2024 Fall CMPT 431 Project Report Fast Fourier Transform

Henry Chau, Bogdan Badea

Introduction (what is the problem you're solving)

The Fast Fourier Transform (FFT) is a computationally efficient algorithm for calculating the Discrete Fourier Transform (DFT), which transforms signals from their original domain (usually time or space) to the frequency domain. It is one of the most important algorithms in science and engineering, with numerous real-world applications, including signal analysis, audio filtering, and image compression.

In this project, we will implement three versions of the FFT:

- A serial version (single-threaded, single-process)
- A parallel version (single-threaded, single-process)
- A distributed version (single/multi-threaded, multi-process)

We will test and compare these implementations to find out which one is the most efficient. We will also look into what makes the fastest FFT algorithms work better and how they improve performance.

Background (what is the state of the art in serial and parallel implementations of the problem

The basic DFT is shown in the equation below. When calculated directly, it has a time complexity of $O(n^2)$, which can be slow for large data sets.

$$Y[i] = \sum_{k=0}^{n-1} X[k]\omega^{ki}, \quad 0 \leq i < 0, \quad \omega = e^{2\pi\sqrt{-1}/n}$$

The FFT is a divide-and-conquer algorithm for computing the DFT, introduced by James Cooley and John Tukey in 1965[1]. Originally developed to detect nuclear tests by the Soviet Union, it later proved to be very useful in many other applications. The FFT works by recursively breaking down the original data into two smaller parts, which reduces the time complexity to $O(n \log n)$.

The equation below shows how the DFT is split into smaller problems(odd and even parts). There are different types of FFT algorithms, such as the Prime-factor FFT, but this project focuses on the Radix 2 Cooley-Tukey FFT.

$$Y[i] = \sum_{k=0}^{(n/2)-1} X[2k]\omega^{2ki} + \sum_{k=0}^{(n/2)-1} X[2k+1]\omega^{(2k+1)i}$$

Fastest Fourier Transform in the West(FFTW) is a popular and efficient implementation of the Fast Fourier Transform, developed by Matteo Frigo and Steven Johnson. It supports serial, parallel, and distributed versions of FFT. FFTW is highly optimized, customizable, and can adapt to different hardware configurations. It uses a plan-based approach: before computation begins, a plan is created at runtime to select the best combination of codelets (optimized C code blocks generated by a specialized compiler)[2]. The executor then runs these codelets to perform the transformation.

Another option is to use the parallel processing power of GPUs to speed up the algorithm. Nvidia's CUDA Fast Fourier Transform library (cuFFT) is a popular tool for performing FFT efficiently on GPUs.

Implementation Details (what you have implemented in the serial, parallel and distributed versions)

For simplicity, we restrict the input size n to powers of two. Inputs that are not powers of two can still be used by adding zero padding.

FFT and inverse FFT (IFFT) are nearly identical operations, with two key differences: the exponent sign of ω is positive in IFFT, and the IFFT results must be normalized at the end. As a result, both can be combined into a single function. A boolean argument is used to determine the direction.

Serial

This is a recursive version of the serial implementation. While it is simple and easy to read, its recursive nature makes it hard to optimize and adapt for parallel version.

```
std::vector<std::complex<double>> recursiveFFT(
    const std::vector<double> &X, std::complex<double> omega) {
    std::vector<std::complex<double>> Y(X.size());
    const int n = X.size();
    if (n == 1) {
        Y[0] = X[0];
        return Y;
    }
    // Populate new vectors containing only odd or even indexed elements
    std::vector<double> xEven;
    for (int i = 0; i < n; i += 2) {
        xEven.push_back(X[i]);
```

```
    }
    std::vector<double> xOdd;
    for (int i = 1; i < n; i += 2) {
        xOdd.push_back(X[i]);
    }
    std::vector<std::complex<double>> left = recursiveFFT(xEven, std::pow(omega, 2));
    std::vector<std::complex<double>> right = recursiveFFT(xOdd, std::pow(omega, 2));
    for (int i = 0; i < n; ++i) {
        Y[i] = left[i % (n / 2)] + std::pow(omega, i) * right[i % (n / 2)];
    }
    return Y;
}
```

This is the FFT algorithm proposed in the *Introduction to Parallel Computing(IPC)* textbook[3]. During computation, the input array remains unordered, requiring bit manipulation to determine the correct indices of j, k and the twiddle factor. A cache is used to avoid repeated calculation of omega. The main advantage of this algorithm is that the results computed at each stage are stored contiguously, making it well-suited for data exchange in multi-process setups. The result needs to be reordered at the end of the algorithm.

```
std::vector<std::complex<double>> iterativeIcpFft(
    const std::vector<std::complex<double>> &X, bool isInverse) {
    const int n = X.size();
    const int r = std::log2(n);
    const std::complex<double> img(0.0, 1.0);
    double exponentSign = isInverse ? 1.0 : -1.0;
    std::complex<double> omegaBase = std::exp(exponentSign * 2.0 * img
        * std::numbers::pi / (double)n);
    // Result array
    std::vector<std::complex<double>> R(X);
    // Auxillary array to hold previous value of R
    std::vector<std::complex<double>> S(R);
    // Store previously calculated omega
    std::map<int, std::complex<double>> omegaCache;
    // Outer loop O(log n), m represent stage
    for (int m = 0; m < r; ++m) {
        // Inner loop O(n)
        for (int i = 0; i < n; ++i) {
            // Let (b_0 b_1 ... b_r-1) be the binary representation of I
            // Clear the bit, j:= (b_0 ... b_m1 0 b_m+1 ... b_r1)
            unsigned int j = i & ~(1 << (r - 1 - m));
            // Set the bit, k:= (b_0 ... b_m1 1 b_m+1 ... b_r1)
            unsigned int k = i | (1 << (r - 1 - m));
            // (b_m b_m-1 ... b-0 ... 0 ... 0)
            unsigned int omegaExp = getFirstNBits(reverseBits(i, r), m + 1) << (r - 1 - m);
            std::complex<double> omega;
            if (omegaCache.find(omegaExp) != omegaCache.end()) {
                omega = omegaCache[omegaExp];
            } else {
                omega = std::pow(omegaBase, omegaExp);
                omegaCache[omegaExp] = omega;
            }
            R[i] = S[j] + S[k] * omega;
```

```
        }
        // Update S with new values
        std::copy(R.begin(), R.end(), S.begin());
    }
    // In-place bit-reversal reordering
    for (int i = 0; i < n; ++i) {
        int reversedIndex = reverseBits(i, r);
        if (i < reversedIndex) {
            std::swap(R[i], R[reversedIndex]);
        }
    }
    // Normalize result if performing IFFT
    if (isInverse) {
        for (int i = 0; i < n; ++i) {
            R[i] /= n;
        }
    }
    return R;
}
```

Parallel

From the iterativeIcp algorithm we can note that each of the iterations of the outerloop depends on the results of the previous iteration, but that there is no such dependency for the inner loop, and that the inner loops only modify their respective indices of the result array, meaning we don't any mutexes for the writes, only barrier to make sure they all stay on the same iteration, and an atomic value to allow them to dynamically go through the array since some indices have more work then others as a result of the twiddlers or index values.

```
std::atomic<int> iterativeIcpFft_covered_i(0);
void multithreaded_iterativeIcpFft_loop_function(const int r, const int n,
            std::vector<std::complex<double>> &R,
            std::vector<std::complex<double>> &S,
            const std::complex<double> omega, std::barrier<> &b, int thread_id,
            int block_size) {
    int start_i;
    int m;
    int i;
    unsigned int omegaExp;
    // Outer loop O(log n)
    for (m = 0; m < r; ++m) {
        if (thread_id == 0) {
            std::copy(R.begin(), R.end(), S.begin());
            iterativeIcpFft_covered_i = 0;
        }
        b.arrive_and_wait();
        // Inner loop O(n)
        start_i = iterativeIcpFft_covered_i.fetch_add(block_size);
        while (start_i < n) {
            for (i = start_i; i < std::min(start_i + block_size, n); ++i) {
                // Let (b_0 b_1 ... b_r-1) be the binary representation of i
                std::bitset<32> j(i);
```

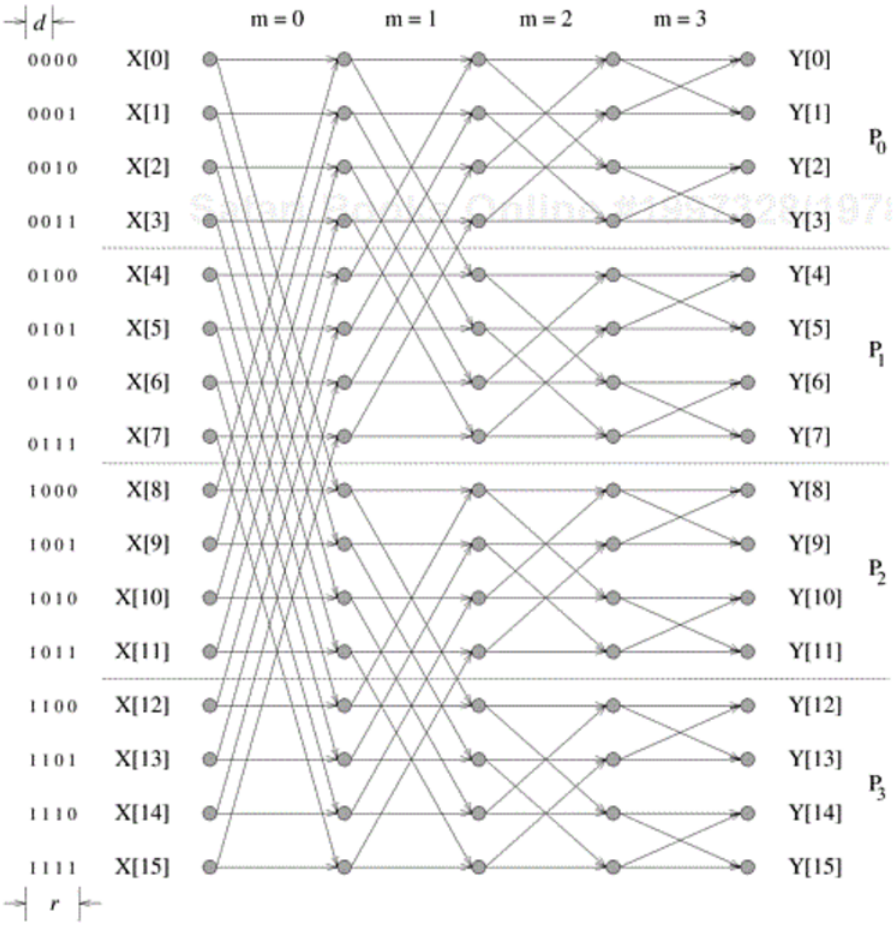
```
                j.reset(r - 1 - m); // (b_0 ... b_m1 0 b_m+1 ... b_r1)
                std::bitset<32> k(i);
                k.set(r - 1 - m); // (b_0 ... b_m1 1 b_m+1 ... b_r1)

                omegaExp = getFirstNBits(reverseBits(i, r), m + 1) << (r - 1 - m);
                // (b_m b_m-1 ... b-0 ... 0 ... 0)
                R[i] = S[j.to_ulong()] + S[k.to_ulong()] * std::pow(omega, omegaExp);
            }
            start_i = iterativeIcpFft_covered_i.fetch_add(block_size);
        }
        b.arrive_and_wait();
    }
}

std::vector<std::complex<double>> multithreaded_iterativeIcpFft(
                    const std::vector<std::complex<double>> &X,
                    bool isInverse, int nThreads, int block_size) {
    const int n = X.size();
    const int r = std::log2(n);
    std::barrier b(nThreads);
    const std::complex<double> img(0.0, 1.0);
    double exponentSign = isInverse ? 1.0 : -1.0;
    std::complex<double> omega = std::exp(exponentSign * 2.0 * img * std::numbers::pi / (double)n);
    std::vector<std::complex<double>> R(X); // Result array
    std::vector<std::complex<double>> S(n); // Auxillary array to hold previous value of R
    std::vector<std::thread> threads;
    for (int i = 0; i < nThreads; ++i) {
        threads.push_back(std::thread(multithreaded_iterativeIcpFft_loop_function, r, n, std::ref(R),
                        std::ref(S), omega, std::ref(b), i, block_size));
    }
    for (auto &t : threads) {
        t.join();
    }
    // In-place bit-reversal reordering
    for (int i = 0; i < n; ++i) {
        int reversedIndex = reverseBits(i, r);
        if (i < reversedIndex) {
            std::swap(R[i], R[reversedIndex]);
        }
    }
    // Normalize result if performing IFFT
    if (isInverse) {
        for (int i = 0; i < n; ++i) {
            R[i] /= n;
        }
    }
    return R;
}
```

Distributed

The figure below, taken from *IPC Ch.13*[3], illustrates how process communication works between each stage in the binary exchange algorithm. In this approach, we observe that $p/2$ pairs of processes

communicate with each other during each communication stage, with each process receiving a new partner at every stage. The communication is performed using a blocking *MPI_Sendrecv* to ensure that the send and receive operations occur almost simultaneously, eliminating the risk of deadlock. Notably, communication only occurs during stages where $m < \log p$

For the last $\log n - \log p$ iterations of the outer loop, the computation is entirely local. This means that these iterations can be treated as a "black-box" operation, making it feasible to utilize other libraries or implement multithreading to further accelerate the local computation.

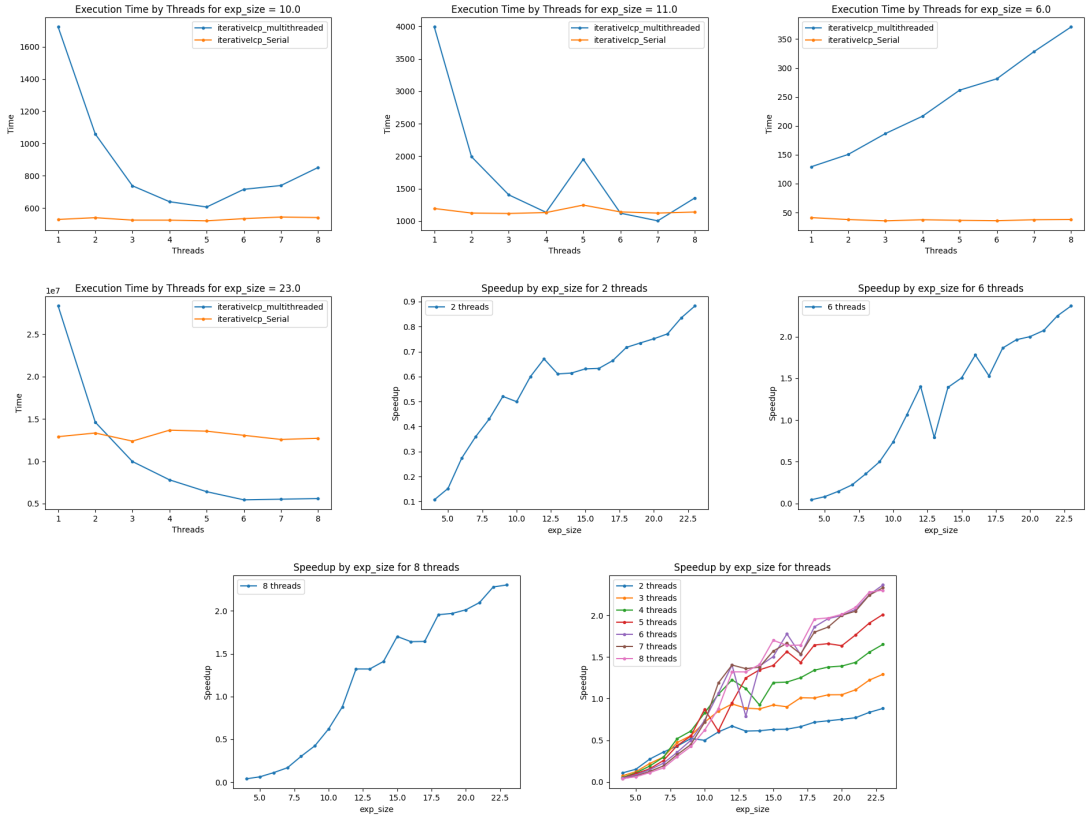



```cpp
std::vector<std::complex<double>> icpFftDistributed(
    const std::vector<std::complex<double>> &X, bool isInverse) {
    int processes, currProcId;
    MPI_Comm_size(MPI_COMM_WORLD, &processes);  // Total number of processes
    MPI_Comm_rank(MPI_COMM_WORLD, &currProcId);  // Current process ID
    const int n = X.size();
    const int r = std::log2(n);
    const int logP = std::log2(processes);
    const int elePerProc = n / processes;
    // Start index of elements owned by this process
    const int startIdx = currProcId * elePerProc;
    const int endIdx = startIdx + elePerProc; // Exclusive end index
    const std::complex<double> img(0.0, 1.0);
    double exponentSign = isInverse ? 1.0 : -1.0;
```

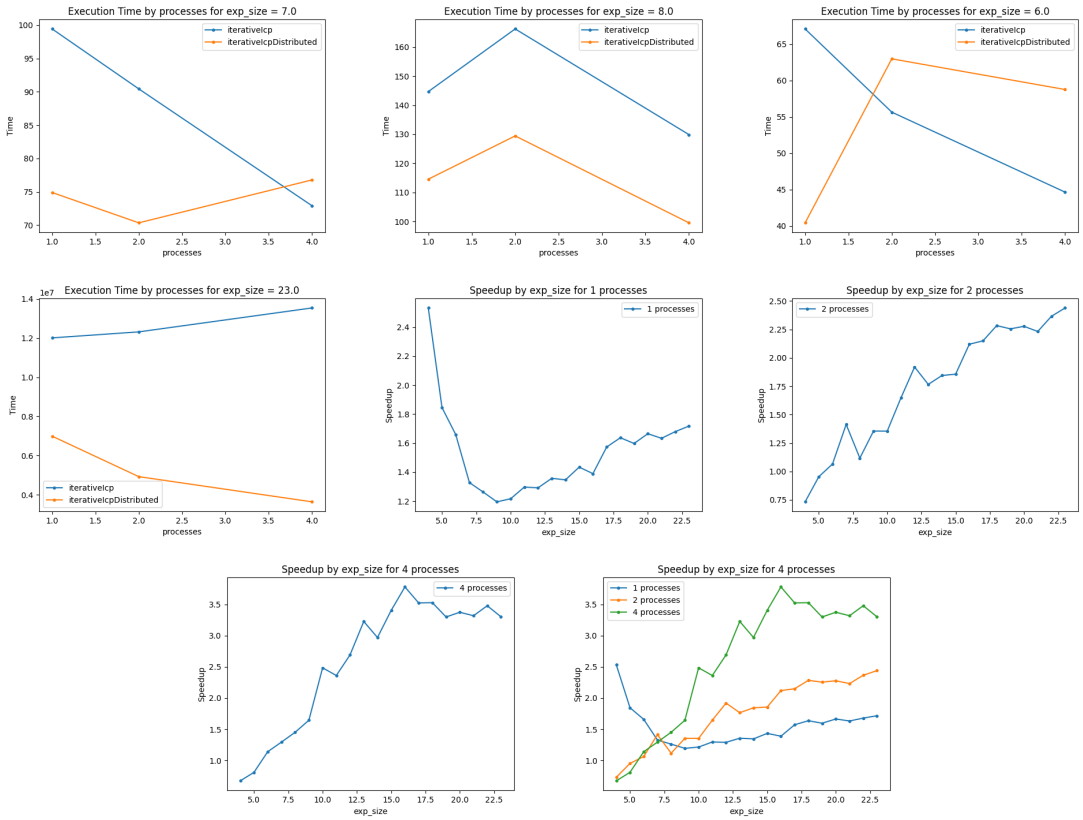
```
    std::complex<double> omegaBase = std::exp(exponentSign * 2.0 * img
        * std::numbers::pi / (double)n);
    std::vector<std::complex<double>> R(n); // Result array
    std::copy(X.begin() + startIdx, X.begin() + endIdx, R.begin() + startIdx);
    std::vector<std::complex<double>> S(n); // Auxillary array to hold previous value of R
    std::copy(R.begin() + startIdx, R.begin() + endIdx, S.begin() + startIdx);
    std::unordered_map<int, std::complex<double>> omegaCache; // Store previously calculated omega
    // Outer loop O(log n), m represent stage
    for (int m = 0; m < r; ++m) {
        // Check if data exchange is needed at this stage
        if (m < (logP)) {
            std::bitset<32> partnerProcIdBits(currProcId);
            partnerProcIdBits.flip(logP - m - 1);
            int partnerProcId = partnerProcIdBits.to_ulong();
            // Exchange data with partner
            MPI_Sendrecv(
                R.data() + startIdx, elePerProc, MPI_DOUBLE_COMPLEX, partnerProcId, m,
                S.data() + partnerProcId * elePerProc, elePerProc, MPI_DOUBLE_COMPLEX,
                partnerProcId, m, MPI_COMM_WORLD, MPI_STATUS_IGNORE
            );
        }
        // Inner loop O(n)
        for (int i = startIdx; i < endIdx; ++i) {
            // Let (b_0 b_1 ... b_r-1) be the binary representation of i
            // Clear the bit, j:= (b_0 ... b_m1 0 b_m+1 ... b_r1)
            unsigned int j = i & ~(1 << (r - 1 - m));
            // Set the bit, k:= (b_0 ... b_m1 1 b_m+1 ... b_r1)
            unsigned int k = i | (1 << (r - 1 - m));
            // (b_m b_m-1 ... b-0 ... 0 ... 0)
            unsigned int omegaExp = getFirstNBits(reverseBits(i, r), m + 1) << (r - 1 - m);
            std::complex<double> omega;
            if (omegaCache.find(omegaExp) != omegaCache.end()) {
                omega = omegaCache[omegaExp];
            } else {
                omega = std::pow(omegaBase, omegaExp);
                omegaCache[omegaExp] = omega;
            }
            R[i] = S[j] + S[k] * omega;
        }
        // Update S with new values
        std::copy(R.begin() + startIdx, R.begin() + endIdx, S.begin() + startIdx);
    }
    // All processes gather result from other processes
    MPI_Allgather(R.data() + startIdx, elePerProc, MPI_DOUBLE_COMPLEX, R.data(),
        elePerProc, MPI_DOUBLE_COMPLEX, MPI_COMM_WORLD);
    // In-place bit-reversal reordering
    for (int i = 0; i < n; ++i) {
        int reversedIndex = reverseBits(i, r);
        if (i < reversedIndex) {
            std::swap(R[i], R[reversedIndex]);
        }
    }
    // Normalize result if performing IFFT
    if (isInverse) {
```

```
69          for (int i = 0; i < n; ++i) {
70              R[i] /= n;
71          }
72      }
73      return R;
74  }
```

Evaluation (results from your experiments)



In our exploration of the parallel implementation, we discovered that leveraging multiple threads only became advantageous when the problem size reached approximately $\sim 2^{11}$. Before this point, the added overhead from thread management outweighed any computational benefits, leading to longer execution times rather than speedups. This suggests that for smaller problem sizes, the cost of parallelization can be prohibitive. However, as the exponent x in 2^x grew larger, the performance gains from parallel processing became increasingly evident. The speedup improved consistently across all thread counts for sufficiently large problem sizes. Notably, the performance was most effective with higher thread counts, reaching optimal results around six threads, beyond which the benefits began to plateau.



In the case of the distributed version, we found that it incurred significantly less overhead compared to the parallel implementation. This reduced penalty allowed it to become effective at smaller problem sizes, with noticeable benefits starting at around 2^7 rather then 2^{11}. Additionally, the speedup achieved by the distributed version was impressive, closely approaching the ideal of perfect parallelization when a small number of processes were used. However, as the number of processes increased, the speedup began to deviate more noticeably from this ideal, with its proportional gains diminishing most visibly for 4 processes and high exp_size. This highlights the efficiency of the distributed approach for smaller numbers of processes while also illustrating the beginning of diminishing returns that occur as more processes are introduced.

Conclusions (summary of results/speedups, what you learned from the project)

Parallelizing the Fast Fourier Transform (FFT) proves to be significantly more efficient when utilizing OpenMPI for distributed processing compared to multithreading. This is evident from our results, which show that even with just four processes, the distributed implementation surpasses the upper limit of the speedup achieved by six threads in the multithreaded version. This suggests that distributed processing is better suited for scaling the FFT, as it manages to overcome the limitations imposed by thread-based approaches, particularly for larger problem sizes. While parallelizing the FFT can offer substantial performance improvements, it is worth noting that the most effective strategy may involve leveraging specialized hardware and libraries designed

for these tasks. For instance, NVIDIA's CUDA Fast Fourier Transform library (cuFFT) offers highly optimized FFT implementations that are specifically designed to run on GPUs. Offloading computational workloads to a GPU using cuFFT can provide far greater performance gains than either threading or distributed processing on CPUs[4]. This highlights the importance of selecting the right computational framework and hardware platform when optimizing FFT computations, especially for applications requiring high-performance solutions.

References

[1] James W. Cooley, John W. Tukey(1965), An Algorithm for the Machine Calculation of Complex Fourier Series

[2] Matteo Frigo, Steven G. Johnson(1998), FFTW: An Adaptive Software Architecture for the FFT

[3] Ananth Grama (2003), Introduction to Parallel Computing, Addison-Wesley

[4] C. Zhang, Y. Xiao, H. Chen and H. Liu, "Parallel Optimization and Hardware Customization for Fast Fourier Transform," 2023 IEEE 14th International Conference on Software Engineering and Service Science (ICSESS), Beijing, China, 2023, pp. 105-110