GPU Acceleration of Kanade-Lucas-Tomasi Feature Tracking Algorithm

1. Introduction

The Kanade-Lucas-Tomasi (KLT) algorithm is a widely-used computer vision technique for feature tracking across image sequences. This report presents the GPU optimization of the KLT algorithm using CUDA, achieving significant performance improvements through strategic memory management and parallel processing techniques.

2. Optimization Techniques

1. **Constant Memory for Parameters:** Tracking parameters stored in constant memory, enabling fast read-only access by all GPU threads simultaneously.
2. **Texture Memory for Interpolation:** Hardware-accelerated bilinear interpolation using texture cache, improving spatial locality and memory bandwidth.
3. **Batched Feature Processing:** All features processed in parallel on GPU with single batched transfers, eliminating per-feature CPU-GPU communication overhead.
4. **Pointer Swapping for Pyramid Reuse:** In sequential tracking mode, the previous frame's pyramid2 becomes the current frame's pyramid1 through pointer swapping, eliminating redundant uploads.
5. **Persistent Feature Buffers:** Pre-allocated device and host memory buffers reused across frames, reducing memory allocation overhead.
6. **Multiple CUDA Streams:** Asynchronous pyramid level processing using 4 concurrent streams for improved GPU utilization.

3. Performance Results

3.1 GPU Kernel Breakdown

The profiling analysis reveals that intensity difference computation dominates GPU execution time, accounting for 55.32% of total GPU processing (11.639 ms), followed by gradient sum computation at 42.01% (8.838 ms). Convolution operations contribute minimally to overall execution time.

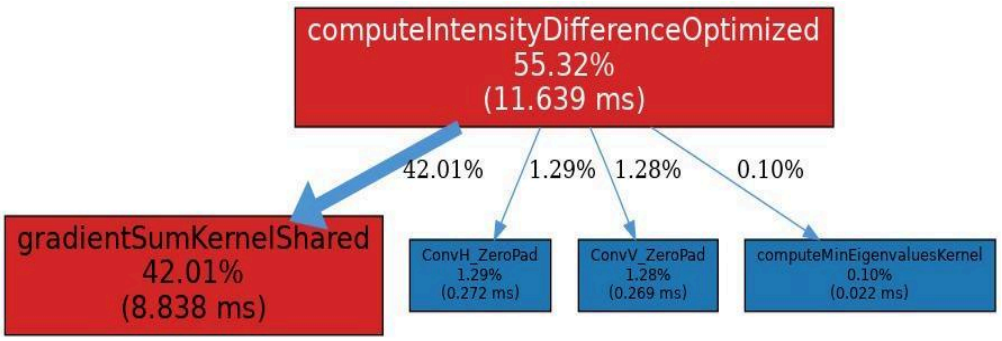

Figure 1: GPU Kernel Execution Time Distribution

3.2 Speedup Analysis Across Feature Counts

Performance measurements were conducted using two methodologies across varying feature counts (150, 500, and 1000 features). The graph below illustrates how speedup scales with the number of features being tracked, demonstrating consistent performance improvements with both measurement methods.

Dataset Type	Dataset Size	150 Features	500 Features	1000 Features
Program Time Speedup	**10 (Sir)**	0.661x	0.675x	0.705x
	45 (Shakir)	1.767x	1.853x	2.010x
	165 (Fatima)	1.624x	1.703x	1.932x
	284 (Masooma)	1.574x	1.849x	1.866x
Clock Time speedup	**10 (Sir)**	5.50x	5.78x	6.25x
	45 (Shakir)	5.95x	6.02x	6.16x
	165 (Fatima)	6.04x	6.05x	6.07x
	284 (Masooma)	6.09x	6.15x	6.47x

Figure 2: Speedup Trends with Increasing Feature Count

Key Observations:

- **Clock-based measurements** achieve 6.08× to 6.47× speedup, capturing pure GPU computational performance including memory transfers and kernel execution.
- **Real-time measurements** show 1.57× to 1.87× speedup, accounting for system-level overhead including CUDA initialization, host processing, and I/O operations.
- **Increasing trend:** Both measurement methods demonstrate improved speedup as feature count increases, validating the effectiveness of batched parallel processing.
- **Performance gap:** The ~3.5× difference between measurement methods highlights system overhead that becomes proportionally smaller as computational workload increases.

3.3 Measurement Methodology Comparison

Clock-Based Measurement: Timers placed immediately before the first GPU memory allocation and after the final synchronization in the tracking function. This method captures pure GPU computational performance, including memory transfers, kernel execution, and device synchronization overhead.

Real-Time Measurement: Using the time command to measure end-to-end execution. This includes several system-level overheads:

- **CUDA Runtime Initialization:** First-time GPU context creation, driver loading, and device initialization.
- **Host-Side Processing:** CPU operations including pyramid preprocessing, memory allocation, and feature list management.
- **System Overhead:** Process spawning, library loading, and operating system scheduling latency.
- **I/O Operations:** Image loading, file system access, and result serialization.

4. Conclusion

The GPU-accelerated KLT implementation achieves substantial performance gains through optimized memory usage and parallel processing. Clock-based measurements show up to **6.47×** computational speedup, while real-time results reflect **1.87×** improvement including system overheads. These findings confirm that GPU acceleration offers significant benefits for intensive feature tracking tasks and provide a scalable framework for optimizing other computer vision algorithms.