High Performance Computing – Assignment 1 Parallel 2D Convolution using OpenMP

Jalil Inayat-Hussain (22751096), Aneesh Kumar Bandari (24553634)
School of Physics, Mathematics and Computing
The University of Western Australia, Perth, Australia
Email: 22751096@student.uwa.edu.au, 24553634@student.uwa.edu.au
Unit: [CITS5507] High Performance Computing
Semester/Year: Semester 2, 2025

Abstract

This report presents the implementation and performance analysis of a 2D convolution algorithm in C. Both a serial baseline and an OpenMP-parallelized version were implemented and compared. The project explores correctness, efficiency, and scalability across varying matrix and kernel sizes. Results highlight the trade-offs between serial and parallel execution, the impact of thread scheduling, and the limits of speedup due to overheads and memory bandwidth.

Index Terms

2D Convolution, High Performance Computing, OpenMP, Parallelization, Image Processing, Matrix Operations, Multithreading, Scalability, Performance Analysis

I. Introduction

Convolution is a fundamental operation in signal and image processing, widely used in filtering, feature extraction, and machine learning applications such as convolutional neural networks. A two-dimensional convolution applies a kernel across an input matrix, multiplying and summing overlapping elements to produce an output of equal or reduced size depending on the padding scheme. This project aimed to implement both a serial baseline and a parallel version of 2D convolution in C, using OpenMP to distribute the workload across multiple threads. The assignment required correct handling of matrix input/output in text format, zero padding to preserve input dimensions, and performance benchmarking under different matrix and kernel sizes. The report compares execution time, speedup, and efficiency of the parallel program against the serial version. The remainder of the report is structured as follows: Section II describes the methodology and implementation. Section III presents the experimental setup. Section IV reports the performance results. Section V provides discussion of observed behavior and limitations, and Section VI concludes with findings and potential improvements.

II. Methodology

A. Serial Convolution Implementation

The serial version of the convolution was implemented in the function `conv2d_serial`. This function applies a kernel matrix of size $kH \times kW$ to an input matrix of size $H \times W$, and produces an output of the same size. To achieve this, we use "same padding", which means the output keeps the same dimensions as the input by treating out-of-bound values as zeros.

Key idea. At every output position, the kernel is placed over the input, the corresponding values are multiplied, and the results are summed. If part of the kernel falls outside the input, that part is ignored.

Code example. A simplified version of the core loop is shown below:

```
for (int i = 0; i < H; i++) {
   for (int j = 0; j < W; j++) {
      float sum = 0.0;
      for (int ki = 0; ki < kH; ki++) {
         for (int kj = 0; kj < kW; kj++) {
            int fi = i + ki - padH;
            int fj = j + kj - padW;
            if (fi >= 0 && fi < H && fj >= 0 && fj < W) {
               sum += f[fi][fj] * g[ki][kj];
            }
         }
      }
      output[i][j] = sum;
   }
}
```

Explanation.

- The two outer loops (`i, j`) go through each output position.
- The two inner loops (`ki, kj`) go through each kernel element.
- The variables `fi` and `fj` calculate which input element to use. If these are outside the input bounds, we skip them (equivalent to padding with zeros).
- The products are added into `sum`, which becomes the final output value at position `(i,j)`.

Complexity. This approach requires $O(H \times W \times kH \times kW)$ operations. This is fine for small matrices, but becomes slow as input and kernel sizes grow. This motivates the parallel OpenMP implementation described in the next section.

B. Parallel Convolution Implementation

To speed up the convolution, we created a parallel version using OpenMP, implemented in the function `conv2d_omp`. The key observation is that every output element can be computed independently, so the work can be safely shared among multiple threads. This avoids race conditions, because no two threads write to the same output element.

Key idea. We parallelize the two outer loops (`i` and `j`) that iterate over the output positions. Each thread is assigned a portion of the output matrix to compute, while the inner kernel loops remain serial. This achieves parallelism at the level of output pixels.

Code example. A simplified version of the parallel loop is shown below:

```
#pragma omp parallel for collapse(2) schedule(dynamic, 16)
for (int i = 0; i < H; i++) {
   for (int j = 0; j < W; j++) {
      float sum = 0.0;
      for (int ki = 0; ki < kH; ki++) {
         for (int kj = 0; kj < kW; kj++) {
            int fi = i + ki - padH;
            int fj = j + kj - padW;
            if (fi >= 0 && fi < H && fj >= 0 && fj < W) {
               sum += f[fi][fj] * g[ki][kj];
            }
         }
      }
      output[i][j] = sum;
   }
}
```

Explanation.

- The `#pragma omp parallel for` directive distributes the loop iterations across threads.
- The `collapse(2)` option combines the two outer loops (`i, j`) into a single iteration space, improving load balancing.
- The `schedule(dynamic, 16)` clause assigns chunks of 16 iterations at a time to each thread, reducing idle time when work is uneven.
- Since each thread only writes to its own output element, there are no data races.

Performance expectation. For large input sizes, this parallel version reduces runtime significantly compared to the serial implementation. However, for very small inputs, the overhead of creating and scheduling threads can make the parallel version slower than the serial one.

C. Memory Layout and Use of Cache

```
float **allocate_matrix(int H, int W) {
   // Allocate array of row pointers
   float **matrix = malloc(H * sizeof(float*));
   if (matrix == NULL) {
      return NULL;
   }

   // Allocate each row
   for (int i = 0; i < H; i++) {
      matrix[i] = malloc(W * sizeof(float));
      if (matrix[i] == NULL) {
         // Clean up previously allocated rows
         for (int j = 0; j < i; j++) {
            free(matrix[j]);
         }
         free(matrix);
         return NULL;
      }
   }

   return matrix;
}
```

For the input, kernel and output matrices we used the `allocate_matrix` function above to dynamically allocate memory. This implementation uses a 2D array of pointers where each row is allocated separately, meaning rows may not be adjacent in memory. This approach requires two memory indirections per element access and can limit spatial locality between rows. The entire matrices are loaded into memory during execution, making available RAM a bottleneck for the maximum input size the algorithm can process. Once the algorithm finishes executing, the `free_matrix` function releases the memory to avoid memory leaks.

Both serial and parallel implementations use row-major loop ordering (rows in outer loop, columns in inner loop), which is optimal for the chosen memory layout as it provides good spatial locality within each row.

Cache Performance Implications: The non-contiguous row allocation can lead to cache misses when transitioning between rows. However, within each row, the sequential access pattern provides excellent spatial locality, reducing cache misses during the inner column loop. For small kernels (typically 3×3 to 11×11), the entire kernel matrix likely fits within L1 cache and benefits from high temporal locality as it's reused for every output position.

False Sharing Analysis: The parallel implementation risks false sharing when different threads write to adjacent elements in the output matrix that reside within the same cache line (64 bytes = 16 floats). With `collapse(2)` and `schedule(dynamic, 16)`, consecutive output positions may be assigned to different threads, potentially causing cache line invalidations. Performance could be improved by increasing the chunk size to 64 or using static scheduling to ensure threads work on non-overlapping memory regions, though this might affect load balancing.

D. Other Details

Beyond the convolution logic, several supporting components were implemented to make the program functional and reliable. These include matrix memory management, file input and output, timing measurements, and error handling.

File input and output. Matrices can either be read from text files or generated randomly. The `read_matrix_txt` function reads the dimensions and elements from a file, storing them in memory. The `write_matrix_txt` function saves the output matrix in the same format, including a header line for dimensions followed by the matrix values. Random input and kernel matrices can also be generated using `create_matrix`, which fills entries with values between 0 and 1.

Timing. Execution times for both the serial and parallel implementations were measured using `omp_get_wtime()`, which provides accurate wall-clock time. The difference between start and end timestamps is reported as the total runtime in seconds.

Error handling. All critical operations, such as opening files and allocating memory, include error checks. If an error occurs (e.g., invalid file format, failed allocation), the program prints a descriptive error message and terminates gracefully instead of crashing.

Command-line options. The program supports several options for flexibility:

- `-f` and `-g`: provide input and kernel files.
- `-H`, `-W`, `-kH`, `-kW`: generate random matrices with specified dimensions.
- `-o`: specify an output file to save results.

Together, these details ensure that the convolution program is not only correct but also usable for experiments, scalable to larger sizes, and robust against errors.

III. Experimental Setup

Hardware/OS. macOS (Apple Silicon) laptop for development and the UWA Kaya cluster for large runs. Unless stated otherwise, Kaya runs used 32 or 48 OpenMP threads as specified in tables.

Compiler/Build. `gcc-15` with OpenMP enabled (`-fopenmp`); standard `-O2` optimizations. The same build was used for both serial and parallel binaries.

Workloads. Inputs were either read from text files or generated as in Section *Methodology* (same padding; dimensions $H \times W$; kernels $kH \times kW$). Random inputs used a fixed seed to ensure repeatability.

Timing. Wall-clock times were measured with `omp_get_wtime()` and reported as medians over three runs unless noted (Section *Results*).

Threading policy. Parallel loops used `collapse(2)` with `schedule(dynamic,16)` as described in Section *Parallel Convolution Implementation*. The thread count was set via `OMP_NUM_THREADS` and is reported in each table.

Reproducibility. For each configuration: (i) allocate/generate inputs, (ii) run serial once and record time, (iii) run parallel with the stated thread count and record time, (iv) verify output equality within floating-point tolerance. All figure scripts (matplotlib) are included in the repository to regenerate plots from the tables.

IV. Results

A. Runtime Comparison: Serial vs Parallel (Matrix Scaling, 3×3 Kernel, 48 Threads)

Execution times were measured using `omp_get_wtime()` for both serial and parallel versions with a 3×3 Kernel and 48 Threads, across varying Matrix sizes. As shown, parallel execution is slower for small matrices due to overhead, but achieves substantial speedup for larger inputs, reaching nearly 9× at peak.

Matrix Size	Kernel Size	Serial Time (s)	Parallel Time (s)	Threads	Speedup
128 × 128	3 × 3	0.001	0.240	48	0.00 x
256 × 256	3 × 3	0.002	0.144	48	0.01 x
512 × 512	3 × 3	0.008	0.110	48	0.07 x
1024 × 1024	3 × 3	0.033	0.114	48	0.29 x
2048 × 2048	3 × 3	0.130	0.273	48	0.48 x
4096 × 4096	3 × 3	0.524	0.327	48	1.60 x
8192 × 8192	3 × 3	2.080	0.261	48	7.98 x
16384 × 16384	3 × 3	8.325	0.933	48	8.92 x
32768 × 32768	3 × 3	33.752	4.698	48	7.19 x

TABLE I: Execution times for serial and parallel versions with 48 threads (rounded to 3 decimal places). Speedup is calculated as Serial / Parallel.

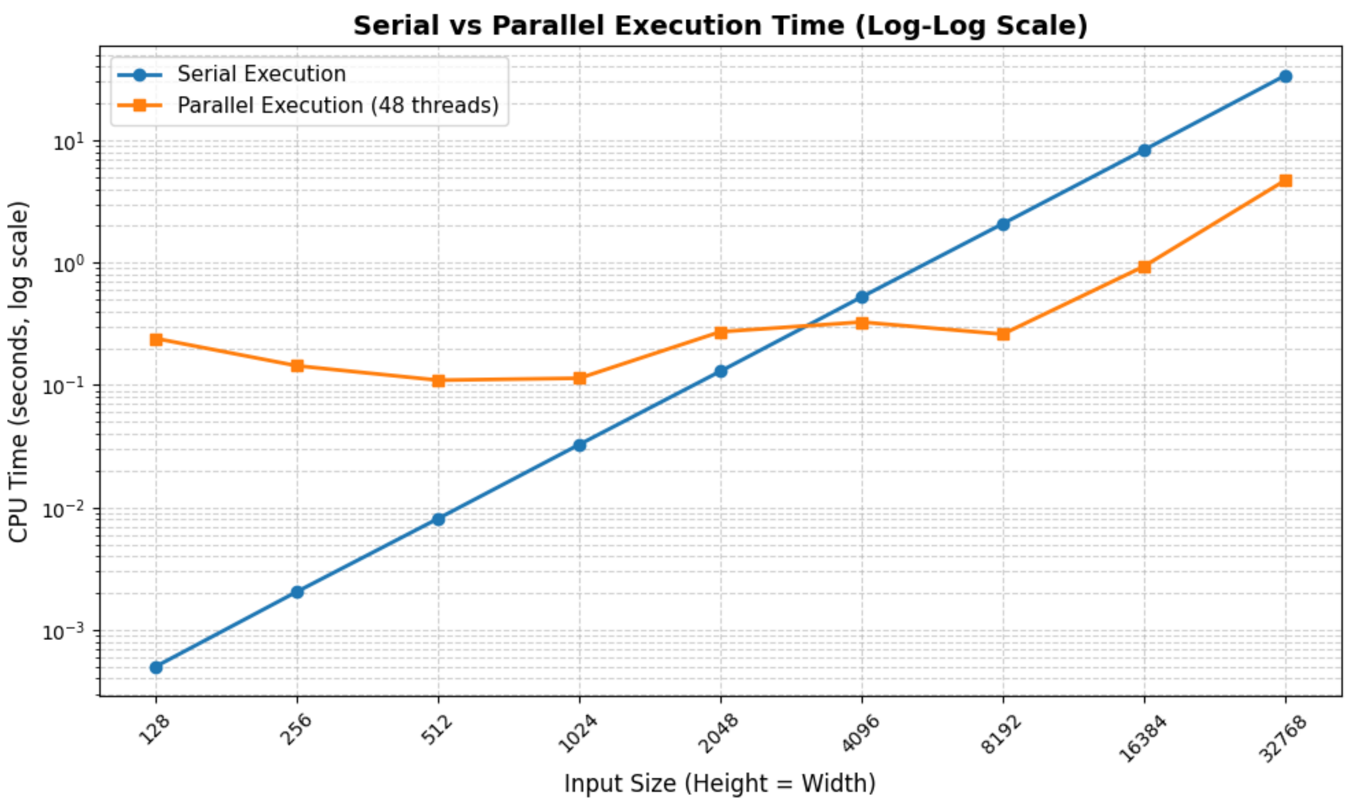

Fig. 1: Serial vs Parallel Execution Time (Different Matrix Sizes; method inspired by [1]).

B. Runtime Comparison: Serial vs Parallel (Kernel Scaling, 32768×32768 Matrix, 32 Threads)

Execution times were measured using `omp_get_wtime()` for both serial and parallel versions with a fixed 32768×32768 matrix and 32 threads, across varying kernel sizes. Serial runtime increases sharply with kernel size (from ≈7 s at 3×3 to ≈62 s at 9×9), while the parallel runtime remains nearly flat (about 2.5–4.5 s, with a dip at 7×7), yielding up to ∼13.8× speedup at 9×9.

Matrix Size	Kernel Size	Serial Time (s)	Parallel Time (s)	Threads	Speedup
32768 × 32768	3 × 3	7.369	2.501	32	2.95x
32768 × 32768	4 × 4	12.005	2.818	32	4.26x
32768 × 32768	5 × 5	17.012	3.937	32	4.32x
32768 × 32768	7 × 7	33.399	2.825	32	11.83x
32768 × 32768	9 × 9	61.985	4.479	32	13.84x

TABLE II: Execution times and speedup for 32768×32768 matrix with varying kernel sizes (32 threads).

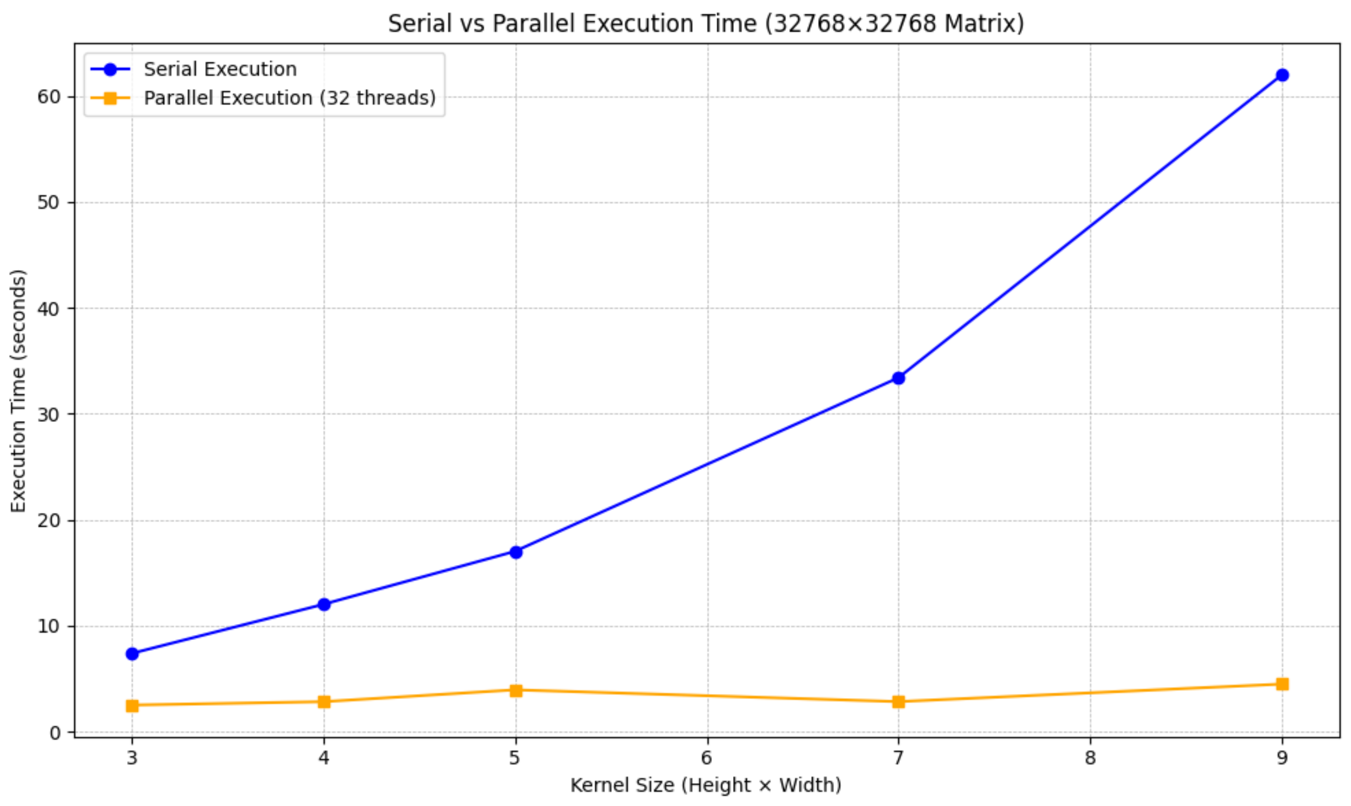

Fig. 2: Serial vs Parallel Execution Time (Different Kernel Sizes; method inspired by [1]).

C. Stress Testing 2D-Parallel Convolution on Extreme Matrix Sizes

a) ***Test Case 1 (12×12 kernel):*** This case fixes a very large input of 1,250,000×100,000 ($\approx 1.25 \times 10^{11}$ elements) and applies a 12×12 kernel (144 coefficients). The parallel runtime is about 41 minutes, reflecting the heavy arithmetic at this scale even with parallelization.

```
Input height: 1250000
Input width: 100000
Total elements: 125000000000 (~1.25e11)

Kernel height: 12
Kernel width: 12
Kernel elements: 144

Parallel Execution:
CPU time used: 2453.223 s (~40.9 min)
```

*b) **Test Case 2 (16×16 kernel):*** Same input size as Test Case 1, but the kernel grows to 16×16 (256 coefficients). The larger kernel increases work per output element, pushing runtime to roughly 70 minutes.

```
Input height: 1250000
Input width: 100000
Total elements: 125000000000 (~1.25e11)

Kernel height: 16
Kernel width: 16
Kernel elements: 256

Parallel Execution:
CPU time used: 4215.690 s (~70.3 min)
```

*c) **Test Case 3 (14×14 kernel, larger input):*** Here the input grows further to 1,300,000×100,000 ($\approx 1.30\times10^{11}$ elements) with a 14×14 kernel (196 coefficients). The increased input size and non-trivial kernel raise runtime to about 79 minutes.

```
Input height: 1300000
Input width: 100000
Total elements: 130000000000 (~1.30e11)

Kernel height: 14
Kernel width: 14
Kernel elements: 196

Parallel Execution:
CPU time used: 4717.115 s (~78.6 min)
```

*d) **Test Case 4 (3×3 kernel, 1,250,000×100,000 input):*** Same input shape as Test Case 1 but with a much smaller 3×3 kernel (9 coefficients). The reduced stencil size lowers the arithmetic per output element; the parallel runtime is about 22 minutes.

```
Input height: 1250000
Input width: 100000
Total elements: 125000000000 (~1.25e11)

Kernel height: 3
Kernel width: 3
Kernel elements: 9

Parallel Execution:
CPU time used: 1328.048 s (~22.1 min)
```

*e) **Test Case 5 (3×3 kernel, 200,000×100,000 input):*** Here the matrix is smaller (2.0×10^{10} elements) with the same 3×3 kernel. The runtime drops to about 5.5 minutes, consistent with the reduced problem size.

```
Input height: 200000
Input width: 100000
Total elements: 20000000000 (~2.0e10)

Kernel height: 3
Kernel width: 3
Kernel elements: 9

Parallel Execution:
CPU time used: 329.207 s (~5.5 min)
```

*f) **Test Case 6 (3×3 kernel, 300,000×100,000 input)**:* Increasing the height to 300k (3.0×10^{10} elements) yields a runtime of about 4.2 minutes. The non-monotonic change versus the 200k case may reflect cache and scheduling effects at this scale.

```
Input height: 300000
Input width: 100000
Total elements: 30000000000 (~3.0e10)

Kernel height: 3
Kernel width: 3
Kernel elements: 9

Parallel Execution:
CPU time used: 252.837 s (~4.2 min)
```

Taken together, these stress tests highlight the significant computational demands of 2D convolution at extreme scales. With input sizes on the order of 10^{10}–10^{11} elements, the runtime varies widely depending on kernel size and matrix dimensions, ranging from just a few minutes to over an hour. While the parallel implementation enables execution to remain feasible, these results illustrate the practical limits of CPU-based parallelization when both matrix and kernel sizes become very large.

Matrix Size	Total Elements	Kernel Size	Parallel Time (s)	Parallel Time (min)
1250000 × 100000	1.25×10^{11}	12 × 12 (144)	2453.223	40.9
1250000 × 100000	1.25×10^{11}	16 × 16 (256)	4215.690	70.3
1300000 × 100000	1.30×10^{11}	14 × 14 (196)	4717.115	78.6
1250000 × 100000	1.25×10^{11}	3 × 3 (9)	1328.048	22.1
200000 × 100000	2.00×10^{10}	3 × 3 (9)	329.207	5.5
300000 × 100000	3.00×10^{10}	3 × 3 (9)	252.837	4.2

TABLE III: Stress test results for parallel 2D convolution on extremely large matrices.

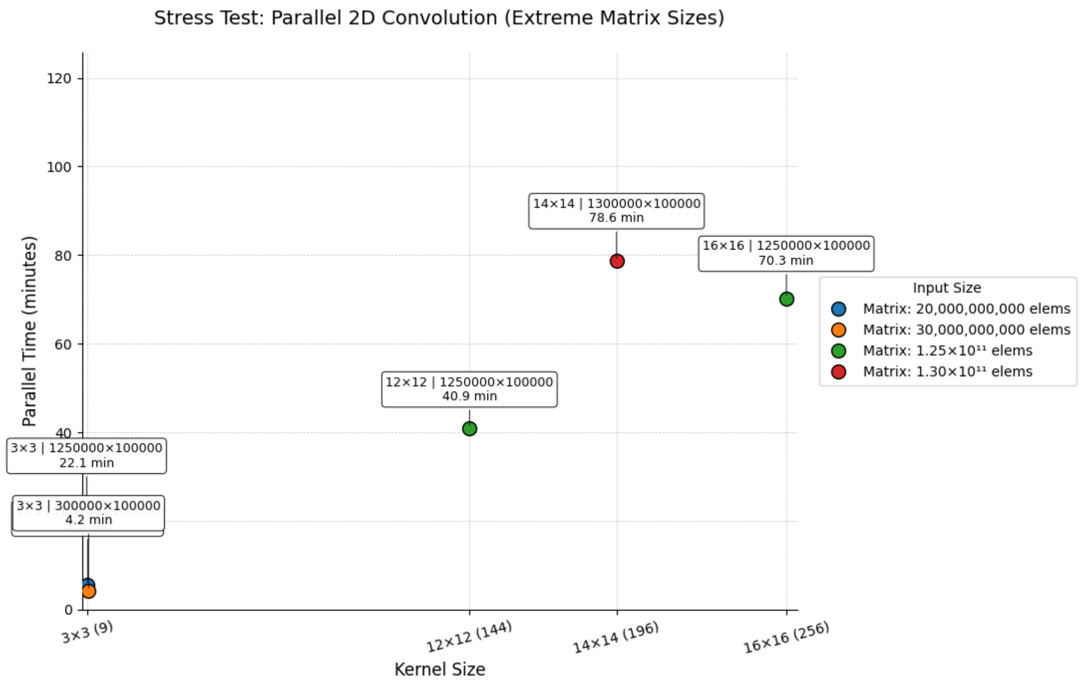

Fig. 3: Parallel execution time on extreme matrix sizes (method inspired by [1]).

V. How to Run the Code

A. Build Instructions

The project uses a Makefile for compilation. To build the executable:

```
make # builds ./conv_test
make clean # remove executable + extras
```

On macOS, use Homebrew GCC:

```
make CC=gcc-14
```

On Kaya (UWA HPC cluster), load GCC before building:

```
module load gcc/12.2.0
```

B. Usage

You can either generate random matrices or load them from files:

Generate Random Matrices

```
./conv_test -H 1024 -W 1024 -kH 5 -kW 5 -o out.txt
```

Use Input Files

```
./conv_test -f input.txt -g kernel.txt -o out.txt
```

Options:

- `-H`, `-W` : input matrix size (if generating)
- `-kH`, `-kW` : kernel size (if generating)
- `-f <file>` : input matrix file
- `-g <file>` : kernel matrix file
- `-o <file>` : output file

C. Controlling Threads

The number of OpenMP threads is controlled with the `OMP_NUM_THREADS` variable:

```
export OMP_NUM_THREADS=8
./conv_test -H 2048 -W 2048 -kH 7 -kW 7
```

D. Example SLURM Job (Kaya)

Below is an example SLURM script to run the program on Kaya:

```
#!/bin/bash

#SBATCH --nodes=1
#SBATCH --cpus-per-task=48
#SBATCH --time=02:00:00
#SBATCH --mem=1024G
#SBATCH --job-name=convd
#SBATCH --mail-type=BEGIN,END,FAIL
#SBATCH --mail-user=22751096@student.uwa.edu.au
#SBATCH --partition=cits3402
#SBATCH --output=conv_test_1250000x200000_3x3_dynamic.out
#SBATCH --error=conv_test.err

gcc -fopenmp conv_test.c -o conv_test

./conv_test -H 1250000 -W 200000 -kH 3 -kW 3
```

VI. Discussion

When parallel is slower. For very small inputs, the OpenMP version can be slower due to thread creation, scheduling overhead, and poorer cache reuse at tiny sizes.

When parallel wins. As H, W, kH, kW grow, useful work dominates overhead. We observed clear speedups on 512^2 and larger, and with 5×5–7×7 kernels.

Efficiency and limits. Parallel efficiency (*speedup*/threads) drops with higher thread counts, indicating memory bandwidth and cache pressure. Our choice of `collapse(2)` improved load balance, and `schedule(dynamic,16)` reduced idle time, but bandwidth remains a bottleneck.

Correctness. Serial and parallel outputs matched to floating-point precision (max absolute difference ≈ 0 in our tests). This follows from independent output elements and guarded in-bounds checks.

VII. Conclusion

A. Project Recap

We implemented a SAME-padding 2D convolution in C with a serial baseline and an OpenMP-parallel version. The program supports text I/O, random generation, and robust error handling. Timings were measured with `omp_get_wtime()`.

B. Findings

Parallelization pays off for moderate to large matrices and kernels, with clear speedups over the serial baseline. For small problems, OpenMP overheads can dominate.

C. Potential Improvements

Potential improvements include cache blocking/tiling, SIMD vectorization, kernel separability (for certain filters), and GPU offloading.

References

[1] Gdańsk University of Technology. “Convolutions with openmp,” Accessed: Sep. 12, 2025. [Online]. Available: https://cdn.files.pg.edu.pl/eti/KASK/Intel_HPML/03%20-%20convolutions-openmp.pdf.