Accelerating Convolution Kernels on Multi-Core CPUs using Parallel Programming

1st Belal Anas Awad
Computer and Systems Eng. Dept.
Faculty of Engineering, Ain Shams University
Cairo, Egypt
21P0072@eng.asu.edu.eg

2nd Mohamed Salah Fathy
Computer and Systems Eng. Dept.
Faculty of Engineering, Ain Shams University
Cairo, Egypt
21p0117@eng.asu.edu.eg

3rd Salma Mohamed Youssef
Computer and Systems Eng. Dept.
Faculty of Engineering, Ain Shams University
Cairo, Egypt
21P0148@eng.asu.edu.eg

4th Salma Hisham Hassan Wagdy
Computer and Systems Eng. Dept.
Faculty of Engineering, Ain Shams University
Cairo, Egypt
21P0124@eng.asu.edu.eg

***Abstract*—Convolution is a fundamental and computationally intensive operation widely used in computer vision and image processing. This project investigates the performance limitations of sequential convolution implementations and explores acceleration using multi-core CPU architectures. We employ profiling techniques to identify bottlenecks, and parallelize the core convolution loops, and conduct a systematic evaluation of the resulting speedup and scalability. The objective is to demonstrate significant performance improvements achievable through standard parallel programming techniques for this common computational kernel, contextualized within the principles of automatic parallelization analysis.**

***Index Terms*—Parallel Computing, OpenMP, MPI, Image Convolution, Image Processing, Performance Analysis, Multi-core CPU, Kernel Acceleration, Speedup, Scalability**

I. Introduction

Convolution is a foundational operation in image processing and computer vision, underpinning tasks such as edge detection, blurring, and feature extraction. It is also a core component in deep learning architectures. Despite its importance, convolution is computationally intensive, particularly for high-resolution and multi-channel images, due to its repeated access to neighboring pixels and large computational footprint. Sequential implementations of convolution, while straightforward, are inherently inefficient on modern multi-core CPUs. They underutilize the available parallel hardware, leading to increased processing time and energy consumption. This inefficiency arises from the inability of sequential code to exploit the concurrency offered by multi-core architectures. As a result, there is a pressing need for parallelization to fully leverage the computational power of modern hardware. Automatic parallelization offers a promising solution by transforming sequential code into parallel code with minimal programmer intervention. This process relies on static or dynamic code analysis to identify parallelizable sections, divide workloads, and coordinate execution across computing resources. However, automatic parallelization remains non-trivial due to challenges such as data dependencies, load balancing, and communication overhead [1]. To address these challenges, this project implements and compares three parallelization approaches for accelerating sequential 2D convolution kernels using: (1) MPI for distributed memory parallelization [2]. (2) Combining MPI with OpenMP to leverage both inter-process and intra-process parallelism, improving thread-level concurrency. MPI enables efficient communication and workload distribution across processes, while OpenMP facilitates shared-memory parallelism within nodes [3]. (3) Leveraging CUDA for GPU-based acceleration, capitalizing on massive thread-level parallelism available in modern GPUs [4]. We begin by profiling the sequential code to identify bottlenecks. Domain decomposition and thread parallelism techniques will then be applied, tailored to each architecture. Each parallelization approach will then be implemented and benchmarked to evaluate performance in terms of speedup, scalability, and efficiency. This work aims to provide insights into the trade-offs and benefits of distributed, shared-memory, and GPU-based parallelization techniques, contributing to the broader understanding of high-performance computing in image processing.

II. Related Work

- **SIMD and Cache Optimization [5]** Recent efforts focus on tailoring convolutions to cache hierarchies and vector units. For example, CSA (Convolution-specific Cache-blocking Analysis) and CSO (Convolution-Slicing Optimization) guide tiling decisions across cache levels, while micro-kernels target peak register performance—yielding up to 26% inference speedup on x86 and POWER10
- **Frequency and Algebraic Methods [6] FFT-based convolution**: Using fast Fourier transforms reduces computational complexity. On many-core CPUs, especially with NUMA-aware designs, FFT-based methods can outperform direct or GEMM-based approaches

Winograd algorithm: Optimized for small kernels (e.g., 3×3), Winograd reduces multiplications but complicates memory access. Some studies suggest FFT-based implementations can outperform Winograd when computation and memory costs are balanced .

III. Proposed Solution

A. MPI Implementation

1) Overview: The Message Passing Interface (MPI) is a standard for distributed-memory parallel programming, enabling efficient computation across multiple nodes in a cluster or supercomputer. In our project, the MPI-only implementation was chosen to address the limitations of single-node memory and to exploit the computational power of multiple machines. By decomposing the input image into blocks and distributing them among MPI processes, we can process very large images and scale the computation to a large number of cores.

A few key concepts are central to understanding our MPI-based approach:

- **Halos:** When an image is divided among processes, each process is assigned a block (submatrix) of the image. To correctly compute convolution at the edges of its block, a process needs pixel values from neighboring blocks. The extra rows and columns of data shared between neighboring processes are called *halos* (or ghost regions). Halos ensure that convolution results at block boundaries are accurate by providing the necessary border data from adjacent blocks.
- **Block Boundaries:** These are the edges of each submatrix (block) assigned to a process. Block boundaries are where data dependencies with neighboring blocks occur. Proper handling of block boundaries is essential, as convolution at these locations requires data from adjacent blocks, which is provided via halos.
- **Non-blocking Communication:** In MPI, non-blocking communication (such as `MPI_Isend` and `MPI_Irecv`) allows a process to initiate a data transfer and immediately continue computation without waiting for the transfer to complete. This enables overlap of communication and computation, improving overall performance. The process can later check or wait for the communication to finish, ensuring that all necessary data has been exchanged before proceeding to the next step.

The design uses a 2D block decomposition, where the image is divided into submatrices (tiles), each assigned to a process. This approach minimizes the communication perimeter relative to the area of each block, reducing the amount of data exchanged between processes. Each process is responsible for reading its assigned block from disk (using parallel MPI I/O), performing the convolution operation locally, and writing the result back to disk. To handle the dependencies at the boundaries of each block, processes exchange halo (border) data with their immediate neighbors (north, south, east, west) using non-blocking communication primitives (`MPI_Isend`, `MPI_Irecv`). Custom MPI datatypes are used to efficiently send and receive rows and columns, further optimizing communication.

2) Implementation Details: The implementation supports both grayscale and RGB images, with separate convolution routines for each. The convolution is performed in-place for a specified number of iterations (loops), and after each iteration, the halo regions are updated to ensure correctness. The use of non-blocking communication allows computation and communication to overlap, reducing idle time and improving overall efficiency. The code is structured to automatically determine the optimal block decomposition based on the number of processes and the image dimensions, balancing the workload and minimizing communication overhead.

3) Challenges and Solutions: One of the main challenges in the MPI implementation is managing data dependencies at block boundaries. This is addressed by exchanging halo data after each iteration. Another challenge is achieving good load balancing, which is handled by the block decomposition strategy. Communication overhead is minimized by using non-blocking communication and by carefully choosing the block sizes. The implementation is portable and can run on any system with an MPI library, making it suitable for a wide range of high-performance computing environments.

4) Performance and Scalability: The MPI-only implementation demonstrates good scalability for large images and a high number of processes. By distributing the workload and overlapping communication with computation, the implementation achieves significant speedup compared to the sequential version. The use of parallel MPI I/O further improves performance by allowing all processes to read and write data concurrently.

B. Hybrid MPI + OpenMP Implementation

1) Motivation and Design: While MPI is effective for distributed-memory parallelism, modern compute nodes often have many CPU cores that can be exploited using shared-memory parallelism. The hybrid MPI + OpenMP implementation combines the strengths of both paradigms: MPI is used for communication and workload distribution across nodes, while OpenMP is used to parallelize computation within each node. This approach maximizes resource utilization and reduces the number of MPI processes required per node, which in turn reduces communication overhead.

The image is divided among MPI processes as in the MPI-only version, but within each process, the main convolution loops are parallelized using OpenMP. The OpenMP `parallel for` directive, often with the `collapse` clause, is used to distribute the work among threads. OpenMP sections are also used to overlap computation and communication, allowing the process to continue computing while waiting for halo data from neighbors.

2) Implementation Details: Each MPI process is responsible for a block of the image and uses OpenMP to parallelize the convolution operation across all available CPU cores. The convolution routines for grayscale and RGB images are adapted to use OpenMP, with careful attention to data

dependencies and memory access patterns. Halo exchange is still performed using MPI, but the number of MPI processes per node is reduced, as each process can now utilize multiple threads. This hybrid approach is particularly effective on modern multi-core and many-core systems.

The implementation also uses OpenMP sections to overlap the communication of halo data with the computation of the interior of the block. This further reduces idle time and improves overall efficiency. The code is designed to be flexible, allowing the number of OpenMP threads to be set at runtime, and to automatically adapt to the available hardware.

3) Challenges and Solutions: The main challenge in the hybrid implementation is managing the interaction between MPI and OpenMP, particularly with respect to data dependencies and synchronization. Careful attention is paid to ensure that halo data is exchanged before the boundary regions are updated, and that threads do not interfere with each other's memory accesses. Load balancing is achieved by combining block decomposition (for MPI) with loop parallelization (for OpenMP). Communication overhead is further reduced by decreasing the number of MPI processes and by overlapping communication with computation.

4) Performance and Scalability: The hybrid MPI + OpenMP implementation achieves superior performance and scalability compared to the MPI-only version, especially on systems with many CPU cores per node. By fully utilizing both inter-node and intra-node parallelism, the implementation can process very large images efficiently and achieve high speedup. The reduction in communication overhead and the ability to overlap communication with computation make this approach well-suited for modern high-performance computing systems.

IV. Approaches Used

A. *CUDA Evaluation and Profiling*

Hardware Platform. The evaluation was performed on a laptop equipped with an *AMD Ryzen™ 9 5900HS* CPU and a discrete **NVIDIA GeForce RTX 3050 (6 GB GDDR6)** GPU. This combination supports efficient multi-threaded execution on the host and high-throughput parallel execution on the device. Key hardware specifications:

- **CPU:** 8 cores × 2 threads/core = 16 logical threads, up to 4.68 GHz
- **Cache:** 512 KB L1, 4 MB L2, 16 MB L3 (shared)
- **Memory:** 16 GB DDR4 @ 3200 MHz, dual-channel
- **GPU:** NVIDIA RTX 3050 Laptop GPU, 2048 CUDA cores, 6 GB GDDR6, boost clock 1740 MHz

Software Stack.

- **OS:** Ubuntu 24.04.2 LTS, Kernel 6.11.0
- **Compiler:** GNU GCC 13.3.0
- **CUDA Toolkit:** 12.9 with Nsight Compute 2025.2.0
- **Profiler:** `ncu (Nsight Compute)` for GPU kernel-level analysis

Parallel Convolution using CUDA. The input image is divided into grids of thread blocks. Each thread is assigned to compute a single output pixel by applying a convolution mask to a neighborhood in the input image. To reduce latency from global memory, shared memory is used to cache overlapping image tiles per block. This minimizes redundant memory fetches and increases performance via data reuse.

CUDA Execution Model. In CUDA's heterogeneous model, the CPU (host) handles memory allocation and kernel invocation, while the GPU (device) performs actual computation across thousands of threads in parallel. Threads are grouped into warps, which are scheduled on Streaming Multiprocessors (SMs).

Profiling with Nsight Compute. The kernel was profiled using the command below, generating a '.ncu-rep' report for detailed GPU performance insights:

[language=bash, caption=Nsight Compute Profiling Command] sudo /usr/local/cuda-12.9/bin/ncu –set full –target-processes all –force-overwrite –export ./results/prof$_{run.ncu-rep}$ ./cuda$_{c}$onvinput.raw19202520100grey

Metrics and Visual Analysis.

Figure 1 shows the distribution of active cycles across SMs, L1/L2 caches, and DRAM, indicating efficient utilization of compute resources. Figure 2 highlights that Compute Throughput reaches nearly 89% of theoretical peak, while memory throughput is comparatively lower (26%), suggesting a compute-bound workload.

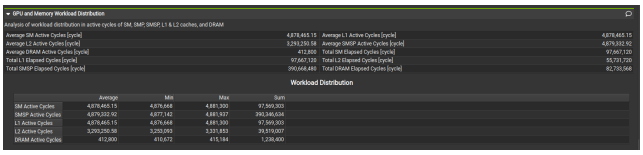

Fig. 1. GPU and Memory Workload Distribution across Active Cycles

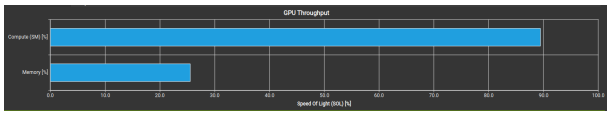

Fig. 2. Compute vs Memory Throughput as Percentage of Speed-of-Light

Warp Behavior and Execution Bottlenecks.

Figure 3 presents the breakdown of warp stalls. The most dominant stall reason was *Tex Throttle*, followed by *Short Scoreboard*, both indicative of memory-related latency. Figure 4 further shows that the FP64 pipeline is over-utilized (89.5% active cycles), potentially introducing bottlenecks.

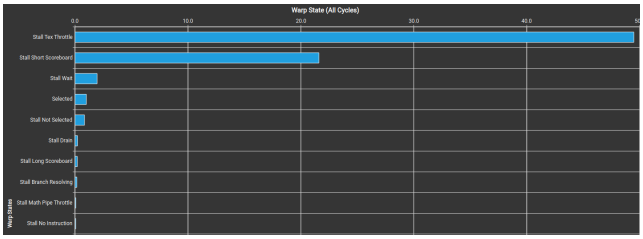

Fig. 3. Warp State Distribution during Kernel Execution

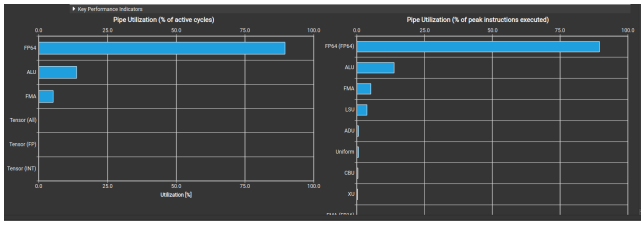

Fig. 4. Pipe Utilization: FP64 Dominance and ALU/FMA Usage

Memory Access and Cache Efficiency.

Memory flow visualization in Figure 5 shows strong L1/L2 cache performance with hit rates of 96% and 98% respectively. Figure 6 shows that 94% of L2 requests are uncoalesced, indicating room for improving memory access patterns via coalescing.

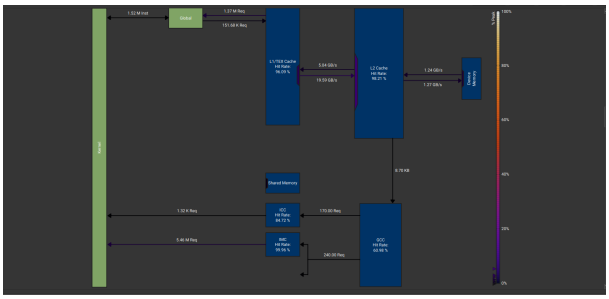

Fig. 5. Memory Access Pattern and Cache Hit Rates

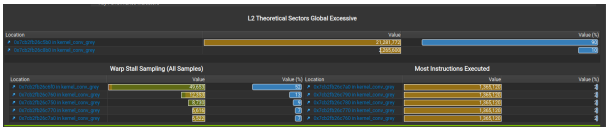

Fig. 6. L2 Cache: Global Access Sector Overuse and Warp Stalls

Occupancy and Shared Memory Optimization.

Nsight's occupancy graphs (Figures 7 and 8) show that using fewer registers and shared memory per block improves occupancy. The configuration around 256 threads per block with modest shared memory usage achieves optimal utilization.

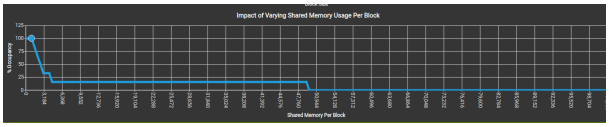

Fig. 7. Occupancy vs Shared Memory Usage per Block

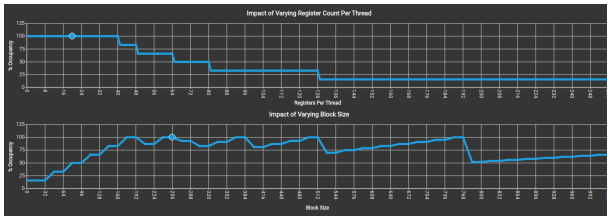

Fig. 8. Occupancy vs Block Size and Register Usage

Performance Summary.

The CUDA kernel demonstrated a speedup of over 11× compared to the sequential CPU implementation:

$$\text{Speedup} = \frac{3100 \text{ ms}}{265 \text{ ms}} \approx 11.7\times$$

TABLE I
PERFORMANCE COMPARISON: CPU VS CUDA

Configuration	CPU Time (ms)	CUDA Time (ms)	Speedup
1920×2520 Image	3100	265	11.7×

Overall, CUDA and Nsight Compute provided rich diagnostics, and the profiling insights suggest future directions for optimizing memory access patterns, reducing warp stalls, and increasing scheduler issue efficiency.

B. Multi-Core CPU

*1) **Weak Scaling**:* **A. Weak Scaling Analysis** Weak scaling maintained constant workload per processing unit by proportionally increasing image size with N threads/processes for MPI-only (mpi_conv) and hybrid OpenMP+MPI (parallelized) versions [7].

1. Aggregate Weak Scaling Performance Metrics (Fig. 9)

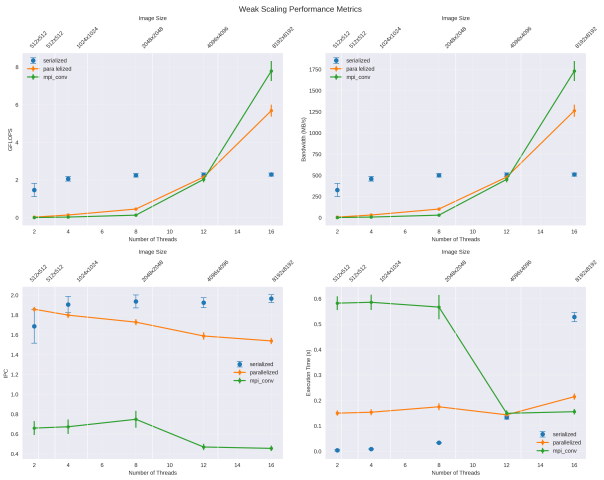

Fig. 9. Weak Scaling Metrics: GFLOPS, Memory Bandwidth, IPC, Execution Time vs. threads.

GFLOPS: Both parallel versions scaled GFLOPS upwards; 'parallelized' consistently higher. **Memory Bandwidth:** Both increased usage; 'parallelized' sustained higher bandwidth.

IPC: ‘parallelized‘ maintained higher, stable IPC (1.8-1.9); mpi_conv lower (0.6-0.7), dipping late. **Execution Time:** ‘parallelized‘ showed excellent weak scaling (near-constant time 0.15-0.18s); mpi_conv higher (0.20-0.25s) with a slight upward trend.

2. Detailed Weak Scaling Analysis via 3D Surface Plots Memory Bandwidth (Fig. 10): ‘parallelized‘ surface consistently higher, indicating superior bandwidth utilization.

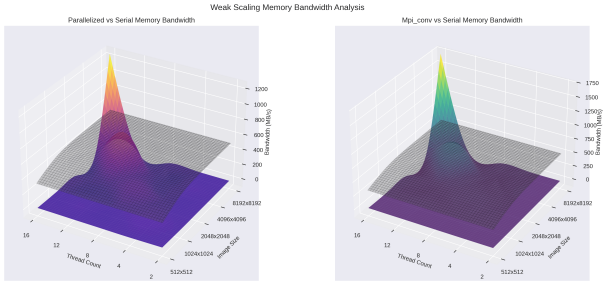

Fig. 10. Weak Scaling Memory Bandwidth (3D): ‘parallelized‘ (L) mpi_conv (R) vs. Serial.

L1 Cache Miss Rate (Fig. 11): ‘parallelized‘ exhibited low, flat L1 miss rates; mpi_conv showed peaks, suggesting poorer L1 locality [8].

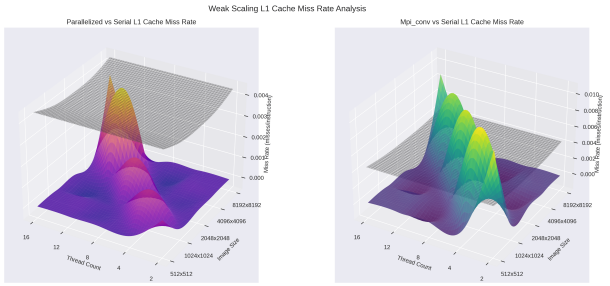

Fig. 11. Weak Scaling L1 Cache Miss Rate (3D): ‘parallelized‘ (L) mpi_conv (R) vs. Serial.

GFLOPS/s (Fig. 12): ‘parallelized‘ GFLOPS surface consistently higher and smoother.

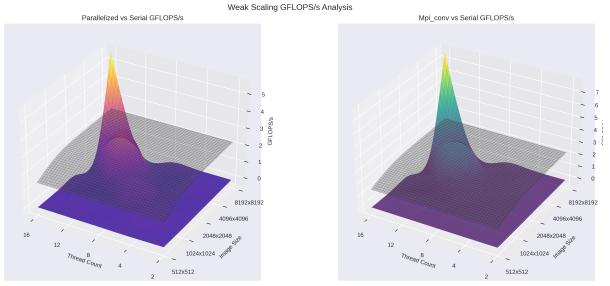

Fig. 12. Weak Scaling GFLOPS/s (3D): ‘parallelized‘ (L) mpi_conv (R) vs. Serial.

Execution Time (Fig. 13): ‘parallelized‘ surface flatter and lower, confirming near-ideal weak scaling.

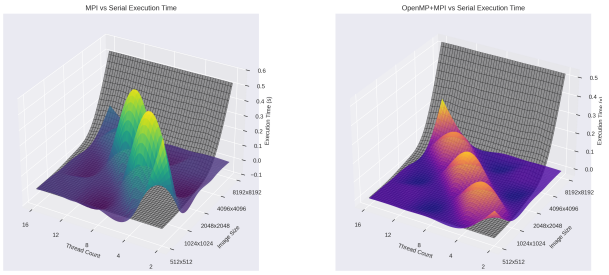

Fig. 13. Weak Scaling Execution Time (3D): mpi_conv (L) ‘parallelized‘ (R) vs. Serial.

3. Summary of Weak Scaling Findings

- Hybrid ‘parallelized‘ (OpenMP+MPI) outperformed MPI-only mpi_conv, achieving near-ideal weak scaling with constant execution time.
- mpi_conv, while scalable, showed lower efficiency, linked to higher L1 miss rates and MPI overheads.
- Hybrid’s superiority likely due to lower OpenMP overheads for intra-node tasks, better shared-memory cache use, and finer load balancing [9].

*2) **Strong Scaling**:* Strong scaling used a fixed problem size (8192x8192 in this case) with increasing units (1-16), analyzed via Amdahl’s Law [1].

1. Speedup and Amdahl’s Law (Fig. 14)

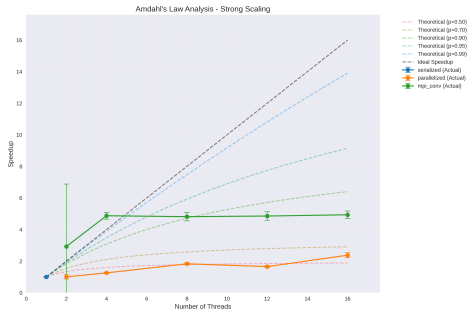

Fig. 14. Strong Scaling Speedup vs. Threads: mpi_conv ‘parallelized‘ vs. Amdahl’s Law [10].

mpi_conv: Good initial speedup (peak 4.9x at 4 threads), then plateaued, limited by sequential portions or MPI overheads. **‘parallelized‘:** Much lower speedup (max 2.35x), significantly limited by overheads for the fixed problem size. **Comparison:** mpi_conv scaled better initially; ‘parallelized‘ struggled with hybrid overheads.

2. Strong Scaling Performance Metrics (Figs. 15-20)

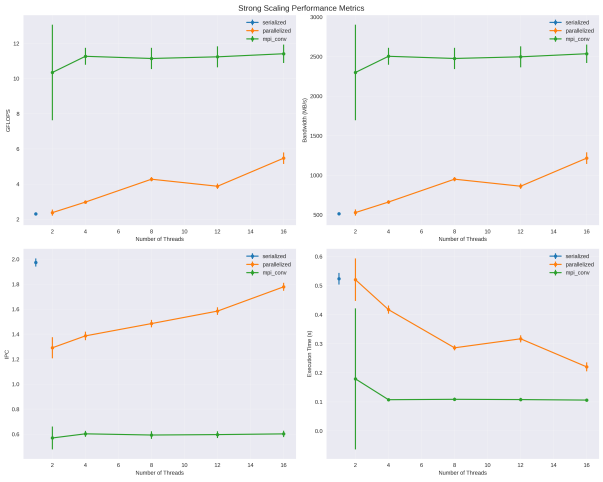

Fig. 15. Strong Scaling Metrics: GFLOPS, Bandwidth, IPC, Exec. Time vs. threads [11].

GFLOPS (Fig. 16): mpi_conv higher throughput, plateaued; ‘parallelized‘ consistently lower.

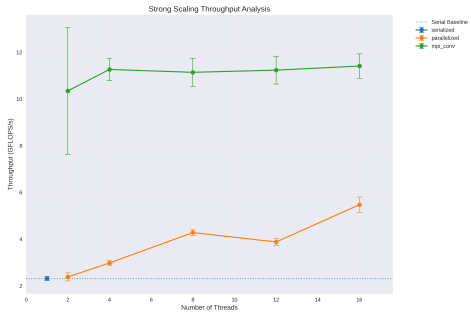

Fig. 16. Strong GFLOPS.

Memory Bandwidth (Fig. 17): mpi_conv higher usage; ‘parallelized‘ lower.

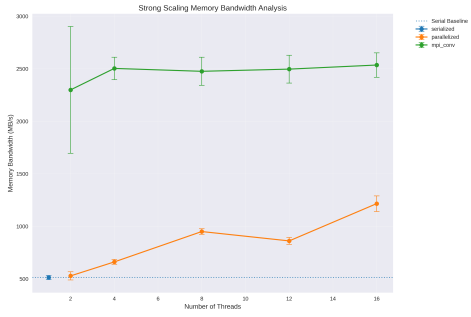

Fig. 17. Strong Mem. Bandwidth.

IPC (Fig. 18): mpi_conv IPC dropped significantly; ‘parallelized‘ IPC increased with threads (efficient small work execution) but overall speedup remained poor.

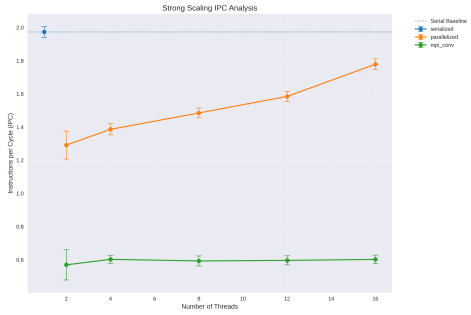

Fig. 18. Strong IPC.

Execution Time: mpi_conv faster, stabilized early; ‘parallelized‘ slower. **Parallel Efficiency (Fig. 19):** mpi_conv high initially then dropped; ‘parallelized‘ low throughout.

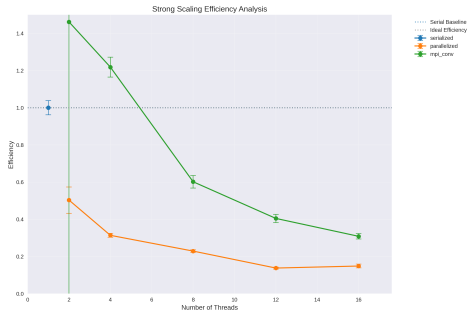

Fig. 19. Strong Efficiency [10].

L1 Cache Miss Rate (Fig. 20): mpi_conv rate doubled; ‘parallelized‘ low/stable near serial [8].

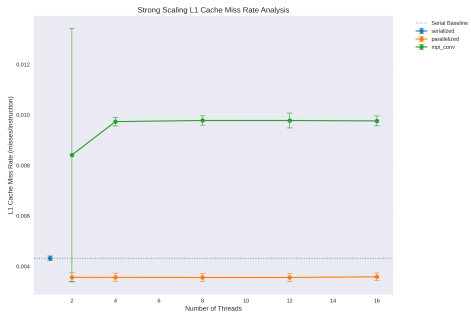

Fig. 20. Strong L1 Miss Rate.

3. Summary of Strong Scaling Findings

- MPI-only mpi_conv outperformed ‘parallelized‘ in strong scaling.
- mpi_conv limited after 4 processes; ‘parallelized‘ struggled with hybrid overheads on the fixed problem size, despite good per-thread IPC and L1 cache behavior.
- Results highlight that optimal parallel strategy is highly dependent on scaling regime and problem size relative to core count.

V. Conclusion and Future Work

This research demonstrated the acceleration of 2D convolution kernels on multi-core CPUs and GPUs using MPI, OpenMP, and CUDA.

The CUDA implementation on an NVIDIA RTX 3050 GPU achieved a significant speedup of approximately 11.7× over the sequential CPU version, highlighting the massive parallelism available on GPUs. Nsight Compute profiling revealed that the kernel was largely compute-bound with high L1/L2 cache hit rates, though opportunities exist for improving memory coalescing and reducing texture unit stalls.

For multi-core CPU parallelism, the hybrid OpenMP+MPI (‘parallelized‘) approach exhibited superior performance in weak scaling scenarios, achieving near-ideal scalability with constant execution time as problem size and core count increased proportionally. This indicates effective management of parallelism overheads and efficient use of shared memory resources [9]. Conversely, in strong scaling experiments with a fixed problem size, the MPI-only (‘mpi-conv‘) version outperformed the hybrid model, particularly up to 4 processes, after which its benefits diminished due to Amdahl's Law limitations [1] and increased MPI communication overheads. The hybrid model struggled in strong scaling due to the combined overheads of MPI and OpenMP becoming significant relative to the reduced work per thread. These contrasting results underscore that the optimal parallelization strategy is highly dependent on the scaling regime and the problem size-to-core ratio.

Future work: Future work will focus on several architectural, tooling, and evaluation dimensions to further advance the performance and applicability of the proposed parallelization strategies.

On the GPU side, additional optimizations of memory access patterns—particularly improvements in memory coalescing and shared memory utilization—are expected to yield further performance gains. Investigating advanced CUDA features such as dynamic parallelism and asynchronous operations may also enhance pipeline efficiency, particularly for more complex or nested computational tasks.

For CPU-based parallelism, future efforts will involve evaluating alternative domain decomposition strategies and implementing more sophisticated hybrid load balancing mechanisms, particularly under strong scaling scenarios. Moreover, systematic experimentation across a broader set of CPU architectures—including variations in cache hierarchies, NUMA configurations, and interconnect topologies—will provide deeper insights into performance portability and architectural sensitivity.

The use of larger compute clusters will also be considered to assess the scalability limits of the current approach. In addition, the potential for heterogeneous acceleration will be explored further through the integration of alternative platforms such as FPGAs, which may offer energy-efficient acceleration for specific kernel classes.

In terms of development tools, comparative analysis involving other parallel programming models (e.g., OpenACC) and the use of auto-parallelizing compilers will be pursued to evaluate their efficacy and applicability relative to the current implementation.

Finally, the evaluation framework will be extended to include metrics such as power consumption, thermal behavior, and performance-per-watt. These considerations are essential for realistic deployment scenarios, particularly in energy-constrained or thermally sensitive environments.

Acknowledgment

We thank Dr. Islam Tharwat Abdel Halim and Eng. Hassan Ahmed for their guidance.

References

[1] G. Hager and G. Wellein, *Introduction to High Performance Computing for Scientists and Engineers*, 2nd ed. Boca Raton, FL: CRC Press, 2021.

[2] P. J. H. Toth and I. T. Cicek, “Efficient parallelization of convolution operations using openmp for image processing applications,” *International Journal of Computer Applications*, vol. 150, no. 4, pp. 12–18, September 2016.

[3] R. Farber, *Parallel Programming with OpenMP*. Morgan Kaufmann, 2011.

[4] NVIDIA Corporation, *CUDA C Programming Guide*, 2021, version 11.4. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide

[5] V. Ferrari, R. Sousa, M. Pereira, J. P. L. de Carvalho, J. N. Amaral, J. Moreira, and G. Araujo, “Improving convolution via cache hierarchy tiling and reduced packing,” *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, 2023. [Online]. Available: https://dl.acm.org/doi/10.1145/3559009.3569678

[6] X. Huang, Q. Wang, S. Lu, R. Hao, S. Mei, and J. Liu, “Numa-aware fft-based convolution on armv8 many-core cpus,” *arXiv preprint arXiv:2109.12259*, 2021. [Online]. Available: https://arxiv.org/abs/2109.12259

[7] J. L. Gustafson, “Reevaluating amdahl's law,” *Commun. ACM*, vol. 31, no. 5, pp. 532–533, 1988.

[8] K. Lee, T. H. Song, S. H. Yoon, K. H. Kwon, and J. W. Jeon, “OpenMP parallel programming using dual-core embedded system,” in *Proc. 2012 International Conference on Ubiquitous Computing and Multimedia (UCM)*. IEEE, 2012.

[9] B. Gawrych and P. Czarnul, “Performance assessment of OpenMP constructs and benchmarks using modern compilers and multi-core CPUs,” in *Proc. 2023 18th Conf. on Computer Science and Intelligence Systems (FedCSIS)*. IEEE, 2023, pp. 973–978.

[10] V. Rajput and A. Katiyar, “Proactive bottleneck performance analysis in parallel computing using OpenMP,” *arXiv preprint arXiv:1311.1907*, 2013. [Online]. Available: https://arxiv.org/abs/1311.1907

[11] Z. Wang, L. Chen, M. Ling, and Q. Han, “Optimizing parallel programs on ARM and GPU architectures,” in *Proc. 2023 IEEE Int. Conf. on High Performance Computing (HiPC)*. IEEE, 2023, pp. 123–130.