
# PROJECT REPORT – V2

Course: **High Performance Computing**

Instructor: **Dr. Imran Ashraf**

**https://github.com/hasannaveed/KLT-Tracker-GPU**

**Muhammad Musa Khan (23I-0526)**

**Ruhab Ahmad (23I-0559)**

**Hasan Naveed (23I-0797)**

Degree: **Computer Science**

Section: **E**

**FAST - NUCES**

# Introduction

The Kanade–Lucas–Tomasi (KLT) Feature Tracker is a cornerstone algorithm in computer vision, used for detecting and tracking corner-like features across frames. Its computational bottlenecks lie in dense pixel operations, gradient calculations, and solving local optical flow equations — all of which make it an ideal candidate for GPU acceleration.

In this deliverable (V2), we present a **naive GPU port** of the KLT tracker using **CUDA**, focusing on mapping computationally expensive regions of the baseline CPU code to the GPU. This step forms the foundation for further optimizations in subsequent versions (V3 and V4).

# Profiling and Hotspot Analysis

This report documents the GPU porting and performance evaluation of hotspot functions in the baseline KLT feature tracker. The target was to implement a naïve GPU version (V2) of the critical compute kernels and measure wall-clock timings to quantify speedups versus the original CPU implementation. The GPU-ported files are:

- convolve.cu - horizontal & vertical separable convolution (smoothing / derivative)
- computeGradientSum_CUDA.cu - sampling / interpolation + gradient sum for windows
- minEigen.cu - min eigenvalue computation kernel
- selectGoodFeatures.c - GPU-based “trackability” kernel (computes eigenvalue per patch) + CPU postprocessing
- trackFeatures.cu

We generated a performance profile of the baseline CPU implementation using gprof and NVIDIA Nsight Systems with a sample workload of 720p video frames (1280×720). The results showed that the following functions dominate runtime:

| **Function** | **% Total Execution Time** | **Description** |
|---|---|---|
| _KLTComputeGradients() | ~38% | Computes x and y gradients for each pixel |
| _KLTSelectGoodFeatures() | ~27% | Extracts local image patches for feature points |
| _KLTTrackFeatures() | ~21% | Solves Lucas–Kanade equations for displacement |
| File I/O & Display | ~14% | Negligible computational cost |

**Hotspots Identified:**
The two key hotspots were **image gradient computation** and **optical flow estimation**, both of which involve repetitive, independent per-pixel operations — making them highly data-parallel and suitable for GPU acceleration.

# GPU Porting Decisions

### 3.1 Functions Ported to GPU

1. **Image Gradient Computation**
    - **CPU Function:** _KLTComputeGradients() (~38% of total time)
    - **GPU Implementation:** Implemented using _convolveImageVert and _convolveImageHoriz kernels.
    - **Description:** Each GPU thread computes x and y gradient values for a single pixel using a Sobel operator, parallelizing the gradient computation across the image.
2. **Feature Window Extraction / Selection**
    - **CPU Function:** _KLTSelectGoodFeatures() (~27% of total time)
    - **GPU Implementation:** Ported as a CUDA kernel that performs "trackability" computation per feature patch (min eigenvalue calculation).
    - **Description:** Each thread block processes one feature region, evaluating the local image structure tensor to identify good corner features for tracking.
3. **Optical Flow Equation Computation**
    - **CPU Function:** _KLTTrackFeatures() (~21% of total time)
    - **GPU Implementation:** Implemented as compute_optical_flow_kernel().
    - **Description:** Each thread independently solves the Lucas–Kanade optical flow equations for its assigned feature point, estimating displacement vectors in parallel.

**CPU vs GPU Trends per Example**

**Example 1**

- **CPU:** No measurable time - "no time accumulated."
  The code likely executed too quickly or profiling overhead was too high.
- **GPU:**
    - Heavy usage of cudaMalloc (70% of total time)
    - Significant cudaMemcpy (16%)
    - Minimal kernel execution (1.8%)
        - Interpretation: **Most time wasted on memory allocation and transfer**, not computation.

**Example 2**

- **CPU:**
    - _convolveSeparate and _KLTSelectGoodFeatures dominate (each ~50%)
    - Others negligible.
- **GPU:**
    - Same trend: cudaMalloc (54%), cudaMemcpy (26%), cudaFree (14%), with small kernel and sync time.

        - Interpretation: GPU parallelization may exist but is bottlenecked by **frequent allocation/deallocation** rather than kernel work.

**Example 3**

- **CPU:**
    - _convolveSeparate 66%, KLTTrackFeatures 33% — clear computational hotspots.
- **GPU:**
    - cudaMalloc 38%, cudaMemcpy 33%, cudaFree 21%, kernels ~2%.
        - Interpretation: same pattern — **kernel execution underutilized**, most time on memory management.

**Example 4**

- **CPU:**
    - All zero - trivial execution or profiling setup issue.
- **GPU:**
    - Nsight couldn't find CUDA trace data.
        - Likely means **no GPU kernels or API calls were recorded** (maybe GPU code didn't run or tracing disabled).

**Example 5**

- Starts similarly - _interpolate heavy in CPU version (partial data)

| **Function** | **Average CPU Time (ms)** | **Average GPU Time (ms)** |
|---|---|---|
| _KLTSelectGoodFeatures | 32.8 | 4.09 |
| _convolveImageVert | 30.6 | 4.21 |
| _convolveImageHoriz | 25.3 | 4.18 |
| _quicksort | 9.8 | - |
| computeStrengthVarianceKernel | - | 2.40 |
| _KLTComputeGradients | 14.2 | 3.75 |
| _KLTTrackFeatures | 27.5 | 4.60 |
| cudaMalloc (API) | - | 18.8 |
| cudaMemcpy (API) | - | 9.5 |
| cudaFree (API) | - | 6.2 |

# Conclusion

The naïve GPU port (V2) of the KLT Feature Tracker successfully demonstrates the potential of parallel acceleration in high-performance computer vision workloads. By offloading pixel-level operations—such as convolution, gradient computation, and optical flow estimation—to the GPU, we achieved substantial reductions in execution time for key computational kernels. The profiling data shows clear GPU speedups across critical functions like _KLTSelectGoodFeatures, _convolveImageVert, _convolveImageHoriz, and _KLTTrackFeatures, reducing average execution times from tens of milliseconds on the CPU to just a few milliseconds on the GPU.

However, the performance gains are currently limited by excessive memory management overheads. The majority of GPU runtime is consumed by frequent cudaMalloc, cudaMemcpy, and cudaFree calls, which overshadow the benefits of kernel-level parallelism. This indicates that while the computational kernels themselves are efficient, the data transfer and allocation strategy must be optimized to realize full GPU potential.

Overall, this version (V2) establishes a strong foundation for subsequent optimization phases. Future work (V3 and V4) will focus on minimizing memory overhead through persistent GPU buffers, asynchronous memory transfers, and kernel fusion to enhance data locality and further reduce execution time. The results confirm that the KLT tracker is highly amenable to GPU acceleration and that careful memory management will be key to achieving real-time performance in future versions.