HW2 - Parallel 2D Convolution using Pthreads

Shiraz University

This assignment requires the **optimization of 2D image convolution using pthreads**. You must parallelize the convolution process over image regions or tiles and analyze performance using **gprof** and **perf**. Your goal is to explore **how thread count, scheduling, and data partitioning** affect performance and cache behavior.

Parallel Algorithms

Prof. Farshad Khunjush

Teacher Assistants

- **AmirMohammad Kamalinia**
- **Amirreza Rezvani**

Parallel 2D Convolution

In this homework, you are going to implement a **2D convolution operation** on an image using **pthreads**. You may use the provided base code that reads and writes PNG/JPEG images using the stb_image and stb_image_write libraries in pure C.

```
// convolve_stb.c
// Build: gcc -o convolve_stb convolve_stb.c -lm
// Usage: ./convolve_stb input.jpg output.png
// Requires stb_image.h and stb_image_write.h in same folder.

#define STB_IMAGE_IMPLEMENTATION
#include "stb_image.h"
#define STB_IMAGE_WRITE_IMPLEMENTATION
#include "stb_image_write.h"

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

// clamp helper
static inline unsigned char clamp8(int v) {
    if (v < 0) return 0;
    if (v > 255) return 255;
    return (unsigned char)v;
}

// get pixel with border replication
static inline unsigned char get_pixel(
    unsigned char *img, int w, int h, int c, int x, int y, int ch)
{
    if (x < 0) x = 0;
    if (y < 0) y = 0;
    if (x >= w) x = w - 1;
    if (y >= h) y = h - 1;
    return img[(y * w + x) * ch + c];
}
```

```
// The main convolution kernel application (kernel is kxk)
void convolve(
    unsigned char *in, unsigned char *out,
    int w, int h, int ch,
    const double *kernel, int ksize)
{
    int r = ksize / 2;

    for (int y = 0; y < h; ++y) {
        for (int x = 0; x < w; ++x) {
            for (int c = 0; c < ch; ++c) {
                double sum = 0.0;
                for (int ky = -r; ky <= r; ++ky) {
                    for (int kx = -r; kx <= r; ++kx) {
                        unsigned char p = get_pixel(in, w, h, c, x + kx
                        double kval = kernel[(ky + r) * ksize + (kx + r
                        sum += p * kval;
                    }
                }
                out[(y * w + x) * ch + c] = clamp8((int)round(sum));
            }
        }
    }
}
////////////////////////////////////////////////////////
int main(int argc, char **argv) {
    if (argc < 3) {
        fprintf(stderr, "Usage: %s input.jpg output.png\n", argv[0]);
        return 1;
    }

    int w, h, ch;
    unsigned char *img = stbi_load(argv[1], &w, &h, &ch, 0);
    if (!img) {
        fprintf(stderr, "Failed to load image %s\n", argv[1]);
        return 1;
    }
    printf("Loaded %dx%d (%d channels)\n", w, h, ch);

    unsigned char *out = malloc((size_t)w * h * ch);
```

```
    if (!out) { fprintf(stderr, "malloc failed\n"); stbi_image_free(img

    // Example Kernel (Change this based on the homework's descriptions
    double kernel[] = {
         -1, 0,  1,
         -2, 0, 2,
         -1, 0, 1
    };
    int ksize = 3;
    ////////////////////////////////////////////////////////////////////
    convolve(img, out, w, h, ch, kernel, ksize);

    // Write output as PNG (stb_image_write detects format from extensi
    if (!stbi_write_png(argv[2], w, h, ch, out, w * ch)) {
        fprintf(stderr, "Failed to write image %s\n", argv[2]);
    } else {
        printf("Wrote %s\n", argv[2]);
    }

    stbi_image_free(img);
    free(out);
    return 0;
}
```

Your task is to apply a convolution filter to an arbitrary image in parallel using multiple threads, and then analyze performance.

General Notes

In all cases, make sure to:

- Check the return code of `pthread_create` to ensure threads are successfully created.
- If your system reports a stack error, increase the pthread stack size before creation.
- Use synchronization primitives (such as mutexes or barriers) if needed to ensure the output image data is correct.

- Use **pthread** to parallelize the convolution process.

You must report **execution time** using `gprof`. Additionally, for deeper analysis, use **`perf`** or **similar tools** to record:

- CPU Cycles
- Instructions (and compute IPC = Instructions Per Cycle)
- Cache Misses
- L1 Data Cache Load Misses

Step 1: Image and Filter Setup

Start from a single reasonably large image that can show meaningful runtime differences. Suggested image size: **2048×2048**. Use two filter sizes:

- **3×3** (typical real-world convolution kernel)
- **31×31** (larger computational kernel for benchmarking)

> Note: In real-world image processing, small filters (3×3, 5×5, 7×7) are common, but large filters are occasionally used in research or simulation to study performance and cache behavior. Just make sure the size is adequate in order to see meaningful runtime differences.

Step 2: Parallelization and Thread Configuration

Use **pthreads** to parallelize your convolution. Each thread should handle a separate region of the image (e.g., a set of rows or tiles). Start with a baseline single-threaded version, then test **a few thread counts** (for example: 2, 4, and 8).

> Your goal is to find the **best thread configuration**, not to test every possibility. Discuss how you chose it — based on your CPU's core count, observed speedup, and overhead, and performance metrics.

Step 3: Loop Optimization

Experiment with

- **three different loop orderings**,
- two different **tiling** (e.g., tile sizes 8×8 and 16×16), and
- two different **loop unrolling** (e.g. 4 and 8).

Only keep configurations that show meaningful differences. Measure and report their performance metrics.

Step 4: Profiling and Analysis

Use `gprof` to measure total runtime, and use `perf` (or a similar tool) to record:

- CPU cycles
- Instructions (and calculate IPC = Instructions Per Cycle)
- Cache misses
- L1 data cache load misses

Discuss:

- How thread count affected performance
- How loop ordering or tiling improved (or worsened) results
- What performance bottlenecks you observed (CPU, memory, cache, etc.)

Report Format

Submit a complete report and your source code as a **ZIP file** to this Quera course by **2025/11/21**.

File Naming Convention

PA-F25-YOURFULLNAME-YOURSTUDENTNUMBER-HW2

Report Contents

1. Automation

- Explain your **Makefile** and how you used **Make** to automate and obtain results for each step.

2. Execution Instructions

- Provide an explanation of how to **compile and run** your code for each part (e.g., Linux command lines).
- Explain briefly the commands and related flags (inside your `Makefile`) you used (specially the compiler flags)

3. Performance Analysis

- Include a **complete analysis** of profiling results from **gprof** and **perf** for each step.
- Provide a **comparison** (preferably in tables and/or graphs. You can automate this part using a plotter script that extracts the desired metrics from the logs and visualizes them) of performance metrics for all tested parameters:
    - Thread counts
    - Kernel sizes
    - Image sizes
    - Loop orders
    - Tiling sizes
    - Unrolling factors

Identify and explain the **best-performing CPU implementation**.

- Include a **hardware analysis** explaining how your CPU characteristics affected profiling results.
    - Try to find meaningful relations between hardware and performance outcomes.

Notes To Consider

- You can use **AI as a tool**, but **DO NOT** use it blindly, you should be the **ENGINEER**, not AI :)
- **DO NOT** use AI to generate the full report.
- You can write your report in Farsi or English.
- Explain **any unusual patterns** in graphs and provide **justifications**.
- **Late submissions** will incur a penalty — please plan ahead to meet the deadlines.