Input-adaptive Parallel Sparse Fast Fourier Transform for Stream Processing

Shuo Chen
Department of ECE
University of Delaware
Newark, DE, USA
schen@udel.edu

Xiaoming Li
Department of ECE
University of Delaware
Newark, DE, USA
xli@udel.edu

ABSTRACT

Fast Fourier Transform (FFT) is frequently invoked in stream processing, e.g., calculating the spectral representation of audio/video frames, and in many cases the inputs are sparse, i.e., most of the inputs' Fourier coefficients being zero. Many sparse FFT algorithms have been proposed to improve FFT's efficiency when inputs are known to be sparse. However, like their "dense" counterparts, existing sparse FFT implementations are input oblivious in the sense that how the algorithms work is not affected by the value of input. The sparse FFT computation on one frame is exactly the same as the computation on the next frame. This paper improves upon existing sparse FFT algorithms by simultaneously exploiting the input sparsity and the similarity between adjacent inputs in stream processing. Our algorithm detects and takes advantage of the similarity between input samples to automatically design and customize sparse filters that lead to better parallelism and performance. More specifically, we develop an efficient heuristic to detect the similarity between the current input to its predecessor in stream processing, and when it is found to be similar, we novelly use the spectral representation of the predecessor to accelerate the sparse FFT computation on the current input. Given a sparse signal that has only k non-zero Fourier coefficients, our algorithm utilizes sparse approximation by tuning several adaptive filters to efficiently package the non-zero Fourier coefficients into a small number of bins which can then be estimated accurately. Therefore, our algorithm has runtime sub-linear to the input size and gets rid of recursive coefficient estimation, both of which improve parallelism and performance. Furthermore, the new heuristic can detect the discontinuities inside the streams and resumes the input adaptation very quickly. We evaluate our input-adaptive sparse FFT implementation on Intel i7 CPU and three NVIDIA GPUs, i.e., NVIDIA GeForce GTX480, Tesla C2070 and Tesla C2075. Our algorithm is faster than previous FFT implementations both in theory and implementation. For inputs with size $N = 2^{24}$, our parallel implementation outperforms FFTW for k up to 2^{18}, which is an order of magnitude higher than prior sparse algorithms. Furthermore, our input adaptive sparse FFT on Tesla C2075 GPU achieves up to 77.2× and 29.3× speedups over 1-thread and 4-thread FFTW, 10.7×, 6.4×, 5.2× speedups against sFFT 1.0, sFFT 2.0, CUFFT, and 6.9× speedup over our sequential CPU performance, respectively.

ICS'14, June 10–13 2014, Munich, Germany.

http://dx.doi.org/10.1145/2597652.2597669.

Categories and Subject Descriptors

G.1.0 [**General**]: Parallel Algorithms

Keywords

Sparse FFT; Input Adaptive; Stream Processing; Parallel Algorithm

1. INTRODUCTION

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

So far the only feature of input value having been leveraged to improve FFT performance is input sparsity. In real world applications, input signals are frequently sparse, i.e., most of the Fourier coefficients of a signal are very small or equal to zero. If we *know* that an input is sparse, the computational complexity of FFT can be reduced. Sub-linear sparse Fourier algorithm was first proposed in [14], and since then, has been extensively studied in the literatures when applied to various fields [13, 6, 2, 7, 12, 1]. However, their runtimes have large exponents in the polynomials of k and $logN$, and their complex algorithmic structures restrict fast and parallel implementations.

A recent highly-influential work [9] presented an improved algorithm in the runtime of $O(k\sqrt{NlogNlogN})$ that makes it faster than FFT for the sparsity parameter k up to $O(\sqrt{N/logN})$. The follow-up work [10] proposed an algorithm with runtime $O(klogNlog(N/k))$ or even the optimal $O(klogN)$. Just like the "dense" FFT algorithms and the earlier sparse FFT algorithms, the latest sparse FFT algorithms are oblivious to input characteristics, because input

sparsity is assumed but not measured. Furthermore, the sparse FFT algorithms' design is fixed for all inputs of the same size. No part in the algorithms is adapted to other input characteristics.

Here we make an interesting observation. We know that in many real-world FFT applications not only inputs are sparse, but at the same time adjacent inputs are similar. For example, in video compression, two consecutive video frames usually have almost identical sparse distribution in their spectrums, and differ only in the magnitudes of some spectrum coefficients. If the FFT on the prior input has been computed, i.e., its spectrum representation is known, and the current input has a similar sparse distribution to the prior input, can the similarity help computing the sparse FFT on the current input? To answer the question, we need to tell whether an input is similar to its predecessor, and how the knowledge about the predecessor's spectral representation can help. This paper answers the two questions and propose a new sublinear and parallel algorithm for sparse FFT. The main contributions of this paper are: 1) a heuristic to detect the sparsity homogeneity, so that we can know when the FFT computation can be simplified with prior knowledge; and 2) an efficient input adaption process to use the sparsity homogeneity as a template to design the customized filters for subsequent similar inputs, so that the filters lead to less waste of calculation on those zero coefficient bins and can better express parallelism in sparse FFT.

Particularly interesting is that the input sparsity and the input simularity make it easier to parallelize FFT calculation. From a very high point of view, our sparse FFT algorithm applies the custom-designed sparse filters to disperse the sparse Fourier coefficients of inputs into separate bins directly in the spectrum domain. During the dispersion, the calculation on those bins are independent. Therefore it leads our sparse FFT to produce a determinatively correct output, and to be non-iterative with high arithmetic intensity as well. Substantial data parallelism is able to be exploited from our algorithm.

Next we briefly introduce existing sparse FFT algorithms and overview our approach. Then we present how we customize filters based on the sparse template, and how we use the designed filters to reduce the overhead and the number of iterations in the sparse FFT algorithm presented in [9], which our work is based on. Moreover, we show how our input adaption process efficiently and effectively classifies homogeneous and discontinuous signals and automatically recovers from input discontinuity. Finally, we evaluate the performance and accuracy of our input-adaptive sparse FFT with FFTW, CUFFT and the latest sparse FFT implementation on synthetic and real video inputs.

2. BACKGROUND AND OVERVIEW

In this section we overview FFT algorithms, including prior works on sparse Fourier transform, and then introduce our contribution in that context.

2.1 Prior Work on Sparse FFT

A naive discrete Fourier transform of a N-dimensional input series $x(n)$, $n = 0, 1, ..., N-1$ is presented as $Y(d) = \sum_{n=0}^{N-1} x(n) W_N^{nd}$, where $d = 0, 1, ..., N-1$ and N-th primitive root of unity $W_N = e^{-j2\pi/N}$. Fast Fourier transform algorithms recursively decompose a N-dimensional DFT into several smaller DFTs [4], and reduce DFT's operational complexity from $O(N^2)$ into $O(NlogN)$. There are many FFT algorithms, or in other words, different ways to decompose DFT problems. Prime-Factor (Good-Thomas) [8] decomposes a DFT of size $N = N_1 N_2$, where N_1 and N_2 are coprime numbers. Twiddle factor calculation is not included in this algorithm. Additionally, Rader's algorithm [15] and Bluestein's algorithm [3] can factorize a prime-size DFT as convolution.

So far, the runtimes of all FFT algorithms have been proved to be at least proportional to the size of input signal. However, if the output of a FFT is k-sparse, i.e., most of the Fourier coefficients of a signal are very small or equal to zero and only k coefficients are large, sparse Fourier transform is able to reduce the runtime to be only sublinear to the signal size N. Sublinear sparse Fourier algorithm was first proposed in [14], and since then, has been extensively studied in many application fields [13, 6, 2, 7, 12, 1]. All these sparse algorithms have runtimes faster than original FFT for sparse signals. However, their runtimes still have large exponents (larger than 3) in the polynomials of k and $logN$, and their complex algorithmic structures are hard to parallelize.

A highly influential work [9] presented an improved algorithm with the complexity of $O(k\sqrt{NlogN}logN)$ to make it faster than FFT for k up to $O(\sqrt{N/logN})$. The work in [10] followed up with an improved algorithm with runtime $O(klogNlog(N/k))$ or even the optimal $O(klogN)$. Basically, the algorithms permute input with random parameters in time domain to approximate expected permutation in spectral domain for binning the large coefficients. The probability has to be bounded to prevent large coefficients being binned into the same bucket. In addition, these algorithms iterate over passes for estimating coefficients, updating the signal and recursing on the reminder. Because dependency exists between consecutive iterations, the algorithms cannot be fully parallelized. Moreover, the selections of the permutation probability and the filter, which are crucial to the algorithms' performance, are predetermined and are oblivious to input characteristics.

2.2 Our Approach

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

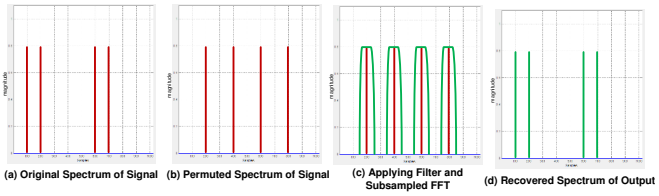

Figure 1: Binning of non-zero Fourier coefficients.

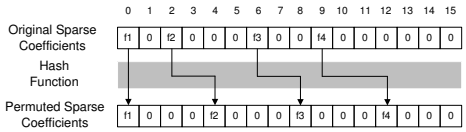

Figure 2: Hash table based permutation.

To help understand the role of spectral template, Figure. 1 illustrates the binning process in our algorithm. Large Fourier coefficients are binned into a small number of buckets and each bucket is designed to have only one large coefficient whose location and magnitude can be then determined. The bucket is represented by an n-dimensional filter D, that is concentrated both in time and frequency [9, 10], to ensure the runtime to be sublinear to N. What binning does is essentially to convolute a permuted input signal with a well-selected filter in spectral domain. During the binning, each bucket receives only the frequencies in a narrow range corresponding to the length of filter D's pass region, and pass regions of different buckets are disjoint. The prerequisite of a pass region having only one large coefficient is to make it possible to evenly space all adjacent coefficients in spectrum later. The information of likely coefficient locations used in the filter tuning is derived from the sparsity template. Particularly, to achieve the expected equal distanced permutation, we make use of a hash table structure to directly permute coefficients in the spectral domain. Fig. 2 shows the example of our hash table based permutation in spectral domain, where f_i denotes non-zero Fourier coefficients and the numbers shown above represent locations of the coefficients.

Note that we do not permute input in time domain to approximate the equal distanced permutation with a certain probability bound, but rather directly permute in spectral domain. In addition, each bucket certainly bins only one large coefficient. Therefore our sparse FFT algorithm is always capable of producing a determinative as well as correct output. Once each bucket bins only one large coefficient, we also need to identify its magnitudes and locations. Instead of recovering the isolated coefficients using linear phase estimation [10], we can easily look up the hash table reversely to identify binned coefficients. As a result, our algorithm has the runtime at most $O(k^2 logN)$.

Furthermore, if the distances of all adjacent frequencies are larger than the minimum length of filter's pass region, we can reduce the number of permutations and therefore further improve the runtime to $O(klogNlog(klogN))$.

Another desirable trait of our algorithm, compared with prior sparse FFT algorithms, is its capability to be fully parallelized. Since our algorithm is non-iterative with high arithmetic intensity, substantial data parallelism can be exploited from the algorithm. The graphical processing units (GPUs) are utilized for the well-suited data parallel computations. In this work we parallelize three main steps in our algorithm on GPU: input permutation, subsampled FFT and coefficient estimation.

3. INPUT ADAPTIVE SPARSE FFT

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

3.1 General Input-Adaptive Sparse FFT

3.1.1 Notations and Assumptions

For a time-domain input signal x with size N (assuming N is an integer power of 2), its DFT is $\hat{x}$. The sparsity parameter of input, k, is defined as the number of non-zero Fourier coefficients in $\hat{x}$. In addition, $[q]$ refers to the set of indices $\{0, ..., q-1\}$. $supp(x)$ refers to the support of vector x, i.e. the set of non-zero coordinates, and $|supp(x)|$ denotes the number of non-zero coordinates of x. Finally, this initial version of algorithm assumes input homogeneity, that is, the locations loc_j of non-zero Fourier coefficients can be estimated from similar prior inputs, where $j \in [k]$. The location template is computed only once for a sequence of signal frames that are similar to each other. The computing of the template by our input-adaptive mechanism is described in section 3.5.

When we find that homogeneity is broken, our algorithm re-calculates the template and restarts the input-adaptation.

3.1.2 Hashing Permutation of Spectrum

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

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

Next we want to determine the shifting distance s between each original location loc and its permuted position idx to be $s_j = idx_j - loc_j, j \in [k]$. Since shifting one time moves all non-zero Fourier coefficients with a constant factor, so in the worst case, only one Fourier coefficient will be permuted

into the right equidistant location. In addition, since we need to permute in total k non-zero coefficients, at most k-time shiftings have to be performed to permute all the coefficients into their equal distanced positions.

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

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

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

3.1.3 Flat Window Functions

In this paper, the method of constructing a flat window function is same as that used in [9]. The concept of flat window function is derived from standard window function in digital signal processing. Since window functions work as filters to bin non-zero Fourier coefficients into a small number of buckets, the pass region of filter is expected to be as flat as possible. Therefore, our filter is constructed by having a standard window function convoluted with a box-car filter [9]. Moreover, we want the filter to have a good performance by making it to have fast attenuation in stopband.

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

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

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

Fig.3 shows an example of Gaussian, Kaiser and Dolph-Chebyshev flat window functions. Note that the spectrum of our filters D is nearly flat along the pass region and has an exponential tail outside it. It means that leakage from frequencies in other buckets can be negligible. By comparing the properties of the three window functions, Dolph-Chebyshev window is the optimal one for us due to its flat pass region as well as its quick and deep attenuation in stopband.

3.1.4 Subsampled FFT

The coefficients binning process convolutes input spectrum with flat window function. In our algorithm, this

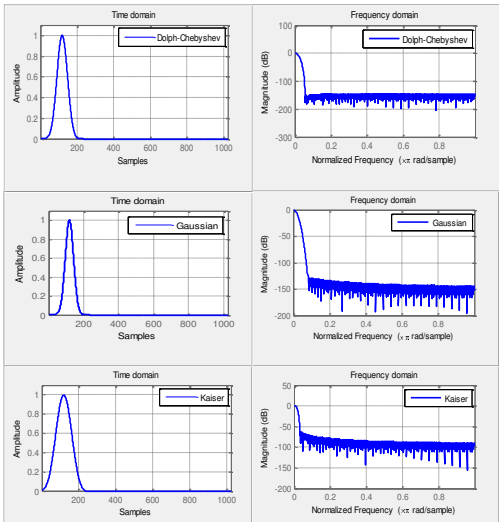

Figure 3: Dolph-Chebyshev, Gaussian and Kaiser flat window functions for $N = 1024$.

convolution is actually performed in time domain by first multiplying input with filter and then computing its subsampled FFT. Suppose we have one N-dimensional complex input series x with sparsity parameter k for its Fourier coefficients, we define a subsampled FFT as $\hat{y}_i = \hat{x}_{iN/k}$ where $i \in [k]$ and N can be divisible by k. The FFT subsampling expects the locations of Fourier coefficients in spectrum domain have been equally spaced. The proof of k-dimensional subsampled FFT has been shown in [9] and the time cost is in $O(|supp(x)| + klogk)$.

3.1.5 Reverse Hash Function for Location Recovery

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

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

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

3.1.6 Basic Algorithm

Combining the aforementioned steps, we can piece together a baseline sparse FFT algorithm. Note that up to this point, we have not introduced input adaptability, yet. Assuming we have a Fourier location template with k known Fourier locations loc and a precomputed filter D,

1. For $j = 0, 1, 2, ..., k-1$, where $j \in [k]$, compute hash indices $idx_j = H(j)$ of permuted coefficients, and determine shifting factor $s_j = idx_j - loc_j$.
2. Compute $y = D \cdot P_s(x)$, therefore $|supp(y)| = |s| \times |supp(D)| = O(|s|\frac{k}{\alpha}log(\frac{1}{\delta}))$. We set $\delta = \frac{1}{4N^2V}$, where V is the upperbound value of Fourier coefficients and $V \leq N$.
3. Compute $u_i = \sum_{l=0}^{\frac{|supp(y)|}{k}-1} y_{i|y|+lk}$ where $i \in [k]$.
4. Compute k-dimensional subsampled FFT $\hat{u}_i$ and make $\hat{z}_{idx} = \hat{u}_i$, where $i \in [k]$.
5. Location recovery for $\hat{z}_{idx}$ by computing reverse hash function to produce $rec = H_r(idx)$ and finally output $\hat{z}_{loc(rec)}$.

The computational complexity The computational complexity of our general sparse FFT algorithm can be derived

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

3.2 Optimized Input-Adaptive Sparse FFT

In this section we introduce several transformations of our algorithm that may improve performance and facilitate parallelization. The complexity of the general adaptive sparse Fourier algorithm is asymptotic to $O(|s|\frac{k}{\alpha}log(\frac{1}{\delta}))$ if initially no adjacent Fourier coefficients are equally distanced. However, if the number of permutations can be reduced, then $|s|$ will be decreased. In fact, it is unnecessary to permute all the Fourier locations to make them equidistant between each other. Since binning the sparse Fourier coefficients is a process of convoluting permuted input spectrum with a customized filter, it is guaranteed that if length of filter's pass region ϵ is less than or equal to half of the shortest distance $dist_{min}$ among all the adjacent locations of non-zero coefficients, i.e. $\epsilon <= dist_{min}/2$, then we don't need to permute all coefficients before we do a FFT. Moreover, in this way, we can get rid of aliasing distortions during the binning and each pass region essentially receives only one large coefficient. If we do not do this, aliasing error occurs and we have to permute all spectral samples.

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

3.2.1 Input-Adaptive Shifting

There are two ways to shift non-zero coefficients. 1) If $k <= \frac{N}{klogN}$, we shift the first unevaluated non-zero coefficient into the truncated region each time; or 2) If $\frac{N}{klogN} < k$, we shift the unevaluated non-zero coefficient by a constant factor $klogN$ each time;

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

3.2.2 Optimized Algorithm

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

1. Apply filter to input signal x:

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

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

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

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

3.3 Hybrid Input-Adaptive Sparse FFT

It is clear from the complexity analysis of our general and optimized sparse FFT algorithms that the two algorithm versions are best suited for different input characteristics. That is, the "optimized" version does not perform better than the general version on all cases. We hybridize the two approaches by at runtime selecting the most appropriate version based on input characteristics.

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

Therefore, the two algorithm versions are selected based on the following heuristic:

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

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

The cost for the selecting process is only $O(k)$, which can be neglected compared with the runtime of either the general version or the optimized version.

3.4 Parallel Input-Adaptive Sparse FFT

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

We use the general sparse FFT implementation to demonstrate how we parallelize our input-adaptive sparse FFT algorithms. The parallelization of the optimized version is similar. Data parallelism exists in the hashed index computation, input filtering and permuting, subsampling FFT, and location recovery. Therefore, we implement a GPU computational *kernel* for each step. First of all, the kernel $HashFunc()$ is responsible to compute hashed indices of permuted coefficients and to determine shift factors. The loop of size k is decomposed into k threads and each thread concurrently works at each index j in the algorithm. In addition, the kernel $Perm()$ with $k^2 logN$ threads is used to apply filter and permutation to input. Each thread multiplies the filter and the shifting factor with input for one element. We parallelize the subsampling of input in kernel $Subsample()$ with total k threads before launching the FFT kernel $TunedFFT()$. Finally we obtain output from location estimation kernel $Recover()$ with k threads parallelizing the loop of algorithm.

3.4.1 Tuned GPU based FFT Library

Our GPU kernel decomposes a 1D FFT of size $N = N_1 \times N_2$ into multi-dimensions N_1 and N_2. Therefore it enables the exploitation of more parallelism for parallel FFT implementation on GPU architectures. All N_1 dimensional 1D FFTs are first calculated in parallel across N_2 dimension. If the size of N_1 is still large after decomposition, we would further decompose each $N_1 = N_{11} \times N_{12}$ sized 1D FFT into two dimensional FFTs with smaller sizes N_{11} and N_{12}, respectively. On GPU, the device memory has much higher latency and lower bandwidth than the on-chip memory. Therefore, shared memory is utilized to increase device memory bandwidth. $N_1W \times N_{11} \times N_{12}$ sized shared memory needs to be allocated, where N_1W is chosen to be 16 for half-warp of threads to enable coalesced access to device memory. The number of threads in each block, for both N_{11} and N_{12}-step FFTs, is therefore $N_1W \times max(N_{11}, N_{12})$ to realize maximum data parallelism on GPU. To calculate each N_1-step 1D FFT, a size N_{11} FFT is executed to load data from global memory into shared memory for each block. Next, all threads in each block are synchronized before data in shared memory is reused by the N_{12}-step FFT and subsequently written back to global memory. Experiment tests show that such shared memory technique effectively hides global memory latency and increases data reuse, both contributing to the performance on GPU. Fig.4 shows the working flow of our GPU based parallelization.

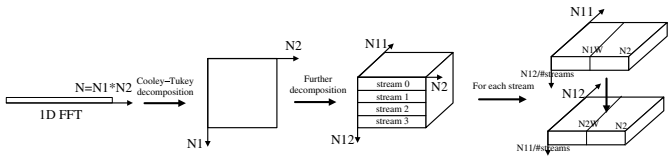

Figure 4: Working flow of GPU parallelization.

3.5 Input Adaption Heuristics and Process

In this section, we describe our overall input adaption process. We first detail when and how Fourier location templates are generated. Then, we elaborate how our sparse FFT adapts to inputs with homogeneous and discontinuous characteristics.

3.5.1 Scenario Establishment

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

3.5.2 Input Adaption for Homogeneous Signals

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

In the beginning, the input signal $x_{i,T0}$ is captured in a video frame at the initial time slot T_0. We generate the Fourier template Tmp once by calculating $x_{i,T0}$'s Fourier transform $\hat{x}_{j,T0}$ using a dense FFT if $\hat{x}$ is not sparse or using a sparse FFT if $\hat{x}$ is sparse. The Fourier template Tmp containing all the locations of non-zero Fourier coefficients and their order for $\hat{x}_i$ at T_0. The cost includes runtime of a full FFT, i.e. $O(FFT)$, plus the time to identify sparse Fourier locations, i.e. $O(N)$. Next, we need to compute Fourier transform for x_{i-m_1} at time T_1. Since the time-shifted x_{i-m_1} corresponds to $\hat{x}_j e^{-b2\pi m_1 j/N}$ in spectral domain, where $b = \sqrt{-1}$, hence the locations of non-zero Fourier coefficients in $\hat{x}_{j,T1}$ is same as those in $\hat{x}_{j,T0}$, but only the coefficients differ. As a consequence, the Fourier template Tmp is used to compute sparse FFT for $x_{i-m_1,T1}$ at T_1 and for $x_{i-m_t,Tt}$ in the following time slots T_t. Therefore, we only compute dense FFT once on each segment of homogeneous inputs. Except for the first input in the segment, our adaptive sparse FFT is used.

3.5.3 Input Adaption for Discontinuous Signals

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

Table 1: Sub-steps parameters of input adaption

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

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

Subsequently, we need to quantitively define discontinuity. We use the first-level deviation dev_i by comparing the outputs of our sparse FFT with that of sampled FFT. We further conduct a second-level deviation metric $dev\_2nd(dev_i, dev_{i-1})$ to determine the relative degree of difference between dev_i for current signal x_i and dev_{i-1} for signal x_{i-1} in the prior frame $i-1$. From the evaluation in section 4.2, if discontinuity occurs at frame i, $dev\_2nd(dev_i, dev_{i-1})$ at the discontinuous point will be much larger than $dev\_2nd(dev_{i-m}, dev_{i-m-1})$, $1 \leq m < \#frames$, of the previously homogeneous cases, and can be accurately separated. Since we only need compute the second-level metric once, the cost for the entire detection is $O(\text{our\_sparse\_fft}) + klogN + O(1)$ and is asymptotic to only $O(\text{our\_sparse\_fft})$. Finally, after the discontinuity has been detected, our algorithm re-calculates the template and resumes the input-adaptation.

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

4. EXPERIMENTAL EVALUATION

In this section we evaluate our input-adaptive sparse FFT implementation and its performance in a real-world-like application. All inputs are double-precision. The evaluation is conducted on three heterogeneous computer configurations. The sequential version is implemented on the Intel i7 920 CPU and the parallel implementation is tuned for three different NVIDIA GPUs, i.e. GeForce GTX480, Tesla C2070 and Tesla C2075. For both sequential and parallel versions, we evaluate our general and optimized sparse FFT approaches, and compare them against four highly-influential FFT libraries: 1) FFTW 3.3.3 [5], the latest FFTW which is one of the most efficient implementations of dense FFT. In FFTW, Streaming Single Instruction Multiple Data Extensions (SSE) on Intel CPU is enabled for better performance. Furthermore, we use two levels of optimizations in FFTW, i.e. ESTIMATE (a basic optimization level marked as 'FFTW' in the plots) and MEASURE (a more aggressively optimized version marked as 'FFTW OPT'). The 4-thread enabled FFTW is used in evaluation of the parallel version. 2) sFFT 1.0 and 2.0 [9], which is one of the fastest sublinear algorithms of sparse FFT. 3) AAFFT 0.9 [11], which is another recent sublinear algorithm with fast empirical runtime. 4) CUFFT 3.2, the NVIDIA CUDA FFT library for GPU-based dense FFT implementation. The GPU performance reported in this paper includes the time for both computation and data transferring between host and device. The configurations of GPUs and CPU are summarized in Table 1.

Table 2: Configurations of GPUs and CPU

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

4.1 Input-Adaptive Sparse FFT

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

4.1.1 Sequential Input-Adaptive Sparse FFT

Fig. 5 and Fig. 6 show our sequential sparse FFT on an Intel i7 CPU. The basic version of our library is labeled as flag 'General', and the average and the best case of our optimized version is marked as 'OPT-AVG' and 'OPT-BEST', respectively. Specifically, our optimized version performs best when all large coefficients reside in only one truncated region of length $O(klogN)$ so that no shifting is needed. The 'OPT-AVG' case instead runs on a random input for 10 times and then takes an average.

In Fig. 5, we fix $k = 64$ but vary N. The running time of FFTW is linear in the signal size N and sFFT 1.0/2.0 shows approximately linear in N when $N > 2^{20}$. However, our sparse FFT's performance appears almost constant as the signal size increases, which reflects the sub-linear complexity of our algorithm. In addition, AAFFT 0.9 is stable over different N but its performance is lower than ours and sFFT. Overall, our approach outperforms over sFFT, FFTW and AAFFT. Our general version, the average case and the optimal case of our optimized library become faster than FFTW with $N \geq 2^{18}$, $N \geq 2^{17}$, and $N \geq 2^{14}$, respectively, while sFFT and AAFFT achieve this goal with much larger input sizes, i.e., $N \geq 2^{19}$ and $N \geq 2^{24}$, respectively.

In Fig. 6, we fix $N = 2^{24}$ but change k. FFTW shows invariance in performance since its complexity is $O(NlogN)$ which is independent to k. Our general sparse FFT main-

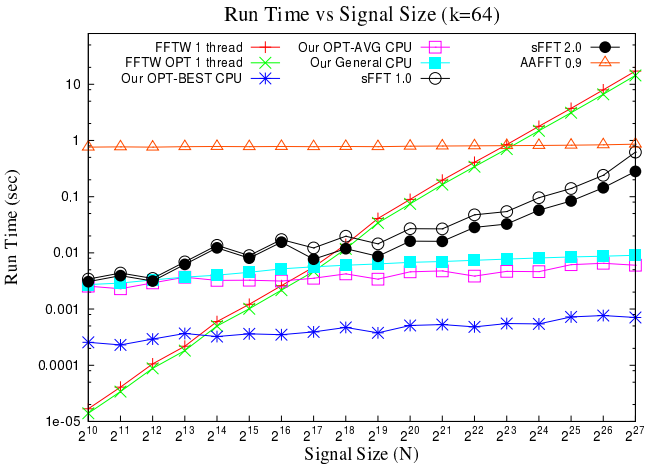

Figure 5: Sequential performance vs. signal size.

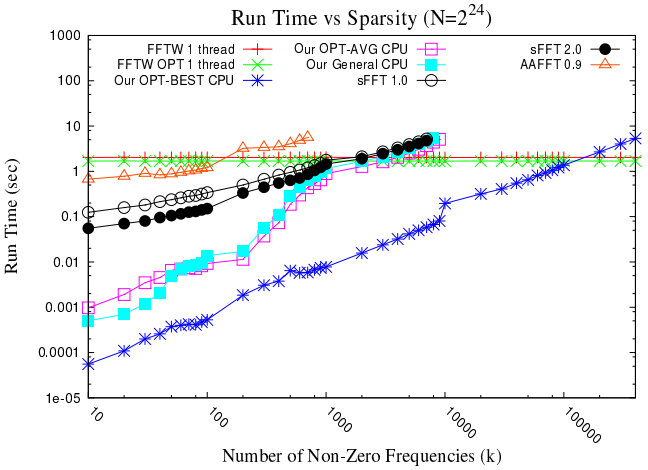

Figure 6: Sequential performance vs. sparsity.

tains its performance superiority over FFTW for k up to 3000 and 2000, respectively. Our optimal version shows a faster performance than FFTW before k reaches 100000. However, sFFT 1.0, sFFT 2.0 and AAFFT 0.9 are faster than basic FFTW only when k is less than 900, 1000 and 100. Therefore, our approach extends the range of input sparsity parameter k in which a sparse FFT outperforms a dense FFT, the range being an indicative and widely used efficiency metric when evaluating a sparse FFT algorithm. Furthermore, our library performs better than all other compared FFT libraries on average.

4.1.2 Parallel Input-Adaptive Sparse FFT

Fig. 7 shows the parallel versions of our sparse FFT on three GPUs. Since there is no parallel version of either sFFT or AAFFT, we only compare to the 4-thread FFTW and CUFFT. In Fig. 7, we fix $k = 64$ and vary N. Both 4-thread FFTW and CUFFT are linear in the signal size N, however, our parallel performance appears constant as N increases. Our general version implementations on the three GPUs are faster than the 1-thread FFTW, the 4-thread FFTW and CUFFT when $N \geq 2^{14}$, $N \geq 2^{16}$ and $N \geq 2^{17}$. Furthermore, the optimal performance of our parallel case is faster

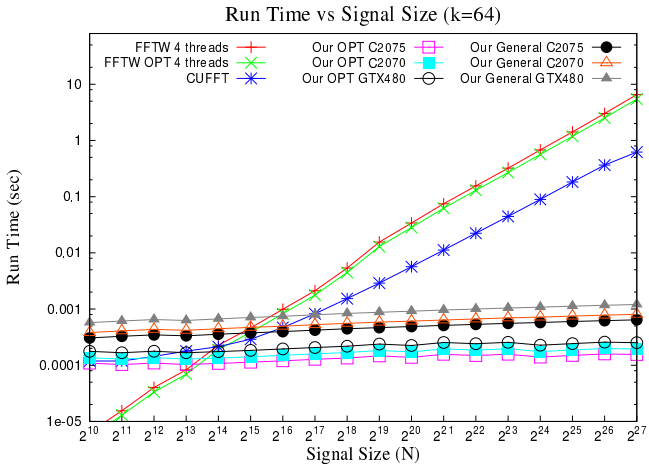

Figure 7: Parallel performance vs. signal size.

than 1-thread FFTW, 4-thread FFTW and CUFFT when $N \geq 2^{12}$, $N \geq 2^{14}$ and $N \geq 2^{14}$.

In Fig. 8, we fix $N = 2^{24}$ and change k. Specifically, our parallel performance of basic version on GTX480, Tesla C2070 and C2075 has a runtime faster than 1-thread FFTW for k up to 30000, 40000, 50000, faster than 4-thread FFTW before k reaches to 20000, 30000, 30000, and faster than CUFFT for k less than 6000, 8000, 9000, respectively. Additionally, the optimal performance of our parallel case is better than 1-thread FFTW, 4-thread FFTW, CUFFT for k up to 500000, 100000, 40000, respectively.

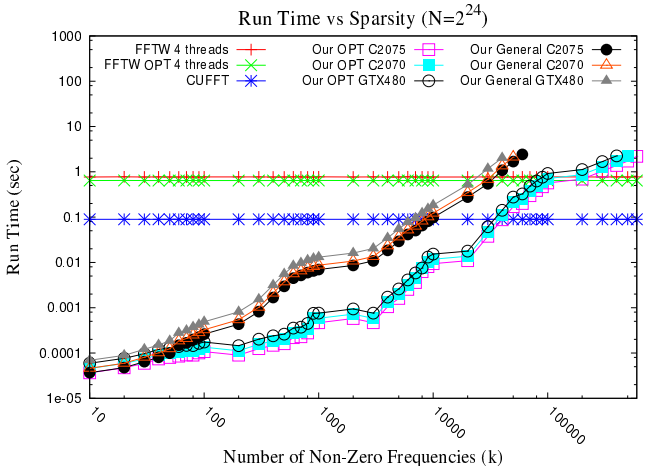

Figure 8: Parallel performance vs. sparsity.

4.2 Detection for Signal Discontinuity

In section 4.1, the performance of our pure sparse-FFT library is evaluated based on homogeneous input signals. When the homogeneity is broken, the heuristic introduced in the section 3.5 is used to detect when the discontinuity happens. Next, we evaluate how well our heuristic works and how much overhead it incurs.

We use test cases similar to the image streaming processing scenario described in section 3.5.3. In total 3 segments of image frames are used. In each segment, the frames are homogeneous except for the last frame at which discontinuity occurs. For the case-1 discontinuity, we keep to use the

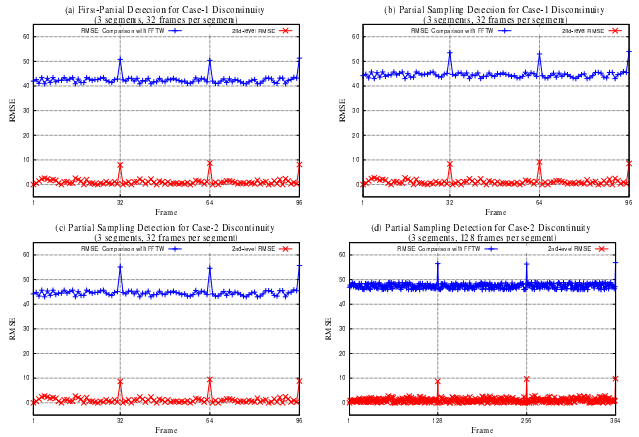

Figure 9: Detection for signal discontinuity.

same signal size N, but re-generate signal with randomly picked amplitudes differing from the homogeneous signals. For the case-2 discontinuity, the signal size is cut down to $N/2$ and its amplitudes are randomly re-generated. The size of each image signal is $N = 2^{22}$ and the sparsity parameter $k = 64$.

Fig. 9(a) and Fig. 9(b) show the first-partial method and the partial sampling detection method for the case-1 discontinuity. There are 3 segments and 32 frames per segment. For each homogeneous frame, it shifts by a displacement of 2^{17} points. We use the Root-Mean-Square-Error (RMSE) between the sampled FFT and our sparse FFT as the deviation metric (Sec. 3.5.3). The RMSE is defined as $\sqrt{\frac{\sum_{i=0}^{N-1}[(F_x-f_x)^2+(F_y-f_y)^2]}{2N}}$.

As shown in Fig. 9(b), our library and the sampled FFT produce almost the same results for homogeneous frames with the 1^{st}-level RMSEs in the small range of $(4.43 \times 10^1, 4.52 \times 10^1)$. However, the outputs are significantly different at the three discontinuity points with the 1^{st}-level RMSEs being 5.35×10^1, 5.31×10^1 and 5.41×10^1, respectively. The spectrally similar signals will produce much closer RMSEs than the discontinuous cases. We can further calculate the 2^{nd}-level RMSE as the RMSE of the 1^{st}-level RMSEs of adjacent frames. The line in the figure represents the 2^{nd}-level RMSE clearly shows values smaller than 2.7 for the homogeneous cases and larger than 8.5 for the discontinuous points. The two boundaries are separated by a large margin. Therefore, the discontinuity is accurately detected at frame 32, 64 and 96 by both sampling methods . Fig. 9(c) shows the partial sampling detection method for case-2 discontinuity with 3 segments and 32 frames per segment. Each homogeneous frame shifts by 2^{17} points. Similarly, discontinuities are detected at frame 32, 64 and 96, and the RMSE of case-2 discontinuity is larger that of the case-1. Fig. 9(d) shows the partial sampling detection for case-2 discontinuity with 3 segments and 128 frames per segment. For each homogeneous frame, it shifts by a factor of 2^{15}. The discontinuities at the frames 128, 256 and 384 are also successfully detected.

4.3 Performance of Input Adaption Process

When discontinuity is detected by the heuristic, the Fourier templates will be re-generated. This section evaluates the impact of this input adaption process to the overall performance.

In this section, the input size is $N = 2^{22}$ and the sparsity parameter $k = 64$. The overall performance is measured including the overhead of the detection heuristic, the recalculation of spectrum templates when discontinuity is found, and the adaptive sparse FFT. Clearly, the more frequently the spectrum templates are calculated, the higher the overhead is, and the lower the performance advantage of our adaptive sparse FFT over the existing input oblivious algorithms. Therefore, we try to determine the break-even point by varying the number of homogeneous frames in a video segment, i.e., in each segment all frames are homogeneous except for the last frame where discontinuity occurs.

Fig. 10 and Fig. 11 illustrates the performance of our input adaption process with 3 segments of frames on CPU and GPUs, respectively. In Fig 10, when the $\#frames \geq 8$, our sequential library on Intel i7 920 CPU is faster than both 1-thread and 4-thread FFTW, while when $\#frames \geq 32$, our library is faster than sFFT 1.0 and 2.0. Moreover, when $\#frames = 128$, on average our algorithm gains 30.2×, 11.5× speedup over the 1-thread and the 4-thread FFTW, and 4.2×, 2.6× speedup over sFFT 1.0 and 2.0, respectively. In Fig. 11, when the $\#frames \geq 8$, our parallel library on the three GPUs is faster than both 1-thread and 4-thread FFTW, while when $\#frames > 16$, our library is faster than sFFT 1.0, 2.0 and CUFFT. Furthermore, when $\#frames = 128$, our implementation on Tesla C2075 GPU achieves 77.2×, 29.3× speedup over 1-thread and 4-thread FFTW, and 10.7×, 6.4×, 5.2× speedup over sFFT 1.0, sFFT 2.0 and CUFFT, respectively. Meanwhile, when $\#frames = 128$, our optimal performance on Tesla C2075 obtains 6.9× speedup against that of our own sequential CPU version.

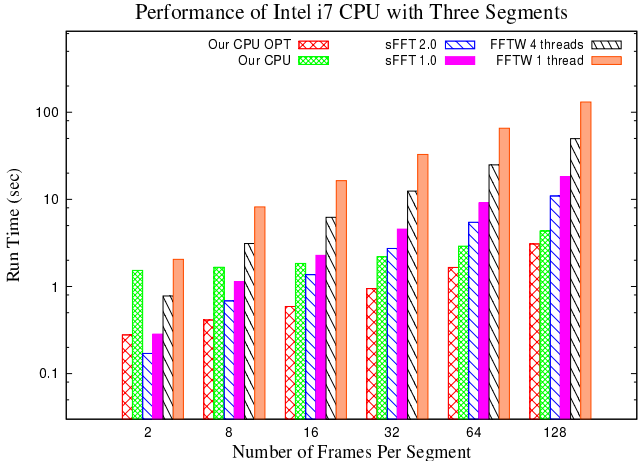

Figure 10: CPU performance with 3 video segments.

4.4 Precision of Our Sparse FFT

The accuracy of our sparse FFT implementation is verified by comparing its complex Fourier transform (F_x, F_y) with the output (f_x, f_y) of FFTW, which is a widely used standard FFT library, for the same double-precision input. The difference in output is quantified as RMSE which has been defined in section 4.2. Lower RMSE value means the two computation routines produce more similar result.

The RMSEs of different signal sizes N and sparsity parameters k are shown in Fig. 12. The RMSE is extremely

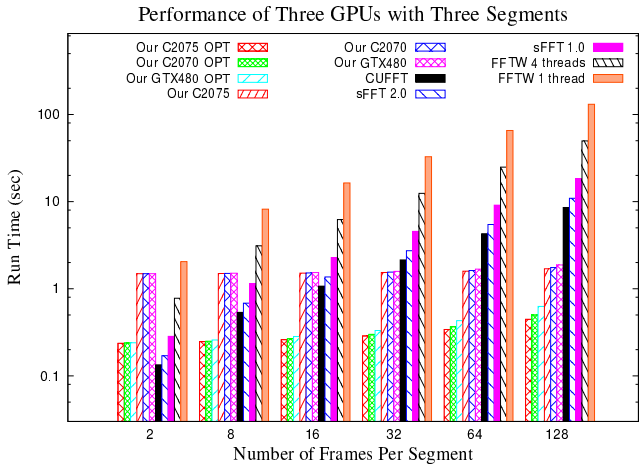

Figure 11: GPU performance with 3 video segments.

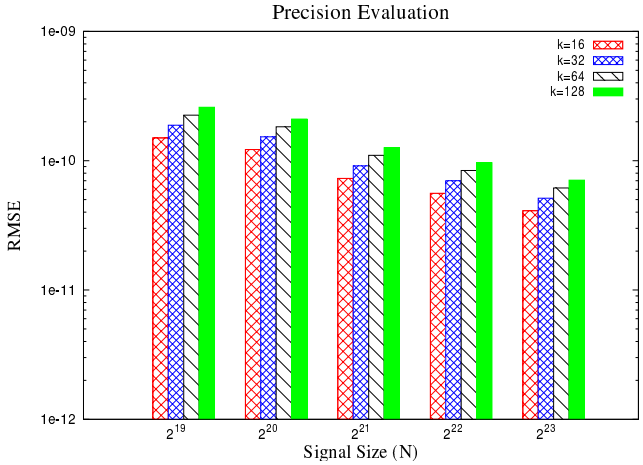

Figure 12: Precision of our algorithm.

small and is in the range of $(4.1 \times 10^{-11}, 1.5 \times 10^{-10})$. Additionally, the RMSE of k-sparse coefficients between our output and FFTW has been measured and is in the range of $(1.61 \times 10^{-8}, 3.12 \times 10^{-8})$. In other words, our sparse FFT produces the same accurate results as FFTW. Interestingly, with the decrease of k for each N, the RMSE decreases. Meanwhile, under the same k, when N increases, the RMSE shows a slight decrease.

5. CONCLUSION

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

Acknowledgement: This work is partly supported by NSF Grant 1115771, AFOSR Grant FA9550-13-1-0213, and gifts from NVIDIA.

6. REFERENCES

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

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

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

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

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

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

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

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

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

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

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

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

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

[14] A. Nukada and S. Matsuoka. Learning decision trees using the fourier spectrum. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 455–464. ACM, 1991.

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