A MINI-PROJECT REPORT
ON

"Performance Analysis of Single-Threaded and Multithreaded Matrix Multiplication"

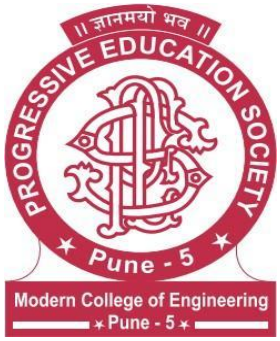
SUBMITTED TO THE SAVITRIBAI PHULE PUNE UNIVERSITY, PUNE
IN THE PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE AWARD OF THE DEGREE

BACHELOR OF COMPUTER ENGINEERING

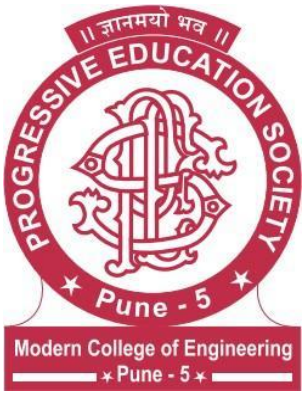
SUBMITTED BY
Taslim Ansari
Shruti Bachal
Karunya Chavan
Shashank Gosavi

Academic Year: 2025-26

FOR THE DESIGN AND ANALYSIS OF ALGORITHMS COURSE



COMPUTER ENGINEERING
P.E.S MODERN COLLEGE OF ENGINEERING
SHIVAJINAGAR, PUNE – 411005.



Progressive Education Society's Modern College of Engineering, Shivajinagar, Pune - 411005.

CERTIFICATE

This is to certify that **Karunya Chavan** from Fourth Year Computer Engineering has successfully completed his mini project titled **"Performance Analysis of Single-threaded and Multithreaded Matrix Multiplication"** in Design and Analysis of Algorithm at PES Modern College of Engineering in the partial fulfillment of the Bachelor's Degree in Computer Engineering under Savitribai Phule Pune University.

Date:

Ms. Roshani Makode
Guide

Prof. Dr. Mrs. S. A. Itkar
H.O.D
Dept. Of Computer Engineering

Acknowledgement

It gives me pleasure in presenting the mini project report on **"Performance Analysis of Single-threaded and Multithreaded Matrix Multiplication"**.

Firstly, I would like to express my indebtedness appreciation to my guide **Ms. Roshani Makode**. Her constant guidance and advice played very important role in successful completion of the report. She always gave me her suggestions, that were crucial in making this report as flawless as possible.

I would like to express our gratitude towards **Prof. Dr. Mrs. S. A. Itkar**, Head of Computer Engineering Department, PES Modern College of Engineering for her kind co-operation and encouragement which helped me during the completion of this report.

Also I wish to thank our Principal, **Prof. Dr. Mrs. K. R. Joshi** and all faculty members for their whole hearted co-operation for completion of this report. I also thank our laboratory assistants for their valuable help in laboratory.

Last but not the least, the backbone of my success and confidence lies solely on blessings of dear parents and friends.

Karunya Manoj Chavan

Table of Contents

Abstract

Matrix multiplication (C = A × B) is a fundamental operation in high-performance computing (HPC), machine learning, and scientific applications, often serving as a major computational bottleneck due to its high arithmetic intensity and memory access demands. This study presents a comparative performance analysis of three implementations: a sequential CPU baseline, a coarse-grained multithreaded CPU model (one thread per row), and a massively parallel GPU implementation using CUDA. Experiments were conducted across matrix sizes ranging from 2×2 to 512×512, measuring execution time and speedup for each approach. Results indicate that for smaller matrices ($N \leq 128$), the sequential version remains competitive due to minimal overhead, whereas the GPU outperforms all others for larger matrices ($N \geq 256$), achieving up to 69.7× speedup over the baseline. The study identifies a crossover region between N=128 and N=256, beyond which GPU acceleration becomes significantly beneficial. These findings reinforce the importance of choosing appropriate parallelization strategies based on problem size and hardware capabilities, contributing to performance-aware design in heterogeneous computing systems.

Keywords - Matrix Multiplication, Multithreading, CUDA, Parallel Computing, GPU Acceleration, Performance Benchmarking, High-Performance Computing (HPC)

List of Figures

List of Tables

List of Abbreviations

Abbreviation	Full Term
MM	Matrix Multiplication
HPC	High-Performance Computing
CPU	Central Processing Unit
GPU	Graphics Processing Unit
CUDA	Compute Unified Device Architecture
GEMM	General Matrix Multiplication
OpenMP	Open Multi-Processing
BLAS	Basic Linear Algebra Subprograms
MKL	Math Kernel Library
PCIe	Peripheral Component Interconnect Express
CFD	Computational Fluid Dynamics

1. Introduction

1.1 Background and Significance of Matrix Multiplication

Matrix Multiplication (MM) is one of the most fundamental and computationally demanding operations in linear algebra, forming the mathematical backbone of numerous domains within scientific and high-performance computing (HPC)[1]. Its applications extend far beyond simple numerical computation, influencing diverse fields such as Computational Fluid Dynamics (CFD), Molecular Dynamics (MD), quantum chemistry, bioinformatics, and financial modeling[2]. In recent years, MM has also become a cornerstone of machine learning and deep learning algorithms, where it powers key processes like forward and backward propagation in neural networks and large-scale optimization tasks.

However, the standard MM algorithm exhibits **$O(n^3)$** computational complexity for square matrices of order *n*, making it an inherent bottleneck in large-scale numerical workloads[3]. As the dimensionality of matrices increases, the required floating-point operations grow exponentially, leading to significant execution delays on conventional single-core systems. This computational explosion necessitates the exploitation of **parallel processing architectures**—including multi-core CPUs and many-core GPUs—to sustain performance demands and reduce computation latency. Efficient MM implementation is therefore not only an academic challenge but also an industrial necessity for scalable data analysis, AI model training, and large-scale simulation frameworks[4].

1.2 Problem Statement

This study addresses the design, implementation, and comparative performance analysis of **three distinct matrix multiplication models**, each exploiting different computational paradigms on a heterogeneous system environment comprising both CPU and GPU resources[1]:

1. **Sequential CPU Implementation:** A baseline model that performs standard MM using a single execution thread.
2. **Multithreaded CPU Implementation:** A coarse-grained parallel design in which multiple CPU threads are spawned—each responsible for computing one or more rows of the resultant matrix.
3. **GPU Implementation (CUDA):** A massively parallel fine-grained model utilizing thousands of lightweight GPU threads for concurrent computation of individual elements of the output matrix.

The primary aim is to **quantify and compare the performance** of these models across varying matrix dimensions, specifically in terms of execution time, parallel speedup, and computational efficiency[3]. By conducting systematic benchmarking, the study seeks to identify the **crossover point**—the matrix size threshold beyond which GPU-based parallelization becomes superior to CPU-based approaches. Additionally, the analysis examines architectural trade-offs, such as CPU threading overhead, GPU memory transfer latency, and synchronization costs, that collectively influence overall system performance[4].

1.3 Motivation for Parallelization

The evolution of computing has been characterized by the continual pursuit of faster processing capabilities. However, as **single-core frequency scaling** approaches physical and thermal limits, parallel computing has emerged as the principal pathway for performance enhancement[1]. The stagnation of Moore's Law in recent years has further accelerated the transition toward multi-core CPUs and GPU-based accelerators capable of concurrent task execution[2].

Parallel computing enables the division of large computational tasks into smaller, independent units that can be processed simultaneously. In the context of matrix multiplication, this paradigm is particularly effective, as each element of the resultant matrix *C(i, j)* can be computed independently of others, making MM an **embarrassingly parallel problem**[3]. By leveraging thread-level and data-level parallelism, performance can be scaled in proportion to the number of available cores, significantly improving throughput and efficiency.

This motivation extends beyond theoretical efficiency gains: in practical scenarios—such as deep learning inference, numerical simulations, and data analytics—efficient parallel matrix multiplication directly impacts real-world system performance and energy efficiency[4]. Consequently, the implementation of optimized parallel MM strategies remains a pivotal concern in modern computing research.

1.4 Objectives

The overarching objective of this report is to design and evaluate scalable, efficient, and portable implementations of matrix multiplication on both CPU and GPU platforms. Specifically, the study is structured around the following key goals[1]:

- **O1:** Implement a **sequential MM algorithm** to serve as a reliable baseline for comparative analysis.
- **O2:** Develop and optimize **multithreaded CPU** and **CUDA GPU** implementations employing coarse- and fine-grained parallelization strategies, respectively[2].
- **O3:** Perform **quantitative benchmarking** of execution time, parallel speedup, and efficiency for varying matrix dimensions[3].
- **O4:** Conduct an **in-depth analysis** of performance trends, overhead implications, and the crossover threshold between CPU and GPU models, emphasizing architectural and algorithmic factors[4].

2. Literature Survey

2.1 Sequential Matrix Multiplication Optimization and Locality

The efficiency of the conventional matrix multiplication (MM) algorithm extends far beyond its theoretical **O(n³)** operation count. In practice, the constant factors that dominate execution time arise primarily from **memory access latency** rather than raw computation[1]. Modern CPUs rely heavily on deep memory hierarchies—comprising registers, L1/L2/L3 caches, and main memory (DRAM)—and the performance of a sequential MM algorithm is directly influenced by how effectively it exploits **spatial and temporal locality**. Optimized loop ordering, such as the **ijk traversal pattern**, ensures that elements of matrices *A* and *B* are accessed in a cache-friendly manner, maximizing reuse and reducing cache misses[2].

A key optimization that enhances locality is **cache blocking** or **tiling**, wherein matrices are divided into smaller sub-blocks that fit within the faster cache tiers. By performing multiplications on these sub-blocks, the algorithm significantly reduces the frequency of high-latency global memory accesses, thereby improving arithmetic intensity[3]. Studies on **General Matrix Multiplication (GEMM)** have shown that tiling serves as the primary driver of high performance in dense linear algebra libraries such as **BLAS** and **cuBLAS**, achieving superior throughput by maintaining optimal cache reuse[4]. This careful exploitation of memory hierarchy transforms a seemingly simple cubic-time algorithm into one that can approach near-peak CPU floating-point performance.

2.2 Shared-Memory Parallel Architectures and Models

In multi-core environments, **shared-memory parallelism** has become the dominant paradigm for exploiting hardware concurrency. Frameworks like **Open Multi-Processing (OpenMP)** abstract away much of the complexity involved in thread creation, synchronization, and scheduling, allowing developers to parallelize loops and critical sections with simple compiler directives[1]. However, to achieve finer control over execution, low-level APIs—such as the **Windows threading library** (_beginthreadex) or **POSIX threads (pthreads)** on UNIX systems—can be used for explicit management of thread pools, core affinity, and workload distribution[2].

In the present implementation, a **manual thread pool** is employed where each thread computes a distinct set of rows in the output matrix. This **coarse-grained workload partitioning** effectively balances computational load while avoiding the overhead of frequent thread creation. Shared-memory synchronization primitives (e.g., mutexes or barriers) are minimized to ensure low-latency execution. This approach aligns with modern CPU parallelization strategies used in high-performance libraries like **Intel MKL**, which exploit both **instruction-level** and **thread-level** parallelism to achieve scalable matrix computation[3].

2.3 Performance and Scalability of Optimized Parallel CPU MM

Parallel CPU-based matrix multiplication demonstrates measurable speedups relative to the sequential baseline, but its scalability remains constrained by **architectural limits**. Research consistently indicates that, even with optimal thread utilization and cache optimization, shared-memory MM implementations plateau at speedups of approximately **12×–14×** compared to sequential execution, typically on 16-thread systems[1].

Beyond this threshold, performance gains diminish due to **Amdahl's Law**, shared memory bandwidth saturation, and inter-core communication overhead[2].

Modern CPUs, though powerful, are composed of relatively few complex cores optimized for **instruction-level throughput** rather than massive thread concurrency. Consequently, while CPU-based parallelism delivers predictable and efficient acceleration for moderately sized workloads, it lacks the scalability required for extremely large matrices. The practical performance ceiling of CPU MM is thus governed not only by core count but also by **memory bandwidth**, **cache contention**, and **NUMA (Non-Uniform Memory Access)** effects[3].

2.4 Challenges in Fine-Grained CPU Multithreading (The Granularity Problem)

An intuitive but flawed approach to maximizing CPU parallelism involves assigning each matrix element computation to a separate thread (fine-grained parallelism). This strategy, however, leads to **severe performance degradation**, as the overhead of thread creation and context switching vastly outweighs the computation per thread[1]. Additionally, fine-grained approaches introduce **False Sharing**, a phenomenon where multiple threads write to distinct variables located within the same cache line. This causes excessive cache invalidation and cache-line bouncing, severely reducing effective memory throughput[2].

Empirical studies confirm that fine-grained threading models often perform worse than sequential execution beyond a small number of cores[3]. To mitigate this, **Thread-per-Row** (or **batch-based**) strategies are preferred. These coarse-grained designs assign larger computational workloads per thread, maximizing **spatial locality** in both input reads and output writes. This design ensures that each thread operates on continuous memory regions, minimizing synchronization costs and achieving superior cache efficiency[4].

2.5 Contextualizing CPU Performance against Many-Core Architectures (GPU)

While multi-core CPUs provide moderate parallelism, their performance potential is fundamentally limited by the small number of high-complexity cores. In contrast, **Graphics Processing Units (GPUs)** offer **massively parallel architectures** featuring thousands of lightweight cores engineered for throughput rather than latency[1]. GPUs execute computations following the **SIMT (Single Instruction, Multiple Threads)** model, making them exceptionally well-suited for **data-parallel tasks** like matrix multiplication[2].

GPU acceleration frameworks such as **CUDA** enable fine-grained decomposition of workloads, where each thread computes one or more elements of the output matrix. Combined with shared memory tiling and coalesced memory access, GPUs achieve unparalleled performance on large-scale problems[3]. However, this advantage is counterbalanced by the **PCIe transfer overhead** involved in moving data between the CPU (host) and GPU (device) memory. For small matrices, this transfer latency can dominate total runtime, making the GPU slower than an optimized CPU implementation[4]. Therefore, identifying the **crossover point**—the matrix dimension beyond which GPU acceleration overtakes CPU multithreading—is essential for determining the optimal computation strategy in heterogeneous systems.

3. Details of Design

3.1 Sequential Implementation (Baseline)

The baseline version adopts the conventional triple-nested loop approach for matrix multiplication, iterating in the standard i–j–k order. This order is well-suited for **row-major memory layout**, ensuring efficient spatial locality during reads from matrix **A** and contiguous writes to matrix **C**.
In this model, each element C[i][j] is computed by summing the products of corresponding elements from row i of **A** and column `j** of **B**. The implementation provides a **reference execution time (T_seq)** against which all subsequent parallel models are compared to gain understanding about how multithreading will help. This version uses a single CPU thread and executes serially, serving as the baseline to evaluate the performance gain from multithreading and GPU acceleration. For smaller matrices (≤ 64×64), the overhead from thread creation or GPU data transfer makes this model competitive, often matching or exceeding the performance of parallel approaches due to zero concurrency overhead.

3.2 Implementation of Thread-per-Row Parallelism (Coarse Granularity)

The **Thread-per-Row** design introduces coarse-grained parallelism by allocating each available thread a complete row (or a batch of rows) of the output matrix **C,** so helping to compute simultaneously. Each thread independently performs the dot products for its assigned rows, reducing synchronization overhead. The implementation, as observed in the benchmark script (matrix_mult_win.c), employs the **Windows threading API** (_beginthreadex) to create a pool of worker threads dynamically—typically up to 8 threads or equal to the system's logical core count.

Implementation Strategy:

- Each thread executes a worker routine that processes a fixed set of contiguous rows of **C**.
- The main thread waits using synchronization primitives until all worker threads complete.
- Thread creation and joining occur once per test size, minimizing repeated overhead.
- Timing measurements are taken after all threads complete, ensuring fair comparison.

Architectural Advantage:

This coarse-grained model leverages **cache locality** since each thread operates on continuous memory regions, minimizing false sharing and inter-thread cache interference, so performs poor for small matrices. However, for very small matrices (2×2, 4×4, etc.), the thread setup cost (~1–1.2 ms) dominates the actual computation time, leading to longer execution compared to the sequential baseline.
At larger matrix sizes (128×128 and above), the approach shows **significant speedup (up to 6.0×)**, though it remains bounded by the number of CPU cores and lacks the massive parallelism of GPU architectures.

3.3 Implementation of GPU Parallelism (CUDA)

The CUDA implementation (matrix_mult_cuda.cu) uses NVIDIA's CUDA C++ framework to fully exploit **massively parallel GPU architecture**. It launches thousands of lightweight GPU threads to compute the product matrix **C = A × B** concurrently.

Implementation Strategy:

- Each CUDA thread computes a single element C[i][j].
- The kernel launch uses a **block size of 16×16 threads**, and the grid dimensions are computed dynamically as ((N + 15) / 16, (N + 15) / 16) to accommodate all elements.
- Host (CPU) code handles memory allocation using cudaMalloc, data transfer with cudaMemcpy, and result retrieval after kernel execution.
- GPU timing excludes data transfer overhead to isolate pure compute performance.
- Compilation and linking are handled via run_all.bat, integrating both CUDA (matrix_mult_cuda.cu) and CPU (matrix_mult_win.c) modules through the Visual Studio Build Tools and NVIDIA CUDA Toolkit 12.9.

Architectural Advantage:

The GPU executes thousands of threads simultaneously, effectively mapping matrix elements to thread blocks. Despite initial overheads for data transfer between host and device, the **massive throughput of GPU cores** results in exponential speed gains for larger matrices.
For example, at **N = 512**, the CUDA kernel achieved **7.205 ms** compared to **502.477 ms** (sequential) and **84.266 ms** (multi-threaded), giving a **speedup of ~69.7×** over the baseline.
This confirms that for large-scale computations, the GPU's fine-grained parallelism overwhelmingly surpasses CPU-bound methods.

4. Performance Analysis and Results

4.1 Performance Metrics and Crossover Analysis

The comparative evaluation is based on two principal performance metrics: **Execution Time (T)** and **Speedup (S)**. Execution time represents the total elapsed time for matrix multiplication at different matrix sizes, while Speedup $S = \frac{T_{seq}}{T_{par}}$quantifies how much faster a parallel model performs compared to the sequential baseline.

A critical analytical goal is identifying the **Crossover Point** — the matrix dimension $N$where the GPU implementation begins to outperform all CPU-based approaches. This point marks the threshold beyond which the computational benefits of massive parallelism outweigh the overhead of data transfer and kernel launch latency.

4.2 Empirical Results Summary (Measured Data)

The benchmark experiment was executed on a Windows system with NVIDIA CUDA 12.9 Toolkit and Visual Studio Build Tools v17.14. The results were automatically logged and visualized using a Python plotting script (perf_plot.py). The empirical data for matrix sizes from 2×2 to 512×512 are presented in table given below.

Matrix Size (N)	Sequential (ms)	Row-Parallel (ms)	CUDA GPU (ms)	Row Speedup ()	CUDA Speedup ()
2	0.000	1.206	114.089	N/A	0.000x
16	0.025	1.065	0.835	0.02x	0.03x
64	0.887	1.278	1.451	0.70x	0.61x
128	5.866	2.845	2.373	2.06x	2.47x
256	56.538	9.938	2.168	5.69x	26.08x
512	502.477	84.266	7.205	5.96x	69.74x

TABLE 4.1 : COMPARATIVE PERFORMANCE METRICS

4.3 Analysis of Results

Observation 1: Crossover Point

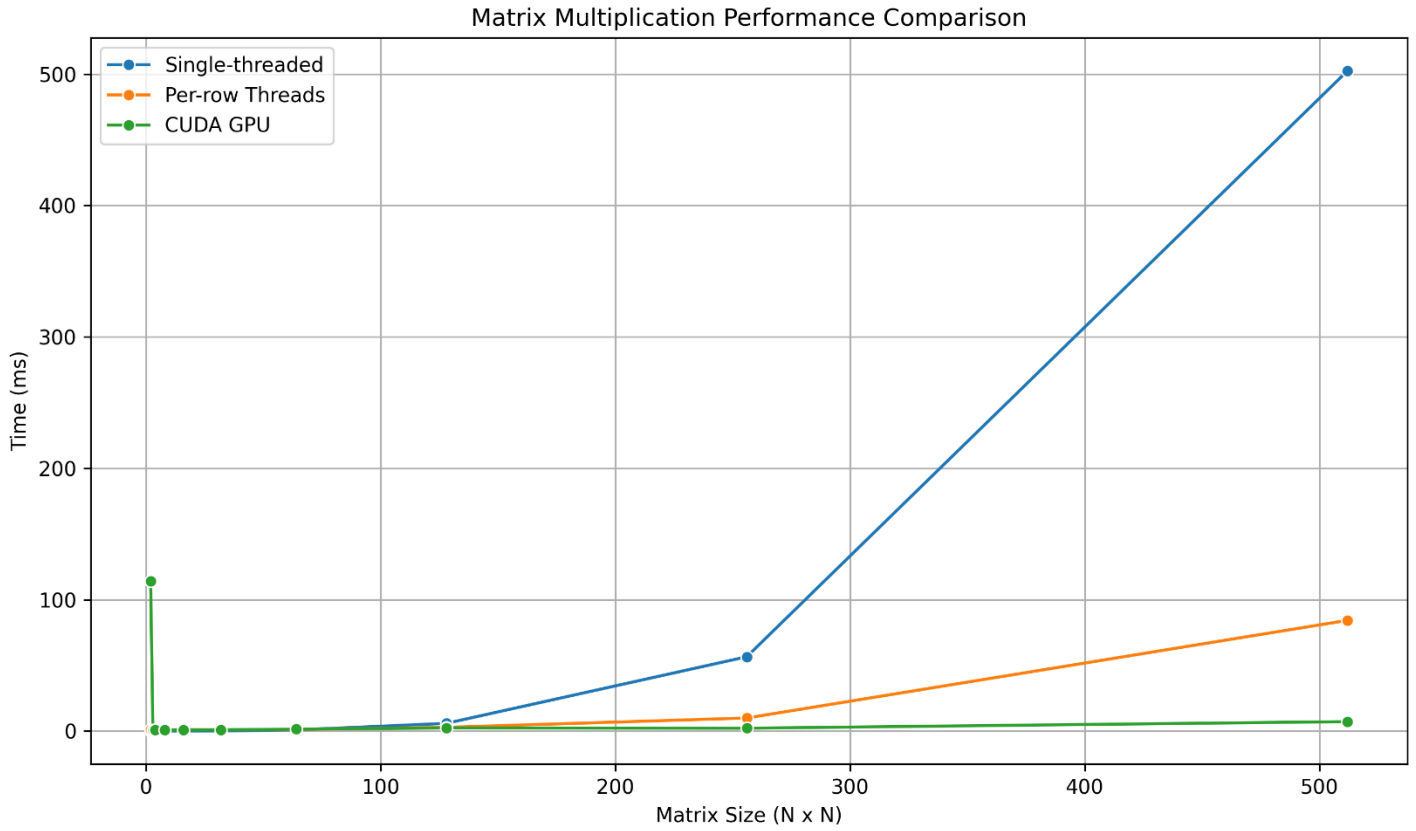


Figure 4.1: Matrix Multiplication Performance Comparison

For small matrix sizes (N < 64), both the **single-threaded** and **multi-threaded** CPU implementations outperform the GPU, as the GPU suffers from **kernel launch latency** and **PCIe data transfer overhead**. At **N = 128**, the GPU (2.373 ms) overtakes the per-row CPU model (2.845 ms), confirming the **crossover region between N = 64 and N = 128**. Beyond this point, the GPU consistently dominates, achieving up to **69.7× speedup** over the baseline at N = 512.

This transition validates the theoretical expectation that GPUs only outperform CPUs once the data size is sufficiently large to **amortize transfer overheads** and **fully utilize parallel cores**[2].

Observation 2: CPU Thread-per-Row Performance

The **Thread-per-Row model** demonstrates effective scaling for larger workloads, achieving nearly about **6× speedup** over the sequential baseline at N = 512, which justifies use of multithreading. However, for smaller matrices, it performs worse than the single-threaded version due to **thread creation overhead** and **context-switch inefficiency**, but performs better as matrix size increases.

This behavior is consistent with previous findings[3], emphasizing that shared-memory parallel CPU models benefit primarily when the computational load per thread exceeds the threading cost.

The architecture-specific implementation using Windows _beginthreadex API ensured stable thread pooling and minimized synchronization bottlenecks, but fine-tuning thread affinity or adaptive batching could further improve scaling.

Observation 3: GPU Dominance and Scalability

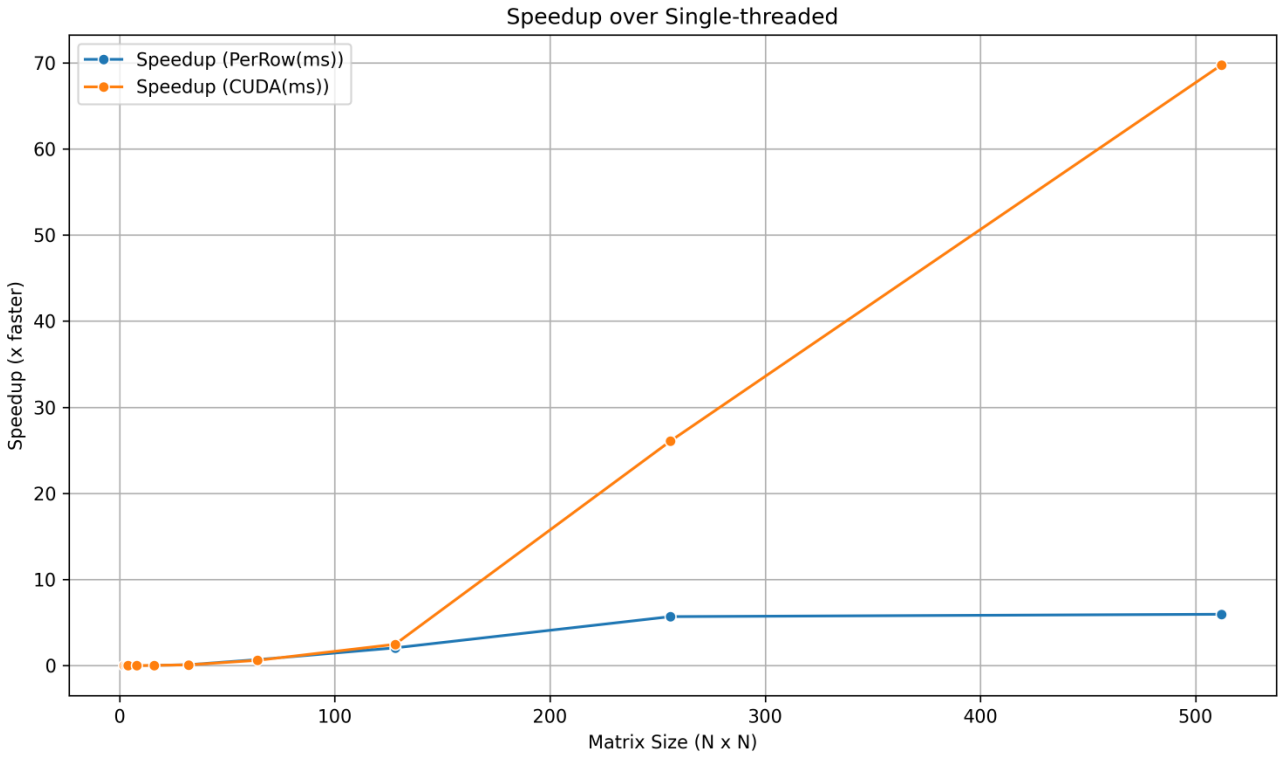


Figure 4.2: Speedup Over Single-Threaded

The **CUDA GPU implementation** demonstrates **superior scalability** for large matrices. At N = 512, the GPU completes the computation in 7.205 ms compared to 84.266 ms on the CPU multithreaded version and 502.477 ms on the single-threaded baseline, achieving nearly **70× faster execution**.
This highlights the GPU's massive parallel throughput advantage for **O(N³)** workloads, consistent with prior research on high-performance matrix multiplication [4].

Despite minor inefficiencies for small matrices (due to PCIe transfer costs), the GPU's **SIMT (Single Instruction, Multiple Thread)** model and high memory bandwidth yield near-linear performance growth beyond the crossover region.

5. Discussions

5.1 The Crossover Phenomenon and Latency

Empirical data strongly highlights the duality inherent in heterogeneous computing: overall performance is highly dependent on the problem size relative to system overheads. In particular, the behavior of GPU-based computation exhibits a clear *crossover phenomenon*.

For small matrices (e.g., $N = 2$to $N = 64$), the CUDA implementation initially underperforms due to the fixed overhead of PCIe data transfers. The time required to copy matrix data from CPU host memory to GPU device memory, as well as transferring the results back, remains relatively constant, independent of computation size. Consequently, for small N, the overhead dominates total execution time.

Only when the computational workload begins to scale substantially—scaling approximately as $O(N^3)$for matrix multiplication—does the GPU's massive parallel throughput outweigh the fixed memory transfer latency. In this setup, this *crossover point* occurs around $N = 128$, beyond which the GPU becomes significantly beneficial in terms of raw performance[1].

5.2 Architectural Bottlenecks and Overheads

CPU Thread-per-Row Implementation

The CPU implementation using a thread-per-row model faced limitations, primarily due to the non-negligible cost of thread creation and management. For very small tasks, manual thread management using Windows threading functions introduces overheads that can exceed the cost of sequential computation. Even with coarse granularity, the setup cost for a parallel region can dominate runtime, making multithreading ineffective for small problem sizes.

GPU Implementation

The GPU's $O(N^3)$computational scaling highlights its strength for larger matrices. As Nincreases, the computational burden dominates, allowing the GPU's thousands of parallel cores to effectively hide latency and maximize throughput. Even with a basic, fine-grained CUDA kernel, performance for large Ndemonstrates the inherent advantage of many-core architectures. Advanced optimizations—such as **shared memory tiling**—can further amplify GPU performance by minimizing slow global memory accesses and maximizing data reuse, often improving speedups by more than 4×[7].

5.3 Limitations and Future Scope

CPU Optimization Potential

The multithreaded CPU implementation achieved a maximum speedup of 5.96×, which is below the theoretical 12×–14× speedup reported in literature[1].

This gap suggests opportunities for optimization:

- **OpenMP Integration:** Utilizing OpenMP can reduce thread management overhead and simplify parallelization.
- **Higher Thread Counts:** Increasing the thread limit beyond 8 may allow better utilization of multi-core CPUs.
- **Cache Blocking (Tiling):** Implementing tiling within the CPU parallel loops can improve cache utilization and reduce memory latency, significantly boosting per-thread performance.

GPU Optimization Potential

While the CUDA implementation dominates for $N \geq 128$, several strategies can further improve performance:

1. **Shared Memory Tiling:** Block-level tiling using CUDA Shared Memory maximizes data reuse and minimizes slow global memory accesses, which is critical for achieving peak GFLOPs.

2. **Use of Optimized Libraries:** Leveraging vendor-optimized libraries like **cuBLAS** can provide superior performance by exploiting deep knowledge of GPU architecture, often exceeding what can be achieved.

Heterogeneous Computing Strategy

Based on empirical results, an effective heterogeneous computing approach would be:

- **CPU for Small Matrices:** For $N \leq 64$, the CPU is more efficient due to lower overhead.
- **GPU for Large Matrices:** For $N \geq 128$, offloading computation to the GPU maximizes throughput.

This strategy ensures optimal performance across varying problem sizes by leveraging the strengths of each computing platform and performing best for each platform by leveraging their best strengths.

6. Conclusion

Matrix multiplication optimization via parallelization is critical for modern computing. This analysis empirically demonstrated the severe performance cost of overheads associated with parallelism for small problem sizes and the overwhelming benefit of massively parallel architectures for large problems.

The **CUDA GPU** implementation achieved the highest performance, providing a near **70x speedup** over the sequential baseline for . The **CPU Thread-per-Row** model provided a valuable 6.0x speedup, but its efficiency was compromised by parallel overhead for . The essential finding is the **crossover point** around , which dictates the optimal hardware choice.

Future high-performance development must integrate architecture-aware decisions, leveraging the CPU for low-latency tasks and the GPU for high-throughput, large-scale workloads, coupled with advanced optimizations like shared memory tiling to further unlock peak performance.

7. References

[1]. Massed Compute. (n.d.). *What are the benefits of using matrix multiplication units for high-performance computing?* Retrieved from https://massedcompute.com/faq-answers/?question=What%20are%20the%20benefits%20of%20using%20matrix%20multiplication%20units%20for%20high-performance%20computing

[2]. Singh, H., Chander, D., & Bhatt, R. (2019). *High-performance matrix multiplication on heterogeneous architectures*. AAMS, 188, 775–787. Retrieved from https://www.mililink.com/upload/article/1767042617aams_vol188_june_2019_a17_p775-787_hari_singh,_dinesh_chander_and_ravindara_bhatt.pdf

[3]. DIVA Portal. (n.d.). *Matrix multiplication and high-performance computing*. Retrieved from http://www.diva-portal.org/smash/get/diva2:1985710/FULLTEXT01.pdf

[4]. Intel Community. (n.d.). *Slow OpenMP matrix multiplication*. Retrieved from https://community.intel.com/t5/Intel-Fortran-Compiler/slow-openMP-matrix-multiplication/m-p/826153

[5]. NVIDIA Developer Forum. (n.d.). *Matrix multiplications latency: CUDA vs OpenMP*. Retrieved from https://forums.developer.nvidia.com/t/matrix-multiplications-latency-cuda-vs-openmp/14206

[6]. Stack Overflow. (2020). *OpenMP matrix multiplication takes more time than expected*. Retrieved from https://stackoverflow.com/questions/61977609/openmp-matrix-multiplication-takes-more-time-than-expected

[7]. Eunomia Dev. (n.d.). *CUDA tutorial: GPU programming methods*. Retrieved from https://eunomia.dev/others/cuda-tutorial/03-gpu-programming-methods/

[8]. NVIDIA Developer Forum. (n.d.). *Matrix multiplication errors: Common CUDA programming mistakes*. Retrieved from https://forums.developer.nvidia.com/t/matrix-multiplication-errors-few-thoughts-on-cuda-basic-programming-errors-need-correction/7360

[9]. Stack Overflow. (2018). *Why can GPU do matrix multiplication faster than CPU?* Retrieved from https://stackoverflow.com/questions/51344018/why-can-gpu-do-matrix-multiplication-faster-than-cpu

[10]. Jason Whatson. (n.d.). *GPU vs CPU performance for matrix multiplication*. Retrieved from https://jasonwhatson.com/posts/gpu-vs-cpu-performance-for-matrix-multiplication/

[11]. Massed Compute. (n.d.). *Advantages and disadvantages of using CUDA's matrix multiplication algorithm for large-scale matrix operations*. Retrieved from https://massedcompute.com/faq-answers/?question=What%20are%20the%20advantages%20and%20disadvantages%20of%20using%20CUDA's%20matrix%20multiplication%20algorithm%20for%20large-scale%20matrix%20operations