.88	0.98	0.55
blocksz = 4	1.79	0.93	0.53
blocksz = 16	1.76	0.91	0.52
blocksz = 32	1.76	0.91	0.51
blocksz = 64	1.76	0.91	0.52

Table 7.21: Execution time (sec) of ARM Version 3 (batchsz = 1, blocksz = 16)

no. of threads	**1**	**2**	**4**
n = 2^{18}	0.08	0.04	0.02
n = 2^{19}	0.24	0.13	0.08
n = 2^{20}	0.44	0.23	0.13
n = 2^{21}	0.73	0.38	0.22
n = 2^{22}	1.10	0.57	0.32
n = 2^{23}	1.75	0.91	0.51
speedup(n = 2^{23})	1.00	1.93	3.45

7.8.8 Exploring Different Block Size for ARM Version 3

The ARM Version 3 is adapted from DSP Version 4, by removing the DSP-specific optimization. In this subsection, we measure the execution time of ARM Version 3 by varying the block sizes. Table 7.20 shows the results. From the result, we could see that for block size equals to 16 generates the best performance.

7.8.9 Execution Time of ARM Version 3

In this subsection we focus on measuring the execution time of ARM Version 3. The batch size is set to 1, and the block size is 16, which generates the best performance. Table 7.21 shows the results.

Table 7.22: Execution time (sec) of regular memory access for DSP Version 4

no. of threads	**1**	**2**	**3**	**4**
$\mathbf{n = 2^{23}}$	0.31	0.18	0.11	0.07
$\mathbf{n = 2^{24}}$	0.22	0.21	0.12	0.08
$\mathbf{n = 2^{25}}$	0.44	0.33	0.19	0.12

7.8.10 Exploring the Performance of Regular Memory Accesses

In this subsection, we explore the performance of regular memory access pattern on TI KeyStone II platform. Specifically, we changed DSP Version 4 code from irregular memory access to `origx` to regular. Table 7.22 shows the performance.

7.9 Summary

In this chapter, we port the sFFT to the Texas Instruments KeyStone II SoC, a novel heterogeneous multicore embedded system. We explored key performance optimization techniques to deliver the best performance of the sFFT. The experimental results indicate that the optimizations can effectively improve the performance.

OpenMP 4.0 provides a high-level programming model to program heterogeneous accelerator-based architectures. Programmers can easily use the `target` and `map` directives to offload the code to the accelerator device and transfer the data between in between. However, programmers still need to strike the balance between the code's portability and performance, which are often non-trivial to achieve at the same time. In order to achieve the optimal performance, programmers need to use

the TI-specific extensions to handle manually the data transfer and data placement in the local memory. As a result, it makes the code "TI-specific" thus loses the portability to other possible architectures. Furthermore, as the OpenMP handles only the parallelization and synchronization of a program, it is still the programmer's responsibility to optimize the computational kernel inside of the "parallel" region. It is usually a non-trivial and tedious task, which requires the specific knowledge of the underlying architecture, and the learning curve is usually quite steep.

Chapter 8

A Heuristic to Further Improve Data Locality for Sparse FFT

8.1 Overview

As is discussed in Chapter 1, there are two major challenges in exploiting the data locality for irregular applications: dynamic irregularity and transformation overhead at runtime. In this chapter, we present an online transformation algorithm which aims to address the challenges. The new algorithm significantly reduces the inherent complexity in finding out an optimal data layout and makes it feasible to generate a better data layout on the fly. We apply the online transformation algorithm to the sFFT. The experimental results show that it can significantly improve the data locality and performance of sFFT.

8.2 An Online Data Transformation Algorithm

8.2.1 Data Reordering and NP-Completeness

Despite that many prior studies have shown promising results using data reordering transformations, the understanding of data reordering for minimizing irregular memory accesses still remain preliminary. The prior work mainly focused on heuristic approaches to exploit the temporal and spatial locality of irregular applications. However, they have not explored fundamental issues on data reorganization and its relation with irregular memory access patterns. As a result, the prior data reordering algorithms they have proposed either lack performance guarantees or are effective to only limited scenarios.

One of the most commonly adopted approaches is consecutive packing (CPACK) algorithm [25]. The essential idea of the CPACK is to use a *first-touch policy* that packs the data into consecutive memory locations in the order they are first accessed in time. Figure 8.1 demonstrates the CPACK algorithm. In Figure 8.1(a), four data elements in the array `A[]` are accessed in the order of `A[9], A[23], A[67], A[103]`. Figure 8.1(b) shows the number of cache misses under the original data layout and access sequence. To calculate the cache miss, we make additional simplistic assumptions: there is fully-associative cache that can hold only one cache block at a time; each holding up to two array elements. Therefore, the data elements are grouped into the cache blocks such as (8,9), meaning elements `A[8]` and `A[9]` are grouped in the same cache block. Since the cache is able to hold only one block a time, each access to the array will results in a cache miss, as is shown in

Access order	0	1	2	3
Data layout	A[9]	A[23]	A[67]	A[103]

(a) Original data layout and access order (simple case)

Original	(8, 9) (22, 23) (66, 67) (102, 103)				
Cache Hit/Miss	✕	✕	✕	✕	# cache misses
Cache block	(8, 9)	(22, 23)	(66, 67)	(102, 103)	4

(b) Cache misses on original data layout (✕: miss ✓: hit)

CPACK	(9,23) (67,103)				
Cache Hit/Miss	✕	✓	✕	✓	# cache misses
Cache block	(9,23)	(9,23)	(67, 103)	(67, 103)	2

(c) Cache misses after applying CPACK algorithm (✕: miss ✓: hit)

Figure 8.1: An example that illustrates the CAPCK algorithm. Simple case that each element is accessed by only once. Assume 2 elements in a cache block and only 1 cache block fits in the cache

Figure 8.1(b). Consequently, the total number of cache misses is 4. Figure 8.1(c) illustrates the cache misses has been reduced to 2 after the data layout has been reordered by the CPACK algorithm. Based on the first-touch policy, the CPACK algorithm packs the original data into two cache blocks: (9,23) and (67,103). As a result, accessing to element `A[23]` and `A[103]` will hit in the cache, and the number of cache misses is reduced by 2.

Although Figure 8.1 shows promising results that cache misses can be substantially reduced by using the CPACK algorithm, it can be extremely complicated if the array elements are accessed repetitively. Consider the example shown in Figure 8.2. Figure 8.2(a) illustrates the scenario that array elements are accessed repetitively. For instance, `A[23]` is accessed at iteration `1, 3` and `5`, respectively. Figure 8.2(b) shows the original data layout and the cache misses. Although certain elements are

accessed more than one times, showing potential temporal locality, each data access still ends up with a miss because the cache can hold only one block at a time. Therefore, the total number of cache misses is 7. In Figure 8.2(c), elements `A[9]` and `A[23]` will be packed into one cache block by the CPACK algorithm since they are accessed consecutively in time. As is shown in Figure 8.2(c), the total cache misses is reduced to 6.

However, it is obvious to see that packing `(A[9], A[23])` into one cache block is not optimal; instead, it should yield better locality if it packs `(A[23], A[67])` into once cache block because `A[23]` and `A[67]` are accessed consecutively twice (See the Figure 8.2(a) for the access order from 3 to 6). Figure 8.2(d) shows the optimal data layout by packing `(A[23], A[67])` together. It can be seen from the Figure that the number of cache misses has been further reduced to 4.

The issue has been largely limiting the applicability of the CPACK. Despite many recent efforts, CPACK has been effective for only the cases where each data element is accessed by only once. In an application with dynamic irregular references, however, this assumption hardly holds: in many irregular applications elements can be accessed repeatedly; in the sFFT code, a data entry of the signal can be sampled more than once.

It is important to note that the optimal data layout shown in Figure 8.2(d) is found by hand. Unfortunately, finding the optimal data layout in memory with minimum cache misses is a NP-complete problem in general [72], making it impractical to solve the problem precisely. Next, we will show that the challenge can be circumvented if a constraint assumed by the approach can be relaxed. We will show that

Access order	0	1	2	3	4	5	6
Data layout	A[9]	A[23]	A[103]	A[23]	A[67]	A[23]	A[67]

(a) Original data layout and access order (complex case)

Original	(8, 9) (22, 23) (66, 67) (102, 103)							
Cache Hit/Miss	✕	✕	✕	✕	✕	✕	✕	# misses
Cache block	(8,9)	(22,23)	(102,103)	(22,23)	(66,67)	(22,23)	(66,67)	7

(b) Cache misses on original data layout (✕: miss ✓: hit)

CPACK	(9,23) (67,103)							
Cache Hit/Miss	✕	✓	✕	✕	✕	✕	✕	# misses
Cache block	(9,23)	(9,23)	(67,103)	(9,23)	(67,103)	(9,23)	(67,103)	6

(c) Cache misses after applying CPACK algorithm (✕: miss ✓: hit)

Optimal	(23,67) (9,103)							
Cache Hit/Miss	✕	✕	✕	✕	✓	✓	✓	# misses
Cache block	(9,103)	(23,67)	(9,103)	(23,67)	(23,67)	(23,67)	(23,67)	4

(d) An optimal data layout that yields minimal cache misses (✕: miss ✓: hit)

Optimal	(9,23) (103,23) (67,23) (67,-)							
Cache Hit/Miss	✕	✓	✕	✓	✕	✓	✕	# misses
Cache block	(9,23)	(9,23)	(103,23)	(103,23)	(67,23)	(67,23)	(67,-)	4

(e) Padding algorithm that generates optimal data layout (✕: miss ✓: hit)

Figure 8.2: An example that illustrates the algorithms of CAPCK, optimal, and padding. Simple case that each element is accessed only once. Assume 2 elements in a cache block and only 1 block fits in the cache

the essence for designing a new data transformation algorithm can be reduced to a classical tradeoff between time and space complexity.

8.2.2 An Online Data Transformation Algorithm that Circumvents the Complexity

In last section, we reveal that finding an optimal data layout through data reordering to minimize the cache misses is generally a NP-complete problem. The rule implies that the essential challenge in data reordering comes from an implicit constraint that the produced new data layout uses no more space than the original; in other words, each item in the original data structure has only one copy in the transformed structure. In fact, if we allow more space to be used, the complexity of the problem would be reduced significantly. In this section, we present a new online data transformation algorithm which aims to circumvent the limitations of the CPACK algorithm especially at the scenario of repeated data accesses.

The idea is based on *padding*. For an irregular reference, say `A[B[i]]`, the padding algorithm creates a new array `A'` such that `A'[i] = A[B[i]]`; the reference to `A[B[i]]` is then replaced with `A'[i]`. Figure 8.2(e) illustrates an example using the padding algorithm. Similar to the CPACK algorithm, it packs the data elements into consecutive memory locations in the order such that they are accessed in time. As is shown in the Figure 8.2(e), `(A[9],A[23])` are packed into the same cache block and `(A[103],A[23])` are packed together, and so on. Therefore, the original array `A` will be replaced with a padded array `A'` with elements `A[9], A[23],`

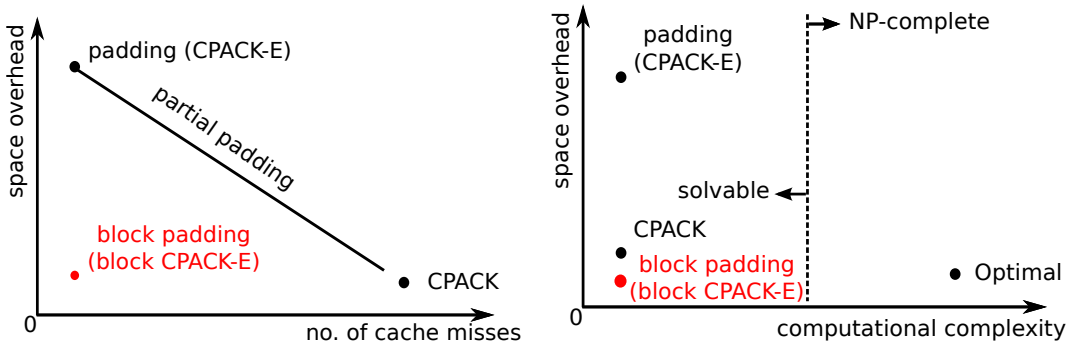
`A[103], A[23], A[67], A[23], A[67]`. As is shown in the Figure 8.2(e), the total number of cache misses has been reduced to 4, the same as the optimal data layout shown in Figure 8.2(d).

The algorithm is able to find better data layout than CPACK in the scenario of repeated data accesses because it allows to create duplicated copies of elements in the transformed array when the original data is accessed repetitively. So it relaxes the constraints that each item in the original array can have only one copy in the transformed array; instead, the size of transformed array is as large as the number of loop iterations that encloses the irregular data references, no matter how large of the original array. In the Figure 8.2(e), the padding algorithm produces 4 cache blocks, doubling the size of the original array. We name the new online data transformation algorithm as CPACK-E algorithm [1], because it extends the CPACK algorithm and circumvents its limitations in finding an appropriate data layout in the case of repeated data accesses.

8.2.3 Time-space Tradeoff

Although CPACK-E algorithm transforms the irregular accesses to regular and generates better data layout than CPACK, it comes at the cost of space overhead. For the data entry accessed by k times, the CPACK-E algorithm will generate k copies of the item in the transformed array. In this subsection, we point out that designing a new data reordering algorithm can be reduced to a classical tradeoff between time and space complexity. We will show how to reduce the space cost in next subsection.

[1] The character *'E'* stands for extension



(a) space vs. transformation quality

(b) space vs. time complexity

Figure 8.3: Positions of various algorithms in the space-time trade-off

Figure 8.3 illustrates the conceptual time-space positions for the several data reordering approaches. Figure 8.3(a) shows the space overhead vs. transformation quality in terms of cache misses. We can see from the figure that CPACK-E (based on the idea of padding) and CPACK fall into two extreme ends of the space cost and transformation quality; CPACK-E generates the better data layout but at the cost of the space overhead. CPACK, on the other hand, has less space cost but loses the guarantee of performance. It is worthy noting that several prior studies used partial padding approach [90] where only partial of the data are duplicated. Although it reduce the space overhead, it compromises the quality of the optimization proportionally, as is shown in the Figure 8.3(a). Figure 8.3(b) presents the time-space tradeoff. The CPACK-Ealgorithm has the time complexity as CPACK, but comes with greater space overhead; finding an optimal data layout by data reordering, on the other hand, has no space overhead but ends with a NP-completeness problem regarding to the time complexity. Next we propose the *block* CPACK-E algorithm which aims to reduce the space overhead of the CPACK-E algorithm.

8.2.4 *Block* CPACK-E Algorithm

In last section, we figure out that it is a NP-complete problem to find out a general data reordering algorithm that can generate the optimal data layout with minimal cache misses. We also reveal that the problem can be further reduced to a tradeoff between time and space. Based on the insights, we proposed a CPACK-E algorithm that is able to produce a better layout at the cost of space overhead. In this section, we present a new *block* CPACK-E algorithm that aims to reduce the space overhead. In fact, the block CPACK-E algorithm only uses several blocks of *constant* space.

The essential idea of the *block* CPACK-E algorithm is to apply the blocking optimization on the CPACK-E algorithm. Therefore, instead of allocating a large array A' to store the transformed data elements with repeated data entries, the block CPACK-E algorithm only requires to allocate a small block of data which has the size of several cache lines.

There are several expected benefits for this block CPACK-E algorithm. First of all, the preserve the benefit of CPACK-E algorithm that can effectively find out a better data layout in memory. Second, it overcomes the drawback of the CPACK-E algorithm with small memory footprint. In particular, the block usually has the size equals to several cache lines. Since it is independent with the problem size, the space cost of the CPACK-E algorithm has been reduced to *constant*. Furthermore, it inherits the benefit of the blocking optimization that spatial and temporal locality is well exploited before a cache block is evicted from the cache.

Complexity Analysis

As is shown in Figure 8.3, the block CPACK-E algorithm could generate better data layout than CPACK with constant space overhead. In the Figure 8.3(b), the block CPACK-E algorithm has the same order of time complexity as CPACK because it only needs to traverse the original irregular array once before producing the transformed data layout. Therefore, the time complexity of block CPACK-E is linear, i.e., $O(n)$, and the space complexity is $O(1)$, namely constant.

8.3 Performance Evaluation

In this section, we evaluate the performance improvement by the CPACK-E algorithm. Since the CPACK-E algorithm is a blocking-based algorithm, we first determine the best block size which delivers the highest performance. Then, we apply the CPACK-E algorithm on the *perm+filter* stage of sFFT, which performs the irregular memory access pattern. We evaluate the performance improvement after applying the CPACK-E algorithm. The experimental results show that the CPACK-E algorithm can further improve the performance of the *perm+filter* kernel by 30% and overall performance improvement by 20%.

8.3.1 Determining the Best Block Size

In the first experiment, we run the experiments multiple times with different block sizes in order to determine the best block size for the CPACK-E algorithm. Table 8.1

Table 8.1: Execution time (sec) of different block sizes for PsFFT ($N = 2^{27}$, $k = 1000$)

no. of threads	**1**	**2**	**3**	**4**	**5**	**6**
block size = 1	3.87	2.09	1.41	1.26	0.94	0.83
block size = 2	2.38	1.23	0.92	0.68	0.61	0.49
block size = 4	2.05	1.12	0.75	0.68	0.60	0.44
block size = 8	1.45	0.78	0.56	0.53	0.42	0.38
block size = 16	1.39	0.75	0.54	0.51	0.39	0.31
block size = 32	1.21	0.65	0.47	0.42	0.36	0.29
block size = 64	1.05	0.59	0.41	0.41	0.34	0.26
block size = 128	0.99	0.56	0.39	0.37	0.29	0.28
block size = 256	0.99	0.54	0.38	0.39	0.30	0.23
block size = 512	0.99	0.54	0.38	0.33	0.30	0.24
block size = 1024	0.99	0.53	0.39	0.32	0.28	0.24
block size = 2048	0.99	0.54	0.39	0.34	0.34	0.26
block size = 4096	0.99	0.53	0.39	0.32	0.32	0.28

shows the execution time different block sizes. We fix the signal size to be relatively large with $n = 2^{27}$ and $k = 1000$, and change the block size from 1 to 4096. We measure the execution time of *perm+filter* for the number of threads from 1 to 6, respectively. It can be seen from the Table 8.1 that the execution time significantly reduces by increasing the block size. It becomes relatively stable for even larger block size (e.g., block size greater than 512), and slightly increases then. For this reason, we choose the block size to be 512 for the rest of the experiments since it can usually deliver the best performance.

Table 8.2: Performance evaluation before and after applying the CPACK-E algorithm on PsFFT – 1 thread

(a) Execution time of the *perm+Filter* stage (in second)

Signal size $2^n, k = 1000$	**After CPACK-E**	**Before CPACK-E**	**Speedup**
19	0.03	0.03	1.06
20	0.07	0.09	1.30
21	0.11	0.15	1.37
22	0.16	0.22	1.37
23	0.25	0.33	1.33
24	0.34	0.46	1.34
25	0.50	0.68	1.36
26	0.68	0.92	1.36
27	1.00	1.37	1.37
28	1.39	1.92	1.39
		average	**1.33**

(b) Execution time of the overall PsFFT (in second)

Signal size $2^n, k = 1000$	**After CPACK-E**	**Before CPACK-E**	**Speedup**
19	0.12	0.12	1.01
20	0.16	0.18	1.13
21	0.18	0.22	1.21
22	0.25	0.31	1.24
23	0.33	0.41	1.25
24	0.43	0.54	1.27
25	0.61	0.78	1.29
26	0.80	1.04	1.30
27	1.30	1.67	1.29
28	1.83	2.37	1.29
		average	**1.23**

Table 8.3: Performance evaluation before and after applying the CPACK-E algorithm on PsFFT – 6 threads

(a) Execution time of the *perm+Filter* stage (in second)

Signal size $2^n, k = 1000$	**After CPACK-E**	**Before CPACK-E**	**Speedup**
19	0.01	0.01	1.20
20	0.02	0.02	1.20
21	0.03	0.04	1.45
22	0.05	0.06	1.34
23	0.08	0.09	1.23
24	0.12	0.13	1.11
25	0.12	0.20	1.65
26	0.19	0.26	1.36
27	0.22	0.38	1.72
28	0.39	0.48	1.24
		average	**1.35**

(b) Execution time of the overall PsFFT (in second)

Signal size $2^n, k = 1000$	**After CPACK-E**	**Before CPACK-E**	**Speedup**
19	0.05	0.05	1.02
20	0.06	0.07	1.10
21	0.06	0.09	1.34
22	0.09	0.10	1.19
23	0.13	0.14	1.10
24	0.17	0.18	1.05
25	0.18	0.25	1.41
26	0.25	0.30	1.24
27	0.43	0.59	1.37
28	0.75	0.84	1.11
		average	**1.19**

8.3.2 Experimental Results

Execution Time – Sequential

In this experiment, we evaluate the performance improvement by applying the CPACK-E algorithm to the sFFT. We measure the sequential execution time of the PsFFT (i.e., on 1 thread). We compare the results of the PsFFT after using the CPACK-E algorithm with the best performed version (i.e., Intel blocking version as was discussed in Chapter 5).

Table 8.2 shows the execution time of PsFFT before and after applying the CPACK-E algorithm. In the experiment, we change the signal size n from $n = 2^{19}$ to 2^{28} and fix the sparsity $k = 1000$. As is shown in the table, the execution time of the *perm+filter* stage is improved by 33% on average. Overall, the performance of PsFFT has been improved by 23%. The results prove the effectiveness of the CPACK-E algorithm.

Execution Time – Parallel

In this experiment, we evaluate the performance improvement of the CPACK-E algorithm on 6 threads. Table 8.3 shows the execution time of the *perm+filter* stage as well as the overall PsFFT algorithm for various signal sizes. It can be seen from the table that the average performance improvement of the *perm+filter* stage is 35%. Overall, the CPACK-E improves the performance of PsFFT by 19% on average. The result shows that the CPACK-E can effectively improve the performance of PsFFT.

8.4 Summary

In this chapter, we propose an online data transformation algorithm called CPACK-Ealgorithm, which can effectively exploit data locality for irregular applications. The CPACK-E algorithm overcomes the limitation of a canonical data transformation approach, CPACK, and points out that designing a new runtime transformation algorithm can be reduced to the time-space tradeoff. Based on the insight, the proposed CPACK-E algorithm can generate a better data layout at the linear time with constant space cost. The experimental results show that the CPACK-E algorithm significantly improves the performance of an irregular computation kernel in sFFT by more than 30% and improves the overall performance of sFFT by 20%.

Chapter 9

Conclusion and Future Work

9.1 Conclusion

In this dissertation, we present efficient parallel implementations for computing the sFFT on three state-of-the-art multicore and massively parallel architectures. We report the bottlenecks in the algorithm that impede the performance. We explore various suitable and optimized solutions to tackle the challenges. The experimental results show that our parallel sFFT is more than 5x and 20x faster than the MIT original sequential sFFT implementation on multicore CPUs and GPUs, respectively. Compared to the full-size standard FFT libraries, the parallel sFFT achieves more than 9x speedup on multicore CPUs and 12x speedup on GPUs for a broad range of signal spectra.

9.2 Future Directions

There are several future directions we intend to explore, building on the work completed. In this dissertation, we mainly focus on implementations of the sFFT algorithm. As a future work, we plan to explore more real applications with a massive need for FFT computations. We plan to study the sparsity of spectra in the applications and to replace the FFT routines by the sFFT. We expect it can significantly improve the performance of the applications.

Furthermore, our current parallel sFFT is implemented on a shared-memory multicore CPUs and a single GPU. For the input data continuously increases, it can be expected that it will eventually exceed the size of the main memory in a single compute node. In future work, we plan to extend the sFFT by exploiting the distributed-memory CPU and GPU clusters. That requires largely to re-design the algorithm by carefully partitioning the workloads among the compute nodes and minimize the communication and synchronization whenever possible.

Bibliography

[1] Bulk synchronous parallel. https://en.wikipedia.org/wiki/Bulk_synchronous_parallel.

[2] FFTW Planer Flags. http://www.fftw.org/doc/Planner-Flags.html.

[3] Intel Math Kernel Library. http://software.intel.com/en-us/intel-mkl.

[4] NVIDIAs Next Generation CUDA Compute Architecture: Kepler GK110. White paper.

[5] OpenMP Compilers. http://openmp.org/wp/openmp-compilers.

[6] The AMD Core Math Library (ACML). http://developer.amd.com/tools-and-sdks/cpu-development/amd-core-math-library-acml/.

[7] The NVIDIA CUDA Fast Fourier Transform library (cuFFT). https://developer.nvidia.com/cufft.

[8] The OpenMP API Specification for Parallel Programming. http://openmp.org/wp/.

[9] *Performance Optimization of Numerically Intensive Codes*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2001.

[10] Thrust Quick Start Guide. http://docs.nvidia.com/cuda/thrust/, July 2013.

[11] Adi Akavia. Deterministic sparse fourier approximation via fooling arithmetic progressions.

[12] I. Al-Furaih and S. Ranka. Memory hierarchy management for iterative graph structures. In *Parallel Processing Symposium, 1998. IPPS/SPDP 1998. Proceedings of the First Merged International ... and Symposium on Parallel and Distributed Processing 1998*, pages 298–302, Mar 1998.

[13] Tolu Alabi, Jeffrey D. Blanchard, Bradley Gordon, and Russel Steinbach. Fast K-selection algorithms for graphics processing units. *J. Exp. Algorithmics*, 17:4.2:4.1–4.2:4.29, October 2012.

[14] Ovidiu C Andronesi, Lixin Shi, Haitham Hassanieh, Wolfgang Bogner, Borjan Gagoski, Aaron Hess, Dylan Tisdall, Andre van der Kouwe, Dina Katabi, and Elfar Adalsteinsson. Correlation chemical shift imaging with sparse-fft and real-time motion and shim correction. In *55th Experimental Nuclear Magnetic Resonance Conference*, 2014.

[15] M.J. Berger and S.H. Bokhari. A partitioning strategy for nonuniform problems on multiprocessors. *Computers, IEEE Transactions on*, C-36(5):570–580, May 1987.

[16] T. Blumensath and M.E. Davies. Normalized iterative hard thresholding: Guaranteed stability and performance. *Selected Topics in Signal Processing, IEEE Journal of*, 4(2):298–309, April 2010.

[17] David Callahan, Steve Carr, and Ken Kennedy. Improving register allocation for subscripted variables. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, PLDI '90, pages 53–65, New York, NY, USA, 1990. ACM.

[18] A. Chandrakasan, V. Gutnik, and T. Xanthopoulos. Data driven signal processing: an approach for energy efficient computing. In *Low Power Electronics and Design, 1996., International Symposium on*, pages 347–352, Aug 1996.

[19] Yifeng Chen, Xiang Cui, and Hong Mei. Large-scale fft on gpu clusters. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, pages 315–324, New York, NY, USA, 2010. ACM.

[20] Sunita Chandrasekaran Cheng Wang and Barbara Chapman. cusFFT: A High-Performance Sparse Fast Fourier Transform Algorithm on GPUs. In *Parallel and Distributed Processing Symposium, 2016 IEEE 30th International*, May 2016.

[21] Barry Cipra. The Best of the 20^{th} Century: Editors Name Top 10 Algorithms. *SIAM News*, 33(4):1, 2000.

[22] Richard Cole. Parallel merge sort. *SIAM Journal on Computing*, 17(4):770–785, 1988.

[23] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, Clifford Stein, et al. *Introduction to algorithms*, volume 2. MIT press Cambridge, 2001.

[24] Raja Das, Mustafa Uysal, Joel Saltz, and Yuan-Shin Hwang. Communication optimizations for irregular scientific computations on distributed memory architectures. *J. Parallel Distrib. Comput.*, 22(3):462–478, September 1994.

[25] Chen Ding and Ken Kennedy. Improving cache performance in dynamic applications through data and computation reorganization at run time. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, PLDI '99, pages 229–241, New York, NY, USA, 1999. ACM.

[26] Chen Ding and Ken Kennedy. Improving effective bandwidth through compiler enhancement of global cache reuse. In *Parallel and Distributed Processing Symposium., Proceedings 15th International*, pages 10–pp. IEEE, 2001.

[27] Wei Ding and Mahmut Kandemir. Capri: Cache-conscious data reordering for irregular codes. In *The 2014 ACM International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '14, pages 477–489, New York, NY, USA, 2014. ACM.

[28] J. Ferrante, V. Sarkar, and W. Thrash. On estimating and enhancing cache effectiveness. In Utpal Banerjee, David Gelernter, Alex Nicolau, and David Padua, editors, *Languages and Compilers for Parallel Computing*, volume 589 of *Lecture Notes in Computer Science*, pages 328–343. Springer Berlin Heidelberg, 1992.

[29] M Frigo and SG Johnson. FFTW, C subroutine library. *URL http://www. fftw. org*, 2005.

[30] Dennis Gannon, William Jalby, and Kyle Gallivan. Strategies for cache and local memory management by global program transformation. *Journal of Parallel and Distributed Computing*, 5(5):587 – 616, 1988.

[31] B. Ghazi, H. Hassanieh, P. Indyk, D. Katabi, E. Price, and Lixin Shi. Sample-optimal average-case sparse fourier transform in two dimensions. In *Communication, Control, and Computing (Allerton), 2013 51st Annual Allerton Conference on*, pages 1258–1265, Oct 2013.

[32] Juan Gómez-Luna, José María González-Linares, José Ignacio Benavides, and Nicolás Guil. An Optimized Approach to Histogram Computation on GPU. *Mach. Vision Appl.*, 24(5):899–908, July 2013.

[33] 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*, page 2. IEEE Press, 2008.

[34] Elana Granston. *Hand-Tuning Loops and Control Code on the TMS320C6000, Literature Number: SPRA666*, August 2006.

[35] Hwansoo Han and Chau-Wen Tseng. Exploiting locality for irregular scientific codes. *IEEE Trans. Parallel Distrib. Syst.*, 17(7):606–618, July 2006.

[36] H. Hassanieh, Lixin Shi, O. Abari, E. Hamed, and D. Katabi. Ghz-wide sensing and decoding using the sparse fourier transform. In *INFOCOM, 2014 Proceedings IEEE*, pages 2256–2264, April 2014.

[37] H. Hassanieh, Lixin Shi, O. Abari, E. Hamed, and D. Katabi. Ghz-wide sensing and decoding using the sparse fourier transform. In *INFOCOM, 2014 Proceedings IEEE*, pages 2256–2264, April 2014.

[38] Haitham Hassanieh, Fadel Adib, Dina Katabi, and Piotr Indyk. Faster gps via the sparse fourier transform. In *Proceedings of the 18th Annual International Conference on Mobile Computing and Networking*, Mobicom '12, pages 353–364, New York, NY, USA, 2012. ACM.

[39] Haitham Hassanieh, Fadel Adib, Dina Katabi, and Piotr Indyk. Faster gps via the sparse fourier transform. In *Proceedings of the 18th Annual International Conference on Mobile Computing and Networking*, Mobicom '12, pages 353–364, New York, NY, USA, 2012. ACM.

[40] Haitham Hassanieh, Fadel Adib, Dina Katabi, and Piotr Indyk. Faster gps via the sparse fourier transform. In *Proceedings of the 18th Annual International Conference on Mobile Computing and Networking*, Mobicom '12, pages 353–364, New York, NY, USA, 2012. ACM.

[41] Haitham Hassanieh, Piotr Indyk, Dina Katabi, and Eric Price. sFFT- Sparse Fast Fourier Transform. `http://groups.csail.mit.edu/netmit/sFFT/`.

[42] Haitham Hassanieh, Piotr Indyk, Dina Katabi, and Eric Price. Nearly optimal sparse Fourier transform. In *Proceedings of the 44th symposium on Theory of Computing*, pages 563–578. ACM, 2012.

[43] Haitham Hassanieh, Piotr Indyk, Dina Katabi, and Eric Price. Simple and practical algorithm for sparse Fourier transform. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1183–1194. SIAM, 2012.

[44] Monika R Henzinger, Philip Klein, Satish Rao, and Sairam Subramanian. Faster shortest-path algorithms for planar graphs. *Journal of Computer and System Sciences*, 55(1):3 – 23, 1997.

[45] C. A. R. Hoare. Algorithm 65: Find. *Commun. ACM*, 4(7):321–322, July 1961.

[46] JA Högbom. Aperture synthesis with a non-regular distribution of interferometer baselines. *Astron. Astrophys. Suppl*, 15(1974):417–426, 1974.

[47] Amir H. Hormati, Mehrzad Samadi, Mark Woh, Trevor Mudge, and Scott Mahlke. Sponge: portable stream programming on graphics engines. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS XVI, pages 381–392, New York, NY, USA, 2011. ACM.

[48] Jiaxi Hu, Zhaosen Wang, Qiyuan Qiu, Weijun Xiao, and D.J. Lilja. Sparse Fast Fourier Transform on GPUs and Multi-core CPUs. In *Computer Architecture and High Performance Computing (SBAC-PAD), 2012 IEEE 24th International Symposium on*, pages 83–91, 2012.

[49] Texas Instruments. *TMS320C6000 Chip Support Library API Reference Guide, Literature Number: SPRU401J*, August 2004.

[50] Texas Instruments. *Optimizing Loops on the C66x DSP, Literature Number: SPRABG7*, November 2010.

[51] Texas Instruments. *TMS320C66x DSP Cache User Guide, Literature Number: SPRUGY8* , November 2010.

[52] M. A. Iwen. Improved Approximation Guarantees for Sublinear-Time Fourier Algorithms. *ArXiv e-prints*, September 2010.

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

[54] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, December 1998.

[55] Eyal Kushilevitz and Yishay Mansour. Learning decision trees using the fourier spectrum. In *Proceedings of the Twenty-third Annual ACM Symposium on Theory of Computing*, STOC '91, pages 455–464, New York, NY, USA, 1991. ACM.

[56] Monica D. Lam, Edward E. Rothberg, and Michael E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IV, pages 63–74, New York, NY, USA, 1991. ACM.

[57] Ilya Lashuk, Aparna Chandramowlishwaran, Harper Langston, Tuan-Anh Nguyen, Rahul Sampath, Aashay Shringarpure, Richard Vuduc, Lexing Ying, Denis Zorin, and George Biros. A massively parallel adaptive fast multipole method on heterogeneous architectures. *CommuNiCAtioNs of the ACm*, 55(5):101–109, 2012.

[58] David Lawlor, Yang Wang, and Andrew Christlieb. Adaptive sub-linear time fourier algorithms. *Advances in Adaptive Data Analysis*, 5(01):1350003, 2013.

[59] Bil Lewis and Daniel J Berg. *Multithreaded programming with Pthreads*. Prentice-Hall, Inc., 1998.

[60] Wai-Hung Liu and Andrew H. Sherman. Comparative analysis of the cuthill-mckee and the reverse cuthill-mckee ordering algorithms for sparse matrices. *SIAM Journal on Numerical Analysis*, 13(2):pp. 198–213, 1976.

[61] Peter Lynch. The dolph-chebyshev window: A simple optimal filter. *Monthly weather review*, 125(4):655–660, 1997.

[62] Yishay Mansour. Randomized interpolation and approximation of sparse polynomials. *SIAM Journal on Computing*, 24(2):357–368, 1995.

[63] Kathryn S. McKinley, Steve Carr, and Chau-Wen Tseng. Improving data locality with loop transformations. *ACM Trans. Program. Lang. Syst.*, 18(4):424–453, July 1996.

[64] John Mellor-Crummey, David Whalley, and Ken Kennedy. Improving memory hierarchy performance for irregular applications using data and computation reorderings. *Int. J. Parallel Program.*, 29(3):217–247, June 2001.

[65] John Mellor-Crummey, David Whalley, and Ken Kennedy. Improving memory hierarchy performance for irregular applications using data and computation reorderings. *International Journal of Parallel Programming*, 29(3):217–247, 2001.

[66] Nicholas Mitchell, Larry Carter, and Jeanne Ferrante. Localizing non-affine array references. In *Parallel Architectures and Compilation Techniques, 1999. Proceedings. 1999 International Conference on*, pages 192–202. IEEE, 1999.

[67] Todd C. Mowry, Monica S. Lam, and Anoop Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS V, pages 62–73, New York, NY, USA, 1992. ACM.

[68] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM.

[69] Akira Nukada, Kento Sato, and Satoshi Matsuoka. Scalable multi-gpu 3-d fft for tsubame 2.0 supercomputer. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 44:1–44:10, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[70] David A. Patterson and John L. Hennessy. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1990.

[71] S. Pawar and K. Ramchandran. Computing a k-sparse n-length Discrete Fourier Transform using at most 4k samples and O(k log k) complexity. *ArXiv e-prints*, May 2013.

[72] Erez Petrank and Dror Rawitz. The hardness of cache conscious data placement. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, pages 101–112, New York, NY, USA, 2002. ACM.

[73] V Podlozhnyuk. Histogram calculation in CUDA. White paper, 2007.

[74] N. Satish, M. Harris, and M. Garland. Designing efficient sorting algorithms for manycore gpus. In *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–10, 2009.

[75] Robin Scheibler, Saeid Haghighatshoar, and Martin Vetterli. A fast hadamard transform for signals with sub-linear sparsity. *CoRR*, abs/1310.1803, 2013.

[76] Jörn Schumacher. High performance sparse fast Fourier transform. Master's thesis, Computer Science, ETH Zurich, Switzerland, 2013.

[77] Lixin Shi, O Andronesi, Haitham Hassanieh, Badih Ghazi, Dina Katabi, and Elfar Adalsteinsson. Mrs sparse-fft: Reducing acquisition time and artifacts for in vivo 2d correlation spectroscopy. In *ISMRM13, Int. Society for Magnetic Resonance in Medicine Annual Meeting and Exhibition*, 2013.

[78] Lixin Shi, Haitham Hassanieh, Abe Davis, Dina Katabi, and Fredo Durand. Light field reconstruction using sparsity in the continuous fourier domain. *ACM Trans. Graph.*, 34(1):12:1–12:13, December 2014.

[79] Dean M. Tullsen and Susan J. Eggers. Effective cache prefetching on bus-based multiprocessors. *ACM Trans. Comput. Syst.*, 13(1):57–88, February 1995.

[80] Cheng Wang, Mauricio Araya-Polo, Sunita Chandrasekaran, Amik St-Cyr, Barbara Chapman, and Detlef Hohl. Parallel sparse fft. In *Proceedings of the 3rd Workshop on Irregular Applications: Architectures and Algorithms*, IA3 '13, pages 10:1–10:8, New York, NY, USA, 2013. ACM.

[81] Cheng Wang, Sunita Chandrasekaran, and Barbara Chapman. An openmp 3.1 validation testsuite. In *Proceedings of the 8th international conference on OpenMP in a Heterogeneous World*, IWOMP'12, pages 237–249, Berlin, Heidelberg, 2012. Springer-Verlag.

[82] Cheng Wang, Sunita Chandrasekaran, Barbara Chapman, and Jim Holt. libeomp: A portable openmp runtime library based on mca apis for embedded systems. In *Proceedings of the 2013 International Workshop on Programming Models and Applications for Multicores and Manycores*, PMAM '13, pages 83–92, New York, NY, USA, 2013. ACM.

[83] Cheng Wang, Sunita Chandrasekaran, Peng Sun, Barbara Chapman, and Jim Holt. Portable mapping of openmp to multicore embedded systems using mca apis. In *Proceedings of the 14th ACM SIGPLAN/SIGBED conference on Languages, compilers and tools for embedded systems*, LCTES '13, pages 153–162, New York, NY, USA, 2013. ACM.

[84] Sam White, Niels Verosky, and Tia Newhall. A cuda-mpi hybrid bitonic sorting algorithm for gpu clusters. In *Parallel Processing Workshops (ICPPW), 2012 41st International Conference on*, pages 588–589. IEEE, 2012.

[85] Barry Wilkinson and Michael Allen. *Parallel programming*, volume 999. Prentice hall New Jersey, 1999.

[86] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, PLDI '91, pages 30–44, New York, NY, USA, 1991. ACM.

[87] Bo Wu, E.Z. Zhang, and Xipeng Shen. Enhancing data locality for dynamic simulations through asynchronous data transformations and adaptive control. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 243–252, Oct 2011.

[88] Yi Yang, Ping Xiang, Jingfei Kong, and Huiyang Zhou. A gpgpu compiler for memory optimization and parallelism management. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, pages 86–97, New York, NY, USA, 2010. ACM.

[89] Paul Yin. *Introduction to TMS320C6000 DSP Optimization, Literature Number: SPRABF2*, October 2011.

[90] Eddy Z. Zhang, Yunlian Jiang, Ziyu Guo, Kai Tian, and Xipeng Shen. On-the-fly elimination of dynamic irregularities for gpu computing. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 369–380, New York, NY, USA, 2011. ACM.