An Input-Adaptive Algorithm for High Performance Sparse Fast Fourier Transform

Shuo Chen and Xiaoming Li

University of Delaware, Newark, DE, USA
schen@udel.edu xli@udel.edu

Abstract. Many applications invoke the Fast Fourier Transform (FFT) on sparse inputs, with most of their Fourier coefficients being very small or equal to zero. Compared with the "dense" FFT algorithms, the input sparsity makes it easier to parallelize the sparse counterparts. In general, sparse FFT algorithms filter input into different frequency bins, and then process the bins separately. Clearly, the performance is largely determined by the efficiency and the effectiveness of those filters. However, sparse FFT algorithms are input-oblivious with regard to filter selection, i.e., input characters are not considered in the design and tuning of their sparse filters, which leads to sub-optimal binning and consequently hurts performance. This paper proposes an input-adaptive sparse FFT algorithm that takes advantage of the similarity between input samples to automatically design and customize sparse filters that lead to better parallelism and performance. More specifically, given a sparse signal that has only k non-zero Fourier coefficients similar to another known spectral representation, our algorithm utilizes sparse approximation to estimate the DFT output in the runtime sub-linear to the input size. Moreover, our work automatically adapts to different input characteristics by integrating and 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, the input-tuned filtering gets rid of recursive coefficient estimation and improves parallelism and performance. We evaluate our input-adaptive sparse FFT implementation in sequential case on Intel i7 CPU and in parallel versions on three NVIDIA GPUs, i.e., NVIDIA GeForce GTX480, Tesla C2070 and Tesla C2075. In particular, our performance is compared to that of the SSE-enabled FFTW and to the results of a highly-influential recently proposed sparse Fourier algorithm. In summary, our algorithm is faster than FFT both in theory and implementation. Furthermore, the range of sparsity k that our approach can outperform dense FFT is larger than that of other sparse Fourier algorithms.

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 $O(Nlog(N))$ steps. The performance of FFT algorithms is known to be determined only by input size, and not affected by the value of input. In real world applications, however, 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. Sublinear sparse Fourier algorithm was first proposed in [16], and since then, has been extensively studied in the literatures when applied to various fields [1, 2, 6, 7, 12, 15]. A recent highly-influential work [9] presented an improved algorithm in the running time of $O(k\sqrt{NlogN}logN)$ to make it faster than FFT for the sparsity factor k up to $O(\sqrt{N/logN})$.

The input sparsity makes it easier to parallelize FFT calculation. From a very high point of view, the sparse FFT algorithms apply time-domain or spectrum-domain filters to disperse inputs into separate bins, the computational complexity of those filters being lower than $O(Nlog(N))$. The number of bins is usually much smaller than N. After

the dispersion, those bins can be further processed separately, for example, even with straightforward DFT algorithms. Combining the two steps, sparse FFT algorithms can achieve complexity lower than $O(Nlog(N))$.

Sparse FFT algorithms are easier to parallelize than normal FFT algorithms. It is because the calculation on the bins are independent. It is also not hard to see that the performance of sparse FFT algorithms, as well as how effectively we can parallelize the following calculation on the bins, are largely determined by the design of input filters. In fact, much of existing work on sparse FFT algorithms focuses on improving the design of sparse filter.

However, the existing filters are really input-oblivious in the sense that their design, unchangeable at runtime, does not consider input characteristics other than input size. The current filters are designed based on the sole assumption that inputs are sparse, but are ignorant to the knowledge of how exactly sparse. The exclusion of input characteristics in filter design appears reasonable at first look, because if we already know how exactly an input is sparse, i.e., its spectrum representation, we don't need to calculate the FFT at all.

Here we make an important observation. What if we don't know the exact spectrum representation of an input, but we know the input has a similar sparsity distribution to another signal whose spectrum representation is known, can the sparsity similarity help improving the sparse FFT on the current input? This paper gives a Yes answer. First of all, the sparsity similarity is common in real-world sparse FFT applications. For example, in video compression, two consecutive video frames usually have almost identical spectrum representations, and differ only in the phases of some spectrum coefficients.

This observation motivates this paper. Our basic idea, also our main innovation and contribution, is to use the sparsity similarity 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. In particular, our input-adaptive sparse FFT implementation particularly benefits FFT calculation in stream processing.

The remaining of this paper is organized as follows. We first briefly introduce existing sparse FFT algorithms and overview our approach. Then we present how we customize filters based on the sparsity template, and how we use the customized filters to reduce the overhead and the number of iterations in the sparse FFT algorithm presented in [9], which our work is based on. And finally, we compare the performance and accuracy of our input-adaptive sparse FFT algorithm with FFTW and the latest sparse FFT implementation on synthetic and real video inputs.

2 Overview of Sparse FFT Algorithms and Our Approach

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

A naive discrete Fourier transform of a N-dimensional input series $x(n)$, $n = 0, 1, ..., N-1$ is presented as $Y(d) = \sum_{n=0}^{N-1} x(n)W_N^{nd}$, where $d = 0, 1, ..., N-1$ and twiddle factor $W_N^{nd} = e^{-j2\pi nd/N}$. Fast Fourier transform algorithms recursively decompose a N-dimensional DFT into several smaller DFTs [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 $N = N_1N_2$, where N_1 and N_2 are co-prime numbers. Twiddle factor calculation is not included in this algorithm. In addition, Rader's algorithm [17] and Bluestein's algorithm [3] can factorize a prime-size DFT as convolution. So far, all FFT algorithms cost time at least proportional to the size of input signal. However, if the output of a DFT is k-sparse, i.e., most of the Fourier coefficients of a signal are very small or equal to zero and only k coefficients are large, the transform runtime can be reduced to only sublinear to the signal size N.

Sublinear sparse Fourier algorithm was first proposed in [16], and since then, has been extensively studied in many application fields [1,2,6,7,12,15]. All these algorithms have runtimes faster than original FFT for sparse signals. However, their runtimes have large exponents (larger than 3) in the polynomial of k and $logN$, and their complex algorithmic structures impose restrictions on fast and parallel implementations.

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

In this paper, we address these limitations by proposing a new sublinear algorithm for sparse fast Fourier transform. Our algorithm has a quite simple structure and leads to a low big-Oh constant in runtime. Our sparse FFT algorithm works in the context that the sparse FFT is invoked on a stream of input signals, and neighboring inputs have very similar spectrum distribution including the sparsity k. The assumption is true for many real-world applications, for example, for many video/audio applications, where neighboring frames have almost identical spectral representations in the locations of large Fourier coefficients, and only differing in the coefficients' values. Our main idea is to use the output of the previous FFT, i.e., the spectral representation of the previous input, as a template to decide, for the current input signal, how to most efficiently bin large Fourier coefficients into a small number of buckets, and each bucket is aimed to have only one large coefficient whose location and magnitude can be then determined. In particular, an n-dimensional filter D that is concentrated both in time and frequency [9, 10] is utilized for binning and to ensure the runtime to be sublinear to N. What binning does is essentially to convolute a permuted input signal with the selected filter in spectral domain. During the binning, each bucket receives only the frequencies in a narrow range corresponding to the length of pass region of the filter D's spectrum, and pass regions of different buckets are disjoint. The prerequisite of having such a pass region had only one large coefficient is to make all adjacent coefficients have equal distance. The information of likely coefficient locations used in the filter tuning is derived from the sparsity template. We make use of a hash table structure to directly permute coefficients in spectral domain to achieve the expected equal distanced permutation. Fig.1 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.

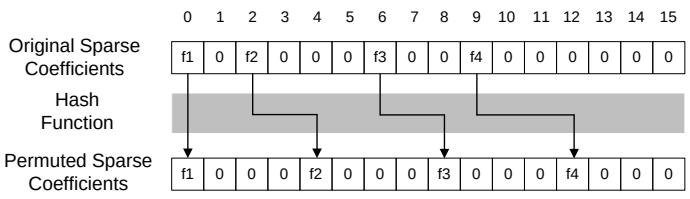

Fig. 1. Hash table based permutation.

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

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

Another notable contribution in our paper is the achievement of parallelization to our sparse FFT algorithm. Since our algorithm is non-iterative with high arithmetic intensity, data parallelism can be exploited from the algorithm. The graphical processing units (GPUs) are used for the well-suited data parallel computations. We parallelize three main sections in our algorithm: permutation to input, subsampled FFT and coefficient estimation.

Our algorithm is evaluated empirically against FFTW, an efficient implementation of FFT with a runtime $O(NlogN)$. For $N = 2^{27}$, our optimized sequential and parallel algorithm outperforms FFTW for k up to about 2^{19} and 2^{21}, which is an order of magnitude higher than that in prior algorithms.

Finally, our algorithm are demonstrated to be adaptive to input characteristics. In our evaluation, we use frames from a video camera recording the movement of an object. At the beginning, we capture a video frame of that object at initial time slot T_0 and utilize our sparse FFT algorithm to generate an output. Then we can use the information of that output to help efficiently calculate sparse FFT outputs in subsequent time slots. As a result, our sparse Fourier Transform algorithm saves much time to do the image/video processing and compression. Fig.2 shows the example of application for our adaptive sparse FFT algorithm.

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

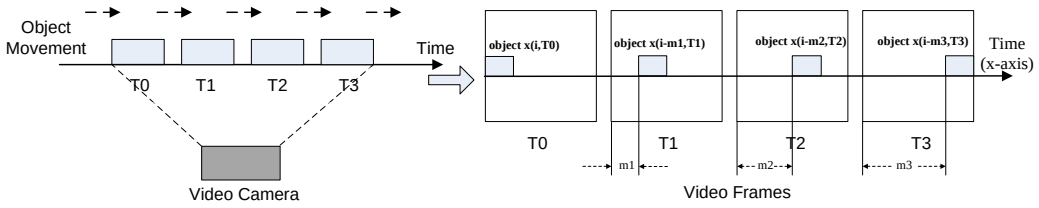

Fig. 2. Application of our input adaptive sparse FFT algorithm.

3 Input Adaptive Sparse FFT Algorithm

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

3.1 General Input-Adaptive Sparse FFT Algorithm

Notations and Assumptions. For a time-domain input signal x with size N (assuming N is an integer power of 2), its DFT is $\hat{x}$. The sparsity parameter of input, k, is defined as the number of non-zero Fourier coefficients in $\hat{x}$. In addition, $[q]$ refers to the set of indices $\{0, ..., q-1\}$. $supp(x)$ refers to the support of vector x, i.e. the set of non-zero coordinates, and $|supp(x)|$ denotes the number of non-zero coordinates of x. Finally, our sparse FFT algorithm works by assuming the locations loc_j of non-zero Fourier coefficients can be estimated from similar prior inputs, where $j \in [k]$. The location template is computed only once for a sequence of signal frames that are similar to each other. The computing of the template by our input-adaptive mechanism which is described in section 3.5. In particular, we invoke existing video processing technology, e.g., [14], to detect the discontinuity in frames' spectral similarities. Therefore, our algorithm is able to compute sparse Fourier transforms for the extracted time-shifting objects within the frames that have homogeneity in the scenes and spectrums, but when we find that homogeneity is broken, our algorithm re-calculates the template and restarts the input-adaptation.

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

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

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

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

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

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

Flat Window Functions. In this paper, the method of constructing a flat window function is same as that used in paper [9]. The concept of flat window function is derived from standard window function in digital signal processing. Since window function works as a filter to bin non-zero Fourier coefficients into a small number of buckets, the pass region of filter is expected to be as flat as possible. Therefore, our filter is constructed by having a standard window function convoluted with a box-car filter [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, 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.

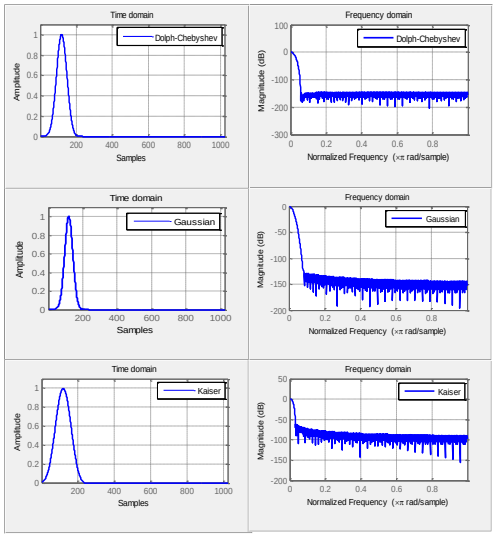

Fig. 3. An example of Dolph-Chebyshev, Gaussian, Kaiser flat window functions for $N = 1024$.

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 an optimal one for us to use due to its flat pass region as well as quick and deep attenuation in stopband.

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

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

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

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

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

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

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

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

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

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

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

3.2 Optimized Input-Adaptive Sparse FFT Algorithm

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

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

Input-Adaptive Shifting. There are two ways to do shifting:

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

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

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

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 Algorithm

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

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

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

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

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

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

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

3.4 Parallel Input-Adaptive Sparse FFT Algorithm

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

Parallelism Exploitation and Kernel Execution. We first parallelize our general sparse FFT implementation. Since data parallelism is a set of homogeneous tasks executed repeatedly over different data elements, we have such parallelism existing in subsections of hashed index computation, filtering and permuting input, subsampling FFT, and location recovery. Therefore, to achieve high performance we construct GPU computational *kernel* for each subsection. First of all, kernel $HashFunc()$, whose number of threads is k, is responsible to compute hashed indices of permuted coefficients and to determine shift factors. The loop of size k is decomposed and each scalar thread in kernel concurrently works as each index j in the algorithm. In addition, kernel $Perm()$ with total number of threads $k^2 logN$ is used to apply filter and permutation to input. Each thread multiplies filter as well as shifting factor with input for one element. We parallelize subsampling to input in kernel $Subsample()$ with total k threads before we launch our well-tuned FFT kernel $TunedFFT()$. Finally we obtain output from location estimation kernel $Recover()$ with k threads parallelizing the loop of algorithm.

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

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

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

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

Tuned GPU based FFT Library. On GPU, our $TunedFFT()$ 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, device memory has much higher latency and lower bandwidth than 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 such GPU based FFT framework.

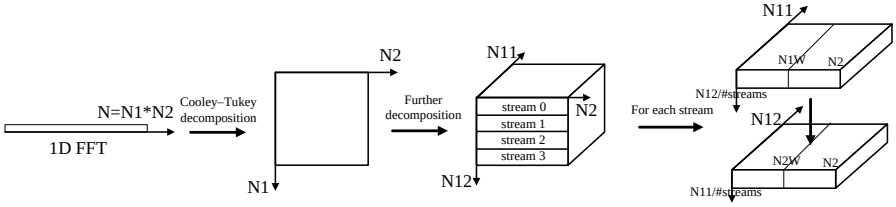

Fig. 4. Working flow of well-tuned GPU based FFT.

3.5 Real-world Application

In this section, we demonstrate how our input adaptive algorithm works in real world. Meanwhile, we illustrate how the Fourier location template is generated. We use a sample of video recording in real application shown in Fig.2. The video sample uses a fix video camera to record the movement of a 2D object along x-coordinate for a duration of time. For each time slot we obtain a 2D video frame containing the object image which can be represented as a 2D matrix $img(g, h)$ whose values stand for color digits, where $g \in [ro]$, $h \in [col]$. The number of rows and columns is ro and col, respectively. Particularly, we substitute the 2D matrix into a row-major 1D array $x_i = x(i = g * col + h) = img(g, h)$. Assuming the interval between the same object in two time-adjacent video frames is m, it can be proved that shifting to $img(g, h)$ is the same to x_i since $img(g, h - m) = x(g * col + h - m) = x_{i-m}$. Therefore, the process of video recording is modeled as a time shifting process to x_i and we want to compute its Fourier transform $\hat{x}_j$ for image/video processing and compression.

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

Moreover, if time shifting factor m_t is known, we can further directly multiply $\hat{x}_{j,T0}$ at initial time T_0 by $e^{-b2\pi m_t j/N}$ to efficiently attain Fourier transform $\hat{x}_{j,Tt}$ at remaining time T_t without a FFT. However, if shifting factor m_t is unknown, we cannot do this to get spectrums for $x_{i-m_t,Tt}$. This situation is feasible in real application. Suppose we use a video recorder to capture several video frames, but sometimes we don't know the time-shifted distance m_t of the two frames. Hence, we have to know m at first. The worst case is to match $x_{i,T0}$ with $x_{i-m_t,Tt}$ and determine m_t in runtime of $O(N^2)$. Nonetheless, such a process can be efficiently executed in $O(N)$ time when applying the algorithm in [13]. Therefore, if m_t is unknown, we spend time of $O(N)$ on finding m_t and $O(k)$ on multiplying $e^{-b2\pi m_t j/N}$. Total runtime is $O(N+k)$. In the evaluation section, we conduct detailed evaluation to show our sparse FFT outperforms the performance of above two situations including known m_t and unknown m_t.

4 Experimental Evaluation

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

The double-precision performance evaluation is conducted on three heterogeneous computer configurations. Sequential implementation is executed on Intel i7 920 CPU and the parallel case is run on three different NVIDIA GPUs, i.e. GeForce GTX480, Tesla C2070 and Tesla C2075. For sequential version, we evaluate our general and optimized sparse FFT approaches, and compare them against three highly-influential FFT libraries: 1) FFTW 3.3.3 [5], the latest FFTW which is the most efficient implementation for computing the dense FFTs. In FFTW, Streaming Single Instruction Multiple Data Extensions (SSE) on Intel CPU is enabled for better performance. 2) sFFT 1.0 and 2.0 [9], which is one of the fastest sublinear algorithms of sparse FFT with an open source library. 3) AAFFT 0.9 [11], which is another recent sublinear algorithm with fast empirical runtime. For the parallel version, since there is no parallel sparse-FFT library for us to use, we only compare our GPU based performance to four threads enabled FFTW. The GPU performance reported here includes time of both computation and data transferring between host and device. Furthermore, all FFTW libraries we use are with two flags, i.e. ESTIMATE (a basic version marked as 'FFTW' in the plots) and MEASURE (an optimal version marked as 'FFTW OPT' in the plots). The configurations of GPUs and CPU are summarized in Table 1.

Table 1. Configurations of GPUs and CPU

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

4.1 General Input-Adaptive Sparse FFT Algorithm

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

Fig.5 shows our sequential sparse FFT on an Intel i7 CPU. In Fig.5.a, we fix $k = 64$ and change N. The running time of FFTW is linear in the signal size N and sFFT 1.0/2.0 shows approximately linear in N when $N > 2^{20}$. However, our general sparse FFT appears almost constant as the signal size increases, which is a result of our

sub-linear property in algorithm. AAFFT 0.9 also shows constantly over different N but its runtime performance is worse than ours and sFFT. Moreover, our approach demonstrates the fastest runtime over sFFT, FFTW and AAFFT. For $N \geq 2^{15}$ our algorithm is faster than FFTW, while sFFT and AAFFT has to reach this goal for $N \geq 2^{19}$ and $N \geq 2^{24}$, respectively. In Fig.5.b, we fix $N = 2^{27}$ and change k. FFTW shows invariance in performance since its complexity is $O(NlogN)$ which is independent on k. Additionally, our general sparse FFT has a faster runtime than basic and optimal FFTW for k up to 11000 and 10000, respectively. However, sFFT 1.0, sFFT 2.0 and AAFFT 0.9 are faster than basic FFTW only when k is less than 8000, 9000 and 1000. Therefore, our approach extends the range of k where our performance is faster than dense FFT. Furthermore, our general algorithm performs better than other compared FFT libraries on average.

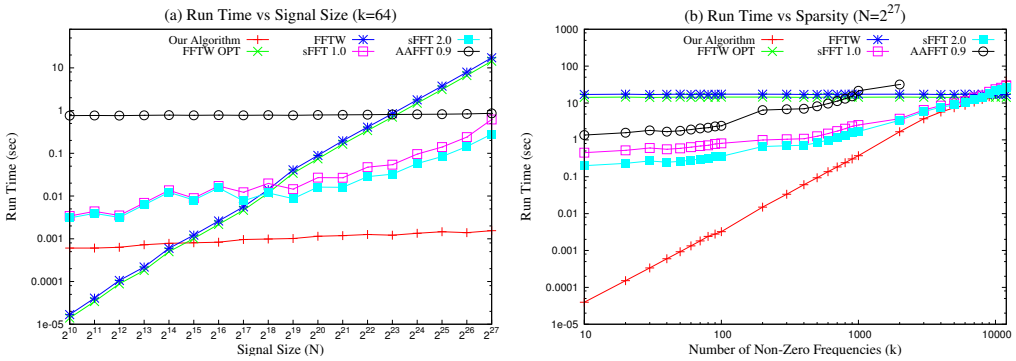

Fig. 5. Performance of our general sparse FFT in sequential case.

Fig.6 shows the parallel versions of our sparse FFT on three high performance GPUs. Since there is no parallel sparse FFT libraries for us to use, we compare our parallel performance to single-thread and 4-thread FFTW. Additionally, to better show our GPU based performance we also add the implementation of our parallel version under 4 CPU threads. In Fig.6.a, we fix $k = 64$ and change N. Both 1-thread and 4-thread FFTW are linear in the signal size N, however, our parallel approach still appears constant as N increases. Moreover, our three GPU implementations are faster than 1-thread and 4-thread FFTW when $N \geq 2^{14}$ and $N \geq 2^{15}$, while our 4-thread CPU case is exceeds 1-thread and 4-thread FFTW only when $N \geq 2^{15}$ and $N \geq 2^{16}$, respectively. In Fig.6.b, we fix $N = 2^{27}$ and change k. Specifically, our parallel performance on GTX480, Tesla C2070 and C2075 has a runtime faster than 1-thread FFTW for k up to 40000, 50000, 60000 and than 4-thread FFTW for k reaching to 20000, 30000, 30000, respectively. Meanwhile, our 4-thread CPU based implementation exceeds 1-thread and 4-thread FFTW only when k is less than respective 15000 and 7000.

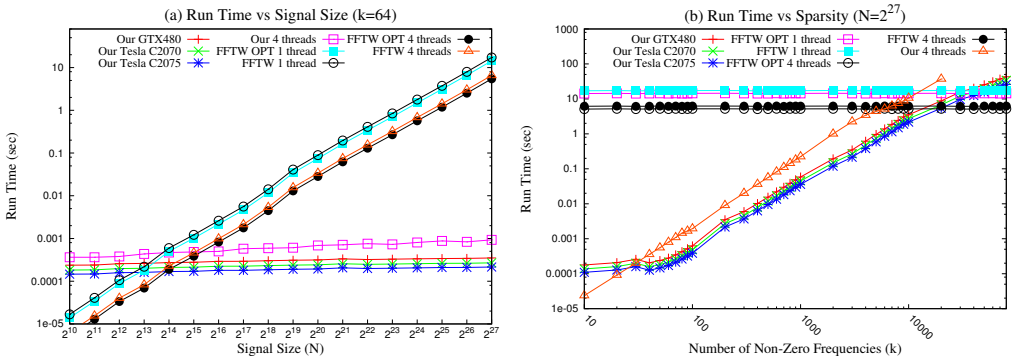

Fig. 6. Performance of our general sparse FFT in parallel case.

4.2 Optimized Input-Adaptive Sparse FFT Algorithm

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

Fig.7 shows our optimized sparse FFT in sequential case. In Fig.7.a, we fix $k = 64$ and change N. Our optimized approach is sub-linear due to its constant curve when N increases. In addition, the optimal and average case of our optimized algorithm is faster than FFTW when $N \geq 2^{14}$ and $N \geq 2^{15}$, respectively. However, sFFT 1.0/2.0 and AAFFT 0.9 has to achieve this purpose for $N \geq 2^{19}$ and $N \geq 2^{24}$, respectively. In Fig.7.b, we fix $N = 2^{27}$ and change k. Our optimal and average case has a runtime faster than FFTW for k up to 1000000 and 25000, respectively. However, sFFT 1.0, sFFT 2.0 and AAFFT 0.9 are faster than basic FFTW only when k is less than 8000, 9000 and 1000. On average, our optimized algorithm performs better than other compared FFT libraries.

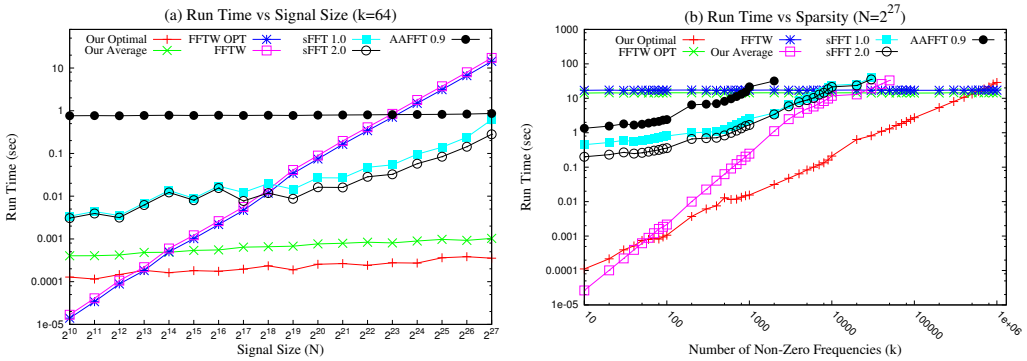

Fig. 7. Performance of our optimized sparse FFT in sequential case.

Fig.8 shows the parallel performance of the optimized algorithm. The experimental configurarion is same as that in our general algorithm. In Fig.8.a, we fix $k = 64$ and change N. Our three GPU implementations are faster than 1-thread and 4-thread FFTW when $N \geq 2^{13}$ and $N \geq 2^{14}$. It is also faster than the dense CUFFT 3.2 when $N \geq 2^{15}$. In Fig.8.b.c.d, we fix $N = 2^{27}$ and change k. The performance of optimal algorithm on GTX480, Tesla C2070 and C2075 has a runtime faster than 1-thread FFTW for k up to 1500000, 2000000, 3000000 and than 4-thread FFTW for k reaching to 500000, 700000, 900000, respectively. Meanwhile, performance of our optimized method in average case on GTX480, C2070 and C2075 is faster than 1-thread FFTW for k up to 50000, 70000, 80000 and than 4-thread FFTW for k reaching to 30000, 40000, 50000, respectively. In addition, our optimal version on the three GPUs is faster than dense CUFFT when k reaches to 70000, 80000, 90000, while our average case on the GPUs exceeds CUFFT for k up to 6000, 7000, 8000. Particularly, in Fig.8.c, CUFFT is also added in our Tesla C2070 test as kernel $TunedFFT()$. As a result, our tuned GPU based FFT kernel is 21% faster than CUFFT counterpart.

4.3 Evaluation of Real-world Application and Accuracy

We show an performance evaluation for better illustrating the real-world application of video recording in section 3.5. Suppose we capture total 10 video frames in 10 time slots. The displacement between the object in adjacent two time slots is set to 2^{10} points. Fig.9.a shows the sequential performance of our hybrid sparse FFT against to the dense FFT performance, i.e. FFTW, and to the performance of two situations including known shifting factor m and unknown m. X-axis represents the time slots T_t. Input signal size is $N = 2^{27}$ points and $k = 64$. The test shows that our hybrid FFT in both sequential and parallel version outperform the performance of all other

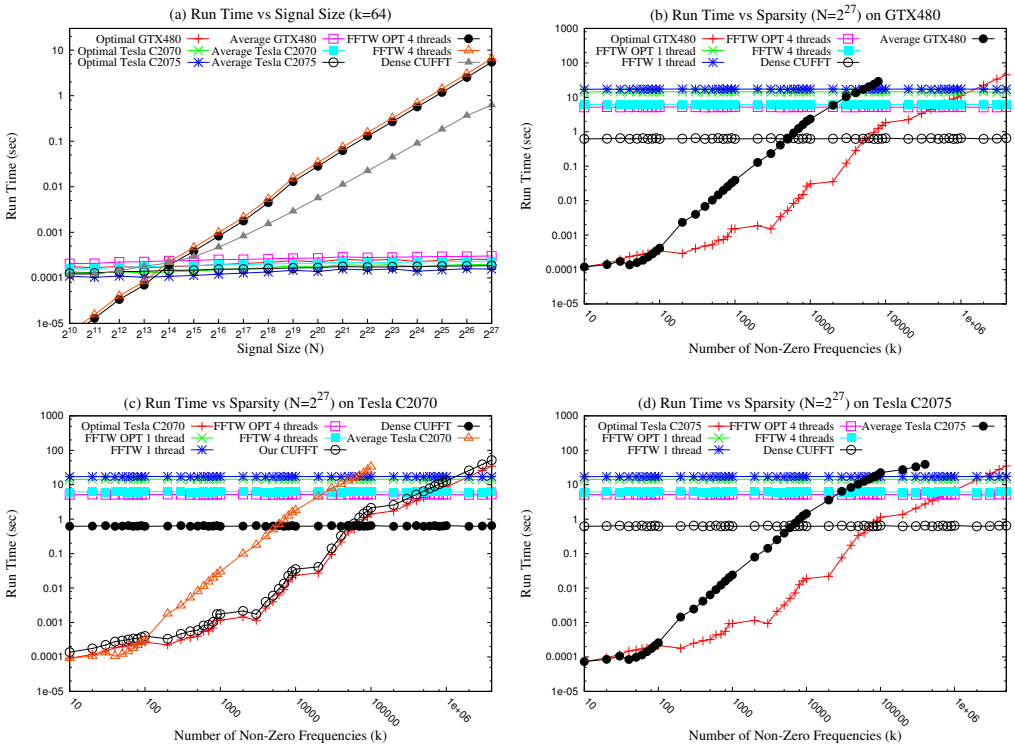

Fig. 8. Performance of our optimized sparse FFT in parallel case.

compared situations. It demonstrates that we can spend time to compute a dense FFT once to preprocess the Fourier location template that we need for the FFTs in remaining time. Then we can save much time by using our hybrid sparse FFT for all the subsequent input signals. Furthermore, if the number of frames is large, our sparse FFT outperforms sFFT as well as AAFFT on average in the real application.

The accuracy of our sparse FFT implementation is verified by comparing its complex Fourier transform (F_x, F_y) with FFTW's output (f_x, f_y) for the same double-precision input data. The difference in output is quantized as root mean square error (RMSE) over the whole data set. The RMSE is defined as $\sqrt{\frac{\sum_{i=0}^{N-1}[(F_x-f_x)^2+(F_y-f_y)^2]}{2N}}$ and is shown in Fig.9.b for $N = 2^{27}$ and different k. Overall, the RMSE is extremely small and demonstrates a good accuracy of our algorithm.

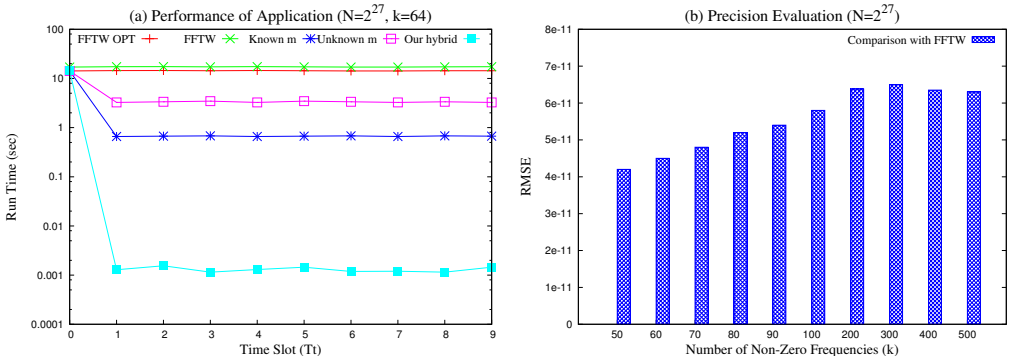

Fig. 9. Performance of a real-world application and accuracy of our algorithm.

5 Conclusion

In this paper, we proposed an input-adaptive sparse FFT algorithm that takes advantage of the similarity between sparse input samples to efficiently compute a Fourier

transform in the runtime sublinear to signal size N. Specifically, our work integrates and tunes several adaptive filters to package non-zero Fourier coefficients into sparse bins which can be estimated accurately. Moreover, our algorithm is non-iterative with high computation intensity such that parallelism can be exploited for multi-CPUs and GPU to improve performance. Overall, our algorithm is faster than FFT both in theory and implementation, and the range of sparsity parameter k that our approach can outperform dense FFT is larger than that of other sparse Fourier algorithms.

References

1. Akavia, A.: Deterministic sparse fourier approximation via fooling arithmetic progressions. In: The 23rd Conference on Learning Theory. pp. 381–393 (2010)
2. Akavia, A., Goldwasser, S., Safra, S.: Proving hard-core predicates using list decoding. In: The 44th Symposium on Foundations of Computer Science. pp. 146–157. IEEE (2003)
3. Bluestein, L.: A linear filtering approach to the computation of discrete Fourier transform. Audio and Electroacoustics, IEEE Transactions on 18(4), 451–455 (1970)
4. Duhamel, P., Vetterli, M.: Fast fourier transforms: a tutorial review and a state of the art. Signal Process. 4(19), 259–299 (Apr 1990)
5. Frigo, M., Johnson, S.G.: The design and implementation of fftw3. Proceeding of the IEEE 93(2), 216–231 (2005)
6. Gilbert, A., Guha, S., Indyk, P., Muthukrishnan, M., Strauss, M.: Near-optimal sparse fourier representations via sampling. In: Proceedings on 34th Annual ACM Symposium on Theory of Computing. pp. 152–161. ACM (2002)
7. Gilbert, A., Muthukrishnan, M., Strauss, M.: Improved time bounds for near-optimal space fourier representations. In: Proceedings of SPIE Wavelets XI (2005)
8. Good, I.: The interaction algorithm and practical Fourier analysis. Journal of the Royal Statistical Society, Series B (Methodological) 20(2), 361–372 (1958)
9. H., H., P., I., Katabi, D., E., P.: Simple and practical algorithm for sparse fourier transform. In: Proceedings of the 23th Annual ACM-SIAM Symposium on Discrete Algorithms. pp. 1183–1194. ACM (2012)
10. Hassanieh, H., Indyk, P., Katabi, D., E., P.: Nearly optimal sparse fourier transform. In: Proceedings of the 44th symposium on Theory of Computing. pp. 563–578. ACM (2012)
11. Iwen, M.: AAFFT (Ann Arbor Fast Fourier Transform) http://sourceforge.net/projects/aafftannarborfa/ (2008)
12. Iwen, M.: Combinatorial sublinear-time fourier algorithms. Foundations of Computational Mathematics 10(3), 303–338 (2010)
13. Knuth, D., Morris, J., Pratt, V.: Fast pattern matching in strings. SIAM Journal on Computing 6(2), 323–350 (1977)
14. Li, L., Huang, W., Gu, I., Tian, Q.: Foreground object detection from videos containing complex background. In: Proceedings of the 11th ACM international conference on Multimedia. pp. 2–10. ACM (2003)
15. Mansour, Y.: Randomized interpolation and approximation of sparse polynomials. In: The 19th International Colloquium on Automata, Languages and Programming. pp. 261–272. Springer (1992)
16. Nukada, A., Matsuoka, S.: Learning decision trees using the fourier spectrum. In: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis. pp. 455–464. ACM (1991)
17. Rader, C.: Discrete Fourier transforms when the number of data samples is prime. Proceedings of the IEEE 56(6), 1107–1108 (1968)