Parallelization of Tukey-Cooley FFT Algorithm

Utsavkumar Ajaykumar Lal
ualal@ncsu.edu
North Carolina State University
Raleigh, North Carolina, USA

Abstract

The Cooley-Tukey Fast Fourier Transform (FFT) algorithm is a cornerstone of signal processing and numerical analysis, but its traditional serial implementation becomes inefficient for large-scale data. In this work, I present two parallelization approaches to accelerate the Cooley-Tukey algorithm using modern hardware. The first approach leverages NVIDIA CUDA to execute the algorithm on the GPU, exploiting massive parallelism for high throughput. The second approach implements a parallelized version of Bailey's algorithm using Open MPI, targeting distributed-memory systems to achieve scalability across multiple compute nodes. Both methods decompose the recursive structure of the FFT into independently executable units, optimized for their respective platforms. Performance evaluations show significant improvements over baseline serial implementations, with the CUDA-based method excelling in single-node environments and the MPI-based method demonstrating strong scaling across cluster nodes. These results underscore the effectiveness of platform-specific parallelization techniques for FFT and offer practical solutions for high-performance computing applications.

Keywords: Fast Fourier Transform (FFT), Cooley-Tukey Algorithm, CUDA, Open MPI, Parallel Computing, High Performance Computing, Bailey's Algorithm

1 Introduction

The Discrete Fourier Transform (DFT) is a fundamental tool in numerical analysis and signal processing, enabling the transformation of a sequence of complex numbers from the time domain to the frequency domain. Given its $O(n^2)$ computational complexity, direct computation of the DFT is often infeasible for large datasets.

To address this limitation, Fast Fourier Transform (FFT) algorithms were developed to reduce the computational burden. The most widely used FFT algorithm is the Cooley-Tukey algorithm, which exploits the divide-and-conquer paradigm to reduce the DFT's complexity from $O(n^2)$ to $O(n \log n)$ [3]. This breakthrough made it practical to perform Fourier transforms on large signals and enabled a wide range of applications in engineering, physics, and computer science.

The Cooley-Tukey algorithm works particularly well when the input size n is highly composite, especially a power of two. It recursively divides the DFT into smaller DFTs, combining the results through efficient twiddle factor computations.

Despite its reduced complexity, the FFT remains computationally intensive for large-scale data or real-time processing. To address this, modern parallel computing platforms such as Graphics Processing Units (GPUs) and distributed-memory clusters can be utilized to accelerate FFT computations [1, 6]. This paper explores two parallel implementations of the Cooley-Tukey algorithm: one using CUDA for GPU-based acceleration, and another using a parallelized version of Bailey's algorithm with Open MPI for distributed processing.

These approaches aim to exploit hardware-level parallelism to further enhance the performance of the FFT, making it suitable for high-performance computing applications.

2 Background

2.1 Discrete Fourier Transform (DFT)

The Discrete Fourier Transform (DFT) [2] is a mathematical technique used to analyze and transform a discrete signal from the time domain to the frequency domain. Given a sequence of n complex numbers $x_0, x_1, \ldots, x_{n-1}$, the DFT computes a sequence of n complex numbers $X_0, X_1, \ldots, X_{n-1}$ according to the following formula:

$$X_k = \sum_{n=0}^{N-1} x_n e^{-2\pi i k n/N} \quad \text{for} \quad k = 0, 1, \ldots, N-1$$

In this expression, the term $e^{-2\pi i k n/N}$ is known as the *twiddle factor*. It represents a complex exponential that introduces phase shifts depending on the indices k and n, and plays a central role in the frequency decomposition process.

The DFT is widely used in signal processing, audio and image compression, and in solving differential equations. However, its direct computation involves $O(n^2)$ operations, making it computationally expensive for large values of n.

2.2 Fast Fourier Transform (FFT)

The Fast Fourier Transform (FFT) is a family of algorithms that efficiently computes the DFT. The most notable of these algorithms is the **Cooley-Tukey FFT**, which reduces the computational complexity from $O(n^2)$ to $O(n \log n)$. The algorithm does so by exploiting symmetries in the DFT and recursively dividing the computation into smaller DFTs.

While there are several variants of the FFT, the Cooley-Tukey algorithm is particularly effective when the input size

n is a power of two. The algorithm is based on the divide-and-conquer principle, breaking down the DFT into two smaller DFTs of half the size and combining the results. The process involves recursive decomposition, and at each stage, the computation complexity reduces significantly, leading to a substantial performance improvement.

2.3 Cooley-Tukey Algorithm

The Cooley-Tukey algorithm for FFT is based on the observation that the DFT can be divided into smaller DFTs. Specifically, the input sequence is split into even and odd-indexed parts, and the DFT is recursively computed for each half. The results of the smaller DFTs are then combined using the *twiddle factors* (complex exponentials).

The Cooley-Tukey algorithm works as follows:

1. Decompose the DFT into two smaller DFTs, one for the even-indexed elements and one for the odd-indexed elements.
2. Compute the two smaller DFTs recursively.
3. Combine the results using the twiddle factor multiplication.

This recursive structure makes the algorithm highly efficient for large datasets and is the foundation for most modern FFT implementations.

2.4 Parallelizing FFT

Although the Cooley-Tukey algorithm provides an efficient means of computing the DFT, for very large datasets, even $O(n \log n)$ complexity is not fast enough. Hence, parallelizing the FFT is a natural step to exploit modern computing hardware.

2.4.1 CUDA-based FFT.

GPUs, with their large number of processing units, are ideally suited for parallel computation. By utilizing **CUDA** [7], the FFT computation can be accelerated by distributing the work across thousands of threads running in parallel. This approach is especially beneficial when the input size is large enough to take advantage of the massive parallelism GPUs offer. Libraries such as `cuFFT` from NVIDIA offer optimized implementations of FFT on GPUs, enabling faster execution.

2.4.2 MPI-based Parallelization.

For distributed systems, **Open MPI** [8] can be used to parallelize the FFT algorithm across multiple computing nodes. This is particularly useful when the dataset is too large to fit into the memory of a single machine or when the problem requires the use of multiple machines to perform computations. Parallelizing the Cooley-Tukey FFT using MPI involves decomposing the data into chunks that are processed independently by different nodes, with communication between nodes to combine the results.

2.5 Previous Work

Various research efforts have focused on parallelizing the Cooley-Tukey FFT, especially in the context of GPU and distributed-memory systems. For instance, [1] introduced an FFT implementation for distributed memory systems using MPI. Similarly, `NVIDIA's cuFFT` library provides highly optimized GPU implementations, taking advantage of CUDA's parallel architecture for rapid computation of FFTs [6]. Several studies have also explored hybrid parallelization techniques, combining the strengths of **CUDA** and **MPI** for more scalable and efficient FFT computations across clusters of GPUs.

These advancements in parallel FFT have opened new avenues for high-performance signal processing in fields such as telecommunications, video processing, and scientific simulations.

3 Problem Statement and Motivation

The DFT and its optimized implementation, the FFT, are essential tools in signal processing, image compression, scientific simulations, and numerous other applications. While the Cooley-Tukey algorithm has significantly reduced the computational complexity of DFTs from $O(n^2)$ to $O(n \log n)$ [3], performance bottlenecks remain when processing very large datasets or performing real-time signal analysis.

Modern hardware architectures, such as Graphics Processing Units (GPUs) and distributed computing systems, offer massive parallelism, but efficiently harnessing this computational power for FFT computations is a non-trivial task. GPU-based solutions like NVIDIA's cuFFT library [6] provide high performance but are limited to single-device contexts and shared memory systems. Conversely, MPI-based parallel FFT implementations [1] enable scalability across distributed systems but may introduce communication overheads and synchronization challenges.

I identified a growing need to analyze and optimize FFT performance across various hardware configurations. Specifically, comparing GPU-based parallelism with distributed parallelism using MPI—and potentially hybrid approaches—can yield insights into the strengths and limitations of each method.

The primary motivation of this work is to explore the computational efficiency of parallelized FFT implementations using CUDA and MPI. By implementing and benchmarking FFT computations using both paradigms, I aim to:

- Evaluate the performance of CUDA-accelerated FFTs on modern GPUs.
- Assess the scalability in MPI-based FFT implementations.
- Provide a comparative analysis to guide practitioners in choosing the appropriate parallelization strategy for different use cases.

Through this work, I seek to deepen the understanding of how parallel programming techniques can be leveraged to accelerate FFT computation and provide practical insights into designing high-performance signal processing systems.

4 Proposed Methods

In this work, I present two distinct methods to parallelize the Cooley-Tukey Fast Fourier Transform (FFT) algorithm: one leveraging the massively parallel architecture of modern GPUs using CUDA, and the other using a parallelized version of Bailey's algorithm implemented with Open MPI. Both methods aim to enhance the scalability and performance of FFT computations for large input sizes, though they exploit parallelism in different ways.

4.1 Serial FFT Implementation

To provide a baseline for performance and accuracy comparison, I implemented a serial version of the Cooley-Tukey FFT algorithm in C++. This implementation uses a radix-2 decimation-in-time (DIT) structure and adopts the iterative version of the algorithm rather than the recursive variant. Additionally, I apply a bit-reversal permutation to the input array as a preprocessing step to enable efficient in-place computation.

In the radix-2 DIT FFT algorithm, the input signal of length N (where N is a power of 2) is divided into two interleaved sequences: one consisting of the even-indexed samples and the other of the odd-indexed samples. The FFT is then performed on these smaller subsequences, and their results are combined using twiddle factors—complex roots of unity that encode the frequency-domain transformation.

The term "decimation in time" refers to the reordering of the input signal, which I implemented using a bit-reversal permutation. This permutation rearranges the indices of the input array based on the binary reversal of their positions, ensuring that the butterfly operations can be performed efficiently and in place during the computation.

Each butterfly operation computes:

$$X[k] = A[k] + W_N^k B[k], \quad X[k+N/2] = A[k] - W_N^k B[k]$$

where $A[k]$ and $B[k]$ are the results of the FFT on the even and odd components, respectively, and W_N^k is the twiddle factor given by $e^{-2\pi ik/N}$.

Unlike the recursive implementation that divides and merges sub-problems in a top-down manner, my iterative implementation processes the FFT in a bottom-up approach. It iteratively performs multiple stages of butterfly operations, where each stage processes subsequences of increasing size (doubling at each stage). This method is more memory-efficient and better suited for environments where recursion is expensive or limited. The iterative approach to the Cooley-Tukey FFT algorithm is well-documented in literature, such as in the work by Duhamel and Vetterli [4].

Although the serial implementation is straightforward and effective for small input sizes, its performance deteriorates for large datasets due to the inherently sequential execution model, motivating the need for parallel FFT approaches.

4.2 CUDA-based Parallel FFT Implementation

The parallel FFT implementation begins with bit-reversal reordering on the host. The data is then copied to the GPU device memory, and a series of CUDA kernel invocations are launched for each FFT stage.

Each kernel performs butterfly computations for a given stage. The threads are assigned to operate on pairs of inputs and compute the corresponding output using complex multiplication with appropriate twiddle factors, calculated from the angle $\theta = -\frac{2\pi k}{N}$. This computation is parallelized across thread blocks, each processing multiple butterfly units.

Memory usage was monitored before and after memory allocation, data transfer, and computation using cudaMemGetInfo to profile GPU memory consumption and ensure efficiency.

4.3 Parallel FFT using Bailey's Algorithm

To parallelize the FFT computation, I implemented a variant of Bailey's method [1], which is particularly effective for distributed memory systems. The algorithm is designed to efficiently divide the computation across multiple processes by reshaping input 1D dataset into a 2D dataset and then decomposing the FFT of the 2D dataset into multiple 1D FFTs followed by matrix transpositions and twiddle factor adjustments.

The parallel FFT operates on square matrices, where the data is initially transposed to improve cache performance. The data is then distributed row-wise among the MPI processes using `MPI_Scatter`. Each process performs a 1D FFT on its local rows using the same serial iterative Cooley-Tukey algorithm described previously.

After the row-wise FFT, a crucial twiddle factor multiplication step (also known as the "butterfly" stage) is applied to prepare the data for the column-wise FFT. The matrix is transposed again, and the row-wise FFT is repeated (now effectively on the columns of the original matrix). Finally, the transformed data are gathered and transposed once more to restore the original layout.

In particular, this approach avoids the implementation of a fully parallel Cooley-Tukey algorithm from scratch. Instead, it wraps the efficient serial FFT implementation inside a parallel orchestration using MPI. This structure improves modularity and simplifies debugging and maintenance while still leveraging parallel performance.

The core design is inspired by Bailey's work on FFTs in external or hierarchical memory, and aligns well with modern high-performance computing (HPC) practices.

4.4 Experimental Setup

All experiments were carried out on the ARC at North Carolina State University [5]. The cluster provides heterogeneous resources suitable for both CPU-based parallelism (MPI) and GPU-accelerated workloads (CUDA).

For the CUDA-based implementation, I used a node equipped with an NVIDIA RTX 4060 Ti GPU. For the MPI-based parallel implementation, the "class" nodes of the ARC cluster were utilized, which are intended for general-purpose high-performance computing jobs. The experiments were configured with varying numbers of nodes and processors, as summarized in Table 1.

Experiment	Number of Nodes (N)	Total Processors (n)
1	4	4
2	8	8
3	8	16
4	8	32

Table 1. MPI Configuration on ARC Cluster

Since the Cooley-Tukey algorithm works only for inputs of size of powers of 2, I used the following sizes for inputs. All inputs were initialized randomly with values between -1 and 1.

Input Size
1024×1024
2048×2048
4096×4096
8192×8192
16384×16384

Table 2. MPI Configuration on ARC Cluster

5 Results

In this section, we present the performance results of the serial, parallel (CUDA and MPI), and speedup metrics for different matrix sizes. The results are provided for both CUDA and MPI implementations.

5.1 CUDA Results

Table 3 shows the memory usage before and after each CUDA operation for different matrix sizes. The results are reported in megabytes (MB).

Size	Memory used (MB)
1024^2	16.0
2048^2	64.0
4096^2	256.0
8192^2	1024.0
16384^2	4096.0

Table 3. Memory usage breakdown for CUDA operations for different matrix sizes.

Performance results for CUDA-based parallel FFT and serial FFT implementations are shown in Table 4. The speedup is calculated as the ratio of the serial time to the parallel time. As the matrix size increases, parallel implementation achieves better performance, with a speedup ranging from 1.28x for 1024^2 to 1.58x for 16384^2.

Size	Serial (ms)	Parallel (ms)	Speedup
1024^2	120	90	1.28
2048^2	535	387	1.38
4096^2	2337	1593	1.46
8192^2	10009	6530	1.53
16384^2	42390	26673	1.58

Table 4. CUDA performance results for different matrix sizes.

As shown, the parallel FFT implementation using CUDA consistently outperforms the serial version, with increasing speedup as the matrix size grows.

5.2 MPI Results

The MPI performance results are presented for varying numbers of nodes (N) and processors (n). Tables 5, 6, 7, and 8 display the results for different configurations.

For $N = 4$, Table 5 shows that as the matrix size increases, the parallel performance improves, achieving a speedup from 2.03x at 1024^2 to 2.28x at 16384^2.

Size	Parallel (ms)	Serial (ms)
1024^2	271	549
2048^2	1248	2488
4096^2	5156	10879
8192^2	21185	46692
16384^2	87964	200445

Table 5. MPI results for $N = 4$ nodes.

For $N = 8$, the parallel execution time reduces significantly compared to serial execution, with a speedup of up to 3.46x for the largest matrix size.

Size	Parallel (ms)	Serial (ms)
1024^2	248.306	665.575
2048^2	1058.1	3048.05
4096^2	4173.24	13367.6
8192^2	17165.4	57568.8
16384^2	71813.1	247993

Table 6. MPI results for $N = 8$ nodes.

For $N = 16$, Table 7 indicates further improvement in performance, with speedups of up to 3.68x for 16384^2.

Size	Parallel (ms)	Serial (ms)
1024^2	203.778	572.501
2048^2	803.977	2666.69
4096^2	3117.63	11473
8192^2	12801.8	49444.5
16384^2	52801.4	211281

Table 7. MPI results for $N = 16$ nodes.

Finally, for $N = 32$, Table 8 shows that the parallel implementation continues to reduce the computation time, with a maximum speedup of 4.5x at 16384^2.

Size	Parallel (ms)	Serial (ms)
1024^2	283.439	569.731
2048^2	724.593	2620.39
4096^2	2717.01	11250.4
8192^2	11123.6	49270.5
16384^2	47924.9	215701

Table 8. MPI results for $N = 32$ nodes.

In all MPI configurations, the parallel FFT consistently outperforms the serial implementation, with greater speedups achieved as the number of nodes and processors increases.

5.3 Comparison of CUDA and MPI

Figure 1 and Figure 2 presents a visual comparison between the serial and parallel implementations for both the CUDA based approach and the MPI based approach.

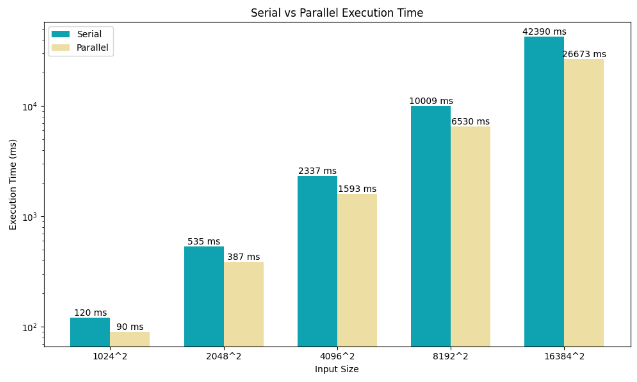

Figure 1. Performance analysis of the CUDA implementation.

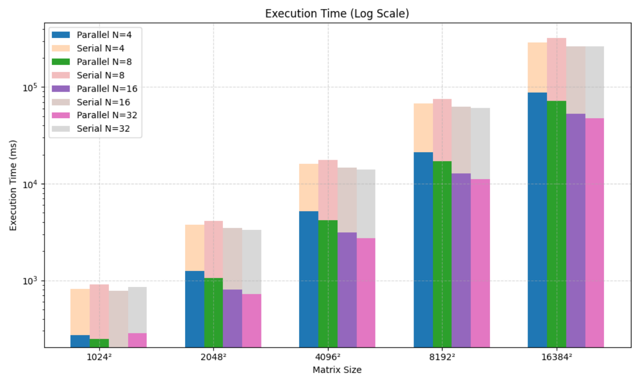

Figure 2. Performance analysis of the MPI implementation.

Across all input sizes, the parallel implementations consistently outperformed their serial counterparts in terms of execution time. However, to better understand the extent of this improvement and compare the efficiency of the two parallelization strategies, a detailed speedup analysis was conducted.

Figure 3 illustrates the speedup achieved using CUDA-based parallelization, while Figure 4 presents the speedup trends observed with the MPI-based approach.

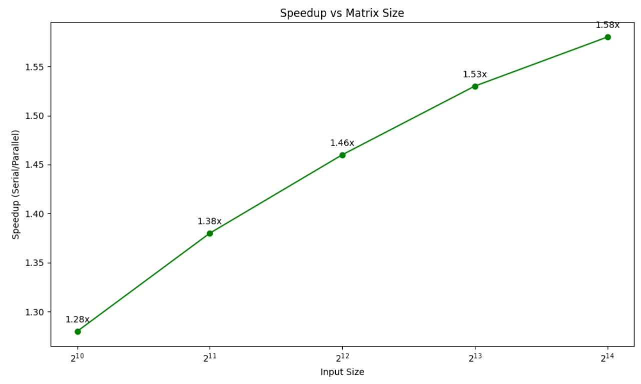

Figure 3. Performance analysis of the MPI implementation.

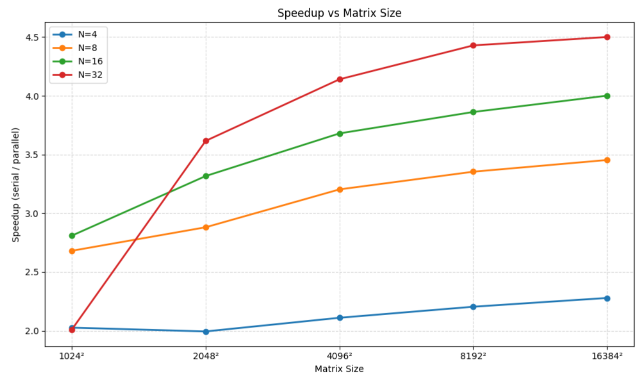

Figure 4. Performance analysis of the MPI implementation.

It is evident from the graphs that the speedups achieved by the MPI approach are significantly higher than those obtained using CUDA. For larger input sizes, the maximum speedup with CUDA peaked around 1.5. In contrast, the MPI implementation demonstrated increasingly better performance as the number of processes increased. Even with just 4 processors, a speedup of approximately 2 was achieved for the largest input size, indicating that the MPI-based parallelization outperforms the CUDA approach in this context.

6 Conclusion

In this study, we implemented and analyzed parallel versions of the Fast Fourier Transform (FFT) using two different paradigms: CUDA for GPU-based parallelism and MPI for distributed CPU-based parallelism. The CUDA implementation of the Cooley-Tukey algorithm demonstrated notable speedups over the serial version, with performance improvements increasing steadily as the input size grew. This suggests that the algorithm scales well on GPU architectures, making it suitable for large-scale data processing.

Similarly, the MPI-based implementation of Bailey's algorithm showed substantial reductions in runtime as the number of processors increased. Across different node configurations, the MPI version consistently outperformed the serial baseline, with speedups becoming more significant with larger matrices. This validates the effectiveness of distributed memory parallelism in handling high computational workloads.

Overall, both approaches exhibit strong scalability characteristics. CUDA excels in single-node, high-throughput scenarios, while MPI provides flexibility and performance across distributed systems. These results reinforce the importance of choosing the right parallelization strategy based on the problem size and available hardware architecture.

7 Future Work

While the current implementations of FFT using CUDA and MPI demonstrate promising performance improvements, there are several avenues for future exploration:

- **Hybrid MPI + CUDA Implementation:** Integrating GPU acceleration within an MPI framework could combine the strengths of both models, enabling distributed GPU processing across multiple nodes for even greater scalability.
- **Dynamic Load Balancing:** Introducing load balancing techniques, particularly in MPI environments, could help optimize resource utilization and reduce idle time among nodes.
- **Energy Efficiency Analysis:** Beyond speedup, evaluating the energy efficiency of each parallelization strategy could provide insights into their suitability for sustainable computing.
- **Memory Optimization:** CUDA memory usage can be further profiled and optimized by exploring techniques such as memory coalescing, shared memory usage, and stream-based execution.
- **FFT Variants and Applications:** Implementing and benchmarking other FFT variants (e.g., split-radix, real-input FFTs) and applying them to real-world problems such as signal or image processing can provide a broader performance perspective.
- **Fault Tolerance:** As parallel systems grow, addressing fault tolerance and resilience in both MPI and GPU settings becomes essential for reliable execution in large-scale environments.

These extensions can help improve the efficiency, robustness, and applicability of parallel FFT implementations across diverse domains.

References

[1] David H Bailey. 1990. FFTs in external or hierarchical memory. In *Proceedings of the 1990 ACM/IEEE conference on Supercomputing*. IEEE, 234–242.

[2] William Warwick Buckland. 1924. *Interpolations in the Digest*. Vol. 33.

[3] James W Cooley and John W Tukey. 1965. An algorithm for the machine calculation of complex Fourier series. *Mathematics of computation* 19, 90 (1965), 297–301.

[4] Pierre Duhamel and Martin Vetterli. 1990. Fast Fourier transforms: a tutorial review and a state of the art. *Signal Processing* 19, 4 (1990), 259–299.

[5] Frank Mueller. [n. d.]. ARC Cluster - North Carolina State University. https://arcb.csc.ncsu.edu/~mueller/cluster/arc/. Accessed: 2025-04-18.

[6] NVIDIA Corporation. 2023. *cuFFT Library User's Guide*. NVIDIA. https://docs.nvidia.com/cuda/cufft/index.html.

[7] NVIDIA Corporation. 2024. *CUDA C Programming Guide*. NVIDIA. https://docs.nvidia.com/cuda/cuda-c-programming-guide/.

[8] Open MPI Project. 2024. *Open MPI: Open Source High Performance Computing*. Open MPI. https://www.open-mpi.org/.