HIGH PERFORMANCE SPARSE FAST FOURIER TRANSFORM USING OPENACC

by

Arnov Sinha

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

Summer 2017

HIGH PERFORMANCE SPARSE FAST FOURIER TRANSFORM USING OPENACC

by

Arnov Sinha

Approved: _____
Sunita Chandrasekaran, Ph.D.
Professor in charge of thesis on behalf of the Advisory Committee, UD-CIS

Approved: _____
Michela Taufer, Ph.D.
Professor in charge of thesis on behalf of the Advisory Committee, UD-CIS

Approved: _____
Kathleen McCoy, Ph.D.
Chair of the Department of Computer and Information Sciences

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

Approved: _____
Ann L. Ardis, Ph.D.
Senior Vice Provost for Graduate and Professional Education

ACKNOWLEDGMENTS

Graduate school is an important and a tough milestone which a student crosses, as it not only leads deep into the subject they have been studying for over 6 years, but also is the important stepping stone towards specialized knowledge and career, which is gained when studying for a Masters degree. I could not have achieved anything without the constant support of my advisor, Dr. Sunita Chandrasekaran. Her guidance, support, encouragement, patience and persistence for the past 2 years for my research has led me to write my thesis and complete my Masters degree. It was her devotement and enthusiasm to the community that motivated me to take up research and write my thesis. I would also like to thank my other committee member Dr. Michela Taufer, who took the time to review my work and offer her valued feedback.

I would also like to mention my CRPL lab members, as they have gone through endless presentation sessions, bug correction, and have given me tons of valuable feedback for helping me make this algorithm more efficient and better.

Last, but not the least, I would like to thank my family for their constant support, patience and belief in me, while i was undergoing my Masters degree in another country far away from them. I could not have achieved anything here without them. I dedicate this research and thesis to my family and especially to my father, Mr. Deba Prasad Sinha, who always believed in me and supported me. He has always been my pillar, and my biggest mentor and even during this thesis, he explained me concepts which i could not understand before. I dedicate all i have done and achieved to you.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

The Sparse Fast Fourier Transform (sFFT) is a recent algorithm developed by Hassanieh et al. at MIT for Discrete Fourier Transforms on signals with a sparse frequency domain. A reference implementation of the algorithm exists and proves that the sFFT can be faster than modern FFT libraries for signals of sparse nature. The algorithm has been parallelized using multiple approaches, such as over multicore by Cheng et al. over GPGPUs using CUDA by Cheng et al. and optimized for serial execution using SSE intrinsics by Schumacher et al.

While the increase in number of cores and memory bandwidth on modern architectures provide an opportunity to improve performance through sophisticated parallel algorithm design, the sFFT is inherently complex, embarrassingly parallel, and numerous challenges need to be addressed to deliver the optimal performance. In this Masters Thesis, we employ a high-level directive-based parallel programming model, OpenACC to create a performance portable sFFT code. We call our implementation, ACCsFFT. Our implementation can target heterogeneous platforms consisting of x86 or Power Processors integrated with GPUs. The performance of our implementation is compared against existing parallel implementations that have used either low-level or proprietary software on CPUs and GPUs.

Several optimizations are proposed and implemented in ACCsFFT. The performance is also compared against the highly optimized parallel FFTW library. We also target GPUs from different families, old to the most modern hardware, to verify if the algorithm is performance portable, scalable and reproducible across generations.

We deliver a high-level parallel sparse FFT library capable of running one version of the code in a serial manner on CPU or in parallel on multicore, GPGPUs and other

architectures that OpenACC currently supports. A programmer would only need to change input parameters of the algorithm for the different runs.

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, and cryptography. FFT (Fast Fourier Transform) is the most widely used and popular algorithm for DFT. FFT is the fastest algorithm based on divide-and-conquer approach. The time complexity of FFT is $O(nlogn)$ time.

FFT has been universal importance in scientific and engineering applications for a long time. But many applications now have input which are sparse in nature, i.e. most of the data in the input signal is zero or corrupted and the significant data points or coefficients are spread across the input signal. FFT on a sparse signal is computationally expensive, as most of the signals computed is unnecessary, depending on the sparsity value k for a signal size n.

Sparse signal FFT is needed in a variety of applications, ranging from DNA sequencing, to Nuclear magnetic resonance, Seismic imaging, Audio and video compressions, to GPS. More examples will be described in detail in Chapter 3. There arises a need to have an algorithm which can target sparse input signal and perform FFT on it which would not be as computationally expensive as performing, general FFT. Hassanieh et al.[31] developed an algorithm for the same known as Sparse FFT, which reduced the time complexity of performing FFT on a sparse input signal to $O(logn\sqrt{nklogn})$.

With the emergence of big data problems, the input size grows significantly, so does the computation time needed to compute the FFT of the same, even when using the sparse FFT for sparse algorithms. The sFFT is inherently a parallel algorithm, which has stages of execution which are good candidates for parallelization. Sparse FFT does pose challenges for parallelization, as it is an irregular algorithm, the computation is directly proportional to the size of the input signal n and the sparsity value k. More on the challenges of parallelization will be discussed in later sections and chapters.

Parallelization enables a better time complexity of the algorithm and opens up a new dimension of the application of the algorithm. With parallelization, a larger input signal size can be used and can support a large amount of sparsity factor. These are important in terms of real world applications like that of DNA sequencing, GPS, Seismic processing, etc. A parallelized Sparse Fast Fourier (sFFT) algorithm can be sped up to more than *10x* of the serial version. This is a significant improvement and can improve the practicality of the algorithm for the community which want to use sFFT for real world applications. That is the main motivation behind this thesis, which we are going to approach.

1.2 Fourier Transform: An Overview

The Fourier Transform is an important and well-known mathematical method with a variety of applications in many scientific disciplines. In its discrete (DFT) form it can be formulated as

$$\hat{x} = DFT_n \cdot x \tag{1.1}$$

where x and $\hat{x}$ are n-dimensional complex input and output vectors. There are many applications for the DFT; for example in signal processing, DNA sequencing, seismic imaging, radio astronomy, nuclear magnetic resonance, noise filtering or numerical solution of PDEs, amongst others.

A straightforward evaluation of equation 1.1 involves $O(n^2)$ operations. Since the DFT is such a useful tool for many applications, there is a need for fast algorithms. The most well-known fast algorithm for DFTs is the *Fast Fourier Transform (FFT)*, originally described by Cooley and Tukey in [25]. The asymptotic runtime of this FFT is $O(nlogn)$ and it is therefore much faster than the straightforward algorithm.

FFT reduced the runtime of the original DFT algorithm, which was revolutionary. it was voted as the top 10 algorithm of the 20th century[26]. Though many optimizations were offered in multiple papers to make the asymptotic runtime of FFT better, none were successful. It is possible to make better algorithms by adding constraints on the input- and output-vectors x and $\hat{x}$ though. The sparse FFT proposed by Hassanieh et al.[31] is one such algorithm.The sFFT can be applied to signals $x \epsilon C^n$ with a sparse frequency domain $\hat{x}$, i.e. only $k < n$ unknown elements of $\hat{x}$ are nonzero and while the time domain signal x is still dense.

Besides the algorithmic improvements, new computer architectures are constantly developed and improved. Modern general purpose CPUs feature multi-level caches, instruction level parallelism or vector instruction sets. Additionally, accelerator technologies like GPUs or FPGAs can be used to boost program performance. Parallelism is becoming increasingly important, as modern desktop CPUs typically package multiple cores, or computers can be connected to compute clusters. With this variety of target platforms it is hard for compilers to generate optimal machine code that makes use of all features and runs at high performance. Thus, manually optimized libraries for specific target platforms are being written for all kinds of algorithms.

The original Cooley/Tukey-FFT and similar algorithms have been implemented in such high-performance libraries like FFTW [28] or CUFFT [9]. These implementations make use of modern computer architecture features and are carefully designed to deliver the highest possible performance.

Similarly the sFFT algorithm have been implemented on these architectures, like sFFT library[35] which uses SSE intrinsics and other optimizations for x86, Parallel sparse FFT (PsFFT) for Multicore using OpenMP[42] and CUDA sparse FFT (cusFFT) for GPGPUs[43], both by Cheng et al. Unlike OpenMP which is a directive based programming language which can allow the same code to be used as serial or over multiple cores, based on the compiler options, CUDA code base cannot be run serially, and you need to have a GPGPU to utilize it. And although there exists a library, sFFT library, they only contain the serial implementations of version 1.0,2.0,3.0 and not of any parallel code base.

In this thesis we are using OpenACC to port sFFT 1.0/2.0 implementation, which will have one code base for both the Multicore and GPGPU architecture ,also support more architectures which will be discussed in the later sections.

1.3 Evolution of hardware architectures

Computer system architectures have always been greatly influenced by underlying trends of hardware and software technologies. We have come a great way from single core processor architecture and beyond the time when stacking processors and communicating using message passing system to create distributed systems, were the only possibility to have something close to a parallel architecture and to run an algorithm faster or in parallel. In this section we will quickly introduce various hardware

of the past, the present and some of the architectures which may become future tech and how we have moved past and left Moore's Law behind.

Single Core:

Processors are chipsets with a single CPU (one processing unit) inside it. Microprocessors have inherently been a single core processor since their inception in the early 1970's and previous focus was to create a faster single core processor which can execute more instruction per clock cycle. This trend went on till the inception of the early dual core/ multicore processors.

Multicore:

After the turn of the century, post 2000, chipsets with two or more CPUs started emerging. Dual core microprocessors is a single chip which has two distinct CPUs that work simultaneously. IBM introduced the first dual core chipsets in its Power 4 chips. And the first x86 based chips were introduced in the year 2005. Dual core systems may use the same or a little bit more energy, but achieve approximately an 80% increase in processing power over single core CPU chips. A chipset that contain more than one CPUs comes under multicore. At one point the industry realized that instead of trying to get a single core to speedup by cramming in more and more transistors, it would be more practical to have multiple cores in a single chip. Since then all applications were updated to support Symmetric Multiprocessing (SMP) which helps the operating system to work on one core and the applications on others. This also paved the path for virtual machines, where one chip can handle multiple operating systems and multiple applications of those operating system at the same time, having essentially 2 or more machines on a single chip.

Coprocessor:

Is a secondary computer processor which helps the main processing chip with functions. Generally, the operations performed by coprocessor are graphics, signal

processing, floating point arithmetics, etc. Coprocessor acts like a accelerator for computationally intensive tasks and accelerate system performance.

Graphic Processing Unit (GPU):

Is an accelerator, which consists of thousands of Arithmetic Logic Units (ALUs) which does computations similar to a processor and coprocessors. GPUs were essentially created to do computations for computer graphics and image processing. But it was later accessed to help a processor do all the computationally heavy tasks, similar to a coprocessor. GPUs also have a bigger memory making them ideal for applications related to deep learning, machine learning, etc.

FPGA:

Field Programmable Gate Array Is an integrated circuit, which is designed to be configurable according to the need of a developer. It consists of many logic blocks which can be reconfigured and interconnected to create many different combinations. This makes it ideal for a wide variety of application, from high-volume applications to state of the art products. Can be used for embedded memory operations, digital signal processing, etc.

1.4 Thesis objective

At the time of writing this thesis, 3 different sFFT algorithm versions were implemented, 3rd was optimized using SSE, and the first two were ported to parallel architectures. The first reference implementation, sFFT Version 1 and 2, were published[17]. The third version was published on the as a part of the sparse FFT library[18]. The reference implementations are written in standard C++ code, single threaded and without any hardware-specific modifications. The FFTW library is used for internal DFT computations.

The goal of this thesis is to take the sFFT reference implementations, port them to various heterogeneous architectures and have a single code base for multiple architectures, and optimize it to obtain a high performance library. Optimizations include, asynchronous operation, atomicity, reductions, improved cache locality, etc.

1.5 Challenges

1.5.1 Parallelism

Sparse FFT is irregular algorithm. The algorithm grows computationally as the size of the input signal grows and the memory access pattern becomes more and more irregular. There are a lot of loop carried dependencies present in the algorithm which hinders optimal parallelism. Thread synchronization is among the main concerns when it comes to parallelism. Since multiple threads/kernels need to be involved and they will be accessing shared resources at the same time, the need to be careful and implement thread synchronization, such as mutex locks, critical sections, asynchronous access and movements, etc. , mechanism comes into play. These should be placed carefully and minimally otherwise will hinder the parallelism process and will result in less efficiency.

1.5.2 Porting sFFT to diverse architectures

In our work, we aim to develop parallel sFFT implementations for Multicore CPUs and GPGPUs using directive based programming and letting the compiler do all the work for efficient porting of the algorithm to the different architectures.

Modern multicore CPUs are optimally designed for operations 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.

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.

An optimal implementation to exploit both of these architectures is more than challenging, so for the purpose of this thesis we will be designing the algorithm as optimally as possible for both, favoring more towards GPGPUs and relying on the compiler technology to optimize the same code efficiently for multicore CPU architecture.

1.5.3 Reproducibility

The most important thing to keep in mind while creating a library is that you have to make sure it is reproducible across various conditions and architecture. For the purpose of that i have selected OpenACC directive based programming model for making sure that the reference and the implementation created by me is reproducible and working. Also, a configuration file is created for fetching the required configuration for the library and a docker image of the same will be uploaded to the website of the implementation done by me, so as to make sure that it is reproducible by all.

1.6 Thesis organization

The thesis, starts off with mentioning the current state of the art algorithm in FFT and sFFT, in chapter 2 and describes the optimizations and parallelizations done until now. Chapter 3 describes the sFFT algorithm in detail and provides analysis of

it. Chapter 4 begins with an overview of parallelism and then describes the OpenACC programming model along with an example. The parallelization approach taken by me along with the challenges we faced and the profiling of the algorithm is shown in chapter 5. We finally conclude and provide our future vision in chapter 6.

Chapter 2

STATE OF THE ART

2.1 FFT implementation

There are many optimized implementation of the original Cooley/Tukey FFT algorithm, among others. They have been optimized for many platforms. Some of the libraries include, CUFFT for GPUs, AMD Core Math Library [2], FFTW for serial and multi threading compatible to multiple x86 - based architectures, and Intel Math Kernel Library (MKL) for Intel processors[4].
FFT is a memory dependent, as it is memory bound algorithm, and depends on the memory subsystem designed and how to utilize and exploit it.

2.1.1 FFTW library

FFTW is an optimized library for multiple x86 - based architectures. The computation in FFTW works by, *executor* that comprises of optimized, decomposable blocks of code known as *codelets*. which is a specialized piece of code which computes part of the transform[27].

A FFTW plan is a data structure which contains combination of codelets applied by the executor. The plan is determined at the runtime by dynamic programming, before the actual computation begins. The planner is responsible to reduce the final execution time and not the number of floating point operations. Finally the planner measures the run time of many execution plans and decides on the fastest one, this is only if the right option is selected for it.

2.2 cuFFT

CUDA is a low level programming language created by NVIDIA to perform actions and manipulate instructions on the cores available on an NVIDIA GPU. Each GPU has compute cores which are effectively known as CUDA cores. These cores perform operations specified by the program on data available on the GPU. CUDA toolkits have a no of highly optimized libraries to run various functions. One of them is cuFFT[9]. The NVIDIA CUDA Fast Fourier Transform (cuFFT) library provides interface for computing FFTs on input data up to 10x faster than serial FFT algorithm. cuFFT uses hundreds and thousands of CUDA cores available by a GPU to deliver floating-point performance without the need to create your own GPU FFT version.

cuFFT uses the Cooley-Tukey [25] and the Bluestein[40, 23] algorithm to compute DFT of complex and real valued input data.

2.3 Other Parallel FFT algorithms

There are many algorithms available since the inception of the original Cooley-Tukey FFT algorithm. Because FFT inherently is a parallel algorithm which can easily be manipulated to do the same. Without going into deep into each of the large amount of algorithms available, here are a few, *radix-2 FFT, IFFT*[39], parallel FFT on an MIMD machine [22], etc.

2.4 Sparse FFT (sFFT)

Sparse FFT algorithm was created specifically to target sparse data and applications requiring the same. it has two main stages. First is the estimation stage where the input signal is broken down and only the large most significant portion of the signal is kept. Second is the location stage, where the now sorted and estimated signal coefficients is located and recovered. The original implementation of the Sparse FFT had two versions, 1.0 and 2.0, created in MIT. the differences between them was

the kind of filter used for estimating and locating the significant coefficients from the sparse input signal. More about the functioning and about 1.0/2.0 version of the sFFT algorithm will be described in the next chapter.

2.5 Recent Developments in Sparse FFT

2.5.1 sFFT 3.0

Version 3 of the algorithm is a high performance optimized version. The main idea of version 3 of sFFT from ETH Zurich is same as version 1 and 2, It features two major improvements.

The first improvement is based on the observation that once a frequency coefficient of the signal was found and estimated, it can be removed from the signal. This can help reduce the computational effort for the algorithm done in the next steps. Although updating all of the signals would require $O(n)$ operations [36]. Although according to the author it is not, necessary to update the input signal, instead it is sufficient enough to update the B-Dimensional output of a measurement (application of filter, DFT and sub-sampling). This way the removal of the effects can be done in $O(B)$ time.

The second major improvement was the scheme for finding the signals significant frequency coordinates using individual measurements. In the original version1 and version 2, multiple location loops were run and their results combined in order to get correct candidate coordinate at a high probability. But according to the paper [30], it is noticed by the author that two distinct measurements (calls to HashToBins) are enough. This helps in loop unroll and provides more parallelism.

2.5.2 sFFT 4.0

Version 4 of the sFFT algorithm uses the same ideas as version 3, but eliminates the restriction that only exact k-sparse signals can be used. It does this by using the same scheme for finding candidate coordinates as version 3, but allowing again more than two distinct measurements and reconstructing a finite number of bits of the coordinates in each measurement. This approach is similar to a binary search, in each step the region of a frequency is further reduced, even if it is allowed to perform adaptive sampling. The details of this algorithm are very complex, and at the time of this writing no implementation of sFFT v4 exists. The details of the algorithm are described in [30]

2.5.3 Parallel Sparse FFT

Parallel Sparse FFT (PsFFT) is an OpenMP optimized code base of version 1 and version 2 of sFFT. PsFFT was the first successful attempt at parallelizing the sFFT code for multicore architecture using OpenMP. It resulted in approximately 5x speedup as compared to the serial sFFT versions.

The authors converted the original code from C++ to C before implementing it. This was done for two reasons. First, they used flat arrays and structs where data is stored sequentially. It leads to better cache utilization. Second, a simpler code structure in C makes it easier to parallelize and performs better also for low embedded systems. It also helps compilers to exploit code optimization techniques.

Some of the optimizations done in PsFFT include:

Index Coalescing

Instead of using critical sections in sections which has loop carried dependence, as it can cause a large performance penalty if no parallelization occurs in the loop, index coalescing approach is used which eliminates the loop carried dependence. When

updating the index value, instead of waiting for its previous result, the basic idea of the index coalescing approach was to directly map the index with each of the loop iterations.

Data affiliated loops

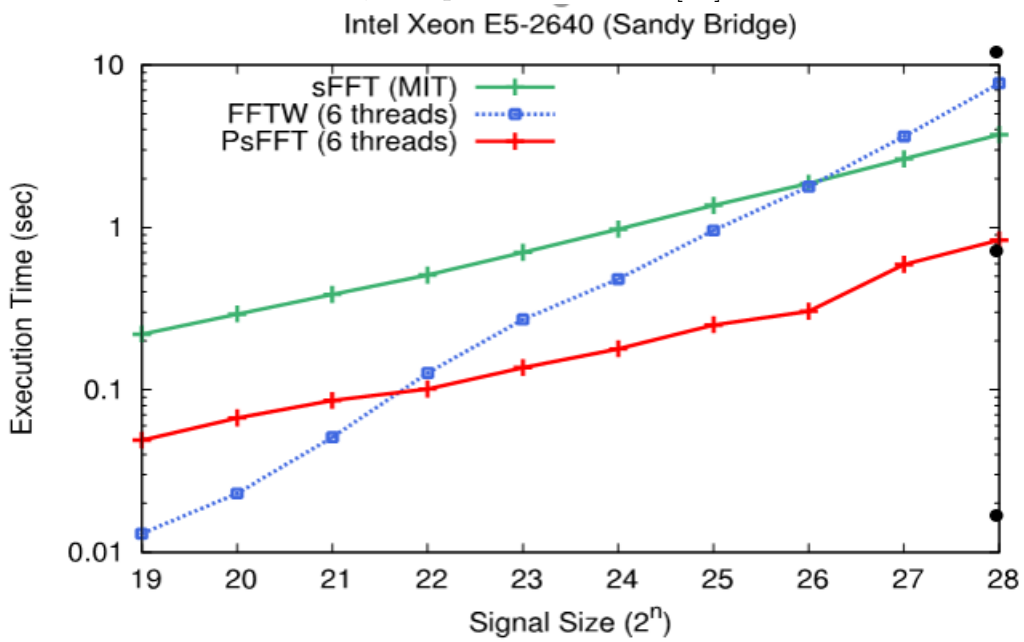
When multiple iterations are running in parallel, there is a chance that same memory is updated by different threads causing collisions. It was avoided by having two separate iterations which were collision free. A two layered iteration space is created where the inner layer is collision-free and suitable for a data-parallel mechanism, such as OpenMP. However, the global synchronization impedes the complete loop to be fully parallelized. It also causes large large overhead when the number of threads increases. To address this problem, enhanced data affiliated loops approach was introduced, where 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.

Blocking Techniques

Most of the time consuming part of the algorithm is memory bounded with poor spatial locality. The authors realized the importance of enhancing the memory hierarchy utilization and to properly redesign the data usage of the algorithm. They did so by keeping the data in cache or registers therefore reducing memory bandwidth pressure.

Cache Blocking: The data is divided up and bucketed into cache sized blocks, after it is filtered. and the operations are carried out on these. This helps in avoiding repeated fetching of data from the main memory. Accessing the buckets and filter is

Figure 2.1: Parallel Sparse FFT result compared to serial implementation and FFTW. $K = 1000$, is kept constant here.[19]



straightforward since they are unit stride. Conversely, accessing the signal itself is irregular due to the large-stride memory access pattern and leads to poor spatial locality.

TLB Blocking: Translation Look-aside Buffer blocking is required to avoid significant amount of page faults associated with the algorithm. The approach is to divide the signal, filter and buckets in blocks of page size. Therefore, the number of blocks is the minimum number of pages that data reside in. This leads to the minimum number of compulsory TLB misses.

Register Blocking: Then next step to avoid collision is to reorganize the data into small dense block sizes, the size of a register of the machine it is being computed on. This eliminates the loads and stores by reusing values that are in registers, and also by avoiding spilling.

Using these optimizations and more, the authors created a parallel OpenMP based algorithm for multicore. They achieved about 4.5X speedup using 8 threads on an Intel Sandy Bridge 8-core multiprocessor. Figure 2.1 shows a result compared to the original serial MIT version of sFFT and the highly optimized FFTW library. Parallelizing the serial implementation of the Sparse FFT algorithm enables us to use a signal of a higher size with more sparsity value. In real world applications the signal size n and the sparsity value k is generally large and the Parallel Sparse FFT implementation can compute the result 4-5x faster than the serial version.

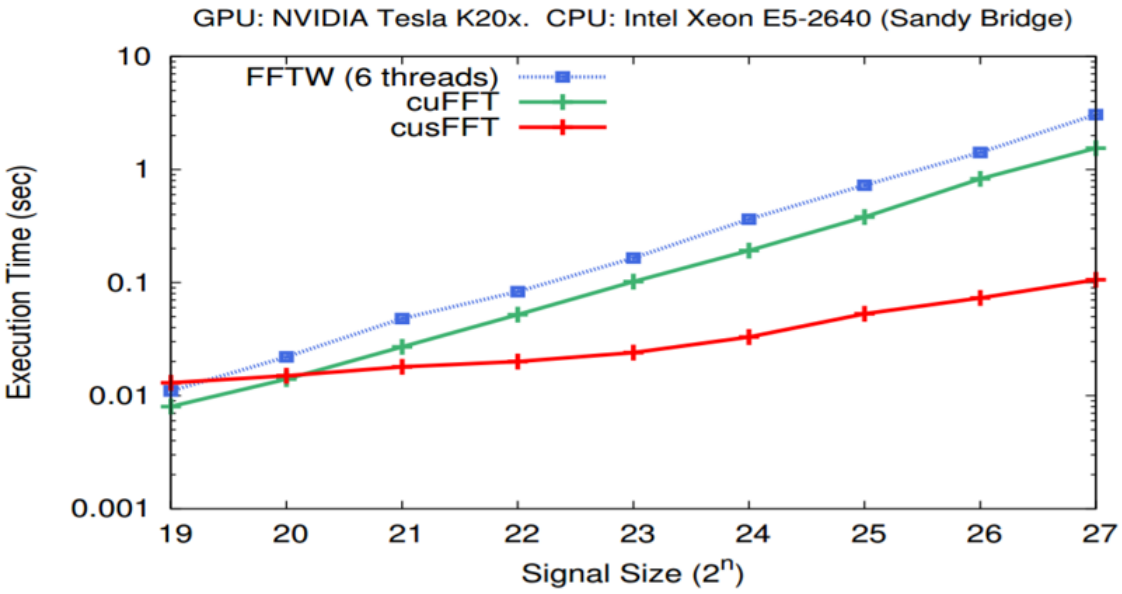
2.5.4 CUDA Sparse FFT

Sparse FFT is an algorithm which is irregular in nature, as the computation depends on the size of the signal and the sparsity value. It also has parts which are memory bounded. GPUs have a large amounts of cores and memory to accompany an irregular algorithm like Sparse FFT. It allows us to improve the performance through optimized parallel algorithm design, but porting it to GPUs is not as easy as multicore, there are multiple challenges involved. If optimizations are not done properly, it will adversely affect the algorithm, and in some cases make it even slower than the serial algorithm version.

At first the serial sFFT, used to create the OpenMP version of Parallel sparse FFT, is profiled. Profiling the sequential sFFT shows time distribution for each of the steps when the signal size n increases and sparsity factor k is fixed and vice versa. It is depicted in figure 2.3.

Left image shows the profiling results when sparsity (k) is fixed and the signal size (n) varies. The right hand size shows the profiling results when sparsity (k) is varying and the signal size (n) is fixed.

Figure 2.2: CUDA Sparse FFT result compared to serial implementation and FFTW.[43]

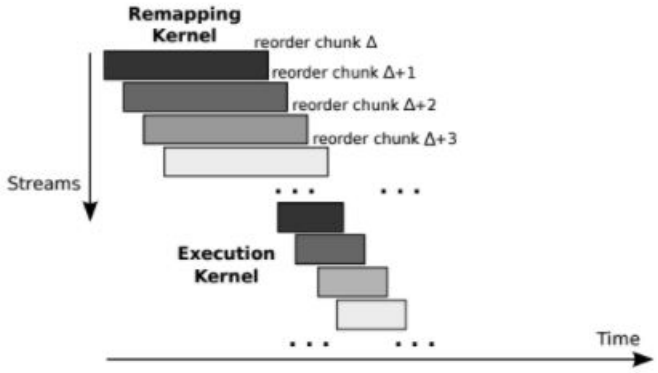


The thread management, also scheduling, synchronization and creation of different threads are completely autonomous i.e. it is managed by the hardware. To achieve the maximum performance on the GPU hardware, requires a deeper understanding of the memory hierarchy and the execution model of the GPU. Various optimizations were made to the algorithm to work efficiently.

Asynchronous Data Layout Transformation

The first two stages of the algorithm permutes the input signal and hashes them into smaller bins. The index number generated for the same is largely non comparable and causes irregularity. This irregular memory access pattern leads to non-coalesced memory global memory access, which in turn creates memory traffic increasing latency and cause bottleneck for achieving optimal performance.

Figure 2.3: Asynchronous data layout transformation[43]



The compiler can be used in cases where the algorithm behaves this way to detect the irregularities and reorders the computations at the compile time. But Sparse FFT is a runtime based algorithm where the input is unknown till runtime and even changes during computation. So it is of no help, as the index is randomly generated and the access pattern is dynamic in nature.

To coalesce the memory access the asynchronous data layout approach was chosen, that can reorder the data dynamically. The original non-coalesced 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.

This method happens in two kernels, first creates a second array data structure which contains the coalesced data. The new order is created based on a desirable mapping technique between threads and data locations. For loop iteration i,

$$A'[i] = A[index]$$

where A us the original input signal of size n.

Second 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.

Fast K-selection Algorithm

In sparse FFT after binning is done of the permuted and filtered signals, we take only the *K-largest* frequency coefficients out of them. This is done by the *cutoff* step. It basically sorts all the lists of bins and then selects the k largest elements from the sorted lists.

This is not going to be a problem if the signal size i small, but as the signal grows, the amount of data grows and sorting the lists becomes more and more expensive. The cost would be typically, for a bin of size B, $BlogB$ for a typical sorting algorithm. For an sFFT algorithm most of the bins are almost empty and only few are large.

IN the fast k-selection algorithm created and described by the authors of cusFFT, they assign B threads and each thread processes one element in the bin. If the values in the bin is larger than the threshold, the element is chosen and index is stored. It is important to note that the threshold values here to be chosen is of utmost importance. 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[41].

2.5.5 Summary

To summarize, this section overviews the existing work in FFT implementation in terms of High performance versions. this section also highlights the various techniques used by others to solve an irregular algorithms over different architectures. The problems they faced, and the approach to solution they took.

sFFT is an NP-complete problem to find an optimal data layout with minimum cache misses in general. It is cumbersome to look for an optimal algorithm which can have both task and data parallelism and to get the optimal performance out of the system and making it reproducible across various architectures, having a better memory layout, either with or without hardware extensions, through data reordering, computation transformation, or their combinations

In this thesis work, we gather all the information and approaches used for creating an optimal algorithm, and use a new programming model and better optimizations to create a scalable parallel sparse FFT algorithm, which we will later describe in chapter 4.

Chapter 3

SPARSE FAST FOURIER TRANSFORM : AN OVERVIEW

3.1 Overview

In this section we breakdown the original Sparse FFT (sFFT) algorithm into several stages and explain the working of the algorithm. Then highlight the computational aspect of the algorithm, analyze them and finally show the differences in terms of sFFT 1.0, 2.0 and 3.0.

3.2 Sparse FFT

Discrete Fourier Transform (DFT) is a numerical algorithm used to convert time domain to frequency domain, the transform operates on discrete data, in our cases a signal whose interval often has units of time. DFT has lots of applications like, Wave simulations, audio/video compressions, cryptography, etc. FFT [28] [24] is a faster divide and conquer approach to DFT. It computes DFT of a signal of size n from time to frequency domain and vice versa with a computational complexity of $O(nlogn)$.

DFT is a computationally expensive algorithm. FFT was a boon for computing DFT faster at $O(nlogn)$ time and was voted as the *10 best algorithm of the 20th century,cipra2000best*. But today the size of the signals computed grows bigger and bigger, so the time to compute FFT also increases. So regardless of the input size, sparsity, structure the computation time does not increase $O(nlogn)$. But many applications have input data signals which are sparse in nature 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 and important.

This results in the algorithm being sub-optimal, because $O(nlogn)$ operations on n input data points lead to only a few number of k non-zero significant outputs, where $k << n$, while the rest of $(n-k)$ coefficients are zero/negligible small. Here, n is the signal size and k is the sparsity factor.

There are many applications which have sparse input data signals, such as Nuclear magnetic resonance[21], Seismic imaging[19], GPS[20, 32, 33, 29], DNA sequencing, biomedical signals[38], medical images[37], audio/video compressions, social graphs, financial graphs and many more. Sparse Fast Fourier Transform addresses all these applications and provides a solution. FFT computes the entire input data with size n, whereas sFFT it filters out the input data and works on a small set of the original which has more significant coefficients which needs to be computed. It finds the *k-largest* output coefficients out of a small compressed FFT computation. This results in substantial performance improvements.

To find the relevant part of the input data which is not empty or corrupt, sFFT employs filters and permutes the signals into small sets of buckets/bins. The filters depending on the version uses Gaussian filter for getting the largest coefficients towards the center of the bucket and Dolph-Chebyshev/Mansour filter for better heuristic estimations. 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 of sFFT

For a time-domain $x \epsilon C^n$ signal of size n, the Fourier spectrum $\hat{x}$, will have k non zero Fourier coefficients. G is the flat window function whereas $\hat{G}$ denotes its spectrum in the frequency domain. The algorithm for the same is given in Figure 3.4.

Figure 3.1: sFFT algorithm Version 1.0

Algorithm 1 SFFT v1.

Input: $x \in \mathbb{C}^n$, $k < n$, $L \in \mathbb{N}$. Output: A k-sparse vector $\hat{x}$.

1. Run a number of L *location loops*, returning L sets of coordinates $I_1, \ldots, I_L$.
2. *Count* the number s_i of occurrences of each found coordinate i, that is: $s_i = |\{r | i \in I_r\}|$.
3. Only keep the coordinates which occurred in at least half of 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 $\hat{x}_i$ as $\hat{x}_i = \text{median}\{x^r_i | r \in \{1, \ldots, L\}\}$. The median is taken in real and imaginary components separately.

sFFT Version 1.0, consists of multiple executions. They are classified into two types. *Location loop* and *Estimation loop*. Location loop creates a list of coefficients which have a probability of being indices of one of the k nonzero coefficients in $\hat{x}$. This loop runs a number of times, the more it runs, the better the probability of locating the k nonzero coefficients in $\hat{x}$. Estimation loop are used to determine the frequency coefficients from the now selected coefficients from $\hat{x}$ to get

$$\hat{x_i} \in I$$

where I is the given set of coordinates from the location loop.

There are 6 main stages of Sparse FFT. Random spectrum permutation, Flat window function, Sub-sampled FFT, Cutoff, Reverse hash function for location recovery, Magnitude estimation. These are explained in detail below.

Stage 1: Random Spectrum Permutation

sFFT algorithm begins by dividing a large set of Fourier coefficients into smaller size buckets by using convolution method with filter(s), which will be discussed in detail in the next section. There are many challenges in this stage, how to deal with input data where two or more large coefficients are located too close to each other, and this cannot be easily segregate via bucketing. To make sure each of the buckets have a single large Fourier coefficient, the algorithm employs random permutation to the input signal so that the nearby large Fourier coefficients in the frequency domain are evenly separated. The distance between the original location and permuted location should be large enough so that two large close coefficients are not placed into the same bucket.

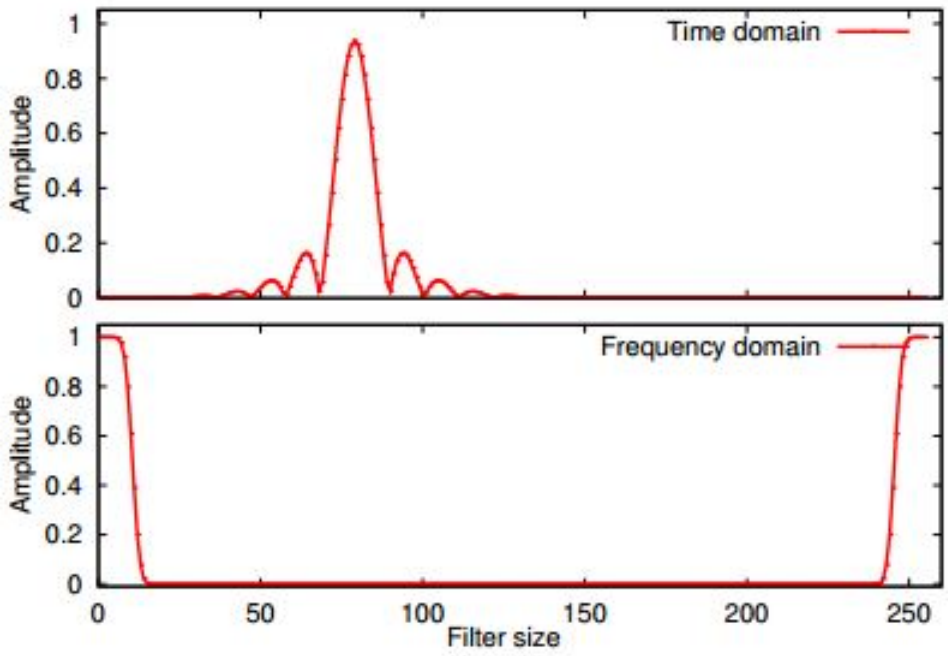
The sFFT employs a hashing-based spectrum permutation method to address issue. Specifically, it defines a *hash* function that maps indices of the original signal spectrum to the permuted locations so that the original locations can then be recovered at the end of the algorithm.

The random permutation stage of the algorithm is used to get different results in subsequent location loop runs, because of two reasons. First iteration of the random permutation loop is only going to contain the correct k non zero coefficients at a constant probability. Second iteration results in many coordinates to be mapped to the same bin. Using multiple runs of this stage with different random permutation helps in reducing the probability of a non significant, very small coefficient or zero, to be mapped into one of the nonzero bins J each time and is therefore falsely considered as one of the candidate coefficient.

Stage 2: Flat Window Function

The Flat window function is used as a filter to extract a certain set of coefficients in x, as sFFT is a sublinear time algorithm, only part of the input data is used to compute FFT. Like discussed in the section above, we need to separate them into

Figure 3.2: Gaussian filter, a Flat Window Function in time (top) and frequency (bottom) domain. Here in this example, the window size $n = 256$

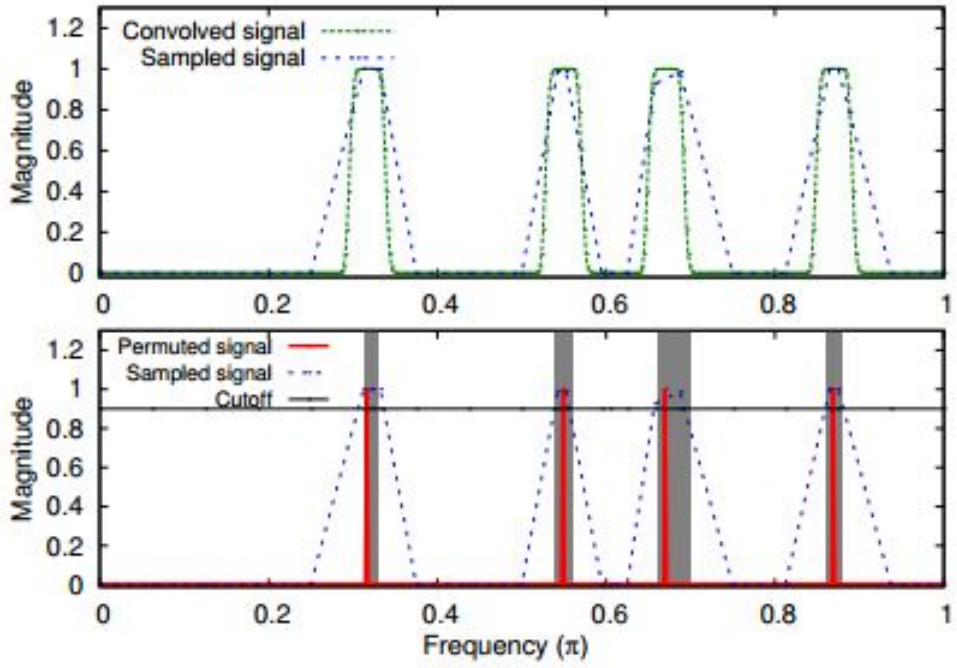


buckets, having a single large frequency coefficient, we can then recover the frequency individually from the buckets. It leads to the sample complexity and the execution time is directly proportional to the number of buckets, which is lower bounded by the signal sparsity k.

The algorithm is then filtered to smoothen out the curves. Gaussian filter is used here. Due to the nature of Gaussian filter, it positions the largest frequency coefficients towards the center and positions the smaller or empty part of the spectrum towards its edges in the time domain.

Along with the Gaussian filter, Dolphstartsh-Chebyshev filter G, depicted in Figure 3.2 is also used. They are both used together because Gaussian and Dolphstartsh-Chebyshev filter together concentrates the signal both in time and frequency resulting in G being close to zero, and the Fourier transform of the same $\hat{G}$ is negligible except

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



for a fraction of the coefficients with an exposed tail outside of it.

Stage 3: Sub-sampled FFT

In this stage, the permuted and now filtered input coefficients x gets the rate of B sampled and is hashed into a set of B bins. Now instead of computing *N-dimensional* FFT, it can compute the subsampled *B-dimensional* FFT, where B is the number of bins. This is done in $O(BlogB)$ time. So now, x is subsampled, summed up and a *B-dimensional* FFT is performed on it (Figure 3.3). Each bin has at most one non-negligible coefficient.

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 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

Stage 5: Reverse Hash Function For Location Recovery

After most of the sparse non-significant coefficients are removed in the previous stage, the rest of the coefficients which were selected as the candidate coefficients have to be reconstructed back by finding the original locations in the frequency domain and estimating the magnitudes.

The purpose of estimation loops, the second type of loops in sFFT version 1, is to reconstruction the exact coefficient values given a set of coordinates I. The implementation of estimation loops is similar to location loops: they share the first 3 steps.

The previous stages define a *hash function*

$$h_\sigma : [n] \to [B]$$

that maps size of n input data to the number of B buckets. The hash functions has to be reversed by removing the phase changes that happened in the previous steps to get back the original locations in the frequency domain. this is done by computing reverse hash function h_r

These stages run for $L = O(logn) location loops$ with different permutation parameters σ and τ, and return the L sets of locations of candidate coefficients $I_1, ..., I_L$. 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. $I' = i \in I_1 \cup ... \cup I_L | s_i > L/2$.

Stage 6: Magnitude Estimation

In this final stage, given the locations I' and frequencies $\hat{x_{I'}^r}$, from location loops, it estimates each frequency coefficient $\hat{x}_i$ as $\hat{x}_i = median(s_i^r | r \in 1, ...L)$. The median is taken in real and imaginary components separately.

Outer Loop

sFFT has two main loops, outer an inner. The outer loops executes all of the stages above. As described above, by running the inner loop multiple times, we increase the probability of finding the locations and recover the magnitude of the large Fourier coefficients. Algorithm 3.1 shows the iteration of the outer loop. And flow diagrams of the same is shown in Figure 3.4.

When multiple frequencies hash to the same bin, a hash collision occurs and the estimation fails. To compensate this, the value of $\hat{x}_i$ can be set to the median of the corresponding outputs of all estimation loops

3.4 Sparse FFT Version 2.0

sFFT version 2.0 is similar to the 1.0 version. The main difference is that sFFT 2.0 applies *Mansour filter*[34], which helps in getting better heuristics in the location recovery stage which is the stage 5 above (Reverse Hash Function and location recovery). This helps in finding out the Fourier coefficients which are large effectively and quickly. This is done by the *Mansour Filter* by restricting the location of the large Fourier coefficients. The algorithm for sFFT version 2.0 is shown in Figure 3.5

Unlike the other filters which have been mentioned in the *Flat Window* subsection above, there is no spectral leakage in *Mansour Filter*. Since this results in lower error rate, Mansour Filter needs fewer iterations to get the result thus the execution

time is reduced. There are drawbacks to using *Mansour Filter* though, the permutations cannot be used to resolve hash collisions, since only the offset is random. However according to [31] this is not an issue in practical algorithms.

3.5 Sparse FFT Version 3.0

sFFT version 3.0 was part of the high performance Sparse Fast Fourier Transform library created by Jorn Schumacher as part of his thesis at the ETH Zurich [35]. The sFFT 3.0 is similar to the version 1 and 2, but it has two major improvements.

The first improvement was based on the observation that frequency coefficients which were located and estimated based of the sFFT stages mentioned above, are repeated again for another iteration, so they can be safely removed so as to reduce the computation. this helps even more in reducing the amount of work to be done in later stages. However updating the whole signal requires *O(n)* operation and is therefore expensive. But there is no need to update the input signal, only *B-Dimensional* output of a measurement (filtered, DFT and subsampling) is enough. This results in *O(B)* operations.

The second improvement made in version 3 is an improved method of locating the large frequency coefficients using individual measurements. In sFFT v1.0 and sFFT 2.0, as described above,runs multiple location loops and the results are combined to get higher probability [30] of locating large coefficients. proves that two distinct calls are enough.

The idea here is to perform the measurements with similar permutations that only differ in the phase-altering parameter. Permutations have two parameters τ and σ. The two calls to HashToBins are performed with the same σ, but one time with $\tau = 0$ and one time with $\tau = 1$.

When no hash collision occurs only a single nonzero frequency coefficient maps to a bin. Since the phase change in the second measurement also depends on the bins coordinate, the coordinate can be reconstructed out of the phase difference of the two measurements[36]. The drawback of this approach is that it is only applicable to exact *k-sparse signals*, i.e. k-sparse signals which are not affected by any noise. The algorithm for the same is shown at Figure 3.6. And the flow diagram is shown in Figure 3.7.

3.6 Analysis of sFFT

Asymptotic runtime

In this section, asymptotic runtime bounds for the Sparse Fast Fourier Transform algorithms are derived. Table 5.5

Algorithm Version	Cost
sFFT v1	$O(logn\sqrt{nklog(n)})$
sFFT v2	$O((logn\sqrt[3]{nk^2log(n)})$
sFFT v3	$O(klogn)$

Figure 3.4: sFFT 1.0 Flow diagram

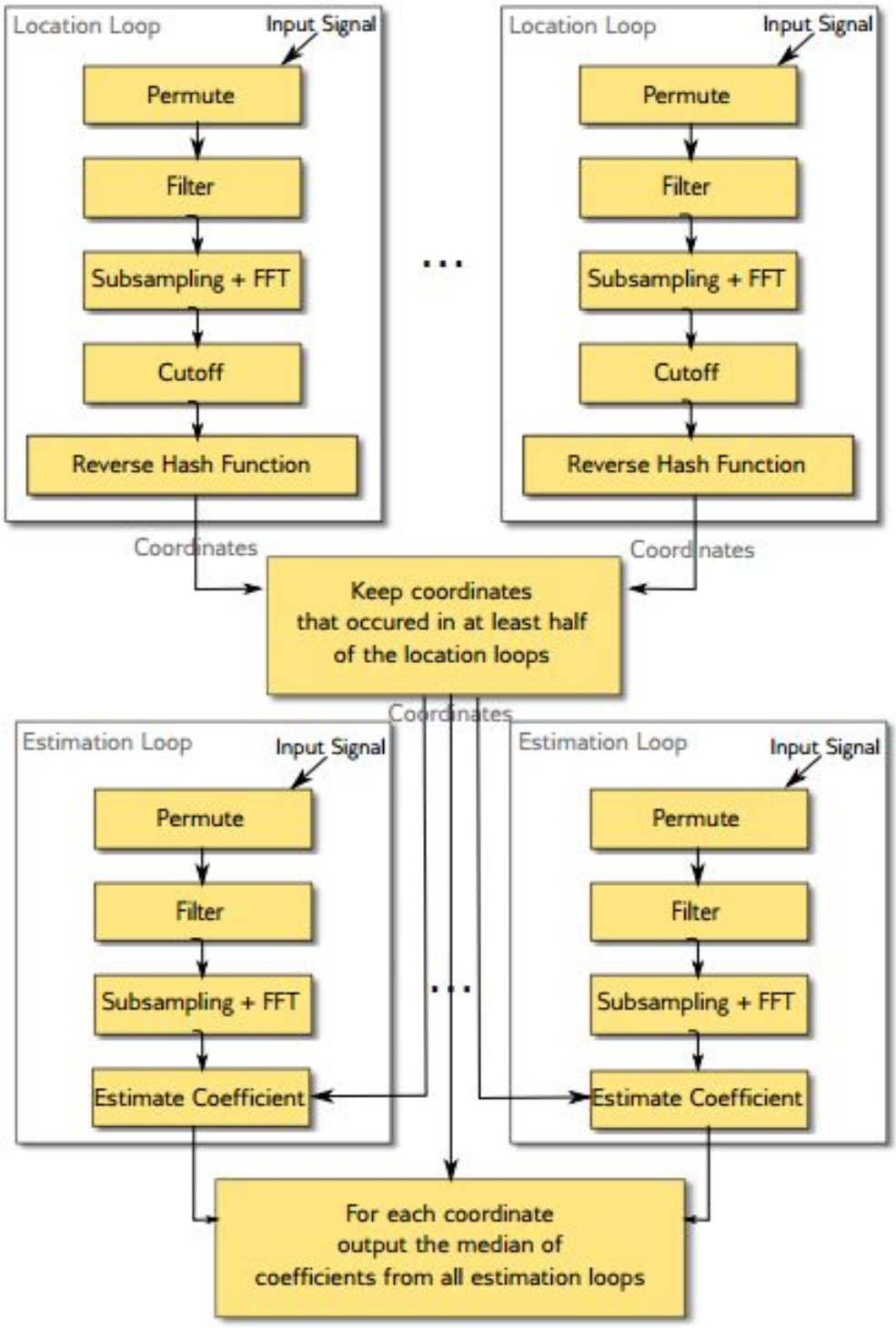


Figure 3.5: sFFT algorithm Version 2.0

Algorithm 2 SFFT v2.

Input: $x \in \mathbb{C}^n$, $k < n$, $L \in \mathbb{N}$. Output: A k-sparse vector $\widehat{x}$.

1. Run a number of L_1 *Mansour loops*, returning L_1 sets of coordinates $I_1, \ldots, I_{L_1}$.
2. Run a number of L_2 *location loops*, returning L sets of coordinates $I_{L_1+1}, \ldots, I_{L_1+L_2}$. Let $L = L_1 + L_2$.
3. *Count* the number s_i of occurrences of each found coordinate i, that is: $s_i = |\{r | i \in I_r\}|$.
4. Only keep the coordinates which occurred in at least half of the location loops. $I' = \{i \in I_1 \cup \cdots \cup I_L | s_i > L/2\}$.
5. Run a number of L *estimation loops* on I', returning L sets of frequency coefficients $\widehat{x^r_{I'}}$.
6. *Estimate* each frequency coefficient $\widehat{x}_i$ as $\widehat{x}_i = \text{median}\{x^r_i | r \in \{1, \ldots, L\}\}$. The median is taken in real and imaginary components separately.

Figure 3.6: sFFT algorithm Version 3.0

Algorithm 3 SFFT v3.

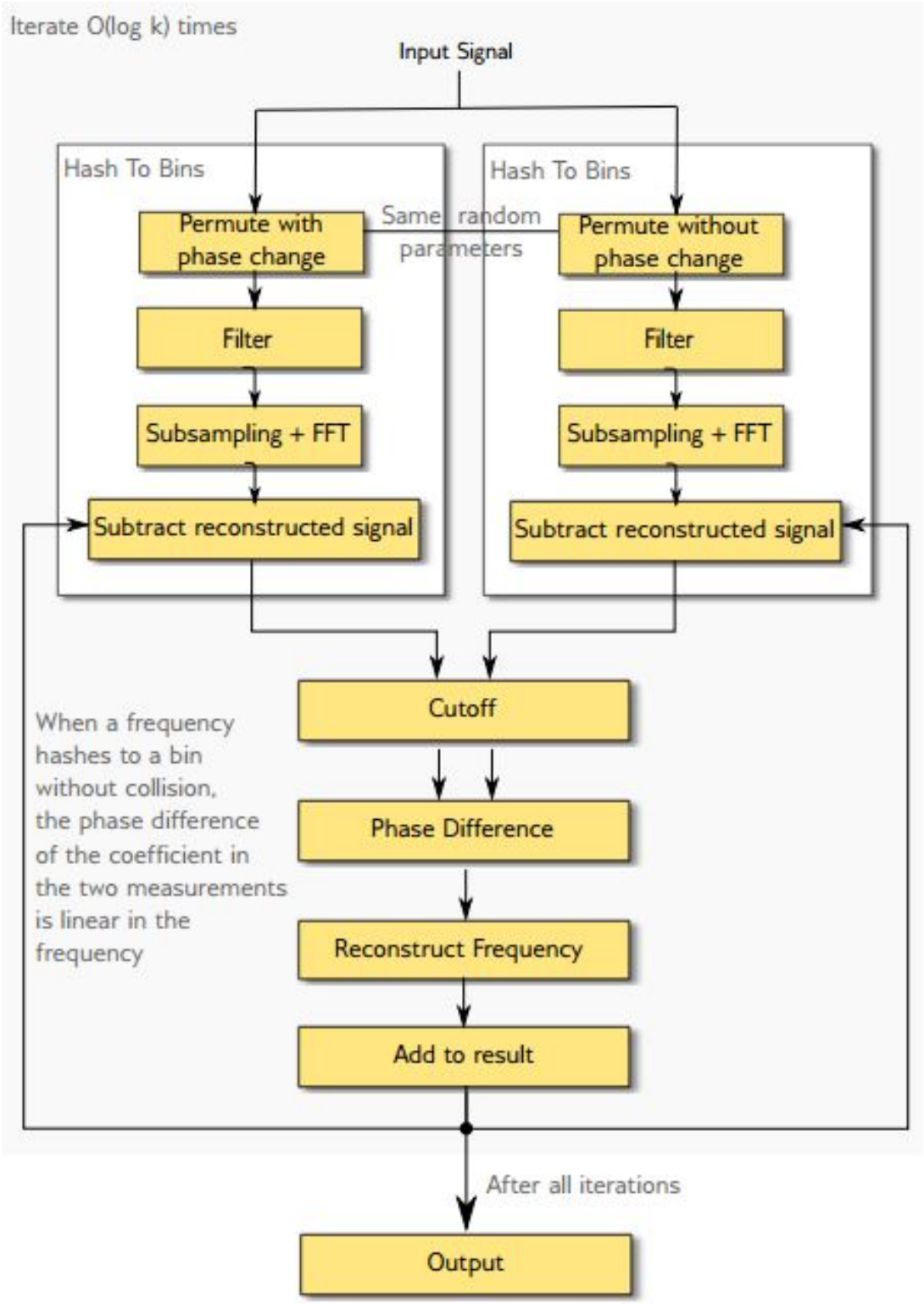
Input: $x \in \mathbb{C}^n$, $k < n$, parameters B, α, β, δ. Output: A k-sparse vector $\widehat{x}$.

1. Initialize $z \in \mathbb{C}^n$ to 0.
2. For $t = 1 \ldots \log k$ do
   (a) Randomly choose an odd number σ and any number b from $[n]$, and generate the permutations $P_{\sigma,0,b}$ and $P_{\sigma,1,b}$.
   (b) Perform two measurements with the function HashToBins using the two permutations on the input signal. Assume the output of the individual measurements are $\widehat{u}$ and $\widehat{u}'$.

   The function HashToBins works like this:
      i. Let $B = k/(2^t \cdot \beta)$.
      ii. Compute $\widehat{y}_{jn/B}$ for $j \in [B]$, where $y = G_{B,\alpha,\delta} \cdot (P_{\sigma,a,b}x)$.
      iii. Remove the effects of the already found frequencies by computing $\widehat{y}'_{jn/B} = \widehat{y}_{jn/B} - \left(\widehat{G'_{B,\alpha,\delta}} * \widehat{P_{\sigma,a,b}z}\right)_{jn/B}$ for $j \in [B]$.
      iv. Return $\widehat{v}_j = \widehat{y}'_{jn/B}$.

   HashToBins works very similar to some steps of SFFT v1 location loops. Constants α, β, δ have to be set appropriately.
   (c) For every nonzero bin $\widehat{u}_j$:
      i. Set $a = \widehat{u}/\widehat{u}'$.
      ii. Assuming there is no hash collision (i.e. only one frequency coordinate was hashed to bucket j), the phase of a is now linear in the frequency coordinate. One can reconstruct it using $i = \sigma^{-1}(\text{round}(\text{phase}(a)\frac{n}{2\pi})) \mod n$.
      iii. $\widehat{u}_j$ can be used as estimate for the magnitude of the coefficient of frequency i. Thus, set $\widehat{z}_i = \widehat{z}_i + round(\widehat{u}_j)$.
3. $\widehat{z}$ is an approximation to the DFT of the k-sparse signal $\widehat{x}$.

Figure 3.7: sFFT 3.0 Flow diagram[36]



Chapter 4

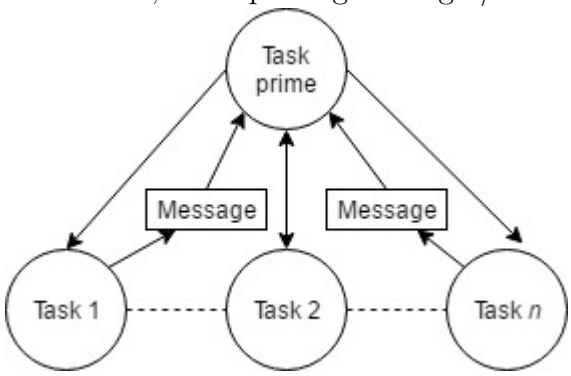
OVERVIEW OF GPU PARALLELISM AND OPENACC

4.1 Intro to Parallelism

Parallelism in computing is a term coined to denote that multiple execution of processes or computations run simultaneously in parallel. Parallelism is growing rapidly now days due to the slow growth of frequency scaling, preventing serial processes to increase in speed. There are many hardwares now which are used for achieving parallelism, such as multicore CPUs, GPGPUs (General Purpose Graphic Processing Unit), DSP, FPGA, coprocessor etc. These when combined and programmed to achieve parallelism are called heterogeneous systems. they operate by communicating with each other. All of these hardware have pros and cons, and functionality which are specific to them. For example, a coprocessor can compile its own set of program to run in parallel. Whereas a GPU is used as a form of accelerator for computation purposes. FPGA's on the other hand can be configured according to the need of a developer and consumer to achieve the functionality which will be beneficial.

When trying to parallelize algorithms, four different types of parallelism which we should know of. They are *Bit-level, Instruction level, Task and Data parallelism.* For the purpose of this thesis, we will only be explaining task based and data based parallelism. *Task Parallelism* is used for distributing and parallelizing computational tasks in an algorithm across multiple parallel threads of execution. A simple way of the same is pipelining tasks of an algorithm. The main idea behind *Task parallelism* is to make sure that tasks execute, concurrently or parallely, independent from other tasks, ideally having their own data sets to work upon. Tasks can execute the same

Figure 4.1: Task Based Parallelism, each of the task are handled by an independent thread of execution, often passing messages/data to each other[41].

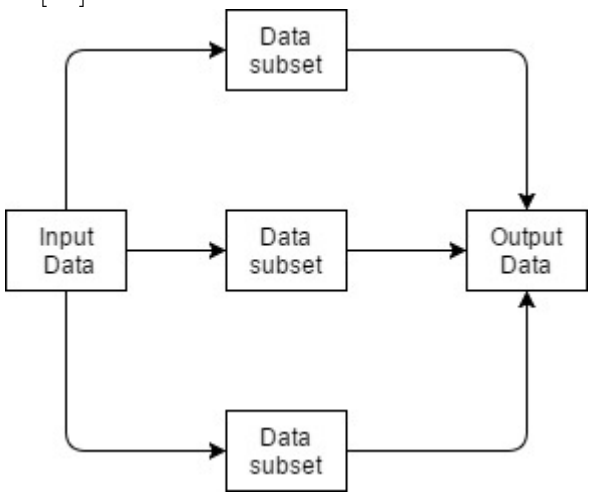


or different portion of the code and communicate by passing data from one thread of execution to the other, as seen in the figure 4.1. Often, depending on the algorithm, the results are combined after all the threads are done with their function, otherwise the results are written separately in another data structure chosen by the programmer.

Data parallelism focuses on distributing data across different threads of execution or nodes. These threads or threads of executions operate on different sets of data in parallel. Each of these threads run operations on the data serially. By using multiple threads of execution, an algorithm can make use of parallel architectures such as multicore processors and massively parallel GPUs. This helps make an algorithm makes faster as multiple threads will be running on parallel.

Parallelizing algorithms is not as easy as it sounds, there are a lot of factors to be considered before parallelizing an algorithm. Some algorithms are inherently parallel, where as some are absolutely non-parallel in nature. if you try to run in parallel an algorithm which is not parallelizable, it will detrimental to the performance. In terms of applications being parallel in nature, they are divided into three classifications, fine-grained, coarse-grained and embarrassingly parallel. Fine grained are those where the subtasks communicate with each other many times a second. Coarse grained

Figure 4.2: Data Parallelism, the data gets divided into different concurrent threads of execution, which in turn results in a faster parallel execution of operations[41].



parallelism are those where subtasks do not communicate that often. Embarrassingly parallel don not communicate at all or rarely do. These are easiest to parallelize as they rarely need synchronizations. There are many things to be aware of also when trying to parallelize an algorithm, such as race conditions, mutual exclusions, synchronizations, and parallel slowdown.

4.2 Introduction to GPUs

Parallel processors such as GPUs are the cornerstone of parallel programming. They have many numerical applications. The computations that arise in these applications lend themselves naturally to efficient parallel implementations. GPGPUs is being increasingly adopted as a general-purpose computing platform to accelerate a vast majority if scientific and engineering applications. For the purpose of the thesis, we have selected NVDIA's GPGPU for our accelerator based architecture to describe. The algorithm implemented by me, theoretically works on varying accelerators from other manufacturers as well.

There are a lot of accelerator architectures available by NVIDIA and they come in wide variety of configurations, we are focusing on Kepler GK100[12] and Pascal GP100 based architectures[10] as a testbed. A full Kepler based configuration consists of an array of 15 Streaming Multiprocessors (SM), each of which features 192 single-precision CUDA cores, and each core has a fully pipelined floating-point and inter arithmetic logic units. Each SM could access up to 65536 registers. A full Pascal based configuration consists of an array of Graphics Processing Clusters (GPCs), Streaming Multiprocessors (SMs), and memory controllers. A full GP100 consists of 6 GPCs, 60 Pascal SMs, and eight 512-bit memory controllers (4096 bits total). Each of the GPC has 10 SMs and each of those SMs has 64 CUDA cores.

All of the thread managements, scheduling, synchronization, and management are managed by the hardware, so essentially the overhead is minimum. The SM schedules threads in groups of 32 parallel threads called warp. Each SM has a warp , one per processing block each with dual instruction dispatch units, so that they can be issued and executed concurrently.

4.2.1 GPU Programming Challenges

To get maximum and optimal performance on a accelerator based GPGPU platform, in a lot of cases it requires a deep understanding of the memory hierarchy and the 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. All threads of a warp should read/write global memory in a coalesced way, 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.

4.3 OpenACC: A Directive Based Programming Model

OpenACC is a directive based portable high-performance parallel programming model, designed to port codes and algorithms to a wide variety of heterogeneous platforms and targeting multiple hardware architectures[14]. All the while making programming less tiresome and giving the programmer control over the code, with less effort than a lower level model like CUDA.

OpenACC is useful to provide hints to the compiler about parallelization possibilities and movement of data from the host to device and back from device to host, although this heavily depends on the kind of architecture OpenACC is targeting. Programmers use directives or pragmas to work along with the compiler to parallelize, tune and finally optimize parallel codes to achieve performance.

It is a portable model, and targets a wide variety of architectures to create one solution for all parallelization needs. This helps in making parallelization possible for multiple architectures and heterogeneous platforms by coding once optimally and

reusing it in multiple platforms.

The current architectures which are currently supported by OpenACC are, GPGPUs, NVIDIA and AMD, multicore, IBM power systems, eg. Power8, and currently support for many others are underway like ARM, Knights Landing, etc.

4.3.1 OpenACC Language Features

Parallel Constructs: Parallel constructs launches a number of parallel gangs in parallel, and each one of them have multiple workers which in turn have vector or SIMD operations. Clauses for this constructs are defined below in Loop Constructs

Kernels Construct: gives the compiler flexibility to decide the way the code segment should be parallelized, how the threads should be managed, synchronized for the targeted device. It also lets the compiler decide if it is safe to parallelize the code segment in the first place. *Parallel* construct on the other hand told the compiler explicitly that the section is safe to parallelize. It has clauses like async, wait, vector_length, etc.

Data Constructs: a device data constructs hints the compiler about the movement of data relevant for parallelization. It also defines the region of the program within which data is accessible by the device. It has clauses like, create, copyin, copyout, etc.

Host Data Construct: makes the address of the device data available for the host. It has clauses, use_device.

Loop Constructs: a loop constructs applies to the immediately following loop or nested loops, and describes the type of device parallelism to be used to execute the iterations of the loop. It has clauses like async, wait, reduction, gang, worker, vector,

collapse, etc.

4.3.2 Examples

Here we will discuss an example of how OpenACC will parallelize *Jacobi Iteration* code and port it to a GPU. We will only show a part of the algorithm in snippets.

Jacobi Iteration: is an iterative algorithm for finding a solution to a standard diagonal linear system of equations.

Below is the main part of the Jacobi Iteration algorithm in C++ code. This is the serial version. The outer while loops is in constant iteration till the solution of the system is found and is converged. The first nested for loop, applies a 2D Laplace operator at each element of a 2D grid[15]. The next nested inner loop copy the output result back as the input for the next iteration.

We first start applying a *kernels directive* on the loop we want to parallelize. This tells the compiler to run the loop till the end of scope on a targeted architecture, in our case here its on a GPU. *Kernels directive* creates parallel accelerator CUDA kernels on an NVIDIA GPU and runs them on parallel.

Post the *kernels* directive, we can insert additional directive which give us more control over the parallelism, like data movement clauses and specifying to the compiler which loop are safe to be run independently and can be parallelized more. The code with OpenACC directives is shown below.

Here in Table 4.2 the *loop independent* directive specifies which loops can be run independently.

Table 4.1: Jacobi iteration serial

```
while ( error > tol && iter < iter_max ) {
        error = 0.f;
    {
        for( int j = 1; j < n-1; j++) {
            for( int i = 1; i < m-1; i++ ) {

                Anew[j*m+i] = 0.25f * ( A[j*m+i+1] + A[j*m+i-1]
                                      + A[(j-1)*m+i] + A[(j+1)*m+i]);
                error = fmaxf( error, fabsf(Anew[j*m+i]-A[j*m+i]));
            }
        }
        for( int j = 1; j < n-1; j++) {
            for( int i = 1; i < m-1; i++ ) {
                A[j*m+i] = Anew[j*m+i];
            }
        }
    }
      if(iter % 100 == 0) printf("%5d,%0.6f\n", iter, error);
        iter++;
} \\end of while loop
```

Table 4.2: Jacobi iteration parallel with OpenACC

```
    while ( error > tol && iter < iter_max ) {
        error = 0.f;

    #pragma acc kernels
    {
        #pragma acc loop independent
        for( int j = 1; j < n-1; j++) {
            for( int i = 1; i < m-1; i++ ) {
                Anew[j*m+i] = 0.25f * ( A[j*m+i+1] + A[j*m+i-1]
                                      + A[(j-1)*m+i] + A[(j+1)*m+i]);
                error = fmaxf( error, fabsf(Anew[j*m+i]-A[j*m+i]));
            }
        }
        #pragma acc loop independent
        for( int j = 1; j < n-1; j++) {
            for( int i = 1; i < m-1; i++ ) {
                A[j*m+i] = Anew[j*m+i];
            }
        }
    }
        if(iter % 100 == 0) printf("%5d, %0.6f\n", iter, error);
        iter++;
    }
```

This algorithm achieves almost 20x of speedup compared to a serial run, and you can notice that none of the lines from the original code is changed and only by adding few directives, which a compatible compiler can read and others can ignore, we can achieve parallelism.

4.3.3 Advantages and Disadvantages

There are a lot of advantages of OpenACC compared to other parallel programming models. The strengths, mentioned below, enabled OpenACC to be selected as the high-performance parallel programming model to be chosen for parallelization of Sparse Fast Fourier Transform

- OpenACC is portable and can be used with a wide variety of architectures
- OpenACC can be combined with other parallel models like CUDA, OpenMP, MPI, etc
- As it can be combined with CUDA, allows us to develop grammar using OpenACC's higher level directives and implement complex functions
- Directive based programming also allows compilers which does not recognize OpenACC model to ignore the directives and compile the code serially
- OpenACC is simpler to use as you don't need to change the source code and just provide hints to the compiler as where the parallelization is possible and how data can be moved
- It does not require significant amount of writeup/changes in an algorithm, so the number of lines of code is lesser than that of CUDA.

OpenACC is gaining a lot of momentum in the scientific community. It is being used in various domains of science, such as computational hydrodynamics[3], astrophysics[3], climate weather Oceans[6]. computational fluid dynamics[1], quantum chemistry[8], computational electromagnetics[5], medical imaging[7], etc.

Chapter 5

PARALLELIZING SFFT USING OPENACC

In this chapter we present a high performance parallel algorithm for computing sFFT on GPGPUs and multicore, namely ACCsFFT, using OpenACC directive based programming model. We will also discuss the sequential implementation of the code we are going to use and the profiling of the same.

Before parallelizing any code, the first thing that anyone should do is profile the code which they are trying to parallelize. The reason for so is to find the hot spots of the algorithm, which tells us how the code works and if it can be run on parallel. Algorithms which have spread out hot spots (computation time) are difficult to be made parallel as compared to the ones which have concentrated hot spots. it also helps us decide if the code can be run on an accelerator or a multicore architecture. There are multiple profiling programs available, Gnu-Prof(gprof), Tau, Vampir, PGI prof (pgprof), Nvidia prof (nvprof), etc.

5.1 Sequential Implementation and Profiling

Here we discuss the sequential implementation and performance evaluation of the original implementation of sFFT by MIT which will serve as a starting point for the parallel implementations. The version used here for parallelization is based on the reimplementation done in [42] of the original MIT version and was dubbed, UH sFFT. MIT and the UH implementation differed on the following points.

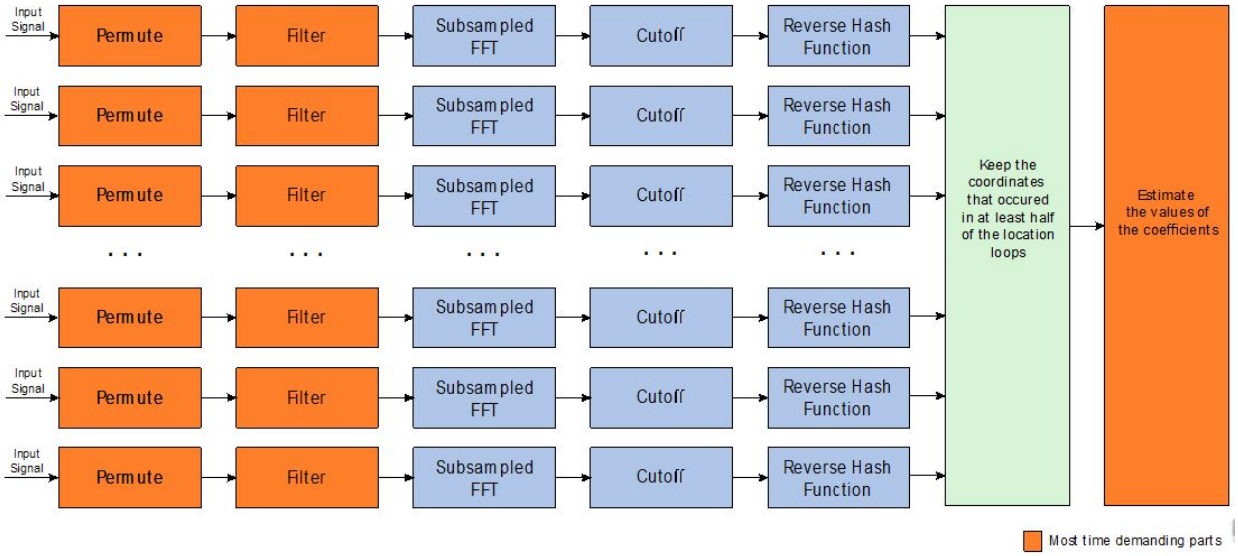
First, the MIT implementation employed C++ STL (Standard Library) and is a sequential only implementation without taking inherent advantage of parallelism inherently present in the algorithm. Some of the data structures used are not thread safe, and certain code structures carry on loop-carried dependencies. This makes parallelizations of the MIT implementation non-trivial.

Second, the MITs 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 and cuFFT are highly optimized to exploit the modern computer architectures.

The UH version is implemented in C instead of C++, and consists of data structures which can easily be used for data parallelism and removed loop carried dependencies. Sparse FFT suffers from 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 MITs 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.

Also a simpler code structure is compiler friendly in C. 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

Figure 5.1: Serial sFFT Profiled stages



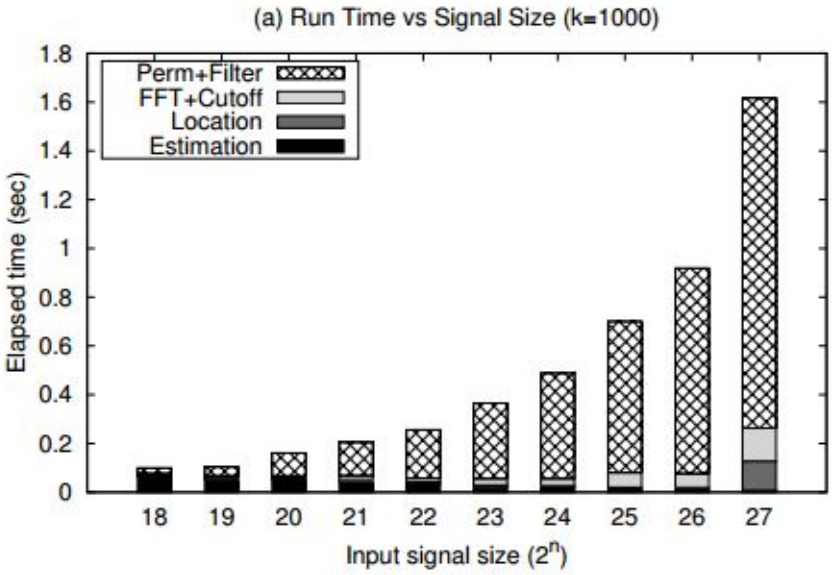
Performance Evaluation

In this subsection, we profile the sequential UH sFFT implementation and study the time distribution and computation of the major stages of the sFFT.

Figure 5.1, shows the various stages, discussed in the previous chapter, of Sparse Fast Fourier Transforms. This is the post profiled image of the serial Sparse Fast Fourier Transform reference algorithm. After profiling the algorithm we notice that the ones highlighted in orange are the most time consuming, computational sections of the algorithm, and are good candidates for parallelization. These are the sections which we will be focusing for parallelization in this thesis.

Figure 5.2 shows the time distribution by varying signal size n from 2^18to2^27 with signal sparsity k fixed to 1000. The time taken on the stages of permutation and filtering, stages 1 and 2, denoted as *perm+filter* in the figure, increases rapidly as the signal size increases. Towards the end of the figure you can even notice that it becomes the most predominant part of the algorithm when the signal size is high. Although if you notice that the *estimation stage* of the algorithm goes down as the signal size

Figure 5.2: Profiling results for main steps of sFFT with sparsity constant and varying signal value



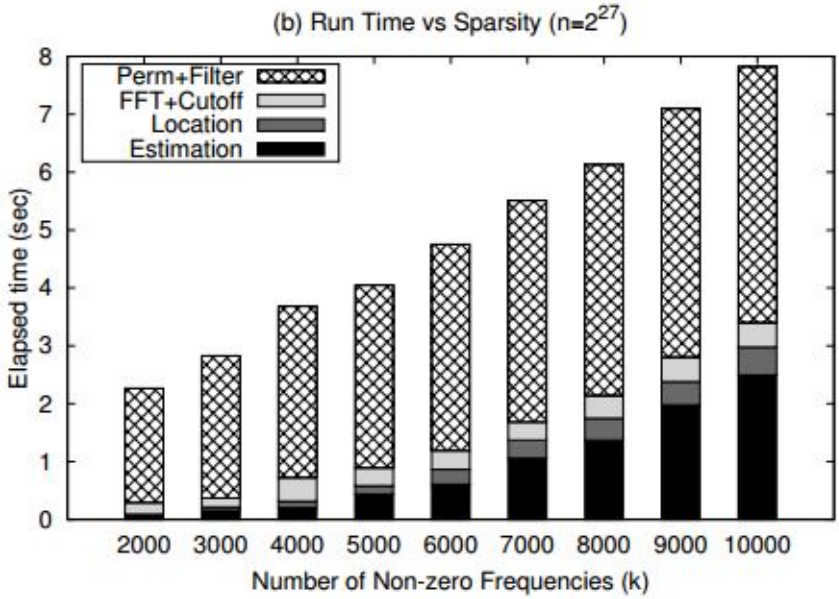
(a) Execution time vs. signal size ($k = 1000$)

increases.

Figure 5.3 shows the time distribution of the sFFT when the sparsity value k changes and the signal size n remains constant. the k sparsity value ranges from 2,000 to 10,000 while the signal size n is fixed at 2^27. Even in this case, the Perm +filter dominates the overall execution time with changing level of sparsity. And as expected the estimation time also increases as the sparsity increases in a signal, this is due to the fact that as relative sparsity increases estimation and recovering the correct coefficients takes more and more time and is difficult inherently.

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

Figure 5.3: Profiling results for main steps of sFFT with signal constant and varying sparsity value



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

due to bulk volume data transfers between host CPUs and accelerators.

In the next few sections, we will discuss ACCsFFT algorithm, a parallel algorithm for computing the sFFT on GPUs, how it was parallelized and the final result of the parallelization. Here we have parallelized only three stages of the algorithm as shown in sFFT profile Figure shown in chapter. These are the most computationally intensive parts of the algorithm. Some of the optimizations are reused from the cusFFT algorithm discussed in chapter 2, like the Fast k-selection algorithm.

5.2 Challenges

For this thesis we will be mainly focusing on achieving maximum performance on a GPU, as well as compare the same code on multicore using OpenACC to Parallel sFFT described in chapter 2. To port an algorithm and achieve optimum performance on an architecture, it requires a lot of understanding of the same. In case of GPU, it requires an understanding of the memory hierarchy and the execution model of the

GPU. It is important to follow the right memory access pattern to the global memory, otherwise it may cause performance bottleneck and hindrance.

One of the challenges that a programmer faces is an effective way of making threads run concurrently. We must find an effective way to partition the workload evenly among the hundreds or even thousands of CUDA cores available on an NVIDIA GPU. The work load should be divided as such that all threads can run independently and there is not a lot of data movements, specially from the host to device and back. If the parallelism is too fine grained , it 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

PCIe bus is not very efficient in transferring data, in fact it is slower as compared to the speed of instructions. So one thing we must always avoid is redundant transfer of data. To achieve a good performance threads should be updating data on the device itself and transfer back to the host when that process is complete. Of course it does depend on the algorithm a lot. But that is the ideal scenario. The data should ideally be present in the GPU global memory in a coalesced form, as a non coalesced memory access can lead to more transaction than necessary.

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.

5.3 Parallelization Stage 1: Analyze

Analyze is the first step towards parallelization. First we need to profile the code to figure out the hot spots of the algorithm, that is the parts of the algorithm

with the most computation. This part of the algorithm is the one which should be a candidate for being parallel, as the algorithm spends most of its time there, so that we can divide the work and make it faster. We have to identify the important loops, functions, and data structures in an algorithm to know where to start and what to offload to the massively parallel GPU.

We also need to know if there are existing libraries which are parallel and highly optimized, available to us. For example, if we already have a Fourier transform library which targets a GPU and parallelizes the computation we do not need to write that part of the code or re-implement it. This way, we can just call the library and used the optimized feature available to us.

We will be using the PGI community edition [16] as our compiler of choice for parallelization with OpenACC, and it comes with several math accelerated libraries inbuilt into it. If we need to use dense or sparse linear algebra, then we would be using cuBLAS or cuSPARSE accelerated library. If FFT is needed then we would be using cuFFT. We should be looking at the existing libraries which we are using and check whether there are already existing parallelized library.

We already did this in the first section of this chapter. The hot spots identified in this algorithm can be seen in Figure 5.1. We use a profiler to check which portion of the code is taking the most computation time in the code. Post that we check that portion of the code to understand when and where the time consumption is happening. This is to help us to identify the accelerating region. Its a good rule of thumb to also check what optimizations the compiler is actually performing on the existing code. Most of the profilers such as *pgprof* will provide us with that information. This will tell us whether the compiler, generated vector SSE code for the loops we are trying to parallelize, the intensity of the loop, how many instructions were prefetched, how many alternate versions were created for the same, etc. All these information will be

useful when performing the next step that is parallelize.

From our profiling data, we came to realize that the most time consuming parts of the Sparse Fast Fourier Transform algorithm is in the $1^{st}, 2^{nd}$ and 3^{rd} stage. That is in the location loop for the first two stage and the estimation loop for the last stage. Details of the stages is explained in Chapter 3. We also come to know from our profiling, that the compiler has auto vectorized many loops and created SSE codes for the same. Also, it has generated multiple versions of the existing loops in the all of these stages. This information will be helpful for the next stage.

Programming Best Practices for Stage 1: Analyze

Do's:

- Always begin with compiling and profiling the code
- Its always good to know how the compiler is currently trying to optimize the existing code
- Multiple profilers is always useful, it give a better perspective
- Check the libraries currently being used, and if the optimized parallel version for your targeted architecture is available. Replace with that
- Identify loops, data structures and function which are good candidate for parallelization

Dont's:

- Never start parallelizing without profiling your code, it may result in a parallel slowdown
- Know the targeted architecture to choose a data structure and directives appropriately

5.4 Parallelization Stage 2: Parallelize

In this stage we begin exploring parallelism, since we now have identified the important portion of the code which we mean to accelerate. Starting with the functions and loops that take a lot of time on a CPU. We now provide hints to the compiler

Table 5.1: sFFT serial code snippet

```
for (int ii=0; ii<loops; ii++){
  for(int i=0; i<B; i++){
        ..
        ..
    for(int j=0; j<round_2; j+=4){
      tmp = ((unsigned)((i_2+j*B)*ai));
      index = tmp & n2_m_1;
      COMPLEX_MULT(index,off3,j);

      index = (unsigned)(tmp + B*2*ai) & n2_m_1;
      COMPLEX_MULT(index,off3,j+2);
    }
```

about parallelism using *pragmas* for C/C++ code.

Table 5.1 shows a snippet from *perm+filter* stage of the UH-sFFT algorithm in *C*. There are multiple nested loops in the algorithm, and is a good candidate for parallelization.

After identification of parts of the algorithm which can be parallelized, we tell the complier to parallelize this section by providing the ***parallel*** pragma to specify that this loop till the end of its scope is ready to be parallelized and it tells the compiler to create a bunch of parallel threads known as ***gangs***, and then distribute the iterations of the loop to those gangs, each one of it runs an iteration of the same.

We don't need to tell the compiler how to divide the loop iteration, the compiler will figure it out on its own based on what it knows about optimizing code in the targeted architecture. It will be different how the compiler targets GPU vs a multicore CPU. By giving the *parallel* clause, we explicitly tell the compiler that this loop is safe to execute.

To avoid explicitly stating to the compiler that this loop is safe, unless we know for sure that it will not cause any parallel scaling problems, we should let the compiler decide whether the loop is safe or not and also let the compiler decide how to parallelize the code. This can be done by the ***kernels*** clause. When we specify the *kernels* clause, like described before, the compiler investigates the section to be parallelized and launches an optimal parallel set of threads based on whether the loop is safe or not to parallelize.

An OpenACC compiler when targeting a GPU, parallelizes portion of the code by launching a bunch of threads in parallel. Each of these thread of execution is called a ***vector***. Vectors run a task specified and offloaded to the GPU by the compiler. When a bunch of *vectors* run simultaneously performing the same task, it causes parallelism. This group of vectors is known as ***worker***. Each one of the *worker* contains 32 *vectors*, depends on the GPU and its architecture also, and perform a parallel task. A ***gang*** is a group of *workers*. Those workers may or may not be working on the same task, or they maybe working on a same task with different data.

The same code shown above, after giving parallel OpenACC clauses looks like this.

The keyword *pragma* specifies that this is a directive pragma and specifies how a compiler should process its input. The *acc* in the directive specifies to the compiler that this is an OpenACC directive.

Programming Best Practices for Stage 2: Parallelize

Do's:

- Begin by adding only parallel clause to see how much speedup is gained as compared to serial implementation of the algorithm

Table 5.2: ACCsFFT after parallelization directives

```
#pragma acc kernels
    for (int ii=0; ii<loops; ii++){
#pragma acc kernels
      for(int i=0; i<B; i++){
            ..
            ..
        for(int j=0; j<round_2; j+=4){
          tmp = ((unsigned)((i_2+j*B)*ai));
          index = tmp & n2_m_1;
          COMPLEX_MULT(index,off3,j);

          index = (unsigned)(tmp + B*2*ai) & n2_m_1;
          COMPLEX_MULT(index,off3,j+2);
        }
```

- Add kernels directive instead of parallel, if you are not sure if it is safe to parallelize
- Add more kernels directive if you want to divide the loop even further
- Add parallel clauses to functions and loops which need to be parallelized by the compiler

Dont's:

- Dont add kernels clause for architectures other than GPU, use parallel clause instead
- Dont add parallel clauses everywhere, be aware of race conditions, synchronizations and parallel slowdown

5.5 Parallelization Stage 3: Optimize

The final stage of parallelization is optimize. Compilers try to optimize as much as possible with the information they have to provide us with a fast code. Generally that is enough, but more often so we have to provide more hints and helps to the compiler to generate even more efficient and optimized code. This stage of parallelization is one of the stages which needs to be repeated constantly to obtain highly parallel and

optimized version of the algorithm in question.

We begin this stage by profiling the parallel code either from stage 2 above, or from the further optimized code which will be generated from this stage. We profile the code again to find the portions of the code which can be optimized further, or are performing poorly due to some constraints, such as data copy, low compute to memory ratio, etc. We also want to know on what the algorithm is spending more time.

After profiling the portion of our code we noticed that the code spends a lot of time transferring data between the host (CPU) and the device (GPU). After every iteration it keeps transferring the data. This is a very common performance limiter. So for further optimization, apart from the parallelization clauses, we need to specify another set of clauses for data movement. A GPU does not share the same memory as that of the CPU. So data needs to be moved from the host CPU to the device GPU at the beginning of the loop scope or at the beginning of the data region. And all the computations can then be done inside the GPU with no or minimal data movement, before finally moving the data back to the host at the end of the data region or end of the loop scope. This can be specified by the *data clause*. Data clause enables us to create, copyin or copyout data from the host to device. Below is the code after we added *data* clauses.

The *data* clauses above contains two parts, *copyin*, where this specifies the data structure with scope to be sent from host to device. And *copyout*, where this specifies the data to be sent back to the host for synchronization. After specifying the *data* clauses, we profile the clause again, a good rule of thumb is to also check the compiler feedback along with the profiled result. This time we notice that it can be further optimized, as the parallelization done by the compiler has poor loop decomposition. The algorithm was parallelized by the compiler and it launched 512 vectors for the loop iterations, which was more than what was needed. This section of the code did

Table 5.3: ACCsFFT Optimized - I

```
#pragma acc data copyin(d_origx[0:2*n], \
d_filter[0:2*filter_size]) \
copyout(d_x_sampt[0:loops*B_2])
  { //beginning of data region

 #pragma acc kernels
    for (int ii=0; ii<loops; ii++){
 #pragma acc kernels
       for(int i=0; i<B; i++){
             ..
             ..
         for(int j=0; j<round_2; j+=4){
           tmp = ((unsigned)((i_2+j*B)*ai));
           index = tmp & n2_m_1;
           COMPLEX_MULT(index,off3,j);

           index = (unsigned)(tmp + B*2*ai) & n2_m_1;
           COMPLEX_MULT(index,off3,j+2);
         }
}//end of data region
```

Table 5.4: ACCsFFT Optimized - II

```
#pragma acc data copyin(d_origx[0:2*n], \
d_filter[0:2*filter_size]) \
copyout(d_x_sampt[0:loops*B_2])
  { //beginning of data region

 #pragma acc kernels loop gang vector(8) independent
    for (int ii=0; ii<loops; ii++){
 #pragma acc kernels loop gang vector(64) independent
      for(int i=0; i<B; i++){
            ..
            ..
        for(int j=0; j<round_2; j+=4){
          tmp = ((unsigned)((i_2+j*B)*ai));
          index = tmp & n2_m_1;
          COMPLEX_MULT(index, off3, j);

          index = (unsigned)(tmp + B*2*ai) & n2_m_1;
          COMPLEX_MULT(index, off3, j+2);
        }
}//end of data region
```

not need that many iterations and was wasting compute resources.

OpenACC provides the developers with complete control over the parallelizations. The *kernels* clause as can be seen above, contains additional information, like the number of vectors to be launched. By specifying *vector*, in the directive, we tell the compiler to generate vectors for the loop. And the number of vectors can be denoted by *vector_length* clause. We still want the compiler to generate *gangs* of *vectors* so we also specify *gang* clause also. Together with the inner loop and the outer loop, a total of 512 $(8 * 64)$ *vectors* are launched. But instead of compute resources being wasted, it is now divided efficiently between the inner and outer loops. Finding the right no of *vectors, workers* and *gangs* to be launched requires trial and error.

We further optimized this section by adding the *independent* clause. It relays information to the compiler that the loop can be parallelized to run independently, cause it to run faster and more efficiently.

Programming Best Practices for Stage 3: Optimize

Do's:

- Always profile the code after every optimizations done in any stage
- Keep an eye for the information provided by the compiler as well as the result of the profiler
- Data clauses are important and unless relying completely on Unified memory, should be specified

Dont's:

- Unless needed, don't vectorize manually. As the compiler can do it for you.
- Dont start optimizing without profiling the code, as it can be detrimental to the parallelization

By adding together all of the above stages, we parallelized part of the *perm+filter* stage of the Sparse FFT algorithm shown above. Rest of the stages of Sparse FFT is done similarly by using the above three stages for parallelization.

5.6 Other Optimizations

Atomicity

Atomicity is an operation in parallel programming, where we explicitly state that no two or more threads or operations can affect a variable at the same time. This is usually to prevent incorrect sequence of operations on a variable and generating wrong data. This is useful when you need to have a synchronized order of operation on a variable. In parallel programming the threads are running at the same time, and there is no fixed time or sequence for them to finish their execution. There is no

Table 5.5: Experimental Setup

SOFTWARE	HARDWARE
CUDA v5.5	NVIDIA K20Xm and K40
CUDA v8.0	NVIDIA K80
PGI v17.4 (Community Edition)	NVIDIA P100
FFTW 3.3.6	Intel Xeon E5 (12 cores)

coherency as a result. This could be detrimental to the operation being executed.

For OpenACC, following 2.5 specs, the atomic operations on a variable can be specified by the keyword *atomic* and specifying the type of atomic operation, eg. read, write, update, etc.

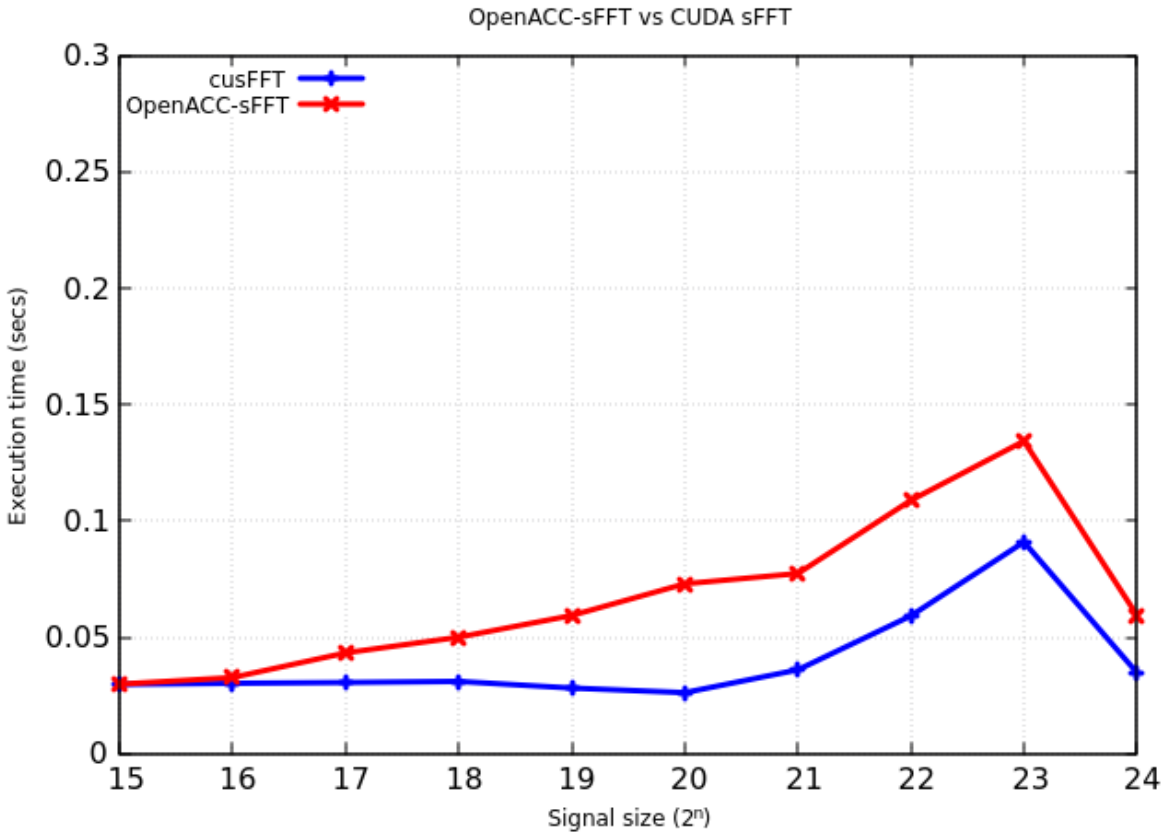
Asynchronous

OpenACC allows asynchronous operations using the *async* keyword and specifying the asynchronous operation. Asynchrony is handling events outside of the main flow of the program and specifying to the compiler that the set of operations in the loop can run independent of each other and dont have to wait. The results generated can then be collected at a later point of the program. Asynchrony helps us to run things in parallel.

5.7 Experimental Setup

Table 5.5 shows the experimental setup used to evaluate the performance of ACCsFFT in this thesis and compare it against the other parallel and serial versions of Sparse Fast Fourier Transform.

Figure 5.4: ACCsFFT vs CUDA Sparse FFT (cusFFT), for a constant K=1000 and N is varied

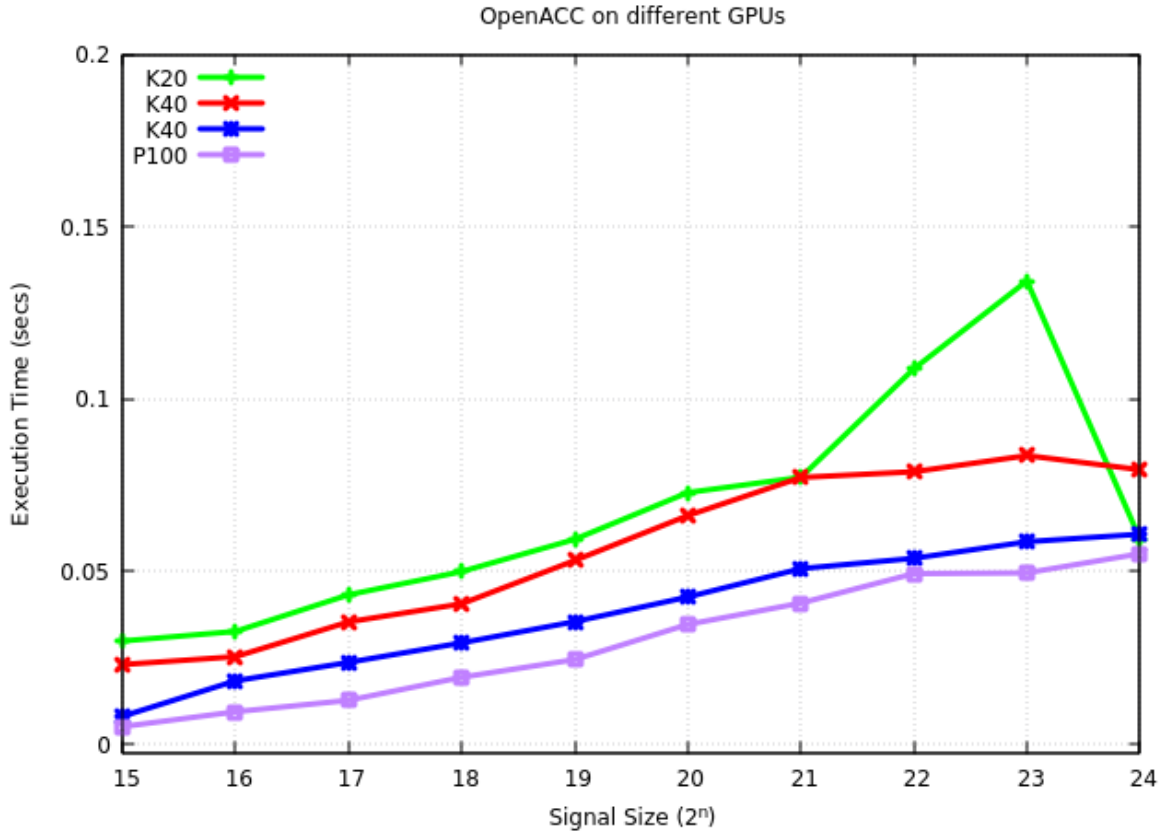


5.8 Performance Evaluation

As shown in the Figure 5.4, we see that ACCsFFT performs relatively close to that of the CUDA Sparse FFT (cusFFT). In most of the experimentation results, we noticed that, ACCsFFT has the same amount of speed up compared to cusFFT in as sparsity decreases and towards the regions where it is really sparse, ACCsFFT performs almost 70% to that of cusFFT.

The main advantage of OpenACC is that, unlike CUDA, the main algorithm has not been modified. And a directive based programming allows us to maintain the

Figure 5.5: ACCsFFT over different NVIDIA GPGPUs, spanning different architectures, for a constant K = 1000



original code, and will work with any compiler, even the ones that do not support OpenACC, serially. The number of lines of code is reduced to almost 10% lesser than that of cusFFT and finally, the same code can be used to work with a Multi-core system and can target other architectures which is available with OpenACC.

The result comparison in the graph is made using CUDA 5.5 and over NVIDIA K20, which is not using NVIDIA Unified memory, more on the same in the later subsections. Figure 5.4 is an extended graph as compared to others, to show the point of divergence between OpenACC and CUDA-sFFT.

Figure 5.6: ACCsFFT vs cusFFT vs sFFT vs PsFFT vs FFTW (threads), for a constant K=1000 and N is varied

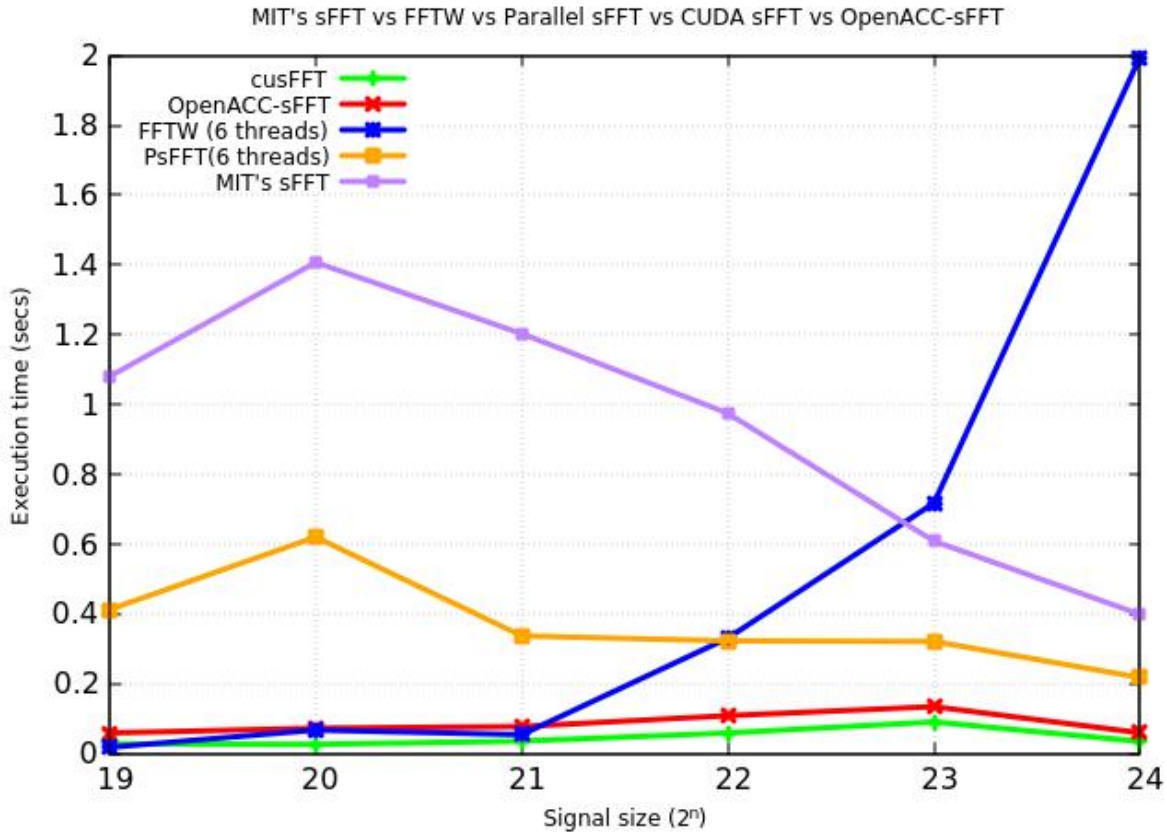


Figure 5.5 shows the result comparison graph between different NVIDIA GPGPUs. These GPGPUs are spanning different architectures and memory management techniques. The aim of the graph is to show a scalable growth of the algorithm, as the hardware improves and more and more CUDA cores are added. As can be seen from the different architectures used and the speed of the algorithm increasing in a constant. Also, helps in showing the reproducibility of the algorithm from old to new generation of hardware. Making it a performance portable algorithm.

The K20 result shown in the figure is using non-unified memory management, and the synchronization are not as optimized when compared to Unified Memory [11].

Figure 5.6 shows the comparison of results between, serial sFFT v2.0 by MIT, CUDA Sparse FFT(cusFFT), Parallel Sparse FFT (PsFFT), FFTW with threads enabled (6 threads) and ACCsFFT. FFTW library performs really well when the sparsity is lower, but as sparsity increases it can be seen to have an exponential growth to the time taken to perform FFT. As explained in chapter 1.

Sparse FFT serial version performs worse as compared to FFTW in the beginning, and then starts picking up, eventually beating FFTW for really sparse input data. Parallel Sparse FFT performs almost 4-5x faster than sFFT. The best among all these, with the least amount of time taken, is by cusFFT, and following closely is ACCsFFT.

This graph is showing the result of the ACCsFFT algorithm in *non-unified memory management*. So as to be able to compare to the original result of CUDA-sFFT algorithm. We could not generate the cusFFT algorithm over newer architectures and is explained in detail in the section below.

5.9 Reproducibility

To compare the speedup of OpenACC, and to find the performance of porting and parallelizing on a GPU, we need a reference parallelized version of the same algorithm on the GPU. We used the CUDA Sparse FFT as the reference algorithm and result.

As can be seen from the Experimental Setup section, we are CUDA v5.5. This is done for reproducing CUDA sFFT result from the original work in [43]. When we tried to use the cusFFT algorithm on newer hardware and software, a NVIDA K80 and CUDA v8.0, we noticed reproducibility issues popping up. When using the newer hardware and software, K80 and CUDA v5.5, which has a better frequency and higher count of CUDA cores and the newer optimized CUDA toolkit, the *cusFFT* started throwing assertion failures. This assertion failure was due to the failure and error in

recovery of the large signal coefficients in the estimation loop from the location loop.

We tried to resolve the situation and used older version of CUDA, CUDA v7.0, here the code worked and was successfully completed without any assertion failures. But the result generated had a high error in recovery of signals and the sFFT results were slower than that of the serial version of the same algorithm.

We could not properly check if the results were fine with an older version of CUDA, CUDA 5.5, used in the paper [43] and that of the newer hardware, due to incompatibility driver issues of both together.

Finally after using the original hardware and software combination, NVIDIA K20Xm and CUDA v5.5, we were successfully able to reproduce the original result. And compare the result to that of the ACCsFFT. The reason as to why *cusFFT* was not successfully reproduced in the newer hardware, was due to thread synchronizations. After investigating heavily, it was evaluated that the original implementation consisted of many thread synchronization techniques which are deprecated now. NVIDIA came with a new memory management model known as NVIDIA Unified Memory model[11] from CUDA 6.0 onwards, which is still used now. This was right after the development and evaluation of the cusFFT over CUDA 5.5. The memory management model deprecated the old synchronization methods created by the algorithm. And these interfered with the memory management model, result i

This brings up an important issue of reproducibility. As newer tools and hardwares are coming up, it is increasingly becoming difficult to reproduce/verify and use original work done by many. Taking this into consideration, the OpenACC code created by us is not using any synchronization methods or directives which has a chance of becoming deprecated in the near possible future. And is developed as such that only small changes may be necessary, should a need for change in the code is required due

to major changes in hardware and software. Also a docker image is being prepared for the same, so the original result and settings can be retrieved by any who wish to expand, reproduce or use the algorithm.

5.10 Summary

ACCsFFT parallelized code may not be faster than CUDA sFFT, but the advantages of using OpenACC programming makes it easier to parallelize application. This also makes the code base easier to maintain, and if further changes need to be made, it can be done much more easily than as compares to a CUDA code, which not only limits multiple heterogeneous architectures, but adds added complexity of a low level language.

The parallel code was created keeping reproducibility in mind, and docker image is created for the same so that it can be reproduced easily. As OpenACC is directive based programming, if changes is needed further down the road due to reproducibility, it can easily be fixed and newer clauses can be added. Even if the code cannot be run on parallel architectures. Regardless of the directives being deprecated or memory models changing in the future, the code will still work serially.

Chapter 6

CONCLUSION AND FUTURE WORK

6.1 Conclusion

The sFFT algorithms are optimal algorithms for computing the Discrete Fourier Transform of input signals with only a few nonzero Fourier coefficients, making it sparse in nature. While there exists reference implementations of these algorithms along with high performance optimized version which are really fast as compared to the state of the art FFT libraries, sFFT is inherently a parallel algorithm, the performance can be improved and allows real world applications to be utilized using the same, if it was made into a parallel algorithms. Now there already exists parallel sFFT algorithms targeting multicore and GPUs separately. But it is riddled with reproducibility issues and lacks a single code base for all parallel implementations.

Guided by this analysis OpenACC parallel programming directive was chosen. And the algorithm was reimplemented by us to make sure we have a performance portable and reproducible algorithm along with having one parallel code for multiple architectures. OpenACC also reduced the number of lines of code as compared to the massive rewrite needed for CUDA by 14%. Making the algorithm easier to manage, and maintainable.

ACCsFFT is the only parallel GPU algorithm available, as CUDA sFFT (cusFFT) is not reproducible in higher architectures and hardwares, as explained in the previous chapter. It improves programmer productivity, by using high-level directive based programming model without compromising on performance or accuracy. Making the ACCsFFT algorithm also maintainable and extensible.

6.2 Future work

In the future, further optimizations will be done in the existing code. As noted in chapter 5, we have focused mainly on three out of the 6 stages in the algorithm. Those were the most time consuming part of the algorithm. The rest of the stages are also good candidates for parallelization and can be optimized. Another optimization which can be done is for the location loop. The n no of location loop can be reduced to 2 inner loops in each filter, Mansour Filter, Gaussian Filter and Permuted Gaussian Filter. This is usually sufficient to reconstruct the signal coefficients with a high probability[36]. This can be used to have fixed set of loops, and perfect for loop enrollment.

The parallel implementation presented in this thesis is based on data parallelism. OpenACC does not fully support task parallelism. As described in chapter 3, Sparse Fast Fourier Transform, has two loop section, the *location loop* and *estimation loop*. Since these are independent tasks, these are good candidates for task parallelism. When location loop finds the location of the large coefficients, we can directly start estimating them, while waiting for more coefficients to be located.

For future work we will target even more architectures such as a Xeon PHI, OpenACC support for the same is expected late 2017, and explore more task parallelism in the existing OpenACC parallel implementation. One mean of doing this is my combining OmpSs programming model[13] along with OpenACC, so that we don't lose portability.

BIBLIOGRAPHY

[1] 3d cfd solver. https://www.openacc.org/success-stories/incomp3d.

[2] The amd core math library (acml). http://developer.amd.com/tools-and-sdks/archive/compute/amd-core-math-library-acml/acml-downloads-resources/.

[3] Cloverleaf. http://developer.amd.com/tools-and-sdks/archive/compute/amd-core-math-library-acml/acml-downloads-resources/.

[4] Intel math kernel library. https://software.intel.com/en-us/mkl.

[5] Lsdalton quantum chemistry. https://www.openacc.org/success-stories/lsdalton.

[6] Maestro and castro. https://www.openacc.org/success-stories/castro-maestro.

[7] Nekcem. https://www.openacc.org/success-stories/nekcem.

[8] Numeca cfd. https://www.openacc.org/success-stories/numeca-international.

[9] Nvidia cuda fast fourier transform (cufft). https://developer.nvidia.com/cufft.

[10] Nvidia pascal white paper. http://www.nvidia.com/object/pascal-architecture-whitepaper.html.

[11] Nvidia unified memory model. https://devblogs.nvidia.com/parallelforall/unified-memory-in-cuda-6.

[12] Nvidias next generation cida compute architecture:. Kepler GK110 white paper.

[13] Ompss programming model. https://pm.bsc.es/ompss.

[14] Openacc. http://www.openacc.org.

[15] Openacc jacobi iteration example. https://devblogs.nvidia.com/parallelforall/openacc-example-part-1/.

[16] Pgi community edition compiler. https://www.pgroup.com/products/community.htm.

[17] Sparse fast fourier transform. https://groups.csail.mit.edu/netmit/sFFT/.

[18] Sparse fast fourier transform library. http://www.spiral.net/software/sfft.html.

[19] Using openacc for irregular computations. on-demand.gputechconf.com/gtc/2017/presentation/s7478-arnov-sinha-using-openacc-to-parallelize-irregular-com pdf.

[20] Haitham Hassanieh Lixin Shi Omid Abari and Ezzeldin Hamed Dina Katabi. Ghz-wide sensing and decoding using the sparse fourier transform.

[21] 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.

[22] Amir Averbuch, Eran Gabber, Boaz Gordissky, and Yoav Medan. A parallel fft on an mimd machine. *Parallel Computing*, 15(1-3):61–74, 1990.

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

[24] Barry A Cipra. The best of the 20th century: Editors name top 10 algorithms. *SIAM news*, 33(4):1–2, 2000.

[25] James W Cooley and John W Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of computation*, 19(90):297–301, 1965.

[26] Jack Dongarra and Francis Sullivan. Guest editors introduction: The top 10 algorithms. *Computing in Science & Engineering*, 2(1):22–23, 2000.

[27] Matteo Frigo and Steven G Johnson. Fftw: An adaptive software architecture for the fft. In *Acoustics, Speech and Signal Processing, 1998. Proceedings of the 1998 IEEE International Conference on*, volume 3, pages 1381–1384. IEEE, 1998.

[28] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation".

[29] 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*, pages 353–364. ACM, 2012.

[30] Haitham Hassanieh, Piotr Indyk, Dina Katabi, and Eric Price. Nearly optimal sparse fourier transform. In *Proceedings of the forty-fourth annual ACM symposium on Theory of computing*, pages 563–578. ACM, 2012.

[31] 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. Society for Industrial and Applied Mathematics, 2012.

[32] Haitham Hassanieh, Lixin Shi, Omid Abari, Ezzeldin Hamed, and Dina Katabi. Ghz-wide sensing and decoding using the sparse fourier transform. In *INFOCOM, 2014 Proceedings IEEE*, pages 2256–2264. IEEE, 2014.

[33] Haitham Hassanieh Fadel Adib Dina Katabi and Piotr Indyk. Faster gps via the sparse fourier transform. *Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology. USA, MobiCom*, 12:22–26, 2012.

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

[35] Jorn Schumacher and Markus Puschel. High-performance sparse fast fourier transforms. In *Signal Processing Systems (SiPS), 2014 IEEE Workshop on*, pages 1–6. IEEE, 2014.

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

[37] Lixin Shi, Ovidiu 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.

[38] Lixin Shi, Haitham Hassanieh, Abe Davis, Dina Katabi, and Fredo Durand. Light field reconstruction using sparsity in the continuous fourier domain. *ACM Transactions on Graphics (TOG)*, 34(1):12, 2014.

[39] Minhyeok Shin and Hanho Lee. A high-speed four-parallel radix-2 4 fft/ifft processor for uwb applications. In *Circuits and Systems, 2008. ISCAS 2008. IEEE International Symposium on*, pages 960–963. IEEE, 2008.

[40] Charles Van Loan. *Computational frameworks for the fast Fourier transform*. SIAM, 1992.

[41] Cheng Wang. *High-Performance Sparse Fourier Transform on Parallel Architectures*. PhD thesis, University of Houston, 2016.

[42] 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*, page 10. ACM, 2013.

[43] Cheng Wang, Sunita Chandrasekaran, and Barbara Chapman. cusfft: A high-performance sparse fast fourier transform algorithm on gpus. In *Parallel and Distributed Processing Symposium, 2016 IEEE International*, pages 963–972. IEEE, 2016.