A GPU Solution for Signal Anomaly Detection in the Frequency Domain

Nathan Jordan
Department of Computer Science
University of Nevada, Reno
Reno, Nevada 89512
Email: njordan@cse.unr.edu

Lee Barford
Agilent Technologies
Santa Clara, CA
Email: lee.barford@gmail.com

Fred Harris
Department of Computer Science
University of Nevada, Reno
Reno, Nevada 89512
Email: fred.harris@cse.unr.edu

***Abstract*—As the extensibility of GPU computing rapidly increases, we often find them useful for different applications in the field of science and engineering. Libraries written for engineering tasks such as CULA (Cuda Linear Algebra), CUFFT (Cuda Fast Fourier Transform), and CUBLAS (Cuda Basic Linear Algebra Subprograms) have made it easier for programmers to achieve a significant performance increase when solving problems in the fields of engineering and math. In signal processing we can use the GPU to perform discrete Fourier transforms on time-domain signal strength to represent the data in the frequency domain. With the data in this format, we can calculate signal strength of various frequencies very efficiently, and further determine if a transmission on a particular frequency has taken place. Speedups in excess of 70 were achievable using a GPU-based implementation utilizing the cuFFT library over a CPU implementation utilizing the most performance optimized CPU-based FFT library, FFTW.**

I. Introduction

The ubiquity and necessity of parallel computing has begun to show itself in recent years, and companies have found the need to adapt their hardware to facilitate parallelism. This is becoming a necessary change to allow for speedups that historically would be delivered by an increase in CPU clock frequency. NVIDIA, harnessing the inherent SIMD (single instruction multiple data) properties of their GPU's, created the CUDA API, which allows programmers to write C/C++ code that takes advantage of the hundreds or thousands of cores present on modern NVIDIA GPU's, without having to learn a new language, or have intimate knowledge of the hardware.

Consequently, many applications that were formerly bound by the lack of floating point throughput of most CPU's can take advantage of the massive parallelism that a GPU can provide. Applications ranging from scientific and engineering number crunching to sorting algorithms (which run many times faster than their CPU counterparts [1]). Even though the performance of everyday algorithms can be improved by the GPU, the place where they most outperform the CPU is in applications which require large amounts of floating point arithmetic. Most scientific and engineering-related fields have large and computationally difficult problems that need to be solved as fast as possible. In this paper, we will look at a potential application of the GPU to solve problems in the field of Electrical Engineering; more specifically signal analysis and processing.

If we are working with a radio signal, we can take the digitized signal and use the GPU to detect anomalies in a particular frequency range. If we can detect that a transmission has occurred on a known frequency, then we can intercept that transmission (provided it is not encrypted), or attempt to triangulate the position of the signal using multiple antennas that are synchronized to a certain degree. This paper presents a GPU algorithm to perform this task significantly faster than a CPU using well-optimized libraries.

Many implementations for signal detection already exist. Wireless communication such as the varying cell network technologies rely on signal detection and correction to function. Popular methods include the QRM-MLD allow for efficient frequency domain signal detection as proposed by T Yamamoto[2], and frequency domain signal processing methods by Howard et. all [3].

The rest of this paper is organized as follows: first some of the background information surrounding the topic will be covered; second, the sequential implementation will be discussed; third, the parallel implementation, including pipelined and non-pipelined versions, will be explained; finally, we will end the paper with some concluding remarks and the future direction of the project.

May 9th, 2013

II. Background

In engineering, the Fourier Transform is applicable to many topics such as differential equations, spectroscopy, and quantum mechanics. In electrical engineering and signal processing, the Fourier transform can be used to switch between the time and frequency domains. Information in the time domain by itself it of little use when trying to infer information about a signal, as it is a representation of many different bands of frequencies superimposed on top of each other. If we perform a Fourier Transform on a signal represented in the time domain, the result is a frequency spectrum that provides the resulting values in the complex plane. The frequency spectrum can be used to see which frequencies have the greatest power in a particular signal. If the signal we are analyzing is a radio wave generated by an antenna, we can determine which frequencies are being transmitted on, and quite possibly intercept the transmission as well.

There are several implementations of the Fourier Transform. The one most commonly used by engineers in real applications is the FFT, or Fast Fourier Transform. As one may suspect, the FFT gets its name from the efficiency in which it computes the transform compared to other methods, however despite its speed, it does not compromise accuracy. There are a few implementations of the FFT, the most common of which being the Cooley- Turkey algorithm, and another being the Bluestein's algorithm, which is used in some cases when the FFT size is near a prime number (where Cooley-Turkey performs poorly).

The FFTW library is one of the most popular FFT libraries in use in open-source software. It is also unsurprisingly by far and away the fastest open-source FFT library. Its only competition is Intel Math Kernel Library (MKL), a proprietary software developed by Intel for the Intel Compiler. These two FFT libraries are comparable in performance, with MKL usually outperforming FFTW for larger FFT's. In this project, we will be using FFTW as our FFT library, due to its popularity, openness, and speed.

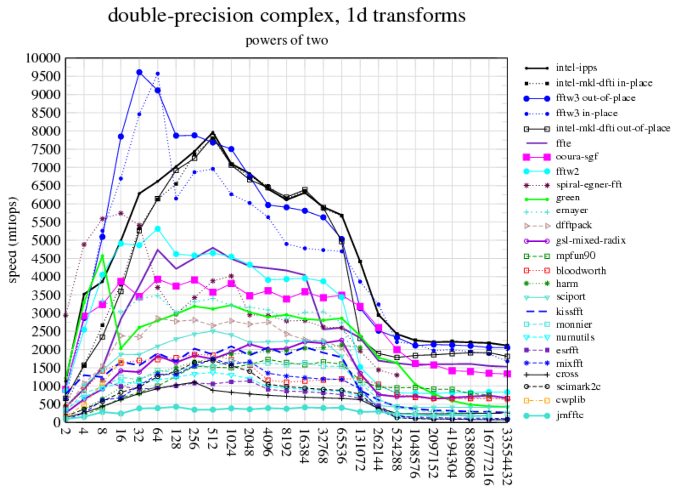


Fig. 1. Benchmarks comparing the performance of FFTW, MKL and other FFT libraries. FFTW and MKL are the fastest, shown in blue and black respectively. [4]

With the introduction of CUDA in 2008, NVIDIA has stepped into the high-performance computing arena. Due to their inherently parallel nature and fast floating point operations, GPU's handily lend themselves to many engineering problems. NVIDIA has created their own implementation of the FFT, called cuFFT (CUDA FFT) for the convenience of programmers writing engineering software. cuFFT is modeled after its CPU-based counterpart FFTW, and implements its algorithms in a similar fashion. To speed up the transition for programmers used to FFTW, NVIDIA made the API for cuFFT almost identical to that of FFTW, making it easy to port existing code to the GPU for a significant speedup. For the parallel implementation of this project, cuFFT was used due to its similarity to FFTW for comparison purposes, and more importantly, to avoid implementing an FFT manually on a GPU.

III. Sequential Implementation

The sequential version of the program is relatively straightforward. Initially, the input to the program is a binary flat file

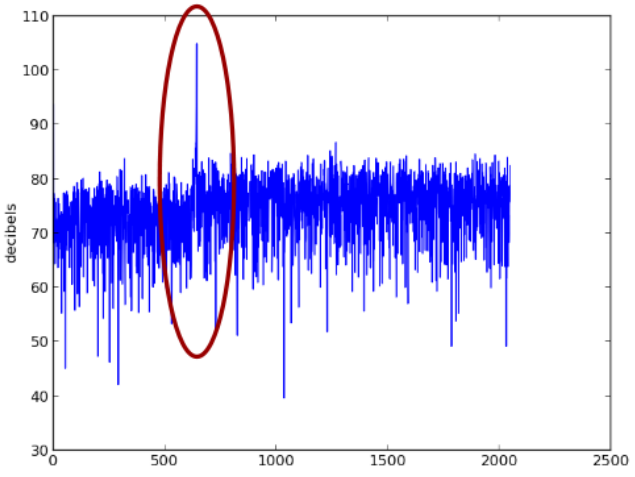


Fig. 2. Following the FFT calculation and signal analysis, a transmission is clearly visible in the frequency domain

consisting of 8-bit integers representing the signal received by an antenna in the time domain. Each 8-bit integer is converted to a single-precision floating point number, and added to a vector for later processing. After the data is read from the file, arrays are allocated for the FFT computation and the main algorithm begins. An FFT of size 4096 (2^{12}) was chosen as it exhibits the best performance when using FFTW and cuFFT. A real-to-complex FFT is performed incrementally on the input signal information and is stored in a result vector that contains both the real and complex portions of the transform as a complex conjugate. After the FFT is calculated, the modulus of the complex result are calculated for each frequency bin to give us the magnitude of the frequency, as seen in Equation 1.

$$|z| = \sqrt{x^2 + yi^2} \tag{1}$$

Once we have the magnitude of each of the frequency bins calculated by the FFT, we can put them into the logarithmic decibel (dB) scale by using the formula described by equation 2.

$$Z = 20 \times \log_{10}(|z|) \tag{2}$$

After this calculation there are two different approaches we can take. The first, which is much simpler, compares the computed signal strength to a constant value set before runtime. If the signal strength for a particular frequency bin exceeds this value, we assume that a transmission has begun on that particular frequency. We can filter out some noise by applying another constant value that is the amount of time that the signal strength for a particular bin must exceed the signal strength constant in order to classify it as a transmission instead of random noise. Given the unsophisticated nature of this method, it is unsurprising that it is not the most most reliable method to determine a legitimate transmission.

A second approach is to dynamically modify these values as the signal strengths are computed during runtime. This allows for the level of noise to change during the transmission,

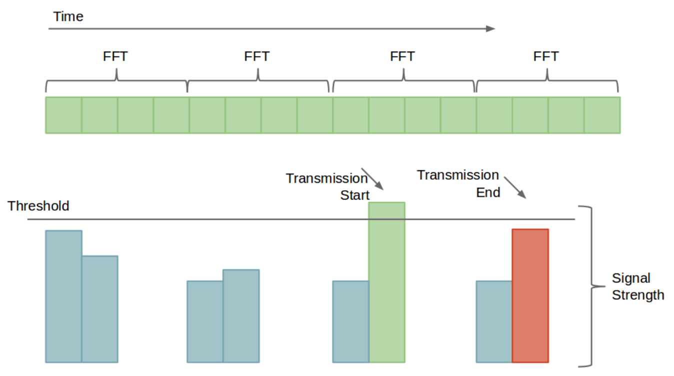


Fig. 3. A graphic showing the operation of the sequential algorithm.

and can be used as a more reliable method to distinguish between noise and actual transmissions. This increased accuracy comes at the cost of additional computational overhead during the execution of the algorithm.

Once a transmission is detected, a new entry for it is created in a transmission log vector containing its frequency, maximum strength, and start time. Also during each iteration, the algorithm will then check to see if a current transmission's strength level has fallen below the threshold value. If this is the case, its end time will be added to the transmission log. Any subsequent rise of that frequency over the threshold value will trigger the creation of a new transmission log.

Once the algorithm has been completed, the transmission log is written to a file of the users choosing for further analysis.

IV. Parallel Implementation

With the parallel implementation, we use an algorithmically-similar approach to our sequential implementation. Like the sequential version, we will be calculating the FFT of our input signal information to transform it to the frequency domain and calculating the complex modulus of the result. Unlike the CPU implementation, however, the signal information is not directly available to the card, and must be copied across the PCI Express (PCIE) bus. The PCIE bus is orders of magnitude slower than any kind of operation performed on a GPU or CPU, including memory accesses. As such, the PCIE bus often becomes a performance bottleneck for GPU programs, especially when the amount of data to be transferred is quite large.

To solve this, we can use a technique called pipelining, which allows us to minimize this bottleneck. Pipelining, which is similar to the instruction pipeline used in the CPU, breaks memory transfer and computation of data into different parts that can be performed independently of each other. Blocks of data are transferred one at a time across the PCIE bus and are computed on once they arrive on the GPU. This way, we can start computing the FFTs and signal analysis right away without having to wait for the entire chunk of data to be transferred across the PCIE bus. For this paper, a non-pipelined approach will be implemented first for comparison purposes, and a pipelined version will follow.

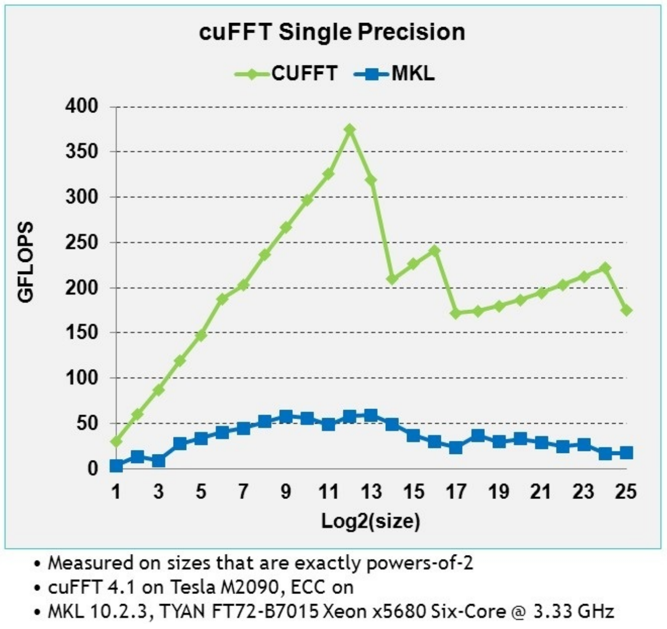


Fig. 4. A single-precision performance comparison between cuFFT and the Intel MKL Library [5]

A. *Non-Pipelined Approach*

To demonstrate the simplicity of achieving a speedup over a sequential implementation, a somewhat trivial approach was implemented first. In this implementation, the algorithm resembles the CPU version more closely than its pipelined counterpart. The signal data is transferred in one large chunk from the host to the GPU, after which the FFT is performed using the cuFFT API and the transmissions are detected using custom CUDA kernels run in parallel. Considering this methods simplicity, it has a few notable limitations. Firstly, we are limited to only computing on as much data that can be held in GPU memory. Secondly, and perhaps more importantly, this method cannot be used in realtime applications, as we are limited to performing the operations required on a finite set of data. Despite these limitations, this method is still able to achieve significant speedup over the sequential implementation.

B. *Pipelined Approach*

To hide the latency created by memory transfers across the PCIE bus, we can implement a pipelining scheme. In this implementation, smaller pieces of data are transferred incrementally to the device. This allows the GPU to begin work on some parts of the data as others are still being transferred over. If each of N pipeline stages are near equal in terms of time required to complete them, we can achieve a theoretical speedup of N over a non-pipelined solution.

To do this, we split our algorithm up into four parts, or stages, to be pipelined. The first stage transfers signal data from the host memory to the GPU. The second stage uses cuFFT to perform an FFT of size 4096 on the input data. The third stage performs signal processing. More specifically, it calculates the complex modulus across multiple FFT results to maximize GPU efficiency, as performing an operation on only 4096 elements will result in GPU memory latency, as many cores will be waiting for memory access. The fourth and final stage detects transmissions by finding frequency bins that exceed the signal threshold. If a frequency has exceeded the threshold it will be noted in a bit array of size 4096. If the signal strength of a frequency that is active in the bit array has fallen below the threshold, the information about that transmission will be noted in several device buffers for later transfer to the host. In the current implementation, it was decided that due to the relatively small amount of transmission data (in the range of 1KB) to be transfered back to the host, that this action should not be pipelined. If re-implemented for realtime analysis of radio signals, this process would have to be pipelined to allow transmissions to be discovered while the program is running and processing signal data. The pipelined process is illustrated in figure 5.

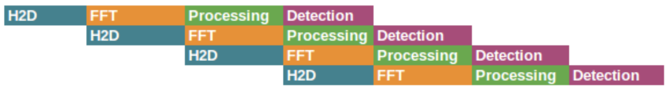


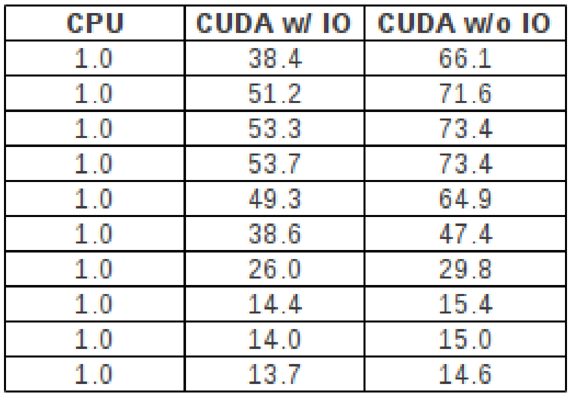
Fig. 5. A graphic briefly showing the pipelined approach to the problem. Each stage is shown in a different color and can be performed independently of one another.

V. Results

The unpipelined implementation of the GPU program achieved a significant speedup over its CPU counterpart, which peaked at around 53. As the number of elements increased however, its speedup began to fall and stabilize at around 13 or 14. This is due to two reasons. The first of which is a sub-optimized transmission recording kernel. The kernel requires atomics to increment the number of transmissions detected, as well as adding a new transmission to the transmission information buffers, which requires atomics to ensure that other threads do not overwrite each others transmissions. This kernel operation could be much further optimized to reduce this performance bottleneck. Secondly, the lack of a dynamic signal threshold creates many more transmissions in a high-noise section of the signal than a low-noise section. Consequently, in these high noise sections that were encountered later in the input file, many more transmissions were created and made the problem even more pronounced. Implementing a dynamic signal threshold would likely solve this issue. The speedup over the CPU version are illustrated in Figures 6 and 7.

The efficiency of the algorithm is quite low. This is largely due to the memory copy to the device from the host, during which time the device is idle. The pipelined approach should solve that problem, and increase efficiency to a more acceptable range. Throughput was significantly higher for the GPU version, both with and without I/O included. Again, pipelining the solution would likely increase the throughput of the GPU version significantly.

Unfortunately, the pipelined implementation was unable to be finished. There were a few different problems that were encountered that presented a significant challenge for this implementation. First and foremost was a problem surrounding the implementation of cuFFT. In the Cuda API, kernel calls are asynchronous as well as asynchronous memory copies.



CPU	CUDA w/ IO	CUDA w/o IO
1.0	38.4	66.1
1.0	51.2	71.6
1.0	53.3	73.4
1.0	53.7	73.4
1.0	49.3	64.9
1.0	38.6	47.4
1.0	26.0	29.8
1.0	14.4	15.4
1.0	14.0	15.0
1.0	13.7	14.6

Fig. 6. Table displaying GPU speedup over CPU implementation with and without I/O.

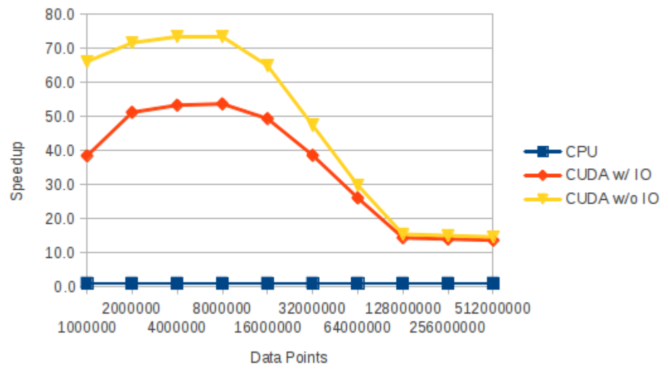
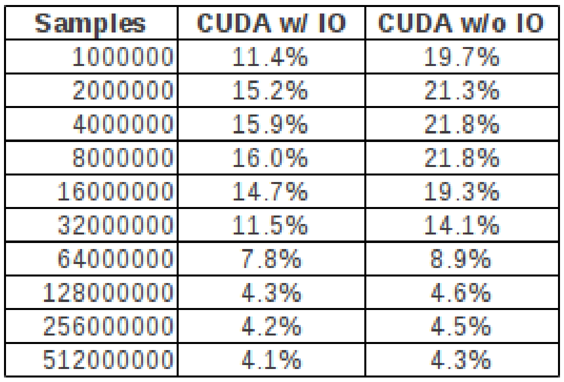


Fig. 7. Graph displaying GPU speedup over CPU implementation with and without I/O.



Samples	CUDA w/ IO	CUDA w/o IO
1000000	11.4%	19.7%
2000000	15.2%	21.3%
4000000	15.9%	21.8%
8000000	16.0%	21.8%
16000000	14.7%	19.3%
32000000	11.5%	14.1%
64000000	7.8%	8.9%
128000000	4.3%	4.6%
256000000	4.2%	4.5%
512000000	4.1%	4.3%

Fig. 8. Table with GPU efficiency with and without I/O. (Based on 336 Cuda cores in the Nvidia GTX460)

Adding an asynchronous memory copy and a kernel call to a Cuda stream is an effective and relatively straightforward way of implementing a GPU pipeline. For this to work, each of the calls in the stream must be asynchronous, as the program running on the CPU must continue to make calls to the Cuda API to manage the other streams. If a blocking call is placed in the pipeline, the other streams will need to wait for that particular operation to finish, which defeats the purpose of creating a pipeline in the first place since different steps cannot be executed at the same time. The cuFFT API call

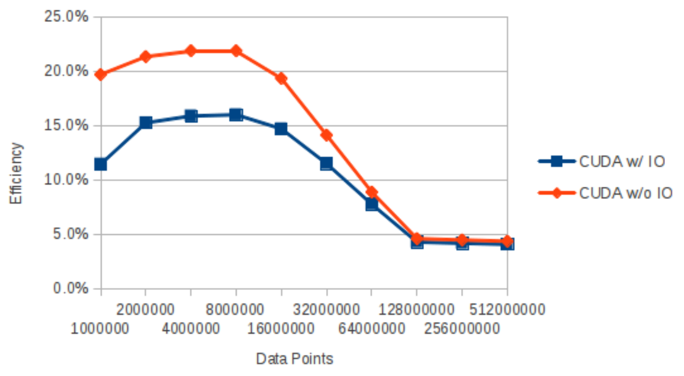
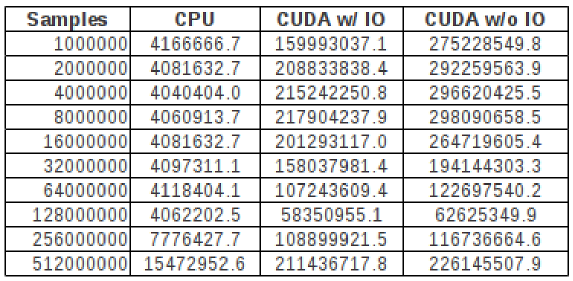


Fig. 9. Graph showing GPU efficiency with and without I/O. (Based on 336 Cuda cores in the Nvidia GTX460)



Samples	CPU	CUDA w/ IO	CUDA w/o IO
1000000	4166666.7	159993037.1	275228549.8
2000000	4081632.7	208833838.4	292259563.9
4000000	4040404.0	215242250.8	296620425.5
8000000	4060913.7	217904237.9	298090658.5
16000000	4081632.7	201293117.0	264719605.4
32000000	4097311.1	158037981.4	194144303.3
64000000	4118404.1	107243609.4	122697540.2
128000000	4062202.5	58350955.1	62625349.9
256000000	7776427.7	108899921.5	116736664.6
512000000	15472952.6	211436717.8	226145507.9

Fig. 10. Table of throughput (in samples/second) of sequential and parallel versions of the program.

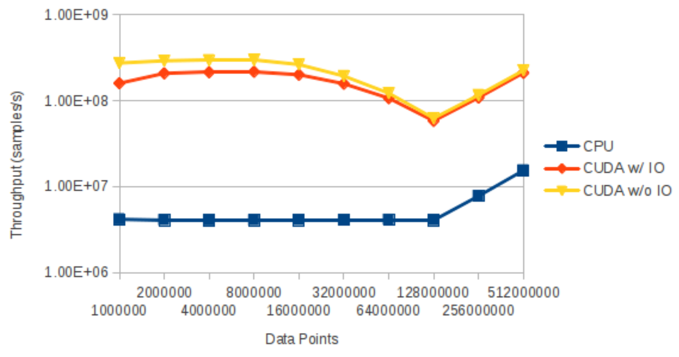


Fig. 11. Table of throughput (in samples/second) of sequential and parallel versions of the program.

to execute a FFT is blocking for an unknown reason. With the FFT being critical to the following steps of the algorithm, we cannot avoid this outright. A possible solution would be to execute the streams each in their separate thread, which would allow the blocking calls to occur without disrupting the other streams. However when calling the FFTs in separate threads, the result was runtime errors. The pipelined version also needs a more complicated transmission detection kernel. Since there are several FFT's and processing being performed at different times, we must traverse all of these to determine when and how long a transmission has taken place.

VI. Conclusion

In this paper, a method for performing signal processing and transmission detection on a GPU was presented. Even with algorithmic deficiencies and less-than-optimized kernels the program was able to achieve a 53 times speedup. Used in a relatively low-noise environment with sparse communication, the method should perform similar to the maximum speedup. With additional work and testing, a pipelined version could also be implemented. This version should improve performance when processing very large sets of signal data by about a factor of four. If implemented efficiently, this approach could be used to monitor radio traffic across a number of frequencies in realtime, or perhaps be used to triangulate the position of multiple radio transmissions simultaneously.

References

[1] E. Sintorn and U. Assarsson, "Fast parallel gpu-sorting using a hybrid algorithm," *Journal of Parallel and Distributed Computing*, vol. 68, no. 10, pp. 1381–1388, 2008.

[2] T. Yamamoto, K. Takeda, and F. Adachi, "Frequency-domain block signal detection with qrm-mld for frequency-domain filtered single-carrier transmission," in *Vehicular Technology Conference Fall (VTC 2010-Fall), 2010 IEEE 72nd*, 2010, pp. 1–5.

[3] S. Howard and K. Pahlavan, "Measurement and analysis of the indoor radio channel in the frequency domain," *Instrumentation and Measurement, IEEE Transactions on*, vol. 39, no. 5, pp. 751–755, 1990.

[4] "3.0 GHz intel core duo, intel compilers, 64-bit mode." [Online]. Available: http://www.fftw.org/speed/CoreDuo-3.0GHz-icc64/

[5] "CUFFT | NVIDIA developer zone." [Online]. Available: https://developer.nvidia.com/cufft