Replicating GPU-SFFT, a GPU based parallel algorithm for computing the Sparse Fast Fourier Transform(SFFT)

Guy Bar Yosef

May 14, 2020

Contents

1 Problem Statement and Motivation

In this project we attempt to implement the GPU-SFFT algorithm described in [1]. We aim to achieve both the speedup improvements specified in the paper, 23x faster than MIT-SFFT and 5x faster than cuFFT, as well as maintain a similar output accuracy as the MIT-SSFT.

2 Introduction and Background

The Fast Fourier Transform (FFT) is an algorithm developed to compute the Discrete Fourier Transform with a time complexity of $O(n\ log\ n)$.

A variation of the FFT, called the Sparse Fast Fourier Transform (SFFT) was developed by MIT in a series of algorithms called MIT-SFFT described in [4]. This set of algorithms attempt to take advantage of the sparsity of signals to compute an even more computationally efficient FFT, bringing the runtime down to $O(log\ n \sqrt[3]{nk^2 log\ n})$, for a signal of size n with only k non-zero frequency coefficients ($k << n$). Even if the input signals are not sparse the MIT-SFFT algorithms will provide a sparse approximation to their FFT. However these SFFT algorithms are sequential in nature and therefore not utilizing the inherent parallel nature of GPUs.

Artiles and Saeed, in [1], present a GPU based parallel algorithm for computing the SFFT, based on the sequential MIT-SFFT algorithms. Their implementation, which they call GPU-SFFT, attempts three main optimizations:

1. To utilize the parallel execution possible in GPUs to unroll the for-loops in MIT-SFFT as well as to ensure coalesced global memory access by the threads in each warp.
2. Minimize the transfer of data between the CPU and GPU.
3. Replace sequential sorting algorithms with the NVIDIA's Thrust library [5] and to compute the reduced-size FFT using NVIDIA's FFT implementation, cuFFT [6].

Artiles and Saeed describe their paper's contribution as twofold:

1. Proposing GPU-SFFT, the prallelized SFFT algorithm.
2. Showing that GPU-SFFT is a high performance algorithm without reducing the accuracy of its output, as compared to MIT-SFFT.

In this project we attempt to replicate both of these contributions.

Artiles and Saeed provide detailed pseudo-code for GPU-SFFT's implementation. Nevertheless the algorithm is relatively complex and so we hope to still gain insight into how a complex algorithm could be implemented on a GPU while taking advantage of different memory optimization techniques such as memory coalescing, shared memory, etc.

2.1 CUDA

The Compute Unified Device Architecture, or CUDA, is an API created by Nvidia to develop software that utilizes Nvidia's GPU resources. There is a C/C++ implementaion that can be compiled using Nvidia's proprietary nvcc compiler.

The particular computer we used in our project is the Nvidia Jetson Nano. The Jetson Nano includes both a CPU and a GPU, allowing us to run the sequential version, MIT-SFFT, and our GPU implementation, GPU-SFFT, sequentially and compare results.

3 Implementation

The SFFT algorithm uses a filter for its frequency binning. As per [4], there are several possible filters to choose from. The one that we chose in our implementation is what they consider the simplest: A Gaussian filter convolved with a rectangular window. We derive this filter, as well as its time-domain representation, on the CPU for both the CPU and GPU implementations, as a preprocessing step before the MIT-SFFT algorithm begins.

3.1 CPU Implementation

The Sparse Fast Fourier Transform algorithms are a set of algorithms developed by MIT [4]. The algorithms themselves are available to the public and can be found here[1]. Nevertheless, one of the goals of this project is to compare the runtimes of the CPU vs the GPU, and as such we coded a CPU-variant of the SFFT that attempts to model as closely as possible the pseudo-code of GPU-SFFT.

The only differences between our CPU implementation and the GPU implementation are:

1. The FFT implementations: In our CPU implementation we used the open-source library FFTW (Fastest Fourier Transform in the West) [2], while our GPU implementation used NVIDIA's cuFFT [6].

2. All the parallelized loop techniques used in GPU-SFFT are transformed into sequential for-loops.

3. Our CPU implementation uses C++'s standard template library, specifically std::sort function and std::vector, while our GPU implementation uses NVIDIA's Thrust library [5].

4 Discussion

We ran into several areas of trouble during our implementation. In our opinion, this was mostly due to the design of the paper: The paper attempted to create a parallelized GPU version of an existing algorithm, the MIT-SFFT. As such, the paper did not go into algorithmic details, choosing to skip on motivations and explinations and instead provide pseudo-code. This let us to be able to implement most of the algorithm fairly quickly, but then get bogged down in several parts that had us concluding that the pseudo-code is either inaccurate or skipping intermediate steps.

Some of the more confusing parts for us include:

1. In the kernel used in the *RevHash* function (algorithm 7 in the paper), the pseudocode specified the following set:

$$I_L = \{i_L \in [n] | (h_\sigma(i_L) \in dJ) \cap (i_L \in dJ_{2\sigma}\} \tag{1}$$

   This set is supposed to be the set of indices of the largest frequency coefficients that map to J under the hash function. The specific variable details here are:

   - n is the length of the input vector.
   - h_σ is the hash function which hashes to indices to bins.

[1] https://groups.csail.mit.edu/netmit/sFFT/code.html

- dJ is the set of indices of the largest frequency coefficients of the input vector.
- $dJ_{2\sigma}$ is the set of indices of the largest frequency coefficients after a preprocessing restriction.

However it was unclear to us where we were supposed to generate these indices from. The hash function, as specified in Eq. 1 with the σ subscript, was not defined in the paper previously. The paper instead provided a slightly different formulation, shown in Eq. 2. Although the difference is slight, the lack of a B in the subscript confuses the meaning of the hash function, especially because neither dJ or $dJ2$ have n elements. As such, our intuition, Eq. 3, does not seem probably.

$$h_{\sigma,B}(i) = floor\left(\frac{i\sigma}{n/B}\right) \tag{2}$$

$$h_{\sigma}(i) = floor\left(\frac{i\sigma}{n}\right) \tag{3}$$

We were unsure how implement this, and settled for taking the indices sequentially from dJ (the set described in Eq. 1 is supposed to grow over the iterations). This meant that we were not using $dJ2$ at all and so we were wasting parts in other parts of the algorithm generating it, but we kept it anyways, assuming that had we understood the implementation correctly it would have been used.

2. In the kernel part of the eval function (Algorithm 8 in the paper), there is, in our understanding, a redundant variable called *pos*. According to the pseudo-code, it seemingly replicates the value in another variable, j, over which a loop is performed. Moreso, the kernel specifies to use the hash function described in Eq. 2, however it does not specify a σ to use and is also under-determined.

We also encountered difficulty when trying to implement unified memory. Unified memory [3] is a concept introduced in CUDA programming as memory that can be accessed in both the host (CPU) and device (GPU). Under the hood, this 'unified' memory could be better described as 'managed' memory, as the CUDA compiler abstracts away the copying that is necessary to move the memory between the two devices (in more modern NVIDIA GPU architectures, the necessary page mapping is handled by the hardware as apposed to the software). This is hinted at by the way one creates a unified memory array: calling the *cudaMallocManaged* function.

Unified memory is used twice in the provided pseudo-code, however it requires a computing capability of 6.0 and above, while the Jetson Nano only has a computing capability of 5.3. Therefore we bypassed it, probably at an efficiency loss, by copying the memory back and forth between the host and device.

Note that this significantly hinders, if not completely nullifies, the 2nd goal of the GPU-SFFT: to minimize the transfer of data between the CPU and GPU.

Finally, we were experiencing a shortage of GPU global memory. In part of the pseudocode, there is an explicit instruction to create an array of size L in one of the kernel functions. This however led to the error "too many resources expected for launch". We modified this kernel function to do without the array (seeing as we were anyways not getting realistic results and being under a time crunch). This modification essentially replaces taking a median of a set with the mean of a set.

As per all these descriptions above, we know our results to be incorrect. Not only is the

5 Conclusion

The GPU-SFFT is a parallelized version of the very popular Sparse Fast Fourier Transform(SFFT) algorithms developed by MIT. By improving on the runtime of the SFFT algorithms, which are themselves an improvement over the standard Fast Fourier Transform implementation, the GPU-SFFT becomes a practical and important tool for many signal processing techniques.

However, we ran into much difficulty when trying to replicate the pseudo-code laid out in the paper. Some of it was on my end, lacking the algorithmic understand necessary for the implementation. However other issues were out of our hands, such as the GPU-SFFT algorithm GPUs with a higher compute capability and RAM than the Jetson Nano.

Nevertheless, we believe this experience gave us a larger understanding of CUDA in general, including its different libraries, different NVIDIA GPU architectures, and the way memory is transferred between host and device.

References

[1] O. Artiles and F. Saeed. "GPU-SFFT: A GPU based parallel algorithm for computing the Sparse Fast Fourier Transform (SFFT) of k-sparse signals". In: *2019 IEEE International Conference on Big Data (Big Data)*. 2019, pp. 3303–3311.

[2] Matteo Frigo and Steven G. Johnson. "The Design and Implementation of FFTW3". In: *Proceedings of the IEEE* 93.2 (2005). Special issue on "Program Generation, Optimization, and Platform Adaptation", pp. 216–231.

[3] Mark Harris. *Unified Memory for CUDA Beginners*. 2018. URL: `https://devblogs.nvidia.com/unified-memory-cuda-beginners/`.

[4] Haitham Hassanieh et al. "Simple and Practical Algorithm for Sparse Fourier Transform". In: *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA '12. Kyoto, Japan: Society for Industrial and Applied Mathematics, 2012, 1183–1194.

[5] *Thrust CUDA Toolkit*. 2019.

[6] *cuFFT CUDA Toolkit*. 2019.

6 Appendix A: Instructions

6.1 CPU Implementation

For the CPU implementation, we use a FFT library called FFTW. To install it, you need to execute:

```
wget fftw.org/fftw-3.3.8.tar.gz

tar -zxf fftw-3.3.8.tar.gz
cd fftw-3.3.8
./configure
make
sudo make install
```

To compile and execute, run:

```
make
./gpuSfft ../inputs/SOME_INPUT.txt ../outputs/small_output.txt
```

6.2 GPU Implementation

For the GPU implementation, we need the CUDA toolkit installed, in particular the nvcc compiler, cuFFT and thrust libraries.

To compile and execute, go to the gpu/ directory and execute:

```
make
./gpu_gpuSfft.out ../inputs/SOME_INPUT.txt ../outputs/small_output.txt
```