HPC Project Report

Accelerating KLT Algorithm

Version 3.0: Memory Optimization

M. Abdullah Siddiqui | 23I-0617

M. Daniyal | 23I-0579

Moiz Ansari | 23I-0523

Introduction

This version focuses on optimizing the GPU implementation through advanced CUDA features such as **constant memory**, **pinned memory**, and **shared memory utilization**. The goal is to minimize global memory latency, improve occupancy, and maximize throughput using efficient memory hierarchy utilization.

Implemented Optimizations

Constant Memory

Since constant memory is limited in size, it needs to be utilized effectively. For the parallelized code, the kernel filter was the perfect candidate for constant memory because:

1. It was read-only
2. It was a constant size memory (array of 71 floating point numbers)

It resulted in better memory access performance than global access.

Pinned Memory

In order to decrease memory communication time, pinned memory was used using the cudaMallocHost. In this way, data transferring time was reduced with the cost of cudaMallocHost API call time. But on observation, the API call time is nearly constant over large images and datasets with reduced data communication time. Without pinned memory, only the data communication time cumulatively becomes more than 92% which affects the GPU performance.

Shared Memory

1D shared memory tile was used including the halo region. The tile dimension was taken as 32x16 threads for maximum occupancy. Each thread loads its corresponding pixel. The border threads load additional pixels for neighboring regions. All data is loaded on shared memory from global memory. This eliminates the redundant memory access from global memory, which is slow, and reuses shared memory, which has 100x faster access than global memory.
The shared memory was utilized in only vertical convolution because in horizontal convolution, the memory access was coalesced, so it wasn't effective to use shared memory for that.

Occupancy

After all of our above optimizations, the occupancy of our kernel code is calculated using Nvidia Nsight Compute (NCU) which is 100%

Performance Results

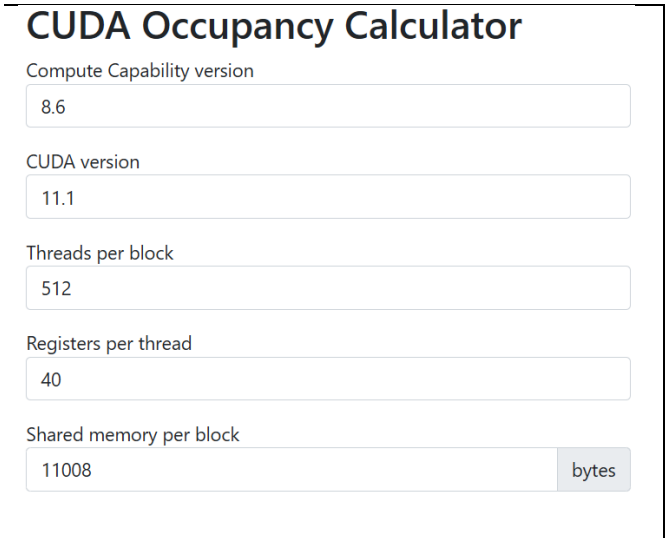
Speedup Factor:

After each kernel run, the speedup factor was calculated using Amdahl's law. For horizontal convolution it was nearly 1.46x while for the vertical convolution, it was 2.25x.

S = T_serial / T_parallel

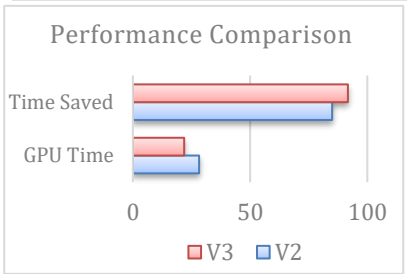
Speedup = 1 / ((1-p) + (p / S))

Overall Metrics:



GPU Occupancy Data

Active Threads per Multiprocessor	1536
Active Warps per Multiprocessor	48
Active Thread Blocks per Multiprocessor	3
Occupancy of each Multiprocessor	1



Detailed Metrics:

Image Size	CPU Time (ms)	GPU Time (ms)	Time Saved (ms)	Overall Speedup
320 × 240	113.58	21.88	91.70	2.81x
1920 × 1080	3713.60	215.77	3497.83	7.64x

The results are for 126 convolution calls. They show that as image size increases, GPU acceleration scales significantly better, demonstrating efficient utilization of parallel threads and memory hierarchy.

Conclusion

The implemented optimizations significantly improved performance, especially for high-resolution images. Shared memory and constant memory contributed the most to speedup, while pinned memory improved data transfer efficiency.

GitHub Private Repository Link

https://github.com/MuhammaDaniyal/Complex_Computing_Problem