Kalpa Publications in Computing

Volume 17, 2023, Pages 1–14

Proceedings of 2023 Concurrent Processes Architectures and Embedded Systems Hybrid Virtual Conference

Accelerating Scientific and Engineering Applications through Cloud-based GPU Computing

Dipesh Rawat[1], Kopal Chakravarty[2], Neelaksh Singh[3], Vijaya Laxmi Pachva[4], and Rakshith Anand Bhootham[5]

[1] Guru Gobind Singh Indraprastha University, Delhi, India
dipeshrawat66@gmail.com
[2] LNM Institute of Information Technology, Jaipur, India
kchakravarty29@gmail.com
[3] Birla Institute of Technology, Mesra, Jharkhand, India
neelaksh48@gmail.com
[4] Kakatiya University, Warangal, India
vaishnavivijayalaxmi.p@gmail.com
[5] Jawaharlal Nehru Technological University, Hyderabad, India
rakshitanand23@gmail.com

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 Transforms, and cuBLAS library- 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.

1 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 GPUs, 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 GPUs, 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 [4]). Although GPUs may significantly improve the performance of algorithms that display data parallelism, it's vital to understand that not all algorithms are suitable for GPU acceleration. Due to memory constraints and data transmission costs, complex, irregular, or data-intensive algorithms may not benefit from the projected speedup and may instead perform worse on GPUs. Considering the peculiarities of the algorithm and the capabilities of the GPU being used, a careful assessment is required to decide whether GPU acceleration is advantageous for a certain application. 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.

It is important to note that while GPUs offer substantial advantages in parallelism and throughput, there are known issues that make them unsuitable for certain types of High-Performance Computing (HPC) algorithms. GPUs are primarily designed for single-precision arithmetic, which means they excel at performing calculations with 32-bit floating-point numbers. However, many HPC applications, especially those in scientific computing, require higher precision, typically 64-bit double-precision arithmetic. GPUs do support double-precision operations, but they are significantly slower than single-precision operations. This can lead to performance bottlenecks when dealing with algorithms that heavily rely on double-precision calculations. The IEEE defines standards for floating-point arithmetic, including the IEEE 754 standard for floating-point representation. GPUs may not always adhere strictly to these standards, which can lead to numerical instability and inaccuracies in certain calculations. This lack of compliance can be a concern for HPC applications where precision and consistency are critical. GPUs are optimized for data-parallel workloads, where the same operation is applied to a large dataset concurrently. While they excel in terms of throughput for such workloads, they may face challenges in handling irregular or memory-bound computations. In HPC applications that involve complex memory access patterns or require frequent data transfers between CPU and GPU memory, latency can become a significant bottleneck, offsetting some of the advantages of parallelism. It should be carefully assessed whether GPUs are suitable for a particular algorithm based on factors such as precision requirements, data access patterns, and compliance with numerical standards.

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[5], and frequency domain signal processing methods by Howard et. all [3].

For the remaining section of this study is organized in the following way: first, a part of the setting info surrounding 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.

2 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.

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, GPUs 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.

3 Sequential Implementation

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

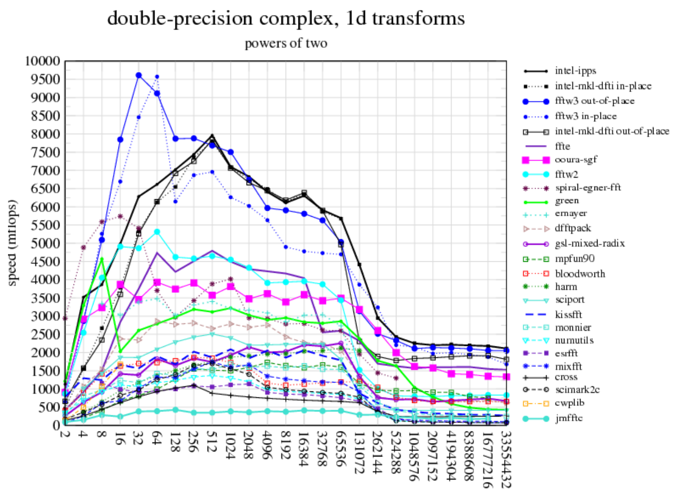


Figure 1: Benchmarks comparing the performance of FFTW, MKL and other FFT libraries. FFTW and MKL are the fastest, shown in blue and black respectively. [1]

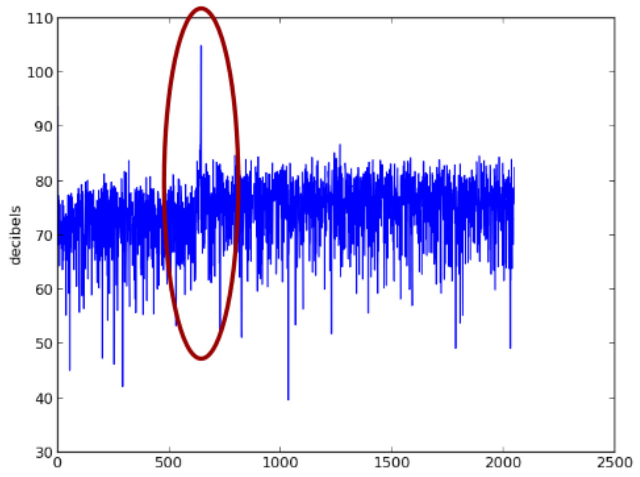


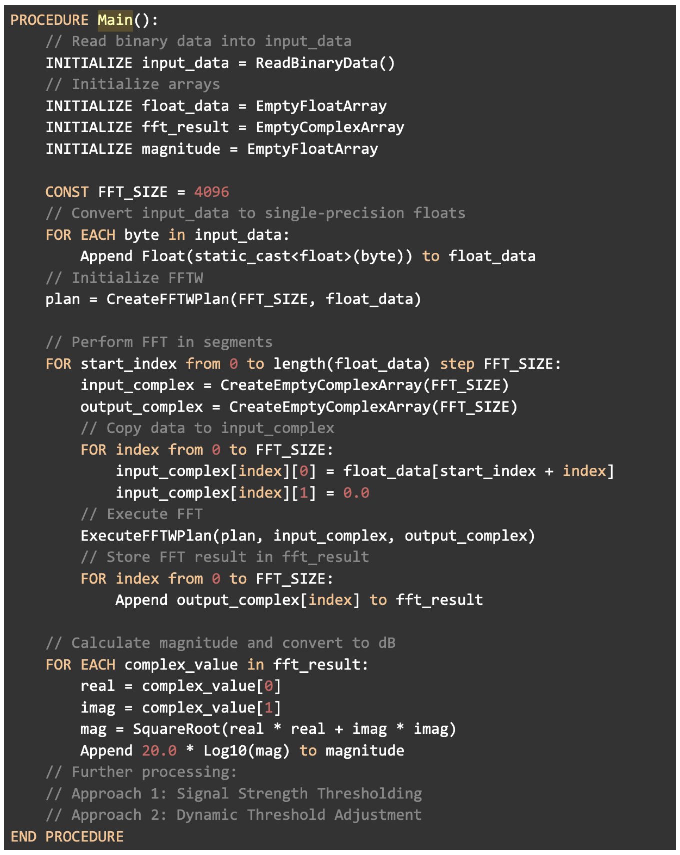
Figure 2: Following the FFT calculation and signal analysis, a transmission is clearly visible in the frequency domain.

blocks of data with a size of 4096 data points were analyzed individually and do not overlap. Typically for audio signal processing, dt might be in the range of microseconds. Modern desktop CPUs can do FFTs at speeds between tens of megaflops and several gigaflops, while consumer-grade GPUs can perform FFTs at speeds between hundreds of gigaflops and over a teraflop. Each block is treated as a separate, distinct segment of the time-domain signal. 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.

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



```
PROCEDURE Main():
    // Read binary data into input_data
    INITIALIZE input_data = ReadBinaryData()
    // Initialize arrays
    INITIALIZE float_data = EmptyFloatArray
    INITIALIZE fft_result = EmptyComplexArray
    INITIALIZE magnitude = EmptyFloatArray

    CONST FFT_SIZE = 4096
    // Convert input_data to single-precision floats
    FOR EACH byte in input_data:
        Append Float(static_cast<float>(byte)) to float_data
    // Initialize FFTW
    plan = CreateFFTWPlan(FFT_SIZE, float_data)

    // Perform FFT in segments
    FOR start_index from 0 to length(float_data) step FFT_SIZE:
        input_complex = CreateEmptyComplexArray(FFT_SIZE)
        output_complex = CreateEmptyComplexArray(FFT_SIZE)
        // Copy data to input_complex
        FOR index from 0 to FFT_SIZE:
            input_complex[index][0] = float_data[start_index + index]
            input_complex[index][1] = 0.0
        // Execute FFT
        ExecuteFFTWPlan(plan, input_complex, output_complex)
        // Store FFT result in fft_result
        FOR index from 0 to FFT_SIZE:
            Append output_complex[index] to fft_result

    // Calculate magnitude and convert to dB
    FOR EACH complex_value in fft_result:
        real = complex_value[0]
        imag = complex_value[1]
        mag = SquareRoot(real * real + imag * imag)
        Append 20.0 * Log10(mag) to magnitude
    // Further processing:
    // Approach 1: Signal Strength Thresholding
    // Approach 2: Dynamic Threshold Adjustment
END PROCEDURE
```

Figure 3: Pseudo code for Initial computation.

Pseudo code for this computation is given in Figure 3. After this calculation there are two different approaches we can take:

3.1 Signal Strength Threshold

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. Signal-to-Noise Ratio (SNR) of the received signal is one of the main factors to set a constant value. The threshold ought to be set higher than the noise level but lower than the anticipated signal level. By doing this, it is made sure that the threshold is sensitive enough to pick up real signals and ignore noise. Recognize the features of the signal's noise as well. Both random and predictable patterns can be seen in noise. The threshold should be calibrated to discriminate noise and signal with high precision. Given the unsophisticated nature of this method, it is unsurprising that it is not the most most reliable method to determine a legitimate transmission. The pseudo code is illustrated in figure 5.

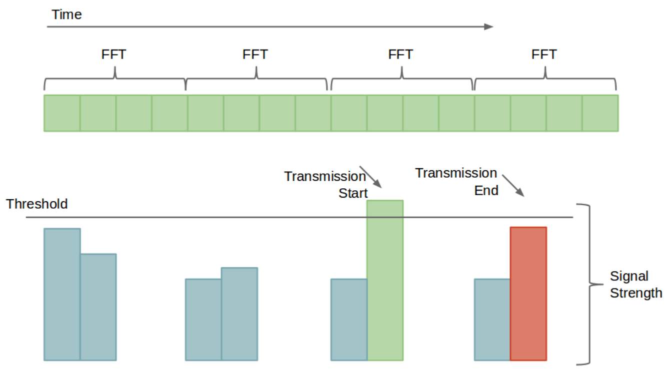
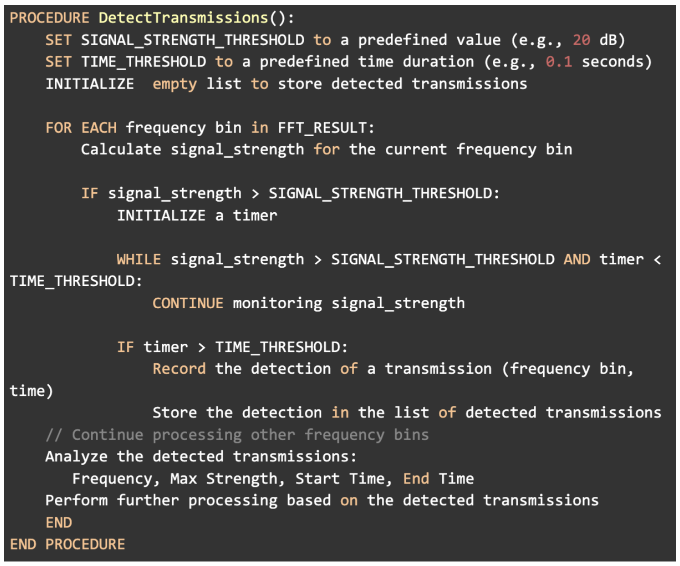


Figure 4: A graphic showing the operation of the sequential algorithm.



```
PROCEDURE DetectTransmissions():
    SET SIGNAL_STRENGTH_THRESHOLD to a predefined value (e.g., 20 dB)
    SET TIME_THRESHOLD to a predefined time duration (e.g., 0.1 seconds)
    INITIALIZE  empty list to store detected transmissions

    FOR EACH frequency bin in FFT_RESULT:
        Calculate signal_strength for the current frequency bin

        IF signal_strength > SIGNAL_STRENGTH_THRESHOLD:
            INITIALIZE a timer

            WHILE signal_strength > SIGNAL_STRENGTH_THRESHOLD AND timer <
TIME_THRESHOLD:
                CONTINUE monitoring signal_strength

            IF timer > TIME_THRESHOLD:
                Record the detection of a transmission (frequency bin,
time)
                Store the detection in the list of detected transmissions
    // Continue processing other frequency bins
    Analyze the detected transmissions:
       Frequency, Max Strength, Start Time, End Time
    Perform further processing based on the detected transmissions
    END
END PROCEDURE
```

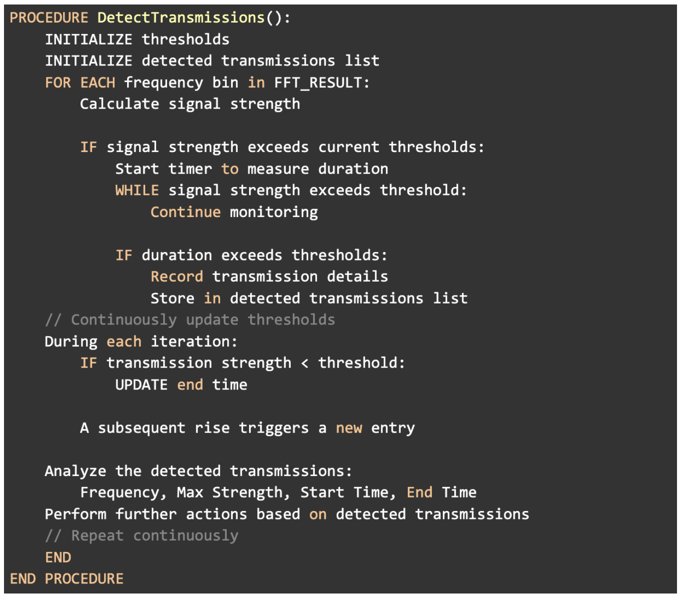
Figure 5: Pseudo code for Signal Strength Threshold.

3.2 Dynamic Threshold Adjustment

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, 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. The pseudo code is illustrated in figure 6.



```
PROCEDURE DetectTransmissions():
    INITIALIZE thresholds
    INITIALIZE detected transmissions list
    FOR EACH frequency bin in FFT_RESULT:
        Calculate signal strength

        IF signal strength exceeds current thresholds:
            Start timer to measure duration
            WHILE signal strength exceeds threshold:
                Continue monitoring

            IF duration exceeds thresholds:
                Record transmission details
                Store in detected transmissions list
    // Continuously update thresholds
    During each iteration:
        IF transmission strength < threshold:
            UPDATE end time

        A subsequent rise triggers a new entry

    Analyze the detected transmissions:
        Frequency, Max Strength, Start Time, End Time
    Perform further actions based on detected transmissions
    // Repeat continuously
    END
END PROCEDURE
```

Figure 6: Pseudo code for Dynamic Threshold Adjustment.

4 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. Each CUDA core works on a portion of the FFT calculation and processes different parts of the input data in parallel. Factors such as the size of the FFT, the GPU type, and the amount of CUDA cores affect how parallelism is used in an FFT computation on a GPU. For the purpose of this study the NVIDIA GTX460 with 336 CUDA cores was used. 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. PCIE bandwidth spans between 1 GB/s to 64 GB/s depending on PCIE generation and lane layout, while internal CPU and GPU data transfer rates can exceed 100 GB/s. 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.

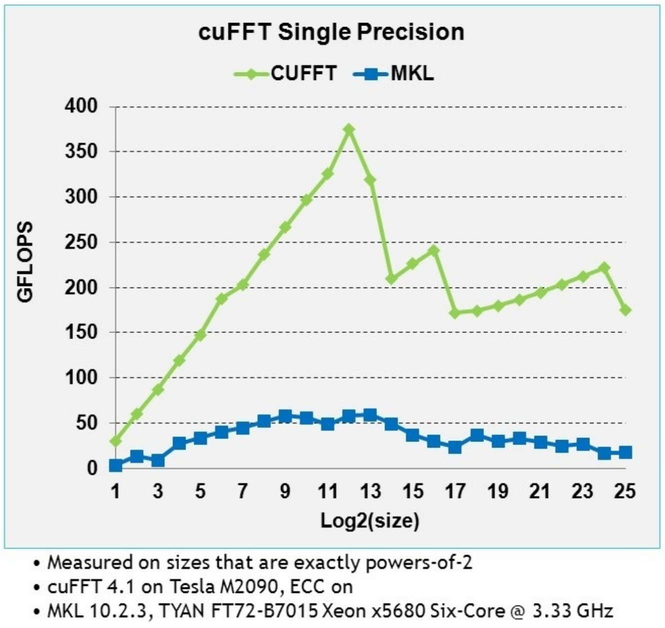
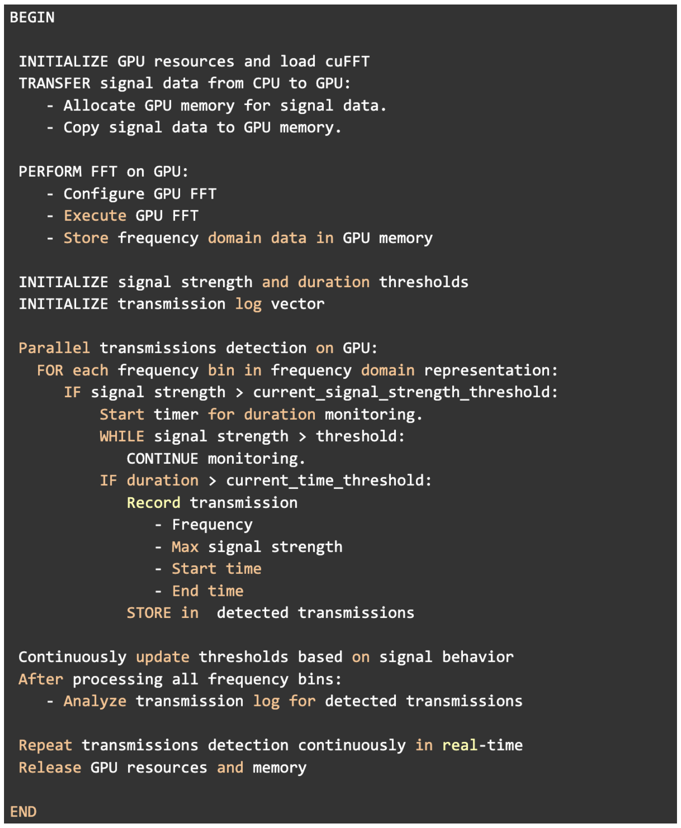


Figure 7: A single-precision performance comparison between cuFFT and the Intel MKL Library. [2]

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.

4.1 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. In this GPU-accelerated implementation, data is efficiently transferred to the GPU, where cuFFT is employed for FFT calculations, followed by parallel execution of custom CUDA kernels. Comprehensive testing and documentation ensure both legality and robust handling of potential race conditions. 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. The pseudo code is illustrated in figure 8.



```
BEGIN

 INITIALIZE GPU resources and load cuFFT
 TRANSFER signal data from CPU to GPU:
    - Allocate GPU memory for signal data.
    - Copy signal data to GPU memory.

 PERFORM FFT on GPU:
    - Configure GPU FFT
    - Execute GPU FFT
    - Store frequency domain data in GPU memory

 INITIALIZE signal strength and duration thresholds
 INITIALIZE transmission log vector

 Parallel transmissions detection on GPU:
   FOR each frequency bin in frequency domain representation:
      IF signal strength > current_signal_strength_threshold:
          Start timer for duration monitoring.
          WHILE signal strength > threshold:
             CONTINUE monitoring.
          IF duration > current_time_threshold:
             Record transmission
                - Frequency
                - Max signal strength
                - Start time
                - End time
             STORE in  detected transmissions

 Continuously update thresholds based on signal behavior
 After processing all frequency bins:
    - Analyze transmission log for detected transmissions

 Repeat transmissions detection continuously in real-time
 Release GPU resources and memory

END
```

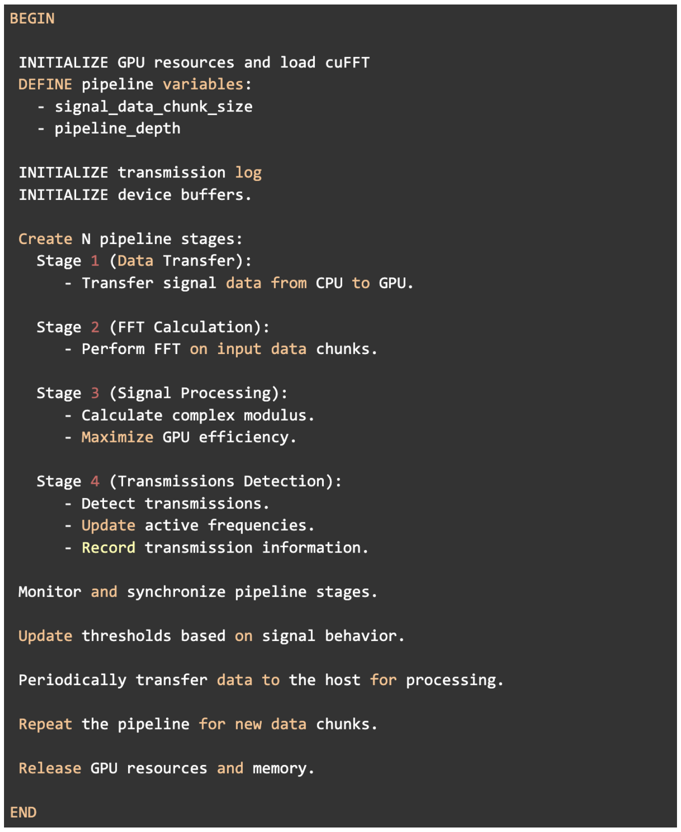
Figure 8: Pseudo code for Non pipelined approach.

4.2 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 pseudo code is illustrated in figure 9 and the pipelined process is illustrated in figure 10.



```
BEGIN

 INITIALIZE GPU resources and load cuFFT
 DEFINE pipeline variables:
   - signal_data_chunk_size
   - pipeline_depth

 INITIALIZE transmission log
 INITIALIZE device buffers.

 Create N pipeline stages:
   Stage 1 (Data Transfer):
      - Transfer signal data from CPU to GPU.

   Stage 2 (FFT Calculation):
      - Perform FFT on input data chunks.

   Stage 3 (Signal Processing):
      - Calculate complex modulus.
      - Maximize GPU efficiency.

   Stage 4 (Transmissions Detection):
      - Detect transmissions.
      - Update active frequencies.
      - Record transmission information.

 Monitor and synchronize pipeline stages.

 Update thresholds based on signal behavior.

 Periodically transfer data to the host for processing.

 Repeat the pipeline for new data chunks.

 Release GPU resources and memory.

END
```

Figure 9: Pseudo code for Pipelined approach.

5 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

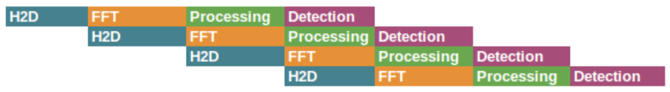


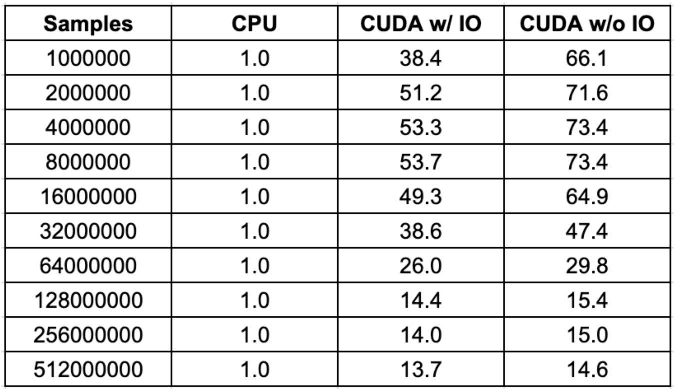
Figure 10: A graphic briefly showing the pipelined approach to the problem. Each stage is shown in a different color and can be performed independently of one another.

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. In this context, the term ”atomics” refers to specialized procedures utilized for thread synchronization of parallel processing. They are crucial to preserving the program's data integrity. The use of atomics could be optimized further to reduce a performance bottleneck. For example, In a parallel transmission counting scenario, atomics are used to maintain accurate counts and buffer updates but can introduce serialization, limiting parallelism. Use of thread-local counters for independent increments may be a solution to this problem. The global transmission counter can be updated by threads separately maintaining their local counts and occasionally accumulating these numbers using atomics. Consider employing buffer pools or thread-specific caches for buffer updates. These sources allow threads to access their own buffers to store transmission data, only using locking methods like mutexes for the final update. These techniques try to ensure data accuracy in situations when atomics are necessary while minimizing the impact of atomics on parallelism. However, thorough testing and profiling are necessary to validate their effectiveness for the specific use case. The speedup over the CPU version are illustrated in Figures 11 and 12.

When scaling the cuFFT algorithm across many GPU cores or processors, an approximate optimal speedup value would likely fall between 0.7 to 0.9, or even higher. In other words, doubling the number of processing units should ideally result in a speedup for the parallelized cuFFT implementation that is at least 70% to 90% of the ideal linear speedup. 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.

6 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



Samples	CPU	CUDA w/ IO	CUDA w/o IO
1000000	1.0	38.4	66.1
2000000	1.0	51.2	71.6
4000000	1.0	53.3	73.4
8000000	1.0	53.7	73.4
16000000	1.0	49.3	64.9
32000000	1.0	38.6	47.4
64000000	1.0	26.0	29.8
128000000	1.0	14.4	15.4
256000000	1.0	14.0	15.0
512000000	1.0	13.7	14.6

Figure 11: Table displaying GPU speedup over CPU implementation with and without I/O.

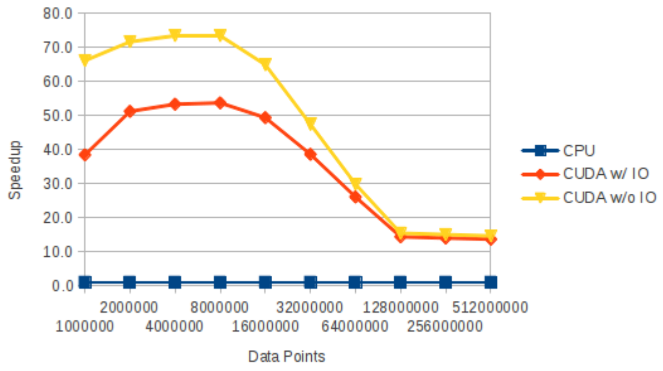
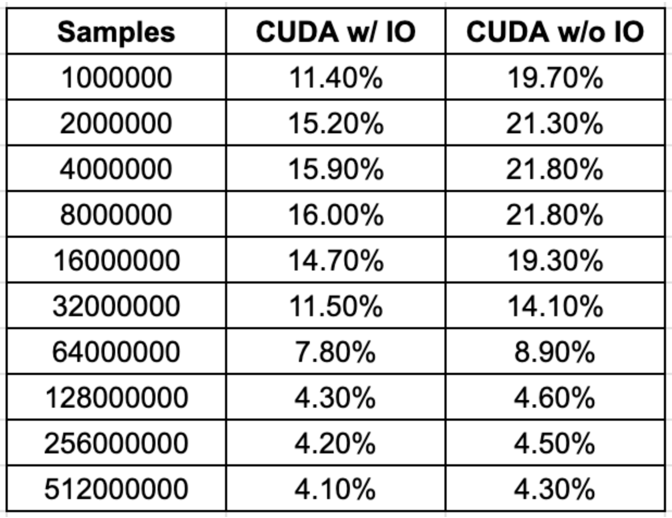


Figure 12: Graph displaying GPU speedup over CPU implementation with and without I/O.

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.

7 Future Work

Unfortunately, the pipelined implementation was unable to finish. 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. 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 outlives the usefulness of creating a pipeline to begin with, since different steps cannot be executed at the same time. The cuFFT API call to execute a FFT is blocking for an unknown reason. The API call should not



Samples	CUDA w/ IO	CUDA w/o IO
1000000	11.40%	19.70%
2000000	15.20%	21.30%
4000000	15.90%	21.80%
8000000	16.00%	21.80%
16000000	14.70%	19.30%
32000000	11.50%	14.10%
64000000	7.80%	8.90%
128000000	4.30%	4.60%
256000000	4.20%	4.50%
512000000	4.10%	4.30%

Figure 13: Table with GPU efficiency with and without I/O. (Based on 336 Cuda cores in the Nvidia GTX460)

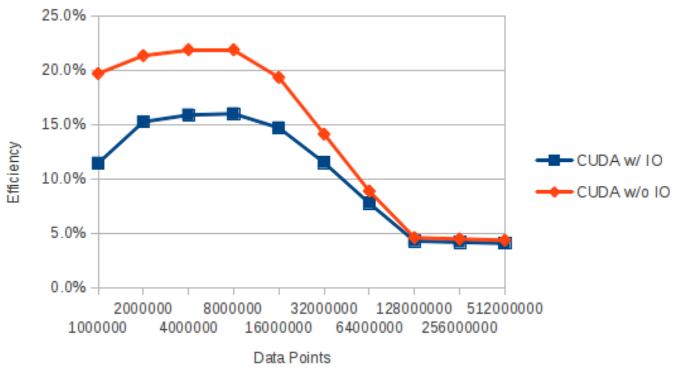
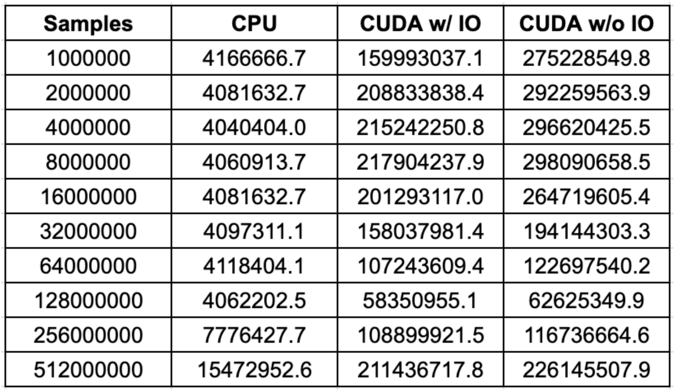


Figure 14: Graph showing GPU efficiency with and without I/O. (Based on 336 Cuda cores in the Nvidia GTX460)

inherently be blocking. 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. In our future work, we plan to address runtime issues by implementing thread-safe cuFFT utilization techniques. Investigating the root causes through profiling tools that can pinpoint where the blocking is occurring , optimizing GPU resource allocation, and exploring advanced techniques such as dynamic parallelism can also help to achieve non-blocking cuFFT execution. 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.



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

Figure 15: Table of throughput (in samples/second) of versions of the program, both sequential and parallel.

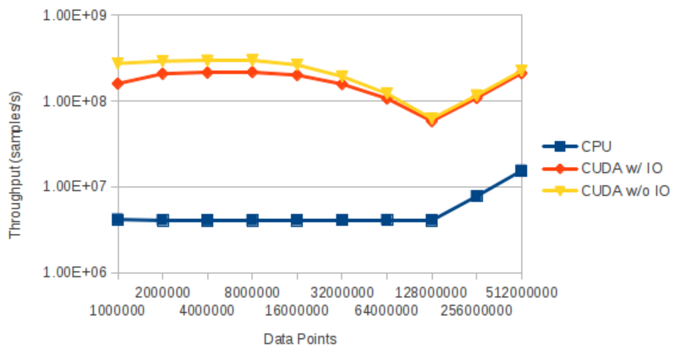


Figure 16: Graph showing throughput (in samples/second) of versions of the program, both sequential and parallel.

References

[1] 3.0 GHz intel core duo, intel compilers, 64-bit mode. http://www.fftw.org/speed/CoreDuo-3.0GHz-icc64/.

[2] CUFFT | NVIDIA developer zone. https://developer.nvidia.com/cufft.

[3] S.J. Howard and K. Pahlavan. Measurement and analysis of the indoor radio channel in the frequency domain. *Instrumentation and Measurement, IEEE Transactions on*, 39(5):751–755, 1990.

[4] Erik Sintorn and Ulf Assarsson. Fast parallel gpu-sorting using a hybrid algorithm. *Journal of Parallel and Distributed Computing*, 68(10):1381–1388, 2008.

[5] 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*, pages 1–5, 2010.