Comparative Study for Vector Addition and Matrix Multiplication in SIMD vs CUDA

Kaylyn King
University of North Texas
Department of Computer Science
Dr. Shu
Computer System Architecture
kaylynking@my.unt.edu

Eric Easley
University of North Texas
Department of Computer Science
Dr. Shu
Computer System Architecture
ericeasley@my.unt.edu

ABSTRACT

This project investigates the performance differences between SIMD and CUDA implementations for data-parallel operations. Using vector addition and matrix multiplication as benchmark tasks, experiments were conducted with SIMD implementations for CPU parallelization and CUDA kernels executed on Google Colab's GPU infrastructure. The results shows that SIMD achieved consistent speedups over scalar execution for smaller input sizes while CUDA demonstrated superior scalability and higher performance as workload size and computational complexity increased. Limitations related to hardware access and optimization levels were noted, suggesting that further tuning could enhance performance results. These findings offer practical insights for selecting appropriate parallelization strategies based on workload characteristics and available computational resources.

1. INTRODUCTION

As paper documents are becoming increasingly digital, there is a growing demand for processing large amounts of data which has introduced the idea of machine learning and image processing. Because of this, there has been a significant increase in the concept of data parallelism, where the same operation is performed concurrently across multiple data elements. This concept plays a critical role in accelerating the computational workloads demanded by today's society.

Two major architectures have emerged to facilitate data parallelism: CPUs equipped with Single Instruction, Multiple Data (SIMD) extensions, and Graphics Processing Units (GPUs) programmed through platforms like Compute Unified Device Architecture (CUDA). Although both approaches have the same goal of accelerating execution, they differ in hardware design and programming models which ultimately differentiates them in terms of operational efficiency.

This paper focuses on a comparative study of SIMD and CUDA programming in the context of data parallelism. Specifically, we implement vector addition and matrix multiplication algorithms to evaluate the execution performance, memory utilization, and scalability of SIMD processing on a CPU and CUDA programming on a GPU.

In this study, we implement and evaluate vector addition and matrix multiplication, two widely used parallel algorithms, on both SIMD processing on the CPU and CUDA programming on the GPU. We benchmark both implementations across a range of input sizes, from 2^10 to 2^20 for vector addition and from 2^6 to 2^10 dimensions for matrix multiplication, and measure execution time and speedup ratios. Additionally, we analyze trade-offs in performance based on data size, memory usage, and programming complexity. Our results provide insight into when SIMD or CUDA-based solutions are most effective for data-parallel tasks at varying levels of input.

The remainder of this paper is organized as follows: Section 2 provides background on data parallelism, SIMD programming, and CUDA programming. Section 3 describes the experimental methodology. Section 4 presents the results. Section 5 analyzes the results. Section 6 concludes the paper with final observations.

2. BACKGROUND

Data parallelism has emerged as a fundamental model in high-performance computing, enabling the concurrent execution of identical operations across large datasets. Two major advances in this space are Single Instruction, Multiple Data (SIMD) extensions for CPUs and Compute Unified Device Architecture (CUDA) programming for GPUs. This section provides an overview of data parallelism principles, explores SIMD and CUDA programming models, and sets the foundation for the experimental evaluations conducted in this study.

2.1 Data Parallelism

Data parallelism is a computing model where the same operation is performed concurrently across multiple datasets, making it a crucial part of programming large data sets. Therefore, data parallelism paved the way for modern technology such as machine learning and image processing. At first, data parallelism was achieved through a single massive parallelism machine with thousands of small processors working together under one control unit in a Single Instruction, Multiple Data (SIMD) style, called the Connection Machine [1]. SIMD extensions in CPUs, Graphics Processing Units (GPUs) for general-purpose computing, and distributed computing frameworks are three important technologies developed from this study. However, we will be focusing on the evolution and application of SIMD and GPU programming in context of data parallelism.

2.2 SIMD Programming

Single Instruction, Multiple Data (SIMD) processing is a parallel processing technique where a single instruction is executed on multiple data points simultaneously. Data is grouped into vectors that are stored in specialized registers within the CPU to perform one operation instruction on the entire vector. Using a conventional scalar operation requires more instructions since the same operation is performed on each element. Therefore, SIMD programming reduces the redundancy by combining the elements into one vector. This difference is demonstrated by four add

instructions to be performed on 4 different elements shown in Fig. 2.1 [2].

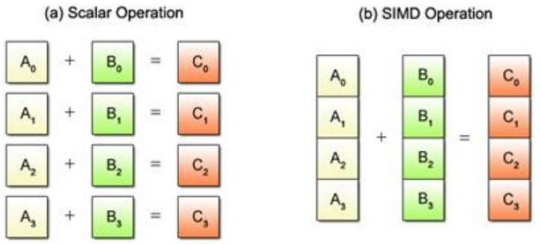

Figure 2.1: Computation of 4 different addition operations performed during a scalar operation (a) and SIMD operation (b) [2]

This performance improvement with minimal additional costs is also supported in a study investigating SIMD-vectorized implementations of scalar product computations [6]. Therefore, SIMD instruction sets have been incorporated into modern CPUs such as Intel's AVX and ARM's NEON to accelerate data-parallel operations [3]. SIMD also has been shown repeatedly to improve the speed of image processing algorithms. SIMD implementations of Fast Fourier Transform algorithms significantly outperformed scalar versions of this algorithm by over 3.3 times the best-performing scalar FFT codes used for image processing [4]. In medical imaging, performance in 3D image reconstruction tasks were able to be substantially improved by using SSE, AVX, and AVX2 SIMD instructions in SIMD-vectorized implementations of the RabbitCT benchmark [5].

In summary, SIMD programming offers substantial performance enhancements for data-parallel tasks by executing a single instruction across multiple data points simultaneously. This approach reduces instruction redundancy and accelerates computations, particularly in applications like image processing and signal analysis. Building upon the principles of SIMD, GPUs have been developed to handle even more extensive parallel workloads. The next section will delve into CUDA programming, exploring how it leverages GPU architectures to further advance data-parallel processing.

2.3 CUDA Programming

Typically, a CPU has between 2 and 64 cores to handle main processing functions of a computer while NVIDIA GPUs can have upwards of thousands of cores to handle graphic and video rendering. Figure 2.2 shows the difference between CPU and GPU architecture. This difference allows GPUs to provide a much higher instruction throughput and memory bandwidth when compared to CPUs [7]. Therefore, programming on the GPU can significantly accelerate computationally intensive applications.

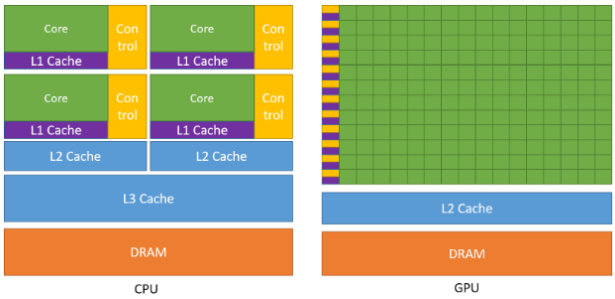

Figure 2.2: Architecture of CPU (left) and GPU (right) [7]

Compute Unified Device Architecture (CUDA) is a parallel computing platform and programming model developed by NVIDIA. CUDA allows programmers to perform general-purpose computing on NVIDIA's GPUs by extending standard programming languages such as C, C++, and Fortran with minimal syntax additions. A CUDA function or kernel is executed on the GPU by multiple threads in parallel with a thread being the smallest unit of execution. The threads are grouped into blocks where the threads within the same block can communicate and synchronize with each other under a shared memory. All blocks can access device-wide memory within the kernel. Figure 2.3 compares this software layout with the hardware layout of the GPU.

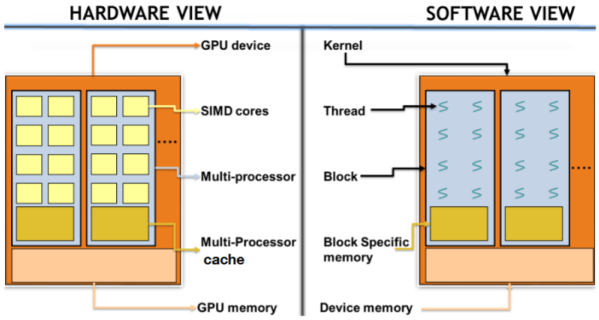

Figure 2.3: Hardware and software layout of a NVIDIA GPU [8]

CUDA offers a multi-tiered memory hierarchy to optimize data access patterns as shown in Figure 2.3. The global memory is accessible by all threads, shared memory is shared among the threads within the same block, and local memory is private to each thread. Due to the size of each memory sector, the global memory has higher latency compared to the shared memory having lower latency and higher bandwidth. Because of this, efficient use of threads and memory hierarchy is essential for high-performance CUDA programming.

The development of CUDA programming is improving modern technology. A study has shown that CUDA accelerated training and inference times for deep learning models for Python frameworks like PyTorch and TensorFlow [9]. CUDA has also been reported for significant performance improvements when utilizing CUDA-enabled GPUs compared to CPU-only implementations for image processing tasks [10]. Both machine learning and image processing involve large amounts of data, underscoring the effectiveness of CUDA programming in

accelerating repetitive, data-intensive computational tasks through data parallelism.

With the fundamentals of data parallelism, SIMD processing, and CUDA programming established, this study aims to evaluate their performance on common parallel computing tasks: vector addition and matrix multiplication. The following section outlines the experimental methodology used to assess each approach, including algorithm design, implementation details, hardware specifications, and performance metrics.

3. METHODOLOGY

This project evaluates the performance of CPU SIMD and GPU programming through vector addition and matrix multiplication.

3.1 Hardware and Software

The only equipment we readily had access to were our laptops. Therefore, the SIMD experiments were conducted on a machine equipped with a 12th Gen Intel Core i7-1255U CPU on Visual Studio Build Tools 2022 software. The CUDA experiments were conducted on Google Colab with Python 3.10 for CuPy CUDA and a NIVIDIA Tesla T4 GPU.

3.2 SIMD Programming

The CPU-based SIMD experiments used the built-in AVX2 function to perform vector addition and matrix multiplication.

3.2.1 Vector Addition

In the vector addition task, two vectors were processed by loading eight floating-point elements at a time into AVX2 256-bit registers using the _mm256_loadu_ps function while performing parallel addition with _mm256_add_ps and storing the results back using _mm256_storeu_ps.

3.2.2 Matrix Multiplication

In the matrix multiplication task, a similar SIMD approach was used. Elements from the input A, B matrices were loaded into AVX2 registers in blocks of eight using the _mm256_loadu_ps function. The corresponding elements from matrix A's row and matrix B's column were multiplied using _m256_mul_ps, and the partial sums were accumulated using _mm256_add_ps to compute the entries of the resulting C matrix.

Timing measurements for SIMD executions were performed using std::chrono::high_resolution_clock to obtain high-precision execution times on the CPU.

3.3 CUDA Programming

For the CUDA experiments, the CuPy library in Python was utilized to interact with NVIDIA's CUDA architecture without manually writing CUDA C kernels.

3.3.1 Vector Addition

Vector addition was implemented using the cp.add() function in CuPy which performs element-wise addition across two vectors. The input array were initialized on the CPU using NumPy and then transferred to the GPU memory space. Timing measurements were taken for both the scalar and GPU vector addition. GPU timings were recorded with synchronization using cp.cuda.Device(0).synchronize() to ensure accurate measurements. The maximum absolute error between the scalar and GPU results was verified to be within 10^-5 across all tested input sizes.

3.3.2 Matrix Multiplication

Matrix multiplication was performed using cp.matmul() for the GPU and np.matmul() for the scalar operations on the CPU. Similar to the vector addition implementations, the matrices were initialized in the CPU memory and then transferred to the GPU for computation. The execution times for both scalar and GPU matrix multiplications were measured separately. Proper synchronization was enforced before and after the GPU operations to prevent inaccurate timing caused by asynchronous kernel launches. Result validity was achieved by comparing the maximum absolute error between the scalar and GPU outputs, which remained within the acceptable floating-point tolerance.

3.4 Input Sizes

For vector addition, the input size N ranged from 2^10 to 2^20 elements. For matrix multiplication, square matrices of sizes 64x64, 128x128, 256x256. 512x512, and 1024x1024 were used.

Each experiment verified its correctness by comparing the SIMD and CUDA results against serial CPU computations. This ensures the maximum absolute error remained within acceptable floating-point tolerance of 1e-3.

3.5 Source Code

All source codes for the SIMD and CUDA implementations are included in Appendix A.

With the experimental methodology established, we now proceed to present the experimental results. Performance measurements, execution times, speedup comparisons, and accuracy validations are detailed in the following section to evaluate the effectiveness of data-parallel programming on modern CPU and GPU architectures.

4. EXPERIMENTS AND RESULTS

To quantitatively evaluate the performance impact of data-parallel programming, experiments were conducted comparing scalar CPU, SIMD-accelerated CPU, and CUDA-accelerated GPU implementations. Vector addition and matrix multiplication were selected for analysis, representing basic and complex data-parallel operations respectively. Execution times, speedup ratios, and accuracy validations were measured across a range of input sizes. This section presents the experimental setup and results.

4.1 Experimental Setup

SIMD experiments were conducted using AVX2 in C++ on Visual Studio Build Tools 2022 software with a 12th Gen Intel Core i7-1255U CPU. CUDA experiments were implemented using the CuPy library for python on Google Colab with a NVIDIA Tesla T4 GPU.

The input sizes tested were vector lengths ranging from 1024 elements to 1,048,576 elements and square matrices with dimensions ranging from 64x64 to 1024x1024.

Each result was validated by checking the maximum absolute error between SIMD/CUDA results and the corresponding CPU scalar results to ensure numerical accuracy within the floating-point tolerance of 1e-3.

4.2 Vector Addition Results

The execution times for vector addition on a scalar CPU algorithm and SIMD CPU algorithm using AVX2 are shown in Table 1.

Table 1. Execution Times for Scalar and SIMD Vector Addition at Size N

N (Vector Size)	Scalar Time (ms)	SIMD Time (ms)
1024	0.0003	0.00026
4096	0.00141	0.00138
16384	0.00752	0.00734
65536	0.02442	0.02347
262144	0.19024	0.17084
1048576	1.00566	0.87722

Figure 4.1 plots these points to further compare the execution times of the scalar and SIMD implementations for vector addition.

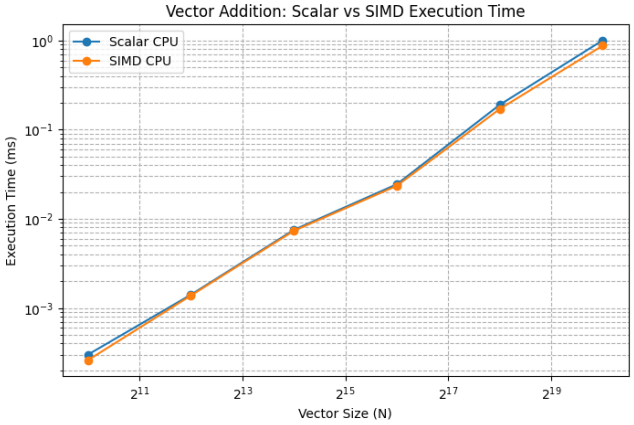

Figure 4.1: Plot of scalar and SIMD vector addition implementation execution times over varying sizes of N

The execution times for vector addition on a scalar CPU algorithm and a CUDA algorithm are shown in Table 2.

Table 2. Execution Times for Scalar and CUDA Vector Addition at Size N

N (Vector Size)	Scalar Time (ms)	CUDA Time (ms)
1024	0.015355	0.115544
4096	0.013104	0.074158
16384	0.014707	0.069715
65536	0.030180	0.056411
262144	0.169231	0.062474
1048576	0.616047	0.102019

Figure 4.2 plots these points to further compare the execution times of the scalar and SIMD implementations for vector addition.

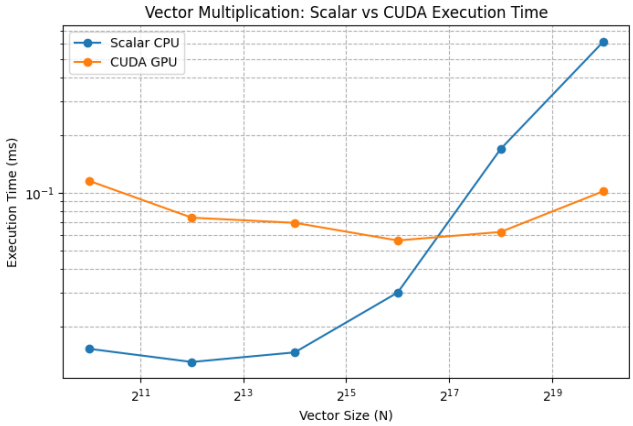

Figure 4.2: Plot of scalar and CUDA vector addition implementation execution times over varying sizes of N

The performance for both SIMD and CUDA implementations is compared to the scalar baseline by dividing the execution time of the scalar time and the SMID or CUDA time to visualize the speedup between each implementation. This is shown in Table 3.

Table 3. Speedup for SIMD and CUDA Vector Addition Compared to Scalar at Size N

N (Vector Size)	Speedup SIMD/Scalar	Speedup CUDA/Scalar
1024	1.15385	0.13289
4096	1.02174	0.17670
16384	1.02452	0.21096
65536	1.04048	0.53500
262144	1.11356	2.70882
1048576	1.14642	6.03855

Figure 4.3 plots these points to further compare the speedups of the SIMD and CUDA implementations when compared to the scalar version for vector addition.

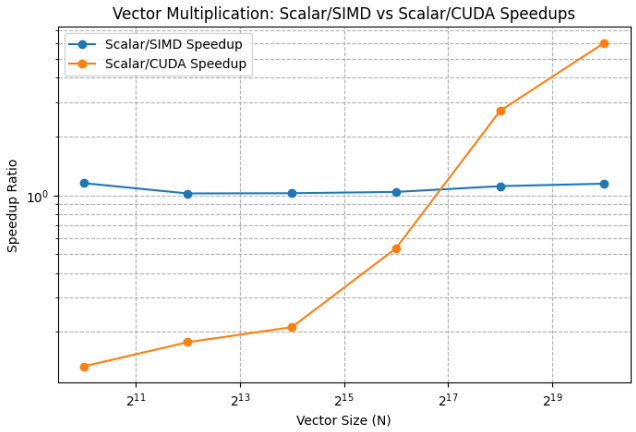

Figure 4.3: Plot of the speedups of SIMD and CUDA vector addition implementations over varying sizes of N

4.3 Matrix Multiplication Results

The execution times for matrix multiplication on a scalar CPU algorithm and SIMD CPU algorithm using AVX2 are shown in Table 4.

Table 4. Execution Times for Scalar and SIMD Matrix Multiplication at Dimension N

N (Matrix Dimension)	Scalar Time (ms)	SIMD Time (ms)
64	0.16536	.67182
128	2.18046	4.2651
256	11.9069	20.5056
512	96.9903	174.914
1024	11819	11752.9

Figure 4.4 plots these points to further compare the execution times of the scalar and SIMD implementations for matrix multiplication.

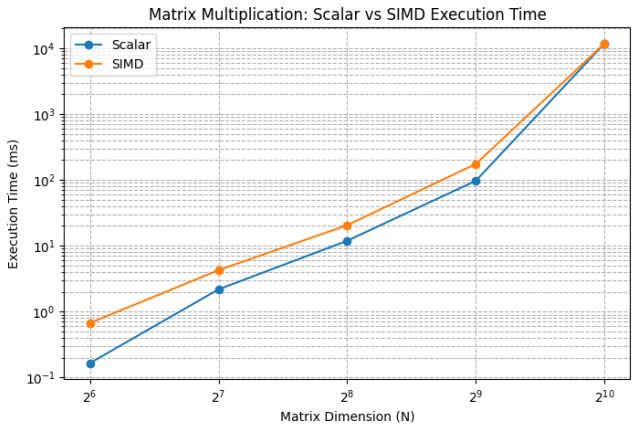

Figure 4.4: Plot of scalar and SIMD matrix multiplication implementation execution times over varying sizes of N

The execution times for matrix multiplication on a scalar CPU algorithm and a CUDA algorithm are shown in Table 5.

Table 5. Execution Times for Scalar and CUDA Matrix Multiplication at Dimension N

N (Matrix Dimension)	Scalar Time (ms)	CUDA Time (ms)
64	0.092	0.504
128	1.153	0.321
256	0.402	0.332
512	2.294	0.392
1024	24.977	1.249

Figure 4.5 plots these points to further compare the execution times of the scalar and CUDA implementations for matrix multiplication.

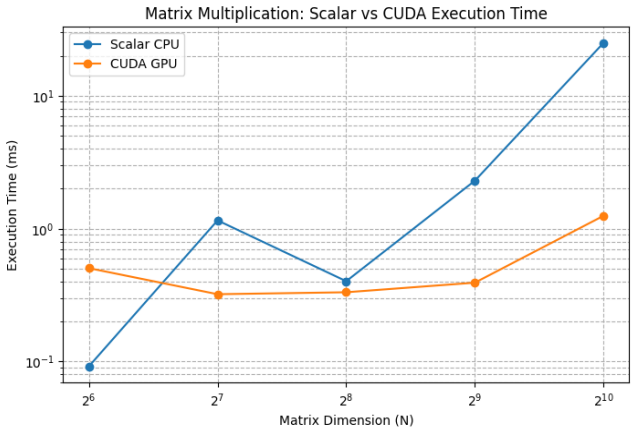

Figure 4.5: Plot of scalar and CUDA matrix multiplication implementation execution times over varying sizes of N

The performance for both SIMD and CUDA implementations is compared to the scalar baseline by dividing the execution time of the SIMD or CUDA time and the scalar time to visualize the speedup between each implementation. This is shown in Table 6.

Table 6. Speedup for SIMD and CUDA Matrix Multiplication Compared to Scalar at Size N

N (Vector Size)	Speedup SIMD/Scalar	Speedup CUDA/Scalar
64	0.2146137	0.18254
128	0.511233	3.59190
256	0.580668	1.21084
512	0.554502	5.85204
1024	1.00562	19.99760

Figure 4.3 plots these points to further compare the speedups of the SIMD and CUDA implementations when compared to the scalar version for matrix multiplication.

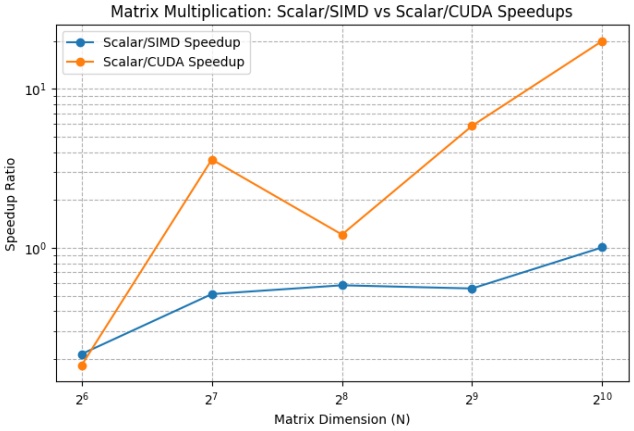

Figure 4.6: Plot of the speedups of SIMD and CUDA matrix multiplication implementations over varying sizes of N

The experimental results presented above highlight significant differences in performance across the scalar, SIMD, and CUDA implementations for both vector addition and matrix multiplication. While raw execution times and speedup ratios provide initial insights into computational efficiency, a deeper analysis is necessary to understand the underlying factors influencing these outcomes. In the next section, we analyze the performance trends, discuss architectural tradeoffs, and explain the observed behavior across varying input sizes and parallel programming models.

5. ANALYSIS

This section provides a detailed analysis of the experimental results obtained from evaluating scalar CPU, SIMD CPU, and CUDA GPU implementations. Performance trends across different input sizes are examined to identify the strengths and limitations of each parallel programming approach. The analysis considers both the magnitude of speedup achieved and the underlying factors influencing performance such as computational overhead, memory access patterns, and hardware architecture. Comparisons are drawn between SIMD and CUDA performance, highlighting the scenarios where each technique is most effective.

5.1 SIMD Performance

The performance for SIMD significantly varied when compared to scalar operations between vector addition and matrix multiplication.

5.1.1 Vector Addition Performance

The speedup for the SIMD implementation of vector addition was consistently greater than one across all tested input sizes, confirming that data parallelism provided tangible performance benefits over the scalar implementation. SIMD processing allowed eight floating-point operations to be executed simultaneously through 256-bit AVX2 vector registers, reducing the number of required instructions and memory access. As the vector size increased, the relative overhead of SIMD initialization became less significant, resulting in a gradual increase in speedup ratios for larger inputs. This demonstrates the effectiveness of SIMD for simple, highly parallelizable tasks as the dataset size grows.

5.1.2 Matrix Multiplication Performance

The speedup for the SIMD implementation of matrix multiplication was consistently less than one across all tested input sizes, indicating the SIMD version performed worse than the scalar baseline. This outcome is largely due to the memory access patterns inherent in matrix multiplication where computation of each output element requires summing products across entire rows and columns. Although element-wise multiplication was vectorized, the irregular memory access to columns of matrix B limited the effectiveness of the SIMD instructions and caused frequent cache misses. Additionally, the overhead of setting up SIMD registers combined with the lack of horizontal summation optimizations further reduced performance gains. As a result, SIMD proved less effective for matrix multiplication compared to simpler operations like vector addition.

In summary, SIMD proved to be highly effective for simple, element-wise operations such as vector addition by consistently outperforming the scalar execution. However, the basic SIMD implementation offered limited benefits due to inefficient memory access patterns and the absence of advanced optimizations. This emphasizes not all computational tasks benefit equally from basic SIMD parallelism without further algorithmic tuning such as cache blocking or loop reordering.

5.2 CUDA Performance

The performance for CUDA between vector addition and matrix multiplication was not as significant as the SIMD performance.

5.2.1 Vector Addition Performance

At smaller input sizes, the CUDA implementation exhibited slower performance compared to the scalar CPU implementation. This initial slowdown is primarily attributed to the overhead associated with launching GPU kernels and transferring data between host and device memory which becomes more significant when the computation workload is relatively small. However, the CUDA implementation began to outperform the scalar CPU implementation substantially as the input size increased specifically at around the input size of 2^17 elements. For large vectors, the parallel execution of thousands of threads and the high memory bandwidth of the GPU enabled significant speedups over the CPU. This highlights CUDA programming becomes highly advantageous for large-scale data-parallel workloads.

5.2.2 Matrix Multiplication Performance

The matrix multiplication experiment for CUDA exhibited a similar performance pattern to that observed in the vector addition experiment. Initially, the CUDA implementation demonstrated slower performance compared to the scalar CPU implementation at smaller input sizes. As with the vector addition experiment, this early disadvantage can be attributed to the fixed overhead of launching GPU kernels and transferring data between the CPU and GPU which dominates when the computational workload is relatively small. However, the CUDA implementation began to outperform the scalar CPU implementation as the input sizes increase. This transition again highlights the CUDA implementation provides substantial performance gains when operating on large-scale data parallel workloads despite initial overheads.

Overall, the CUDA algorithms demonstrated significant performance improvements across both computational tasks with

particularly strong results in matrix multiplication. The inherent design of matrix multiplication of involving large numbers of independent multiplication operations allowed the GPU's massive parallel processing capabilities and high memory bandwidth to be fully utilized at large scales despite the kernel launch overhead initially limiting the performance on small input sizes. As a result, the CUDA implementation delivered increasingly greater speedups as the vector and matrix size grew, outperforming the scalar CPU execution by a wide margin. These results reinforce CUDA programming is especially advantageous for highly parallel, computation-intensive workloads like vector addition and matrix multiplication.

5.3 SIMD vs CUDA Performance

A comparison between the SIMD and CUDA implementations relative to the scalar CPU baseline reveals significant differences in their performance characteristics and scalability across varying input sizes.

5.3.1 Vector Addition

In the vector addition experiments, the speedup ratio for the SIMD implementation remained nearly constant across all input sizes, reflecting the relatively low overhead and consistent efficiency of SIMF operations for simple computations. In contrast, the speedup ratio for the CUDA implementation increased dramatically as the input size grew. This trend highlights the impact of GPU overhead with fixed costs such as kernel launches and memory transfers negatively impacting the results of the smaller input sizes on the GPU. Thus, CUDA excels when the workload is large enough to fully exploit the computational resources of the GPU while SIMD offers consistent gains for small to moderately sized datasets.

5.3.2 Matrix Multiplication

In the matrix multiplication experiments, the performance gap between SIMD and CUDA implementations became even more pronounced. The SIMD implementation struggled to deliver significant speedup compared to the scalar CPU baseline due to inefficient memory access patterns and the complexity of performing multiple dependent operations per output element. In contrast, the CUDA implementation leveraged the inherent parallelism of matrix multiplication to achieve substantial speedup as input sizes increased. Unlike vector addition where both SIMD and CUDA offered consistent improvements at different scales, matrix multiplication revealed the limitations of SIMD and the strength of CUDA for highly parallel and computation-intensive workloads

Overall, the experimental results demonstrate both SIMD and CUDA implementations can significantly improve execution times compared to scalar CPU execution. SIMD consistently provided speedup for simple operations like vector addition but showed limited effectiveness for the more complex task of matrix multiplication. In contrast, CUDA delivered substantial performance gains at larger input sizes for both vector addition and matrix multiplication despite its higher initial overhead. These observations highlight the strengths and tradeoffs of each parallelization approach, setting the foundation for a deeper discussion of limitations and opportunities for further improvement.

5.4 Limitations and Improvements

While the experimental results demonstrated clear performance gains for both SIMD and CUDA implementations, several limitations specific to each approach must be considered. This section discusses the limitations separately for SIMD and CUDA followed by proposed improvements to strengthen future evalutations.

5.4.1 SIMD Limitations

The SIMD implementations did not apply more aggressive techniques such as loop unrolling, vector prefetching, or manual alignment of data structures to further improve performance. Moreover, SIMD instructions were tuned for a specific CPU architecture available during testing. The performance results may not directly transfer to CPUs with different vector unit widths or cache hierarchies, limiting the generality of the findings.

5.4.2 CUDA Limitations

The CUDA experiments were run on Google Colab which provides limited shared access to GPU resources. As a result, the measured performance could be affected by hardware variability, resource contention, and background processes beyond user control. Furthermore, Colab restricts access to advanced GPU profiling and tuning tools, making it difficult to fully optimize or diagnose kernel behavior.

Another limitation was the lack of fine-tuning for CUDA kernel configurations. Thread block size, grid dimensions, and memory access optimizations were kept at default settings. Without these optimizations, the full potential performance of CUDA may not have been realized in the experiments.

5.4.3 Improvements

Future work can address these limitations by running CUDA experiments on dedicated GPU hardware, allowing for consistent and uncontested performance measurements. Access to profiling tools like NVIDIA Nsight Compute would inform deeper kernel optimizations such as tuning thread block sizes and optimizing memory accesses.

For SIMD, improvements could include apply more architecture-specific optimizations like loop unrolling, cache aware data layout, and explicit vector prefetching. Using compiler optimization reports and performance analysis tools could help fine-tune SIMD code to match the hardware capabilities more closely.

Expanding the benchmark set to cover a wider range of workloads would also provide a broader evaluation of both SIMD and CUDA effectiveness across different types of parallel tasks.

In summary, the experiments highlight the significant advantages of both SIMD and CUDA approaches compared to scalar execution, particularly as the problem sizes increased. While SIMD provided consistent improvements through efficient vectorization, CUDA demonstrated greater scalability and performance growth in more complex tasks like matrix multiplication. However, limitations in hardware resources, optimization techniques, and experimental setup introduced constraints that should be addressed in future work. These findings establish a strong foundation for understanding the practical trade-offs between SIMD and CUDA and setup for broader conclusions regarding data-parallel performance strategies.

6. CONCLUSION

This project aimed to evaluate and compare the performance of SIMD and CUDA implementations for data-parallel workloads using vector addition and matrix multiplication as case studies. The results showed that SIMD consistently outperformed scalar execution for smaller data sizes, offering reliable speedups

through efficient vectorization. However, CUDA demonstrated superior scalability and higher overall performance as input sizes increased and computational complexity grew, particularly in matrix multiplication tasks.

The experimental findings were influenced by limitations in hardware access, optimization strategies, and measurement scope, suggesting that the reported performance gains represent a baseline rather than peak achievable performance. Nonetheless, the results provide valuable insights into the practical trade-offs between SIMD and CUDA for different types of parallel workloads.

Understanding these tradeoffs is critical for optimizing application performance across diverse computing platforms. Future work could involve running tests on dedicated hardware, applying more aggressive optimization techniques, and expanding the range of benchmarked applications to further validate and extend these findings.

7. REFERENCES

[1] W. Daniel Hillis and Guy L. Steele Jr. 1986. DATA PARALLEL ALGORITHMS. Retrieved from https://rsim.cs.illinois.edu/arch/qual_papers/systems/3.pdf

[2] Basics of SIMD Programming. Retrieved from https://ftp.cvut.cz/kernel/people/geoff/cell/ps3-linux-docs/CellProgrammingTutorial/BasicsOfSIMDProgramming.html

[3] Hossein Amiri and Asadollah Shahbahrami. 2019. SIMD programming using Intel vector extensions. *Journal of Parallel and Distributed Computing* 135: 83-100. https://doi.org/10.1016/j.jpdc.2019.09.012

[4] Franz Franchetti Jr,, Stefan Kral, Juergen Lorenz, and Christoph W. Ueberhuber. 2003. *Efficient utilization of SIMD extensions*. Retrieved from https://users.ece.cmu.edu/~franzf/paper/ieee-si.pfd

[5] Johannes Hofmann, Jan Eitzinger, Georg Hagar, and Gerhard Wellein. 2014. Comparing the performance of different x86 SIMD instruction sets for a medical imaging application on modern multi- and manycore chips. *Proceedings of the Workshop on Programming Models for SIMD/Vector Processing (WPMVP)*: 57-64. https://dl.acm.org/doi/10.1145/2568058.2568068

[6] Johannes Hofmann, Dietmar Fey, Michael Riedmann, Jan Eitzinger, Georg Hager, and Gerhard Wellein. 2016. Performance analysis of the Kahan-enhanced scalar product on current multi-core and many-core processors. *Concurrency and Computation Practice and Experience* 29, 9. https://doi.org/10.1002/cpe.3921

[7] NVIDIA. CUDA C++ Programming Guide. Retrieved from https://docs.nvidia.com/cuda/cuda-c-programming-guide/#fn1

[8] Dr. Tong Shu. *Basic CUDA Programming.* Computer System Architecture, University of North Texas, 2025

[9] Ming Li, Ziqian Bi, Tianyang, Yizhu Wen, Qian Niu, Junyu Liu, Benji Peng, Sen Zhang, Xuanhe Pan, Jiawei Xu, Jinlang Wang, Keyu Chen, Caitlyn Heqi Yin, Pohsun Feng, and Ming Liu. 2024. Deep Learning and Machine Learning with GPGPU and CUDA: Unlocking the Power of Parallel Computing. *arXiv.org*. Retrieved from https://arxiv.org/abs/2410.05686

[10] Daniel Castano-Diez, Dominik Moser, Andreas Schoenegger, Sabine Pruggnaller, and Achilleas S. Frangakis. 2008. Performance evaluation of image processing algorithms on the GPU. *Journal of Structural Biology* 164, 1: 153-160. https://doi.org/10.1016/j.jsb.2008.07.006

8. APPENDIX A: Source Code Listings

This appendix contains the full source code used for the SIMD and CUDA experiments, including vector addition and matrix multiplication implementations. Each subsection provides the corresponding code for CPU and GPU execution along with basic experimental setup.

8.1 SIMD Implementation

The following code listings implement SIMD-based vector addition and matrix multiplication using AVX functionality, along with their corresponding main() functions for testing and performance measurements.

8.1.1 Vector Addition

This code performs vector addition on CPU using both a scalar loop and AVX2 for comparison. A main() function runs the tests across increasing input sizes, measure execution times, and verifies correctness.

```
// SIMD vector addition using AVX2
#include <iostream>
#include <immintrin.h>
#include <chrono>
#include <cmath>
#include <cstdlib>

// Perform scalar vector addition
void vector_add_scalar(const float* A, const float* B, float* C, int N) {
    for (int i = 0; i < N; ++i)
        C[i] = A[i] + B[i];
}

// Perform SIMD addition with AVX2
void vector_add_avx2(const float* A, const float* B, float* C, int N) {
    int i = 0;
    for (; i <= N - 8; i += 8) {
        // Load 8 floats from each array
        __m256 a = _mm256_loadu_ps(&A[i]);
        __m256 b = _mm256_loadu_ps(&B[i]);

        // Add the vectors
        __m256 c = _mm256_add_ps(a, b);

        // Store the result
        _mm256_storeu_ps(&C[i], c);
    }
    // Handle remaining elements
    for (; i < N; ++i) {
        C[i] = A[i] + B[i];
    }
}

// Run tests across different input sizes and measure performance
void run_test(int N) {
    float* A = (float*)_aligned_malloc(N * sizeof(float), 32);
    float* B = (float*)_aligned_malloc(N * sizeof(float), 32);
    float* C_scalar = (float*)_aligned_malloc(N * sizeof(float), 32);
```

```
    float* C_simd = (float*)_aligned_malloc(N * sizeof(float), 32);

    // Initialize input arrays
    for (int i = 0; i < N; ++i) {
        A[i] = i;
        B[i] = i * 0.5f;
    }

    const int trials = 10;
    double scalar_total = 0;
    double simd_total = 0;

    // Run scalar trials
    for (int t = 0; t < trials; ++t) {
        auto start = std::chrono::high_resolution_clock::now();
        vector_add_scalar(A, B, C_scalar, N);
        auto end = std::chrono::high_resolution_clock::now();
        scalar_total += std::chrono::duration<double, std::milli>(end - start).count();
    }

    // Run SIMD trials
    for (int t = 0; t < trials; ++t) {
        auto start = std::chrono::high_resolution_clock::now();
        vector_add_avx2(A, B, C_simd, N);
        auto end = std::chrono::high_resolution_clock::now();
        simd_total += std::chrono::duration<double, std::milli>(end - start).count();
    }

    // Verify result correctness
    for (int i = 0; i < N; ++i) {
        if (fabs(C_scalar[i] - C_simd[i]) > 1e-5f) {
            std::cerr << "Mismatch at index " << i << ": " << C_scalar[i] << " vs " << C_simd[i] <<
"\n";
            break;
        }
    }

    double scalar_avg = scalar_total / trials;
    double simd_avg = simd_total / trials;

    std::cout << "N = " << N << "\tScalar Avg: " << scalar_avg << " ms" << "\tSIMD Avg: " << simd_avg
<< " ms" << "\tSpeedup: " << (scalar_avg / simd_avg) << "x\n";

    _aligned_free(A);
    _aligned_free(B);
    _aligned_free(C_scalar);
```

```
    _aligned_free(C_simd);
}


int main() {
    std::cout << "Running SIMD tests...\n";
    for (int exp = 10; exp <= 20; exp += 2) {
        int N = 1 << exp;
        run_test(N);
    }
    std::cout << "Done.\n";
    return 0;
}
```

8.1.2 Matrix Multiplication

This code implements matrix multiplication on CPU using both a scalar and an AVX2 approach. The main() function initializes input matrices, execute tests across varying matrix sizes, measures performance, and checks result accuracy.

```
// SIMD matrix multiplication using AVX2
#include <iostream>
#include <chrono>
#include <immintrin.h>
#include <malloc.h>
#include <cmath>

// Scalar matrix multiplication
void matmul_scalar(const float* A, const float* B, float* C, int N) {
    for (int i = 0; i < N; ++i)
        for (int j = 0; j < N; ++j) {
            float sum = 0.0f;
            for (int k = 0; k < N; ++k)
                sum += A[i * N + k] * B[k * N + j];
            C[i * N + j] = sum;
        }
}

// SIMD matrix multiplication using AVX2
void matmul_simd(const float* A, const float* B, float* C, int N) {
    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < N; ++j) {
            __m256 sum_vec = _mm256_setzero_ps();
            int k = 0;
            for (; k <= N - 8; k += 8) {
                // Load 8 elements of row A and gather corresponding elements of column B
                __m256 a_vec = _mm256_loadu_ps(&A[i * N + k]);

                float b_temp[8];
```

```
                for (int t = 0; t < 8; ++t)
                    b_temp[t] = B[(k + t) * N + j];

                __m256 b_vec = _mm256_loadu_ps(b_temp);

                // Multiply and accumulate
                sum_vec = _mm256_fmadd_ps(a_vec, b_vec, sum_vec);
            }

            // Horizontal sum of SIMD register
            float sum_array[8];
            _mm256_storeu_ps(sum_array, sum_vec);
            float sum = 0.0f;
            for (int t = 0; t < 8; ++t)
                sum += sum_array[t];

            // Handle remaining elements
            for (; k < N; ++k)
                sum += A[i * N + k] * B[k * N + j];

            C[i * N + j] = sum;
        }
    }
}

void run_test(int N) {
    float* A = (float*)_aligned_malloc(N * N * sizeof(float), 32);
    float* B = (float*)_aligned_malloc(N * N * sizeof(float), 32);
    float* C_scalar = (float*)_aligned_malloc(N * N * sizeof(float), 32);
    float* C_simd = (float*)_aligned_malloc(N * N * sizeof(float), 32);

    // Initialize matrices
    for (int i = 0; i < N * N; ++i) {
        A[i] = static_cast<float>(i % 100) * 0.5f;
        B[i] = static_cast<float>(i % 100) * 0.25f;
    }

    const int trials = 5;
    double scalar_total = 0;
    double simd_total = 0;

    // Run scalar
    for (int t = 0; t < trials; ++t) {
        auto start = std::chrono::high_resolution_clock::now();
        matmul_scalar(A, B, C_scalar, N);
        auto end = std::chrono::high_resolution_clock::now();
```

```
        scalar_total += std::chrono::duration<double, std::milli>(end - start).count();
    }

    // Run SIMD
    for (int t = 0; t < trials; ++t) {
        auto start = std::chrono::high_resolution_clock::now();
        matmul_simd(A, B, C_simd, N);
        auto end = std::chrono::high_resolution_clock::now();
        simd_total += std::chrono::duration<double, std::milli>(end - start).count();
    }

    // Verify correctness
    for (int i = 0; i < N * N; ++i) {
        if (fabs(C_scalar[i] - C_simd[i]) > 1e-2f) {  // 1e-2 because floating point error increases
            std::cerr << "Mismatch at index " << i << ": " << C_scalar[i] << " vs " << C_simd[i] <<
"\n";
            break;
        }
    }

    # Calculate speedups
    double scalar_avg = scalar_total / trials;
    double simd_avg = simd_total / trials;

    # Print results
    std::cout << "N = " << N << "\tScalar Avg: " << scalar_avg << " ms" << "\tSIMD Avg: " << simd_avg
<< " ms" << "\tSpeedup: " << (scalar_avg / simd_avg) << "x\n";

     # Free matrices
    _aligned_free(A);
    _aligned_free(B);
    _aligned_free(C_scalar);
    _aligned_free(C_simd);
}

int main() {
    std::cout << "Running SIMD Matrix Multiplication tests...\n";
    for (int exp = 6; exp <= 10; ++exp) {  // 2^6 = 64, 2^7 = 128, ..., 2^10 = 1024
        int N = 1 << exp;
        run_test(N);
    }
    std::cout << "Done.\n";
    return 0;
}
```

8.2 CUDA Code

The following code listings implement CUDA-based vector addition and matrix multiplication using CuPy, a GPU array library for Python that interfaces with CUDA. Each experiment includes timing, error checking, and scaling tests across input sizes.

8.2.1 Vector Addition

This python script performs vector addition on both CPU and GPU using NumPy and CuPy. Execution times are recorded for comparison, and output is validated across different input sizes.

```
import cupy as cp
import numpy as np
import time
import pandas as pd
import matplotlib.pyplot as plt

vector_results = []

# Perform vector addition
def run_vector_addition(N):
    # Create random vectors
    A_cpu = np.random.rand(N).astype(np.float32)
    B_cpu = np.random.rand(N).astype(np.float32)

    A_gpu = cp.asarray(A_cpu)
    B_gpu = cp.asarray(B_cpu)

    # CPU timing
    start_cpu = time.perf_counter()
    C_cpu = np.add(A_cpu, B_cpu)
    end_cpu = time.perf_counter()
    cpu_time = (end_cpu - start_cpu) * 1000

    # GPU timing
    cp.cuda.Device(0).synchronize()
    start_gpu = time.perf_counter()
    C_gpu = cp.add(A_gpu, B_gpu)
    cp.cuda.Device(0).synchronize()
    end_gpu = time.perf_counter()
    gpu_time = (end_gpu - start_gpu) * 1000

    # Verify correctness
    C_gpu_cpu = cp.asnumpy(C_gpu)
    error = np.max(np.abs(C_cpu - C_gpu_cpu))

    # Print results
    print(f"N = {N} | CPU Time: {cpu_time:.6f} ms | CUDA Time: {gpu_time:.6f} ms | Max Error:
{error:.6f}")

    vector_results.append({
        "N": N,
```

```
        "CPU Time (ms)": cpu_time,
        "CUDA Time (ms)": gpu_time,
        "Max Error": error
    })

# Run experiments
for exp in range(10, 21, 2):
    N = 2 ** exp
    run_vector_addition(N)
```

8.2.2 Matrix Multiplication

This Python script carries out matrix multiplication on CPU and GPU using NumPy and CuPy. Timings are collected, and numerical errors between CPU and GPU results are computed to ensure result accuracy across multiple matrix sizes.

```
import cupy as cp
import numpy as np
import time

# Perform matrix multiplication
def run_matrix_multiplication(N):
    # Create random matrices
    A_cpu = np.random.rand(N, N).astype(np.float32)
    B_cpu = np.random.rand(N, N).astype(np.float32)

    A_gpu = cp.asarray(A_cpu)
    B_gpu = cp.asarray(B_cpu)

    # CPU timing
    start_cpu = time.perf_counter()
    C_cpu = np.matmul(A_cpu, B_cpu)
    end_cpu = time.perf_counter()
    cpu_time = (end_cpu - start_cpu) * 1000

    # GPU timing
    cp.cuda.Device(0).synchronize()
    start_gpu = time.perf_counter()
    C_gpu = cp.matmul(A_gpu, B_gpu)
    cp.cuda.Device(0).synchronize()
    end_gpu = time.perf_counter()
    gpu_time = (end_gpu - start_gpu) * 1000

    # Verify correctness
    C_gpu_cpu = cp.asnumpy(C_gpu)
    error = np.max(np.abs(C_cpu - C_gpu_cpu))

    # Print results
```

```
    print(f"N = {N} | CPU Time: {cpu_time:.3f} ms | GPU Time: {gpu_time:.3f} ms | Max Error:
{error:.6f}")

    return N, cpu_time, gpu_time

# Run experiments
for exp in range(6, 11):  # N=2^6=64 up to N=2^10=1024
    N = 2 ** exp
    run_matrix_multiplication(N)
```