PROJECT REPORT – V1

Course: **High Performance Computing**

Instructor: **Dr. Imran Ashraf**

GitHub link: Repository

Muhammad Musa Khan (23I-0526)

Ruhab Ahmad (23I-0559)

Hasan Naveed (23I-0797)

Degree: **Computer Science**

Section: **E**

FAST - NUCES

PROFILING AND HOTSPOT ANALYSIS

We compiled and executed the baseline implementation using the provided Makefile and generated execution profiles with **gprof** and **gprof2dot**. Profiling was carried out on realistic workloads using the provided test image sequences.

The results reveal the following major hotspots:

Function	**% Time Spent (approx.)**	**Description**
KLTTrackFeatures()	~45%	Core tracking routine – computes optical flow for each feature.
KLTComputeGradientImages()	~20%	Computes image gradients (Ix, Iy) required for optical flow equations.
KLTSelectGoodFeatures()	~15%	Detects corners based on the eigenvalues of the gradient matrix.
Memory Operations (copy/pyramid)	~10%	Includes building Gaussian pyramid and copying image patches.
Other overheads	~10%	Input/output, feature bookkeeping, and other minor operations.

Analysis:

- The majority of time is spent in **tracking features** (Lucas–Kanade optical flow computation).
- **Gradient computation** and **good feature selection** are also computationally intensive, but they are highly data-parallel and independent across pixels/features.
- Pyramid construction involves repeated image scaling and filtering, which is also parallelizable.

These profiling results guide us in deciding which parts of the algorithm should be ported to the GPU for maximum speedup.

FUNCTIONS SELECTED FOR GPU PORTING

Based on profiling and GPU suitability, the following functions will be ported:

1. KLTTrackFeatures() (Primary Target)

- **Why GPU?**
  - o This function processes thousands of features independently, solving small linear systems per feature using local image patches.
  - o Each feature can be handled by an independent CUDA thread or warp.
  - o The memory access pattern (patch extraction and gradient use) can benefit from **shared memory** to avoid redundant global memory fetches.

- **Expected Benefit:** Significant speedup as this is the most computationally expensive step (~45%).

2. KLTComputeGradientImages()

- **Why GPU?**
    - Gradient computation involves convolution-like operations across all pixels in the image.
    - This is a highly data-parallel task since each pixel can be processed independently.
- **Expected Benefit:** Efficient GPU memory coalescing can drastically improve throughput, reducing CPU load by ~20%.

3. KLTSelectGoodFeatures()

- **Why GPU?**
    - The eigenvalue calculation of the gradient matrix per pixel is embarrassingly parallel.
    - Can be mapped to GPU threads operating independently on pixels.
- **Expected Benefit:** Reduced CPU bottleneck during feature detection (~15%).

4. Image Pyramid Construction

- **Why GPU?**
    - Pyramid creation involves down sampling and Gaussian filtering, both of which are data-parallel.
    - GPUs are well-suited for convolution and image resizing.
- **Expected Benefit:** Moderate performance improvement, reduced memory overheads.

FUNCTIONS REMAINING ON CPU

Some operations will remain on the CPU:

- **Feature bookkeeping:** Managing active/inactive features, handling occlusion, and high-level decision-making are lightweight and not computationally intensive.
- **I/O operations:** Reading frames, writing outputs, and visualization are not bottlenecks.

This ensures that GPU resources are focused on heavy compute tasks while the CPU handles lightweight orchestration.

EXPECTED CHALLENGES AND CONSIDERATIONS

- **CPU–GPU Communication Overhead:** We must carefully manage data transfers between CPU and GPU to avoid negating performance gains. For example, we will keep image pyramids and gradients resident on the GPU across iterations.

- **Numerical Stability:** Floating-point precision on GPU may differ slightly from CPU results. Validation will be performed to ensure tracking accuracy.
- **Scalability:** Performance gains will depend on number of features and frame resolution. The GPU solution should scale better than CPU for larger workloads.
-

WORK DISTRIBUTION

1. Muhammad Musa Khan worked on the profiling tasks.
2. Ruhab Ahmad contributed by writing the README file and developing the Makefiles.
3. Hasan Naved made the initial commit and later worked on generating the PNG images.

MAKEFILE DESCRIPTION

This Makefile is designed to **compile, build, and profile** a C project involving the KLT feature tracker. It automates the process of compiling source files, creating a static library, building example executables, and generating profiling reports.

1. Compiler and Flags

- **Compiler:**
  CC = gcc: Uses GCC as the default C compiler.
- **Flags:**
  - -DNDEBUG disables assert() checks for optimized runs.
  - -pg enables profiling with gprof.
  - FLAG2 = -DKLT_USE_QSORT (commented by default) allows switching between custom quicksort and the standard qsort().
  - CFLAGS collects all flags for compilation.

2. Project Structure

- **Examples:**
  example1.c … example5.c are small test programs that demonstrate usage.
- **Library sources:**
  The variable ARCH lists the core implementation files (e.g., klt.c, trackFeatures.c, etc.).
- **Libraries:**
  LIB specifies system library paths (-L/usr/local/lib, -L/usr/lib) and math library (-lm).

3. Build Rules

- **General rule:**
  .c.o: compiles .c files into .o object files.
- **Static library (libklt.a):**
  - All object files from ARCH are archived into libklt.a.
  - This library is then linked when building the examples.

- **Example executables:**
  Each exampleN links against libklt.a with optimization (-O3) and profiling enabled.

4. Utility Targets

- **depend on:** regenerates dependency information (via makedepend).
- **Clean**: removes object files, libraries, executables, profiling outputs, and temporary files.

5. Profiling Support

- **profile:**
  - Runs each example with profiling enabled.
  - Generates gprof text reports (profiles/gprof_exampleX.txt).
- **all-graph:**
  - Converts gprof reports into **call graphs (PNG)** using gprof2dot and dot.
- **clean-profile:**
  - Cleans up profiling-related files (reports, graphs, gmon.out).

6. Summary

This Makefile:

- Compiles and links a **static library (libklt.a)**.
- Builds **five example executables**.
- Provides a **profiling workflow** using gprof and gprof2dot.
- Includes cleaning rules to keep the workspace tidy.

It's well-structured, separating compilation, building, and profiling into distinct targets, making it easy for developers to build and analyze the performance of the KLT tracker.

CONCLUSION

In summary, the profiling analysis of the baseline KLT implementation identified **feature tracking, gradient computation, feature selection, and pyramid construction** as major performance hotspots. These functions are computationally intensive yet highly parallelizable, making them ideal candidates for GPU porting.

By targeting these functions in **V2 (Naive GPU Implementation)**, we expect significant speedups compared to the CPU baseline. Further optimizations such as memory hierarchy usage, kernel launch configurations, and minimizing CPU-GPU communication will be explored in later versions (V3 and V4).

The private GitHub repository has been initialized with the baseline (V1) code, and commits will reflect contributions of each member as the project progresses.