Multi-thread programming on Raspberry Pi

1. Project Description

- **Objective:** The primary objective of this project is to gain hands-on experience with multi-threaded programming on embedded systems on Raspberry Pi. The project focuses on two multi-threading techniques: OpenMP and pthreads. These techniques will be applied to optimize in edge detection using the Canny edge detector.
- **Goals:**
    - Understand and apply OpenMP and pthreads to:
        - Modify the Gaussian blur algorithm in the Canny edge detection process to parallelize the computation in both the x- and y-directions.
        - Analyze the performance improvements.
    - Compare performance: After implementing multi-threading techniques using OpenMP and pthreads, performance comparisons will be made with the sequential code (without threading) to analyze improvements in execution time.
    - Apply multi-threading to a camera system: The project will extend the application of multi-threading to a camera-capturing-and-edge-detection system, implementing both OpenMP and pthreads optimizations to compare the effects on CPU usage and processing time.
- **Expected Observations:**
    - Performance improvements in the image processing tasks through parallelization using OpenMP and pthreads.

- A deeper understanding of how multi-threading can be used to optimize computationally intensive tasks on embedded systems like the Raspberry Pi.
- Insights into the advantages and limitations of using OpenMP versus pthreads in the context of real-time image processing.
- A working system capable of capturing images, processing them using the Canny edge detection algorithm, and displaying the results with optimized performance through multi-threading techniques.

- These results will be presented with performance benchmarks comparing execution times before and after applying multi-threading, along with CPU and wall-time metrics for the camera-based system

2. Experimental Setup

2.1 Methodology Overview

- The goal of this project was to implement and compare three versions of the Canny edge detection algorithm on a Raspberry Pi (or similar embedded system) in order to observe performance differences:
- **Single-Threaded Version (canny_local.c):**
  - Uses a straightforward Gaussian blur in the x and y directions sequentially.
  - Serves as a baseline for comparison.
- **OpenMP Version (canny_local_omp.c):**
  - Uses OpenMP directives (#pragma omp parallel for) to parallelize the x and y convolution loops.
  - Configurable thread count (4 in our tests).
- **Pthreads Version (canny_local_pthreads.c):**
  - Divides the x and y convolution loops among multiple pthreads.

- Configurable thread count (4 in our tests), with columns (for x-blur) or rows (for y-blur) partitioned among threads.

➔ In each of these versions, the main structure of the Canny pipeline remains the same (derivative computation, non-max suppression, hysteresis, etc.); only the Gaussian smoothing step is parallelized.

2.2 Code and Commands Used

a. Compilation

- Single-threaded:

  gcc canny_local.c -lm -o canny_local

- OpenMP version:

  gcc canny_local_omp.c -lm -fopenmp -o canny_local_omp

- Pthreads version:

  gcc canny_local_pthreads.c -lm -lpthread -o canny_local_pthread

b. Running the Programs

For all three versions, we used the same input image and the same parameters to keep the tests consistent. The `time` command is used to measure real (wall‑clock) time and CPU time.

- time ./canny_local test.pgm 1.0 0.2 0.6
- time ./canny_local_omp test.pgm 1.0 0.2 0.6
- time ./canny_local_pthread test.pgm 1.0 0.2 0.6

2.3 Camera Capture Integration

- Reuse `camera_canny.c` code from assignment 1b, and modify the `canny_util.c` following the adjustments made in `canny_local_omp.c` and `canny_local_pthreads.c`

- Integrate each of the modified `canny_util.c` variants into our camera application.
- Inside the code, the time was measured for three main stages:
  - Capturing the image from the camera.
  - Processing with the Canny edge detection (including the Gaussian blur).
  - Saving the output image or frame to storage.

  ➔ Allow comparisons as to how multi-threading affects the entire pipeline, not just the Gaussian blur.
- Thread Configuration: By default, 4 threads are used in the parallel implementations. This is set by either #define N_T 4 (in the pthreads version) or omp_set_num_threads(4) (in the OpenMP version).

3. Results

- **Step 1**
  - **Scope of OpenMP**
    - OpenMP is a shared-memory, directive-based parallel programming model for C, C++, and Fortran.
    - It uses #pragma omp directives to indicate which portions of code should run in parallel.
    - It handles thread creation, work sharing, and synchronization under the hood, allowing us to focus on which loops or code regions to parallelize.
  - **Advantages of using OpenMP**
    - Easy Integration: Simple compiler directives let us parallelize existing sequential code with minimal changes.
    - Shared-Memory Focus: It efficiently leverages multicore CPUs where all threads share the same memory.
    - Portability and Standardization: Supported by most compilers (GCC, Clang, Intel), making your parallel code highly portable.
    - Incremental Parallelization: We can gradually add directives to different parts of our program to optimize performance step by step.
- **Step 2**

Differences Between canny_local.c and canny_local_omp.c:

- OpenMP Header and Setup:
  - Added #include <omp.h> to utilize OpenMP functions and pragmas.
  - Introduced code to set the number of threads:

```
int n_thread = 4;
omp_set_dynamic(0);
omp_set_num_threads(n_thread);
```

- Parallelizing the Gaussian Smoothing in x-direction:
  - In the original gaussian_smooth, the x‐direction blur loop is sequential:

```
for(r=0; r<rows; r++){
 for(c=0; c<cols; c++){
  // compute dot, sum
 }
}
```

  - In canny_local_omp.c, that loop is wrapped in an OpenMP parallel directive:

```
#pragma omp parallel private(c, cc, dot, sum)
{
 #pragma omp for
 for(r = 0; r < rows; r++){
  for(c = 0; c < cols; c++){
   // compute dot, sum
  }
 }
}
```

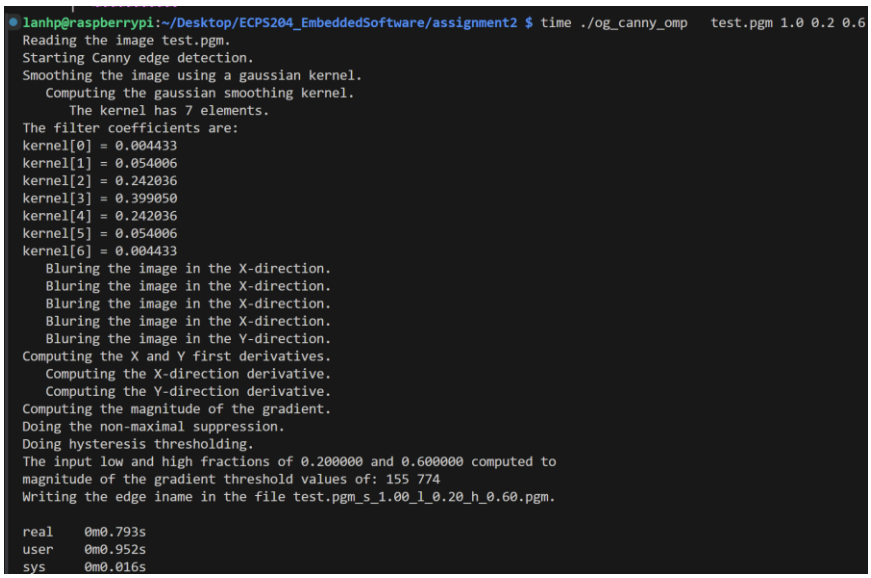
  ➔ This tells the compiler to divide the loop iterations among multiple threads.
- Parallelizing the Gaussian Smoothing in y-direction:
  - Similarly, a second #pragma omp parallel for is added around the y-direction loop, allowing it to run in parallel as well.

        - This again distributes iterations among the threads while each thread maintains its own local variables (dot, sum, etc.).
    - Private vs. Shared Variables:
        - Variables used independently within each loop iteration (e.g., dot, sum, c, cc, etc.) are declared as private.
        - Shared data (like the image array and kernel) remain visible to all threads.
    - No Other Changes to the Canny Pipeline: The rest of the Canny edge detection steps remain unmodified. Only the two main blur loops in gaussian_smooth() are parallelized.

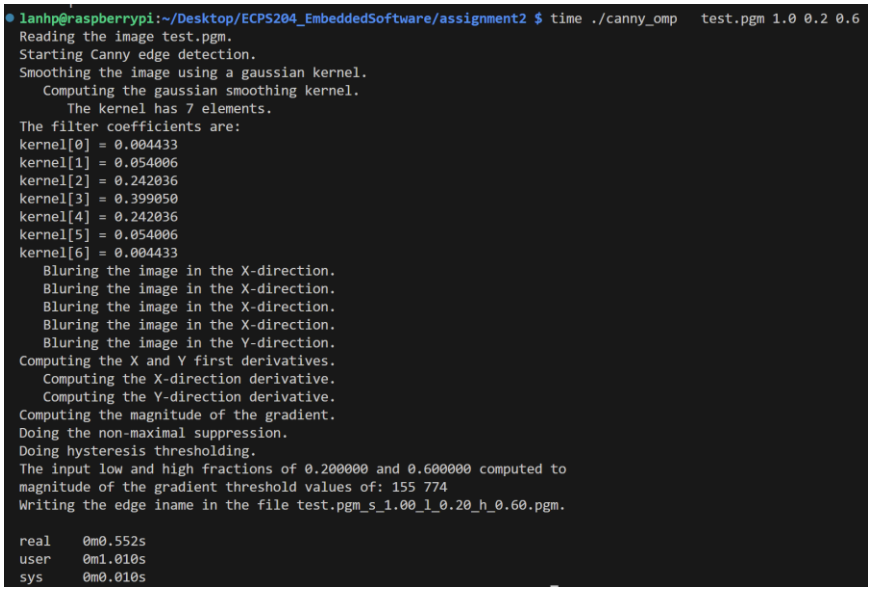
- **Step 3 + 4**

  Commenting on the Performance Improvement:

    - Parallel Speedup: Because the x - direction and y - direction blur loops in gaussian_smooth() are now parallelized, the workload is split among multiple CPU cores. This typically reduces the total runtime, especially for larger images where the smoothing stage dominates.
    - Overheads and Scalability: While there may be speedups, note that some overhead exists (thread creation, synchronization, etc.). Real speed gains depend on factors like image size, number of threads, and CPU core count.
    - Snapshot of Results:
        - Before adjusting blur y-direction:



        - After adjusting blur y-direction:



- **Step 5**
  - Scope of pthread
    - POSIX Threads (pthread): A standardized C library for creating and managing threads at a low level.
    - Shared Memory Model: Threads within a process share the same address space, allowing direct access to common data.
    - Fine - Grained Control: We can explicitly create, manage, synchronize, and destroy threads as needed.
    - Portability: Widely supported on Unix - like operating systems (Linux, macOS, etc.), following POSIX standards.
  - Advantages of Using pthread
    - Flexibility: We can tailor thread creation, scheduling, and synchronization precisely to your application's needs.
    - Potential Performance Gains: By distributing workloads across multiple CPU cores, many applications experience a decrease in execution time.
    - Low - Level Access: Because it's a thin wrapper around system calls, pthread gives us more direct control over thread behavior (compared to high - level APIs like OpenMP).
    - Portability & Standardization: Since pthreads follow POSIX, code using it can be recompiled on many different POSIX - compliant systems without major changes.

- **Step 6**
    - Thread Structures and Functions:
        - Added a struct (e.g., thread_args_x) containing pointers to shared data (image, kernel) and iteration bounds (col_s, col_e).
        - Created a new function, blur_x(), which runs the x-direction blur for a slice of columns.
    - Thread Creation and Joining:
        - In gaussian_smooth(), the code now creates multiple pthreads (e.g., pthread_create(&thread[i], ...))—each thread processes a distinct set of columns.
        - Calls to pthread_join() ensure the main thread waits for each worker thread to finish.
    - Partitioning the Work:
        - Instead of a single for loop over all columns, the columns are split among threads based on col_s and col_e.
        - This parallelizes the x-direction convolution loop across multiple CPU cores.
    - No Changes to Other Steps: Aside from the new pthread setup for the x-direction blur in the gaussian_smooth function, the rest of the Canny edge detection pipeline remains the same.
- **Step 7 + 8**
    - Commenting on Performance Improvement:
        - Threaded x-direction blur can often reduce total runtime by distributing columns among multiple threads.
        - The benefit depends on image size, number of available CPU cores, and thread overhead.
        - We typically see speedups proportional to the core count if the workload (e.g., the blur loops) is large enough to offset thread creation and synchronization costs.
    - Snapshot of Results:
        - Before adjusting blur y-direction:

```
lanhp@raspberrypi:~/Desktop/ECPS204_EmbeddedSoftware/assignment2 $ time ./og_canny_pthreads   test.pgm 1.0 0.2 0.6
Reading the image test.pgm.
Starting Canny edge detection.
Smoothing the image using a gaussian kernel.
   Computing the gaussian smoothing kernel.
      The kernel has 7 elements.
The filter coefficients are:
kernel[0] = 0.004433
kernel[1] = 0.054006
kernel[2] = 0.242036
kernel[3] = 0.399050
kernel[4] = 0.242036
kernel[5] = 0.054006
kernel[6] = 0.004433
   Bluring the image in the X-direction.
   Bluring the image in the X-direction.
   Bluring the image in the X-direction.
   Bluring the image in the X-direction.
   Bluring the image in the Y-direction.
Computing the X and Y first derivatives.
   Computing the X-direction derivative.
   Computing the Y-direction derivative.
Computing the magnitude of the gradient.
Doing the non-maximal suppression.
Doing hysteresis thresholding.
The input low and high fractions of 0.200000 and 0.600000 computed to
magnitude of the gradient threshold values of: 155 774
Writing the edge iname in the file test.pgm_s_1.00_l_0.20_h_0.60.pgm.

real    0m0.766s
user    0m0.832s
sys     0m0.020s
```

- After adjusting blur y-direction:

```
lanhp@raspberrypi:~/Desktop/ECPS204_EmbeddedSoftware/assignment2 $ time ./canny_pthreads   test.pgm 1.0 0.2 0.6
Reading the image test.pgm.
Starting Canny edge detection.
Smoothing the image using a gaussian kernel.
   Computing the gaussian smoothing kernel.
      The kernel has 7 elements.
The filter coefficients are:
kernel[0] = 0.004433
kernel[1] = 0.054006
kernel[2] = 0.242036
kernel[3] = 0.399050
kernel[4] = 0.242036
kernel[5] = 0.054006
kernel[6] = 0.004433
   Bluring the image in the X-direction.
   Bluring the image in the X-direction.
   Bluring the image in the X-direction.
   Bluring the image in the X-direction.
   Bluring the image in the Y-direction.
Computing the X and Y first derivatives.
   Computing the X-direction derivative.
   Computing the Y-direction derivative.
Computing the magnitude of the gradient.
Doing the non-maximal suppression.
Doing hysteresis thresholding.
The input low and high fractions of 0.200000 and 0.600000 computed to
magnitude of the gradient threshold values of: 155 774
Writing the edge iname in the file test.pgm_s_1.00_l_0.20_h_0.60.pgm.

real    0m0.679s
user    0m0.950s
sys     0m0.025s
```

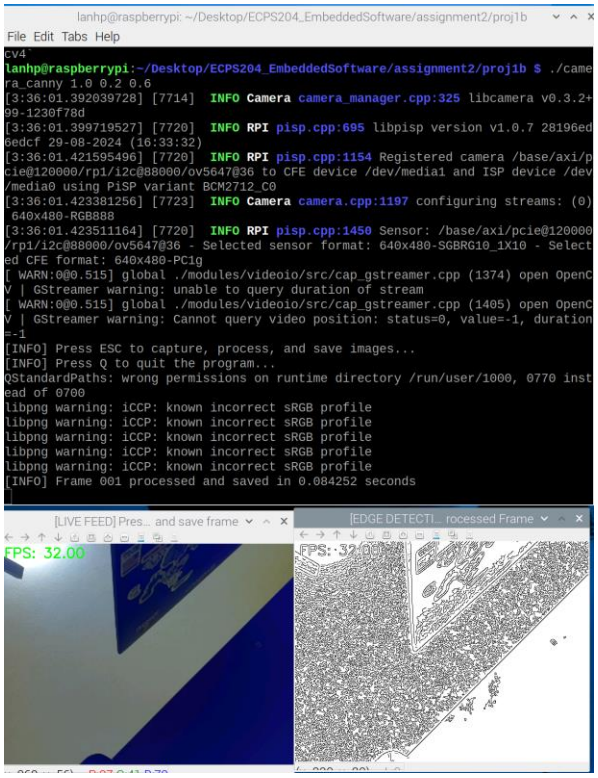
- **Step 9**
  - Below is a concise comparison of the no multi-threading, OpenMP, and pthreads results based on the timings provided:

Version	Real (Wall) Time	User Time	Sys Time
No Multi-Threading (canny_local)	0.947s	0.881s	0.009s
OpenMP (canny_omp)	0.569s	0.856s	0.020s
pthreads (canny_pthreads)	0.675s	0.836s	0.024s

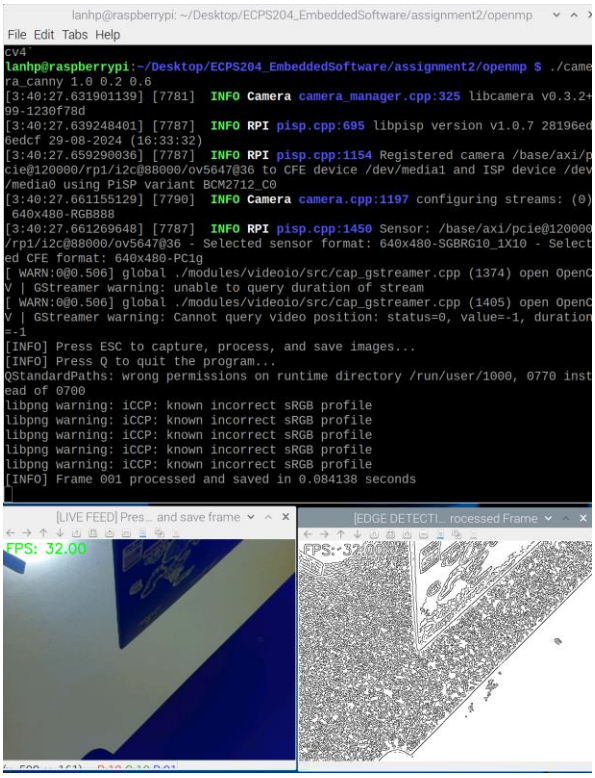
  - Observations and Analysis

- Faster Real (Wall - Clock) Time with Parallel Approaches: Both OpenMP (0.569s) and pthreads (0.675s) are noticeably faster than the sequential version (0.947s).
- OpenMP vs. Pthreads: In this particular test, OpenMP slightly outperforms pthreads in terms of wall - clock time (0.569s vs. 0.675s). This can vary depending on how the threads are scheduled and how the loops are partitioned.
- User Time vs. Real Time: With multi-threading, you often see higher user CPU time compared to the real (wall) time, because multiple cores are working simultaneously. The key metric for most applications is the real (elapsed) time, which shows how quickly the job finishes from start to end.

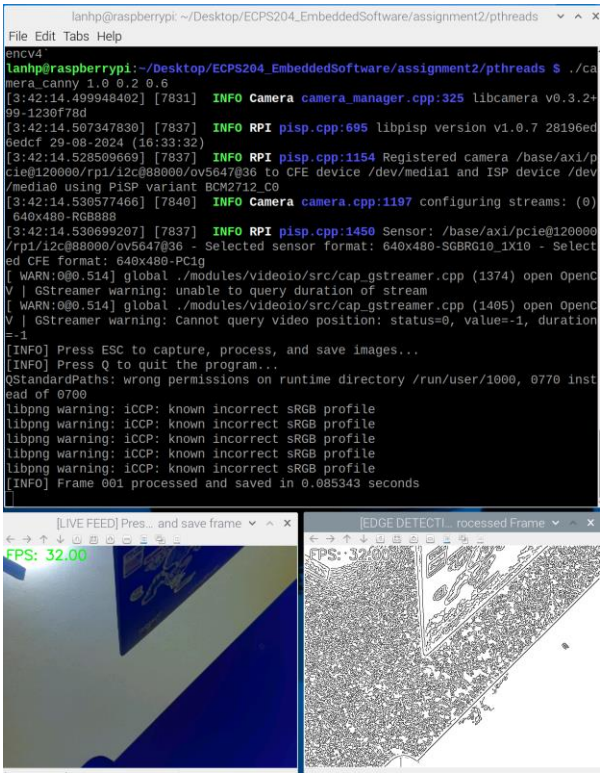
- **Step 10**
  - No multi-threading: Frame processed and saved in 0.084252 seconds



  - OpenMP: Frame processed and saved in 0.084138 seconds



- pthreads: Frame processed and saved in 0.085343 seconds



- Observations:

- Minimal Differences: The times are nearly the same across all three configurations—differences are in the order of milliseconds.
- Possible Reasons for Similar Times:
  - Overhead Masking Gains: The overhead of capturing the frame and writing it to disk may dominate the total runtime, overshadowing any speedups gained during the blur loops.
  - Small Image or Fast Hardware: If the captured frame is small or the CPU is relatively fast, the Gaussian smoothing step is short anyway, so parallelization yields only minor improvements.
  - Thread Overhead: Creating and synchronizing threads can also introduce overhead, balancing out potential speedups for small processing tasks.

➔ Even though the Canny edge detection can be parallelized, the entire camera pipeline's total runtime (capture + process + save) may not see a large net improvement.

4. Problems and Discussion

- **Expectations vs. Reality**
  - Results Alignment: In some cases, the performance gains from parallelization did not fully meet initial expectations, especially when the image size was relatively small or when camera capture and disk I/O dominated the total runtime.
  - Possible Overhead: Thread creation, synchronization, and limited parallel regions might reduce the net speedups. Additionally, overhead from capturing images and saving output can mask improvements in the computational stages.
- Reasons for Agreement or Disagreement
  - Threading Overhead: While multithreading improves performance on CPUbound tasks, overhead can dilute these gains if the task (Gaussian blur in this case) does not run long enough to offset the cost of thread management.
  - I/O Bottlenecks: In a camera capture pipeline, I/O operations (capturing frames and writing images to disk) might be the bottleneck, limiting the relative impact of parallelizing the blur.
- Methodology Improvements
  - Larger Test Images: Using higher‐resolution images or running many frames in batch mode could make the Gaussian blur computation more significant, highlighting multi‐threading gains.

    - Repeated Measurements: Running multiple trials under controlled conditions (minimal background processes, stable CPU temperature) provides more accurate and consistent timings.
    - Profiling Tools: Using profilers (e.g., gprof, perf) can help locate performance hotspots or confirm that I/O remains the bottleneck.
- External Sources and References
    - The primary references for pthreads and OpenMP were official documentation and tutorials, such as OpenMP.org and POSIX Threads (pthreads) Tutorials.