GPU-SFFT: A GPU based parallel algorithm for computing the Sparse Fast Fourier Transform(SFFT)

Oswaldo Artiles, *Life Senior Member*, IEEE, Fahad Saeed, *Senior Member*, IEEE

***Abstract*—The Sparse Fast Fourier Transform (SFFT) is an algorithm developed by the MIT, to compute the discrete Fourier transform of a signal with a sublinear time complexity: $O(\log\ n\sqrt[3]{nk^2\ \log\ n})$, for a signal of size n, and sparsity level k, that is the number of non-zero coefficients in the frequency domain. In this paper, we propose a highly scalable GPU-based parallel algorithm called GPU-SFFT for computing the SFFT of k-sparse signals. We exploit the computational power provided by the modern GPUs to achieve a high performance algorithm. The design and implementation of GPU-SFFT was based on three goals: to unroll the for loops in the sequential MIT-SSFT to increase the parallelism by maximizing the number of concurrent threads executing independent instructions, to minimize the transfer of data between the CPU and the GPU as much as possible, and to use high performance sorting algorithms available in the Thrust library for CUDA, and to compute the reduced size FFTs of the algorithm with cuFFT, the NVIDIA CUDA Fast Fourier Transform (FFT) library. GPU-SFFT is 37x times faster than the MIT-SFFT and 5x faster than cuFFT.**

I. Introduction

The Discrete Fourier Transform (DFT) is one of the most important algorithm for the analysis of the spectral representation of signals in engineering and science, with a wide range of applications from astronomy to medical imaging, and from seismology to cryptography. The DFT algorithm compute the Fourier transform of a discrete signal of size n with time complexity of $O(n^2)$. This time complexity results in unacceptable performance for processing the big data sets characteristic of modern applications. The Fast Fourier Transform(FFT) is the fastest and most widely used algorithm to process the DFT of a signal of size n with a time complexity of $O(n\ log\ n)$ [2]–[5]. However, the FFT algorithm may be too slow to process the DFT of signals with sizes of terabytes, even though these signals may be sparse in the frequency domain, that is only k frequency coefficients are different than zero, where $k << n$. Moreover, in many applications, for example in medical imaging is sometimes hard to obtain sufficient amount of data to compute the DFT with the desired accuracy.

The suboptimal performance of the FFT algorithm to compute the DFT of k-sparse signals, and the existence of an important set of domains(video, audio, medical imaging, spectroscopy, GPS, seismic data) with signals that are sparse in the frequency domain, motivated the development of sublinear algorithms, that is algorithms with runtime complexity proportional to the level of sparsity, which is considerably smaller than the size of the input data and that use only a subset of this data to compute the frequency coefficients which are significant [6]–[8]. The Sparse Fast Fourier Transform (SFFT) algorithms developed at MIT [12] are sublinear algorithms that by exploiting the inherent sparsity of natural signals, are faster than the FFT algorithms for k-sparse data. The MIT-SFFT algorithm implements a solution to the problem of computing the DFT, $\hat{x}(f)$, of a signal, $x(t)$, of size n, with only k non-zero frequency coefficients, the other $n-k$ coefficients are zero. For general signals, the MIT-SFFT computes an approximation $\hat{x}'(f)$ of $\hat{x}(f)$. The time complexity of the version 2.0 of the MIT-SFFT algorithm is $O(\log\ n\sqrt[3]{nk^2\ \log\ n})$ with a sparsity level range of $O(n/\sqrt{\log\ n})$, that is, the algorithm is faster than FFT up to $O(n/log\ n)$. [12].

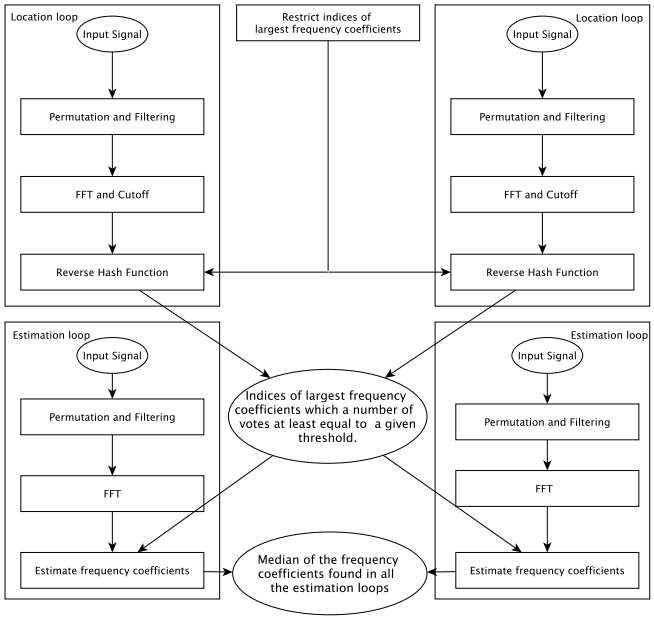

Fig. 1: A simplified flow diagram of the outer loop of the version 2.0 of the MIT-SFFT algorithm

MIT – SFFT algorithm: Figure 1 shows a simplified flow diagram of the the computational stages of the MIT-SFFT algorithm [12]. The *outer loop* of the algorithm is divided into *location loops* and *estimation loops*. The location loops in Figure 1 implement the stages of *permutation and filtering*, *FFT and cutoff*, and *reverse hash function*. The MIT-SFFT algorithm is based on binning the large Fourier coefficients of the significant frequencies, present in the input

Dr Artiles (oarti001@fiu.edu) and Dr Saeed (fsaeed@fiu.edu) are with the School of Computing and Information Sciences, Florida International University, Miami,Florida

signal, into a small set of B bins, where $B = O(k)$ is a parameter that divides n. The collision problem of having non-zero frequencies in the same bin is solved by separating two coefficients that are close to each other, locating them on separate bins, so that there is only one large frequency per bin. The permutation stage of the location loop performs the random permutation of the input signal so that adjacent Fourier coefficients in the frequency domain are evenly separated. The process of binning the frequency coefficients is implemented by filtering the permuted input signal. The filter suppresses the frequency coefficients that hash out of the bin while passing through frequency coefficients that hash into the bin. The utilization of a filter guarantees therefore the goal of binning one frequency coefficient per bin, minimizing the leakage of frequencies from one bin to another [12]. The hashing-based spectrum permutation method implemented in the MIT-SFFT maps indices of the original signal spectrum to the permuted locations so that the original locations can then be recovered in the estimation loops. After permuting and filtering the input signal, the original problem has been reduced from a n-dimensional DFT to a B-dimensional DFT. This size reduced DFT is computed by the FFT algorithm with time complexity $O(B \log B)$.After computing the B-dimensional DFT, each bin in the frequency domain contains at most one potential large coefficient. However, in a k-sparse DFT, it is likely that many of the coefficients in the bins are close to zero. Since the algorithm guarantees that the probability of missing a large coefficient is low if the top $O(k)$ samples are selected. Then, in the cutoff stage, the indices of the top k coefficients of maximum magnitude are selected form the set of B bins in the frequency domain, and the indices of the other $B - k$ coefficients are discarded. In the reverse hash function stage the k largest coefficients found in the cutoff stage are reconstructed by finding the original locations in the frequency domain. The version 2.0 of the MIT-SFFT adds a heuristic to restrict the location of indices of largest coefficients using a filter that has no leakage at all. The intersection of these indices with the indices of k largest coefficients found in the cutoff stage is computed in the reverse hash function.In a voting approach, this function give a vote to an index, every time the index appears in the intersection of both set of indices, when the number of votes reaches a threshold, the index is added to the output of the function. The output of the location loops are the indices of the largest frequency coefficients that have a number of votes at least equal to a given threshold [12]. The estimation loops share the permutation, filtering, and the FFT stages described for the location loops, with the goal of having one large frequency coefficient per bin with high probability. Given the set of indices computed by the location loops, and the bins in the frequency domain, the *estimation coefficients* stage of the estimation loops return the indices and the values of the k largest frequency coefficients. In order to compensate the collision of large frequencies in the same bin, the output of the estimation loops is the median of the values of the frequency coefficients found on all the estimation loops.

CUDA computer model : The programmable Graphics Processing Units (GPUs) is a highly parallel, multithreaded, many core processor with very high computational power and memory bandwidth [13]. The GPU is organized into an array of highly threaded streaming multiprocessors (SMs). Each SM contains several cores that share control logic and instruction cache. The GPU architecture is called Single Instruction Multiple Thread (SIMT), on which hundred of threads on each core can concurrently execute the same instruction [1]. In 2006, NVIDIA introduced CUDA, a general purpose parallel computing platform and programming model what allows programmers to use CUDA-enabled GPUs to solve many complex computational problems [13]. A key component of the CUDA programming model is the kernel, the code that implements the instruction to be executed by the threads on the GPU device. The threads executing the instructions in a kernel are grouped into one or two dimensional *grids*. The grids are made up of threads *blocks*. The threads in a block are organized in groups of 32 called *warps*. All the threads in a warp execute the same instruction at the same time. The CUDA memory model contains different types of programmable memory spaces: Registers, shared memory, local memory, constant memory, texture memory and global memory. Each thread has private local memory. Each thread block has shared memory visible to all threads of the block and with the same lifetime as the block. Using shared memory improves the performance when threads inside the block need to access the same data multiple times. All threads have access to the same global memory. Global memory is the largest, highest latency, and most used memory on a GPU. Data transferred from CPU to GPU resides in global memory. Transferring data between CPU and GPU is a slow process with a negative impact in the performance of a CUDA code, hence this type of transfers should be minimized. Coalesced memory access occur when all the 32 threads in warp access adjacent memory locations. Ensuring coalesced global memory access is an important goal for high performance GPU based algorithms [1].

The GPU based parallel algorithm for computing the Sparse Fast Fourier Transform(SFFT), GPU-SFFT, presented in this paper was implemented in CUDA C using the functionality available in the sequential C++, version 2.0 of the MIT-SFFT package. Three goals were defined for the project of defining and implementing the GPU-SFFT package. The first goal was to unroll the for loops in the sequential MIT-SSFT to increase the parallelism by maximizing the number of concurrent threads executing independent instructions, as well as ensuring coalesced global memory access by the threads in each warp. The second goal was to minimize the transfer of data between the CPU and the GPU as much as possible. The third goal was to replace the sequential sort algorithms in the MIT-SFFT by the high performance sorting algorithms available in the Thrust library for CUDA [14], and to compute the reduced size FFTs of the algorithm with cuFFT, the NVIDIA CUDA Fast Fourier Transform (FFT) library [15].

The comparison of the execution times of the GPU-SFFT with the execution times of the MIT-SFFT, shows that the speedup obtained with the GPU-SFFT was up to $50x$ when the times for data transfer between the CPU and the GPU were not considered, and up to $23x$ when these times were included. Moreover, an up to 5x speedup was obtained when

the execution times of the GPU-SFFT are compared to the corresponding times of cuFFT. For all the experiments performed with the GPU-SFFT, the Mean Absolute Error (MAE) was below 10^{-3}, and there were not missing frequencies in the computed SFFT output signal.

Contributions of the paper: The main contributions of the paper are:

1) We propose GPU-SFFT, a GPU based parallel algorithm for computing the Sparse Fast Fourier Transform(SFFT) of a input signal of size n, with only k frequencies coefficients different than zero in the frequency domain.
2) The experimental results shows that GPU-SFFT is a high performance algorithm without reducing the accuracy of its output as compared to the sequential version of the MIT-SSFT package.

Organization of the paper: The paper is organized as follows: Section II describes the functionality of the GPU-SSFT algorithm, including the techniques used to implement such functionality. Section III is dedicated to the experiments and results. The conclusions are presented in Section IV.

II. GPU based Sparse Fast Fourier Transform Algorithm

GPU-SFFT is a GPU-based parallel algorithm which implements the parallel version of the functionality corresponding to the stages shown in Figure 1 for the MIT-SFFT sequential algorithm. This section includes the description of the functionality of the GPU-SSFT algorithm, including the techniques used to implement such functionality. The description is based on pseudocodes that map to the stages shown in Figure 1.

A. GPU-SFFT outer loop function

Algorithm 1 GPU outer loop function.

1: **Input**: $hx[n]$,$dfilt_t[fs]$,$dfilt_f[fs]$
2: **Output**: $\hat{hx}[IF]$
3: **procedure** OutLpGPU(hx,n,$dfilt_t$,$dfilt_f$,fs,B,B_t,W,L,L_c,L_t,L_l)
4: $dx[n] \leftarrow$ cudaMalloc(n)
5: cudaMemcpy(dx, hx)
6: $dbins_t[L*B] \leftarrow$ cudaMalloc($L*B$)
7: $dbins_f[L*B] \leftarrow$ cudaMalloc($L*B$)
8: $dI[n] \leftarrow$ cudaMalloc(n)
9: $dJ_2[L_c*B_t] \leftarrow$ cudaMalloc(L_c*B_t)
10: $dH_\sigma[L] \leftarrow$ cudaMallocManaged(L)
11: **for** $i \leftarrow 0, L_c$ **do**
12: LocLargeCoefGPU(dx,B_t,n,W,$dJ_2[i*B_t]$)
13: **end for**
14: $IF \leftarrow 0$
15: **for** $i \leftarrow 0, L$ **do**
16: $\sigma \leftarrow random()\ mod\ n$ ▷ $gcd(\sigma, n) = 1$
17: $dH_\sigma[i] \leftarrow \text{modInv}(\sigma, n)$ ▷ $dH_\sigma[i]\sigma = 1(mod\ n)$
18: $dJ[B_t] \leftarrow$ cudaMalloc(B_t)
19: PermFilterGPU(dx,B,$dfilt_t$,fs,$dbins_t$,$dH_\sigma[i]$)
20: FFTCutoffGPU(dJ,$dbins_t$,$dbins_f$,B,B_t)
21: **if** $L < L_l$ **then**
22: RevHashGPU(dI,dJ,B_t,B,n,L_t,dJ_2,W,IF,σ)
23: **end if**
24: **end for**
25: $\hat{hx} \leftarrow$ EstValGPU($dI, IF, dbins_f, dfilt_f, B, n, L, dH_\sigma$)
26: **end procedure**

One of the goals in the design of the GPU-SFFT algorithm was to minimize the transfer of data between the CPU and the GPU as much as possible. The first procedure to achieve this goal is to compute the time and frequency components of the filters on the GPU(device) side as preprocessing stage for the GPU-SFFT algorithm. The second procedure is to transfer the data of CPU(host) input signal of size n to the device side only one time at the starting of the computation. Both techniques are represented in the algorithm 1 for the outer loop function of the GPU-SFFT. The input of the algorithm 1 are the host input signal, $hx[n]$, and the device time and frequency components of the filters, $dfilt_t[fs]$and $dfilt_f[fs]$ respectively, where fs is the size of the filters. Lines 4 and 5 of the algorithm 1, show the allocation of GPU global memory for the device input signal, $dx[n]$, and the memory transfer of $hx[n]$ to $dx[n]$. Lines 6 to 12 of the algorithm 1, show the allocation of GPU global memory for the internal variables. There are two for loops in the algorithm 1, that due to their small size of (L =number of loops < 60) are computed in the host side. The functions which are called in the for loops of the algorithm 1 implement the GPU version of the stages shown in Figure 1. These functions are described in the following sections.

B. GPU-SFFT: Permutation and filtering

Algorithm 2 Sequential function to permute and filter the input signal x.

1: **Input**: $x[n]$,$filt_t[fs]$
2: **Output**: $bins_t[B]$
3: **procedure** PermFilter(x,B,$filt_t$,fs,$bins_t$,$H_{\sigma,i}$)
4: $index \leftarrow 0$
5: **for** $i \leftarrow 0, fs$ **do**
6: $bins_t[i\ mod\ B] + \leftarrow x[index] * filt_t[i]$
7: $index \leftarrow (index + H_{\sigma,i})\ mod\ n$
8: **end for**
9: **end procedure**

The algorithm 2 is the sequential version of the permutation and filtering stage in Figure 1. This algorithm implements(line 6) the hash function given Equation 1 [12] which maps each of the n elements of the input signal to one of B bins.

$$h_{\sigma,B}(i) = floor\Big(\frac{i\sigma}{n/B}\Big) \quad \triangleright\ h_{\sigma,B} : [n] \rightarrow [B] \qquad (1)$$

On a GPU version of the algorithm 2, a thread collision can occur when multiple threads, for example threads $i, B+i, 2B+i, ...$, try to update the same bin concurrently, introducing time delays that negatively impact the performance of the parallel algorithm. The algorithm 3 is the GPU version of the sequential algorithm 2. This algorithm is designed to solve the thread collision with a tiling based approach [11]. The number of colliding threads is approximately equal to $T = floor(fs/B)$,where fs is the size of the filter. Since fs is not divisible by B, when the filter vector is partitioned into T tiles of size B, there are $R = fs \bmod B$ remaining elements of the filter after the T tile. Then, when the permuted and filtered components of the input signal are binned in $bins_t[i\ mod\ B]$, for the iteration i, each element of the filter, $filt_t[i]$, which is convoluted with the permuted input signal, are in different tiles for each iteration. After the $i = T * B$

iteration of the for loop finishes, the remaining R iterations access the last R elements of the filter.

Algorithm 3 GPU function to permute and filter the input signal x.

```
1: Input: dx[n],dfilt_t[fs]
2: Output: dbins_t[B]
3: procedure PMFILTERGPU(dx,B,dfilt_t,fs,dbins_t,dH_{σ,i})
4:     dbins_t[B] ← CUDAMEMSET(dbins_t, 0, B)
5:     T ← fs/B, R ← fs mod B
6:     if n < 2^27 then
7:         PFTKERN(dbins_t,dx,dfilt_t,n,B,dH_{σ,i},T,R)
8:     else
9:         PFKERN(dbins_t,dx,dfilt_t,n,B,dH_{σ,i},fs)
10:    end if
11: end procedure

12: procedure PFTKERN(dbins_t,dx,dfilt_t,n,B,dH_{σ,i},T,R)
13:    i ← threadIdx.x + blockIdx.x * blockDim.x
14:    if i < B or i == B then
15:        if i < B then
16:            for j ← 0, T do
17:                id ← i + B
18:                dbins_t[i]+ ← dx[(id * dH_{σ,i}) mod n] * dfilt_t[id]
19:            end for
20:        end if
21:        if i == B then
22:            for j ← 0, R do
23:                id ← j * T + i
24:                dbins_t[j]+ ← dx[(id * dH_{σ,i}) mod n] * dfilt_t[id]
25:            end for
26:        end if
27:    end if
28: end procedure

29: procedure PFKERN(dbins_t,dx,dfilt_t,n,B,dH_{σ,i},fs)
30:    i ← threadIdx.x + blockIdx.x * blockDim.x
31:    if i < fs then
32:        dbins_t[i mod B]+ ← dx[(i * dH_{σ,i}) mod n] * dfilt_t[i]
33:    end if
34: end procedure
```

The kernel PFTKERN implements this tilling approach to unroll the sequential for loop of the sequential algorithm 2. Each thread on this kernel bins the components of the permuted and filtered input signal that correspond to one bin, independently of each other thread on the kernel, making this computation free of collisions. Each thread computes the first for loop (lines 17 to 22) of size T. After this for loop finishes, the remaining R components of the filter are convoluted with the permuted input signal in the second for loop (lines 24-28) of PFTKERN . In the sequential algorithm 2, the permutation $index$ (line 7) has an implicit dependence on the index i of the for loop. In the GPU version, the computation of the indices of the permuted input signal, the filter, and the bins on PFTKERN are explicitly dependent of the index of the thread.

The experimental results show that for input signals with sizes $n < 2^{27}$, the performance of PFTKERN is much better than the performance of the corresponding for loop on the sequential version. The sizes of the for loops on this kernel are $T < 30$ and $R < 1000$, which are efficiently computed by one thread within times that are smaller than the time delays caused by the potentials thread collisions. However, for input signals with sizes $n \geq 2^{27}$, the value of R are in the range $[2000, 250000]$, increasing the time to compute the second for loop on PFTKERN , and therefore degrading the performance of the algorithm. In order to obtain the expected high performance, the kernel PFKERN (lines 32 to 38) was added to the algorithm 3. This kernel is designed for a direct unrolling of the corresponding sequential for loop. PFKERN has therefore the time delays introduced by the collision of about T threads trying to update the same bin, but does not have the time delays caused by the computation of the second for loop on PFTKERN , which are greater than the corresponding collision times. PFKERN has therefore a higher performance than PFTKERN for input signals with sizes $n \geq 2^{27}$. Hence, in order to guarantee the high performance of the algorithm 2 for all the input signals, PFTKERN is selected for input signals with sizes $n < 2^{27}$, and PFKERN for sizes $n \geq 2^{27}$ (lines 7-11).

C. GPU-SFFT: FFT and Cutoff

Algorithm 4 GPU function to compute the FFT of the bins vector in the time domain, and to find and sort the $B_t = 2k$ indices of the largest frequency coefficients in the bins vector in the frequency domain.

```
1: Input: dbins_t[B]
2: Output: dbins_f[B], dJ[B_t]
3: procedure FFTCUTOFFGPU(dJ,dbins_t,dbins_f,B,B_t)
4:     dbins_f ← CUFFT(dbins_t, B)
5:     dJ ← CUTOFFGPU(dJ, dbins_f, B_t, B)
6: end procedure
```

Algorithm 5 GPU function to find and sort the $B_t = 2k$ indices of the largest frequency coefficients in the input vector.

```
1: Input: dŷ[m]
2: Output: dId[B_t]
3: procedure CUTOFFGPU(dId,B_t,dŷ,m)
4:     dsamples_s[m] ← CUDAMALLOC(m)
5:     dsamples_I[m] ← CUDAMALLOC(m)
6:     SKERN(dŷ, dsamples_s, dsamples_I, m)
7:     THRUST::SORT(dsamples_s)
8:     cutoff ← dsamples_s[m − B_t − 1]
9:     id ← 0
10:    CKERN(dId, cutoff, dsamples_I, m, id, B_t)
11:    THRUST::SORT(dId)
12: end procedure

13: procedure SKERN(dŷ,dsamples_s,dsamples_I,m)
14:    i ← threadIdx.x + blockIdx.x * blockDim.x
15:    if i < m then
16:        dsamples_s[i] ← ||dŷ||^2
17:        dsamples_I[i] ← dsamples_s[i]
18:    end if
19: end procedure

20: procedure CKERN(dId,cutoff,dsamples_I,m,id, B_t)
21:    i ← threadIdx.x + blockIdx.x * blockDim.x
22:    if i < m then
23:        if dsamples_I[i] > cutoff and id < B_t then
24:            dId [ATOMICADD(id, 1)] ← i
25:        end if
26:    end if
27: end procedure
```

The algorithms 4 and 5 are the GPU implementation of the FFT and Cutoff stages on Figure 1. In the algorithm 4, the FFT of the bins vector in the domain time is computed using the cuFFT library. The bins vector in the frequency domain is the input to the algorithm 5, on which the $B_t = 2k$ indices of the largest frequency coefficients on this vector are computed

and sorted.
The algorithm 5 allocate device memory for two vectors: $dsamples_s$ and $dsamples_I$. Both vectors are filled with the square of the magnitudes of the input vector on the SKERN kernel. After sorting the $dsamples_s$ vector, with the sort functionality of the Thrust library [14], a cutoff value is computed. The kernel CKERN implements the unrolling of the corresponding for loop on the sequential version of the MIT-SFFT algorithm. The cutoff value is used on CKERN . to compute the indices of the largest frequency coefficients. These largest frequency coefficients are the elements in the $dsamples_I$ with values greater or equal to the cutoff.

D. GPU-SFFT: Function to restrict the indices of the largest frequency coefficients of the input signal.

The function to restrict the the location of the $2k$ largest frequency coefficients in the DFT of the input signal x is a heuristic that was added as a preprocessing stage in the version 2.0 of the MIT-SFFT (see Figure 1), to improve the performance of the algorithm [12]. This function implements an aliasing filter which is very efficient because has not leakage. The algorithm 6 implements the GPU version of this function. The output of the kernel LLCKERN is the filtered input signal dx'. After computing the FFT of dx',$d\hat{y}$, by cuFFT. The set of indices of the $2k$ largest frequencies contained in $d\hat{y}$ are computed by the procedure CUTOOFGPU on the algorithm 5.

Algorithm 6 GPU function to restrict the location of the $2k$ largest frequency coefficients in the DFT of the input signal x.

Input: $dx[n]$
Output: $dJ_2[B_t]$
procedure LOCLARGECOEFGPU(dx,B_t,n,W,dJ_2)
  $dx'[W] \leftarrow$ CUDAMALLOC(W)
  $d\hat{y}[W] \leftarrow$ CUDAMALLOC(W)
  $\sigma \leftarrow n/W, \tau \leftarrow random() * \sigma$
  LLCKERN(dx', dx, W, τ, σ)
  $d\hat{y} \leftarrow$ CUFFT(dx', W)
  $dJ_2 \leftarrow$ CUTOFFGPU($dJ_2, B_t, d\hat{y}, W$)
end procedure

procedure LLCKERN(dx',dx,W,τ,σ)
  $i \leftarrow$ threadIdx.x + blockIdx.x * blockDim.x
  if $i < W$ **then**
    $dx'[i] \leftarrow dx[\tau + i * \sigma]$
  end if
end procedure

E. GPU-SFFT: Reverse hash function

The GPU version of the reverse hash stage of Figure 1 is implemented by the algorithm 7 by unrolling the corresponding for loop on the sequential version of the MIT-SFFT algorithm.The goal of the algorithm 7 is to reverse the hash function (Eq. 1) to compute the true indices of the largest frequency coefficients that have been hashed to non empty bins [12]. The input of the algorithm 7 is the set of indices of frequency coefficients,dJ, computed by the algorithm 5, and the set of indices of frequency coefficients, dJ_2, computed by the algorithm 6. For each location loop, the algorithm 7 computes the set of indices I_L of the largest frequency coefficients that map to J under the hash function and that are in the permuted set of indices $dJ_{2\sigma}$, that is,

$$I_L = \{i_L \in [n] | (h_\sigma(i_L) \in dJ) \cap (i_L \in dJ_{2\sigma})\} \quad (2)$$

where $dJ_{2\sigma} = (dJ_2 * \sigma) \bmod W$. The algorithm 7 implements a voting approach [12] on which every time an index is added to the set I_L, the index will get a vote (line 15). The output of the algorithm 7 is the set of indices dI of the largest frequency coefficients which get a number of votes equal or greater than a given parameter L_t, that is

$$dI = \{i_d \in I_j | dV[i_d] \geq L_t\} \quad (3)$$

where $I_j \in I_L$.

Algorithm 7 GPU function to reverse the hash function and to return the set of indices of the largest frequency coefficients which occurred in at least L_t of the location loops.

Input: $dJ[B]$,$dJ_2[B_t]$
Output: $dI[IF]$
procedure REVHASHGPU(dI,dJ,B_t,B,n,L_t,dJ_2,W,IF,σ)
  $dV[n] \leftarrow$ CUDAMALLOC(n)
  $dV[n] \leftarrow$ CUDAMEMSET($dV, 0, n$)
  $dJ_{2\sigma}[B_t] \leftarrow$ CUDAMALLOC(B_t)
  $dJ_{2\sigma}[\leftarrow (dJ_2 * \sigma)\ mod\ W$
  RHKERN(dI,dJ,dV,$dJ_{2\sigma}$,L_t,IF,W)
end procedure

procedure RHKERN(dI,dJ,dV,$dJ_{2\sigma}$,L_t,IF,W)
  $i \leftarrow$ threadIdx.x + blockIdx.x * blockDim.x
  if $i < B_t$ **then**
    $I_L = \{i_L \in [n] | (h_\sigma(i_L) \in dJ) \cap (i_L \in dJ_{2\sigma})\} \leftarrow i_{L,i}$
    ATOMICADD($dV[i_{L,i}], 1$)
    if $dV[i_{L,i}] == L_t$ **then**
      dI [ATOMICADD($IF, 1$)] $\leftarrow i_{L,i}$
    end if
  end if
end procedure

F. GPU-SFFT: Estimate values

Algorithm 8 GPU function to estimate the values of the largest coefficients given the indices of such coefficients.

Input: $dI[IF]$,$dfilt_f[fs]$,$dbins_f[L * B]$
Output: $h\hat{x}[IF]$
procedure EVALGPU(dI,IF,$dbins_f$,$dfilt_f$,B,n,L,dH_σ)
  $d\hat{x}[IF] \leftarrow$ CUDAMALLOC(IF)
  EVKERN($d\hat{x}$,dI,IF,$dbins_f$,L,n,dH_σ,B,$dfilt_f$)
  CUDAMEMCPY($h\hat{x}, d\hat{x}$)
  return $h\hat{x}$
end procedure

procedure EVKERN($d\hat{x}$,dI,IF,$dbins_f$,L,n,dH_σ,B,$dfilt_f$)
  $i \leftarrow$ threadIdx.x + blockIdx.x * blockDim.x
  if $i < IF$ **then**
    $pos \leftarrow 0$
    for $j \leftarrow 0, L$ **do**
      $id \leftarrow (dH_\sigma[j] * dI[i])\ mod\ n$
      $x'_v[pos] \leftarrow bins_f[h_{\sigma,B}(I[i])]/filt_f[id\ mod\ (n/B)]$
      pos_{++}
    end for
    $d\hat{x}[I[i]] \leftarrow median(x'_v)$
  end if
end procedure

The GPU version of the estimate frequency coefficients stage of Figure 1 is implemented by the algorithm 8 by

unrolling the corresponding for loop on the sequential version of the MIT-SFFT algorithm. This algorithm is based on the theorem that the DFT of a signal in the time domain corresponds to phase rotation in the frequency domain: $x(n-\tau) \Leftrightarrow e^{-2\pi f\tau}\hat{x}(f)$ [9]. The goal of the algorithm 8 is therefore to compute the true values of the largest frequency coefficients that have been hashed to non empty bins, by removing the phase rotation introduced by the permutation and filtering of the input signal in the time domain [12]. The input of the algorithm is the set of indices of frequency coefficients,dI, computed by the algorithm 7. Every thread of the kernel EVKERN of the algorithm runs a for loop of size equal to the total number of loops (L), on which the index $I(i)$ is permuted, and the value of the largest frequency coefficient $bins_f[h_{\sigma,B}(I[i])]$ is divided by the corresponding component of the frequency component of the filter to remove the phase rotation. It is possible that more than one frequency hash to the same bin, hence to compensate errors due to this hash collision, the median of the values computed in the for loop is assigned to the $I(i)$ component of the $SFFT(x) = \hat{x}$. The device memory transfer of the SFFT output signal, $d\hat{x}$ to the host memory, $h\hat{x}$, completes the output of the algorithm.

III. Experiments and results

In this section we present the results of the experiments performed to compare the performance of the proposed GPU-SFFT algorithm with the sequential MIT-SFFT algorithm [10], and with the cuFFT, the NVIDIA CUDA Fast Fourier Transform (FFT) library [15], performances. We were not able to compare the performance of GPU-SFFT with other similar algorithm proposed on reference [16] because neither the code nor the parameters used in their experiments were available.
The input signal, x, to the experiments in the time domain were computed as inverse DFTs of a signals,$\hat{y}$, in the frequency domain with k randomly chosen elements equal to 1 and $n-k$ elements equal to zero. The output SFT signals of the experiments, $\hat{x}$, were compared to the corresponding input signal in the frequency domain. Only the experiments whose results replicated exactly the input signals, that is with no missing components, were accepted as valid results. For all the experiments presented in this section, the Mean Absolute Error (MAE) is defined as:

$$MAE = \frac{1}{k}\sum_{i=0}^{k} |\hat{x}[i] - \hat{y}[i]| \tag{4}$$

All the experiments presented in this section were performed on a Linux server with Ubuntu operating system version 16.04.5, 44 Intel Xeon Gold processors, clock speed 2.1 GHz, and 125 GB of RAM. The GPU in this server is a NVIDIA Titan Xp, with 30 SM, 128 cores/SM, maximum clock rate of 1.58 GHz, 12196 MB of global memory, and CUDA version 10.1 with CUDA capability of 6.1.

A. Experimental results

As an intermediate stage to the design and implementation of GPU-SFFT. we port the MIT-SFFT in C++ to C. This version called MIT-SFFTC is more compatible with the CUDA C language easing the implementation of GPU-SFFT. We used this version for all the experiments included in this section.
The experiments presented in this section were divided in two sets. The first set of experiments have input signals with sizes in the range $[2^{19}, 2^{27}]$, and a level of sparsity of $k = 1000$. The second set of experiments have input signals with sparsity levels in the range $[1000, 43000]$, and a size of $n = 2^{27}$.The parameters used in the experiments are given in the Tables I to IV, the symbols used in these tables to identify the parameters are described in the Sparse Fourier Transform Code Documentation [10].
For the first set of experiments, Figure 2 compares the execution times of the MIT-SFFTC and the GPU-SFFT when the I/O times to transfer the input signal from the host to the device are not included. The GPU-SFFT times reflects the impact of the parallelization by being nearly independent of the signal size and by being much lower than the times of the MIT-SFFTC. The speedup obtained with the GPU-SFFT vs MIT-SFFTC (Figure 6) has a maximum of 17x with an average of 8x. When the I/O times are included, GPU-SFFT is faster than MIT-SFFTC for all signal sizes, and faster than cuFFT for signal sizes greater than 2^{21} (Figure 3). For this case, the speedup obtained with GPU-SFFT vs MIT-SFFTC has a maximum of 4x and an average of 3x, and the speedup of GPU-SFFT vs cuFFT has a maximum of 5x and an average of 3x.

For the second set of experiments, Figure 4 compares the execution times of the MIT-SFFTC and the GPU-SFFT when the I/O times are not included. The GPU-SFFT times are much lower than the times of the MIT-SFFTC in the complete range of sparsity levels. The speedup obtained with the GPU-SFFT vs MIT-SFFTC i(Figure 7) has a maximum of 37x with an average of 27x. When the I/O times are included, GPU-SFFT is still faster than both MIT-SFFTC and cuFFT for all the sparsity levels (Figure 5). For this case, the speedup obtained with GPU-SFFT vs MIT-SFFTC has a maximum of 21x and an average of 13x, and the speedup of GPU-SFFT vs cuFFT has a maximum of 5x and an average of 3x.

$\mathbf{GPU-SFFT\ accuracy}$: Figures 8 and 9 show than the MAE (Equation 4) for both set of experiments is below 10^{-3}, with all the k frequencies components in the output of GPU-SFFT successfully recovered. Hence, GPU-SFFT shows a better performance than both the MIT-SFFTC and cuFFT with high levels of accuracy.

IV. Conclusions and future work

In this paper, we proposed GPU-SFFT a GPU-based parallel algorithm for computing the SFFT of k-sparse signals. GPU-SFFT was designed to achieve a high performance algorithm by unrolling the for loops in the sequential MIT-SSFT [12] to increase the parallelism by maximizing the number of concurrent threads executing independent instructions, GPU-SFFT is 37x times faster than the MIT-SFFT and 5x faster than cuFFT, the NVIDIA CUDA Fast Fourier Transform (FFT) library [15]. For the further direction of this study we will focus on improving the performance of GPU-SFFT and by applying it to the solution of practical problems.

V. Acknowledgements

This research was supported by National Science Foundations (NSF) under Award Numbers CAREER OAC-1925960. The content is solely the responsibility of the authors and does not necessarily represent the official views of the National Science Foundation

References

[1] J. Cheng, M. Grossman, and T. McKercher. *Professional CUDA C Programming*, John Wiley and Sons, Ltd., Indianapolis, Indiana, USA, 2014.

[2] J. W. Cooley and J. W. Turkey. "An Algorithm for the Machine Calculation of Complex Fourier Series", Math. Comp. **19**, No. 90, 297-301, 1965.

[3] J. W. Cooley, P. W. Lewis, and P. D. Welch, "Historical Notes on the Fast Fourier Transform", Proceedings of the IEEE, **55**, No. 10, 1967

[4] G. Danielson, and C. Lanczos"Some improvements in practical Fourier analysis and their application to x-ray scattering from liquids". Journal of the Franklin Institute. 233 (4): 365–380, 1942.

[5] J. Dongarra and F. Sullivan "Guest Editors Introduction to the top 10 algorithms". Computing in Science Engineering, 2 (1): 22–23, January 2000

[6] A. Gilbert, S. Guha, P. Indyk, M. Muthukrishnan, and M. Strauss. "Near-optimal sparse Fourier representations via sampling"., STOC,Proceedings of the thirty-four annual ACM Symposium on Theory of Computers, Montreal,Quebec,Canada,May 2002.

[7] A. Gilbert, M. Muthukrishnan, and M. Strauss. "Improved time bounds for near-optimal space Fourier representations", Proceedings of SPIE,The International Society for Optical Engineering, January 2004.

[8] A. Gilbert and P. Indyk. "Sparse recovery using sparse matrices". Proceedings of the IEEE, Vol. 98, No. 6, 937-947, 2010.

[9] J. O. Smith III *Mathematics of the Discrete Fourier Transform with Audio Applications*, BookSurge Publishing, USA, May 2008.

[10] H. Hassanieh, P. Indyk, D. Katabi, and E. Price. *Sparse Fourier Transform Code Documentation*, https://groups.csail.mit.edu/netmit/sFFT/code.html.

[11] D. B. Kirk and W. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*, Third Edition, Morgan Kauffman, Cambridge, MA, USA, 2017.

[12] H. Hassanieh, *The Sparse Fourier Transform*, ACM Book Series No. 19, Morgan and Calypool Publishers, USA, 2018.

[13] NVIDIA corporation. *CUDA PROGRAMMING GUIDE*, PG-02829-001 V10.1, August 2019.

[14] NVIDIA corporation. *THRUST QUICK START GUIDE*, DU-06716-001 V10.1, August 2019.

[15] NVIDIA corporation. *CUFTT LIBRARY USER GUIDE*, DU-06716-001 V10.1, August 2019.

[16] C. Wang, S. Chandrasekaran, and B. Chapman, "cusFFT: A High-Performance Sparse Fourier Transform Algorithm on GPUs". 2016 IEEE International Parallel and Distributed Processing Symposium, 963-972, May 2016.

TABLE I: Parameters MIT-SFFT ($n = 2^q, k = 1000$)

q	19	20	21	22	23	24	25	26
B	3.85	1.4	2.16	0.683	0.99	0.663	0.662	0.478
Comb-cst	256	128	128	256	256	128	128	128
loc-loops	3	2	2	2	2	2	2	2
est-loops	5	6	5	5	3	5	5	5
thre-loops	2	2	2	2	2	2	2	2
Comb-loops	2	2	2	2	2	2	2	2

TABLE II: Parameters GPU-SFFT ($n = 2^q, k = 1000$)

q	19	20	21	22	23	24	25	26
B	3.85	1.4	2.16	0.683	0.99	0.663	0.662	0.478
Comb-cst	256	128	128	256	256	128	128	128
loc-loops	3	2	2	2	2	2	2	2
est-loops	5	5	5	5	3	5	5	5
thre-loops	2	2	2	2	2	2	2	2
Comb-loops	2	2	2	2	2	2	2	2

TABLE III: Parameters MIT-SFFT ($n = 2^{27}$)

k/1000	1	7	13	19	25	31	37	43
B	0.68	0.77	0.665	0.66	0.79	0.69	0.70	0.8
Comb-cst	128	4096	4096	4096	8192	16384	32768	32768
loc-loops	3	2	2	2	2	3	3	3
est-loops	4	5	9	9	9	9	10	9
thre-loops	2	2	2	2	2	2	2	2
Comb-loops	2	2	2	2	2	2	2	2

TABLE IV: Parameters GPU-SFFT ($n = 2^{27}$)

k/1000	1	7	13	19	25	31	37	43
B	0.68	0.77	0.665	0.66	0.79	0.69	0.70	0.8
Comb-cst	128	4096	4096	4096	8192	16384	32768	32768
loc-loops	3	2	2	2	2	3	3	3
est-loops	4	5	7	9	9	9	8	8
thre-loops	2	2	2	2	2	2	2	2
Comb-loops	2	2	2	2	2	2	2	2

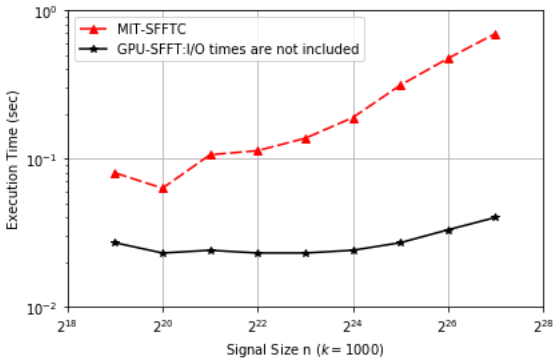

Fig. 2: Execution times vs signal size. I/O times no included

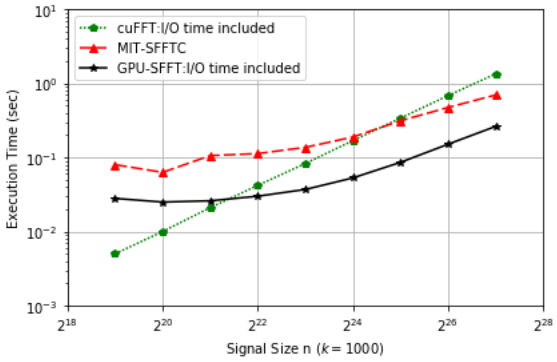

Fig. 3: Execution times vs signal size. I/O times included.

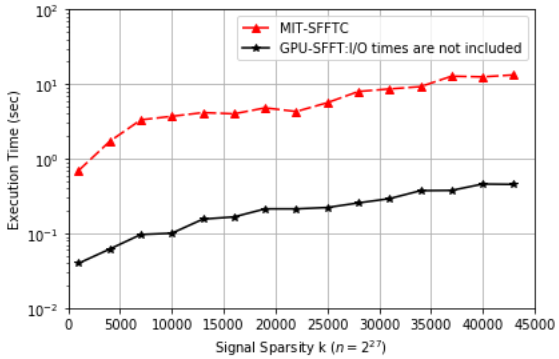

Fig. 4: Execution times vs signal sparsity. I/O times no included.

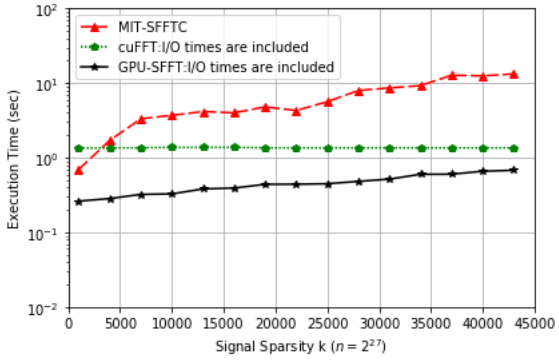

Fig. 5: Execution times vs signal sparsity. I/O times included.

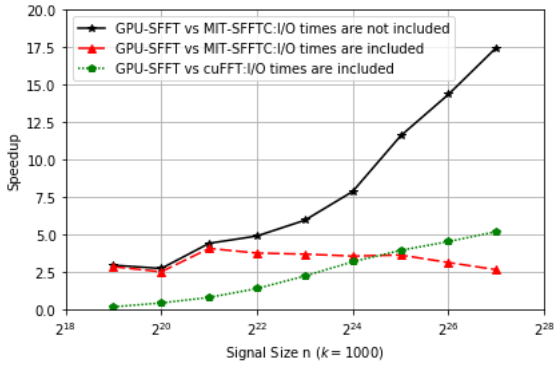

Fig. 6: Speedup of GPU-SFFT vs input signal size.

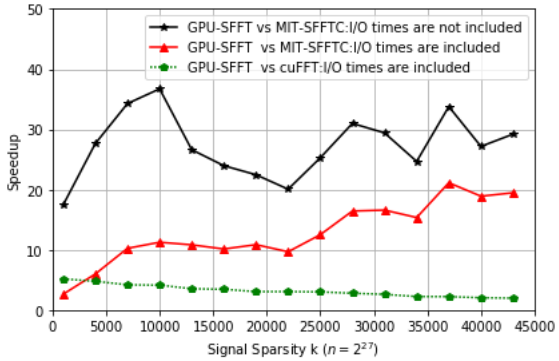

Fig. 7: Speedup of GPU-SFFT vs input signal sparsity.

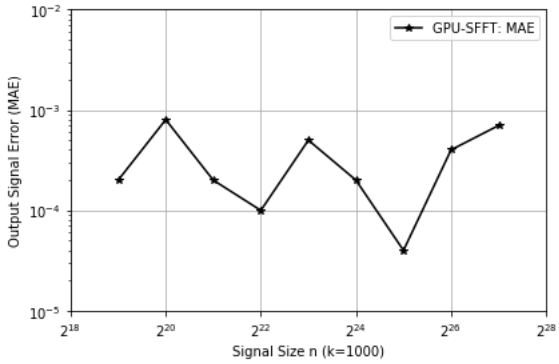

Fig. 8: MAE vs input signal size.

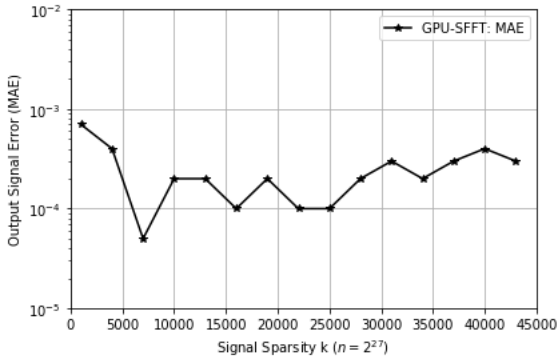

Fig. 9: MAE vs input signal sparsity.