CS4110 - High Performance Computing with GPUs
CCP Deliverable 3 - Report

Menahil Ahmad (23I-0546)
Hiyam Rehan (23I-0039)
HPC-A

Github Repository:

https://github.com/mena-aq/KLT-CCP

Overview

As previously discussed, the initial optimization was relatively naive and introduced bottlenecks, particularly due to the overhead of frequent *cudaMalloc* and *cudaMemcpy* operations. To address this, we implemented improved strategies leveraging advanced optimization techniques such as efficient memory hierarchy utilization and CUDA streams. This report shows our findings.

Methodology

In optimising the KLT Feature Tracking application, we first targeted the primary bottleneck identified in V2: device memory allocations and host-device communication. To mitigate redundant data transfers, reusable device buffers were introduced for storing float images, image pyramids, and feature lists. Pyramid metadata was relocated to constant memory, enabling a unified source for all image and gradient pyramids. Convolution performance was further improved through shared-memory tiling with coalesced global memory accesses, effectively reducing global memory traffic. Additionally, the three fixed sigma values used in the application were precomputed into constant Gaussian and derivative kernels to minimise computation and transfer overhead. Host feature list memory was pinned to accelerate per-frame data transfers, and CUDA streams were employed to overlap pyramid construction with feature tracking. Finally, the atomic addition operation used in tracking was replaced with sequential accumulation, as the small feature window size rendered atomic operations inefficient.

By implementing these optimisation techniques, we achieved up to a ***9.8x*** speedup over our initial GPU baseline and up to a ***24x*** improvement compared to CPU execution, as detailed in the results below.

Results

To evaluate the performance gains from our optimisations, we tested our implementation across three datasets of varying sizes. ***Dataset 2 (300 frames, 640×360 px)*** represents a small workload, ***Dataset 4 (500 frames, 960×540 px)*** a medium workload, and ***Valley (200 frames, 1920×1080 px)*** a large workload.

Version	**Dataset**	**Avg runtime**	**Speedup V1**	**Speedup V2**
V1	**Dataset 2**	3.4611	-	-
	Dataset 4	12.9267	-	-
	Valley	26.0776	-	-
V2	**Dataset 2**	1.7812	1.4x	-
	Dataset 4	5.2474	1.8x	-
	Valley	7.9810	2.5x	-
V3 (V3.7)	**Dataset 2**	0.3267	7.9x	5.5x
	Dataset 4	0.8015	11.7x	6.5x
	Valley	0.8113	24.4x	9.8x

Table 1.1 - Performance Comparison of V1 (CPU), V2(unoptimised GPU), and V3(optimised GPU) across 3 main datasets

	v3.1	**v3.2**	**v3.3**	**v3.4**	**v3.5**	**v3.6**	**v3.7**
Speedup V1	9.8x	9.9x	9.8x	11.2x	11.5x	10.4x	11.8x
Speedup V2	5.5x	5.5x	5.5x	6.3x	6.5x	5.8x	6.5
Remarks	Reusable Buffers	Shared memory convolve	Pinned memory feature list	Async image transfer	Const memory for gauss kernels	Remove atomic add	Coalescing convolve

Table 1.2 - Observed Speedup from V1(CPU baseline) and V2(GPU baseline) of each optimisation in V3

As shown in Table 1.2, the introduction of reusable buffers and CUDA streams yielded the most substantial speedup, while optimisations targeting the memory hierarchy (shared and constant memory) provided comparatively smaller improvements. This reinforces our earlier observation from V2 that host–device communication was the dominant performance bottleneck. Furthermore, we observed that higher-resolution datasets achieved greater overall speedups, suggesting that the GPU's parallel throughput scales more effectively with increased computational intensity than with the number of frames alone.

Discussion

Profiling our application reveals insight into the success of our targeted optimisations. By reusing buffers, we were able to cut down on host-device communication time (*cudaMemcpy*) by ***96.6%***, and memory transfer size by ***91.4%***. By reducing global memory accesses in our kernels, total kernel execution time was reduced by ***10.8%***.
The extent of our optimisation was limited by several factors, including the inherent sequentialism present in feature tracking (unable to parallelise pyramid computation etc.), data dependence between smoothing and tracking, frequent per-frame updates to host feature data etc. Furthermore, to reduce memory transfer times we had to implement a subsampling kernel, adding slightly (*0.9%*) to our kernel time.
Overall, these results represent a substantial step toward an efficient and scalable GPU implementation of KLT feature tracking.

Future Work

In our next stage of development, we aim to leverage OpenACC directives to explore compiler-driven optimisations. By abstracting our current CUDA implementation, we hope to assess performance portability and scaling opportunities across different GPU architectures, while maintaining a speedup close to our optimized CUDA implementation.