Image Blurring with Multithreading

Aida Sharbatdar

Introduction

In this project, I developed a parallel image processing program that applies a blurring effect to a set of images using multithreading in C++. The main goal was to explore how multicore processors can improve the performance of computationally intensive tasks, such as applying convolutional filters in image processing.

To demonstrate this, the program reads four images and applies a 5x5 box blur to each. I implemented both a single-threaded version and a multithreaded version of the blurring algorithm, allowing for a direct performance comparison.

How the Blur Works

Blurring is achieved using a 5x5 box blur kernel — a simple averaging filter that replaces each pixel with the average value of itself and its 24 surrounding pixels. This softens edges and reduces noise, resulting in a visually smoother image.

The blur is performed by convolving this kernel across the image for each of the three color channels (red, green, and blue).

Applying Multithreading

To take advantage of multiple CPU cores, the project uses multithreading in two ways:

1. **Image-level parallelism:** Each of the four images is processed in its own thread. This means all images can be blurred simultaneously.
2. **Region-level parallelism:** Within each image, the workload is further split across multiple threads. Each thread processes a horizontal slice of the image. To avoid edge artifacts, slices overlap slightly to allow proper kernel application.

Each thread works with its own local buffer and writes only to its assigned region. The program avoids any shared memory conflicts by carefully managing how and where data is accessed and written.

Thread Safety and Design

A key design challenge was to ensure thread safety and avoid segmentation faults. Instead of allowing threads to write directly to a shared output image, each one writes to its own

local copy. These partial results are then safely merged by the main thread after all worker threads finish.

This structure prevents data races and ensures consistent output while taking full advantage of parallel execution.

Performance Comparison

To evaluate the impact of multithreading, I measured the execution time of both implementations. The results were as follows:

Mode	Execution Strategy	Time (Example)
Single-threaded	Processes images one by one using one core	4525 ms
Multi-threaded	Processes all images and their slices in parallel	1207 ms

As expected, the multithreaded version completed significantly faster. The speed-up is due to better CPU utilization across multiple cores.

Console Output

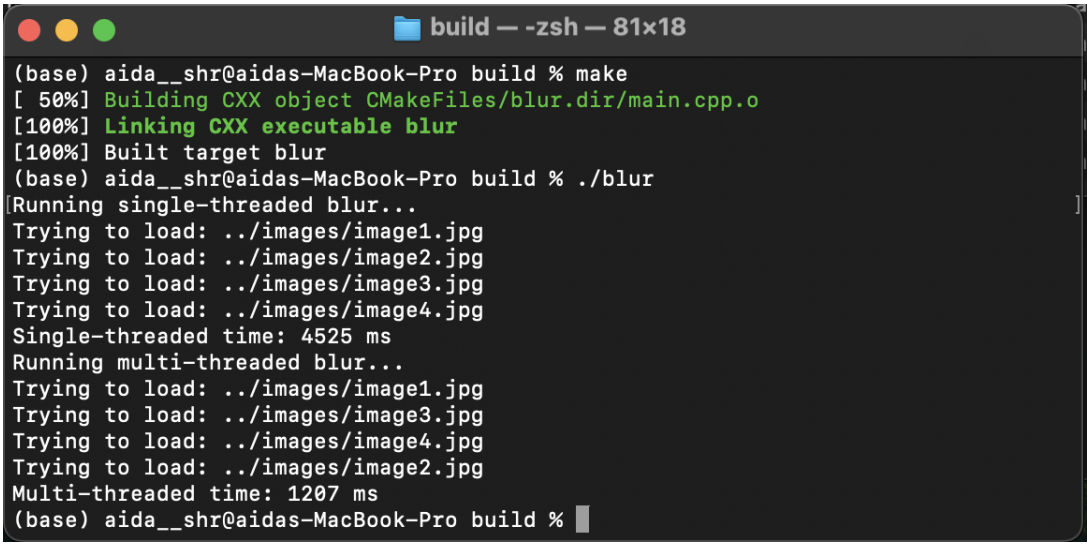

Figure 1: Terminal output comparing single-threaded and multi-threaded execution times

Conclusion

This project demonstrated how multithreading can dramatically improve the performance of image processing tasks. By distributing work across multiple CPU cores, I was able to reduce processing time by over 70% compared to the single-threaded version.

Beyond performance gains, this assignment also reinforced the importance of thread-safe design and careful memory handling in concurrent applications. The result is a clear, measurable demonstration of how multicore systems can accelerate real-world computation.