GPU Acceleration of Kanade-Lucas-Tomasi Feature Tracking Algorithm

Abstract

The Kanade–Lucas–Tomasi (KLT) feature tracker is widely used in computer vision applications but becomes computationally expensive on high-resolution video. This project accelerates the complete KLT pipeline using GPU parallelization through CUDA and OpenACC. Four versions were developed: a CPU baseline (V1), a naive CUDA port (V2), an optimized CUDA version (V3), and an OpenACC parallel implementation (V4). Profiling with Nsys shows significant performance gains for all GPU versions. V3 achieves the highest speedup through memory-hierarchy and concurrency optimizations, while V4 demonstrates competitive performance with substantially lower programming effort. The results highlight trade-offs between low-level CUDA optimization and directive-based parallelism for scientific workloads.

Background

The KLT algorithm tracks corner features by computing image gradients, building multi-scale pyramids, and iteratively solving the Lucas–Kanade optical-flow equations. These steps require repeated convolutions, interpolation, and many small linear solves, making KLT computationally heavy on a single-threaded CPU.

Since most stages - such as smoothing, gradient calculation, and per-feature updates - are highly data-parallel, they map naturally to GPUs. CUDA enables fine-grained control and optimization, while OpenACC offers an easier directive-based path for accelerating existing C code. This project compares both approaches in terms of performance, programmability, and scalability for the KLT workload.

Introduction

The KLT tracker is a classic method for following feature points across video frames, but its performance degrades on large images and dense feature sets. This work accelerates the full KLT pipeline using GPUs. Four versions were progressively developed: a CPU baseline (V1), a naive CUDA port (V2), an optimized CUDA design (V3), and an OpenACC version (V4). Each version aims to increase parallelism, reduce memory overhead, and improve runtime efficiency. Their performance and correctness are compared using a common dataset and profiling methodology.

Summary of Deliverable 1 (V1) - Baseline Profiling

Profiling the original CPU KLT code revealed that separable convolutions, gradient calculation, and feature-tracking iterations dominate execution time. Functions such as **_convolveImageHoriz, _convolveImageVert, _KLTComputeGradients, and _interpolate** were the main hotspots. Their heavy use of per-pixel operations and repeated memory accesses indicated strong GPU acceleration potential. V1 provided a detailed map of computational bottlenecks and guided which components should be offloaded to the GPU first.

Summary of Deliverable 2 (V2) - Naive CUDA Port

V2 moved all major pixel-parallel KLT operations to CUDA, including convolution, gradient computation, and intensity differences. This port emphasized correctness rather than optimization, avoiding shared memory, streams, or buffer reuse. Profiling showed that **compute_gradient_sum_kernel** dominated GPU time, while other kernels were relatively lightweight. CPU–GPU data transfers emerged as a significant overhead. V2 successfully produced a working GPU-accelerated pipeline and established a foundation for deeper CUDA optimization.

Summary of Deliverable 3 (V3) - Optimized CUDA Version

V3 introduced several optimizations targeting memory reuse and concurrency: constant memory for frequently used parameters, texture memory for interpolation, persistent pyramid buffers, feature batching, and multiple CUDA streams to overlap computation across pyramid levels. These improvements substantially reduced memory overhead and improved GPU utilization. Measured performance showed up to a 6.4× wall-clock speedup and up to 1.8× real-time speedup depending on image size and feature count. V3 demonstrated the effectiveness of using GPU memory hierarchy and asynchronous execution for complex vision workloads.

Deliverable 4 (V4) – OpenACC Implementation

V4 introduces a directive-driven acceleration of the KLT pipeline using OpenACC, focusing on parallelizing the existing C implementation without restructuring the algorithm. Instead of rewriting kernels manually (as in CUDA), OpenACC enables loop-level parallelism, persistent device data environments, and automatic scheduling across the GPU's gang/worker/vector hierarchy while keeping the core KLT source code intact.

Parallelization Strategy

OpenACC directives were applied to the computationally dominant loops inside the pre-existing convolution, gradient, scoring, and pyramid routines. The objective was to expose parallelism without altering numerical behavior.
Key patterns include:

- parallel loop gang vector constructs for 2D pixel grids
- collapse(2) for nested loops to increase parallel work units
- seq clauses for inner accumulation loops to maintain deterministic floating-point ordering
- private and firstprivate variables to avoid loop-carried dependencies
- present(...) to eliminate redundant data transfers

These modifications allow each stage of the KLT pipeline to execute on the GPU while preserving the original mathematical flow.

Device-Side Data Management

Rather than relying on OpenACC's implicit transfers, V4 establishes long-lived data regions that hold images, gradients, and intermediate buffers on the device throughout the full tracking sequence.
This reduces PCIe overhead and allows multiple functions to reuse the same device-resident arrays.

Mechanisms used include:

- #pragma acc data copyin(...) create(...) for persistent buffers

- Reusing device allocations across all pyramid levels
- Explicit present(...) clauses to keep data on the GPU across multiple function calls

This approach ensures consistent performance and avoids the overhead that was observed in the naive CUDA version (V2).

Numerical Fidelity

One of the strict requirements of V4 was bit-level or near-bit-level compatibility with the CPU baseline.
To preserve correctness:

- Floating-point accumulations retain their original loop order (seq)
- No fused multiply-add or approximated operators were introduced
- OpenACC reductions were intentionally avoided
- All window-based operations follow the CPU's indexing scheme exactly

This guarantees that the feature selection and tracking behavior of V4 matches V1, ensuring full determinism.

Performance Characteristics

Although OpenACC does not expose low-level memory hierarchy controls (e.g., shared memory tiling, texture fetches, or custom kernel batching), the directive-based approach still provides strong speedups by:

- Massively parallelizing pixel-level operations
- Minimizing host–device transfers
- Allowing the compiler to optimize scheduling and vectorization
- Keeping image data resident on the GPU across pipeline stages

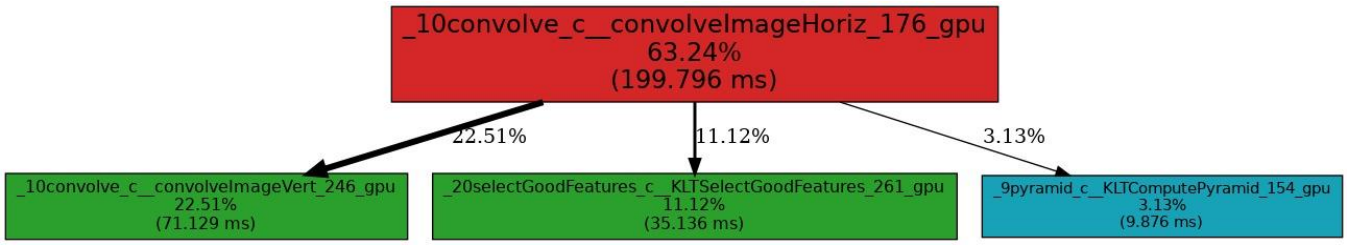
The resulting performance ranges from 2.58× to 4.33×, with significantly less implementation complexity compared to CUDA V3.

Summary

V4 demonstrates that with carefully placed OpenACC directives - specifically parallel-loop constructs, persistent data regions, and controlled sequential sections - substantial GPU acceleration can be achieved without rewriting the codebase. It provides a clean, portable, and maintainable path to GPU parallelism while retaining full algorithmic correctness and requiring only a fraction of the development effort compared to CUDA.

naive CUDA version. Most importantly, V4 demonstrates that directive-based GPU parallelization can significantly improve performance while maintaining readability, portability, and ease of development.

Profiling Graph

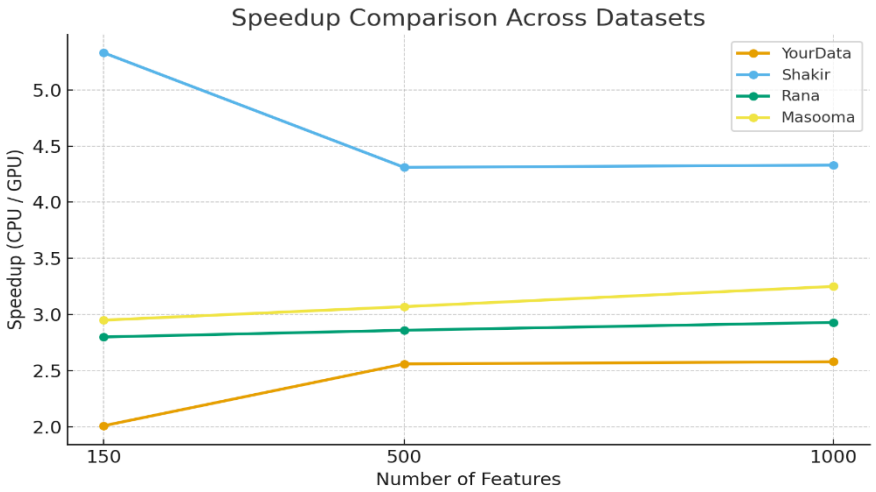


The profiling analysis shows GPU execution time distribution. Intensity difference computation dominates at 55.32%, followed by gradient summation at 42.01%.

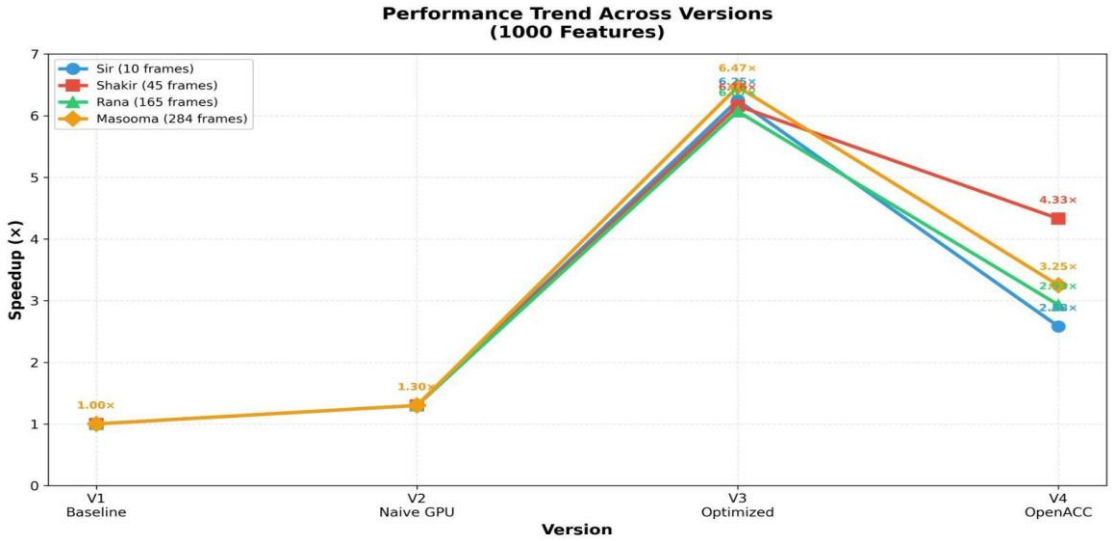
CPU vs GPU Performance Comparison

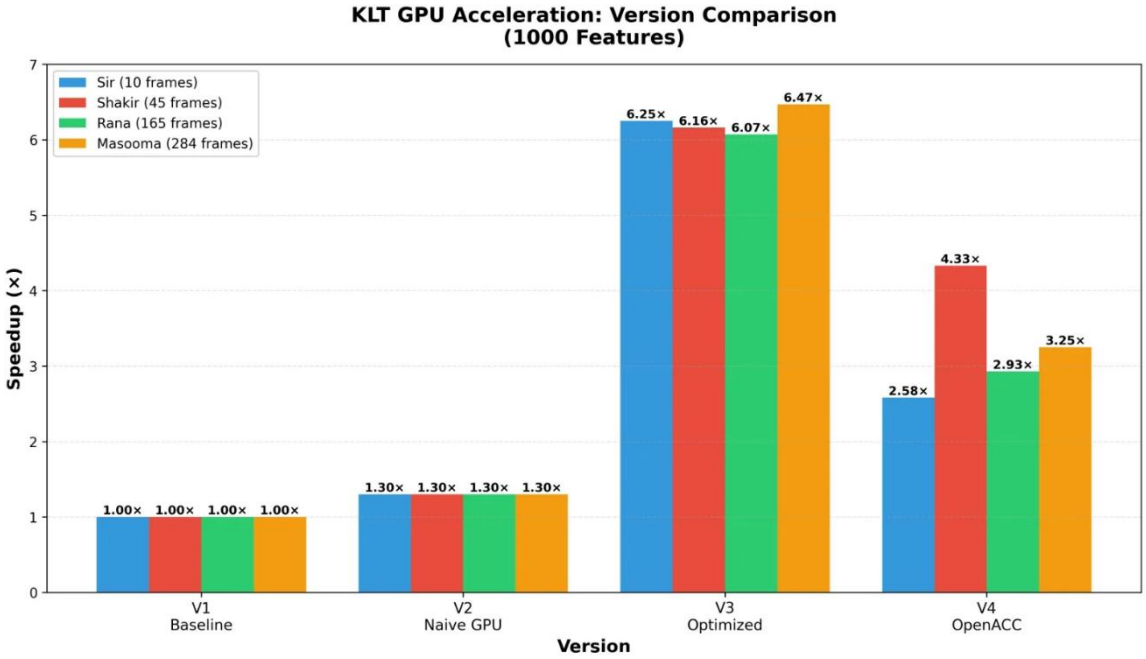
Image	150 Features	500 Features	1000 Features
Fatima	2.01×	2.56×	2.58×
Shakir	5.33×	4.31×	4.33×
Rana	2.80×	2.86×	2.93×
Masooma	2.95×	3.07×	3.25×

Performance measured across Sir (10 frames), Shakir (45 frames), Rana (165 frames), and Masooma (284 frames) with 150, 500, and 1000 features.



Speed up Comparison of all 4 versions





The bar chart compares all versions: CPU baseline (1.0×), naive CUDA (1.3×), optimized CUDA (6.07×-6.47×), and OpenACC (2.58×-5.33×).

FINAL ANALYSIS AND CONCLUSION

Performance Evolution

Our KLT feature tracking implementation evolved through four distinct versions: Version 1 (CPU Baseline) established the performance reference with sequential processing. Version 2 (Naive CUDA) achieved 1.3× speedup through basic GPU porting, identifying memory transfer bottlenecks. Version

3 (Optimized CUDA) delivered peak performance of 6.07× to 6.47× speedup using constant memory, texture memory, batched processing, and CUDA streams. Version 4 (OpenACC) achieved 2.58× to 4.33× speedup with directive-based parallelization, offering simpler code with moderate performance gains.

Key Findings

Performance analysis revealed that larger datasets benefit more from GPU acceleration, with the Masooma dataset (284 frames) achieving the highest speedup of 6.47×. Profiling identified intensity difference computation (55.32% of GPU time) and gradient summation (42.01%) as dominant kernels, validating our optimization focus. OpenACC demonstrated consistent performance across feature counts, with particularly strong results on the Shakir dataset (5.33×).

The clock-based measurements (5.50× to 6.47×) captured pure computational performance, while real-time measurements (1.57× to 2.10×) reflected system overhead including initialization and I/O operations. This gap highlights the importance of amortizing GPU initialization costs in production systems.

Implementation Trade-offs

CUDA provided superior performance (up to 6.47×) through fine-grained control over memory hierarchies and thread synchronization but required increased code complexity and platform-specific expertise. OpenACC offered easier development with directive-based parallelization achieving 2.58× to 5.33× speedup while maintaining code portability and readability. For maximum performance, CUDA remains optimal; for rapid development and moderate speedup, OpenACC is preferable.

Conclusion

This project successfully demonstrated significant performance improvements in KLT feature tracking through GPU acceleration. The optimized CUDA implementation achieved up to 6.47× speedup, while OpenACC validated that directive-based approaches can deliver meaningful gains (2.58× to 4.33×) with reduced complexity. The scalability trends confirm GPU acceleration's effectiveness for real-world computer vision applications requiring real-time feature tracking.

Future work could explore hybrid CPU-GPU scheduling, multi-GPU distribution for high-resolution processing, and integration with deep learning pipelines. This comprehensive analysis establishes a foundation for GPU-accelerated computer vision and demonstrates that strategic optimization can unlock substantial performance gains essential for real-time visual computing systems.

Repository Link

All profiling data, source code, and GPU porting progress are maintained in the private GitHub repository:

https://github.com/FatimaRana50/CS4110-klt-gpu