Sri Lanka Institute of Information Technology

BSc (Hons) in Computer Science

Year 3 Semester 1

SE3082 - Parallel Computing

Assignment 3 - Parallelized Matrix Multiplication

Student ID	IT23292154
Name	W. M. Chamudini

Contents

1. Introduction

Matrix multiplication is a core computational task in scientific computing, machine learning, computer graphics, engineering simulations, numerical analysis and data-intensive applications. The traditional matrix multiplication algorithm has a computational complexity of $O(n^3)$, making it an important benchmark for evaluating the efficiency of parallel programming models. Its high arithmetic intensity and predictable computation patterns make it suitable for studying CPU multithreading, distributed processing, and GPU acceleration.

This assignment implements matrix multiplication using four paradigms:

1. **Serial C Implementation**
2. **OpenMP (Shared-Memory Parallelism)**
3. **MPI (Distributed-Memory Parallelism)**
4. **CUDA (GPU-based Parallelism)**

All implementations follow the same operational flow:

- Matrices A and B are initialized deterministically.
- The result matrix C is computed using A x B.
- Row-wise checksums and global checksum validate correctness.
- Execution time is measured for performance comparison.

A key feature in all implementations is <u>controlled output printing:</u>

- Full matrices are printed only when all dimensions ≤ 10.
- Row sums are printed only when results matrix(C) dimensions ≤ 50.
- Otherwise, only the summary and checksum are displayed.

  This ensures readability while supporting extremely large matrices.

The objective is to evaluate the performance, parallelization efficiency, scalability and trade-offs of each model.

2. Parallelization Strategies

2.1 Serial Implementation

The serial version uses the classical triple-nested loop:

```
for (int i=0;i<N_A_row;++i) {
  for (int j=0;j<N_B_col;++j) {
    double sum = 0.0;
    for (int k=0;k<N_A_col;++k) {
      sum += A[(long) i * N_A_col + k] * B[(long) k * N_B_col + j];
    }
    C[(long) i * N_B_col + j] = sum;
  }
}
```

Key Decisions

- Row-major indexing for efficient memory access.
- clock() timing for measuring runtime.
- Deterministic initialization, allowing checksum validation.
- **Output control rules**:
  Full matrices printed only when dimensions ≤ 10.
  Row sums printed only when dimensions ≤ 50.
  Otherwise, only global checksum displayed.

The serial version acts as the baseline for speedup comparisons in OpenMP, MPI and CUDA.

2.2 OpenMP Implementation

OpenMP parallelizes the multiplication on a shared-memory CPU using multithreading.

Parallelization Strategy

The approach parallelizes the outer loop (rows of A) using:

```
#pragma omp parallel
#pragma omp for schedule(static)
```

Why This Works

- Each thread writes to a distinct set of rows in C → no race conditions.
- A and B are shared in memory → minimal overhead.
- <u>Static scheduling</u> evenly divides rows among threads.
- Very efficient up to the number of physical cores.

Advantages

- Minimal synchronization.
- Easy to implement.
- Scales well on multi-core CPUs.

Limitations

- Performance limited by CPU core count.
- Thread overhead noticeable for small matrices.

2.3 MPI Implementation

MPI applies distributed-memory parallelism, suitable for multi-node systems or simulated multi-process environments.

Parallelization Strategy

- Process 0 initializes A and B.
- B is broadcast to all processes using MPI_Bcast().
- Rows of A are divided manually and sent using MPI_Send() / MPI_Recv().
- Each process computes its portion of C.
- Results gathered using MPI_Gatherv(), supporting uneven partition sizes.

Why This Works

- Perfect row-wise parallelism.
- Each process handles separate matrix blocks → no data conflicts.
- Scales across machines or clusters.

Challenges

- Communication overhead.
- Non-uniform row distribution requires careful offset management.
- MPI_Gatherv needed for uneven distributions.

Advantages

- Ideal for distributed systems.
- Scales beyond single-machine CPU limitations.

2.4 CUDA Implementation

CUDA offloads the computation to the GPU, using thousands of lightweight threads.

Parallelization Strategy

- Each GPU thread computes one element of matrix C.
- Threads are arranged in 2D blocks and grids.
- Computation inside kernel performs the dot product.

Key GPU Concepts Used

- Thread indexing via blockIdx, threadIdx.
- Block size tuning (1×1, 2×2, 4×4, 8×8, …).
- High memory bandwidth utilization.

Advantages

- Excellent performance for large matrices.
- Massive parallelism from GPU SMs.

Challenges

- Kernel launch overhead makes small matrices slow.
- GPU memory limits extremely large matrices.
- Efficient block size selection is critical ("occupancy tuning").

3. Runtime Configurations

3.1 Hardware Configuration

- **CPU:** Intel 11th Gen i5-1145G7
- **Cores:** 4 cores / 8 threads
- **RAM:** 8 GB (system), 16 GB (VM total)
- **GPU:** NVIDIA Tesla T4 - 16 GB VRAM (Google Colab)

3.2 Software Environment

- **OS:** Windows 11 (WSL2)
- **Linux Subsystem:** Ubuntu 24.04.1 LTS
- **Compilers:**
    GCC 13.3.0 (OpenMP built-in)
    MPICH 4.2.0 / OpenMPI 4.1.6
    NVCC 12.5
- **Tools:**
    mpirun for MPI execution
    CUDA Runtime API
- **Platforms Used**: Local (WSL2) for OpenMP/MPI, Google Colab for CUDA

3.3 Compilation Summary

Implementation	Command
Serial	gcc serial_mat_mul.c -o serial_mat_mul
OpenMP	gcc -O3 -fopenmp openmp_mat_mul.c -o openmp_mat_mul
MPI	mpicc -O3 mpi_mat_mul.c -o mpi_mat_mul
CUDA	nvcc -O3 -arch=sm_75 cuda_mat_mul.cu -o cuda_mat_mul

4. Performance Analysis - For large sized matrices

Serial C Code

```
madara@DESKTOP-95OCCOJ:~$ ./serial_mat_mul 4000 400 400 4000
Summary:
Matrix A size: 4000 x 400
Matrix B size: 400 x 4000
Result Matrix C size: 4000 x 4000
Execution time: 20.005880 seconds
Checksum (sum of all elements) = 5.821108e+18
```

4.1 OpenMP - Threads vs Execution Time

```
madara@DESKTOP-95OCCOJ:~$ gcc -O3 -fopenmp openmp_mat_mul.c -o o
penmp_mat_mul
madara@DESKTOP-95OCCOJ:~$ ./openmp_mat_mul 4000 400 400 4000 1
Summary:
Matrix A size: 4000 x 400
Matrix B size: 400 x 4000
Result Matrix C size: 4000 x 4000
Threads used: 1
Execution time: 3.139375 seconds
Checksum (sum of all elements) = 5.821108e+18
madara@DESKTOP-95OCCOJ:~$ ./openmp_mat_mul 4000 400 400 4000 2
Summary:
Matrix A size: 4000 x 400
Matrix B size: 400 x 4000
Result Matrix C size: 4000 x 4000
Threads used: 2
Execution time: 1.770128 seconds
Checksum (sum of all elements) = 5.821108e+18
madara@DESKTOP-95OCCOJ:~$ ./openmp_mat_mul 4000 400 400 4000 4
Summary:
Matrix A size: 4000 x 400
Matrix B size: 400 x 4000
Result Matrix C size: 4000 x 4000
Threads used: 4
Execution time: 0.954210 seconds
Checksum (sum of all elements) = 5.821108e+18
```
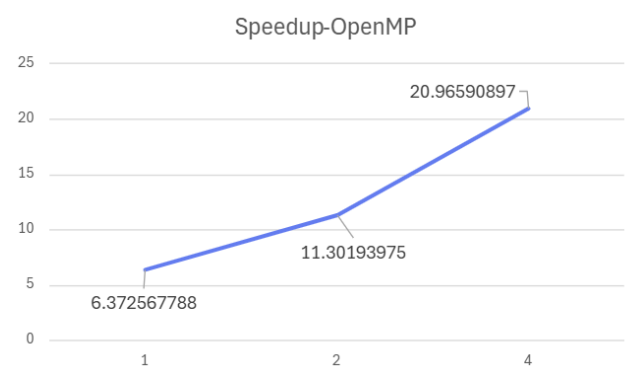
Observations:

- Significant speedup up to number of physical CPU cores.
- Hyperthreading yields diminishing returns.
- Best performance at 4 threads (based on above runtime tests).

Graphs:
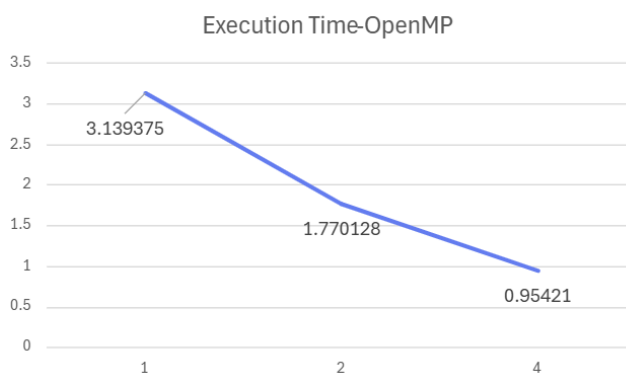
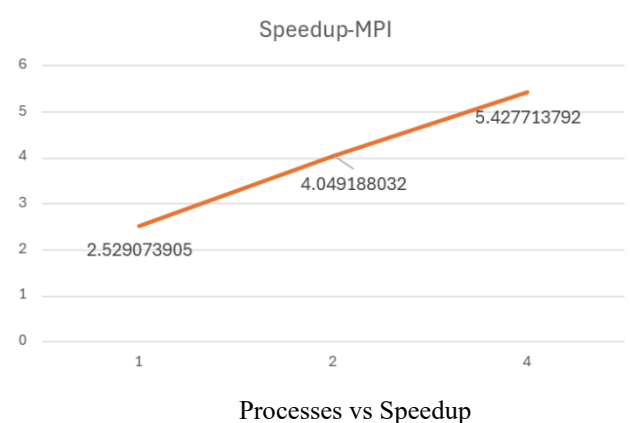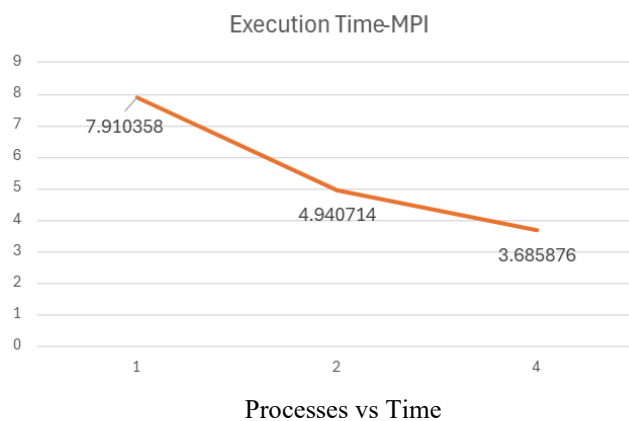Threads vs Time

Threads vs Speedup

4.2 MPI - Processes vs Speedup

```
madara@DESKTOP-95OCCOJ:~$ mpirun --allow-run-as-root --oversubsc
ribe -np 1 ./mpi_mat_mul 4000 400 400 4000
Summary:
Matrix A size: 4000 x 400
Matrix B size: 400 x 4000
Result Matrix C size: 4000 x 4000
Execution time(seconds): 7.910358
Checksum (sum of all elements)= 5.821108e+18
madara@DESKTOP-95OCCOJ:~$ mpirun --allow-run-as-root --oversubsc
ribe -np 2 ./mpi_mat_mul 4000 400 400 4000
Summary:
Matrix A size: 4000 x 400
Matrix B size: 400 x 4000
Result Matrix C size: 4000 x 4000
Execution time(seconds): 4.940714
Checksum (sum of all elements)= 5.821108e+18
madara@DESKTOP-95OCCOJ:~$ mpirun --allow-run-as-root --oversubsc
ribe -np 4 ./mpi_mat_mul 4000 400 400 4000
Summary:
Matrix A size: 4000 x 400
Matrix B size: 400 x 4000
Result Matrix C size: 4000 x 4000
Execution time(seconds): 3.685876
Checksum (sum of all elements)= 5.821108e+18
```

Observations:

- Speedup improves as process count increases.
- Beyond a point, communication overhead dominates.
- 4 processes outperform 2 due to better row distribution.

Graphs:

Processes vs Time

Processes vs Speedup

4.3 CUDA - Block Size vs Runtime

```
[6]      !./cuda_mat_mul 4000 400 400 4000 1
✓ 1s
  ∨      Summary:
         Matrix A size: 4000 x 400
         Matrix B size: 400 x 4000
         Result Matrix C size: 4000 x 4000
         Execution time(seconds): 0.767903
         Checksum (sum of all elements) = 5.821111e+18
```

```
[7]   ▶  !./cuda_mat_mul 4000 400 400 4000 2
✓ 0s
  ∨  •••  Summary:
         Matrix A size: 4000 x 400
         Matrix B size: 400 x 4000
         Result Matrix C size: 4000 x 4000
         Execution time(seconds): 0.305794
         Checksum (sum of all elements) = 5.821111e+18
```

```
[8]      !./cuda_mat_mul 4000 400 400 4000 4
✓ 0s
  ∨      Summary:
         Matrix A size: 4000 x 400
         Matrix B size: 400 x 4000
         Result Matrix C size: 4000 x 4000
         Execution time(seconds): 0.139503
         Checksum (sum of all elements) = 5.821111e+18
```

```
[9]   ▶  !./cuda_mat_mul 4000 400 400 4000 8
✓ 0s
  ∨  •••  Summary:
         Matrix A size: 4000 x 400
         Matrix B size: 400 x 4000
         Result Matrix C size: 4000 x 4000
         Execution time(seconds): 0.071058
         Checksum (sum of all elements) = 5.821111e+18
```

Observations:

- Block size 8 provided the best performance.
- Very small block sizes underutilize GPU cores.
- Very large block sizes reduce SM occupancy.

Graphs:

Block Size vs Execution Time

Block Size vs Speedup
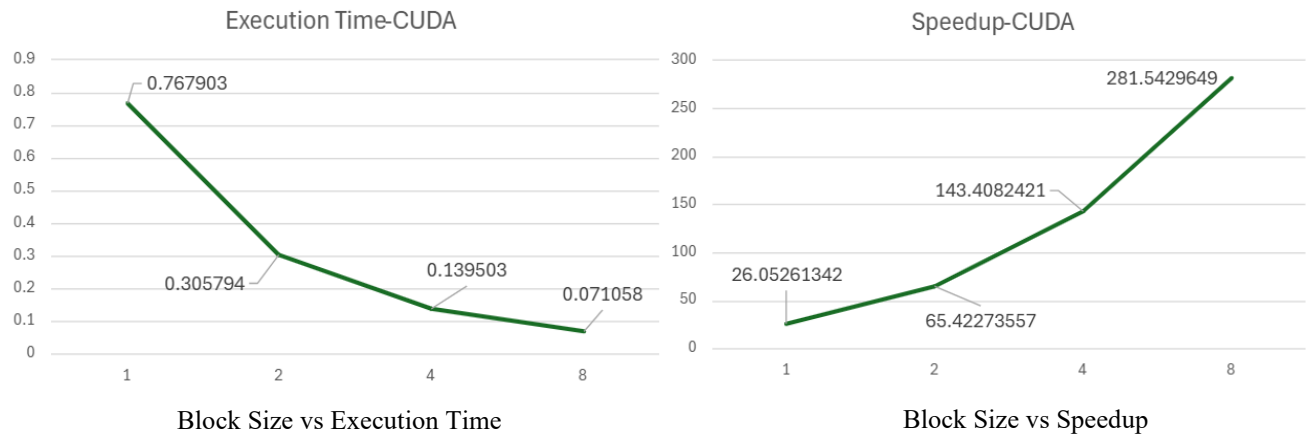
4.4 Overall Comparison

Large sized matrices (4000x400, 400x4000)			
	num	time	speedup
serial	1	20.00588	1
openmp	1	3.139375	6.372568
	2	1.770128	11.30194
	4	0.95421	20.96591
mpi	1	7.910358	2.529074
	2	4.940714	4.049188
	4	3.685876	5.427714
cuda	1	0.767903	26.05261
	2	0.305794	65.42274
	4	0.139503	143.4082
	8	0.071058	281.543

Method	Speed	Best Use Case
Serial	Slow	Small inputs / baseline
OpenMP	Fast	Multicore CPUs
MPI	Fast	Multi-node and distributed workloads
CUDA	Faster	Large matrices, GPU systems

Summary

- CUDA is the fastest for large-scale workloads.
- MPI scales well, especially on clusters.
- OpenMP is ideal for single-machine CPU parallelism.
- Serial is useful only for debugging or teaching.

5. Critical Reflection

Challenges

- Ensuring correct indexing for matrix multiplication across Serial, OpenMP, MPI, and CUDA implementations.
- Avoiding race conditions in OpenMP by assigning full rows to threads.
- Handling uneven row distribution in MPI and managing communication overhead.
- Selecting correct CUDA grid/block dimensions and managing GPU memory limits.
- Preventing excessively large outputs, which required limiting full-matrix printing.

Limitations

- Program accepts only matrix dimensions, not custom matrix data (matrices are auto generated).
- Serial version becomes slow due to $O(n^3)$ complexity.
- OpenMP performance plateaus due to memory bandwidth limitations.
- MPI overhead increases with more processes, reducing speedup.
- CUDA performance depends heavily on block size and GPU occupancy; small matrices underperform.

Future Enhancements

- Add support for user-input matrix data instead of auto generated matrices.
- Implement CUDA shared memory tiling for better performance.
- Apply SIMD vectorization and cache tiling for Serial and OpenMP.
- Use pinned memory to speed up CUDA data transfers.
- MPI_Scatterv instead of manual send/receive for improved load balancing.

Lessons Learned

- Matrix multiplication is highly parallelizable but behaves differently across architectures.
- OpenMP scales well on multicore CPUs; MPI is effective for distributed workloads.
- CUDA provides the highest performance for large matrices but needs careful tuning.
- Algorithm design must align with the strengths of each hardware platform.

6. Resources

YouTube Demonstration Video: https://youtu.be/odvmAwtQgtg

Appendices Document:
https://drive.google.com/file/d/1kbN261VmPZvQgAuIhjLBD9FTk5c3r0nM/view?usp=sharing