CS4110-High-Performance Computing (HPC) — CCP Deliverable 1 Report

Performance Analysis and Baseline Profiling of KLT Feature Tracker

Complex Computing Problem:

KLT Feature Tracker Acceleration on GPUs

Submitted by:

Muhammad Nasir Bilal	23I-0659
Ahmed Asim	23I-0070

Github: https://github.com/Nasir-Bilal/KLT-GPU-Accelerato

Instructor:

Dr. Imran Ashraf

Department of Computer Science

FAST National University of Computer and Emerging Sciences (NUCES) Islamabad, Pakistan

1. Introduction

The KLT (Kanade-Lucas-Tomasi) feature tracker is a fundamental computer vision algorithm used to detect and track distinctive points (features) across video frames. This project focuses on accelerating the KLT feature tracker using GPUs, with the goal of improving computational performance while maintaining correctness and accuracy.

Key points:

- Feature tracking detects corners and edges to follow motion in videos.
- KLT uses optical flow assumptions and a multi-scale pyramid approach for robustness.
- GPU acceleration aims to parallelize compute-intensive parts of the algorithm to exploit data-level parallelism.

Source Code Explanation

The V1 source code implements the sequential KLT feature tracker. The main workflow involves loading input frames, detecting strong corner features, tracking these features across frames using optical flow, and refining results with optional pyramid scaling.

Utility functions handle image I/O, memory management, and storing feature lists/tables. Profiling wrappers measure execution time to identify computational hotspots for GPU acceleration.

Examples:

- **Example 1:** Finds the 100 best features in a single image and tracks them to the next frame. Saves feature locations to text and PPM files, and prints them to the screen.
- **Example 2:** Similar to Example 1, but replaces any lost features in the second image to maintain a constant number of tracked features.
- **Example 3:** Tracks 150 features across multiple frames (up to 10). Stores results in a feature table and writes each frame’s feature list to PPM files. Sequential mode speeds up processing.

- **Example 4:** Demonstrates reading a feature table from a file, modifying features across frames, and writing updated tables to text files. Also shows feature history extraction.

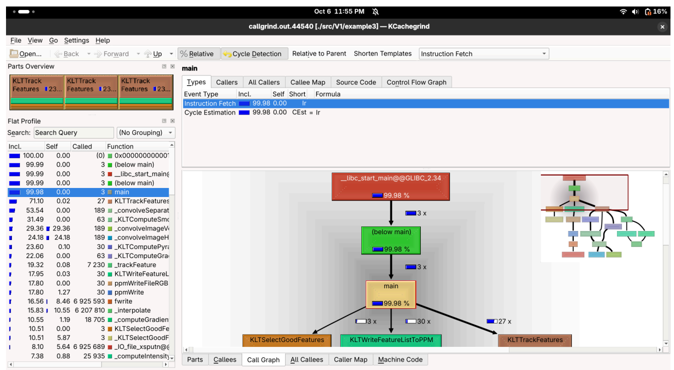
- **Example 5:** Shows manual adjustment of tracking parameters (e.g., minimum distance, window size, pyramid levels) to tweak the tracker's behavior and accuracy.

2. Profiling and Hotspot Identification

For profiling we are using 2 tools.

1. **gprof** identifies CPU hotspots and function-level bottlenecks.

2. **KCacheGrind** helps visualize complex call relationships and pinpoint parallelizable code.



Together, these tools allow us to analyze the code from multiple perspectives: **where the time is spent, how functions interact, and how efficiently computations are performed**.

So here is the flat profile generated using gprof.

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 45.45      0.05     0.05       63     0.79     0.79  _convolveImageHoriz
 27.27      0.08     0.03  2069270     0.00     0.00  _interpolate
  9.09      0.09     0.01     6235     0.00     0.00  _computeGradientSum
  9.09      0.10     0.01       63     0.16     0.16  _convolveImageVert
  9.09      0.11     0.01        1    10.00    12.86  _KLTSelectGoodFeatures
```

Observations:

- The function `_convolveImageHoriz` alone accounts for approximately 45% of the total execution time, making it the primary hotspot of the application.
- The `_interpolate` function follows as the second heaviest contributor, consuming around 27% of runtime.

- While the remaining functions are less significant individually, they still contribute non-trivially to execution time.

Insight: Focusing optimization efforts on `_convolveImageHoriz` and `_interpolate` has the potential to yield the most substantial performance gains, potentially improving overall runtime by up to ~70%.

2. Calls vs time

- `_interpolate` is called 2,069,270 times. While the measured time per call is small, the aggregate cost is significant due to the call volume; therefore, optimizing the contexts that repeatedly invoke _interpolatemay produce larger overall speedups than micro-optimizing_interpolate() alone.
- _KLTSelectGoodFeatures is called once, total time 0.01 s → not frequent, so impact is limited, but minor improvements are applied.

3. Where GPU acceleration might help

Convolution functions (_convolveImageHoriz, _convolveImageVert) are data-parallel and good candidates for GPU offload. _computeGradientSum and _computeIntensityDifferenceLightingInsensitive also contain large numbers of similar windowed operations and can be parallelized on CPU (SIMD/OpenMP) or offloaded to GPU. Although _interpolate() is inexpensive per call, its very high call count means caller-level optimizations (caching precomputed interpolations, inlining, loop vectorization, parallelizing outer loops) can be more effective; GPU acceleration remains attractive, especially when kernel designs avoid redundant interpolations (or use hardware/texture interpolation).

4. Takeaways for optimization

Prioritize `_convolveImageHoriz` and `_convolveImageVert` for the largest gains. Also optimize `_KLTSelectGoodFeatures` and caller functions such as `_computeGradientSum` — e.g., cache windowed interpolations, inline `_interpolate()`, and apply SIMD/OpenMP — to reduce the cost of repeated `_interpolate` calls. Use GPU offload or SIMD where appropriate, and re-profile after each change to measure speedup and find new hotspots..

3. Amdahl’s-law

Formula:

Formula (single component):
Speedup = $1 / ((1 - P) + P / S)$
where P = fraction of work sped up, S = speedup factor for that part.

General form (multiple components):
Overall Speedup = $1 / \Sigma_i (f_i / s_i)$
where f_i = fraction of total time spent in component i ($\Sigma f_i = 1$), and s_i = speedup applied to component i (use $s_i = 1$ if unchanged).

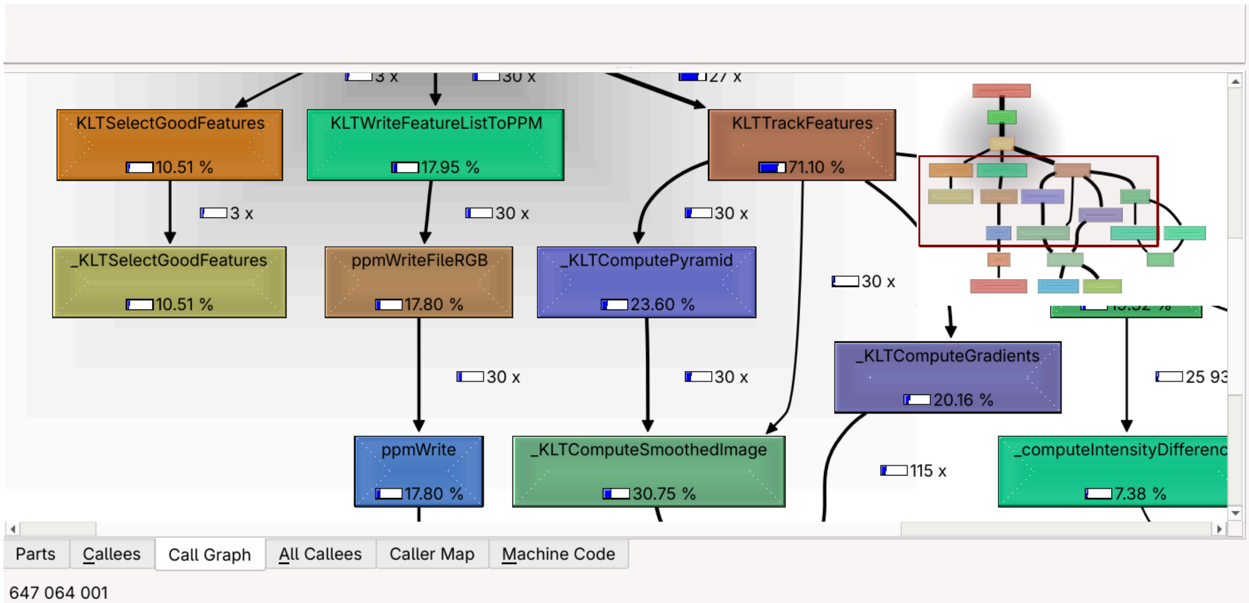
Baseline fractions (from gprof):

- `_convolveImageHoriz`: 45.45%
- `_interpolate`: 27.27%
- `_computeGradientSum`: 9.09%
- `_convolveImageVert`: 9.09%
- `_KLTSelectGoodFeatures`: 9.09%

Key scenarios (theoretical overall speedups):

- *ConvolveHoriz ×10* → **~2.11×** overall
- *Cache/interpolate effective ×5* → **~1.72×** overall
- *ConvolveHoriz ×10 + Interpolate ×5* → **~3.42×** overall

These are ideal upper bounds assuming perfect scaling and no extra overhead; always re-profile after implementing changes and confirm memory/accuracy tradeoffs.


3 x
30 x
27 x
KLTSelectGoodFeatures
10.51 %
KLTWriteFeatureListToPPM
17.95 %
KLTTrackFeatures
71.10 %
3 x
30 x
30 x
_KLTSelectGoodFeatures
10.51 %
ppmWriteFileRGB
17.80 %
_KLTComputePyramid
23.60 %
30 x
_KLTComputeGradients
20.16 %
30 x
30 x
ppmWrite
17.80 %
_KLTComputeSmoothedImage
30.75 %
115 x
_computeIntensityDifferenc
7.38 %
Parts
Callees
Call Graph
All Callees
Caller Map
Machine Code
647 064 001