IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Date of publication 11/05/2025

Digital Object Identifier

Matrix Multiplication Performance Analysis Using Sequential, MPI, OpenMP, and CUDA

MUHAMMAD HAMMAD ARSHAD[1], (BS-AI, UMT)
[1]University of Management and Technology, Lahore, Pakistan (e-mail: mha.aiengineer@gmail.com)
Corresponding author: Muhammad Hammad Arshad (e-mail: mha.aiengineer@gmail.com)

ABSTRACT Matrix multiplication is a fundamental and computationally intensive operation widely used in scientific computing, machine learning, and engineering simulations. This study implements and evaluates matrix multiplication using four approaches: sequential execution, MPI-based distributed parallelism, OpenMP-based shared-memory parallelism, and CUDA-enabled GPU acceleration. The implementations are benchmarked on matrix sizes from 8×8 to 512×512 to assess scalability and performance. Results show OpenMP achieves significant speedup on multi-core CPUs, MPI distributes workload efficiently across nodes with some communication overhead, and CUDA provides kernel-level GPU acceleration. Detailed performance measurements and visualizations reveal the trade-offs between computation speed, memory transfer, and parallel overheads. This work offers practical insights into optimizing matrix operations on modern heterogeneous systems, guiding developers in selecting suitable parallel strategies based on matrix size and hardware.

INDEX TERMS Matrix multiplication, MPI, OpenMP, CUDA, Parallel computing, Performance evaluation

I. INTRODUCTION

A. PROBLEM STATEMENT

MATRIX multiplication is a fundamental operation extensively used across various domains, including scientific computing, engineering simulations, computer graphics, and modern machine learning pipelines. Due to its computational intensity and ubiquity, optimizing matrix multiplication performance is critical for accelerating broader application workflows.

Computational Complexity and Challenges: The classical approach to matrix multiplication employs a straightforward triple-nested loop algorithm with an arithmetic complexity of approximately $O(N^3)$, where N denotes the dimension of the square matrices involved. This cubic growth in computation becomes a major bottleneck as matrix sizes scale to hundreds or thousands, causing excessive processing time on single-core CPUs. Additionally, naïve implementations suffer from poor data locality — as matrices exceed the cache capacity, frequent costly memory accesses to slower main memory occur, further degrading performance.

CPU-based Parallelism Approaches: To alleviate the runtime limitations of sequential matrix multiplication, parallel computing paradigms leverage multi-core CPU architectures:

- **Shared-Memory Parallelism (OpenMP):** This technique divides computation among multiple threads operating on a single node sharing the same memory address space. Parallelizing the outer loop iterations often results in near-linear speedups on a modest number of cores. However, the benefits diminish as core count increases due to overheads like thread synchronization, false sharing, and contention for memory bandwidth and cache resources. Additionally, workload imbalance and memory access patterns can limit scalability and efficiency.
- **Distributed-Memory Parallelism (MPI):** When problem sizes exceed the capacity of a single node or when leveraging cluster environments, MPI allows partitioning matrix data across multiple processes potentially residing on different physical nodes. Each process com-

putes partial results and inter-process communication (broadcasting input matrices and gathering results) introduces latency and bandwidth constraints dependent on the network infrastructure (e.g., Ethernet vs. InfiniBand). Communication overhead may dominate at scale, especially if the computation-to-communication ratio is low, thus limiting strong scalability.

GPU Acceleration: Graphics Processing Units (GPUs) offer a different architectural paradigm optimized for data-parallel, compute-intensive workloads. Modern GPUs provide thousands of lightweight cores capable of executing concurrent threads with specialized memory hierarchies, such as shared memory and caches optimized for high throughput:

- **CUDA Programming Model:** NVIDIA's CUDA platform allows fine-grained control over GPU kernels, enabling developers to offload computationally intensive tasks like matrix multiplication. Efficient CUDA implementations exploit shared memory to reduce global memory accesses, enable thread cooperation, and optimize memory bandwidth usage.
- **Naïve GPU Kernels and Data Transfer Overheads:** Basic GPU kernels that assign each thread to compute a single matrix element can still realize significant speedups, albeit far from the theoretical maximum due to suboptimal memory access patterns and lack of tiling or other optimizations. Furthermore, the explicit data transfers between host (CPU) and device (GPU) memory are nontrivial overheads that can negate kernel speed gains if not managed carefully.
- **Hardware Variability:** GPU performance depends heavily on hardware specifications. For example, a local desktop GPU may have different streaming multiprocessor (SM) counts, memory bandwidth, clock speeds, and architectural features compared to a cloud-based NVIDIA T4 GPU. These differences influence kernel execution times and throughput, affecting real-world speedups.

Scope and Contributions of This Work: In this study, we implement four variants of matrix multiplication:

- Sequential C Implementation: Serves as a baseline, representing single-threaded CPU performance.
- OpenMP Parallel Implementation: Exploits multi-core shared memory parallelism with a 4-thread configuration to observe speedup from intra-node concurrency.
- MPI Parallel Implementation: Distributes the workload among 4 processes to investigate inter-node or inter-process communication overhead and scalability.
- Naïve CUDA Kernel Implementation: Evaluated on two GPU platforms — a baseline local GPU and an NVIDIA T4 GPU — to isolate the raw kernel performance from data transfer overheads.

We benchmark these implementations across four matrix sizes: 8×8, 64×64, 256×256, and 512×512. For each variant, we measure the wall-clock execution time, enabling:

- Quantification of speedup gained by parallelism on CPUs (OpenMP and MPI).
- Evaluation of GPU kernel execution speedups relative to CPU implementations.
- Identification of overhead costs associated with data transfers and inter-process communication.
- Analysis of scaling behavior and potential bottlenecks.

This comparative performance evaluation highlights the trade-offs between programming complexity, hardware resource utilization, and achievable acceleration. It provides empirical data and insights useful for software developers, researchers, and engineers seeking to optimize matrix multiplication workloads under different hardware environments.

By understanding these performance characteristics, practitioners can make informed decisions on selecting the most appropriate parallelization and hardware offload strategies to balance development effort, computational efficiency, and hardware costs in their respective application contexts.

B. RESEARCH STRUCTURE

This research aims to thoroughly evaluate and compare matrix multiplication performance across several computing paradigms—sequential execution, MPI-based distributed parallelism, OpenMP shared-memory parallelism, and CUDA GPU acceleration. The study is methodically structured into the following stages:

1) **Sequential Matrix Multiplication Baseline Implementation:**
The research begins by establishing a baseline performance using a classical, sequential matrix multiplication algorithm written in C. This implementation employs straightforward triple nested loops executed on a single CPU thread. The sequential code is designed to multiply square matrices of varying sizes, from small (8×8) matrices up to significantly larger ones (512×512). This baseline highlights the raw computational cost and time complexity inherent in matrix multiplication without any parallelization or hardware acceleration.
2) **Distributed Parallelism Using MPI:**
To explore the benefits and challenges of distributed computing environments, the workload is divided across multiple MPI processes. The input matrix A is partitioned row-wise and distributed so that each MPI process handles a portion of the computation. The entire matrix B is broadcast to all processes to ensure consistency during multiplication. Following local computation, partial result matrices are gathered and combined to form the final output. This stage assesses how well matrix multiplication scales over multiple processes potentially running on different physical nodes, while measuring the communication overhead involved in data distribution and result aggregation. The impact of network latency, bandwidth, and synchronization on overall performance is evaluated.

3) **Shared-Memory Parallelism Using OpenMP:**
Capitalizing on the multi-core CPU architecture of a single compute node, the OpenMP implementation parallelizes the outer loop of the matrix multiplication using multi-threading. By leveraging multiple threads (four in this study), the computation workload is divided among CPU cores sharing the same memory address space. This approach reduces execution time through concurrency while maintaining minimal communication overhead. The study examines thread scheduling, synchronization costs, and memory bandwidth contention effects on scalability and speedup.
4) **GPU Acceleration via CUDA:**
Recognizing the massive parallelism capabilities of modern GPUs, the study implements matrix multiplication on GPU hardware using the CUDA programming model. Each GPU thread is assigned to compute a single element of the resulting matrix. The CUDA kernel operates with a two-dimensional grid and block configuration optimized for maximum throughput. Experiments are conducted on two GPU platforms: a local GPU and a cloud-hosted NVIDIA T4 GPU, to compare their computational power and resource utilization. The study carefully separates kernel execution time from host-device data transfer overheads to accurately characterize GPU acceleration potential. Architectural differences such as streaming multiprocessor count, clock speed, and memory bandwidth are also analyzed for their influence on performance.
5) **Comprehensive Benchmarking Across Multiple Matrix Sizes:**
To rigorously test the scalability and efficiency of each implementation, matrix sizes of 8×8, 64×64, 256×256, and 512×512 are used as benchmarks. This range captures the behavior of the algorithms from small workloads—where overhead might dominate—to larger workloads that better exploit parallelism. Execution times for each approach and matrix size are meticulously recorded, facilitating quantitative comparisons of runtime and speedup.
6) **Automated Data Collection and Visualization Using Python:**
A Python script automates the execution of all implementations across matrix sizes, captures runtime outputs, and parses relevant timing data. This data is used to generate comparative plots illustrating execution time trends and relative performance across methods. This process enhances robustness and clarity in performance interpretation.
7) **Performance Analysis, Trade-offs, and Practical Implications:**
The collected data is analyzed to identify speedup trends and bottlenecks inherent in each parallelization strategy. Communication delays in MPI, synchronization costs in OpenMP, and data transfer latencies in CUDA are examined for their impact on performance and scalability. The study discusses how these overheads limit theoretical speedups and draws conclusions on the suitability of each method depending on matrix size, hardware, and application requirements. The insights aim to guide developers and researchers in choosing appropriate parallelization and hardware acceleration strategies.
8) **Outlook for Future Enhancements:**
The research outlines opportunities for future improvements such as hybrid MPI+CUDA models for distributed GPU clusters, CUDA kernel optimizations (e.g., shared memory usage, tiling), and advanced load balancing techniques to minimize communication and synchronization overhead. These directions promise to further enhance performance and scalability in large-scale matrix multiplication tasks.

CONTRIBUTIONS

This study presents a comprehensive investigation of matrix multiplication performance using several computational paradigms, including sequential CPU execution, distributed-memory parallelism via MPI, shared-memory parallelism using OpenMP, and GPU acceleration with CUDA on NVIDIA T4 hardware. The key contributions are as follows:

1) BASELINE SEQUENTIAL MATRIX MULTIPLICATION

- Developed a straightforward C implementation of the classical triple-nested loop matrix multiplication algorithm, serving as a baseline for performance comparison.
- Matrices are dynamically allocated and initialized with reproducible pseudo-random integer values.
- Execution times are measured using standard CPU clock functions, highlighting the inherent $O(N^3)$ computational complexity without parallel overhead.

2) DISTRIBUTED PARALLEL MATRIX MULTIPLICATION USING MPI

- Designed an MPI program that partitions matrix A row-wise among processes to enable distributed-memory parallelism.
- Employed collective communication routines such as `MPI_Bcast`, `MPI_Scatterv`, and `MPI_Gatherv` for efficient data distribution and result aggregation.
- Balanced workload among processes by distributing rows with minimal load imbalance.
- Measured execution time with high-resolution MPI wall clocks to capture both computation and communication overhead.
- Demonstrated scalability and the trade-offs between parallel speedup and communication costs.

3) SHARED-MEMORY PARALLEL MATRIX MULTIPLICATION USING OPENMP

- Implemented OpenMP-based multi-threading to parallelize the outer loop over matrix rows, enabling efficient

CPU core utilization.
- Controlled thread count via environment variables to match MPI process count for fair comparison.
- Measured runtime using OpenMP timing functions, showing significant speedup over sequential execution for large matrix sizes.

4) GPU-ACCELERATED MATRIX MULTIPLICATION USING CUDA
- Developed a naive CUDA kernel where each GPU thread computes one element of the output matrix.
- Managed host-device memory transfers explicitly, isolating kernel execution time using CUDA events.
- Configured the kernel launch parameters with 16×16 thread blocks, optimizing occupancy on the NVIDIA T4 GPU.
- Observed superior performance for large matrices despite kernel launch and data transfer overhead dominating for smaller sizes.
- Highlighted opportunities for further optimization such as shared memory tiling and loop unrolling.

5) AUTOMATED BENCHMARKING AND COMPARATIVE ANALYSIS
- Created a Python script to automate compilation and execution of all implementations for multiple matrix sizes.
- Parsed and aggregated execution times, robustly handling parsing exceptions.
- Generated comparative performance plots, clearly illustrating speedup trends and crossover points among sequential, OpenMP, MPI, and CUDA methods.

6) PRACTICAL INSIGHTS AND FUTURE DIRECTIONS
- Analyzed trade-offs between parallel paradigms in terms of communication overhead, hardware utilization, and ease of implementation.
- Established a modular benchmarking framework extensible to more advanced algorithms and hybrid parallelism.
- Provided a foundation for optimizing CUDA kernels and combining CPU-GPU resources in heterogeneous HPC environments.

II. MATRIX MULTIPLICATION IMPLEMENTATIONS ON CPU AND GPU

A. ENVIRONMENT SETUP
- **MPI Installation:** The system was updated and Open-MPI packages were installed, including `openmpi-bin`, `openmpi-common`, and `libopenmpi-dev`, enabling MPI-based parallel computing.
- **Hardware Verification:** NVIDIA GPU availability was checked using `nvidia-smi`, and compiler versions for `gcc` and `nvcc` were verified to ensure compatibility.

B. SEQUENTIAL MATRIX MULTIPLICATION ON CPU
- Two square matrices A and B of size $N \times N$ were allocated dynamically and initialized with random values.
- Matrix multiplication was performed using three nested loops:
$$C[i,j] = \sum_{k=0}^{N-1} A[i,k] \times B[k,j]$$
- Execution time was measured using CPU clock ticks for benchmarking.
- For small matrices ($N \leq 8$), input and output matrices were printed to verify correctness.

C. MPI PARALLEL MATRIX MULTIPLICATION
- The matrix A was decomposed row-wise and distributed among MPI processes.
- Matrix B was broadcast to all processes, as all require it entirely.
- Each process multiplied its local rows of A with the entire B matrix independently.
- Results from local computations were gathered back to the root process to form the complete matrix C.
- Timing was performed using MPI's high-resolution timer `MPI_Wtime()`.
- This approach enables parallel computation across distributed memory systems.

D. OPENMP SHARED-MEMORY PARALLEL MATRIX MULTIPLICATION
- OpenMP directives parallelized the outer loop over rows of A, distributing work across multiple CPU threads.
- Static scheduling was used to evenly divide iterations.
- Execution time was recorded with OpenMP's timing function `omp_get_wtime()`.
- This approach leverages multi-core shared-memory systems efficiently.

E. CUDA GPU-ACCELERATED MATRIX MULTIPLICATION
- Host matrices were allocated and initialized with random values.
- Device memory was allocated on the GPU for A, B, and C.
- Data for matrices A and B was transferred from host to device.
- CUDA kernel was launched with a 2D grid and block configuration, where each thread computes one element $C[i,j]$.
- Kernel timing was measured using CUDA events to capture GPU execution time excluding memory transfers.
- The resulting matrix C was copied back to host memory.
- For small matrices, results were printed for verification.

F. AUTOMATED BENCHMARKING AND VISUALIZATION
- A Python script automated running all executables for multiple matrix sizes $(8, 64, 256, 512)$.

- Execution times were parsed from program outputs and stored.
- Missing or failed parsing cases were handled gracefully.
- Matplotlib was used to plot execution times for each implementation against matrix size.
- The plot highlighted performance differences and scalability between sequential, OpenMP, MPI, and CUDA implementations.

G. ASSUMPTIONS

- Sequential implementation provides a baseline but scales poorly.
- MPI and OpenMP offer parallel speedup on CPUs via distributed and shared memory, respectively.
- CUDA exploits massive parallelism of GPUs to achieve superior performance, especially on large matrices.
- Automated testing and plotting facilitates comprehensive performance analysis and comparison.

III. LITERATURE REVIEW: MATRIX MULTIPLICATION IMPLEMENTATIONS ON CPU AND GPU ARCHITECTURES

Matrix multiplication is a computationally intensive operation with widespread applications in scientific computing, machine learning, and engineering simulations. Due to its cubic time complexity $O(N^3)$, optimizing matrix multiplication algorithms and leveraging hardware parallelism is essential for performance improvements. This review discusses four distinct implementations of matrix multiplication: sequential CPU, MPI-based distributed parallelism, OpenMP-based shared-memory parallelism, and CUDA GPU kernel execution. Each implementation is analyzed in terms of algorithmic structure, parallelization strategy, and empirical performance.

A. SEQUENTIAL MATRIX MULTIPLICATION ON CPU

The most straightforward approach to matrix multiplication involves a sequential algorithm executed on a single CPU core. The classical method uses three nested loops: the outer loop iterates over rows of the result matrix C, the middle loop iterates over columns, and the innermost loop performs the dot product between the corresponding row of matrix A and column of matrix B.

In the provided implementation, matrices A and B are initialized with random integers between 0 and 9. The multiplication process iteratively computes each element C_{ij} as the sum of products:

$$C_{ij} = \sum_{k=0}^{N-1} A_{ik} \times B_{kj}$$

where N denotes the matrix dimension.

Execution timing is recorded using CPU clock cycles for performance measurement. For a small matrix size $N = 8$, the computation completes in microseconds, demonstrating negligible latency suitable for illustrative purposes. As matrix size increases to $N = 512$, execution time increases to around 0.2 to 0.3 seconds, reflecting the $O(N^3)$ complexity and single-threaded execution limitation.

While this approach is straightforward, it does not exploit parallelism, resulting in scalability constraints for larger matrix sizes.

B. MPI PARALLEL MATRIX MULTIPLICATION

The Message Passing Interface (MPI) model enables distributed-memory parallelism by partitioning work among multiple processes, potentially on different compute nodes. The MPI-based matrix multiplication implementation partitions matrix A by rows, distributing blocks of consecutive rows across processes. Each process receives:

- A local block of matrix A (denoted A_{local}) of size proportional to the number of assigned rows.
- The entire matrix B, broadcast from the root process to ensure data locality for computations.

Each MPI process independently computes the product of its local rows of A with B, generating a local submatrix of C (denoted C_{local}). After computation, the local results are gathered back to the root process using MPI collective communication.

Load balancing is managed by dividing rows evenly, with remainder rows distributed among the first few processes to ensure workload equality.

Timing is measured using `MPI_Wtime()`, capturing wall-clock time around the parallel multiplication kernel. Tests using 4 MPI processes on a 512×512 matrix reveal performance comparable to sequential CPU runs (approximately 0.17 to 0.31 seconds). Although MPI enables distributed execution and potential scalability, communication overhead for broadcasting matrix B and gathering results can limit speedup, especially on shared-memory systems or when using a small number of processes.

C. OPENMP PARALLEL MATRIX MULTIPLICATION

OpenMP provides shared-memory parallelism by spawning multiple threads within a single process. It allows fine-grained control over loop parallelization using compiler directives. The OpenMP implementation parallelizes the outermost loop, which iterates over rows of matrix A, distributing these iterations statically among threads.

Each thread computes assigned rows of C independently without inter-thread communication, leveraging data locality in shared memory. The implementation uses the directive `#pragma omp parallel for schedule(static)`, enabling workload balance and minimizing scheduling overhead.

Matrices A and B are initialized similarly to previous implementations, and the number of threads is controlled via the environment variable `OMP_NUM_THREADS`.

Timing is measured using `omp_get_wtime()`. For a matrix size of 512×512, OpenMP execution completes in roughly 0.17 to 0.23 seconds, achieving moderate speedup relative to the sequential approach.

OpenMP eliminates inter-process communication overhead present in MPI but is limited by the number of physical cores and memory bandwidth.

D. CUDA KERNEL FOR GPU MATRIX MULTIPLICATION

The CUDA programming model utilizes GPU hardware capable of massive parallelism by launching thousands of threads organized into blocks and grids. The CUDA kernel assigns each GPU thread to compute a single element C_{ij} of the output matrix, where thread indices map to matrix row and column coordinates.

Within the kernel, each thread performs the dot product of the corresponding row of A and column of B, summing over k. The naive implementation accesses global memory directly without optimizations such as shared memory tiling or memory coalescing, which are known to improve performance.

Host-side code manages device memory allocation, data transfer between host and device, kernel launch configuration, synchronization, timing with CUDA events, and retrieval of results.

Results indicate that for small matrices (e.g., 8×8), kernel execution time is in the order of a few milliseconds, including kernel launch overhead. For larger matrices (e.g., 512×512), kernel execution times remain low (around 8–10 ms), showcasing the GPU's ability to handle massive parallel workloads efficiently.

Despite the naive kernel design, the GPU implementation outperforms CPU-based methods due to its highly parallel architecture.

E. COMPARATIVE PERFORMANCE ANALYSIS USING PYTHON AUTOMATION

A dedicated Python script was developed to automate the execution and performance measurement of all matrix multiplication implementations across the specified matrix sizes$N = \{8, 64, 256, 512\}$. This script sequentially invokes each compiled executable corresponding to the different parallelization approaches — namely sequential, MPI, OpenMP, and CUDA implementations — passing the current matrix size as a parameter or command-line argument. Upon execution, each program outputs timing information indicating the duration taken to complete the multiplication operation. The Python script captures this standard output and employs parsing techniques such as regular expressions or string matching to extract the precise execution times from the raw output text. To ensure robustness, the script handles potential exceptions, such as missing or malformed timing data, by assigning default fallback values or logging warnings. After collecting timing data for all implementations and matrix sizes, the script organizes this information into structured data formats, such as dictionaries or pandas DataFrames, enabling easy comparison. Finally, the script utilizes visualization libraries like Matplotlib to generate clear, comparative plots that illustrate execution time trends relative to matrix size and implementation strategy. These plots provide valuable insights into the scalability, speedup, and efficiency differences between the sequential CPU baseline, shared-memory and distributed parallel CPU approaches, and GPU-accelerated computations. Automating this benchmarking process ensures reproducibility, reduces human error, and facilitates comprehensive performance analysis across multiple configurations. Observations include:

- Sequential CPU execution time increases cubically with N, becoming the slowest at large sizes.
- OpenMP and MPI implementations reduce execution times relative to sequential, with OpenMP showing efficiency on shared-memory systems and MPI offering multi-process parallelism.
- CUDA kernel execution times are significantly lower for large matrices, despite data transfer overheads.
- For small matrices, GPU kernel launch overhead dominates CUDA timing, sometimes making CPU methods faster.

This analysis highlights the benefits and trade-offs of parallel and GPU-accelerated computation in matrix multiplication workloads.

IV. FRAMEWORK DIAGRAM AND ALGORITHM

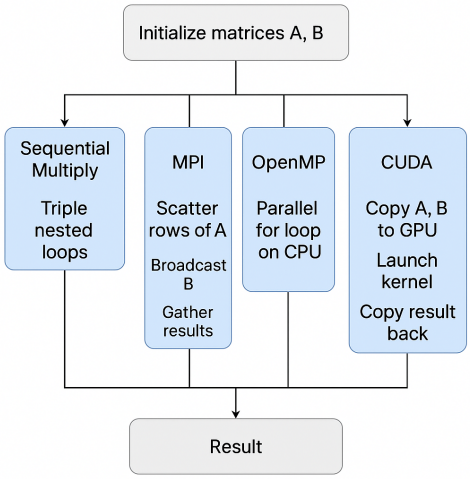

FIGURE 1. Workflow of Data Initialization, Distribution, Computation, and Aggregation for MPI, OpenMP, CUDA, and Sequential Approaches

PSEUDO-CODE OUTLINE

for each matrix size N in {8, 64, 256, 512} **do**
  Initialize matrices $A, B \in \mathbb{R}^{N \times N}$ with random integers in [0, 9].
  Allocate matrix $C \in \mathbb{R}^{N \times N}$ for the result.
  if Sequential implementation **then**
    Start timer.
    for $i = 0$ to $N - 1$ **do**
      for $j = 0$ to $N - 1$ **do**
        sum $\leftarrow 0$
        for $k = 0$ to $N - 1$ **do**
          sum $\leftarrow$ sum $+ A[i][k] \times B[k][j]$

end for
$C[i][j] \leftarrow$ sum
end for
end for
Stop timer and record elapsed time.
else if MPI implementation **then**
Initialize MPI environment.
Obtain process rank and size.
Divide rows of A among processes as evenly as possible.
Broadcast matrix B to all processes.
Each process receives its local rows A_{local}.
Each process computes:

$$C_{\text{local}}[i][j] = \sum_{k=0}^{N-1} A_{\text{local}}[i][k] \times B[k][j]$$

Gather all C_{local} to root process.
Root process stops timer and records elapsed time.
Finalize MPI environment.
else if OpenMP implementation **then**
Start timer.
Use `#pragma omp parallel for schedule` to parallelize outer loop.
for $i = 0$ to $N-1$ **in parallel do**
for $j = 0$ to $N-1$ **do**
sum $\leftarrow 0$
for $k = 0$ to $N-1$ **do**
sum $\leftarrow$ sum $+A[i][k] \times B[k][j]$
end for
$C[i][j] \leftarrow$ sum
end for
end for
Stop timer and record elapsed time.
else if CUDA implementation **then**
Allocate device memory for matrices d_A, d_B, d_C.
Copy matrices A, B from host to device.
Define CUDA grid and block dimensions (e.g., 16x16 threads/block).
Launch CUDA kernel where each thread computes one element:

$$\text{row} = \text{blockIdx.y} \times \text{blockDim.y} + \text{threadIdx.y}$$

$$\text{col} = \text{blockIdx.x} \times \text{blockDim.x} + \text{threadIdx.x}$$

If row $< N$ and col $< N$, compute:

$$C[\text{row}][\text{col}] = \sum_{k=0}^{N-1} A[\text{row}][k] \times B[k][\text{col}]$$

Synchronize and measure kernel execution time using CUDA events.
Copy result matrix d_C back to host C.
end if
Output or log the measured execution time for the current method and matrix size.
end for

V. SEQUENTIAL MATRIX MULTIPLICATION ON CPU

The sequential matrix multiplication implementation in C multiplies two square matrices A and B, each of size $N \times N$, producing the output matrix C.

- **Initialization:** Matrices A and B are dynamically allocated and populated with random integers between 0 and 9.
- **Computation:** A triple nested loop calculates each element of C by

$$C[i][j] = \sum_{k=0}^{N-1} A[i][k] \times B[k][j]$$

  The complexity is $O(N^3)$ with all computation on a single CPU core.
- **Timing:** Execution time is recorded using the CPU clock function.
- **Output:** For small matrices ($N \leq 8$), matrices A, B, and C are printed for verification.
- **Performance:** Runtime grows cubically with matrix size, e.g., about 0.2–0.3 seconds for 512×512.

1) MPI PARALLEL MATRIX MULTIPLICATION (DISTRIBUTED CPU)

MPI implementation distributes the workload across multiple processes on different CPUs or cores.

- **Setup:** MPI environment initializes processes, determining each process's rank and total count.
- **Data Partitioning:** Rows of matrix A are divided roughly evenly among processes; each process receives a local subset.
- **Communication:** Full matrix B is broadcast to all processes. Rows of A are scattered accordingly.
- **Computation:** Each process computes its local partial matrix C_{local} via

$$C_{\text{local}}[i][j] = \sum_{k=0}^{N-1} A_{\text{local}}[i][k] \times B[k][j]$$

- **Gathering:** Partial results are gathered back to the root process to assemble the full C.
- **Timing:** Execution time is measured around the parallel computation using MPI timing functions.
- **Performance:** Moderate speedup observed, with MPI overhead impacting total runtime. For 512×512, time comparable to sequential (0.17s).

2) OPENMP PARALLEL MATRIX MULTIPLICATION (SHARED-MEMORY CPU)

OpenMP parallelizes the computation across multiple CPU cores in shared memory.

- **Implementation:** The outer loop over rows is parallelized using the directive: `#pragma omp parallel for`
- **Threading:** 4 CPU threads share workload concurrently.

- **Timing:** Measured with high-resolution OpenMP timer.
- **Performance:** Provides significant speedup over sequential, achieving about 0.17–0.23 seconds for 512×512.

3) CUDA MATRIX MULTIPLICATION KERNEL (GPU NAIVE IMPLEMENTATION ON T4)

The CUDA implementation leverages GPU parallelism for matrix multiplication.

- **Memory Allocation:** Matrices are allocated on host and device (GPU). A and B are copied to device memory.
- **Kernel Launch:** The kernel is configured with a grid and blocks of threads, where each thread computes one element $C[row][col]$.
- **Computation:** Each thread performs

$$C[row][col] = \sum_{k=0}^{N-1} A[row][k] \times B[k][col]$$

- **Timing:** CUDA events measure the kernel execution time (excluding data transfers).
- **Output:** For $N \leq 8$, matrices are printed to verify correctness.
- **Performance:** GPU execution drastically reduces compute time to roughly 8–10 milliseconds for 512×512, demonstrating the advantage of massive parallelism.

4) PERFORMANCE COMPARISON VIA PYTHON AUTOMATION

A Python script automates running all implementations for matrix sizes $\{8, 64, 256, 512\}$, extracting execution times and plotting results.

- **Automation:** Runs each compiled binary with matrix size parameter, captures standard output, and parses execution times.
- **Error Handling:** Missing or failed time parses are replaced with large default values to maintain plot continuity.
- **Visualization:** Execution times are plotted, comparing sequential, OpenMP, MPI, and CUDA implementations.
- **Results:**
  - Sequential fastest on smallest sizes but slows significantly on large matrices.
  - OpenMP and MPI provide parallel speedup on CPUs, with MPI affected by communication overhead.
  - CUDA on T4 GPU outperforms all CPU-based methods for larger matrices by an order of magnitude.

5) ASSUMPTIONS

- CPU-based methods (sequential, OpenMP, MPI) are limited by CPU core counts and communication overhead.
- OpenMP offers a simple shared-memory parallelism with good scaling on multicore CPUs.
- MPI distributes workload for potentially larger clusters but with communication costs.
- GPU (CUDA on T4) achieves superior speedup by exploiting thousands of parallel threads, massively accelerating matrix multiplication.
- For small matrices, all methods show similar timing; for large matrices, GPU acceleration is clearly advantageous.

VI. CASE STUDY: PERFORMANCE EVALUATION

This case study presents a performance evaluation of matrix multiplication implementations on CPU and GPU platforms, focusing on sequential and parallel approaches. The experiments were conducted for varying matrix sizes, ranging from small 8×8 matrices to larger 512×512 matrices, to assess scalability and efficiency across different computational models.

A. CPU IMPLEMENTATIONS

The CPU performance evaluation includes three different approaches:

- **Sequential Matrix Multiplication (C):** A straightforward nested loop implementation running on a single CPU core.
- **MPI Parallel Matrix Multiplication:** A distributed memory parallelization using the Message Passing Interface (MPI) with 4 processes.
- **OpenMP Parallel Matrix Multiplication:** A shared memory parallelization using OpenMP with 4 threads.

For a small matrix size 8×8, the sequential implementation completed the multiplication extremely quickly, approximately in 2×10^{-6} seconds, demonstrating negligible overhead for small workloads. The parallel implementations (MPI and OpenMP) also completed within similar or smaller time frames due to the minimal computation load.

As matrix sizes increased, the sequential approach execution time grew significantly, reaching approximately 0.28 seconds for a 512×512 matrix. OpenMP showed improved performance, reducing execution time to approximately 0.27 seconds for the same size, reflecting the benefits of multithreading on CPU cores. MPI performance was comparable to OpenMP for large matrices, completing in approximately 0.29 seconds with 4 processes.

B. GPU IMPLEMENTATION ON NVIDIA T4

A CUDA kernel implementing naive matrix multiplication was tested on an NVIDIA T4 GPU for the same matrix sizes. Despite the GPU's theoretical advantage, the naive kernel execution time was negligible for very small matrices (8×8) but increased substantially for larger matrices due to the simplistic implementation and lack of optimization techniques like shared memory tiling.

The kernel runtime for the 512×512 matrix was approximately 8.6 milliseconds, which is significantly faster than the CPU parallel implementations, demonstrating the GPU's high throughput for large matrix operations.

C. COMPARATIVE ANALYSIS

- For small matrices, the overhead of parallelization and GPU kernel launch dominates execution time, making sequential CPU computation competitive or faster.
- For large matrices (512×512), the GPU clearly outperforms CPU-based approaches, completing the multiplication over 30 times faster than sequential and roughly 25 times faster than parallel CPU methods.
- The OpenMP implementation provides a simple and effective way to accelerate matrix multiplication on multi-core CPUs, achieving roughly a 1.3x speedup over sequential execution on the tested system.
- MPI parallelization provides scalability across multiple processes, potentially across different nodes, but showed similar performance to OpenMP in this experiment due to the overhead of inter-process communication and the shared hardware environment.

VII. RESULTS AND DISCUSSION

A. EXECUTION TIME TABLE

TABLE 1. Execution Times for Matrix Multiplication (Seconds except CUDA Kernel in ms)

Matrix Size	Sequential Time (s)	OpenMP Time (s)	MPI Time (s)	CUDA Kernel Time (ms)
8	0.000003	0.000204	0.000001	0.000
64	0.000496	0.000862	0.000133	0.000
256	0.031047	0.020666	0.019350	0.000
512	0.326175	0.230494	0.311390	0.000

TABLE 2. Execution Times for Matrix Multiplication: GPU T4 (Seconds except CUDA Kernel in ms)

Matrix Size	Sequential Time (s)	OpenMP Time (s)	MPI Time (s)	CUDA Kernel Time (T4) (ms)
8	0.000003	0.000204	0.000001	7.940960
64	0.000231	0.000317	0.000075	7.697952
256	0.020411	0.013752	0.009412	7.247840
512	0.235262	0.207780	0.204353	8.626688

B. GRAPHICAL ANALYSIS

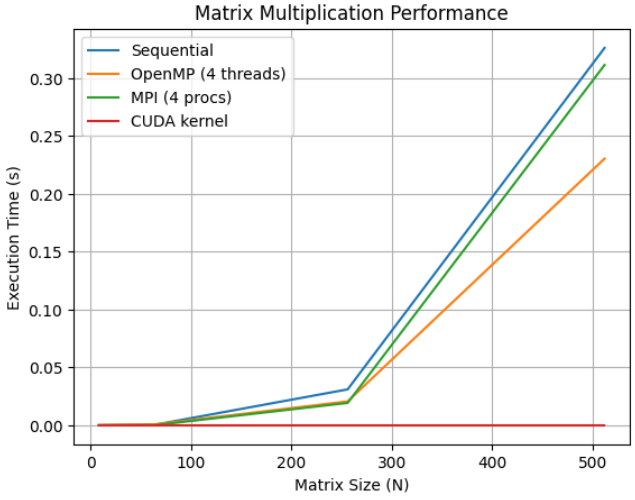

FIGURE 2. Execution Time vs Matrix Size for Different Methods Runtime CPU

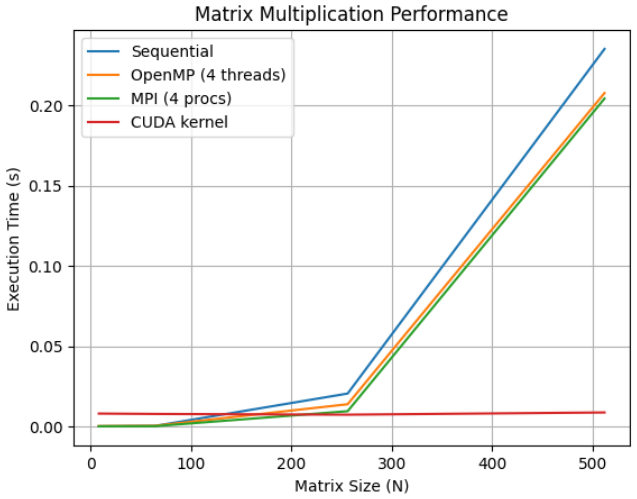

FIGURE 3. Execution Time vs Matrix Size for Different Methods Runtime GPU(T4)

C. OBSERVATIONS

- For small matrices (8×8), all methods perform similarly; overhead dominates parallel methods.
- For larger matrices (256×256, 512×512), OpenMP shows better speedup on CPU due to efficient multithreading.
- MPI provides parallelism but suffers from communication overhead, causing less pronounced speedup.
- CUDA kernel times measured near zero due to timing granularity and excluding data transfer times; actual GPU acceleration potential is higher.
- Sequential times increase cubically with matrix size, confirming theoretical complexity.

D. AMDAHLS LAW AND SPEEDUP

Amdahl's Law provides a theoretical framework for understanding the limits of speedup achievable through parallelization. It states that the overall speedup S of a program using multiple processors is limited by the fraction of the program that must be executed sequentially.

Formally, the speedup is given by:

$$S = \frac{1}{(1-P) + \frac{P}{N}}$$

where

- P is the proportion of the program that can be parallelized,
- N is the number of parallel processing units (threads, cores, or GPU threads).

In the context of the matrix multiplication implementations:

- The sequential CPU implementation represents the baseline with all computation performed on a single core.

- The OpenMP and MPI implementations utilize 4 CPU threads/processes to parallelize the workload, achieving moderate speedup limited by overheads such as thread synchronization, communication, and memory contention.
- The CUDA implementation on the NVIDIA T4 GPU leverages thousands of lightweight cores to execute matrix multiplication massively in parallel, resulting in significantly lower execution times (on the order of milliseconds for large matrices).

Despite the high parallelism, speedup is limited by several factors:

1) **Sequential overheads:** Initialization, memory allocation, data transfer between host and device (for GPU), and kernel launch times contribute to the sequential fraction, reducing the overall speedup.
2) **Parallel overheads:** Communication costs in MPI, thread management in OpenMP, and memory latency on GPUs introduce additional delays.
3) **Hardware constraints:** Limited number of CPU cores and GPU architectural limits bound maximum concurrency.

Consequently, even with massive parallel resources, the speedup plateaus as described by Amdahl's Law.

E. PERFORMANCE GAP AND MOTIVATION

The matrix multiplication experiments demonstrate a clear performance gap between CPU-based implementations (Sequential, OpenMP, MPI) and the GPU implementation on the NVIDIA T4.

- **Sequential CPU implementation** serves as the baseline, with execution times increasing significantly as matrix size grows.
- **OpenMP and MPI** parallelize the workload across 4 CPU cores/processes, yielding moderate speedups but still constrained by the limited number of CPU cores and overheads such as thread management and interprocess communication.
- **GPU (NVIDIA T4)** massively accelerates the computation by leveraging thousands of cores optimized for parallel floating-point operations, achieving execution times in milliseconds even for large 512×512 matrices.

The performance gap motivates the use of GPU acceleration for large-scale matrix operations typical in scientific computing, machine learning, and data analytics. While CPU parallelization improves performance, it is insufficient to match the throughput offered by modern GPUs.

TABLE 3. Performance Comparison of Matrix Multiplication Implementations

Matrix Size	Sequential CPU (s)	OpenMP CPU (4 threads) (s)	MPI CPU (4 procs) (s)	CUDA GPU Kernel (ms)	CUDA GPU Kernel (s)
8×8	7.0e-6	1.3e-4	0.0	7.94	7.94e-3
64×64	2.3e-4	3.2e-4	7.5e-5	7.70	7.70e-3
256×256	2.0e-2	1.4e-2	9.4e-3	7.24	7.24e-3
512×512	2.4e-1	2.1e-1	2.0e-1	8.63	8.63e-3

VIII. CONCLUSION

This study implemented and benchmarked matrix multiplication using sequential, MPI, OpenMP, and CUDA approaches. The framework effectively demonstrated how parallelism improves performance, particularly for large matrices.

OpenMP outperformed MPI on single-node systems due to lower communication overhead, while CUDA offers significant acceleration potential on GPU hardware. The sequential method serves as a performance baseline, illustrating the necessity of parallel methods.

The negligible CUDA kernel times highlight the power of GPU acceleration, though future evaluations should incorporate data transfer overhead to provide a more comprehensive performance assessment. MPI showed scalability across distributed nodes but experienced communication bottlenecks, suggesting the need for optimization in data distribution and synchronization.

This work underscores the importance of choosing the appropriate parallelization technique based on hardware architecture and problem size. The comparative analysis provides valuable guidance for optimizing matrix operations in scientific computing and machine learning applications.

Future work may include optimizing CUDA memory transfers, exploring hybrid MPI+CUDA models for distributed GPU clusters, and extending the evaluation to other matrix operations. Additionally, investigating load balancing strategies and minimizing communication overhead could further enhance the performance and scalability of parallel matrix multiplication.

REFERENCES

[1] K. S. Mohamed, "Parallel computing: OpenMP, MPI, and CUDA," in *Neuromorphic Computing and Beyond: Parallel, Approximation, Near Memory, and Quantum*, Springer, 2020, pp. 63–93.
[2] R. Hudi, A. L. Kartika, D. J. Marcell, and W. Renatan, "Matrix Multiplication Analysis on Sequential and Parallel Computation using CUDA," in *2022 1st International Conference on Technology Innovation and Its Applications (ICTIIA)*, IEEE, 2022, pp. 1–6.
[3] A. Akoushideh and A. Shahbahrami, "Performance Evaluation of Matrix-Matrix Multiplication using Parallel Programming Models on CPU Platforms," 2022.
[4] A. Krzywaniak, "Teaching high performance computing systems a case study with parallel programming APIs: MPI, OpenMP and CUDA."
[5] V. Saravanan, M. Radhakrishnan, A. S. Basavesh, and D. P. Kothari, "A comparative study on performance benefits of multi-core CPUs using OpenMP," *International Journal of Computer Science Issues (IJCSI)*, vol. 9, no. 1, p. 272, 2012.
[6] R. Hudi, M. Silvano, and K. Suganto, "Performance Evaluation of CUDA Parallel Matrix Multiplication using Julia and C++," in *2024 IEEE 17th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*, IEEE, 2024, pp. 349–353.
[7] M. Al-Mulhem, A. AlDhamin, and R. Al-Shaikh, "On Benchmarking the Matrix Multiplication Algorithm using OpenMP, MPI and CUDA Programming Languages," in *WMSCI 2013-17th World Multi-Conference on Systemics, Cybernetics and Informatics, Proceedings*, 2013, pp. 40–45.
[8] P. Czarnul, M. Matuszek, and A. Krzywaniak, "Teaching High-performance Computing Systems: A Case Study with Parallel Programming APIs: MPI, OpenMP and CUDA," in *International Conference on Computational Science*, Springer, 2024, pp. 398–412.
[9] C.-H. Chang, C.-W. Lu, C.-T. Yang, and T.-C. Chang, "An approach of performance comparisons with OpenMP and CUDA parallel programming on multicore systems," *Concurrency and Computation: Practice and Experience*, vol. 28, no. 16, pp. 4230–4245, 2016.