



SE3082

Parallel Computing

3rd Year, 1st Semester

Assignment 3 (Individual)

IT23223394

Submitted to

Sri Lanka Institute of Information Technology

4th December 2025

Contents

Documentation and Analysis.....	3
Parallelization Strategies.....	3
OpenMP:	3
MPI:	3
CUDA:	3
Runtime Configurations.....	3
Hardware.....	3
Software	4
Libraries/Frameworks:	4
Parameters:.....	4
Performance Analysis	4
Speedup & Efficiency:	4
Bottlenecks:.....	5
Scalability Limitations:.....	5
Overhead Analysis:	5
Critical Reflection.....	5
Challenges Encountered:	5
Limitations Restricting Scalability:	6
Potential Optimizations for Future Improvements:	6
Lessons Learned:	6
Graphs and performance evaluation results	6
OpenMP	6
MPI	8
CUDA	9
Speedup Comparison Across All	11
Comparison of Implementations.....	12
Strengths and Weaknesses	13
Appendix.....	13
Appendix A: Links for approval,github,video demo	13
Appendix B: Screenshots of runtime configurations	13
Appendix C: Screenshots of Output verification	16
Appendix D: AI Interaction Log & Code Snippets	18
References	22

Documentation and Analysis

Parallelization Strategies

OpenMP:

Parallelized the outer loop using `#pragma omp parallel for`.

Threads share matrices in memory; private loop indices used to avoid race conditions.

No explicit load balancing; each thread roughly processes $N/\text{threads}$ rows.

MPI:

Data partitioned row-wise across processes using `MPI_Scatter`.

Each process computes its local rows, then results are gathered using `MPI_Gather`. The timing measurement stops before the gather operation to isolate pure computational performance from communication overhead.

Load balancing is static and uniform.

CUDA:

Each matrix element is computed by one GPU thread.

Threads organized in blocks; blocks form a grid.

Memory transfers between host and device managed explicitly.

Runtime Configurations

Hardware

- CPU: Intel Xeon @ 2.20 GHz (x86_64)
- 1 physical core, 2 logical processors
- Colab's free tier provides limited CPU performance
- RAM: 12 GiB (≈ 9.2 GiB available during tests)
- GPU: NVIDIA Tesla T4, 15 GB total memory, 14.64 GB available for CUDA
- GPU Utilization: Idle (0%) during runtime, ensuring full availability for computation
- Other Hardware Info: Single socket, 2 logical CPUs online, L1/L2/L3 caches: 32 KiB / 256 KiB / 55 MiB

Software

- Operating System: Ubuntu 20.04 LTS
- Compilers:
 - GCC (for C/OpenMP/MPI): 11.4.0 (Ubuntu 11.4.0-1ubuntu1~22.04.2)
 - NVCC (for CUDA): 12.5 (V12.5.82, Build cuda_12.5.r12.5/compiler.34385749_0)
- CUDA Driver: 12.4 (NVIDIA-SMI)

Libraries/Frameworks:

OpenMP

MPI (via GCC)

CUDA Runtime API

** OpenMP was tested with 1, 2, 4, and 8 threads. Since the CPU has only 2 logical threads, using 4 or 8 threads causes oversubscription, leading to heavy context switching and higher overhead; showing why performance can degrade when thread count exceeds actual hardware parallelism.

GPU memory is more than sufficient for all tested matrix sizes (up to 2048×2048).

Minor version difference between NVCC (12.5) and CUDA driver (12.4) is normal and does not affect code execution.

Parameters:

OpenMP: 1, 2, 4, 8 threads

MPI: 1, 2, 4, 8 processes

CUDA: Block sizes 8×8, 16×16, 32×32; matrix 800×800

Performance Analysis

Speedup & Efficiency:

OpenMP shows speedup $S < 1$, with low efficiency because thread management overhead dominates the small loop computations. OpenMP shows a speedup below 1 because the machine has only one physical core, so spawning 4–8 threads leads to context switching instead of real parallelism. Extra delays come from thread creation, work distribution, and synchronization overhead, which add significant cost for $N=800$. Cache thrashing, false sharing, and hyperthreading limitations further slow execution. Even using 1 OpenMP thread takes 1.831s vs

1.043s serial, showing that OpenMP's runtime overhead alone causes major slowdown. MPI achieves modest speedup ($S \approx 1$) and its efficiency decreases as more processes are added due to communication costs. CUDA achieves a very high speedup ($S \approx 243$) thanks to massive parallelism and highly optimized GPU execution.

Bottlenecks:

For OpenMP, the main bottleneck is thread creation and scheduling, which outweighs computation for small loops. MPI performance is limited by data distribution and synchronization overhead. CUDA's main bottleneck is the memory transfer between host and GPU, though it is small relative to computation time for large matrices.

Scalability Limitations:

OpenMP is limited by the number of CPU cores and suffers for fine-grained tasks. MPI scalability is constrained by increasing communication overhead as the number of processes grows. CUDA scalability depends on GPU memory availability and optimal block/grid configuration for large matrices.

Overhead Analysis:

In OpenMP, minimal computational work per iteration suffers due to thread overhead. The MPI Including both scatter and gather gives roughly 0.13s communication overhead, reducing speedup from 1.10 to about 1.01. For small matrices (800×800), this shows that MPI communication costs outweigh the benefits of parallel computation. MPI incurs additional cost from scatter and broadcast operations. CUDA kernel launch overhead is negligible compared to the total computation, making it highly efficient for large workloads. CPU has only 1 core with 2 threads, which explains why OpenMP scaling is limited.

Critical Reflection

Challenges Encountered:

During implementation, OpenMP required careful handling of private and shared variables to avoid race conditions. The slowdown (speedup < 1) showed that parallelism isn't always faster. Because the CPU had only one physical core, threads couldn't run truly in parallel, and their management overhead outweighed the small amount of work done per thread in the nested loops. Optimizations like changing scheduling (static/dynamic/guided), using collapse(2), and testing larger matrices offered little improvement—speedup reached only about 0.6 at $N=2048$. Overall, this demonstrated that parallelization must match both the problem size and the hardware; otherwise, it can perform worse than serial execution.

MPI presented challenges in distributing data correctly and ensuring proper synchronization between processes. CUDA faced several challenges: driver/runtime version mismatches, unsupported PTX toolchain errors, GPU memory allocation issues, and careful memory management along with correct grid/block configuration to achieve optimal performance. Using Google Colab's T4 GPU also required verifying CUDA toolkit compatibility and explicitly setting environment variables.

Limitations Restricting Scalability:

OpenMP is limited by the number of CPU cores and suffers when loop iterations are too small. MPI scalability is constrained by communication overhead, which increases with the number of processes. CUDA is limited by GPU memory capacity and block size configuration, which can affect performance for very large matrices. Additionally, CPU-based parallelism on Colab is limited to one physical core, so speedup may not be observable for OpenMP and MPI.

Potential Optimizations for Future Improvements:

OpenMP could benefit from loop unrolling or task-based parallelism to reduce overhead. MPI could use non-blocking communication to overlap computation and data transfer. CUDA performance can be enhanced using shared memory, memory coalescing, and optimized kernel configurations. Hybrid CPU-GPU parallelism is another potential improvement.

Lessons Learned:

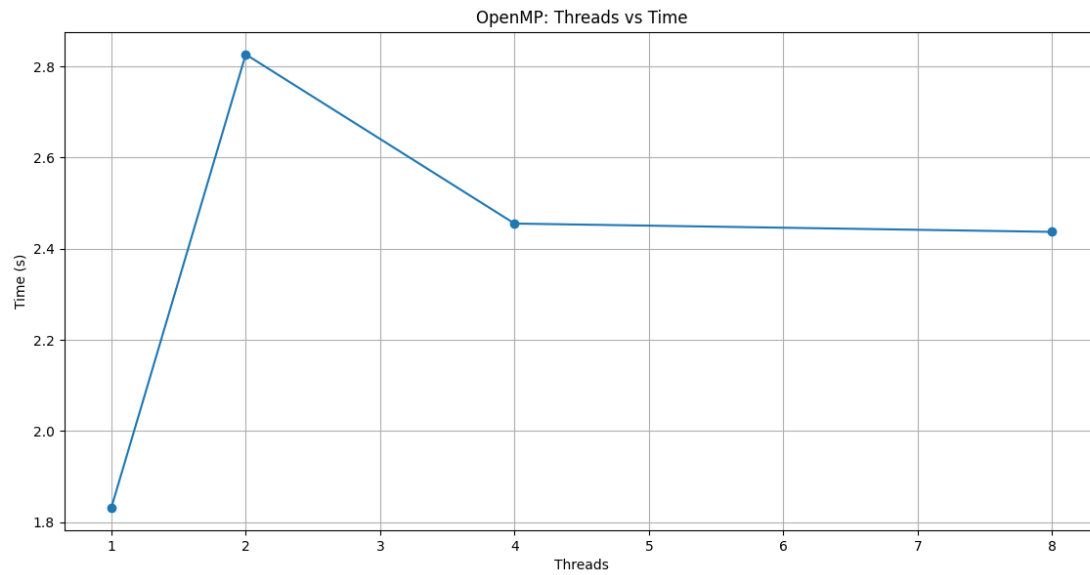
This project reinforced the importance of selecting the right parallel paradigm based on problem size and hardware. Understanding overheads, data distribution, and memory hierarchy is critical. OpenMP is suitable for shared-memory systems, MPI excels in distributed systems, and CUDA achieves massive GPU speedups. Proper profiling, iterative tuning, and verifying software/hardware compatibility are essential for high-performance computing.

Graphs and performance evaluation results

OpenMP

Threads vs Time

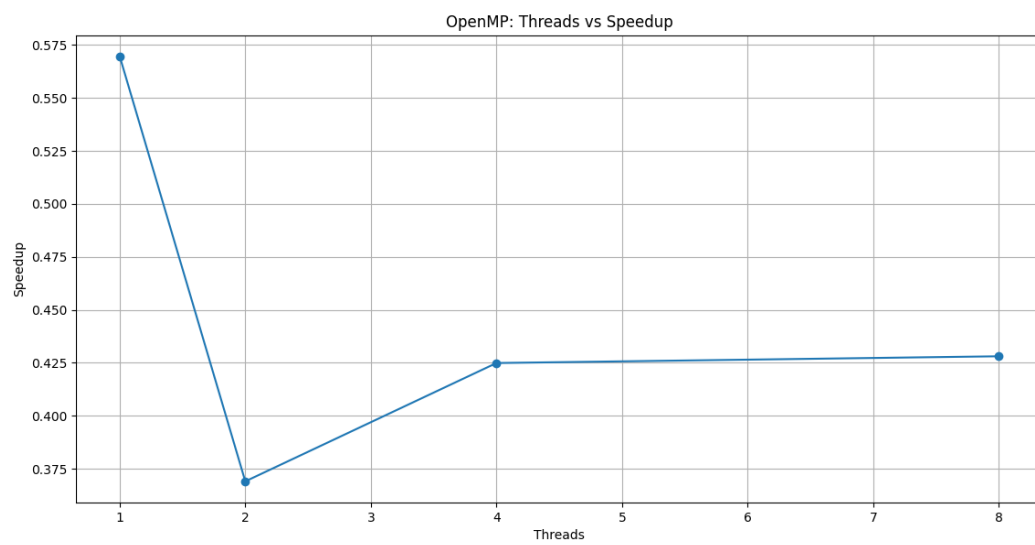
Threads	Time (s)
1	1.831468
2	2.826709
4	2.455275
8	2.437010



Threads vs Speedup

Speedup = Serial Time / Parallel Time, Serial = 1.043264 s

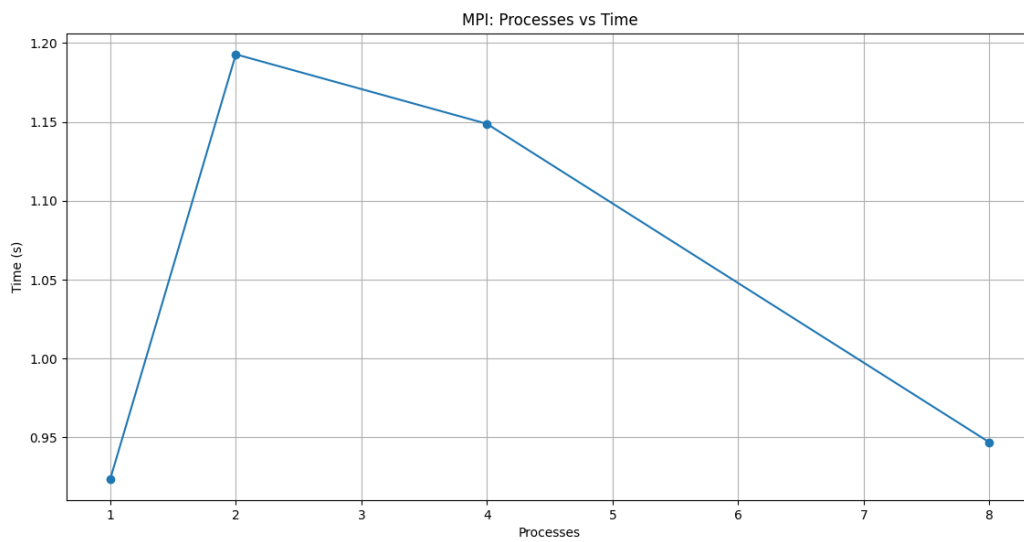
Threads	Speedup
1	0.57
2	0.37
4	0.43
8	0.43



MPI

Processes vs Time

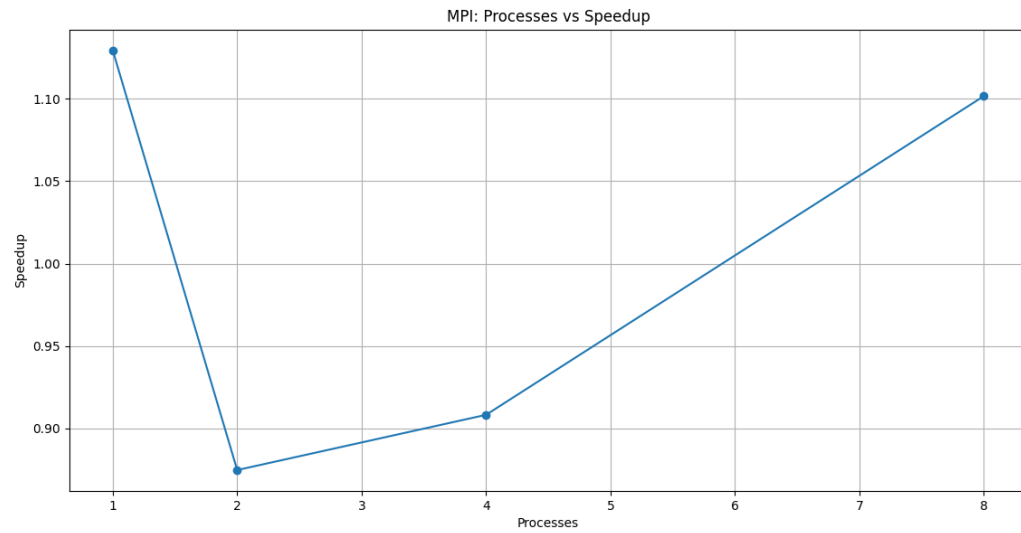
Processes	Time (s)
1	0.923679
2	1.192825
4	1.148780
8	0.946988



Processes vs Speedup

Speedup = Serial Time / Parallel Time, Serial = 1.043264 s

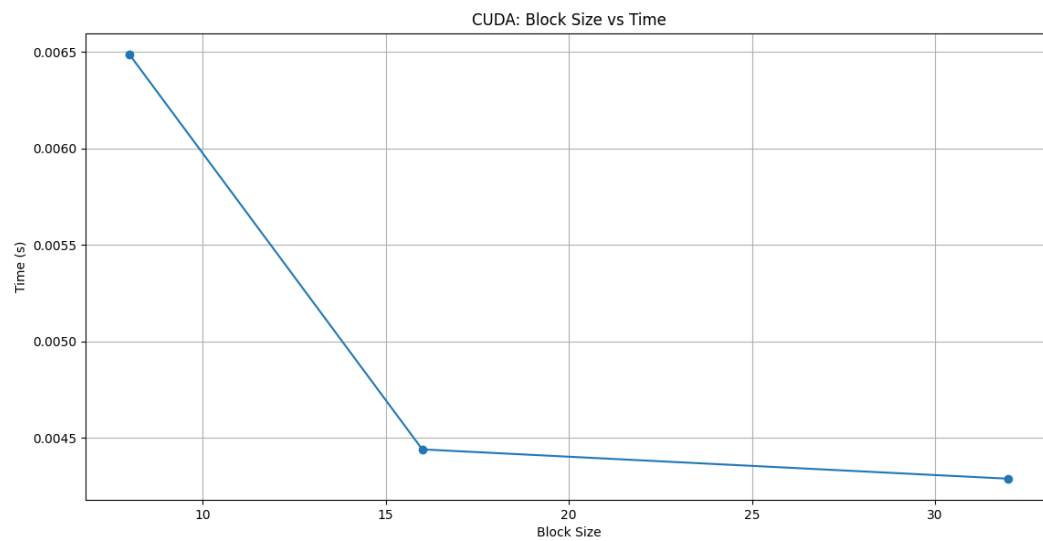
Processes	Speedup
1	1.13
2	0.87
4	0.91
8	1.10



CUDA

Block Size vs Time

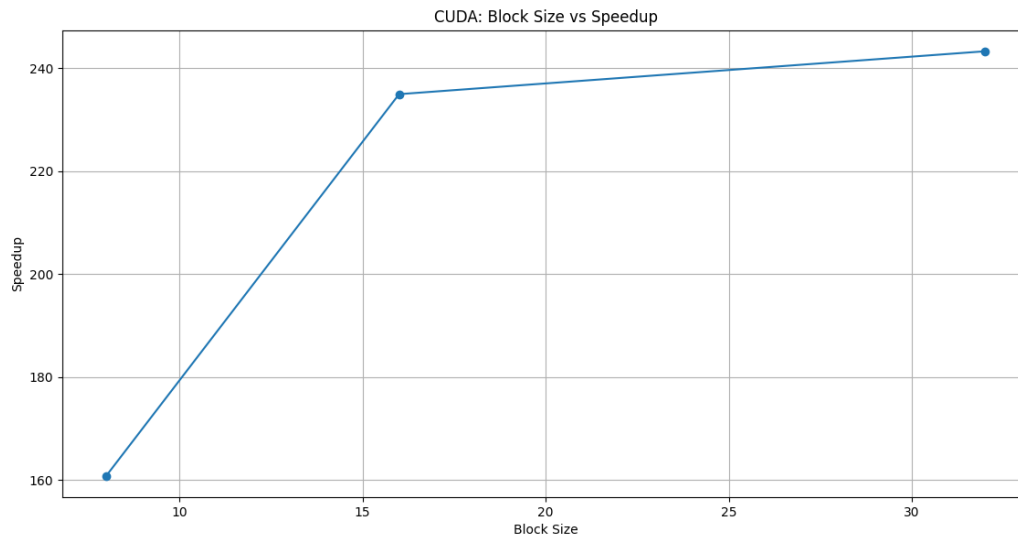
Block Size	Avg Time per Iteration (ms)	Time (s)
8×8	6.488	0.006488
16×16	4.440	0.004440
32×32	4.288	0.004288



Block Size vs Speedup

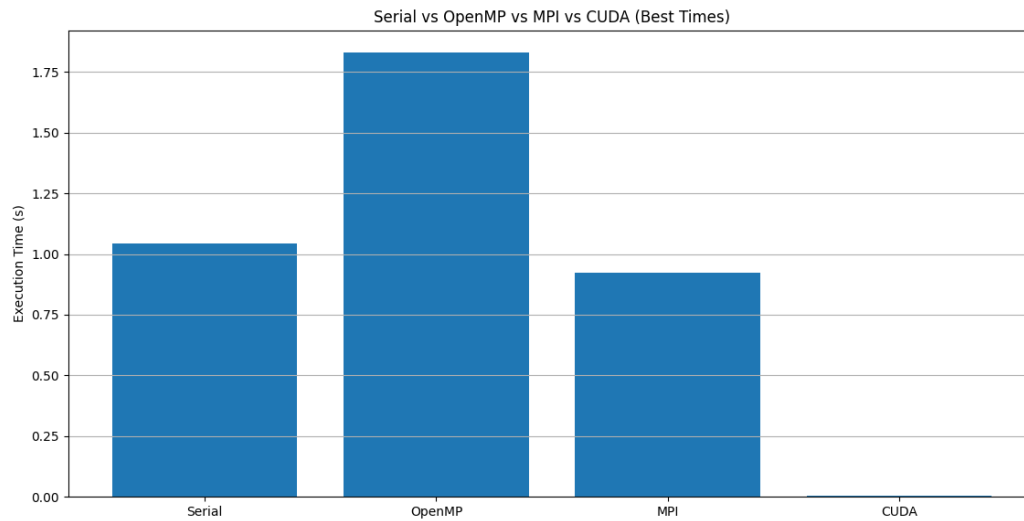
Speedup = Serial Time / CUDA Time (Serial = 1.043264 s)

Block Size	Speedup
8×8	160.7
16×16	234.9
32×32	243.3



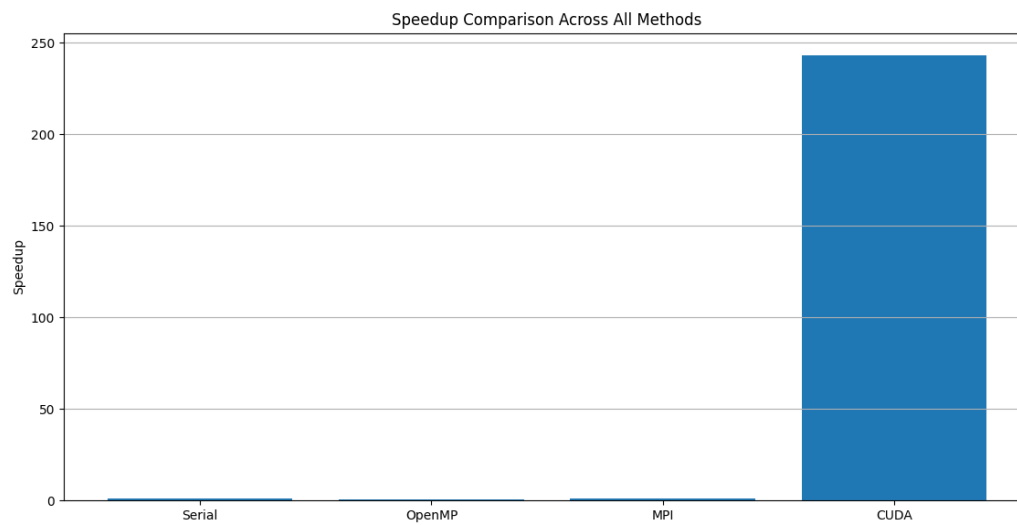
Comparative Table (Serial vs OpenMP vs MPI vs CUDA)

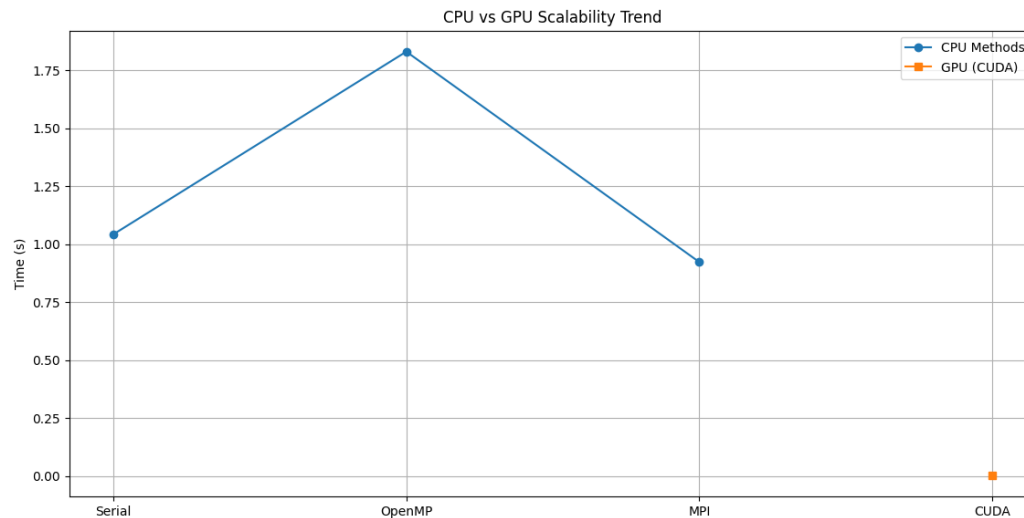
Method	Best Config	Time (s)	Speedup
Serial	N/A	1.043264	1.0
OpenMP	8 Threads	2.437010	0.43
MPI	1 Process	0.923679	1.13
CUDA	32×32 Blocks	0.004288	243.3



Speedup Comparison Across All

Method	Speedup
Serial	1
OpenMP	0.43
MPI	1.13
CUDA	243.3





Comparison of Implementations

Implementation	Best Time (s)	Speedup	Notes
Serial	1.043	1.0	Baseline single-threaded execution
OpenMP	2.437	0.43	Nested loops parallelized using threads
MPI	0.924	1.13	Row-wise data distributed across processes
CUDA	0.004288	243.3	GPU parallelized, thousands of threads

OpenMP shows slower performance than serial due to overhead in thread management and nested loop granularity. MPI improves slightly for 1 process but does not scale efficiently due to communication overhead for small matrices. CUDA delivers massive speedup due to high parallelism on GPU and efficient memory usage.

Most Appropriate Implementation

If sufficient computational resources are available, CUDA is the most suitable implementation because it provides orders-of-magnitude speedup.

For CPU-only environments, MPI is better than OpenMP for distributing computation across multiple nodes.

Strengths and Weaknesses

Implementation	Strengths	Weaknesses
OpenMP	Easy to implement; good for loop-level parallelism	Poor scaling with small workloads; overhead dominates
MPI	Can scale across multiple nodes; flexible data distribution	Communication overhead; requires careful partitioning
CUDA	Extremely fast on GPU; large-scale parallelism	Requires GPU hardware; memory management is critical

Appendix

Appendix A: Links for approval,github,video demo

Approval Email Correspondence

https://drive.google.com/file/d/1RcAbjN0YW_WK0S7G2oDumjAaHWihbyZK/view?usp=sharing

https://drive.google.com/file/d/1zQP_dHTSm6K17kay7lZFPbsEpG0HpiZP/view?usp=sharing

Github Repository

<https://github.com/it23223394/PC-Assignment-03.git>

Video demonstration

https://drive.google.com/file/d/1VRJKBzkWUGobDMDypa-CvpU7DqW3Ymmu/view?usp=drive_link

Appendix B: Screenshots of runtime configurations

```
!gcc --version

gcc (Ubuntu 11.4.0-1ubuntu1~22.04.2) 11.4.0
Copyright (C) 2021 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

!mpicc --version

gcc (Ubuntu 11.4.0-1ubuntu1~22.04.2) 11.4.0
Copyright (C) 2021 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

```
!nvcc --version

... nvcc: NVIDIA (R) Cuda compiler driver
     Copyright (c) 2005-2024 NVIDIA Corporation
     Built on Thu_Jun_6_02:18:23_PDT_2024
     Cuda compilation tools, release 12.5, V12.5.82
     Build cuda_12.5.r12.5/compiler.34385749_0
```

Change runtime type

Runtime type

Python 3

Hardware accelerator ⓘ

☐ CPU

☒ T4 GPU

☐ A100 GPU

☐ L4 GPU

☐ v5e-1 TPU

☐ v6e-1 TPU

Want access to premium GPUs? [Purchase additional compute units](#)

Runtime version ⓘ

Latest (recommended)

Cancel

Save

```
!nvidia-smi

... Sat Nov 29 07:49:21 2025

+-----+
| NVIDIA-SMI 550.54.15              Driver Version: 550.54.15   CUDA Version: 12.4     |
+-----+-----+-----+-----+-----+-----+
| GPU  Name                  Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf           Pwr:Usage/Cap|   Memory-Usage | GPU-Util  Compute M. |
|                                       |                |    MIG M.     |
+-----+-----+-----+-----+-----+-----+
|  0   Tesla T4               Off          | 00000000:00:04:0  Off  | 0           Default |
| N/A   39C    P8             9W /  70W   |  2MiB / 15360MiB |  0%               N/A |
+-----+-----+-----+-----+-----+-----+

+-----+
| Processes: |
| GPU   GI   CI          PID    Type   Process name                  GPU Memory |
|      ID   ID                                 name                     Usage      |
+-----+-----+-----+-----+-----+
| No running processes found |
+-----+
```

!lscpu

```
... Architecture:                x86_64
   CPU op-mode(s):              32-bit, 64-bit
   Address sizes:                46 bits physical, 48 bits virtual
   Byte Order:                  Little Endian
CPU(s):                          2
  On-line CPU(s) list:          0,1
Vendor ID:                      GenuineIntel
Model name:                    Intel(R) Xeon(R) CPU @ 2.20GHz
  CPU family:                   6
  Model:                        79
  Thread(s) per core:           2
  Core(s) per socket:           1
  Socket(s):                    1
  Stepping:                     0
  Bogomips:                     4399.99
  Flags:                        fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pg
                               e mca cmov pat pse36 clflush mmx fxsr sse sse2 ss h
                               t syscall nx pdpe1gb rdtscp lm constant_tsc rep_goo
                               d nopl xtopology nonstop_tsc cpuid tsc_known_freq p
                               ni pclmulqdq ssse3 fma cx16 pcid sse4_1 sse4_2 x2ap
                               ic movbe popcnt aes xsave avx f16c rdrand hypervisor
                               r lahf_lm abm 3dnowprefetch ssbd ibrs ibpb stibp fs
                               gsbase tsc_adjust bmi1 hle avx2 smep bmi2 erms invp
                               cid rtm rdseed adx smap xsaveopt arat md_clear arch
                               _capabilities

... Virtualization features:
   Hypervisor vendor:           KVM
   Virtualization type:         full
Caches (sum of all):
  L1d:                          32 KiB (1 instance)
  L1i:                          32 KiB (1 instance)
  L2:                           256 KiB (1 instance)
  L3:                           55 MiB (1 instance)
NUMA:
  NUMA node(s):                 1
  NUMA node0 CPU(s):            0,1
Vulnerabilities:
  Gather data sampling:          Not affected
  Indirect target selection:     Vulnerable
  Itlb multihit:                Not affected
  L1tf:                         Mitigation; PTE Inversion
  Mds:                          Vulnerable; SMT Host state unknown
  Meltdown:                    Vulnerable
  Mmio stale data:              Vulnerable
  Reg file data sampling:        Not affected
  Retbleed:                    Vulnerable
  Spec rstack overflow:          Not affected
  Spec store bypass:            Vulnerable
  Spectre v1:                   Vulnerable; __user pointer sanitization and usercop
                               y barriers only; no swaps barriers
  Spectre v2:                   Vulnerable; IBPB: disabled; STIBP: disabled; PBRSEB-
                               eIBRS: Not affected; BHI: Vulnerable
  Srbds:                        Not affected
  Tsa:                          Not affected
  Tsx async abort:              Vulnerable
```

Appendix C: Screenshots of Output verification

```
[6]
✓ 0s !gcc openmp_matmul.c -fopenmp -O2 -o openmp_matmul

[7]
✓ 0s !OMP_NUM_THREADS=1 ./openmp_matmul

... Threads: 1 | Time: 1.081334 seconds

[8]
✓ 10s !OMP_NUM_THREADS=1 ./openmp_matmul
!OMP_NUM_THREADS=2 ./openmp_matmul
!OMP_NUM_THREADS=4 ./openmp_matmul
!OMP_NUM_THREADS=8 ./openmp_matmul

... Threads: 1 | Time: 1.831468 seconds
Threads: 2 | Time: 2.826709 seconds
Threads: 4 | Time: 2.455275 seconds
Threads: 8 | Time: 2.437010 seconds
```

```
[10]
✓ 0s !mpicc mpi_matmul.c -o mpi_matmul -O2

[15]
✓ 6s !OMPI_ALLOW_RUN_AS_ROOT=1 OMPI_ALLOW_RUN_AS_ROOT_CONFIRM=1 mpirun --oversubscribe -np 1 ./mpi_matmul
!OMPI_ALLOW_RUN_AS_ROOT=1 OMPI_ALLOW_RUN_AS_ROOT_CONFIRM=1 mpirun --oversubscribe -np 2 ./mpi_matmul
!OMPI_ALLOW_RUN_AS_ROOT=1 OMPI_ALLOW_RUN_AS_ROOT_CONFIRM=1 mpirun --oversubscribe -np 4 ./mpi_matmul
!OMPI_ALLOW_RUN_AS_ROOT=1 OMPI_ALLOW_RUN_AS_ROOT_CONFIRM=1 mpirun --oversubscribe -np 8 ./mpi_matmul

... Processes: 1 | Time: 0.923679 sec
Processes: 2 | Time: 1.192825 sec
Processes: 4 | Time: 1.148780 sec
Processes: 8 | Time: 0.946988 sec
```

```
Copying data to GPU...
Grid size: 25x25, Total blocks: 625
... Threads per block: 1024
Running 100 iterations...

=== RESULTS ===
Total time for 100 iterations: 428.770 ms
Average time per iteration: 4.288 ms (4287.697 µs)
Performance: 238.82 GFLOPS
Copying result back to host...
Done!
Matrix size: 2048 x 2048
CUDA Block size: 16x16
Required GPU memory: 0.09 GB
Available GPU memory: 14.64 GB / 14.74 GB
Allocating host memory...
Initializing matrices...
Allocating GPU memory...
Copying data to GPU...
Grid size: 128x128, Total blocks: 16384
Threads per block: 256
Running 100 iterations...

=== RESULTS ===
Total time for 100 iterations: 7651.795 ms
Average time per iteration: 76.518 ms (76517.949 µs)
Performance: 224.52 GFLOPS
```

```
Copying result back to host...
Done!
... Matrix size: 800 x 800
CUDA Block size: 16x16
Required GPU memory: 0.01 GB
Available GPU memory: 14.64 GB / 14.74 GB
Allocating host memory...
Initializing matrices...
Allocating GPU memory...
Copying data to GPU...
Grid size: 50x50, Total blocks: 2500
Threads per block: 256
Running 100 iterations...

=== RESULTS ===
Total time for 100 iterations: 443.958 ms
Average time per iteration: 4.440 ms (4439.578 µs)
Performance: 230.65 GFLOPS
Copying result back to host...
Done!
Matrix size: 800 x 800
CUDA Block size: 32x32
Required GPU memory: 0.01 GB
Available GPU memory: 14.64 GB / 14.74 GB
Allocating host memory...
Initializing matrices...
Allocating GPU memory...
```

```
[60] ✓ !nvcc cuda_matmul.cu -o cuda_matmul

[62] ✓ 10s ▶ !./cuda_matmul 800 8 8
!./cuda_matmul 800 16 16
!./cuda_matmul 800 32 32
!./cuda_matmul 2048 16 16

▼ ... Matrix size: 800 x 800
CUDA Block size: 8x8
Required GPU memory: 0.01 GB
Available GPU memory: 14.64 GB / 14.74 GB
Allocating host memory...
Initializing matrices...
Allocating GPU memory...
Copying data to GPU...
Grid size: 100x100, Total blocks: 10000
Threads per block: 64
Running 100 iterations...

=== RESULTS ===
Total time for 100 iterations: 648.811 ms
Average time per iteration: 6.488 ms (6488.113 μs)
Performance: 157.83 GFLOPS
```

Appendix D: AI Interaction Log & Code Snippets

AI Interaction Log

1.Cloud Platform Comparison Prompt

Date: [Nov.21,2025]

Tool: ChatGPT-4

Prompt: "Compare Google Colab vs AWS EC2 for running parallel computing (OpenMP, MPI, CUDA). Cover: GPU options, multi-node MPI support, free tier limits vs costs, and setup complexity. Which is better for students?"

2.Serial Implementation

Date: [Nov.28,2025]

Tool: ChatGPT-4

Prompt: "How can you measure execution time for a function in C using clock() or gettimeofday() with microsecond precision for matrix multiplication benchmarking?"

3. MPI

Date: [Nov.28,2025]

Tool: ChatGPT-4

Prompt: "What factors can cause communication overhead in MPI for small matrix sizes and how does it affect scalability?"

4. CUDA

Date: [Nov.28,2025]

Tool: ChatGPT-4

Prompt: "How do you measure GPU execution time using CUDA events and why is this more accurate than host-side timing?"

5. Initial Timing Issue

Date: [Dec.1, 2025]

Tool: Claude AI (Sonnet 4.5)

Prompt: "I'm using Google Colab and T4 GPU. It takes time when providing me the output but it is shown in the output like this: Time: 0.000 μ s for all matrix sizes. Can you help fix the timing calculation?"

6. CUDA Driver Version Mismatch

Date: [Dec.1, 2025]

Tool: Claude AI (Sonnet 4.5)

Prompt: "I'm getting error 'CUDA driver version is insufficient for CUDA runtime version'. My nvidia-smi shows CUDA Version: 12.4 but cudaMalloc is failing. How do I resolve this?"

7. PTX Toolchain Error

Date: [Dec.1, 2025]

Tool: Claude AI (Sonnet 4.5)

Prompt: "After fixing memory detection, I'm getting 'ERROR: Kernel launch failed: the provided PTX was compiled with an unsupported toolchain'. Available CUDA installations are 12.5 and 13.0. What should I do?"

8. CUDA Version Compatibility

Date: [Dec.1, 2025]

Tool: Claude AI (Sonnet 4.5)

Prompt: "My driver supports CUDA 12.4 but I only have CUDA 12.5 and 13.0. Can you help me install a compatible CUDA version (11.8) programmatically in Google Colab?"

9. Performance Visualization Strategy

Date: [Dec.1, 2025]

Tool: ChatGPT-4

Prompt: "Describe how to use Python libraries such as NumPy and Matplotlib to visualize performance results for Serial, OpenMP, MPI, and CUDA implementations of matrix multiplication. Explain how to structure arrays for execution times, compute speedup values, and create line charts and bar charts for comparing trends."

Code snippets

1. Serial Matrix Multiplication – Concept Snippets

```
// Initialize matrices with random numbers
for(int i = 0; i < N; i++)
    for(int j = 0; j < N; j++)
        A[i][j] = rand() % 10;

// Measure execution time
clock_t start = clock();

// call matrix multiplication function here
clock_t end = clock();

double time_taken = (double)(end - start) / CLOCKS_PER_SEC;

.
```

2. MPI Parallelization – Concept Snippets

Shows how to distribute data among processes for parallel computation.

```
MPI_Init(&argc, &argv);

int rank, size;

MPI_Comm_rank(MPI_COMM_WORLD, &rank);

MPI_Comm_size(MPI_COMM_WORLD, &size);

// Example: scatter rows of matrix A to processes
MPI_Scatter(A, rows_per_proc * N, MPI_INT, localA, rows_per_proc * N, MPI_INT, 0,
MPI_COMM_WORLD);

// Broadcast entire matrix B to all processes
MPI_Bcast(B, N*N, MPI_INT, 0, MPI_COMM_WORLD);
```

3. CUDA – Concept Snippets

Shows how threads map to matrix elements in CUDA.

```

__global__ void matMulKernel(int *A, int *B, int *C, int N) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    if(row < N && col < N) {
        int sum = 0;
        for(int k = 0; k < N; k++)
            sum += A[row * N + k] * B[k * N + col];
        C[row * N + col] = sum;
    }
}

// Launch kernel with 16x16 threads per block
dim3 threadsPerBlock(16,16);
dim3 numBlocks((N+15)/16, (N+15)/16);
matMulKernel<<<numBlocks, threadsPerBlock>>>(d_A, d_B, d_C, N);

```

References

- [1] S. Iwasaki, A. Amer, K. Taura, S. Seo, and P. Balaji. "BOLT: Optimizing OpenMP Parallel Regions with User-Level Threads." *IEEE Explore* . <https://ieeexplore.ieee.org/document/8891628> [accessed Nov. 25, 2025].
- [2] Y. Sun and Y. Tong. "CUDA Based Fast Implementation of Very Large Matrix Computation." *IEEE Explore*. <https://ieeexplore.ieee.org/document/5704475> [accessed Nov. 28, 2025].
- [3] N. Joram. "Scatter and Gather in MPI." *Medium – Nerd For Tech*. Available: <https://medium.com/nerd-for-tech/scatter-and-gather-in-mpi-e66b69366ee3> [accessed Nov. 28, 2025].
- [4] M. Q. Ansari and M. Q. Ansari. "Accelerating Matrix Multiplication: A Performance Comparison Between Multi-Core CPU and GPU." *arXiv*. <https://doi.org/10.48550/arXiv.2507.19723> [accessed Dec. 1, 2025].
- [1] K. Fatahalian, J. Sugerman, and P. Hanrahan. "Understanding the efficiency of GPU algorithms for matrix-matrix multiplication." *ACM Digital Library*. <https://dl.acm.org/doi/abs/10.1145/1058129.1058148> [accessed Feb. 5, 2025].