HPC Project Report

Accelerating KLT Algorithm

Version Final

M. Abdullah Siddiqui | 23I-0617

M. Daniyal | 23I-0579

Moiz Ansari | 23I-0523

Table of Contents

Introduction

This final stage of our HPC project summarizes all optimizations applied to the KLT feature tracker.

1. We begin by selecting an appropriate profiler and executing the baseline code.
2. Profiling results are analyzed to understand performance bottlenecks and runtime distribution.
3. Using the profiling insights, we identify functions suitable for GPU acceleration.
4. We theoretically estimate the expected speedup using Amdahl's Law.
5. Based on these findings, we apply multiple optimization strategies: constant memory, pinned memory, shared memory, and finally OpenACC.

The upcoming sections explain each of these steps and optimizations in detail.

Code Profiling

The KLT project code was profiled using gprof commands. The output of the gprof was a raw text file which was difficult to understand. So, the output was visualized using a python script that uses graphViz and dot. The visualization made it easy to understand the highly intensive functions and their respective calls and number of calls.

High Compute Intensive Functions

The profiled output showed three major compute intensive functions performing Horizontal Convolution, Vertical Convolution and Interpolation. But on larger dataset, the interpolation was negligible and KLTSelectGoodFeatures started taking a small portion of the total runtime. Following were the details captured:

Operation	Image Size	% of runtime
_convolveImageVert	320 × 240	39.74
_convolveImageHoriz	320 × 240	35.90
_interpolate	320 × 240	14.10
_convolveImageVert	1920 × 1080	58.05
_convolveImageHoriz	1920 × 1080	27.07
_KLTSelectGoodFeatures	1920 × 1080	7.32

Together, these account for **89.74%** of the total execution time for the small image set and **92.44%** for the larger dataset. We selected horizontal and vertical convolution for parallelization for the following reasons:

1. Parallel running ability on the GPU
2. No dependency
3. High runtime, parallelizing will provide better optimization

Interpolation was ignored for the meanwhile because the function itself was an O(1) but it was called nearly 2 million times. So, it would not have given much difference on parallelization.

Theoretical Speedup

We parallelized two functions that are convolve Horizontal and Vertical as only these two were taking a considerable amount of time and were parallelizable. Interpolate was not considered because its major time was spent on calling rather than computations.

By Amdahl's law:

P = 0.3974 + 0.3590 = 0.7564

Speed up = 1 / (1 - 0.7564) = 4.105x

So, the theoretical max speed up that can be achieved is 4.1x.

Functions Parallelization

It was the time when we needed to include CUDA kernels with the KLT project code. A separate .cu file was made and all the cuda kernel definitions were included there. Another .h file was made to link the .cu file with the rest of the project. In this way, file structure was setup, and we could easily start writing cuda kernels afterwards.

Kernels

Inside the kernels, we used global indexing for accessing global data. The indices were 2 dimensional (i and j) since the kernel configuration was 2D. The indices were then used according to the pattern in which CPU code's array was accessed. In a nutshell, the left and right (in Horizontal Convolution) or top and bottom (in Vertical Convolution) convolution were simply done by threads while the middle convolution was also done by threads, but they contained a loop for adding up the neighboring elements.

Kernel Configuration

2D kernel configuration was used to align with the 2D nature of the image. Then initially 32 by 32 block dimension was used and row/32 by col/32 grid dimension was used. But after successfully running the kernel, the configuration was changed to 16 by 32 block dimension and row/16 and

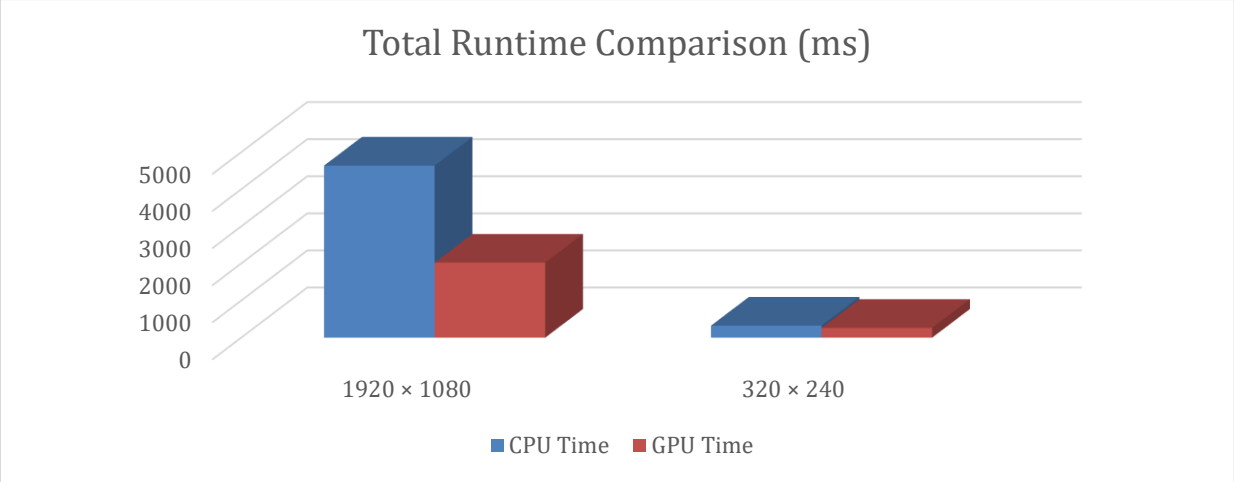
col/32 grid dimension for optimization reasons. The time was reduced, and occupancy was increased when run on the 16 by 32 configuration.

Achieved Speedup & Occupancy

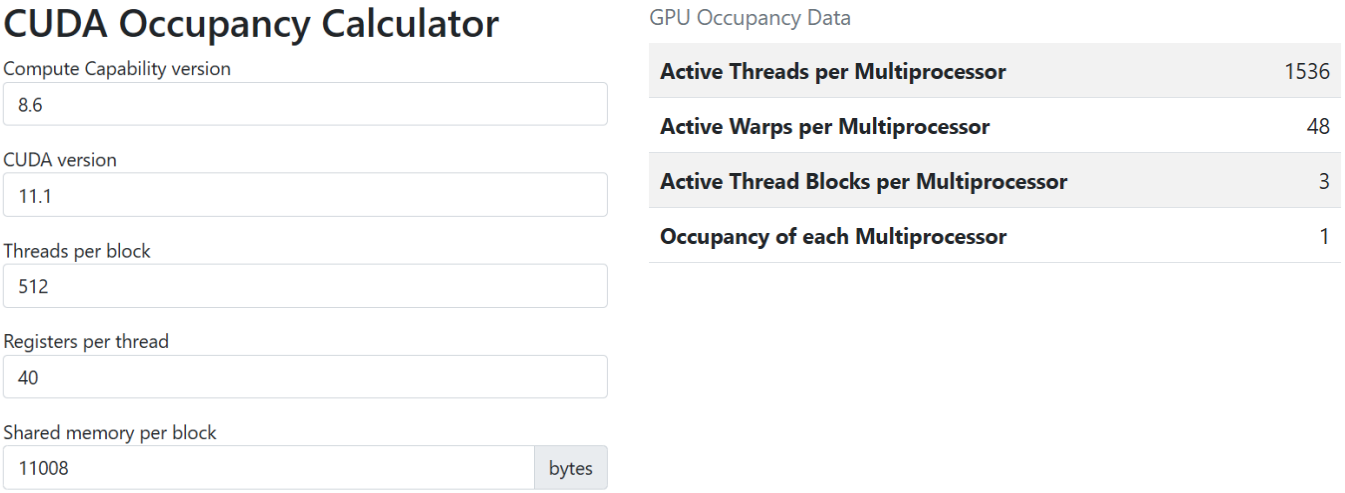
Speedup

Image Size	CPU Time (ms)	GPU Time (ms)	Time Saved (ms)	Speedup
320 × 240	~324.46	~263.24	~61.22	1.23x
1920 × 1080	~4639.02	~2022.81	~2616.21	2.29x

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.



Occupancy



The CUDA kernels achieved full 100% occupancy, with all 1536 threads per SM actively utilized. This indicates that our configuration efficiently leverages the hardware's available warps and

registers. However, this does not guarantee maximum performance, it only shows that the thread-level parallelism is maximized.

Optimizations

Data Communication Optimization

To reduce CPU–GPU transfer overhead, **pinned (page-locked) memory** was used on the host. This allowed faster and more predictable data transfers via DMA compared to standard pageable memory. By keeping intermediate image data on the GPU and transferring only necessary input/output buffers, communication time, often a major bottleneck, was significantly reduced.

Data Access Optimization

GPU memory accesses were optimized using **shared memory** and **constant memory**. Shared memory was employed to store per-block tiles of the image, enabling threads to reuse neighboring pixels without repeatedly reading from slow global memory. The convolution kernels were stored in constant memory, leveraging the GPU's broadcast cache for fast read-only access across all threads. These optimizations improved memory throughput, reduced latency, and allowed threads to perform convolution efficiently.

Speedup Achieved

Image Size	CPU Time (ms)	GPU Time (ms)	Time Saved (ms)	Speedup
320 × 240	~324.46	~231.29	~93.17	1.4x
1920 × 1080	~4639.02	~1834.07	~2804.95	2.53x

Smaller datasets have a larger memory overhead, so parallelization is more beneficial for larger datasets.

OpenACC Directives

Index Notation

For the high intensive functions, convolve horizontal and vertical, it was noted that the code is written with pointer access notation. So, it was converted to the index notation to make it easy for the ACC to understand the code for parallelization. Further the restrict keyword was used with the pointer for telling the compiler that the pointer is not aliased.

Data Construct & Clauses

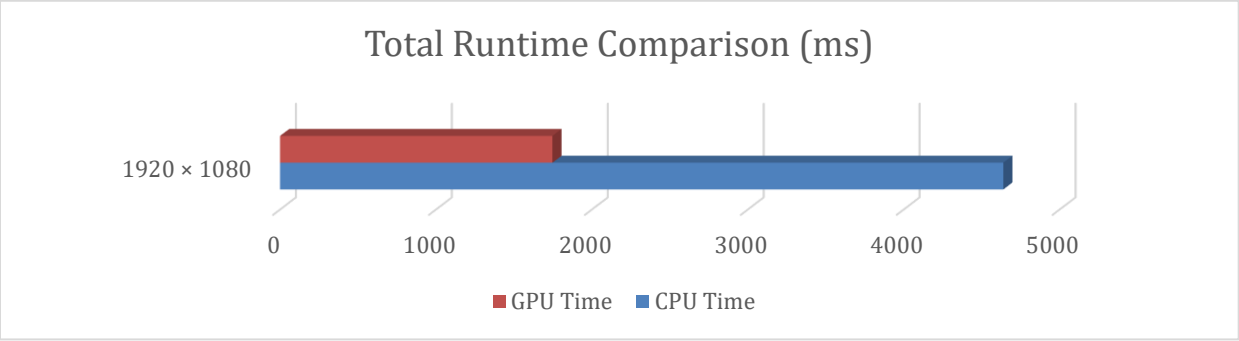
It was noted that the ptrrow and kernel data array was read-only throughout the function and only the ptrout array was written. So, the data construct was used along with the clauses copyin for the ptrrow and kernel.data to allocate and copy on entering and do not copy back on exit. While the copyout clause was used for the ptrout array to only allocate memory on device and copy on exit. In this way, we optimized the data transferring.

Loop Construct & Clauses

The code was double nested with three serial nested loops. So for the outer loop, parallel loop construct was used along with the clause gang and vectors. This ensures that the outer loop iterations will be divided into the gangs (or blocks) with many parallel vectors (or threads). Inside the outer loop, for the inner nested loops, a parallel loop construct along with the vector clause is used to ensure that the inner independent loop iterations are divided into the parallel vectors of the gangs.

Speedup Achieved

CPU Time: 4639.02 ms
GPU Time: 1747.31 ms
Time Saved: ~2891.71 ms
Speedup: 2.65



Conclusion

- **In Deliverable 1 (D1)**, our objective was to profile the original KLT implementation and identify computational bottlenecks. Using tools such as “gprof”, we analyzed function-level runtime distribution. The profiling showed that the majority of total execution time

was dominated by two convolution routines:

- _convolveImageHor()
- _convolveImageVert()

These functions were therefore selected as the primary candidates for GPU acceleration.

- **In Deliverable 2 (D2)**, we ported both convolution routines to CUDA. We created GPU kernels for horizontal and vertical convolution and integrated them into the original CPU code using wrapper functions (**_convolveImageHorizontalUsingGPU**() and **_convolveImageVerticalUsingGPU**). In this stage, the focus was on correctness and achieving baseline GPU acceleration. The code structure was updated to include CUDA device memory allocation, data transfer, GPU kernel launches, and result retrieval.

- **In Deliverable 3 (D3)**, we optimized the CUDA implementation using advanced memory techniques. Specifically:
  - **Shared Memory:** Used to reduce redundant global memory accesses and accelerate neighborhood-based convolution.
  - **Pinned Memory:** Introduced to speed up host-to-device and device-to-host memory transfers.
  - **Constant Memory:** Employed to store convolution kernels, taking advantage of fast cached access.
- During this deliverable, we also integrated detailed timing mechanisms using CUDA events and CPU timers. The code now records GPU compute time, CPU compute time, memory transfer time, convolution call count, and accuracy (mean/max difference between CPU and GPU).

- **In Deliverable 4 (D4)**, we explored directive-based acceleration using **OpenACC**. This provided an alternative approach to parallelization without manually writing CUDA kernels. We annotated the convolution loops with OpenACC pragmas and compared performance, ease of use, and portability with CUDA implementations. This deliverable served as a high-level comparison between low-level (CUDA) and high-level (OpenACC) GPU programming models.

GitHub Private Repository Link

https://github.com/MuhammaDaniyal/Complex_Computing_Problem