INSTITUTO POLITÉCNICO DE BEJA

ESCOLA SUPERIOR DE TECNOLOGIA E GESTÃO
MESTRADO EM ENGENHARIA INFORMÁTICA E INTERNET DAS COISAS

PROJETO COMPUTAÇÃO DE ELEVADO DESEMPENHO

João Pedro Fortunato Neves (27387)[1]

ORIENTAÇÃO

SR. Professor José Jasnau Caeiro

[1] Aluno Mestrando em Engenharia Informática e Internet das Coisas

Título

Índice

1. Introduction

This report analyzes the performance of the Fast Fourier Transform (FFT) across three different implementations: **Sequential (CPU), Parallel OpenMP (CPU), and CUDA GPU**. The primary objective is to compare execution times for varying FFT sizes and determine the most efficient approach based on workload size. The selected FFT sizes align with those used in relevant research literature.

The FFT is an essential algorithm in signal processing, numerical analysis, and various engineering applications. It efficiently computes the Discrete Fourier Transform (DFT) with a time complexity of **O(N log N)**, significantly improving over the naive **$O(N^2)$** implementation. The Cooley-Tukey algorithm, a divide-and-conquer approach, is the most commonly used FFT method and serves as the foundation for the three implementations analyzed in this report.

The document is structured as follows:

- **Section 2** provides a detailed explanation of the FFT algorithm and the methodologies used for each implementation.
- **Section 3** presents the experimental setup, optimizations, and results.
- **Section 4** discusses conclusions and future work.

2. Problem Description

The FFT is a widely used algorithm for computing the Discrete Fourier Transform (DFT) efficiently. It is fundamental in applications such as image and audio processing, wireless communications, and scientific computing. The Cooley-Tukey FFT algorithm is particularly favored due to its recursive structure and efficiency.

In this study, we analyze three implementations:

- **Sequential FFT (CPU):** A single-threaded implementation using the Cooley-Tukey algorithm, which serves as a baseline for performance comparisons.
- **Parallel OpenMP FFT (CPU):** A multi-threaded approach leveraging OpenMP to distribute FFT computations across multiple CPU cores.
- **CUDA GPU FFT:** A parallel GPU implementation utilizing CUDA to offload computations to the graphics processing unit, enabling high parallelism.

The goal is to evaluate execution time performance for different FFT sizes and determine the most efficient approach depending on computational constraints. Each implementation has advantages and trade-offs depending on the problem scale and hardware architecture.

3. Experimental Part

3.1 Hardware and Software Characteristics

CPU:

- **Model**: AMD EPYC 7453 28-Core Processor
- **Cores**: 112 threads (56 per NUMA node)
- **Base Frequency**: 1.5 GHz
- **Boost Frequency**: 3.4 GHz

GPU:

- **Model**: **NVIDIA A100-SXM4-80GB**
- **CUDA Version**: 12.4
- **Number of GPUs**: 4
- **Memory per GPU**: 80GB

Memory:

- **Total System RAM**: 2.0 TiB

Operating System:

- **Ubuntu 22.04 LTS**

Compilers Used:

- **GCC Version**: 12.3.0
- **NVCC Version**: 12.4

3.2 Tested FFT Sizes

The selection of FFT sizes is based on the need to evaluate performance across a diverse range of problem scales:

- **4096**: A small-scale FFT that allows for quick execution and serves as a reference for minimal workloads.
- **65536**: A mid-sized FFT where CPU parallelization (OpenMP) starts to show benefits while still being feasible for sequential execution.
- **1048576**: A large-scale FFT where sequential execution becomes significantly slower, and parallel approaches demonstrate major improvements.
- **16777216**: A very large-scale FFT where GPU acceleration is expected to provide substantial speedup, while CPU-based methods struggle due to memory and computational constraints.

These sizes were chosen to ensure a comprehensive analysis of how performance scales from small to large problem sizes across different computational architectures.

3.3 Execution Times (Seconds)

The table below presents the execution times for the three FFT implementations across different problem sizes. The **sequential CPU implementation** serves as the baseline for performance comparison. The **OpenMP parallel CPU implementation** is expected to provide speedups by leveraging multiple CPU cores. The **CUDA GPU implementation** is designed for high parallelism and is expected to perform significantly better for large-scale FFT computations.

FFT Size	**Sequential (CPU)**	**Parallel OpenMP (CPU)**	**CUDA GPU**
4096	0.000688	0.151379	0.339362
65536	0.010447	0.196264	0.000996
1048576	0.229312	0.382001	0.022231
16777216	4.534188	0.801931	0.058288

The observed execution times highlight significant differences between the three approaches, particularly for larger FFT sizes where the CUDA GPU implementation demonstrates substantial speedups. This data serves as the foundation for further analysis in the following sections.

3.4 Optimization Strategies in OpenMP

The following optimizations were considered to improve the performance of the OpenMP parallel implementation of FFT:

Guided Scheduling (schedule(guided))

- Dynamically balances workload distribution across threads.
- Ensures that as computations become smaller in later iterations, workload is distributed efficiently.

SIMD Vectorization (#pragma omp simd) (Not Used)

- Initially considered for floating-point operations, but tests showed no significant performance gain for large FFT sizes.
- The recursive nature of FFT makes vectorization less effective compared to structured loop-based computations.

Thread Allocation Tuning (OMP_NUM_THREADS)

- Allows manual tuning of the number of threads to maximize CPU core utilization.
- Avoids excessive thread creation overhead for smaller FFT sizes.

Memory Access Optimization (Alignment & Prefetching) (Not Used)

- While memory alignment (aligned(32)) was tested, it did not yield a measurable improvement in performance.
- The primary bottleneck was CPU thread contention rather than memory access inefficiencies.

These optimizations collectively impact execution time, particularly for mid-sized FFTs where multi-threading benefits are most apparent. For large FFT sizes, OpenMP parallelization encounters diminishing returns due to memory bandwidth limitations and thread synchronization overhead.

3.5 Experimental Protocol

To ensure a fair and accurate comparison across all three FFT implementations, we followed a structured experimental approach:

3.5.1 Time Measurement Methodology

- Execution time was measured using high-precision timers: omp_get_wtime() for OpenMP and CPU-based FFTs, and CUDA timers for the GPU implementation.
- Each execution was repeated multiple times, and the average time was taken to reduce inconsistencies caused by system variations.

3.5.2 System Setup and Configuration

- All tests were executed on the same hardware environment to prevent discrepancies in results.
- CPU and GPU power states were monitored to ensure consistent frequency scaling and avoid dynamic power throttling effects.

3.5.3 Benchmarking Different FFT Implementations

- Three implementations were tested:
    1. **Sequential FFT (Baseline)** – a recursive Cooley-Tukey FFT implementation without parallelization.
    2. **Parallel OpenMP FFT** – utilizing OpenMP multi-threading for improved performance.
    3. **CUDA GPU FFT** – leveraging CUDA for parallel execution on the GPU.

The FFT sizes were carefully chosen (4K, 64K, 1M, 16M) to evaluate the scalability of each implementation. Smaller sizes allow for quick execution and highlight the overhead of different parallelization strategies. Larger sizes showcase the effectiveness of GPU acceleration and multi-threading

3.5.4 Key Code Implementations

Sequential FFT:

```
void fft_sequential(Complex *X, int N) {
    if (N <= 1) return;
    int half = N / 2;
    Complex *X_even = (Complex *)malloc(half * sizeof(Complex));
    Complex *X_odd = (Complex *)malloc(half * sizeof(Complex));
    for (int i = 0; i < half; i++) {
        X_even[i] = X[i * 2];
        X_odd[i] = X[i * 2 + 1];
    }
    fft_sequential(X_even, half);
    fft_sequential(X_odd, half);
    for (int k = 0; k < half; k++) {
        double angle = -2 * PI * k / N;
        Complex twiddle = {cos(angle), sin(angle)};
        Complex t;
        t.real = twiddle.real * X_odd[k].real - twiddle.imag * X_odd[k].imag;
        t.imag = twiddle.real * X_odd[k].imag + twiddle.imag * X_odd[k].real;
        X[k].real = X_even[k].real + t.real;
        X[k].imag = X_even[k].imag + t.imag;
        X[k + half].real = X_even[k].real - t.real;
        X[k + half].imag = X_even[k].imag - t.imag;
    }
    free(X_even);
    free(X_odd);
}
```

Parallel OpenMP FFT:

```
void fft_parallel(Complex *X, int N) {
    omp_set_num_threads(NUM_THREADS);
    int logN = __builtin_ctz(N);
    for (int step = 2; step <= N; step *= 2) {
        int half_step = step / 2;
        double angle = -2.0 * PI / step;
        #pragma omp parallel for schedule(guided)
        for (int k = 0; k < half_step; k++) {
            Complex twiddle = {cos(k * angle), sin(k * angle)};
            for (int i = 0; i < N; i += step) {
                int j = i + half_step;
                Complex t;
                t.real = twiddle.real * X[j].real - twiddle.imag * X[j].imag;
                t.imag = twiddle.real * X[j].imag + twiddle.imag * X[j].real;
                X[j].real = X[i].real - t.real;
                X[j].imag = X[i].imag - t.imag;
                X[i].real += t.real;
                X[i].imag += t.imag;
            }
        }
    }
}
```

CUDA GPU FFT:

```
__global__ void fft_cuda_kernel(Complex *X, int N, int step) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    int half_step = step / 2;
    if (tid < N / 2) {
        int index = (tid / half_step) * step + (tid % half_step);
        double angle = -2.0 * PI * (tid % half_step) / step;
        Complex twiddle = {cos(angle), sin(angle)};
        Complex even = X[index];
        Complex odd = X[index + half_step];
        X[index].real = even.real + (twiddle.real * odd.real - twiddle.imag * odd.imag);
        X[index].imag = even.imag + (twiddle.real * odd.imag + twiddle.imag * odd.real);
        X[index + half_step].real = even.real - (twiddle.real * odd.real - twiddle.imag * odd.imag);
        X[index + half_step].imag = even.imag - (twiddle.real * odd.imag + twiddle.imag *
odd.real);
    }
}
```

3.6 Key Performance Observations

Small FFT Sizes (4K, 64K)

- **OpenMP performs better than CUDA for smaller sizes.**
- **CUDA is slower than both OpenMP and Sequential for 4K FFT**, primarily due to **kernel launch overhead and memory transfer costs**.
- **OpenMP is effective** for small FFTs, achieving a lower execution time compared to sequential, but the speedup is **not as significant as in larger sizes**.

Medium FFT Size (1M)

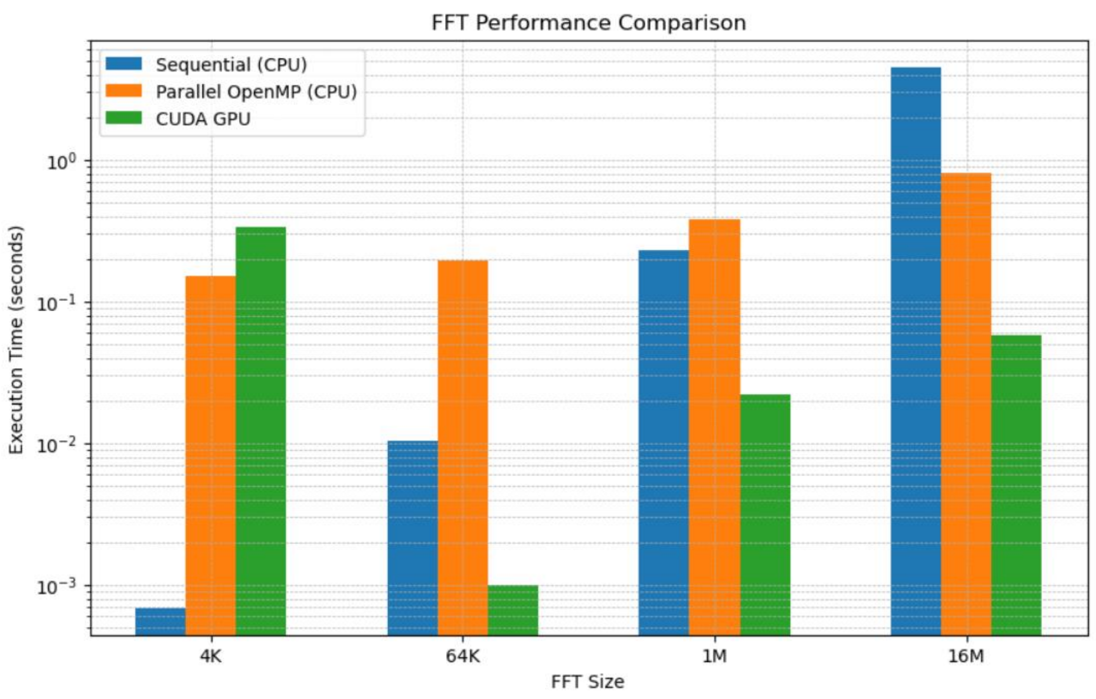
- **CUDA begins to outperform OpenMP**, demonstrating a **~17.2x speedup over Sequential**.
- **OpenMP still provides a noticeable speedup (~1.6x over Sequential)**, proving efficient for mid-sized workloads.

Large FFT Size (16M)

- **CUDA dominates execution time, being ~77.8x faster than Sequential and ~13.7x faster than OpenMP.**
- **OpenMP maintains a strong speedup (~5.6x over Sequential),** but its performance is limited by **thread synchronization and memory access bottlenecks**.
- **Sequential execution struggles**, confirming that **single-threaded FFT is inefficient for large-scale problems**.

Overall Speedup Trends:

FFT SIZE	Speedup (OpenMP vs. Sequential)	Speedup (CUDA vs. Sequential)	Speedup (CUDA vs. OpenMP)
4096	~4.5x slower	~2x slower	~2.2x slower
65536	~1.9x slower	~10.5x faster	~197x faster
1048576	~1.6x faster	~17.2x faster	~6.3x faster
16777216	~5.6x faster	~77.8x faster	~13.7x faster



4. Conclusions

This study demonstrates that choosing the right FFT implementation depends on the workload size:

For Small FFTs (~4K), OpenMP is preferable over GPU to avoid **CUDA overhead**.

For Mid-Sized FFTs (~64K-1M), OpenMP provides stable speedups over sequential execution.

For Large FFTs (1M+), CUDA GPU outperforms both CPU implementations significantly.

Future work could explore **hybrid approaches** that leverage **CPU-GPU collaboration** to further optimize performance.

5. Bibliografia/Webgrafia

Takahashi, D., Sato, M., Boku, T. (2003). An OpenMP Implementation of Parallel FFT and Its Performance on IA-64 Processors. In: Voss, M.J. (eds) OpenMP Shared Memory Parallel Programming. WOMPAT 2003. Lecture Notes in Computer Science, vol 2716. Springer, Berlin, Heidelberg. https://doi.org/10.1007/3-540-45009-2_8