

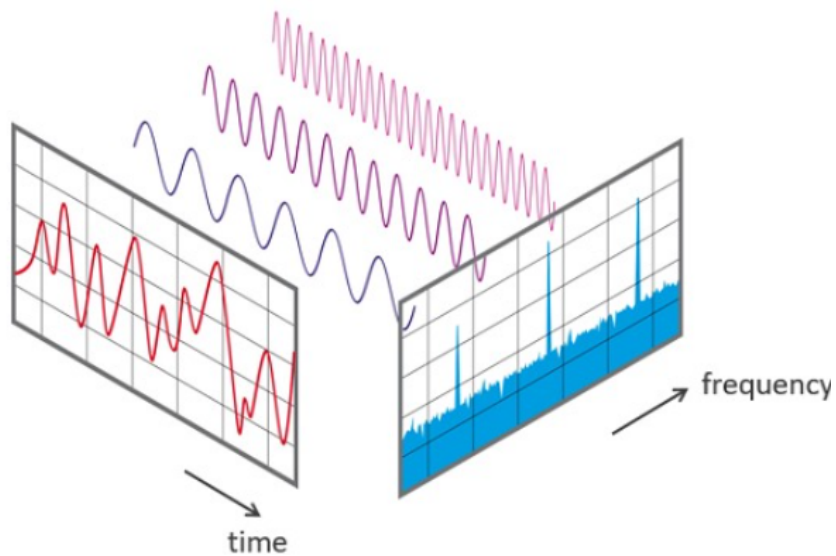
---

# Implementation and Analysis of Fast Fourier Transform (FFT) in Parallel Computing

---

Zainab Hasan and Ikhlas Khan

May 2025



# 1 Introduction

## 1.1 What is FFT?

The Fast Fourier Transform (FFT) is a fast and efficient algorithm used to analyse signals by converting them from the time domain to the frequency domain. In simpler terms, it takes a sequence of numerical values—typically representing a signal that changes over time, such as sound waves, sensor readings, or electrical signals—and breaks it down into the individual frequency components that contribute to the overall signal. These frequency components include both the amplitude (how strong a frequency is) and the phase (how it aligns in time). The output is another list of values, typically complex numbers, where each value corresponds to a specific frequency and indicates how much of that frequency is present in the original signal.

## 1.2 Importance of FFT

FFT is of critical importance in modern computing and engineering applications. It enables efficient and real-time processing of large data streams by providing rapid transformation between time and frequency domains. Applications such as audio and image compression, digital filtering, radar and sonar systems, biomedical signal analysis (e.g., ECG and EEG), and wireless communications all rely heavily on FFT for accurate and timely results. The algorithm's performance and precision make it a vital tool in both theoretical research and practical applications.

## 1.3 Motivation for Parallelizing FFT

Despite the FFT's computational efficiency, processing large-scale datasets on a single processor can still be time-consuming and resource intensive. With the growing demand for high-resolution data in fields like scientific simulations, real-time media processing, and large-scale analytics, single-threaded FFT implementations often become a performance bottleneck. To address this, parallelization of FFT is essential. By distributing the computational workload across multiple processors or computing nodes, significant speedup can be achieved. Parallel FFT implementations enable better scalability, reduced execution time, and improved utilization of modern multi-core and distributed computing architectures. This makes it feasible to apply FFT to very large datasets in high-performance computing (HPC) environments.

## 2 Methodology

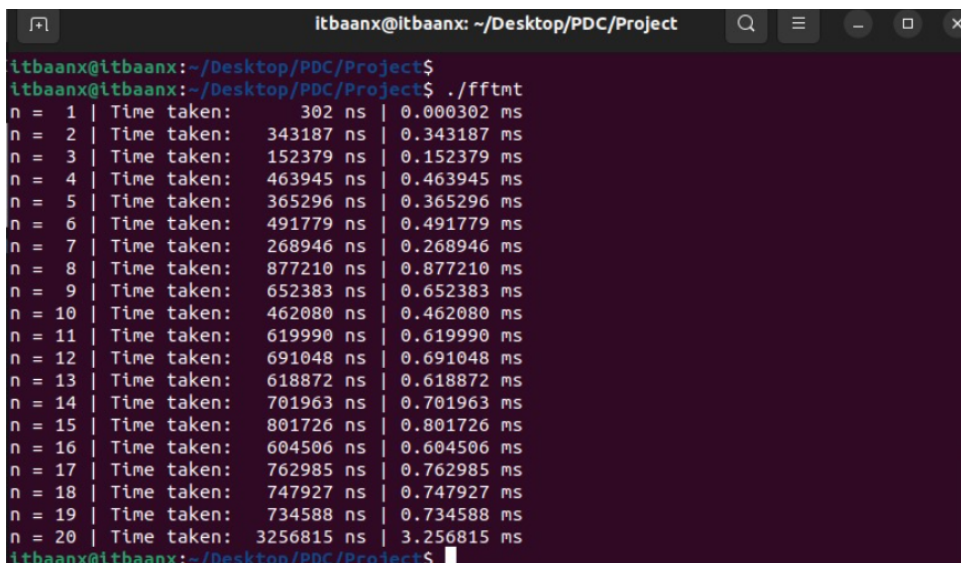
### 2.1 Pthread-based FFT (Multithreading)

The pthread-based FFT implementation uses manual multithreading to speed up the divide-and-conquer steps of the FFT algorithm. The input signal is an array of complex numbers, which is divided into two arrays: one containing elements at even indices and the other containing elements at odd indices. This separation is the core of the recursive FFT algorithm.

To parallelize this process, the program uses the POSIX threads (pthreads) library. It checks if the number of active threads is below a defined maximum (MAX\_THREADS). If so, it spawns two threads to compute the FFT of the even and odd parts concurrently. This is done using `pthread_create`, and after the threads finish their work, the main thread waits for them using `pthread_join`.

To ensure that no more than the maximum number of threads are created, a global thread counter is maintained, protected by a `pthread_mutex`. This ensures that multiple threads don't increment or decrement the counter simultaneously, avoiding race conditions.

Once the recursive calls return, the results are merged using the butterfly computation, which combines the even and odd results back into the original array. Temporary memory used for even and odd arrays is then freed. This process repeats recursively. The time for the FFT execution is recorded using `clock_gettime` to analyze the performance for different input sizes.



```

itbaanx@itbaanx: ~/Desktop/PDC/Project
itbaanx@itbaanx:~/Desktop/PDC/Project$ ./ffmt
n = 1 | Time taken: 302 ns | 0.000302 ms
n = 2 | Time taken: 343187 ns | 0.343187 ms
n = 3 | Time taken: 152379 ns | 0.152379 ms
n = 4 | Time taken: 463945 ns | 0.463945 ms
n = 5 | Time taken: 365296 ns | 0.365296 ms
n = 6 | Time taken: 491779 ns | 0.491779 ms
n = 7 | Time taken: 268946 ns | 0.268946 ms
n = 8 | Time taken: 877210 ns | 0.877210 ms
n = 9 | Time taken: 652383 ns | 0.652383 ms
n = 10 | Time taken: 462080 ns | 0.462080 ms
n = 11 | Time taken: 619990 ns | 0.619990 ms
n = 12 | Time taken: 691048 ns | 0.691048 ms
n = 13 | Time taken: 618872 ns | 0.618872 ms
n = 14 | Time taken: 701963 ns | 0.701963 ms
n = 15 | Time taken: 801726 ns | 0.801726 ms
n = 16 | Time taken: 604506 ns | 0.604506 ms
n = 17 | Time taken: 762985 ns | 0.762985 ms
n = 18 | Time taken: 747927 ns | 0.747927 ms
n = 19 | Time taken: 734588 ns | 0.734588 ms
n = 20 | Time taken: 3256815 ns | 3.256815 ms
itbaanx@itbaanx:~/Desktop/PDC/Project$

```

Figure 1: Output of multithreading

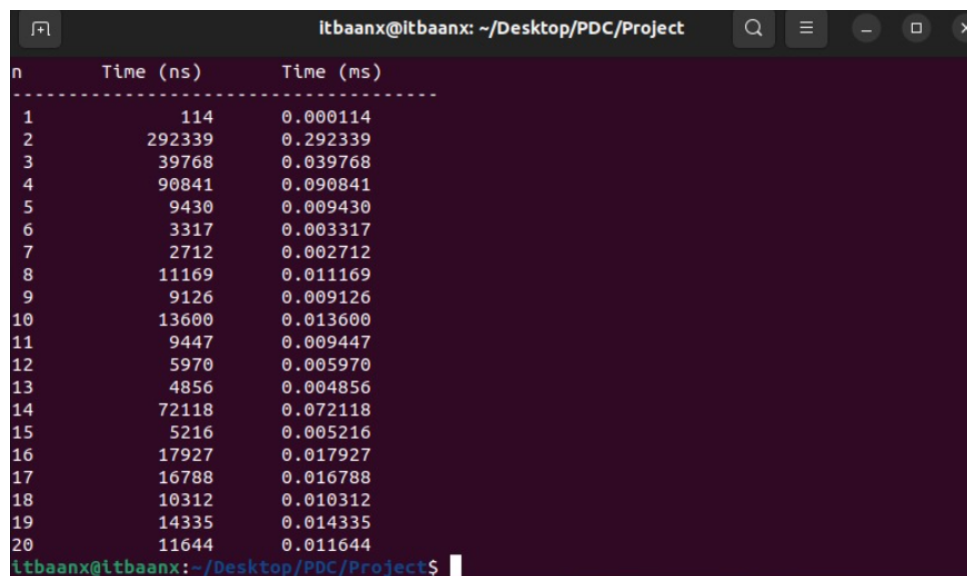
## 2.2 OpenMP-based FFT

The OpenMP version simplifies parallelism by leveraging compiler directives. The FFT algorithm remains recursive and follows the same divide-and-conquer strategy: splitting the input array into even and odd indexed elements.

Parallelism is introduced through the `#pragma omp parallel sections` directive, which tells the compiler to run the two recursive FFT calls in parallel, each in its own thread. This makes the code much simpler and easier to maintain compared to the pthread version, as there's no need to manually manage threads, joins, or mutexes.

The OpenMP runtime automatically manages the thread pool and decides how to schedule the parallel sections. This leads to more efficient and cleaner execution, especially on systems with multiple cores. The butterfly combination step and memory deallocation are handled the same way as in the pthread version.

The time taken for the FFT computation is measured using `clock_gettime`, and this performance is logged for various input sizes. Since OpenMP handles most of the parallelization complexity, this approach provides good scalability and performance with minimal developer effort.



n	Time (ns)	Time (ms)
1	114	0.000114
2	292339	0.292339
3	39768	0.039768
4	90841	0.090841
5	9430	0.009430
6	3317	0.003317
7	2712	0.002712
8	11169	0.011169
9	9126	0.009126
10	13600	0.013600
11	9447	0.009447
12	5970	0.005970
13	4856	0.004856
14	72118	0.072118
15	5216	0.005216
16	17927	0.017927
17	16788	0.016788
18	10312	0.010312
19	14335	0.014335
20	11644	0.011644

Figure 2: Output of Openmp

## 2.3 Cuda

This CUDA program performs the Fast Fourier Transform (FFT) on complex input data using the power of GPU parallelism to accelerate the computation. The FFT is a widely used algorithm for converting data from the time domain to the frequency domain. The program begins by ensuring that the input size is a power of 2, as this is optimal for the Cooley-Tukey FFT algorithm. If the input size is not already a power of 2, the array is padded with zeros.

Next, the `bit_reverse` kernel reorders the elements of the array based on bit-reversed indices. This step is crucial because the FFT algorithm requires data to be arranged in a specific order. The bit-reversal logic is implemented using parallel threads, allowing this step to be executed efficiently on the GPU.

After the reordering, the core computation takes place in the `fft_kernel`. This kernel applies the Cooley-Tukey algorithm in multiple stages. In each stage, it computes complex multiplications using exponential functions (known as twiddle factors) and combines elements using addition and subtraction. The number of stages is  $\log_2(N)$ , and with each stage, the size of the subarrays being processed doubles. All these operations are performed in parallel by multiple GPU threads, greatly speeding up the process.

To test the performance of the implementation, the program runs the FFT for input sizes from 1 to 20 (each adjusted to the next power of 2) and uses CUDA events to measure the time taken on the GPU. This test was conducted on Kaggle, which provides access to powerful GPUs in a cloud-based environment, making it a suitable platform for running CUDA programs. Finally, the execution time for each input size is printed, demonstrating how the algorithm scales with increasing input.

```
n = 1 | Time taken: 0.794080 ms
n = 2 | Time taken: 0.348608 ms
n = 3 | Time taken: 0.330912 ms
n = 4 | Time taken: 0.335968 ms
n = 5 | Time taken: 0.332640 ms
n = 6 | Time taken: 0.322752 ms
n = 7 | Time taken: 0.335264 ms
n = 8 | Time taken: 0.352288 ms
n = 9 | Time taken: 0.423648 ms
n = 10 | Time taken: 0.370080 ms
n = 11 | Time taken: 0.394112 ms
n = 12 | Time taken: 0.362304 ms
n = 13 | Time taken: 0.371360 ms
n = 14 | Time taken: 0.370528 ms
n = 15 | Time taken: 0.366400 ms
n = 16 | Time taken: 0.375328 ms
n = 17 | Time taken: 0.435168 ms
n = 18 | Time taken: 0.464544 ms
n = 19 | Time taken: 0.432768 ms
n = 20 | Time taken: 0.444128 ms
```

Figure 3: Output of Cuda

## 2.4 Comparison: Pthreads vs OpenMP vs Cuda

The pthread-based FFT gives the programmer low-level control over thread creation, synchronization, and termination. This can be powerful but comes at the cost of increased complexity and susceptibility to bugs such as race conditions and deadlocks. Moreover, pthreads can incur overhead from frequent thread creation and mutex operations, and the programmer is solely responsible for managing recursion depth and thread concurrency.

In contrast, OpenMP abstracts away much of this complexity using compiler directives. It simplifies thread management, enables automatic load balancing, and promotes code readability through constructs like `pragma omp parallel sections`. OpenMP achieves good performance with minimal code and is generally easier to debug and maintain than pthreads, making it suitable for scalable, real-world applications.

CUDA, on the other hand, shifts the computation to the GPU, which offers massive parallelism with thousands of threads. This allows FFT to be performed on a much larger scale and often at significantly higher speed—especially for large datasets. However, CUDA introduces a different kind of complexity: the programmer must handle device memory management, kernel launches, and GPU-specific optimizations. While it demands understanding of GPU architecture, the performance gains can be substantial for data-parallel problems like FFT.

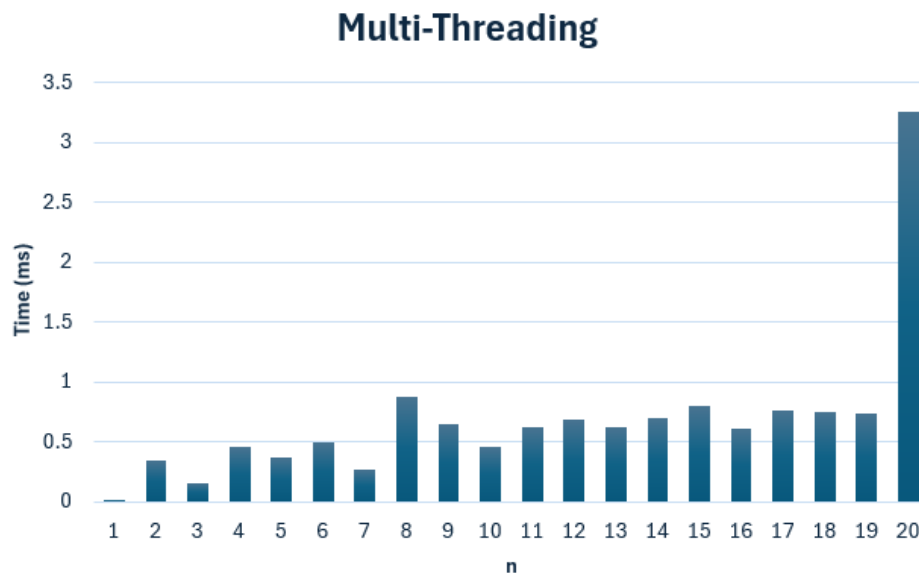


Figure 4: Multithreading

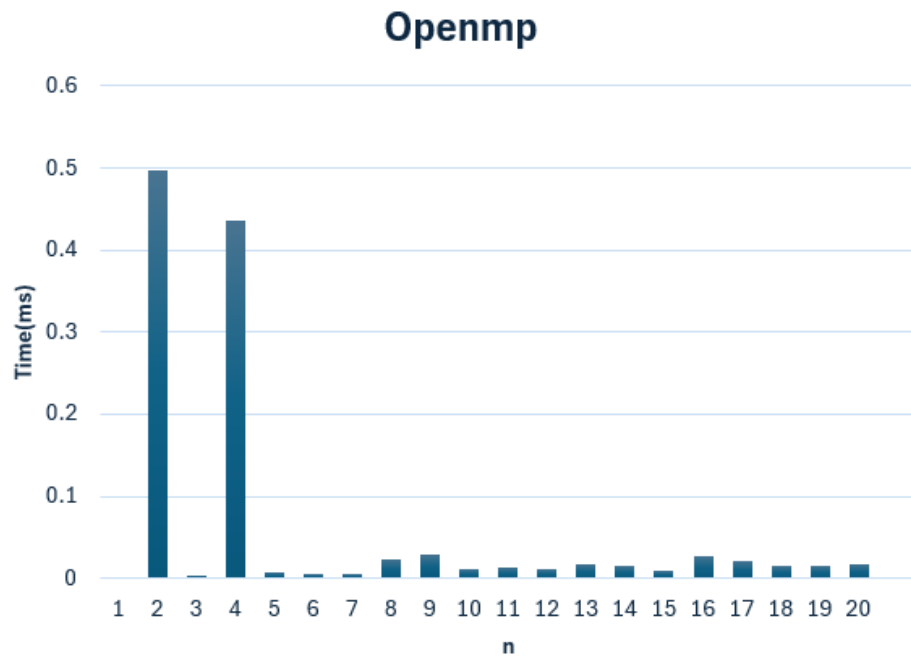


Figure 5: Openmp

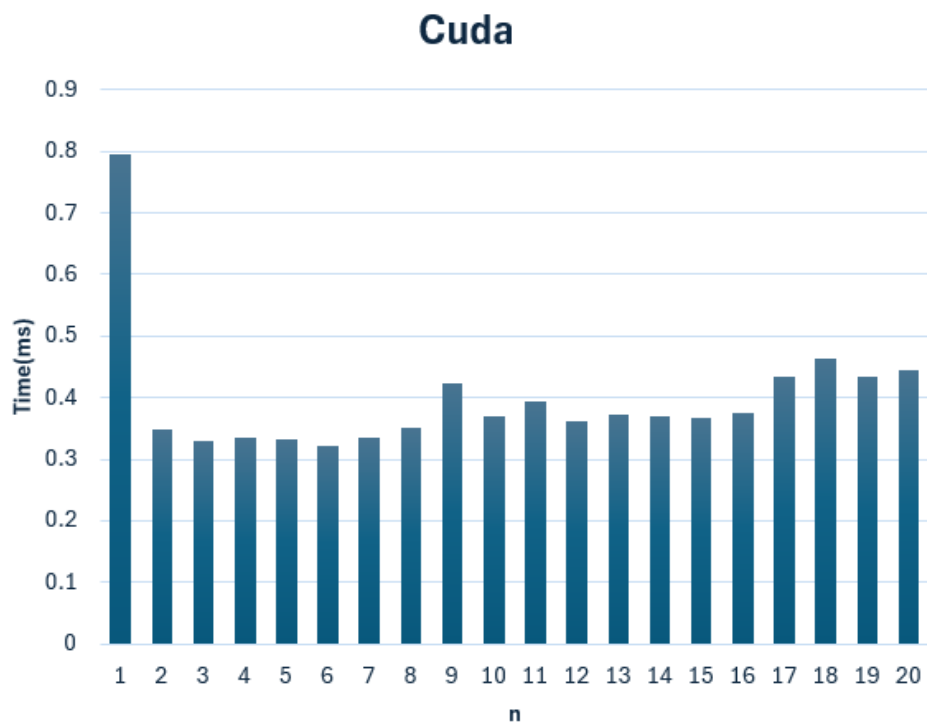


Figure 6: Cuda

## 3 Challenges and Future Work

### 3.1 Challenges and Limitations

One noticeable challenge in both implementations is the significant execution time for larger input sizes. Despite using multithreading or OpenMP, the recursive nature of the FFT leads to increasingly deep function calls and high memory allocation for even and odd arrays at each level. In the pthread version, thread creation and synchronization overhead further slows down performance, especially when too many threads are spawned. Additionally, there's no load balancing — certain branches of recursion may finish early, leaving other threads idle. In both versions, memory usage becomes inefficient due to repeated dynamic allocations, and the lack of in-place computation results in unnecessary overhead. Another limitation is that neither version includes error checking, input size padding to the nearest power of two (which FFTs often require), or optimizations for special cases like real-input-only FFTs.

### 3.2 Future Work and Improvements

To improve performance and scalability, future work can include implementing an iterative in-place FFT version to reduce recursion depth and memory overhead. Memory pooling or reusing buffers can reduce frequent allocations and deallocations. For the pthread version, using a thread pool instead of creating and destroying threads on each recursive call would greatly improve efficiency. In both versions, input size validation and padding to the next power of two can make the FFT more robust. Additionally, hybrid parallelism (such as combining OpenMP with SIMD instructions like AVX) could be explored to better utilize CPU resources. Profiling tools like gprof or perf can also be used to identify bottlenecks and guide further optimization. Lastly, integrating support for real-only FFTs or multidimensional FFTs (e.g., 2D/3D for image/audio processing) would extend the applicability of the code.

## 4 Conclusion

The Fast Fourier Transform (FFT) is a cornerstone algorithm in signal processing, enabling efficient transformation from time to frequency domains. While inherently efficient, its performance on large datasets can be limited on single-threaded systems. This report explored the motivation for parallelizing FFT and presented two implementations: one using pthreads for manual multithreading and another leveraging OpenMP for higher-level parallelism. The pthread-based approach provided granular control over thread behavior, while OpenMP offered a cleaner and more maintainable alternative with automatic thread management. A comparative analysis highlighted OpenMP's advantages in scalability and simplicity. However, both implementations face challenges such as recursive overhead, inefficient memory usage, thread synchronization issues, and lack of input validation. To address these, future work should focus on iterative in-place FFT, memory pooling, thread pooling for pthreads, and hybrid parallelism using SIMD. Enhancing robustness through input padding and supporting real-only or multidimensional FFTs can further extend applicability. Overall, parallel FFT remains a vital topic in high-performance computing, and careful optimization can significantly enhance its effectiveness in modern applications.



## 5 CODES

### 5.1 Multi threading

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <math.h>
4  #include <complex.h>
5  #include <pthread.h>
6  #include <time.h>
7
8  #define PI 3.14159265358979323846
9  #define MAX_THREADS 4
10
11 int thread_count = 0;
12 pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
13
14 typedef struct {
15     complex double *x;
16     int n;
17 } FFTArgs;
18
19 void *FFT_thread(void *args);
20
21 void FFT(complex double *x, int n) {
22     if (n <= 1) return;
23
24     complex double *even = malloc(n/2 * sizeof(complex double));
25     complex double *odd = malloc(n/2 * sizeof(complex double));
26
27     for (int i = 0; i < n/2; i++) {
28         even[i] = x[i*2];
29         odd[i] = x[i*2 + 1];
30     }
31
32     pthread_t thread1, thread2;
33     FFTArgs args1 = {even, n/2};
34     FFTArgs args2 = {odd, n/2};
35
36     int spawn_threads = 0;
37
38     pthread_mutex_lock(&mutex);
39     if (thread_count < MAX_THREADS) {
40         thread_count++;
41         spawn_threads = 1;
42     }
43     pthread_mutex_unlock(&mutex);
44
45     if (spawn_threads) {
46         pthread_create(&thread1, NULL, FFT_thread, &args1);
47         pthread_create(&thread2, NULL, FFT_thread, &args2);
48         pthread_join(thread1, NULL);
49         pthread_join(thread2, NULL);
50
51         pthread_mutex_lock(&mutex);
52         thread_count--;
53         pthread_mutex_unlock(&mutex);
54     } else {
55         FFT(even, n/2);
56         FFT(odd, n/2);
57     }
58
59     for (int k = 0; k < n/2; k++) {
60         complex double t = cexp(-2.0 * I * PI * k / n) * odd[k];
61         x[k] = even[k] + t;
62         x[k + n/2] = even[k] - t;
63     }
64
65     free(even);
66     free(odd);
67 }
68
69 void *FFT_thread(void *args) {

```

```

70     FFTArgs *fftArgs = (FFTArgs *)args;
71     FFT(fftArgs->x, fftArgs->n);
72     return NULL;
73 }
74
75 int main() {
76     for (int n = 1; n <= 20; n++) {
77         complex double *x = malloc(n * sizeof(complex double));
78
79         // Generate random input
80         for (int i = 0; i < n; i++) {
81             double real = rand() % 10; // 0 to 9
82             double imag = rand() % 10;
83             x[i] = real + imag * I;
84         }
85
86         struct timespec start, end;
87         clock_gettime(CLOCK_MONOTONIC, &start);
88
89         FFT(x, n);
90
91         clock_gettime(CLOCK_MONOTONIC, &end);
92         long elapsed_ns = (end.tv_sec - start.tv_sec) * 1e9 + (end.tv_nsec - start.
tv_nsec);
93         double elapsed_ms = elapsed_ns / 1e6;
94
95         printf("n = %2d | Time taken: %8ld ns | %.6f ms\n", n, elapsed_ns, elapsed_ms);
96
97         free(x);
98     }
99
100     return 0;
101 }

```

Listing 1: Multi-threading C++

## 5.2 OpenMP

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <math.h>
4  #include <complex.h>
5  #include <omp.h>
6  #include <time.h>
7
8  #define PI 3.14159265358979323846
9
10 void FFT(complex double *x, int n) {
11     if (n <= 1) return;
12
13     complex double *even = malloc(n/2 * sizeof(complex double));
14     complex double *odd = malloc(n/2 * sizeof(complex double));
15
16     for (int i = 0; i < n/2; i++) {
17         even[i] = x[i*2];
18         odd[i] = x[i*2 + 1];
19     }
20
21     // Parallel sections for FFT calls on even and odd
22     #pragma omp parallel sections
23     {
24         #pragma omp section
25         FFT(even, n/2);
26
27         #pragma omp section
28         FFT(odd, n/2);
29     }
30
31     for (int k = 0; k < n/2; k++) {
32         complex double t = cexp(-2.0 * I * PI * k / n) * odd[k];
33         x[k] = even[k] + t;
34         x[k + n/2] = even[k] - t;

```

```

35     }
36
37     free(even);
38     free(odd);
39 }
40
41 int main() {
42     srand((unsigned int)time(NULL));
43
44     printf("n\tTime (ns)\tTime (ms)\n");
45     printf("-----\n");
46
47     for (int n = 1; n <= 20; n++) {
48         complex double *x = malloc(n * sizeof(complex double));
49
50         // Fill with random complex numbers
51         for (int i = 0; i < n; i++) {
52             double real = rand() % 10 + 1;
53             double imag = rand() % 10 + 1;
54             x[i] = real + imag * I;
55         }
56
57         struct timespec start, end;
58         clock_gettime(CLOCK_MONOTONIC, &start);
59
60         FFT(x, n);
61
62         clock_gettime(CLOCK_MONOTONIC, &end);
63
64         long elapsed_ns = (end.tv_sec - start.tv_sec) * 1e9 +
65                         (end.tv_nsec - start.tv_nsec);
66         double elapsed_ms = elapsed_ns / 1e6;
67
68         printf("%2d\t%10ld\t%.6f\n", n, elapsed_ns, elapsed_ms);
69         free(x);
70     }
71
72     return 0;
73 }

```

Listing 2: OpenMP C++

### 5.3 CUDA

```

1  %%writefile cuda_fft.cu
2  #include <cuda.h>
3  #include <cuComplex.h>
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <math.h>
7
8  #define PI 3.14159265358979323846
9
10 __device__ cuDoubleComplex complex_exp(double theta) {
11     return make_cuDoubleComplex(cos(theta), sin(theta));
12 }
13
14 __global__ void fft_kernel(cuDoubleComplex *X, int N, int step) {
15     int tid = blockIdx.x * blockDim.x + threadIdx.x;
16     int i = tid * step * 2;
17     if (i + step < N) {
18         for (int j = 0; j < step; j++) {
19             cuDoubleComplex t = cuCmul(complex_exp(-2.0 * PI * j / (2.0 * step)), X[i +
20             j + step]);
21             cuDoubleComplex u = X[i + j];
22             X[i + j] = cuCadd(u, t);
23             X[i + j + step] = cuCsub(u, t);
24         }
25     }
26
27 __global__ void bit_reverse(cuDoubleComplex *X, int N, int logN) {

```

```

28     int tid = blockIdx.x * blockDim.x + threadIdx.x;
29     if (tid >= N) return;
30     unsigned int rev = 0;
31     unsigned int x = tid;
32     for (int i = 0; i < logN; i++) {
33         rev = (rev << 1) | (x & 1);
34         x >>= 1;
35     }
36     if (rev > tid) {
37         cuDoubleComplex temp = X[tid];
38         X[tid] = X[rev];
39         X[rev] = temp;
40     }
41 }
42
43 void cuda_fft(cuDoubleComplex *h_X, int N) {
44     cuDoubleComplex *d_X;
45     cudaMalloc(&d_X, sizeof(cuDoubleComplex) * N);
46     cudaMemcpy(d_X, h_X, sizeof(cuDoubleComplex) * N, cudaMemcpyHostToDevice);
47     int logN = log2(N);
48     int blockSize = 256;
49     int numBlocks = (N + blockSize - 1) / blockSize;
50     bit_reverse<<<numBlocks, blockSize>>>(d_X, N, logN);
51     cudaDeviceSynchronize();
52     for (int step = 1; step < N; step *= 2) {
53         int numThreads = N / (2 * step);
54         int blocks = (numThreads + blockSize - 1) / blockSize;
55         fft_kernel<<<blocks, blockSize>>>(d_X, N, step);
56         cudaDeviceSynchronize();
57     }
58     cudaMemcpy(h_X, d_X, sizeof(cuDoubleComplex) * N, cudaMemcpyDeviceToHost);
59     cudaFree(d_X);
60 }
61
62 int next_power_of_2(int n) {
63     int p = 1;
64     while (p < n) p <<= 1;
65     return p;
66 }
67
68 int main() {
69     for (int n = 1; n <= 20; n++) {
70         int N = next_power_of_2(n);
71         cuDoubleComplex *x = (cuDoubleComplex *)malloc(N * sizeof(cuDoubleComplex));
72         for (int i = 0; i < n; i++) {
73             double real = rand() % 10;
74             double imag = rand() % 10;
75             x[i] = make_cuDoubleComplex(real, imag);
76         }
77         for (int i = n; i < N; i++) {
78             x[i] = make_cuDoubleComplex(0, 0);
79         }
80
81         cudaEvent_t start, stop;
82         cudaEventCreate(&start);
83         cudaEventCreate(&stop);
84         cudaEventRecord(start);
85         cuda_fft(x, N);
86         cudaEventRecord(stop);
87         cudaEventSynchronize(stop);
88         float ms = 0;
89         cudaEventElapsedTime(&ms, start, stop);
90         printf("n = %2d | Time taken: %.6f ms\n", n, ms);
91         free(x);
92     }
93     return 0;
94 }

```

Listing 3: Cuda C++