Image/Video Processing and Optimizations

Student: Sam Nowak
Date: Spring 2025

Overview

This project applies several key concepts—spatial and temporal locality, as well as pipelining—to image and video edge detection. I have implemented both a sequential Java version and an optimized CUDA version. We compare their runtimes to demonstrate how caching and pipelining can significantly improve performance.

1 Process

All implementations follow the same process:

1. Load the input as a `BufferedImage` and convert its pixels into an array.
2. Convert the image to grayscale.
3. Add a one-pixel border of padding around the grayscale image to simplify boundary handling.
4. Apply a 3×3 convolution kernel twice: once for horizontal edges and once for vertical edges.
5. Combine the two edge maps into a single output image.
6. For video processing, repeat this pipeline for each frame.

Figure 1: * Original Image

Figure 2: * Grayscale

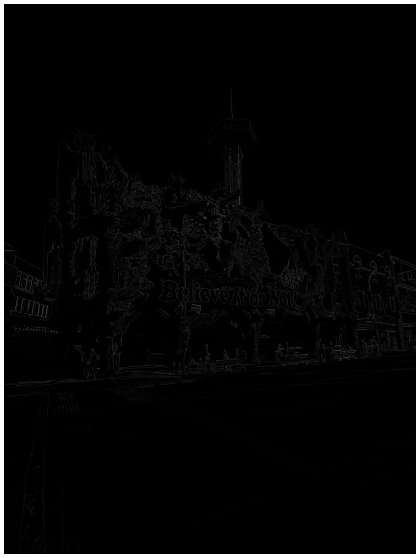

Figure 3: * Edge Output

Figure 4: Processing Pipeline

Sequential Java Version

In the sequential Java implementation, each step runs one after the other on the CPU. Each pixel is processed in turn, and we must complete one operation across the entire image before moving on to the next. This lack of parallelism and multiple full-image traversals lead to significant runtime overhead.

CUDA Optimized Version

Our CUDA implementation parallelizes each stage of the process:

- **Thread-level parallelism:** Launch one thread per pixel for grayscale conversion and edge detection, immediately speeding up the workload.
- **Tiling with shared memory:** Load each tile plus its one-pixel halo into shared memory to use spatial and temporal locality, reducing global-memory traffic.
- **Constant memory for kernels:** Store the 3×3 convolution kernels in CUDA constant memory for faster access.
- **Pipelining with CUDA streams and pinned memory:** Use multiple CUDA streams and page-locked host buffers to overlap H_2D/D_2H transfers with kernel execution, maximizing throughput. In this project I only used two.

Results

We ran both versions on the same test video, measuring end-to-end processing time over 10 trials:

- **Java (sequential) times (ms):** 24897, 24824, 24012, 23953, 24598, 24243, 25046, 25014, 24561, 25639
  Average: 24,678.7 ms
- **CUDA (optimized) times (ms):** 3796, 3771, 3878, 4217, 3485, 3761, 3890, 3955, 3680, 3741
  Average: 3,817.4 ms
- **Speedup:** $24{,}678.7/3{,}817.4 \approx 6.46\times$

Conclusion

These results confirm that leveraging spatial/temporal locality, parallel execution, and pipelining drastically improves performance. By reducing global-memory accesses, processing pixels in parallel, and overlapping transfers with computation, our CUDA version achieves a $6.5\times$ speedup over the regular sequential approach.