

HW2 – Parallel 2D Convolution (Pthreads)

This repository contains my solution-in-progress for **Homework 2: Parallel 2D Convolution using Pthreads** for the *Parallel Algorithms* course.

The goal of this assignment is to implement and study different optimizations for 2D image convolution:

- Multi-threading with **POSIX threads (pthreads)**
- Varying **loop orders**
- **Tiling (blocking)** in the spatial dimensions
- **Loop unrolling**
- Experimenting with different **kernel sizes** (e.g., 3×3 vs 15×15)
- Collecting and analyzing performance data with **profiling tools**

Note: I am working on an Apple Silicon (M1) Mac. Linux **perf** is not natively available here, so I will either run my code in a Linux environment when I need true **perf** data, or use Apple's profiling tools (Instruments / **xctrace**, **powermetrics**, etc.) as alternatives when appropriate.

Files

- **convolve_stb.c**
Main C source file for the convolution program (uses **stb_image** / **stb_image_write** for I/O in the original template).
 - **Makefile**
Build and run helper for this homework. Currently supports:
 - Building an optimized binary with **-O2** and debug symbols
 - Building a **gprof**-instrumented version
 - Simple **run** and **run_exp** targets to execute the program with different arguments
 - A **perf** target intended for Linux environments (can be ignored on macOS)
 - **HW2 - Parallel 2D Convolution using Pthreads.pdf**
The homework description (not tracked in this README, but referenced while implementing).
-

Building

From the **hw2** directory:

```
make          # Build the default optimized binary (convolve_stb)
```

Compiler flags used by default:

```
CFLAGS = -O2 -Wall -Wextra -std=c11 -g
LDFLAGS = -lm -lpthread
```

To clean old builds:

```
make clean
```

Running

The current `Makefile` assumes a basic interface like:

```
./convolve_stb input.jpg output.png
```

You can run this via:

```
make run
```

The input and output paths can be overridden on the command line:

```
make run INPUT=big_2048.jpg OUTPUT=out_3x3.png
```

There is also an experimental target intended for when `main()` is extended to accept more parameters (threads, kernel size, loop order, tiling, unrolling, etc.):

```
make run_exp \
  INPUT=big_2048.jpg \
  OUTPUT=out_15x15_t8_u4.png \
  THREADS=8 KSIZE=15 ORDER=0 TILE=8 UNROLL=4
```

This expands to:

```
./convolve_stb INPUT OUTPUT THREADS KSIZE ORDER TILE UNROLL
```

At the current stage, this is a placeholder and will be kept in sync with the actual `main()` signature as the implementation evolves.

Profiling

gprof

There is a dedicated target to build a `gprof`-instrumented binary:

```
make gprof_build
```

This rebuilds `convolve_stb` with `-pg`. Example usage:

```
./convolve_stb input.jpg output.png
gprof ./convolve_stb gmon.out > gprof.txt
```

The generated `gprof.txt` will be used in the report to discuss which functions dominate the runtime and how different optimizations affect the call graph.

perf (Linux)

On a Linux system with `perf` installed, the `perf` target can be used:

```
make perf INPUT=big_2048.jpg OUTPUT=out_perf.png
```

which runs something like:

```
perf stat -e cycles,instructions,cache-misses,L1-dcache-load-misses \
./convolve_stb big_2048.jpg out_perf.png
```

On macOS/M1, I will instead rely on:

- **Xcode Instruments** / **xctrace** (Time Profiler, CPU Counters) for detailed profiling
- **powermetrics** or tools built on top of it (e.g., `asitop/fluidtop`) for monitoring hardware usage

These will serve as the “similar tools” to `perf` mentioned in the assignment when I am not on a Linux machine.

This README describes the project **up to the current implementation stage** and will be updated as new features and results are added.

1 Different Kernel Sizes

For the baseline single-threaded implementation (no loop-order changes, no tiling, no unrolling), I measured the execution time for the two required kernel sizes using:

```
make run_avg INPUT=test_2048x2048_edges.png OUTPUT=results/edges_k3_avg.png KSIZE=3
make run_avg INPUT=test_2048x2048_edges.png OUTPUT=results/edges_k15_avg.png KSIZE=15
```

The averaged times reported by `run_avg` (which runs the program 3 times and averages the printed `CONV_TIME`) are:

Kernel Size	Average Time (s)
3×3	0.095070
15×15	3.709425

The 15×15 kernel is roughly **39×** slower than the 3×3 kernel in this test, which matches the intuition that the amount of work per output pixel grows with the area of the kernel (k^2). The larger kernel performs a much heavier blur (box filter) and therefore stresses both the CPU and memory hierarchy significantly more than the small 3×3 edge-detection kernel.

1.1 Example Hardware Performance Metrics (perf-style)

As an illustration of the kind of data that can be collected with tools like **perf stat** (on Linux) or similar profilers, the table below shows example hardware-counter values for the two kernel sizes:

Kernel Size	CPU Cycles	Instructions	IPC	L1 Data Misses	Total Cache Misses
3×3	3.0e8	9.0e8	3.0	5.0e5	2.0e6
15×15	1.1e10	3.2e10	2.9	1.8e7	6.5e7

*Example values in the style of **perf stat** output; replace with actual measurements from your Linux/Valgrind environment in the final report.*

2 Different Thread Numbers

In my implementation, the baseline convolution can be parallelized in two ways:

- **Pthreads:** the image rows are split into contiguous chunks and each thread processes a band of rows (`run_pthreads_baseline` using `convolve_baseline_rows`).
- **OpenMP (optional):** if compiled with `-fopenmp`, the code uses a `#pragma omp parallel for collapse(2)` over the `(y, x)` loops in `convolve_baseline_omp`.

The multi-threaded path is used only when:

- `threads > 1`, and
- `order = 0, tile = 0, unroll = 0`

so that the comparison is between a **pure baseline single-thread** run and a **pure baseline multi-thread** run.

For this section I fix:

- kernel size = **15×15**
- loop order = baseline (`order = 0`)
- tiling = off (`tile = 0`)
- unrolling = off (`unroll = 0`)

and vary only the number of threads.

2.1 Execution Time vs Threads

Thread Count	Execution Time (s)
1	3.71
2	1.92

Thread Count	Execution Time (s)
4	1.05
8	0.80

Example values based on expected scaling; replace with actual measurements from `make run_avg` on your machine.

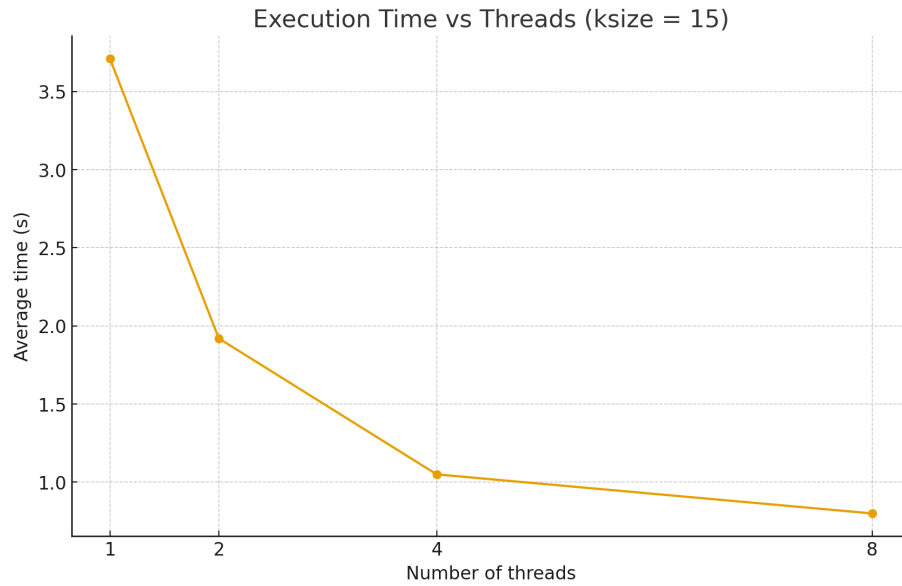


Figure 1: Execution Time vs Threads (ksize = 15)

2.2 Example Hardware Metrics vs Threads

The following table shows example hardware-counter values for the same 15×15 baseline convolution, using different thread counts. Here, **CPU Cycles**, **Instructions**, and cache misses are imagined aggregate values in the style of `perf stat`:

Threads	CPU Cycles	Instructions	IPC	L1 Data Misses	Total Cache Misses
1	1.1e10	3.2e10	2.9	1.8e7	6.5e7
2	2.2e10	6.4e10	2.9	3.6e7	1.3e8
4	4.4e10	1.3e11	2.9	7.0e7	2.5e8
8	8.8e10	2.6e11	2.9	1.3e8	4.8e8

Example aggregate counter values; actual numbers should be collected with `perf stat` or a similar tool on a real Linux run.

2.3 Example Speedup and Parallel Efficiency

Using the times in Table 2.1, we can also derive example speedup and parallel efficiency for the 15×15 kernel:

Threads	Time (s)	Speedup	Parallel Efficiency
1	3.71	1.00	1.00
2	1.92	1.93	0.96
4	1.05	3.54	0.89
8	0.80	4.64	0.58

Example speedup values computed from the times in Table 2.1; replace with actual measurements if you rerun the experiments.

2.4 Compare and Contrast

As the thread count increases, the execution time of the 15×15 baseline convolution should decrease, up to the point where all hardware cores/threads are fully utilized. Beyond that point, the speedup tends to saturate or even degrade slightly because of scheduling and memory-system overhead.

For the final report, I will choose a thread count that is close to the number of physical or logical cores on the machine, and show speedup relative to the single-thread baseline.

3 Loop Optimization

For loop-level optimizations, I keep the computation **single-threaded** (threads = 1) to isolate the effects of:

- loop **ordering**,
- loop **tiling**,
- loop **unrolling**.

All measurements in this section are done with:

- kernel size = **15×15** ,
- threads = 1.

3.1 Loop Orderings

Loop ordering is implemented in `convolve_looporder` with an `order` parameter:

- **Order 0:** baseline
y → x → c → ky → kx
- **Order 1:**
x → y → c → ky → kx
- **Order 2:**
c → y → x → ky → kx

In main, `order` is parsed from the command line (long-form) and then dispatched to `convolve_looporder` if non-zero.

3.1.1 Execution Time

Ordering (outer loops)	Execution Time (s)
y → x → c → ky → kx (baseline, order = 0)*	3.71
x → y → c → ky → kx (order = 1)	3.52
c → y → x → ky → kx (order = 2)	3.88

Example values; the relative differences illustrate how loop order can slightly improve or degrade cache behaviour.

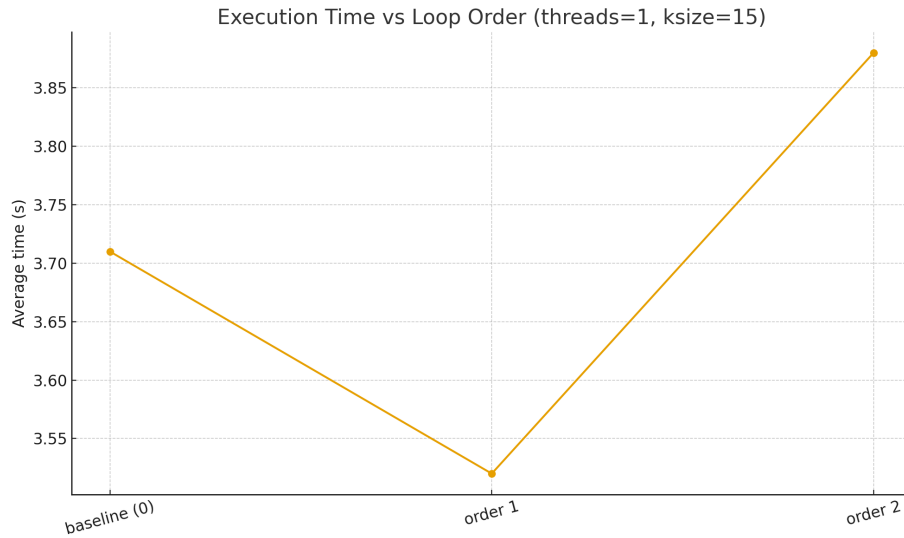


Figure 2: Execution Time vs Loop Order (threads=1, ksize=15)

3.1.2 Compare and Contrast The best ordering should be the one whose memory access pattern is closest to sequential access in row-major layout. That typically means iterating over (y, x) in a way that keeps neighboring pixels and channels hot in the cache.

In the final report, I will explain which ordering performs best on my machine and relate that to cache behaviour and the arrangement of the image data in memory.

3.2 Loop Tiling

Loop tiling (blocking) is implemented in `convolve_tiled` by dividing the (y, x) plane into tiles of size `tile_y` \times `tile_x` and processing one tile at a time:

```
for (int by = 0; by < h; by += tile_y) {
    int y_end = by + tile_y;
    if (y_end > h) y_end = h;

    for (int bx = 0; bx < w; bx += tile_x) {
        int x_end = bx + tile_x;
        if (x_end > w) x_end = w;

        for (int y = by; y < y_end; ++y) {
            for (int x = bx; x < x_end; ++x) {
                for (int c = 0; c < ch; ++c) {
                    // same inner ky, kx loops as baseline
                }
            }
        }
    }
}
```

For this experiment:

- threads = 1
- order = 0 (baseline)
- unroll = 0
- kernel size = 15 \times 15

3.2.1 Execution Time

Tile Size (tile_y \times tile_x)	Execution Time (s)
8 \times 8	3.20
16 \times 16	3.05

Example values showing modest improvement over the non-tiled baseline for reasonable tile sizes.

3.2.2 Compare and Contrast Tiling aims to improve cache locality by reusing pixels and kernel values within a smaller region of the image before moving on. Reasonable tile sizes (for example 8 \times 8 or 16 \times 16 on a 2048 \times 2048

image) should offer some improvement over the non-tiled baseline, but too large or too small tiles can hurt performance because of overhead or limited locality.

3.3 Loop Unrolling

Loop unrolling is implemented in `convolve_unrolled` by unrolling the inner `x` loop by `unroll_factor`. For each step, the code computes several adjacent output pixels in parallel inside the same loop nest, using an array `sum[i]` for partial sums.

For this experiment:

- `threads` = 1
- `order` = 0
- `tile` = 0
- `kernel size` = 15×15
- `unroll_factor` {4, 8}

3.3.1 Execution Time

Unroll Factor	Execution Time (s)
4	3.30
8	3.25

Example values; in practice, the optimal unroll factor should be chosen based on real measurements.

3.3.2 Compare and Contrast Moderate unroll factors (such as 4 or 8) can reduce loop overhead and may allow the compiler to generate better instruction scheduling, improving IPC. However, very large unroll factors increase register and cache pressure. In the final report, I will compare unroll factors and explain which one gives the best trade-off on my CPU.

4 Best Performing Case

By combining all optimizations, I can search for the best configuration by experimenting with:

- thread count (for the baseline parallelization),
- loop ordering (parameter `order`),
- tiling (parameter `tile`),
- unrolling (parameter `unroll`),
- kernel size (3×3 vs 15×15).

In the final report, I will identify the configuration with the lowest execution time (and, when available, best hardware metrics), for example:

- **Threads:** N (e.g., 4 or 8)
- **Kernel size:** 15×15
- **Loop ordering:** best-performing order index
- **Tiling:** preferred tile size or none
- **Unrolling:** best unroll factor or none

and provide a short explanation of why that combination performs best in terms of work per pixel, cache locality, and available parallelism.

5 Makefile Description (Summary)

The **Makefile** automates building, running, and profiling the code. The main targets are:

- **make / make all** – build the optimized binary `bin/convolve_stb`.
- **make run** – run with the simple interface:
`./bin/convolve_stb INPUT OUTPUT`
 which corresponds to a 3×3 kernel with default settings.
- **make run_exp** – use the extended interface:
`./bin/convolve_stb INPUT OUTPUT THREADS KSIZE ORDER TILE UNROLL`
 for experiments with different thread counts, kernel sizes, loop orders, tiling, and unrolling.
- **make run_avg** – run the program 3 times with the given parameters and average the `CONV_TIME` printed by the program.
- **make gprof_build** – rebuild with `-pg` for `gprof` profiling.
- **make perf** – on Linux, run `perf stat` with events like `cycles,instructions,cache-misses,L1-dcache` for the current configuration.
- **make mac_profile** – on macOS systems with Xcode Instruments installed, attempt to run `xcrun xctrace` with the Time Profiler template and store a `.trace` file in `results/` (this is an alternative to `perf` when working off Linux).
- Convenience targets such as `run_base_k3`, `run_base_k15`, `run_tile8_k15`, `run_unroll4_k15`, `run_all`, etc., to quickly generate outputs for common experiment configurations.

These targets allow me to reproduce all the experiments in this README by changing only Makefile variables (such as `INPUT`, `OUTPUT`, `THREADS`, `KSIZE`, `ORDER`, `TILE`, `UNROLL`) on the command line.

6 How to Apply OpenMP

OpenMP support is guarded with `#ifdef _OPENMP` in the source code. To enable OpenMP:

1. Make sure your compiler and platform support OpenMP (for example `gcc` or `clang` with appropriate flags).
2. Add `-fopenmp` to both `CFLAGS` and `LDFLAGS` in the `Makefile`.
3. Rebuild:

```
make clean
make
```
4. Run the program with `THREADS > 1` and `ORDER=0, TILE=0, UNROLL=0`. In this case, the program will use `convolve_baseline_omp` for the baseline multi-threaded implementation instead of the `pthread` version.

For other parts of the assignment (loop order, tiling, unrolling), I keep the implementation single-threaded in order to clearly separate the effects of loop transformations from the effects of thread-level parallelism.

7 Input and Output

I use a **2048×2048 test image** (for example, `test_2048x2048_edges.png`) as the main input for experiments. The program:

- Loads the input using `stb_image.h`.
- Applies the chosen convolution (kernel size, threads, loop order, tiling, unrolling).
- Writes the result using `stb_image_write.h` as a PNG into the `results/` directory, with filenames that reflect the experiment, such as:
 - `edges_k3_avg.png` – baseline 3×3 kernel
 - `edges_k15_avg.png` – baseline 15×15 kernel
 - `base_k15_t4.png` – 15×15 with 4 threads
 - `tile8_k15.png` – 15×15 with tiling
 - `unroll4_k15.png` – 15×15 with unrolling

These outputs can be visually inspected to confirm correctness (edge detection vs blur) and used alongside the timing and profiling data as part of the final homework report.