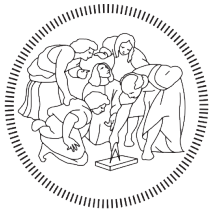


POLITECNICO
MILANO 1863

Parallel Fast Fourier Transform

Francesco Virgulti
Edoardo Pierluigi Leali
Mohamed Zeyad Mohamed Mandour

Supervised by:
Prof. Luca Fromaggia

June 2025

Contents

1 Introduction

The Fast Fourier Transform (FFT) is a cornerstone of modern signal processing and numerical computation. It efficiently computes the Discrete Fourier Transform (DFT), reducing the complexity from $O(N^2)$ to $O(N \log N)$. Although the algorithm was formally introduced by Cooley and Tukey in 1965, its mathematical foundations date back to the work of Carl Friedrich Gauss in the early 19th century.

FFT algorithms are widely used in scientific, engineering, and real-time applications due to their computational efficiency and versatility. Common use cases include:

- Audio and speech signal processing
- Image compression and reconstruction
- Data analysis and pattern recognition
- Wireless and digital communications

In this work, we present a modern implementation of the FFT and its variants, including iterative, recursive, GPU-parallel (CUDA), and distributed (MPI) versions. Our approach builds on the classical Cooley–Tukey framework while leveraging contemporary C++20 features such as concepts and `if constexpr` to improve type safety, generality, and performance.

We also provide a mathematical foundation for both one-dimensional and two-dimensional FFTs, followed by detailed implementation strategies and benchmarking analyses. Our goal is to demonstrate not only the theoretical underpinnings of FFT but also its practical relevance across computing architectures.

2 FFT Mathematical Foundation

The Discrete Fourier Transform (DFT) maps a sequence of complex numbers from the time domain to the frequency domain. Given a sequence $x_0, x_1, \ldots, x_{N-1}$, the DFT is defined as:

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

Here, W_N is the primitive N-th root of unity, often called the *twiddle factor*. Each X_k represents the amplitude and phase of the k-th frequency component in the signal. A direct computation of this formula requires $O(N^2)$ operations, due to the nested summation over all N input values for each of the N output frequencies.

Matrix Representation The DFT can be written in matrix form as:

$$\mathbf{X} = \mathbf{W}_N \cdot \mathbf{x},$$

where $\mathbf{x} \in C^N$ is the input vector, and $\mathbf{W}_N \in C^{N \times N}$ is the DFT matrix with entries $(\mathbf{W}_N)_{j,k} = W_N^{jk}$. While conceptually elegant, this form highlights the inefficiency of the naive approach: full matrix-vector multiplication with $O(N^2)$ complexity.

Fast Fourier Transform (FFT) The FFT algorithm reduces this complexity to $O(N \log N)$ by exploiting the periodic and symmetric properties of W_N. The most well-known approach, the Cooley–Tukey algorithm, recursively decomposes a DFT of size N into two DFTs of size $N/2$ by separating even and odd-indexed terms:

$$X_k = \sum_{n=0}^{N/2-1} x_{2n} W_N^{2nk} + W_N^k \sum_{n=0}^{N/2-1} x_{2n+1} W_N^{2nk}.$$

Each recursion step reduces the problem size by half, leading to a recursive time complexity of:

$$T(N) = 2T(N/2) + O(N) \Rightarrow T(N) = O(N \log N).$$

This makes the FFT suitable for real-time processing and high-resolution spectral analysis.

Two-Dimensional FFT

The FFT extends naturally to multidimensional signals. For a 2D signal (e.g., an image) represented by a matrix $f_{j,k}$ of size $N_1 \times N_2$, the two-dimensional DFT is defined as:

$$F_{m,n} = \sum_{k=0}^{N_2-1} \sum_{j=0}^{N_1-1} e^{-2\pi ikn/N_2} \cdot e^{-2\pi ijm/N_1} \cdot f_{j,k}.$$

This can be separated as:

$$F_{m,n} = \sum_{k=0}^{N_2-1} e^{-2\pi ikn/N_2} \left(\sum_{j=0}^{N_1-1} e^{-2\pi ijm/N_1} \cdot f_{j,k} \right),$$

showing that the 2D DFT is separable: we can first apply a 1D DFT to each row, then to each column of the result. This allows efficient reuse of 1D FFT code for 2D transforms, provided both N_1 and N_2 are powers of two, a requirement for radix-2 implementations.

Although this approach is not always optimal for specialized architectures (e.g., GPU memory layout), it provides a simple and robust method for computing the 2D FFT in most environments.

3 Implementation Decisions

This section summarizes the main architectural and algorithmic choices behind the `fft` project, emphasizing extensibility, performance, and correctness.

All FFT variants share a unified interface through the abstract base class `BaseTransform`. C++20 features like concepts and `if constexpr` enable type-safe, dimension-aware specializations for both 1D and 2D transforms.

The following implementations extend this interface:

- **Iterative FFT**: Cooley–Tukey algorithm with bit-reversal permutation.
- **Recursive FFT**: Divide-and-conquer variant included for completeness.
- **MPI Parallel FFT**: Enables distributed computation for large-scale problems.
- **GPU Parallel FFT**: CUDA-based acceleration via `ParallelFourier`.

4 Iterative FFT

Our starting point is the *iterative Cooley–Tukey algorithm*, a widely used method for computing the Fast Fourier Transform (FFT) [**?**]. The algorithm transforms a time-domain signal into its frequency-domain representation in $\mathcal{O}(n \log n)$ time.

This method consists of two main steps:

- **Bit-reversal permutation:** This step ensures that input indices are reordered correctly before applying the FFT. Given an input array X of length n, we define $m = \log_2(n)$ as the number of bits required to represent n in binary form. The reordering is performed by reflecting each index around the center bit, located at position $m/2$.

  For example, given an array of length 8, we represent each index using 3 bits ($m = 3$) and reverse their order:

$$001 \to 100$$
$$110 \to 011$$
$$111 \to 111$$

This operation ensures the correct memory access pattern required for efficient FFT computation.

- **Butterfly operations:** These operations are performed iteratively over $\log_2(n)$ stages. At each stage, input values are combined in pairs using complex multiplications with precomputed twiddle factors (roots of unity). The butterfly structure allows efficient recombination of smaller DFTs into larger ones. A butterfly operation updates two values $X[m]$ and $X[m + d/2]$ as follows:

$$X[m] = x + \omega \cdot y, \quad X[m + d/2] = x - \omega \cdot y,$$

where $x = X[m]$, $y = X[m + d/2]$, ω is a complex twiddle factor, and d is the size of the current segment in the iteration.

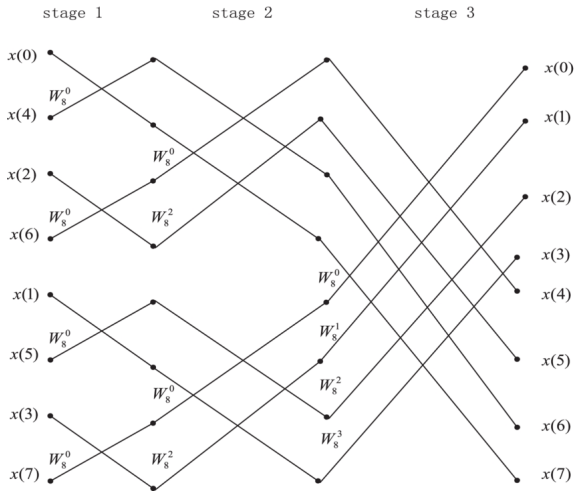


Figure 1: Signal flow graph representing the butterfly effect

Time Convergence Analysis Figure 2 shows the computation time as a function of the problem size N on a log-log scale for both the iterative and recursive FFT implementations. A reference slope of $\mathcal{O}(N \log N)$ is included for comparison, serving as a theoretical benchmark for the expected performance of FFT algorithms.

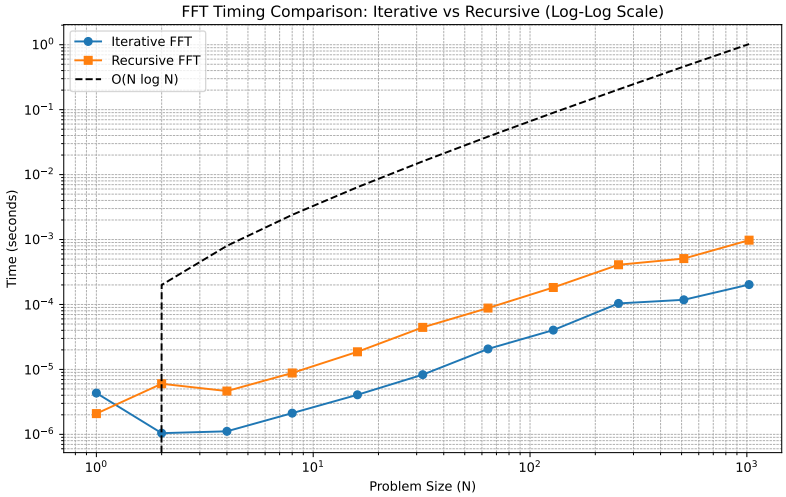


Figure 2: Computation time of iterative and recursive FFT implementations on a log-log scale, with reference $\mathcal{O}(N \log N)$ complexity.

5 CPU Parallel Approach - MPI Implementation

This section outlines the parallelization strategies applied to the Fast Fourier Transform (FFT) algorithm on CPU architectures. The implementation uses a combination of MPI and OpenMP granting a distributed memory paradigm while at the same time maximizing the performance of each core by dividing the work among different threads.

5.1 Architecture Overview

The `MpiIterativeFourier` class extends the base transform interface to support both 1D and 2D distributed FFTs. Key architectural features include:

- **Hybrid Parallelism**: Combines MPI processes with OpenMP threads within each process.
- **Row-Column Decomposition**: For 2D FFTs, the algorithm performs row-wise then column-wise transformations using an intermediate transposition step.
- **Efficient Communication**: Utilizes collective MPI operations (`MPI_Scatter`, `MPI_Gather`) to reduce communication overhead.
- **In Root Transposition**: Minimizes memory overhead when the whole matrix is transposed.

5.2 FFT Data Distribution Strategy

1D

The 1D FFT is implemented using the Cooley-Tukey algorithm with distributed butterfly operations. The data distribution follows these steps:

1. Bit-reversal permutation is performed on rank 0.
2. Input data is evenly scattered across processes.
3. Each process performs local FFT computations on its assigned chunk.
4. For butterfly stages that span across processes, inter-process communication is performed.
5. Final results are gathered and reassembled on rank 0.

In particular considering the example in figure 1, it is evident that in stage 1 we are going to have $(n/2) = 4$ blocks of iteration, each consisting of 2 computations, in the second stage $(n/4) = 2$ blocks, each consisting of 4 computations, etc... So considering our input of size 8 and for example 4 processors we could give each processor a block in stage 1, then we are gonna use 2 for stage 2, until the last stage where the computation is simplified by delegating it to a single core. This ensures efficient parallelization while avoiding synchronization complexity in the critical final stage.

2D

The 2D FFT completely overturns the approach used in the 1D case. Basically we completely forget about the butterfly operation and instead divide the computation od 1D FFT of rows and columns to different processes.

The steps followed by the 2D MPI implementation are teh following:

1. Rows of the input matrix are distributed among MPI processes.
2. Each process applies a 1D FFT to its assigned rows, each being distributed among threads.
3. Results are gathered and transposed on rank 0.
4. Transposed data is redistributed for column-wise 1D FFT computation, again dividing the intra-process works with OpenMP.
5. After column transforms, results are optionally normalized (for inverse FFT) and transposed back to their original orientation.

5.3 Communication Patterns

To achieve efficient distributed computation, MPI collective memory operations are employed:

Listing 1: MPI Communication

```
MPI_Scatter(rank == 0 ? sendBuff.data() : nullptr, local_rows * cols, MPI_DOUBLE_COMPLEX,
            recBuff.data(), local_rows * cols, MPI_DOUBLE_COMPLEX, 0, comm);

MPI_Gather(recBuff.data(), local_rows * cols, MPI_DOUBLE_COMPLEX,
           rank == 0 ? recvbuf.data() : nullptr, local_rows * cols, MPI_DOUBLE_COMPLEX,
           0, comm);
```

5.4 Performance and Scalability Considerations

Several design decisions are made to improve performance:

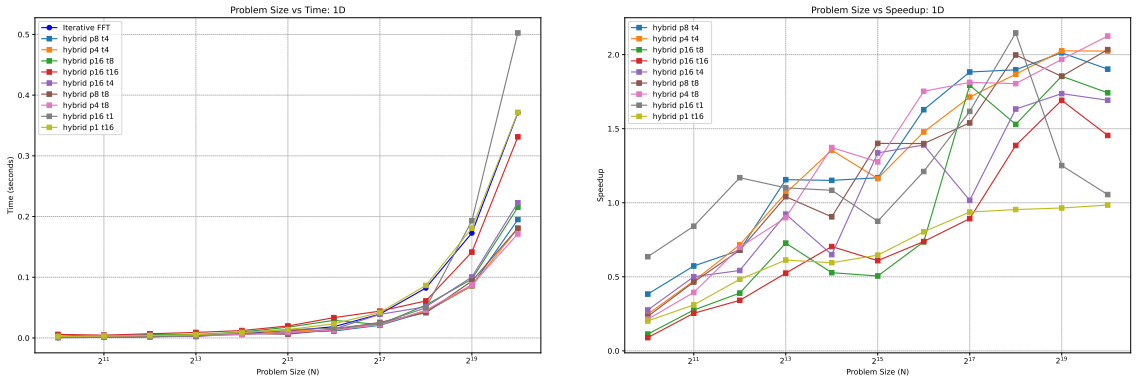
- **In-place Transposition**: Reduces memory overhead using the `transpose2D_more_efficient` routine.
- **Flattened Buffers**: Ensures data contiguity during MPI communication. In fact, whenever rows and columns of the matrix are distributed among the processes a `flatten` and then `unflatten` routine is necessary since MPI memory distribution works only on 1 dimensional data.
- **OpenMP Parallelization**: Enhances intra-process performance through thread-level parallelism.
- **Normalization Strategy**: For inverse FFT, normalization is applied locally by all processors directly on the still flattened data, therefore improving data locality of the `#pragma for loop`.

Note: keeping memory passing calls just on the rank 0 process was not feasible given the limited amount of processes in our devices.

5.5 Performance Evaluation

The implementation is benchmarked against the FFT iterative implementation. The script `run.sh` supports testing across various process and thread configurations, giving a more detailed performance analysis.

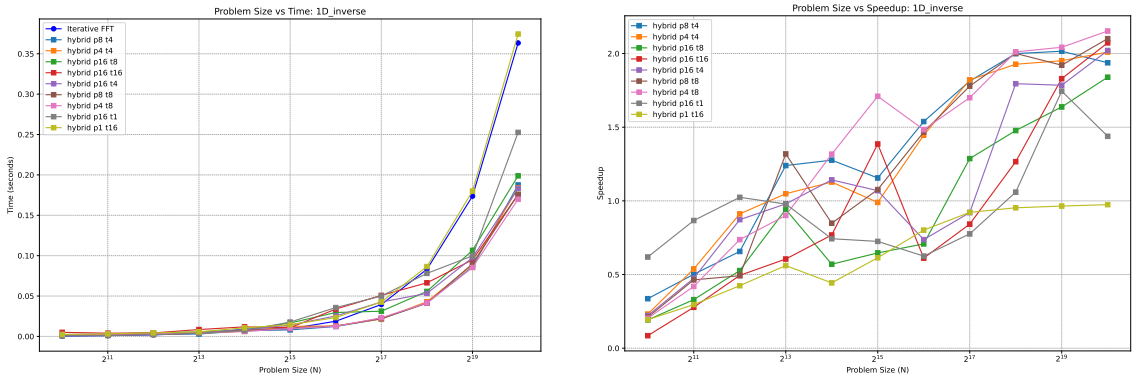
1D performance



(a) Execution time vs input size

(b) Speedup vs input size

Figure 3: Performance of 1D FFT using MPI across varying input sizes.



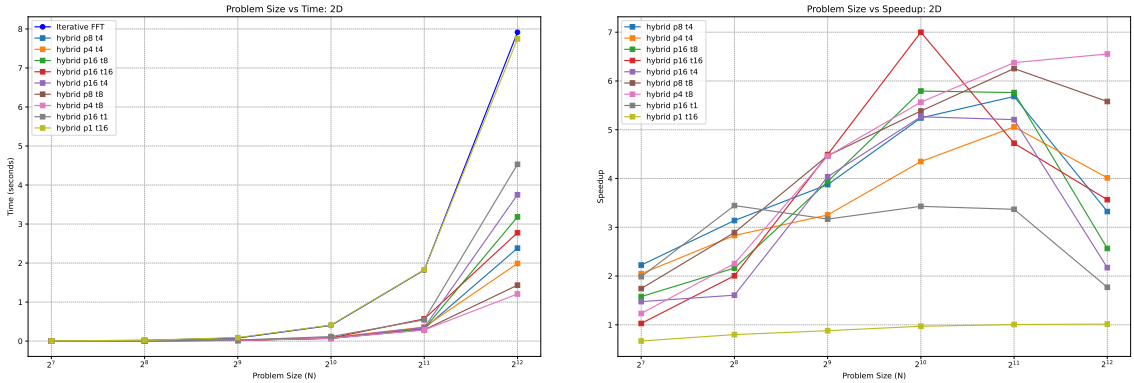
(a) Execution time vs input size

(b) Speedup vs input size

Figure 4: Performance of 1D Inverse FFT using MPI across varying input sizes.

All configurations, except the one using only a single process, show a somewhat relevant speedup after a certain input size is reached. The best possible configurations are the one with a good balance between the number of processes and threads, especially the ones using 4/8 processes and threads. This shows that for 1D computation there is not enough parallel work to warrant the memory and message overhead created by the usage of MPI.

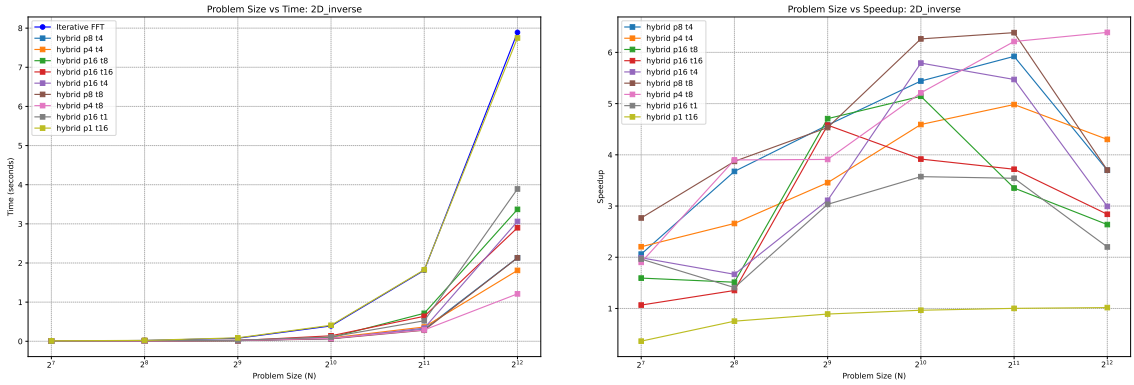
2D performance



(a) Execution time vs input size

(b) Speedup vs input size

Figure 5: Performance of 2D FFT using MPI across varying input sizes.



(a) Execution time vs input size

(b) Speedup vs input size

Figure 6: Performance of 2D Inverse FFT using MPI across varying input sizes.

A more robust and consistent speedup comes into play in the 2D FFT computation, with peaks of more than 6x speedup compared to the iterative version. It is important to note that all considerations regarding butterfly optimization were discarded. Instead, the workload was divided at a higher level, assigning each process a set of rows or columns to which the FFT was applied.

Except for the 1 process configuration, all other configurations exhibit a significant speedup. It is evident how only in the 2D case there is enough parallel computation to make the memory and communication overhead caused by MPI processes convenient.

Given the limitations of the device used to run the MPI version, attempting to brute-force speedup by simply adding more processes proves counterproductive. As in the 1D case, a more balanced configuration once again turns out to be the most optimal.

Moreover, we observed that for input sizes larger than 2^{12}, the memory required to perform the local 2D computation exceeded the available memory on the device's cores, thereby limiting the speedup achievable by the processes.

6 GPU Parallel Approach - CUDA Implementation

To fully exploit the parallelism offered by modern GPU architectures, the traditional butterfly-based FFT computation is reformulated into a single-element computation model. This transformation ensures that each thread operates independently and avoids concurrent writes to the same memory location, thereby eliminating the need for atomic operations and reducing synchronization overhead.

6.1 Implementation Idea

The parallel implementation introduces two auxiliary data buffers, T and Z, each with the same dimensions as the input array X. These buffers alternate roles across computation stages to ensure memory access conflicts are avoided.

Write Location Assignment In each FFT stage $j \in [1, \log_2(n)]$, the stride length is defined as $d = 2^j$. For a given thread index `thread_index`, we define a boolean function:

$$\text{is\_primary\_writer}(\texttt{thread\_index}) = \begin{cases} 0, & \text{if } (\texttt{thread\_index} \bmod d) < \frac{d}{2} \\ 1, & \text{otherwise} \end{cases}$$

which determines whether the current thread is responsible for computing and writing to the forward-pair positions `thread_index` and `thread_index` $+ d/2$, or the reverse-pair `thread_index` and `thread_index` $- d/2$. Specifically:

- If is_primary_writer(`thread_index`) = true, then the thread writes to $T[\texttt{thread\_index}]$ and $T[\texttt{thread\_index} + d/2]$.
- If is_primary_writer(`thread_index`) = false, then the thread writes to $Z[\texttt{thread\_index}]$ and $Z[\texttt{thread\_index} - d/2]$.

This scheme guarantees conflict-free writes, as no two threads access the same memory location in a given stage.

Value Computation The values written to T and Z depend on the role of the current thread:

- For primary writers (is_primary_writer(`thread_index`) = true), the thread simply forwards its input value:

$$T[\texttt{thread\_index}], T[\texttt{thread\_index} + d/2] \leftarrow X[\texttt{thread\_index}].$$

- For non-primary writers, a twiddle factor is applied:

$$Z[\texttt{thread\_index}], Z[\texttt{thread\_index} - d/2] \leftarrow W_N^p \cdot X[\texttt{thread\_index}],$$

  where $W_N = e^{-2\pi \mathrm{i}/d}$ and $p = (\texttt{thread\_index} - d/2) \bmod d$.

Output Update Once all pairwise contributions have been written, each element of the output array is updated as follows:

$$X[\texttt{thread\_index}] = \begin{cases} T[\texttt{thread\_index}] + Z[\texttt{thread\_index}], & \text{if is\_primary\_writer}(\texttt{thread\_index}) = \text{true} \\ T[\texttt{thread\_index}] - Z[\texttt{thread\_index}], & \text{otherwise} \end{cases}$$

This procedure is repeated across all FFT stages until the final result is obtained.

6.2 Memory Management and Access Optimization

The GPU implementation employs *unified memory* via `cudaMallocManaged`, allowing shared memory space between host and device and eliminating the need for explicit memory transfers. All major arrays, including the input, output, and intermediate buffers, are allocated using this method.

To ensure high throughput, the implementation is designed around *coalesced memory access*. During computation, threads are assigned contiguous blocks of data, aligning with the GPU's memory transaction model. This is particularly relevant in the batched 1D and 2D FFT kernels, where threads within a warp access adjacent memory addresses, reducing access latency and maximizing bandwidth.

For 1D FFTs, memory allocation is proportional to the vector size N, potentially padded to match warp or block sizes. In the 2D case, memory is allocated for the full matrix size $N_1 \times N_2$, ensuring column and row operations can proceed without reallocation. After computation, results are accessed directly from unified memory and all resources are released using `cudaFree`.

This design enables efficient parallel computation of FFTs on GPUs, balancing algorithmic structure, memory efficiency, and hardware utilization.

6.3 Stream Utilization for 2D FFT

To further accelerate the 2D FFT computation, the implementation exploits CUDA streams to enable concurrent execution of independent operations. Specifically, after performing the FFT along the rows, the algorithm processes each column in parallel by assigning a separate CUDA stream to each column. This approach allows the GPU to overlap the computation of different columns, maximizing device occupancy and reducing overall execution time. Each stream launches a kernel dedicated to computing the FFT for its assigned column, ensuring that memory accesses remain coalesced and that there are no dependencies between streams. By leveraging multiple streams in this way, the implementation achieves a high degree of parallelism and efficiently utilizes the available GPU resources, particularly for large matrix sizes where the number of columns is substantial. This strategy is especially effective on modern GPUs, which are capable of handling many concurrent streams and overlapping computation with memory transfers.

6.4 Performance Evaluation

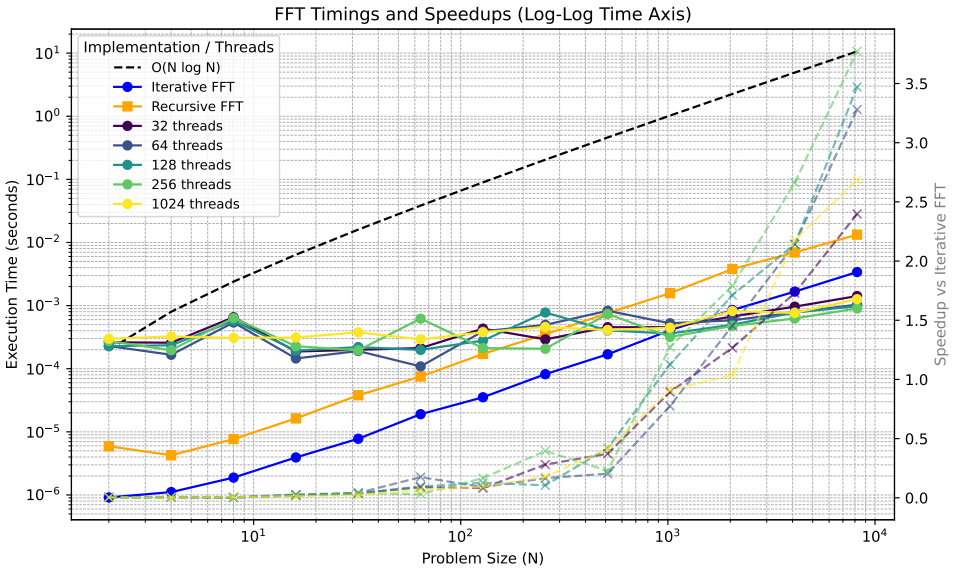


Figure 7: 1D FFT: Execution time and speedup comparison on a log-log scale. The right Y-axis shows execution time across increasing problem sizes, while the left Y-axis shows the speedup relative to the sequential iterative implementation. Each CUDA configuration is represented by a line labeled with the number of threads. Theoretical complexity $O(N \log N)$ is also plotted for reference.

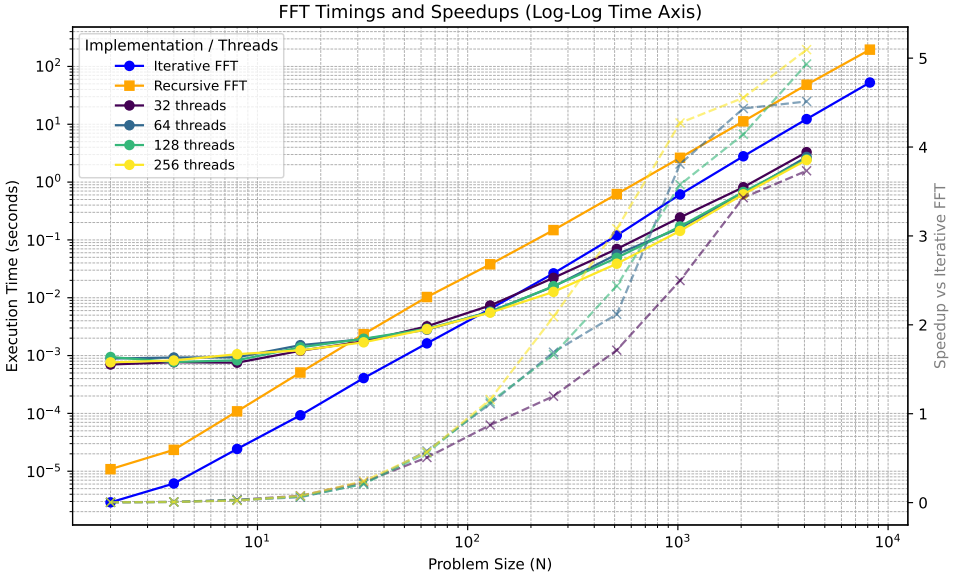


Figure 8: 2D FFT: Execution time and speedup comparison. Similar to the 1D case, performance is evaluated across multiple thread configurations, with speedup shown relative to the sequential iterative FFT. The chart illustrates that higher thread counts generally improve performance, especially for larger input sizes.

From the speedup curves in Figures 7 and 8, we observe that increasing the number of threads per block generally leads to lower execution times and higher speedup values, especially for large input sizes. However, the marginal gains diminish at higher thread counts. For small to mid-sized inputs,

performance with 128 or 256 threads per block is often comparable to that with 1024 threads.

In CUDA, this can be attributed to the trade-off between thread-level parallelism and hardware resource utilization. Using fewer threads per block may result in higher occupancy and lower register pressure, which can benefit some kernels. However, for compute-intensive workloads, larger thread blocks can improve throughput by better utilizing GPU cores and reducing launch overhead.

Therefore, the optimal configuration depends on the specific workload and resource demands. In our case, configurations with 128 or 256 threads per block strike a good balance between performance and resource efficiency, making them suitable for a wide range of input sizes without significantly compromising execution speed.

7 Image Compression via 2D FFT

Image compression is a key application of the Fast Fourier Transform (FFT), enabling the reduction of data size by eliminating frequency components that contribute least to visual perception.
This section presents the image compression pipeline developed using 2D FFT and two frequency filtering strategies: magnitude-based and band-pass. The goal is to evaluate which method best balances compression and visual quality.

7.1 Compression Pipeline

The compression process includes the following steps:

1. Convert image to grayscale and normalize.
2. Apply 2D FFT via row-column decomposition.
3. Compute the magnitude spectrum of the transformed image.
4. Apply filtering using one of the following methods:
    - **Magnitude-based filtering**: Sort frequency coefficients by magnitude and discard the bottom X%, retaining the top (100–X)% most significant components.
    - **Band-pass filtering**: Retain frequency coefficients lying within a defined circular band in the 2D frequency plane, based on distance from the spectrum center.
5. Apply inverse 2D FFT and rescale to obtain the reconstructed image.

7.2 Implementation Notes

The image preprocessing and visualization steps were implemented using OpenCV, a widely adopted library for computer vision. OpenCV was used for loading images, converting them to grayscale, normalizing pixel values, and exporting results in standard formats such as PNG and JPEG. Images were resized to a fixed resolution and converted to 32-bit floating point to ensure numerical stability.
Reconstructed outputs were saved in `images/image_output/`, while diagnostic plots—such as FFT magnitude and PSNR—were generated via Python and stored in `images/plot_result/`.
The FFT logic was developed in C++ using a modular and extensible architecture. The 2D FFT was computed via row-column decomposition, building on a reusable 1D engine. Frequency data was stored as `std::vector<std::vector<std::complex<double>>>`, enabling both direct access and efficient manipulation. Filtering strategies (e.g., magnitude-based and band-pass) operated directly on this representation. The `IterativeFourier<T>` template class provided both forward and inverse transform capabilities, and the design supports easy integration of alternative FFT variants without altering the filtering pipeline.

7.3 Frequency Spectrum Visualization

To understand the effect of different filtering strategies in the frequency domain, we compare the FFT magnitude spectra before and after applying two types of filters: magnitude-based thresholding and band-pass filtering.

In the first approach, all frequency components are sorted in ascending order by magnitude, and a threshold is set to discard the bottom X% of coefficients, retaining only the top (100–X)%. For instance, a 25% threshold removes the weakest 25% of frequencies, highlighting components that contribute most to the image's energy, regardless of their spatial location.

Conversely, band-pass filtering preserves frequencies based solely on their radial distance ρ from the spectrum center (DC component), using a circular mask defined by an inner and outer radius. This selects a frequency band, discarding components outside this range. Unlike magnitude-based filtering, this method may retain low-energy components while excluding strong ones if they fall outside the specified band.

Both techniques are visualized below using a logarithmic scale to emphasize contrast. The first figure compares the original spectrum with the result of discarding the lowest 25% of magnitudes. The second shows the effect of a band-pass filter retaining 75% of frequencies centered around the middle range.

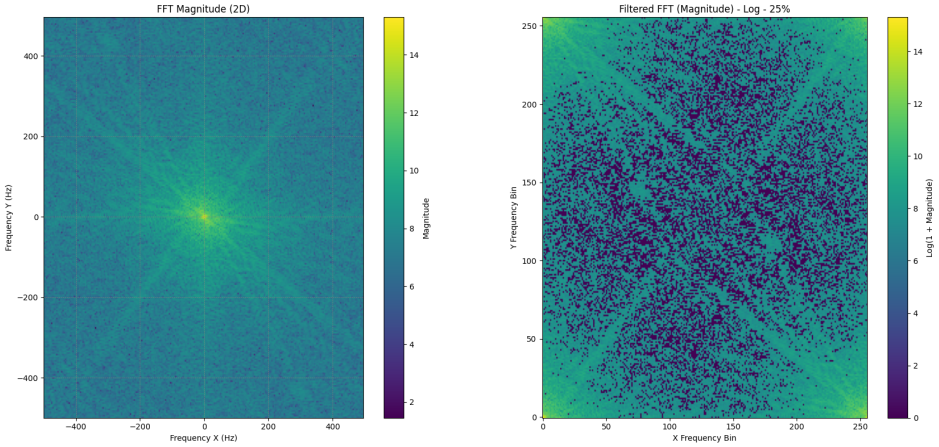


Figure 9: FFT magnitude spectra (log scale). Left: Original image spectrum. Right: After removing the bottom 25% of frequency components by magnitude.

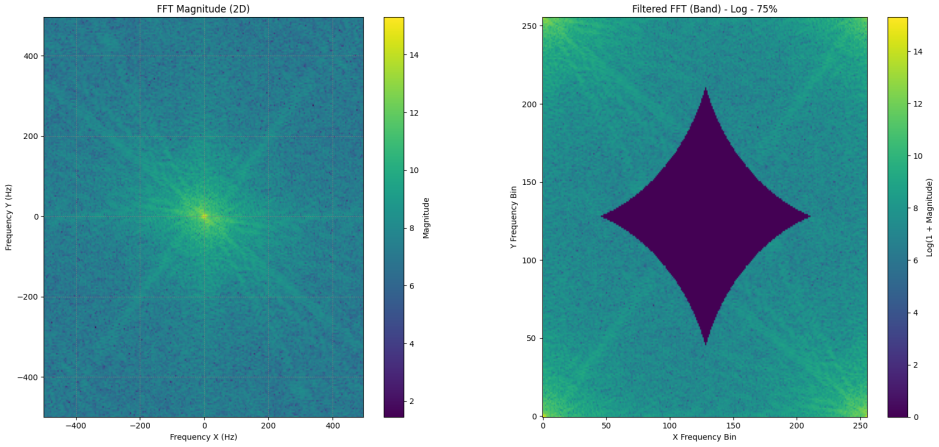


Figure 10: FFT magnitude spectra (log scale). Left: Original image spectrum. Right: After applying a band-pass filter (75% width around mid frequencies).

7.4 Visual Comparison at Different Compression Levels

We compare reconstructed images using both strategies at five compression thresholds (5%, 25%, 50%, 75%, 95%).

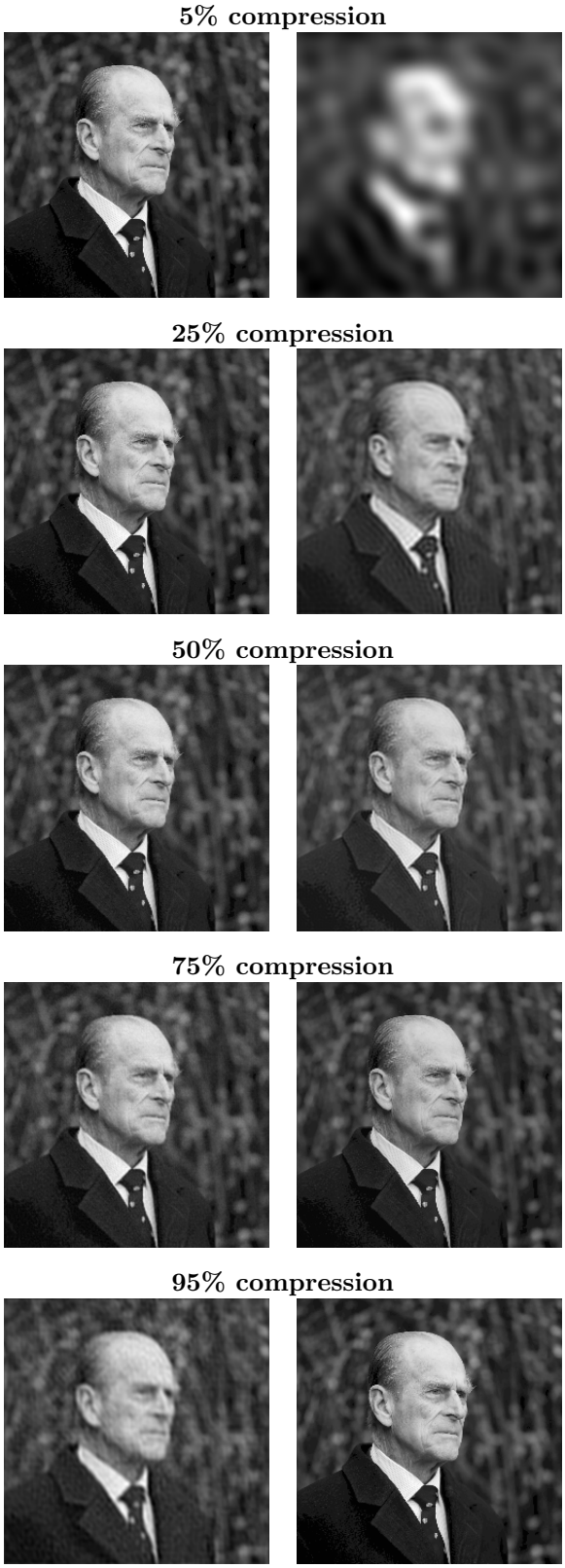


Figure 11: Reconstructed images at different compression thresholds. Left: Magnitude-based filtering. Right: Band-pass filtering.

7.5 Quantitative Analysis

We evaluate the reconstruction quality using PSNR (Peak Signal-to-Noise Ratio), a standard metric in image processing. Higher PSNR values correspond to better visual fidelity, with values above 30 dB generally indicating acceptable quality.

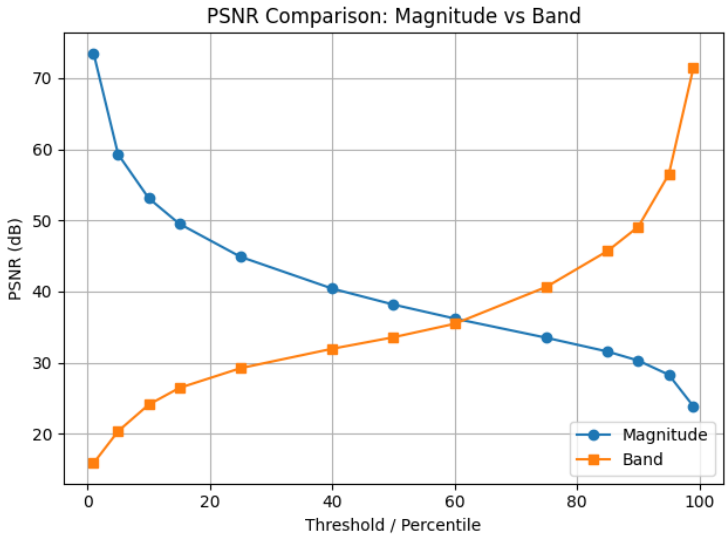


Figure 12: PSNR comparison between magnitude-based and band-pass filtering. Magnitude filtering consistently achieves higher PSNR at lower compression thresholds.

7.6 Error Comparison

To quantitatively compare the two filtering strategies, we analyze the reconstruction error (RMSE) as a function of the compression threshold.

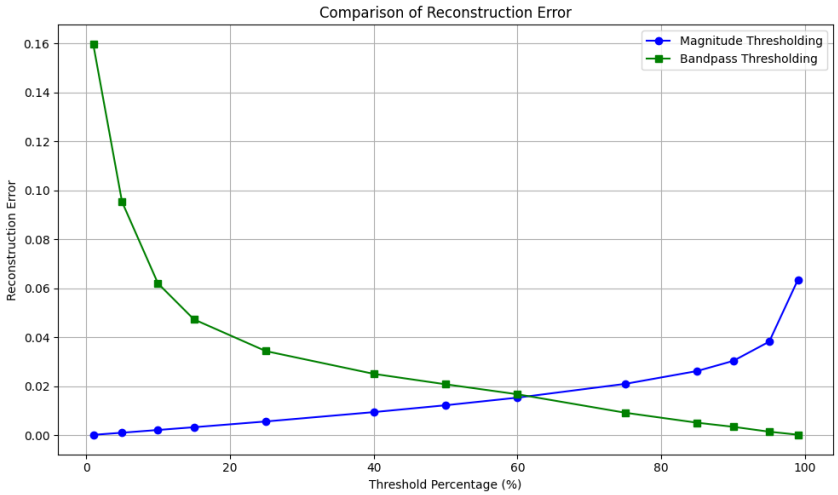


Figure 13: Reconstruction error (RMSE) vs. compression percentage. Magnitude thresholding exhibits lower error at low compression levels, while band-pass filtering performs better as more frequency components are retained.

As shown in the plot, magnitude-based filtering yields significantly lower reconstruction error when only a small percentage of the spectrum is retained. However, as more frequencies are preserved, band-pass filtering becomes more effective in reducing the error.

7.7 Discussion

Magnitude-based filtering consistently preserves more visual detail than band-pass filtering at the same compression levels. This is because it retains the most relevant frequency components based on their energy, regardless of position. In contrast, the band-pass approach may discard high-magnitude coefficients simply because they lie outside the selected radial band, leading to visible artifacts and lower PSNR.

Conclusion: Magnitude-based thresholding offers a superior balance between compression ratio and visual fidelity, making it the preferred choice for FFT-based image compression.

8 Conclusion

Our results highlight the central importance of the FFT algorithm in computational tasks and demonstrate the effectiveness of our parallelization strategies across both CPU (MPI) and GPU (CUDA) platforms. The proposed implementations achieve significant speedups and performance improvements compared to the baseline sequential approach.

However, there is still room for optimization. In particular, reordering indices dynamically at each computation step could reduce memory access overhead. Additionally, further improvements could be achieved by more fully exploiting the available hardware, such as better load balancing across cores or leveraging shared memory and instruction-level parallelism more aggressively.

Overall, our work provides a flexible and performant FFT framework that supports multiple architectures and dimensionalities, offering a strong foundation for further development in high-performance spectral computation.

Furthermore, we applied FFT to a practical use case—image compression—showing how frequency domain filtering techniques affect both the compression ratio and the visual fidelity of reconstructed images. Quantitative metrics such as PSNR and RMSE confirmed that magnitude-based filtering provides more effective compression at low thresholds compared to simple spatial band-pass methods.