CS4110 - High Performance Computing with GPUs
CCP Deliverable 2 - Report

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

Github Repository:

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

Overview

In the previous stage of our work (V1), we identified convolution and interpolation routines as the primary hotspots in the Kanade-Lucas-Tomasi (KLT) Feature Tracking Application. This report presents our implementation of the proposed naive GPU port, which includes convolution kernels and parallel feature tracking to exploit fine-grained parallelism.

Experimental Setup

To port our application to the GPU, we utilised the CUDA framework. At this stage, our primary focus was on implementing routines for memory allocation, data transfers, and kernel launches. Building on the work from V1, we developed convolution and feature tracking kernels to perform pixel-wise convolutions and track features between frames in parallel. To evaluate performance, we profiled the application using *NVIDIA Nsight Systems* (*nsys*), collecting metrics on CUDA API calls, kernel execution times, and communication overhead. We conducted 3 profiling runs for Dataset 2 (95 frames of 400×400 pixels) and Dataset 3 (10 frames of 320×240 pixels), averaged the results to smooth out variability, and computed per-frame times to enable fair comparison across datasets.

Results

Our results are summarised in **Tables 1.1 to 1.4.**
As expected, device memory allocation and host-device communication dominate overall application time. Memory transfers account for roughly 1 to 2% of total runtime across datasets, with H2D communication dominating over D2H communication, primarily due to the naive implementation design that performs separate memory copies per kernel launch.
Additionally, it is interesting to note that the smaller dataset (Dataset 3) exhibits proportionally higher memory allocation overhead (~15%) relative to its total runtime, suggesting that fixed setup costs become more pronounced at lower workloads.
Kernel execution itself constitutes less than 0.3% of total time, confirming that our current bottleneck lies in CPU–GPU interaction and data management rather than computation.

Component	**Average Time (ns)**	**Average % of Total Time Spent**
Total Execution Time	7,744,730,669	100%
Memory Transfer Time	132,946,790	1.72%
Memory Allocation Time	148,030,380	1.91%
Kernel Time	20,796,047	0.27%

Table 1.1 - *Performance Analysis on Dataset 2 (95 400x400 frames)*

Component	**Average Time (ns)**	**Average % of Total Time Spent**
Total Execution Time	660,203,013	100%

Memory Transfer Time	6,887,358	1.04%
Memory Allocation Time	98,062,381	14.85%
Kernel Time	1,603,213	0.24%

Table 1.2 *- Performance Analysis on Dataset 3 (10 320x240 frames)*

Dataset	H2D Transfer (ns / MB)	D2H Transfer (ns / MB)	% of Total Memory Time
Dataset2	79.4M ns / 915 MB	52.9M ns / 502 MB	60% / 40%
Dataset3	4.09M ns / 44 MB	2.81M ns / 25 MB	59% / 41%

Table 1.3 *- Memory Transfer Breakdown (Host to Device (H2D) and Device to Host (D2H))*

Kernel	% Time	Calls / frame
trackFeatureKernel	81.63%	1
convolveVertKernel	9.43%	6
convolveHorizKernel	8.95%	6

Table 1.3 *- Kernel Time Distribution*

Discussion

Comparing our (naive) GPU implementation to the CPU code, our kernel is working effectively as it takes up less than 1% of the computational time. However, as a result, a significant portion of the overall runtime is now spent on cudaMemcpy and cudaMalloc operations, representing data transfer and memory allocation overheads between the host and device. These communication costs limit the achievable speedup and highlight the need for further optimization through memory reuse or asynchronous data transfers. In the current implementation, there are many small memory allocations and frequent data copies, along with non-optimal block and thread configurations, all of which can be improved in future versions.

Future Optimisation Opportunities

For the future versions, we plan to explore different ways to make our GPU version faster. We may adjust the kernel launch settings to use the GPU more efficiently and test occupancy improvements by moving reusable image data to shared memory and changing thread and block configurations. Our main focus will be on reducing communication overhead, since most of the current time is spent on data transfers and memory allocations (cudaMemcpy and cudaMalloc). We will also look into using different parts of the GPU memory hierarchy like shared, texture, or constant memory to reduce heavy global memory access. Additionally, we will try reusing the same device memory for pyramid data across frames to reduce repeated allocations and improve overall runtime efficiency. These are possible directions we will test, depending on how each change affects the overall results.