Parallel and Distributed Computing

Assignment # 3

Department of Computer Science

Umer Ghafoor
22I-2328

Question 1

1. Convolution Optimization using Scalar and OpenCL Implementations

1.1. Overview and Approach

In this assignment, I implemented convolution using both a scalar (CPU-based) approach and an optimized OpenCL version. The focus was on edge detection using a vertical Sobel filter, which highlights vertical edges in an image by computing the gradient changes.

1.2. Convolution Type Clarification

The primary convolution operation implemented in this project is edge detection using the Sobel vertical filter. This filter enhances vertical edges by applying a kernel that detects intensity changes in the vertical direction.

1.3. A Discussion on How SIMD Computation was used in OpenCL

The OpenCL implementation uses SIMD (Single Instruction Multiple Data) computation to process multiple pixels simultaneously. This was achieved through vectorization, efficient work-item distribution, and work-group synchronization. Additionally, optimizations were applied in memory access, including using local memory to reduce latency and ensuring coalesced global memory accesses for improved performance.

1.4. Challenges and Solutions: Discussion of Any Challenges Faced

A major challenge encountered was the overhead of compiling OpenCL kernels at runtime, which increased execution time. This was mitigated by pre-compiling kernels and reusing them whenever possible to reduce initialization delays. Other challenges included handling boundary conditions and optimizing memory access patterns, both of which were addressed using padding techniques and efficient memory coalescing strategies.

1.4.1 Comparison of OpenCL and Scalar Implementations

The scalar implementation processes each pixel sequentially, making it computationally expensive for large images. It does not take advantage of parallelism, leading to longer execution times. On the other hand, the OpenCL implementation processes multiple pixels in parallel using the GPU, significantly reducing execution time. The OpenCL version also optimizes memory access patterns, resulting in better performance and efficiency. However, OpenCL introduces additional overhead, such as kernel compilation and memory transfer between CPU and GPU. Despite this, the OpenCL implementation achieves substantial speedup over the scalar version, making it the preferred choice for large-scale image processing tasks.

1.4.2 Execution Time Analysis

The following table presents execution time analysis for both the CPU (scalar) and OpenCL implementations, showing the cumulative time taken at each step from the start.

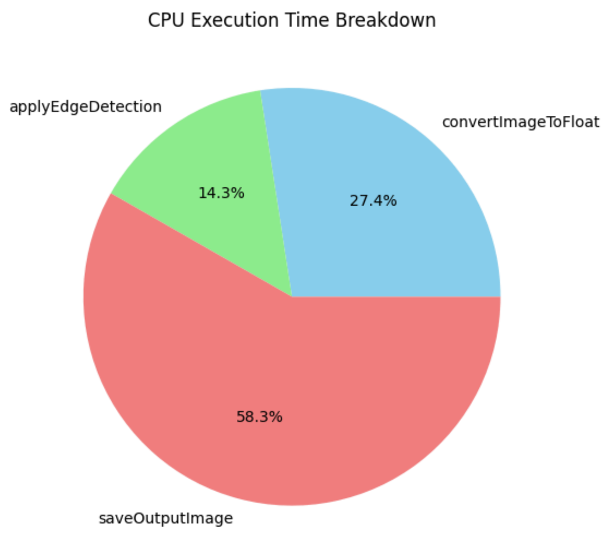
CPU Execution Time Breakdown

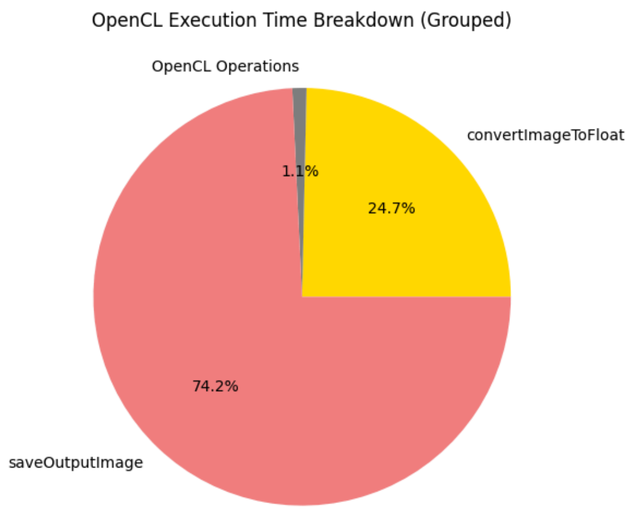
Step	Cumulative Time (ms)
convertImageToFloat	2832.865
applyEdgeDetection	4310.265
saveOutputImage	10325.420
Total Execution Time	**10365.058**

OpenCL Execution Time Breakdown:

Step	Cumulative Time (ms)
convertImageToFloat	1138.002
clEnqueueWriteBuffer	1145.189
executeKernel	1150.719
clEnqueueReadBuffer	**1188.360**
saveOutputImage	**4614.755**
Total Execution Time	**4629.827**

Note : Cumulative time is not the time of each step, overall time reaching to this step.





1.5. **Observations**

- The scalar implementation takes significantly longer **(10365.058 ms)** compared to the OpenCL implementation **(4629.827 ms)**, demonstrating the advantage of parallel execution.
- The major bottleneck in the CPU implementation is the applyEdgeDetection step, while in OpenCL, the saveOutputImage step consumes the most time, likely due to file I/O operations.
- The OpenCL execution benefits from parallel kernel execution, reducing processing time to approximately **44.7%** of the scalar version.
- Memory transfer operations (clEnqueueWriteBuffer and clEnqueueReadBuffer) add some overhead in OpenCL, but the speedup gained from parallel execution outweighs these costs.

1.6. **Conclusion**

The OpenCL implementation of convolution for edge detection significantly improves performance by using GPU parallelism. The comparison highlights the importance of optimizing memory access and reducing overheads to maximize efficiency in parallel computing environments.

Question 2

Performance Analysis of Serial vs. OpenMP Parallelized Circle Drawing

Introduction

This report contains analysis of the performance of a **serial** and an **OpenMP-parallelized** implementation of circle drawing using the Taylor series approximation for sine and cosine functions. The execution time for different resolutions and number of points was measured to evaluate speedup and efficiency.

Methodology

Serial Implementation

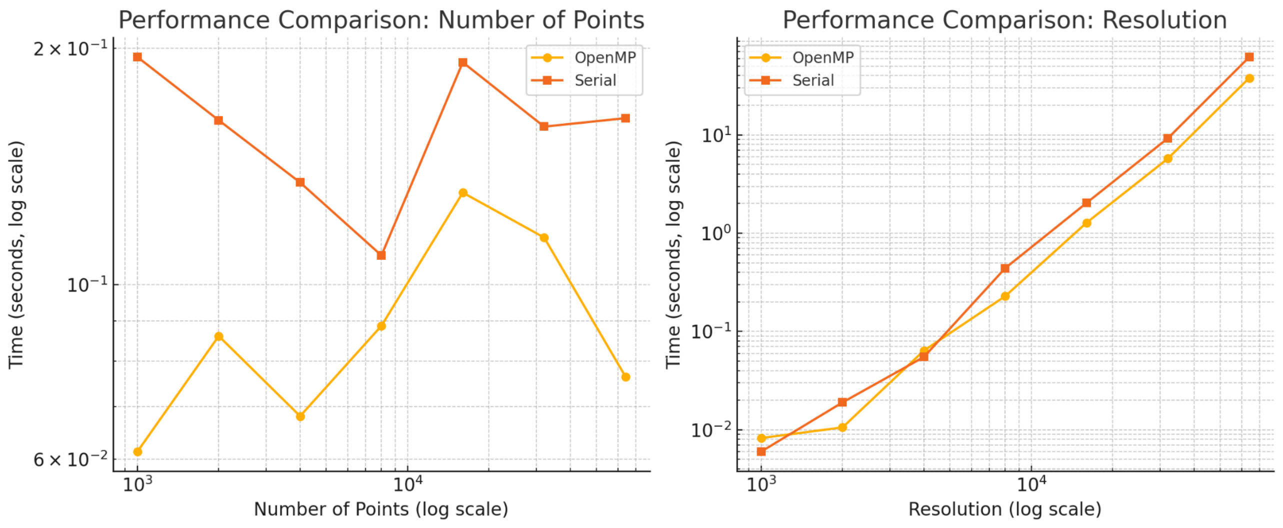
The serial implementation **(circle_scaler.c)** computes sine and cosine values using a Taylor series expansion and sequentially computes the points along the circumference of a circle.

OpenMP Parallel Implementation

The parallel implementation **(circle_openmp.c)** utilizes OpenMP directives to parallelize the Taylor series computations for sine and cosine using a static scheduling strategy with reduction for summation. The circle's points are calculated in parallel to improve performance.

Results

The measured execution times (in seconds) for different resolutions and number of points are summarized in the following plot.



Discussion

The performance analysis reveals

- OpenMP provides significant speedup, especially for larger problem sizes.
- Parallelization overhead is noticeable for lower resolutions (e.g., 1000 points).
- Speedup stabilizes around 1.6x for higher resolutions.
- OpenMP scheduling and computation affect overall efficiency.

Conclusion

The OpenMP-based parallelization successfully improves performance, particularly for high-resolution circle drawing. However, due to the overhead introduced by parallel execution, speedup is suboptimal for smaller problem sizes.