CS4110 - High Performance Computing with GPUs
CCP Deliverable 4 - Report

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

Github Repository:

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

Abstract

The Kanade-Lucas-Tomasi (KLT) feature tracker is a fundamental algorithm in computer vision, widely used for motion estimation, object tracking, and visual odometry. However, real-time KLT tracking on high-resolution imagery remains computationally prohibitive on conventional CPUs, limiting its deployment in time-critical applications such as autonomous navigation and augmented reality. This paper presents a comprehensive GPU acceleration study of the KLT algorithm. We develop a naive CUDA implementation and systematically apply optimizations including reducing allocations and data transfers, asynchronous computation via streams, and shared memory utilization. Additionally, we implement an OpenACC version to evaluate the performance versus programmer productivity trade-off in directive-based programming models.

Our optimised CUDA implementation achieves ***24x*** speedup over the CPU baseline on a HD (High Definition) dataset, while the OpenACC implementation delivers ***7.8x*** speedup. Major performance gains in our CUDA optimization were obtained by reduced device-host communication (*5.7x* from CPU baseline), and parallelism through asynchronous lookahead data transfer and kernel execution (*7.2x* from CPU baseline).

Our results demonstrate that while OpenACC offers rapid development and easy performance gains, feature tracking algorithms like KLT benefit significantly from the fine-grained control CUDA provides, including explicit memory management and low-level optimization of irregular access patterns.

Introduction

A Kanade–Lucas–Tomasi (KLT) tracker is a feature extraction technique used to identify and follow changes in scale and motion across a sequence of images. The algorithm performs per-pixel analysis on each frame to detect salient features and propagate them across subsequent frames. As a result, its computational workload increases significantly with higher-resolution datasets, leading to reduced performance on standard CPU-based execution. To address this, our project focuses on accelerating the provided C implementation using High-Performance Computing techniques on GPUs.

To achieve this, we first profiled the code to identify bottleneck functions and re-implemented these components using CUDA kernels. After establishing a baseline speedup, we further optimized the kernels by reducing overhead and leveraging the GPU memory hierarchy to further improve performance. Finally, to explore an alternative and potentially more portable parallelization strategy, we implemented OpenACC kernels to evaluate whether additional performance gains could be achieved.

Background and Related Work

Recent research has explored leveraging GPU architectures to accelerate KLT and related optical flow algorithms. GPUs, with their massively parallel processing units and hierarchical memory, are particularly suited to image processing workloads. Different frameworks, including CUDA, OpenACC, OpenMP and OpenCL have been utilized to accelerate feature tracking across multiple algorithms. Farias et. al (2009) achieved 90x speedup using CUDA on a 1024x1024 video stream, while extracting 1000 features. Fassold et al. (2009) achieved 10x speedup on a High Definition workload, with marginal differences in tracking due to data precision. Overall, the inherently parallel operations in feature tracking, including convolutions and interpolations, make KLT Feature Tracking well-suited for GPU acceleration.

Implementation

V1- Sequential Baseline

We began with the provided C implementation of the KLT tracker and evaluated its performance using both the supplied dataset and a larger external dataset. To identify the most computation-intensive sections of the code, we profiled the baseline implementation using gprof and visualized the results with gprof2dot, allowing us to pinpoint the primary bottleneck functions for GPU acceleration.

We found our main bottlenecks to be:

- **_convolveImageHoriz() and _convolveImageVert(),** taking about ~67% of execution time due to repeated matrix convolution operations (for each frame)
- **_interpolate(),** taking ~20% time, mostly bc of the frequency of it's calls.

To tackle these bottlenecks, we moved on to the optimization stage.

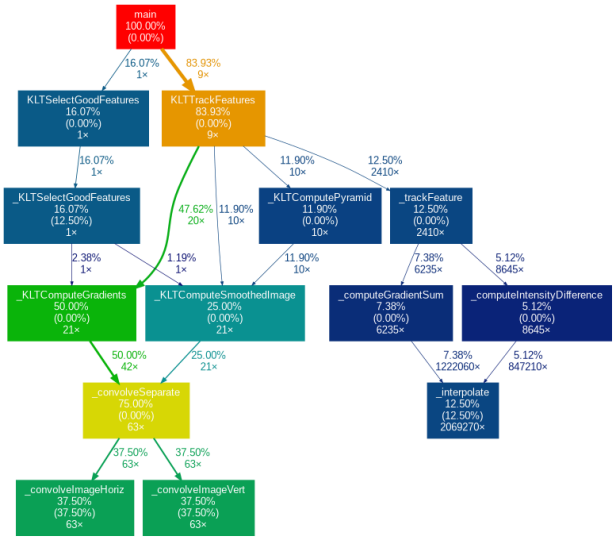


Fig 1.1 - *Sample callgraph of KLT Feature Tracker generated using gprof2dot*

V2 - Naive CUDA implementation

From there, we ported the identified bottleneck functions to CUDA kernels to enable parallel execution. This naive CUDA kernelization achieved up to a 2.5× speedup on the large dataset. However, the improvement was limited by increased host–device communication, as memory transfer overheads began to dominate the runtime. This highlighted the need for further optimisation beyond basic kernelization, particularly in reducing data movement and managing memory more efficiently.

V3 - Optimised CUDA

To further improve performance, we applied several optimisation strategies, including effective use of the GPU memory hierarchy such as shared memory, constant memory, and pinned memory. We also significantly reduced the number of Host-to-Device and Device-to-Host transfers, which proved critical in lowering overhead. With these iterative optimisations, we achieved up to a 24× speedup on the largest dataset. A detailed breakdown of the incremental gains for each optimisation step is shown later in Table 1.2.

V4 - OpenACC

As a final approach, we decided to experiment with OpenACC directives to evaluate the trade-off between programmability and performance. While our OpenACC acceleration was easier to implement, with minimal changes to the codebase, we only were able to achieve a maximum speedup of 7.8x, primarily through parallelising the major bottlenecks in the convolution kernels. Further attempts, including optimising data transfers between host and device, unexpectedly led to performance degradations, leading us to retain a straightforward convolution-based optimisation as the most effective OpenACC approach.

Experimental Setup

All experiments are performed on a GPU-enabled server equipped with NVIDIA GeForce RTX 3080 (Compute Capability 8.6). To report timings fairly, we run each experiment three times and report average runtimes. GPU performance metrics (kernel time, memory transfers, CUDA API behaviour) are collected using Nsight Systems (nsys).

To evaluate the performance of our optimisations, we use three datasets of varying scale and resolution: ***Desk Scene*** (10 frames, 320×240), representing a **small** workload; ***Cityscape*** (300 frames, 640×360), representing a **medium**-sized workload; and ***Valley*** (200 frames, 1920×1080), representing a **large** HD-resolution workload. This range allows us to assess how performance scales with both frame count and image resolution.

Results

The following section presents the performance of our KLT implementations, from the initial naive CUDA port, to our optimisation, and finally, our OpenACC implementation. We report absolute runtimes, kernel-level breakdowns, memory-transfer costs, and overall speedup relative to the CPU baseline, to present a comprehensive analysis of our accelerated KLT Feature Tracker.

Overall Performance Comparison

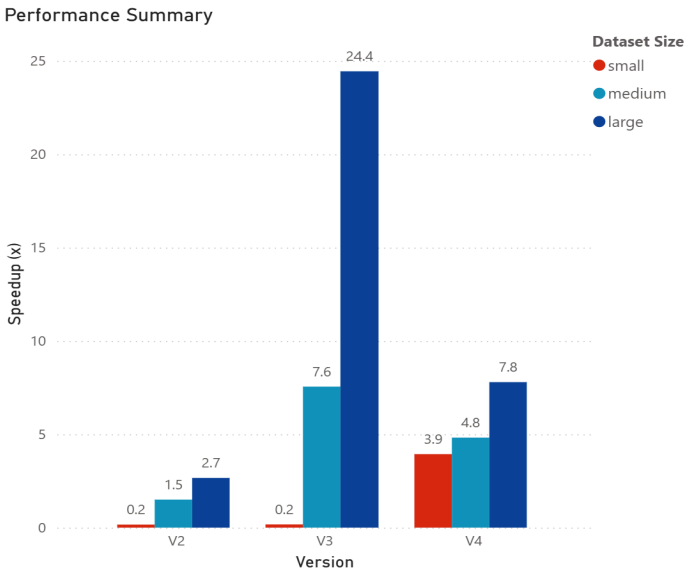


Fig 2.1 - *Performance Summary of V2 (Naive CUDA port), V3 (CUDA optimisation) and V4 (OpenACC GPU port) across our workloads*

While our smaller dataset showed slower performance in our CUDA implementations due to memory transfer overheads, there was a large performance improvement for our HD dataset, with our optimised CUDA tracker showing an improvement of 99.2% from a small to large dataset. Our OpenACC implementation, while giving speedup across all datasets, does not scale as well, with only a 50% performance gain over small to large datasets.

Version	Workload	Runtime (s)	Speedup
V1: CPU baseline	Small	0.0346	-
	Medium	2.70	-
	Large	19.8	-
V2: Naive GPU CUDA	Small	0.213	0.16x
	Medium	1.80	1.5x
	Large	7.45	2.6x
V3: Optimised CUDA	Small	0.201	0.17x
	Medium	0.357	7.5x
	Large	0.811	24x
V4: OpenACC	Small	0.0397	3.9x
	Medium	2.712	4.8x
	Large	11.4	7.8x

Table 2.2 - *Overall feature tracking speedup across workloads*

To analyse the impact of specific optimisations, we run our workloads against the following versions:
V1: CPU baseline
V2: Naive CUDA port
V3.1: Reusable Buffers and Constant Memory Pyramid Metadata
V3.2: Shared Memory Utilization in Convolutions
V3.3: Pinned Memory Implementation of Feature List
V3.4: Asynchronous lookahead frame transfer
V3.5: Constant Memory Gaussian Kernels
V3.6: Removal of Atomic Operations
V3.7: Coalesced Global Memory Access Loading Pattern in Convolutions
V4: OpenACC implementation of Convolution

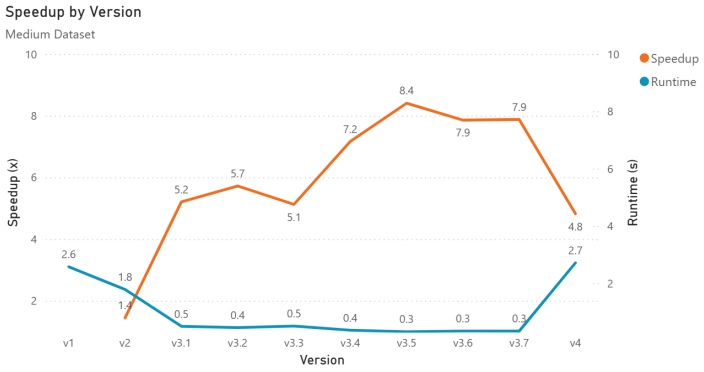


Fig 2.3 - *Speedup by Version on medium workload, with measured average runtime*

As observed in Fig 1.3, our CUDA optimisations produced consistent performance improvements, with a temporary drop observed during the introduction of pinned feature lists, which subsequently enabled a larger gain from V3.3 to V3.4. In contrast, the OpenACC implementation exhibited limited speedup, only slightly outperforming the naive CUDA port (V2). This indicates that applications such as the KLT Feature Tracker benefit from fine-grained control over memory access and communication, which high-level directives cannot fully replicate. The complexity of the original CPU implementation further constrains the performance achievable via OpenACC, with additional overhead required to adapt the code leading to performance degradation. Indeed, our OpenACC experiments revealed up to a 3× slowdown when applying data and parallelisation directives, with optimised convolution emerging as the most effective approach for balancing performance and implementation effort.

Kernel Breakdown

Insight into our kernel performance reveals that while tracking was the major bottleneck in our naive CUDA port,our optimisations achieved the greatest speedup in this kernel, resulting in a V3 kernel distribution closely aligning with the CPU baseline. Due to the complexity and observed performance degradation of tracking in OpenACC, we omitted this kernel from our implementation, observing almost equal distribution of kernel time across both convolutions.

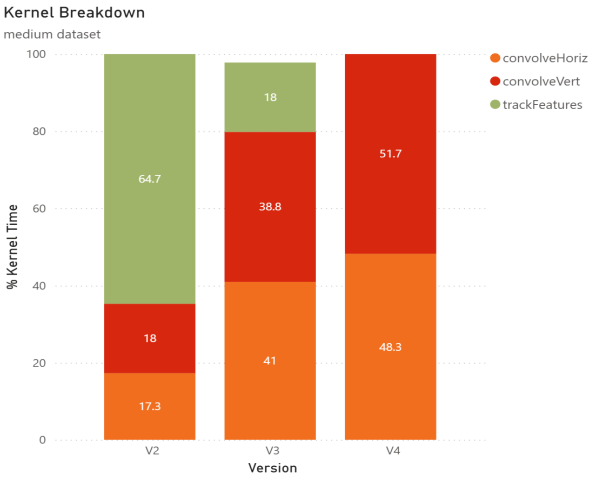
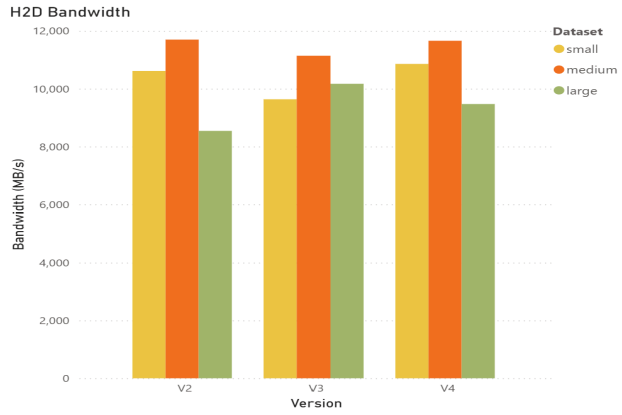


Fig 3.1 - *Kernel Breakdown of medium workload across versions (% kernel time)*

		Kernel Time (ms / %)		
	Workload	**Horizontal Convolution (_convolveHoriz)**	**Vertical Convolution (_convolveVert)**	**Feature Tracking (_trackFeature)**
V2	Small	0.144 (9%)	0.152 (9.5%)	1.30 (81.5%)
	Medium	7.08 (17.3%)	7.34 (18%)	26.4 (64.7%)
	Large	29.3 (36.4%)	30.5 (37.9%)	20.7 (25.7%)
V3	Small	0.167 (34.9%)	0.176 (34.9%)	0.130 (27.1%)
	Medium	0.218 (41%)	0.777 (38.8%)	0.361 (18%)
	Large	34.8 (49.4%)	32.1 (45.5%)	2.50 (3.5%)
V4	Small	11.4 (52.2%)	10.5 (47.8%)	-
	Medium	832 (48.3%)	891 (51.7%)	-
	Large	4,107 (49%)	4,273 (51%)	-

Table 3.2 - *Detailed kernel runtimes (ns and % kernel time) across versions and workloads*

Bandwidth Utilisation



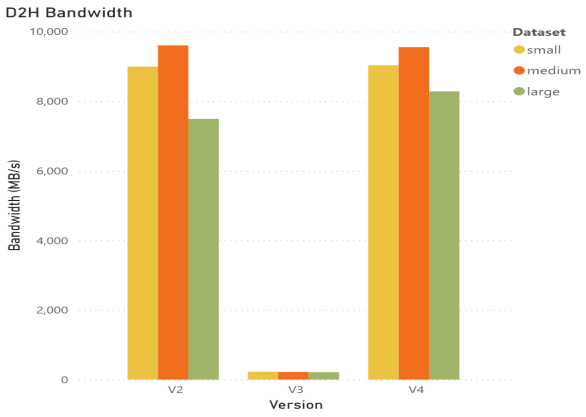


Fig 4.1 *- (a) H2D and (b) D2H Bandwidth utilisation across versions and workloads*

Analysing our bandwidth utilisation, we observe a drastic reduction in D2H transfers in V3 due to the usage of pinned feature list memory. This shows the overhead page faults can introduce in host-device communication. It is also notable that the medium dataset shows the highest bandwidth utilization, highlighting the potential performance of our application under realistic workload conditions.

Discussion

Our results highlight several key insights regarding GPU optimisation of the KLT Feature Tracking application. CUDA optimisations provided substantial performance improvements, particularly through buffer reuse, pinned memory, and concurrent streams, demonstrating that careful control over memory allocation and kernel execution is critical for high-performance implementations. In contrast, OpenACC offered modest speedups but failed to scale efficiently across datasets,

underscoring the limitations of compiler-directed directives for applications with fine-grained memory and computational dependencies.

The trade-offs between implementations are evident across our observations, while OpenACC provides quick initial gains, it is constrained by the underlying CPU-based algorithm, resulting in lower performance ceilings and limited scalability across workloads. On the other hand, CUDA offers substantial performance improvement and scaling, but at the cost of extensive manual tuning at a fine-grained level.

It is also noteworthy that our optimisations were constrained by inherent sequential dependencies within the KLT algorithm, such as sequential pyramid traversal, predefined algorithmic choices, and the necessity for frequent per-frame data updates, which collectively limit the maximum achievable performance gains.

Conclusion

Overall, our work demonstrates that while both CUDA and OpenACC can accelerate the KLT Feature Tracker, CUDA delivers significantly higher performance due to its fine-grained control over memory, kernels, and parallelism. This is mainly because OpenACC is not well suited for our use-case, as the KLT algorithm needs more detailed control over data movement and computation than OpenACC can provide, leading to much lower speedups compared to CUDA.