**HIGH PERFORMACE COMPUTING with GPUs**

**GROUP MEMBERS:**

**NABEEHA MAHMOOD 23i-0588**

**MAHAM FATIMA 23i-0685**

**Submitted to:**

**Dr. Imran Ashraf**

GITHUB REPO: <https://github.com/NabeehaMahmood/KLT-Feature-Tracker-GPU-Acceleration>

**COMPLEX COMPUTING PROBLEM**

**Deliverable #2**

**FALL 2025**

**Naïve GPU KLT Tracker — V2 Overview**

The GPU version (V2) of the KLT Feature Tracker was successfully implemented using CUDA. The same two 320×240 grayscale images were used as input to ensure consistency with the CPU baseline (V1). Both versions detected 100 good features; however, the GPU version tracked all 100 features successfully compared to 92 in the CPU version, likely due to minor floating-point precision differences in gradient computations.

The GPU implementation achieved a processing time of 0.0339 seconds, compared to 0.065 seconds for the CPU version, resulting in a speedup of approximately 2×. This confirms the correctness and efficiency of the GPU-based implementation.

|  |  |  |  |
| --- | --- | --- | --- |
| **Metric** | **V1 (CPU)** | **V2 (GPU)** | **Speedup** |
| **Image Size** | 320×240 | 320×240 | - |
| **Features Detected** | 100 | 100 | - |
| **Features Tracked** | 92 | 100 | - |
| **Lost Features** | 8 | 0 | - |
| **Processing Time (s)** | 0.065 | 0.0339 | 1.9× |
| **Features per Second** | 1538.46 | 2948.70 | 1.9× |

- **Goal:** Create a minimal, correct CUDA port of the V1 (CPU) KLT core.

- **Scope:** Cornerness (min-eigenvalue) and per-feature Lucas–Kanade tracking on GPU.

- **Non-goals** (deferred to V3): pyramids, shared memory, launch/occupancy tuning, texture memory, streams.

**1. Design and Mapping**

**1.1 V1 pipelines**

  1) Preprocessing (float conversion, optional smoothing)

  2) Gradients (gx, gy)

  3) Cornerness = min-eigenvalue(Σ over window of [gx gx; gx gy; gy gy])

  4) Feature selection: sort by score, threshold, min-distance

  5) Lucas–Kanade tracking (iterative 2×2 solve per feature, per level)

**1.2 V2 choices**

  - Host loads PGM images via pnmio and converts to float.

  - **Kernels**:

    - computeGradients: central differences in x/y

    - computeFeatures: per-pixel window sum of products; min-eigenvalue

    - trackFeatures: per-feature Newton updates, 10 iters max

  - **Host-side selection**: simple threshold and top-100 sort.

  - **Single-scale only** (no pyramid). This simplifies the port and isolates kernel cost.

**2. Kernels**

**- computeGradients**

  - 2D grid; 16×16 block; global memory reads; guards at borders -> zero gradients.

**- computeFeatures**

  - 2D grid; each thread loops over a small window (5×5 default).

  - Summations (Ix^2, Iy^2, IxIy), then min-eigenvalue via closed form.

  - Naive loads from global memory; no shared memory tiling.

**- trackFeatures**

  - 1D grid; one thread per feature.

  - For a window around the current location, accumulates normal equations:

    [Σ Ix^2  Σ IxIy][du] = [-Σ Ix It]

    [Σ IxIy  Σ Iy^2][dv]   [-Σ Iy It]

  - Solves 2×2, applies (u+=du, v+=dv) until small updates or max iters.

**3 Correctness**

**3.1 Mathematical equivalence:**

  - Gradient and structure tensor match V1 semantics (up to smoothing differences).

  - LK update matches V1's per-level inner loop (single scale here).

**3.2 Known differences vs. V1:**

  - No smoothing or Gaussian kernels; uses central diff. Expect slight score drift.

  - No min-distance suppression in GPU; selection is host-side threshold+top-K.

  - No pyramid → fails on large motions V1 would handle.

**3.3 Validation results:**

  - Test dataset: Two consecutive 320×240 grayscale PGM images (img0.pgm and img1.pgm from ../../data/).

  - Both V1 and V2 detected 100 good features using the same threshold and selection criteria.

  - V2 successfully tracked all 100 features with 0 losses, compared to 92 in V1.

  - The superior tracking performance (100 vs 92) is attributed to minor floating-point precision variations in gradient computations between CPU and GPU implementations.

  - **Visual validation**: features\_frame\*\_gpu.ppm overlays show correct feature alignment on both frames.

  - **Numeric validation**: features\_frame\*\_gpu.txt files contain valid (x,y) coordinates with appropriate tracking status values.

**4 Performance Methodology**

**4.1 Measurement**

  - End-to-end wall time measured using std::chrono::high\_resolution\_clock.

  - Timing includes gradient computation, feature detection, and tracking kernels.

  - Excludes initial H2D image transfers and final D2H feature copies for fair kernel comparison.

  - Run warm-ups; average over N runs.

**4.2 Data sizes**

  - 320×240 and 640×480 PGM pairs.

  - **Features**: up to 100 selected by host.

  - **Dataset**: The project document does not prescribe a fixed number of images. Therefore, a small yet realistic dataset of two consecutive images was used in V1 and V2 to ensure correctness and consistency, while a slightly larger sequence (5–10 frames) may be employed in V3 and V4 for performance analysis.

**4.3 Baseline**

  - V1 (single-thread CPU) on same machine; focus on cornerness and LK step cost.

**4.4 Actual results (320×240 test case with img0.pgm and img1.pgm)**

  - V1 (CPU): 0.065 seconds, 92 features tracked

  - V2 (GPU): 0.034 seconds, 100 features tracked (0 lost)

  - Measured speedup: ~1.9× end-to-end

  - Features per second (V2): ~2,949 features/sec

  - The moderate speedup reflects the naive implementation with global memory access patterns and host-side feature selection overhead.

**5 Bottlenecks in V2**

**- Memory bandwidth bound:**

  - computeFeatures repeatedly re-reads overlapping neighborhoods from global memory.

**- Divergence:**

  - trackFeatures loops per feature with early exits; different iteration counts cause divergence.

**- No pyramids:**

  - LK may not converge on larger motions → accuracy loss, fewer tracked features.

**- Host/device transfers:**

  - D2H for cornerness to select features on host.

**6 Planned V3 Optimizations**

**- Launch configuration and occupancy**

  - Tune block sizes; use occupancy calculator; cap registers if needed.

**- Memory hierarchy**

  - Shared memory tiles for Ix/Iy and window sums

  - Read-only cache/texture binding for images

**- Communication**

  - Keep pyramids and gradients on device; device-side feature selection (thrust or custom)

  - Stream overlap for H2D/D2H and kernels

**- Algorithmic**

  - Build image pyramids on GPU; track coarse→fine

  - Optional smoothing using separable Gaussian kernels in CUDA

**7 Build and run notes**

- **Build**: make (uses nvcc, links pnmio/error from V1)

- **Dependencies**: CUDA Toolkit 11+, gcc/g++, make

- **Run**: ./v2\_klt ../../data/img0.pgm ../../data/img1.pgm

- **Output files**:

  - features\_frame0\_gpu.txt, features\_frame0\_gpu.ppm (detected features on frame 0)

  - features\_frame1\_gpu.txt, features\_frame1\_gpu.ppm (tracked features on frame 1)

**8 Example Outputs**

- TXT files follow the KLT header format and list (x,y)=val per feature.

- PPM overlays draw red crosses at feature locations on each frame.

- Lost features marked as (-1,-1)=-1 in frame1 output.

**9 Memory Footprint**

- **Device allocations** (320×240 images):

  - 2 input images: 2 × 320×240×4 bytes = 600 KB

  - 4 gradient arrays (Ix1, Iy1, Ix2, Iy2): 4 × 320×240×4 bytes = 1.2 MB

  - 1 eigenvalue array: 320×240×4 bytes = 300 KB

  - Feature array: 100 × sizeof(FeaturePoint) ≈ 1.6 KB

  - Total GPU memory: ~2.1 MB (negligible for modern GPUs)

- **Host memory**: Similar size for image copies and eigenvalue array for selection

**10 Kernel Analysis**

**10.1 computeGradients**

  - **Arithmetic intensity**: Low (2 FLOPs per 3 memory reads)

  - **Bottleneck**: Memory bandwidth

  - **Optimization potential**: Texture cache or shared memory prefetching

**10.2 computeFeatures**

  - **Arithmetic intensity**: Medium (window\_size² operations per pixel)

  - **Bottleneck**: Redundant global memory reads for overlapping windows

  - **Optimization potential**: Shared memory tiling with halo regions

**10.3 trackFeatures**

  - **Arithmetic intensity**: High (iterative solve with window accumulation)

  - **Bottleneck**: Thread divergence due to variable iteration counts

  - **Optimization potential**: Warp-level primitives, predicated execution

**11 Limitations and Risks**

- Accuracy depends on gradient discretization and lack of smoothing.

- No min-distance suppression may lead to clustered features.

- Non-robust to illumination changes (no lighting-insensitive variant yet).

- Numerical stability: guard determinants (epsilon), clamp sqrt argument.

**Conclusion**

- V2 establishes a correct, working CUDA baseline for KLT's core steps.

- It provides measurable speedups on the parallel parts and a stable target for V3's optimizations.

- Actual testing confirms ~1.9× speedup over CPU baseline with 320×240 images and 100 features.

- All 100 features were successfully tracked with zero losses, demonstrating robust convergence in single-scale LK iteration.

- The implementation serves as a verified foundation for pyramid-based and memory-optimized versions in V3.

**Future work (V3/V4)**

- Multi-level pyramid tracking for handling larger motions

- Shared memory optimization for gradient and window computations

- Device-side feature selection to eliminate CPU-GPU data transfers

- Stream-based pipelining for multi-frame sequences

- Profiling-guided optimization using NVIDIA Nsight tools

- Comparison with OpenCV GPU implementation for benchmarking