cu*s*FFT : A High-Performance *Sparse* Fast Fourier Transform Algorithm on GPUs

Cheng Wang[†], Sunita Chandrasekaran[‡], and Barbara Chapman[†]
[†]Department of Computer Science, University of Houston, Houston, TX, USA
Email: {cwang35, chapman}@cs.uh.edu
[‡]Department of Computer and Information Sciences, University of Delaware, Newark, DE, USA
Email: schandra@udel.edu

***Abstract*—The Fast Fourier Transform (FFT) is one of the most important numerical tools widely used in many scientific and engineering applications. The algorithm performs $O(nlogn)$ operations on n input data points in order to calculate only small number of k large coefficients, while the rest of $n - k$ numbers are zero or negligibly small. The algorithm is clearly inefficient, when n points input data lead to only $k \ll n$ non-zero coefficients in the transformed domain. MIT in 2012 developed a sparse FFT (sFFT) algorithm that provides a solution to this problem. In this paper, we explore the challenges and propose effective solutions to efficiently port sFFT to massively parallel processors, such as GPUs, using CUDA. GPGPUs are being increasingly adopted as popular HPC platforms because of their tremendous computing power and remarkable cost efficiency. However, sFFT algorithm is a complex and computationally challenging memory-bound algorithm that is not straightforward to be implemented on GPUs. In this paper, we present some of the optimization strategies such as index coalescing, loop splitting, asynchronous data layout transformation, linear time selection algorithm that are required to compute sFFT on such massively parallel architectures. Our CUDA-based sFFT, cu*s*FFT, performs over 10x faster than the state-of-the-art cuFFT library on GPUs and over 28x faster than the parallel FFTW on multicore CPUs.**

***Keywords*-Numerical algorithms, Fourier Transforms, CUDA, GPGPU**

I. Introduction

Discrete Fourier Transform (DFT) is a fundamental numerical algorithm used in a wide variety of disciplines including audio, communication, wave simulations and cryptography to name a few. It has been of universal importance in scientific and engineering applications for a long time. Fast Fourier Transform (FFT) [1] is the fastest known approach that computes the DFT of an arbitrary n-length signal from/to time (space) to/from frequency (wavenumber) domain, with a computational complexity of $O(nlogn)$.

With the emergence of big data problems, in which the size of the processed data can easily exceed gigabyte or even terabytes, it is challenging to acquire, process and store a sufficient amount of data to compute the FFT in a timely manner. On the other hand, any algorithm for computing the FFT must take time at least proportional to its output size, which is $O(n)$, irrespective of the structure and *sparsity* of the data in the transformed domain. However, in many applications, the output of the FFT is *sparse*, i.e., most of the Fourier coefficients of a signal in frequency domain are negligibly small or equal to zero. Many applications of interest, e.g., audio, image, and video data, seismic signals, bio-medical signals, financial data, social graph data, cognitive radio applications and so on, can fall into the sparse Fourier spectrum. In this case, the FFT is suboptimal because $O(nlogn)$ operations on n input data points lead to only k number of non-zero/significant outputs, where $k \ll n$, and $n - k$ zero/negligibly small coefficients for a k-sparse output signal. This motivates the need for an algorithm that computes the FFT in sub-linear time, i.e, in an amount of time that is considerably smaller than the size of the processed data.

The sparse Fourier Transform (sFFT) [2], [3] provides a precise solution to address this problem. Unlike the FFT whose execution time is proportional to the data size n, the sFFT can use only a considerably small subset of the input data to compute a compressed FFT for only number of the k large coefficients, thus achieves substantially performance improvements. Specifically, the sFFT employs special signal processing filters, notably the Gaussian and Dolph-Chebyshev filters [4], to sample the input signals and bin them into a small set of buckets; each bucket contains potentially only one large Fourier coefficient, of which the location and magnitude can then be determined.

sFFT is a fairly new numerical algorithm and it is of key importance for numerous scientific applications due to so many real-world applications falling into the sparse Fourier spectrum. It is a natural path to improve the performance through parallel computing techniques on state-of-the-art computing architectures. In this paper, we present a high-performance parallel algorithm for computing sFFT on GPUs, namely cu*s*FFT, using CUDA. GPUs are being increasingly adopted as popular high performance computing platforms due to their tremendous computing power and reasonable cost efficiency.

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

We implement the cu*s*FFT using CUDA. We compare the performance of cu*s*FFT with that of the NVIDIA CUDA FFT (cuFFT) [5], which is a highly optimized FFT implementation for NVIDIA GPUs. The experimental results show that the cu*s*FFT is significantly faster than cuFFT for large data sets, being up to 15x speedup. We also evaluate the performance of cu*s*FFT with that of the multicore CPU implementations using OpenMP (PsFFT) developed and published in our prior work [6], as well as against the parallel FFTW, a heavily used FFT library on CPUs. We demonstrate that cu*s*FFT on GPUs is highly competitive, typically more than 4 and up to 29 times faster than PsFFT and FFTW, respectively, on an Intel Sandy Bridge architecture. Finally, we compare the output of the cu*s*FFT with FFTW. We also demonstrate that the good performance of cu*s*FFT is not at the cost of the reduced numerical accuracy.

Our main contributions are as follows:

- **cu*s*FFT:** We design and implement a parallel sFFT algorithm using CUDA, namely cu*s*FFT, on GPUs. To the best of our knowledge, this paper is among the first few efforts taken to port sFFT on GPUs, and the cu*s*FFT is faster than all the previously published work [7].
- **Optimizations:** We propose multiple optimization techniques that can effectively address the performance challenges of the algorithm on GPUs.
- **Performance improvements:** The cu*s*FFT obtains substantial performance improvements over full-size FFT on both GPUs and multicore CPUs without losing numerical accuracy, demonstrating a promising opportunity to replace the FFT primitives in numerous scientific applications.

The rest of the paper is organized as follows: In Section II we present a brief introduction to GPU and CUDA. We also discuss the recent development efforts in both FFT and sparse FFT. Section III briefly describes the serial sFFT algorithm in [2], to give an overview of the algorithm. In Sections IV and V, we discuss the challenges in effectively mapping sFFT on GPUs, and the strategies we employ to find solutions to exploit the immense power of GPUs. We show the experimental results in section VI. Section VII offers concluding remarks.

II. Background

A. Introduction to GPU and CUDA

Before discussing the design and implementation of our cu*s*FFT algorithm on GPUs, we very briefly review the salient details of NVIDIA's current GPU architecture and the CUDA parallel programming model.

Different NVIDIA GPUs offer different hardware configurations. In this paper, we focus on state-of-the-art Kepler GK110 architecture [8] as a testbed. A full Kepler GK110 configuration consists of an array of 15 Streaming Multiprocessors(SM), each of which features 192 single-precision CUDA cores, and each core has fully pipelined floating-point and integer arithmetic logic units. Each SM could access up to 65536 registers. Thread management, including thread creation, scheduling and synchronization are managed entirely by the hardware, so essentially the overhead is minimum. To efficiently manage the large number of threads on the hardware, the SM schedules threads in groups of 32 parallel threads called a *warp*. In the Kepler GK110 architecture, each SM features four warp schedulers, each with dual instruction dispatch units, allowing four warps to be issued and executed concurrently.

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

From the programmer's perspective, a CUDA program invokes parallel functions called *kernels* that execute across many parallel threads. A group of threads are organized as a *block*, and an array of blocks are organized as a *grid*. A block is a group of concurrent threads that can interact with each other through synchronization and per-block shared memory space private to that block. When invoking a kernel, the programmer specifies both the number of blocks and the number of threads per block to be created when launching the kernel. So we can essentially map the CUDA's hierarchy of threads to the hierarchy of processors on a GPU. A GPU executes on one or more kernel grids; an SM executes one or more blocks; and CUDA cores and other execution units in the SM execute thread instructions.

B. FFT Implementations

There are many vendor-specific FFT libraries optimized for specific platforms. They include cuFFT [5] for NVIDIA's GPUs, AMD Core Math Libraries (ACML) [9] for AMD's APUs, and Intel Math Kernel Library (MKL) [10] for Intel processors. FFTW [11] is another widely used open-source FFT library portable to many x86-based architectures. Due to the memory-bound nature of FFT algorithm, its performance heavily depends on the design of memory subsystem and how well it can be exploited. Some of the research work on FFT optimizations for various computer architectures include [12]–[14].

C. Sparse FFT

The first sublinear sparse Fourier algorithm was proposed in [15]. Over the past few years, the topic has been extensively studied, from the algorithmic [2], [3], [16], [17], implementation [6], [18] and application [19] perspectives.

A recent breakthrough research from MIT [2] presented an improved algorithm and reduced the time complexity of sFFT to $O(logn\sqrt{nklogn})$ which is asymptotically faster than its prior studies. Therefore more applications with "denser" signal spectrum could also achieve performance speedups from the sFFT algorithm. There exists few sFFT implementations [7], [18], [20]. However, they are either a sequential prototype implementation only or an implementation that is barely optimized. To our best of knowledge, we are the first effort of a parallel sFFT algorithm on GPUs and we achieve the best performance so far.

III. Serial Sparse FFT

In this section, we present a simplified description of sFFT that is discussed more from a numerical algorithm point of view in [3]. We believe that this will help the reader to understand the complexities in the algorithm from a parallel programming point of view.

The functionality of sFFT is hashing the Fourier coefficients into a small number of buckets. Since the signal is sparse in the frequency domain, each bucket is likely to have only one large coefficient, which can then be located (to find its position) and estimated (to find its value). For the algorithm to be sublinear, the binning of Fourier coefficient has to be done in sublinear time. This is achieved by using Gaussian and Dolph-Chebyshev filters, which are concentrated both in time and frequency domain. The algorithm achieves a runtime of $O(logn\sqrt{nklogn})$, which is faster than FFT for k up to $O(n/logn)$. We breakdown the sFFT into the following steps:

A. Major steps in sFFT

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

Step 1: Random Spectra Permutation The sFFT bins large Fourier coefficients into a small number of buckets by convolving the permuted input signal with a well-designed filter. To guarantee that each bucket receives only one large Fourier coefficient, which can then be accurately located (to find its position) and estimated (to find its value), the algorithm permutes the input signal so that the adjacent Fourier coefficients in frequency domain are well separated. In addition, the distance between the original location and permuted location should be large enough so that adjacent coefficients are not be binned into the same bucket.

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

According to the definition 1, permuted signal in time domain with a shifting factor of τ will lead to phase changes in the frequency domain. This helps in separating the spectra and bin coefficients to the correect buckets.

Step 2: Flat Window Function For the sFFT algorithm to be sublinear, only portions of the input signal can perform the FFT operation. Merely sampling partial points of the input signal, however, is impossible because it leads to spectral leakage, i.e., the discontinuities introduced by splicing the signal will appear as sharp components spread out in the frequency domain. To minimize the effects of spectral leakage, sFFT employs a flat window function working as a filter to bin non-zero Fourier coefficients into a small number of B buckets, where $B = O(\sqrt{nk/logn})$. Specifically, sFFT uses Gaussian and Dolpstartsh-Chebyshev filter, G, because its frequency response is nearly flat inside the pass region and has an exponential tail outside it. So it is unlikely that adjacent Fourier coefficients binned into a particular bucket "leaks" to other.

Step 3: Subsampled FFT After the permutation and filtering steps, the permuted signal gets binned into a smaller number of buckets. Then the reduce-sized FFT is performed. Since we permute the spectra and separate large coefficients, there is a high probability that only one large coefficient potentially exists in each bucket. Next step would be locate and determine its magnitude.

Step 4: Cutoff After step 3, each bucket contains at most one potential large coefficient. In the $k - sparse$ signal spectra, the number of buckets, $B \gg k$, since background noises add to the the signal spectra. We further reduce the amount of computation needed by selecting only the top k coefficients of maximum magnitude. We recover the locations and magnitudes only for those top coefficients selected.

Step 5: Reverse Hash Function for Location Recovery After removing non-significant coefficients in step 4, the binned coefficients have to be reconstructed, by finding its location in the transformed domain and estimating its magnitude. Since steps 1-4 define a hash function that maps each of the coefficients into one of the B buckets. This function has to be reversed by removing the phase change due to the permutation.

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

Step 6: Magnitude Reconstruction In this step, take the sets of locations I' and frequencies $\widehat{x^r_{I'}}$ from location loops, estimate each frequency coefficient $\widehat{x}_i$ as $\widehat{x}_i = median\{x^r_i | r \in \{1, \cdots, L\}\}$. The median is taken in real and imaginary components separately.

IV. GPU Sparse FFT

In this section, we present a parallel algorithm for computing the sparse FFT on GPUs. First we profile the sequential sFFT code and plot the time distribution for the major steps in sFFT. Then we discuss the challenges in identifying the

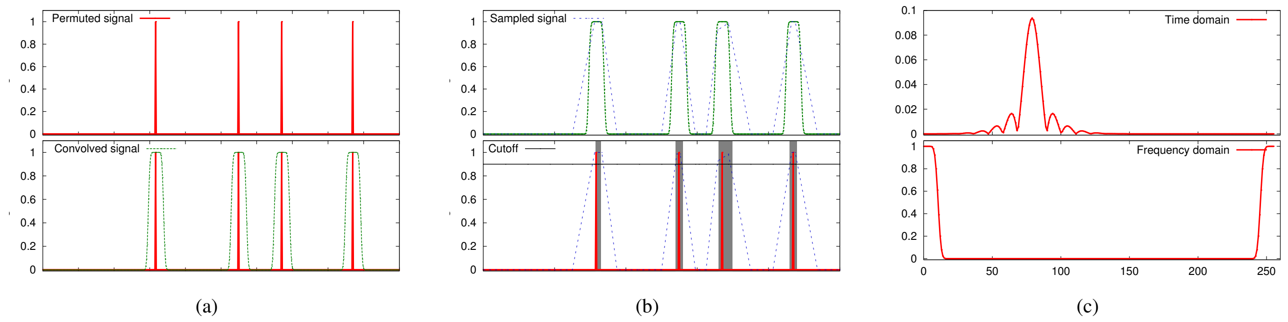


Fig. 1. Example inner loop of the algorithm

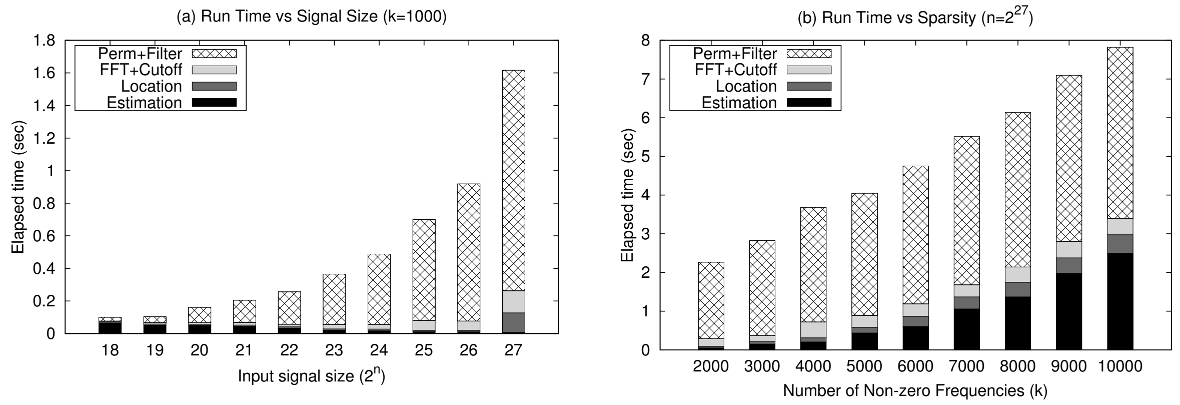


Fig. 2. Profiling results for main steps of sparse FFT

suitable parallelization strategies for porting the sFFT onto GPUs. Finally, we introduce how we address the challenges and port each step of sFFT onto GPUs. The optimizations are discussed in Section V.

A. Profiling Results

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. Figure 2(a) shows the execution time of sFFT with increase in n from 2^{18}(0.26 million points) to 2^{27}(0.13 billion points) with fixed $k = 1000$. We see that the permutation+filter (steps 1&2, i.e. permutation + flat window function) increase dramatically when n increases representing the most time consuming steps. Note that the time taken by estimation loops (step 6) decreases with an increase in the signal size n, which is counter-intuitive. This is because the size of the estimation loop is determined by the *relative* signal sparsity. When we fix the signal sparsity k and increase the signal size n, the relative sparsity actually decreases instead. Figure 2(b) shows the time distribution of sFFT when the signal sparsity k increases with fixed signal size n. It is expected to see that the perm+filter and estimation steps gradually dominate the execution time. In this work, although we focus on optimizing primarily the most time consuming steps, we still port the entire algorithm to GPU to avoid the overhead due to bulk volume of PCIe data transfers between CPU and GPU.

B. GPU Challenges

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

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

C. GPU Sparse FFT

Step 1-2: Random Spectrum Permutation and Filtering The sFFT starts with convolving the permuted input signal with a well-designed filter and binning them into a small number of buckets. Algorithm 1 shows the code snippet of the inner loop in the sequential implementation. Noted that the iterations in the loop is `filter_size` rather than signal size `n` because the tails of the filter is almost zero, so it is not needed to convolve the signal points to a zero-sized filter. Among the several challenges to parallelize the algorithm for

Algorithm 1 Pseudo code for serial permutation and filtering

```
1: Input: $signal[n], filter[filter\_size]$
2: Output: $buckets[B]$
3: procedure PERMFILTER($signal, filter, buckets, B, n,$
   $filter\_size$)
4:     Initialize: $index \leftarrow init\_val$
5:     while $gcd(a, n) \neq 1$ do                    ▷ a,n are co-prime
6:         $a \leftarrow random() \bmod n$
7:     end while
8:     $ai \leftarrow mod\_inverse(a)$              ▷ ai is modular inverse of a
9:     for $i \leftarrow 1,\ B$ do
10:        $buckets[i] \leftarrow 0$                    ▷ initialize buckets
11:    end for
12:                   ▷ Main body of spectrum permutation and filtering
13:    for $i \leftarrow 1,\ filter\_size$ do
14:        $buckets[i \bmod B] += signal[index] \times filter[i]$
15:        $index \leftarrow (index + ai) \bmod n$
16:    end for
17: end procedure
```

GPUs, the first challenge is a loop-carried dependence while updating the variable `index` that prevents the loop from being parallelized. The `index` determines and stores the stride distance while permuting the input signal that depends on its previous value. The second challenge is "hash collision" occurring due to multiple permuted signals binned into the same bucket.

We use an *index mapping* algorithm to address the loop-carried dependence issue. Specifically, an observation from Figure 1 is that the values of the `index` among iterations follow the pattern such that: `init_val, (ai + init_val) % n, (2 * ai + init_val) % n` for loop iteration `i = 0, 1, 2`. Therefore, the idea of the *index mapping* is to directly map the value of `index` to the loop iterator `i` without relying on its prior value, as is shown in Figure 3. As a result, the complexity of obtaining the value of `index[i]` only relies on the loop iterator `i` and can be therefore parallelized.

Regarding the hash collision issue, consider for instance, in Algorithm 1 for loop iterations `i = 0, B, 2B, ...`,

```
int index = init_val;
for(int i=0; i < filter_size; i++) {
  ...
  // Original index calculation
  index = (index + ai) % n;

  // After index mapping
  index = (i * ai + init_val) % n;
  ...
}
```

Fig. 3. Index mapping in signal permutation and filter

signal on these points will be binned into the same bucket, leading to the hash collision. Parallelizing this step is a challenge since on GPUs, thousands of threads may update the same bucket simultaneously. Collisions among the threads will serialize the thread execution, seriously impeding the performance.

In order to avoid collisions, the conventional way to compute histogram on GPUs is to let each thread keep its own private copy of sub-histogram, local reduction is collision free. In the final stage, combine all sub-histograms into a single histogram using atomic operations. This approach, however, has two drawbacks. First of all, the per-thread approach requires replication of size of $B \times N$ bins, where B is number of bins and N is the number of threads. For GPUs with thousands of threads, it is quite obvious that this approach can be extremely expensive in terms of memory space. Furthermore, to minimize the data access latency, most of the implementations [21], [22] utilize on-chip GPU shared memory to store the private sub-histograms. Unfortunately, the size of the shared memory is limited to 48KB even if the state-of-the-art Nvidia Tesla K20x GPUs is considered. This significantly limits the number of bins to 256 or even smaller for a conventional histogram computation [21]. This is a major limitation for sFFT that has number of buckets, $B = \sqrt{nk/logn}$. For instance, for a signal size of $n = 2^{18}$ and $k = 1000$, B could be as large as 3,816; B can get larger for even larger input signal size. As a result, for the elements with complex double data types in the buckets, the maximum number of sub-histograms for the size of 48 KB shared memory is $48 * (1024\ byte/16\ byte)/3816 = 0.8$,(16 bytes is the size of an element in the array, which is complex double type) implying that the shared memory could not even hold a single sub-histogram. Some of the histogram computation approaches in [21], [22] use per-warp or per-block replication instead of the per-thread approach, i.e, privatize the sub-histograms per each warp or per thread block. Since a warp usually consists of 32 threads and a thread block size could reach to as large as 512 for mainstream GPUs, this approach could significantly relieve shared memory pressure. However, the main drawback of this approach is that it still needs atomic operations to update the sub-histograms within a warp or thread block. The usage of atomic operations can be a major bottleneck to good performance.

Another observation with respect to histogram computation is accessing the input data array is predictable, but accessing

Algorithm 2 Pseudo code for GPU permutation and filtering

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

the bucket array is data-dependent (random). The issue with sFFT however, is just the opposite: accessing the input signal is random while accessing the buckets is predictable, as shown in Algorithm 1. Specifically, the number of rounds to access the buckets, `filter_size`, can be divided into `filter_size/B` rounds. The `filter_size` is divisible by `B`, since both of them are powers of 2. As a result, there is no collision within a round, i.e., *intra*-round, and collision occurs only across rounds, i.e., *inter*-round. For instance, updating the buckets when i equals to `(0,B)` is collision-free and collision only occurs when `i = 0, B, 2B, etc..`

Based on this observation, we propose a *loop partition* approach. The basic idea is to partition the original loop iteration into two nested loops, where the outer loop is collision-free and suitable to be mapped to each CUDA thread and the inner loop is processed by one thread. Algorithm 2 shows the pseudo code with loop partition employed for perm+filter step. The outer loop, which has the size of the buckets, is parallelizable and mapped to each CUDA thread, each thread executes the reduction operation for one bucket, independently. Compared to the conventionally GPU histogram reduction approaches, the loop partition approach has mainly three advantages: (a) Bucket replication is not required. Each CUDA thread is responsible for one bucket. (b) Does not require the sub-histograms to merge to a single histogram, as a result we avoid using atomic operations (c) Strikes a balance between fine and coarse grained parallelism, since the size of inner loop is very small, each thread does a small amount of work.

Step 3: B-dimensional cuFFT After binning the spectra into smaller number of B buckets, a B-dimensional FFT is performed. In our GPU algorithm, we simply employ the cuFFT [5] to perform this function. Furthermore, since this step will be repeated for the number of `outer_loops` times, of each calculates the same dimension of cuFFT, we use the batched mode of cuFFT and compute cuFFT only once. By sharing the twiddle factors, the batched cuFFT combines the number of `outer_loops` transforms into one function call, which is much faster than repeatedly calling the cuFFT function.

Step 4: Cutoff After we apply the B-dimensional cuFFT, there are buckets containing potentially large Fourier coefficients in frequency domain. Since the spectra is sparse in frequency, very few of them are potentially large. The objective of the cutoff function is to select the top k largest number of coefficients in the magnitude and store their locations. In the baseline implementation, we apply the sort&select method, as is shown in Algorithm 3. Specifically, we first sort the B buckets in a decreasing order and store the locations of values of the top k largest elements. Here we use the sorting algorithm to perform the cutoff function for several reasons. First of all, sorting is a key building block of many algorithms. It has received a large amount of attention in both sequential and parallel implementations [23]–[25]. For GPU algorithms, the studies in [26]–[28] suggest increase in performance using the right sorting algorithms. In this paper, we use Nvidia's Thrust library [29] to implement the sort&select algorithm. Thrust is a CUDA-based library performing GPU-accelerated sort, scan, transform, and reduction operations. Using Thrust has several benefits: Thrust is Nvidia's official library integrated with state-of-the-art CUDA API. As a result, our implementation depending on the Thrust could achieve better robustness, performance and maintains, compared to relying on any other external third party libraries. Nevertheless, sorting is sub-optimal especially for $k \ll n$. We discuss an alternate technique, in the next section, to fulfill the cutoff function.

Algorithm 3 Pseudo code for sort & select

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

Step 5: Reverse Hash Functions for Location Recovery The steps from 1-4 define a hash function that maps each coefficient into one of the buckets. The "real" locations of large coefficients need to be reversed by eliminating the phase changes caused by spectrum permutation and filtering. Algorithm 4 shows the pseudo code for the location recovery on GPUs. We launch a number of k threads and each thread independently computes the reverse hash function to recover the location of the potential large coefficients. As described in Section III, steps 1-5 repeats a number of *location loops* times, so for each inner loop (determining the location), we increment the occurrences(score) of the location recovered. Once the score of the frequency equals to the threshold, we consider this location of coefficient to be large and add it to `hits`.

Step 6: Magnitude Reconstruction The purpose of this step is that given the locations of large coefficients we need to reconstruct the magnitudes. Since the inner loops are repeated for a number of L loops, given a specific location $r \in I'$, we

Algorithm 4 Pseudo code for GPU location recovery

```
1: Input: $J[k], score[n], a, B$
2: Output: $hits[num\_hits]$
3: procedure LOCRECOVERY($J, hits, score, k, num\_hits, a, n, B$)
4:     for all $tid \leftarrow 1, k$ in parallel do
5:         $my\_J \leftarrow J[tid]$
6:         $low \leftarrow (ceil((my\_J - 0.5) \times n/B) + n) \bmod n$
7:         $high \leftarrow (ceil((my\_J + 0.5) \times n/B) + n) \bmod n$
8:         $loc \leftarrow (low \times a) \bmod n$
9:         for $j \leftarrow low, high$ do
10:            $atomicAdd(score[loc], 1)$
11:            if $score[loc] == threshold$ then
12:                $atomicAdd(num\_hits, 1)$
13:                $hits[num\_hits] \leftarrow loc$
14:            end if
15:            $loc \leftarrow (loc + a) \bmod n$
16:        end for
17:    end for
18: end procedure
```

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

Algorithm 5 Pseudo code for GPU magnitude reconstruction

```
1: Input: $hits[num\_hits], buckets[B], ai, filter[filter\_size]$
2: Output: $loc[num\_hits], val[num\_hits]$
3: procedure MAGRECON()
4:     for all $tid \leftarrow 1, num\_hits$ in parallel do
5:         $my\_hits \leftarrow hits[tid]$
6:         $n\_div\_B \leftarrow n/B$
7:         for $j \leftarrow 1, outer\_loops$ do
8:             $permuted\_loc \leftarrow ai[j] \times my\_hits \bmod n$
9:             $hashed\_to \leftarrow permuted\_loc / n\_div\_B$
10:            $dist \leftarrow permuted\_loc \bmod n\_div\_B$
11:            if $dist > n\_div\_B/2$ then
12:                $hashed\_to \leftarrow (hashed\_to + 1) \bmod B$
13:                $dist \leftarrow dist - n\_div\_B$
14:            end if
15:            $dist \leftarrow (n - dist) \bmod n$
16:            $mag[j] \leftarrow buckets[j][hashed\_to] / filter\_freq[dist]$
17:        end for
18:        $sort(mag, outer\_loops)$
19:        $median \leftarrow (outer\_loops - 1)/2$
20:        $loc[tid] \leftarrow my\_hits$
21:        $val[tid] \leftarrow mag[median]$
22:    end for
23: end procedure
```

V. OPTIMIZATIONS

In Section IV, we had presented some baseline optimizations to parallelize sFFT for GPUs. We also discussed some architectural challenges that the GPUs pose. In this section, we present some improved optimization techniques for the most time consumed portions of the algorithm.

A. Asynchronous Data Layout Transformation

Note that in the first two steps of the sFFT (perm+filter), the algorithm permutes the input signals and bins them into a smaller number of buckets. As shown in Algorithm 2 (line 10), since the `index` is calculated as `index = (i * ai ) % n`, the data reference pattern to the input signal (`signal[index]`) is largely strided. Such irregular memory reference access pattern leads to non-coalesced global memory accesses on GPUs, which creates memory traffic and is a significant bottleneck for achieving good performance.

Prior work [30], [31] relies on static compiler-based techniques that detects the irregularities and reorders the computations at compile time. In sFFT, since the ai is randomly generated, the irregular access pattern is *dynamic*, i.e. it will remain unknown until run time and even change during computation. Such dynamic nature severely limits the compiler techniques to be effectively adopted for the sFFT problem.

To coalesce the memory access, we propose an asynchronous data layout transformation technique that can reorder the data on-the-fly. 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. Figure 4 shows an example illustrating the use of asynchronous data layout transformation technique to hide the overhead. The *remapping kernel* creates a chunk of new array, A', containing the coalesced data. The new order is created based on a desirable mapping technique between threads and data locations, i.e, for loop iteration i, $A'[i] = A[index]$, where A is the original input signal. The second kernel, *execution kernel*, computes the original program but directly accesses the reordered data A'. The chunk size is empirically chosen to be the bucket size B. So after reordering a chunk of B-size data, the second kernel launches a number of B threads that computes the B elements in a batch. Note that there is no data dependence between two remapping kernels (so does execution kernel), so remapping the data chunk δ and $\delta + i$ can happen concurrently. The maximum depth of the concurrency depends on the CUDA compute capability that is different for different GPU architecture. For GK110 used in this paper, the maximum number of concurrent executed kernels is 32.

B. Fast K-selection Algorithm

In step 4 of the algorithm, we applied a cutoff function to select the k largest elements from a set of B buckets. As described in Section IV, we basically sort the entire list and select the k largest elements from the sorted list. However, when the amount of data grows sorting the entire list becomes more and more expensive: $BlogB$ operations with a typical sorting algorithms only leads to top k useful information, where $k \ll B$. Therefore, one could expect to accomplish this task in a time proportional to the data size, i.e., at linear time. Beyond sFFT, there also exists numerous applications requiring this possibility where only the k largest values in a

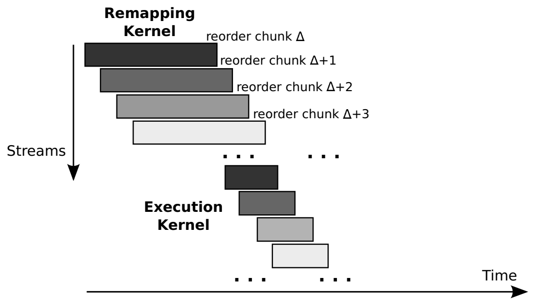


Fig. 4. Asynchronous data layout transformation

Algorithm 6 Pseudo code for GPU fast k-selection algorithm

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

list are retained while the remaining entries are ignored or set to zero [32], [33]. Linear-time fast selection algorithms have received a large amount of attention in sequential and parallel implementations [23], [34]. For GPU algorithms, Alabi *et. al* [35] proposed a *BucketSelect* algorithm. Similar to bucket sorting, the proposed algorithm works well only when the data is uniformly distributed, i.e., the number of elements assigned to each bucket is roughly equal. For the sFFT algorithm, however, only very few of the buckets are large while the rest of them are almost empty.

In this paper, we propose a fast selection algorithm which is simple but effective in sFFT. As shown in Algorithm 6, we assign a number of B threads and each thread processes one element in the buckets. If the value in the buckets is greater than the threshold, the element is chosen and index is stored. It is noted that the choice of the threshold is important for the algorithm. If it is too small, many small coefficients will be picked up and falsely treated as "large". On the other hand, if the threshold is too large, some useful large coefficients will be lost. In this paper, we choose the threshold based on a key observation: only the k out of B buckets are large while rest of them are very small with similar order to amplitude. Consequently, the value of the threshold is chosen to be in the same order of the "small" noise coefficients, of which the value could be empirically obtained from past experience. Most of the time, this approach will yield slightly more than the number of k elements, but this is ignored in the step 4 of the sFFT.

VI. EVALUATION

We evaluate the performance of sparse FFT on GPUs in this section. We compare the performance with cuFFT, which is the fastest implementation for computing the dense FFT with the runtime of $O(nlog(n))$ on NVIDIA GPUs. We also evaluate on both of our implementations: the baseline implementation described in Section IV and the effects of optimizations discussed in Section V. For completeness, we also compare with OpenMP version of sFFT (PsFFT) on multicore CPUs from our prior work [6] and the parallel FFTW, one of the most widely used dense FFT library for multicore CPUs.

A. Experimental Setup

Table I and Table II show the experimental configurations for this evaluation. We evaluated the cu*s*FFT on NVIDIA Kepler K20x GPU and Intel Sandy Bridge E5-2640 processor. The CUDA compiler is NVIDIA's nvcc compiler with version 5.5. The CPU compiler we chose is Intel compiler 13.1 which delivers the best performance on Intel Sandy Bridge architecture. We have also used -O3 optimization flag for both CPU and GPU compilers. The FFTW library used is the latest 3.3 version with multi-threading configured.

B. Experimental Results

In Figure 5(a), we have fixed the sparsity parameter $k = 1000$. We report the runtime of the algorithms compared for different signal sizes n ranging from 2^{18} to 2^{27}. We plot the average execution time for cu*s*FFT baseline version, optimized version, cuFFT, as well as OpenMP sFFT(PsFFT) and FFTW on CPUs. As expected, Figure 5(a) shows that the runtimes of cu*s*FFT and cuFFT are approximately linear in the log scale. However, the slope of the line for cu*s*FFT is less than the slope for cuFFT, which is a result of cu*s*FFTs sub-linear runtime. The figure also shows that for signal sizes $n > 2^{22}$ both baseline and optimized versions of cu*s*FFT are faster than the cuFFT in recovering the exact 1000 non-zero coefficients. The gap becomes bigger when the data size grows, meaning that speedup of cu*s*FFT will be more than than cuFFT.

Figure 5(b) shows that we fix the signal size to $n = 2^{27}$ and evaluate the average execution time of cu*s*FFT while the number of non-zero frequencies k ranges from 100 to 1000. Since both cuFFT and FFTW have a runtime of $O(nlogn)$, they are independent of the number of non-zero frequencies k, as can be seen in Figure 5(b). Nevertheless, as and when the sparsity k increases, the execution time of sFFT increases very slowly.

In Figure 5(c), we plot the speedup of cu*s*FFT over cuFFT. From the figure we can see that the speedup grows very fast with the signal size n, which shows that our cu*s*FFT algorithm benefits a lot on large signal size. For the signal size $n = 2^{27}$, the cu*s*FFT is 15x and over 9x faster than the cuFFT for the optimized and baseline version, respectively.

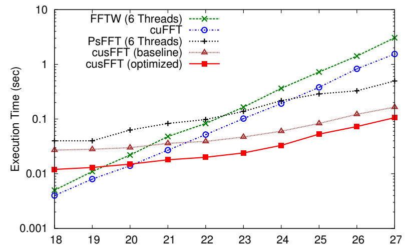
We also compare cu*s*FFT with the parallel FFTW and the OpenMP version of sFFT (PsFFT) we developed before on multicore CPUs. Figure 5(d) shows the speedup of the cu*s*FFT over the parallel FFTW on CPUs. From the figure

TABLE I
GPU TEST-BENCH

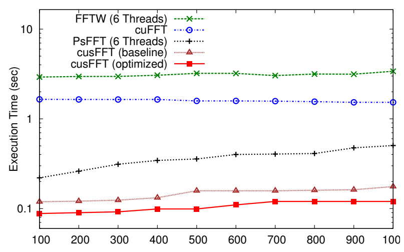
GPU Type	CUDA Capability	CUDA Cores	Processor Clock	Shared Memory	Global Memory	Memory Bandwidth
Tesla K20x	3.5	2688 cores / 14 SMs	732 MHz	64 KB	6 GB	250 GB/s

TABLE II
CPU TEST-BENCH

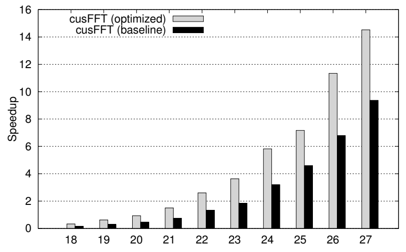
Processor	Architecture	Cores	Processor Clock	L1 Cache	L2 Cache	L3 Cache	DRAM
Intel(R) Xeon(R) CPU E5-2640	Sandy Bridge	6	2.50 GHz	6 x 32 KB D/I	6 x 256 KB	15 MB	64 GB



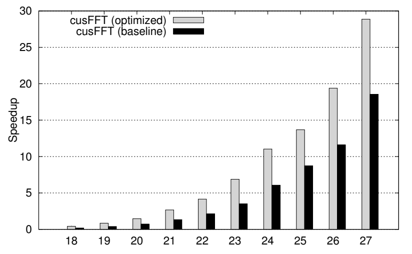
(a) Execution time v.s. Signal size ($k = 1000$)



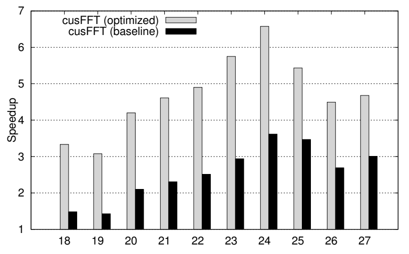
(b) Execution time v.s. Signal sparsity ($n = 2^{27}$)



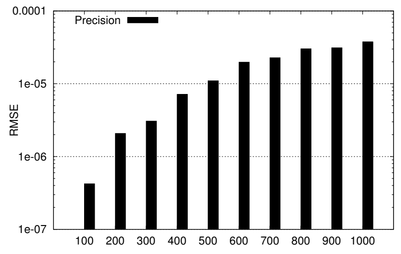
(c) Speedup of cu*s*FFT over cuFFT ($k = 1000$)



(d) Speedup of cu*s*FFT over FFTW (6 threads, $k = 1000$)



(e) Speedup of cu*s*FFT over PsFFT (6 threads, $k = 1000$)



(f) Accuracy ($n = 2^{27}$)

Fig. 5. Execution time and accuracy with varying signal size n and sparsity k

we could see that the optimized cu*s*FFT is 0.5x to 29x faster to parallel FFTW for the signal size ranges from 2^{18} to 2^{27}. In Figure 5(e), we compare the speedup of cu*s*FFT over the PsFFT. It can be seen from the figure that the speedup grows with the signal size, to the peak speedup 6.6x for $n = 2^{24}$, and then slightly goes down when the signal size becomes larger. That is mainly due to the data transfer time between the host and device offsets the performance gains on the GPU computation. But the average speedup of the cu*s*FFT on GPUs is over 4x faster than PsFFT on CPUs.

Figure 5(a) to Figure 5(e) also show that the optimized cu*s*FFT is on average 2x faster to the baseline version, which proofs the effects of optimization strategies discussed in Section V.

Last, we also check that cu*s*FFT's good performance is not at the expense of reduced accuracy. Essentially, our cu*s*FFT ensures the same numerical accuracy to the original sequential algorithm. In addition, we compare the output of cu*s*FFT, $\hat{x}$, with the FFTW, $\hat{y}$, and compute the L_1 error, which is the accumulated error per large coefficient defined as $\frac{1}{k}\sum_{0<i<n} |\hat{x_i} - \hat{y_i}|$. Figure 5(f) plots the average L_1 error for $n = 2^{27}$ and different k values, which shows that the error is extremely small thus preserving the algorithm's accuracy.

VII. Conclusion

The reference implementation of MIT's sFFT proves that the algorithm can be faster than modern FFT libraries. In this paper we improve MIT's implementation to target modern parallel architectures that are proving to be powerful engine for computationally demanding applications. We introduce an efficient parallel algorithm for computing sparse FFT on GPUs. We report the bottlenecks in the algorithm that impedes the performance. We explore several suitable and optimized solutions to tackle these issues and demonstrate that our cusFFT algorithm is over 10x faster than cuFFT for large data size, and over 28x compared to the parallel FFTW on multicore CPUs. As part of the future work, we will continue to explore the performance of the algorithm on other emerging parallel architectures, such as DSPs and Intel Xeon Phi.

Acknowledgment

The authors would like to thank Piotr Indyk and Eric Price from MIT for their discussion and comments. We are grateful to the anonymous reviewers for their constructive comments. This research is supported by Shell International E&P Inc. We very much thank Detlef Hohl from Shell International E&P Inc. for his support and insights on this work.

This research work is also supported by the Office of the Assistant Secretary of Defense for Research and Engineering (OASD(R&E)) under agreement number FA8750-15-2-0119. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the US NSF, or the Office of the Assistant Secretary of Defense for Research and Engineering (OASD(R&E)) or the U.S. Government.

References

[1] B. Cipra, "The Best of the 20^{th} Century: Editors Name Top 10 Algorithms," *SIAM News*, vol. 33, no. 4, p. 1, 2000.

[2] H. Hassanieh, P. Indyk, D. Katabi, and E. Price, "Simple and practical algorithm for sparse Fourier transform," in *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM, 2012, pp. 1183–1194.

[3] ——, "Nearly optimal sparse Fourier transform," in *Proceedings of the 44th symposium on Theory of Computing*. ACM, 2012, pp. 563–578.

[4] P. Lynch, "The dolph-chebyshev window: A simple optimal filter," *Monthly weather review*, vol. 125, no. 4, pp. 655–660, 1997.

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

[6] C. Wang, M. Araya-Polo, S. Chandrasekaran, A. St-Cyr, B. Chapman, and D. Hohl, "Parallel sparse fft," in *Proceedings of the 3rd Workshop on Irregular Applications: Architectures and Algorithms*, ser. IA3 '13. New York, NY, USA: ACM, 2013, pp. 10:1–10:8. [Online]. Available: http://doi.acm.org/10.1145/2535753.2535764

[7] J. Hu, Z. Wang, Q. Qiu, W. Xiao, and D. Lilja, "Sparse Fast Fourier Transform on GPUs and Multi-core CPUs," in *Computer Architecture and High Performance Computing (SBAC-PAD), 2012 IEEE 24th International Symposium on*, 2012, pp. 83–91.

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

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

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

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

[12] N. K. Govindaraju, B. Lloyd, Y. Dotsenko, B. Smith, and J. Manferdelli, "High Performance Discrete Fourier Transforms on Graphics Processors," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. IEEE Press, 2008, p. 2.

[13] A. Nukada, K. Sato, and S. Matsuoka, "Scalable multi-gpu 3-d fft for tsubame 2.0 supercomputer," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 44:1–44:10. [Online]. Available: http://dl.acm.org/citation.cfm?id=2388996.2389056

[14] Y. Chen, X. Cui, and H. Mei, "Large-scale fft on gpu clusters," in *Proceedings of the 24th ACM International Conference on Supercomputing*, ser. ICS '10. New York, NY, USA: ACM, 2010, pp. 315–324. [Online]. Available: http://doi.acm.org/10.1145/1810085.1810128

[15] E. Kushilevitz and Y. Mansour, "Learning decision trees using the fourier spectrum," in *Proceedings of the Twenty-third Annual ACM Symposium on Theory of Computing*, ser. STOC '91. New York, NY, USA: ACM, 1991, pp. 455–464. [Online]. Available: http://doi.acm.org/10.1145/103418.103466

[16] M. Iwen, "Combinatorial sublinear-time fourier algorithms," *Foundations of Computational Mathematics*, vol. 10, no. 3, pp. 303–338, 2010.

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

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

[19] H. Hassanieh, F. Adib, D. Katabi, and P. Indyk, "Faster gps via the sparse fourier transform," in *Proceedings of the 18th Annual International Conference on Mobile Computing and Networking*, ser. Mobicom '12. New York, NY, USA: ACM, 2012, pp. 353–364. [Online]. Available: http://doi.acm.org/10.1145/2348543.2348587

[20] H. Hassanieh, P. Indyk, D. Katabi, and E. Price, "sFFT- Sparse Fast Fourier Transform," http://groups.csail.mit.edu/netmit/sFFT/.

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

[22] J. Gómez-Luna, J. M. González-Linares, J. I. Benavides, and N. Guil, "An Optimized Approach to Histogram Computation on GPU," *Mach. Vision Appl.*, vol. 24, no. 5, pp. 899–908, Jul. 2013. [Online]. Available: http://dx.doi.org/10.1007/s00138-012-0443-3

[23] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein *et al.*, *Introduction to algorithms*. MIT press Cambridge, 2001, vol. 2.

[24] R. Cole, "Parallel merge sort," *SIAM Journal on Computing*, vol. 17, no. 4, pp. 770–785, 1988.

[25] B. Wilkinson and M. Allen, *Parallel programming*. Prentice hall New Jersey, 1999, vol. 999.

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

[27] S. White, N. Verosky, and T. Newhall, "A cuda-mpi hybrid bitonic sorting algorithm for gpu clusters," in *Parallel Processing Workshops (ICPPW), 2012 41st International Conference on*. IEEE, 2012, pp. 588–589.

[28] I. Lashuk, A. Chandramowlishwaran, H. Langston, T.-A. Nguyen, R. Sampath, A. Shringarpure, R. Vuduc, L. Ying, D. Zorin, and G. Biros, "A massively parallel adaptive fast multipole method on heterogeneous architectures," *CommuNiCAtioNs of the ACm*, vol. 55, no. 5, pp. 101–109, 2012.

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

[30] A. H. Hormati, M. Samadi, M. Woh, T. Mudge, and S. Mahlke, "Sponge: portable stream programming on graphics engines," in *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, ser. ASPLOS XVI. New York, NY, USA: ACM, 2011, pp. 381–392. [Online]. Available: http://doi.acm.org/10.1145/1950365.1950409

[31] Y. Yang, P. Xiang, J. Kong, and H. Zhou, "A gpgpu compiler for memory optimization and parallelism management," in *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '10. New York, NY, USA: ACM, 2010, pp. 86–97. [Online]. Available: http://doi.acm.org/10.1145/1806596.1806606

[32] T. Blumensath and M. Davies, "Normalized iterative hard thresholding: Guaranteed stability and performance," *Selected Topics in Signal Processing, IEEE Journal of*, vol. 4, no. 2, pp. 298–309, April 2010.

[33] J. Högbom, "Aperture synthesis with a non-regular distribution of interferometer baselines," *Astron. Astrophys. Suppl*, vol. 15, no. 1974, pp. 417–426, 1974.

[34] M. R. Henzinger, P. Klein, S. Rao, and S. Subramanian, "Faster shortest-path algorithms for planar graphs," *journal of computer and system sciences*, vol. 55, no. 1, pp. 3–23, 1997.

[35] T. Alabi, J. D. Blanchard, B. Gordon, and R. Steinbach, "Fast K-selection algorithms for graphics processing units," *J. Exp. Algorithmics*, vol. 17, pp. 4.2:4.1–4.2:4.29, Oct. 2012. [Online]. Available: http://doi.acm.org/10.1145/2133803.2345676