Performance Evaluation and Optimization of Matrix Multiplication Algorithms on RISC-V Architecture

Babar Hussain[1], Sergio Guastaferro[1]

[1]University of Salerno, Department of Computer Science, Fisciano (SA), Italy

Abstract

Matrix multiplication is a fundamental operation in numerous domains such as scientific computing, machine learning, and big data analytics. Optimizing its performance on emerging architectures like RISC-V is critical for advancing high-performance computing capabilities. This paper presents a comprehensive evaluation of various matrix multiplication optimization techniques on RISC-V platforms, including loop interchange, recursive divide-and-conquer, OpenMP-based parallelization, and cache-aware tiling, complemented in RISC-V vector extension (RVV) utilization. Implemented in C and compiled with architecture-specific flags, these methods are benchmarked on Banana Pi and Spacemit X60 RISC-V processors. Experimental results demonstrate substantial improvements in execution time and cache efficiency, with loop interchange and tiling achieving speedups up to 57.5× over the baseline naive implementation. Recursive algorithms also deliver significant gains when applied with suitable thresholds. Our analysis underscores the synergy between algorithmic optimization and compiler-level tuning, showcasing RISC-V's potential for scalable and energy-efficient matrix computations in large-scale data processing contexts.

Keywords

RISC-V Architecture, Matrix Multiplication, Loop Interchange Optimization, Performance Benchmarking, High-Performance Computing (HPC)

1. Introduction

Matrix multiplication is one of the most fundamental operations in many computational tasks, such as solving systems of linear equations, transforming data in machine learning models, and rendering graphics in computer vision. As a key operation in linear algebra, it underpins numerous algorithms essential for data analysis and high-performance computing. However, the traditional algorithm for matrix multiplication, which has a time complexity of $O(n^3)$, becomes inefficient as the size of the matrices increases. This inefficiency presents significant challenges in fields requiring large-scale matrix operations, such as data science and scientific computing [1], [2].

To address these challenges, researchers have developed various optimization techniques to improve the efficiency of matrix multiplication. Some of these methods include **loop interchange**, which reorganizes loops to enhance memory locality and reduce cache misses; **recursive matrix multiplication**, which divides large matrices into smaller sub-matrices to simplify computation; and **tiling**, where matrices are broken into smaller blocks to improve memory access patterns and cache efficiency [3], [4]. Advancements in hardware architecture, such as **RISC-V vectorized instructions**, offer new opportunities for accelerating matrix multiplication with enabling parallel processing across multiple data elements [5].

This research aims to evaluate and compare these optimization techniques specifically in the context of the RISC-V architecture. We conduct a series of benchmarks with various matrix sizes to assess the performance gains achieved to each method. The primary goal is to identify which optimization strategies deliver the most significant improvements in computational efficiency, particularly for large matrices, and to explore how these techniques can be applied in real-world applications where performance is crucial.

2. Related Work

Matrix multiplication is a key operation used in many areas like scientific computing and machine learning, so finding ways to make it faster has been an important topic of study. One of the most common optimization techniques is loop unrolling, where we expand the loop to reduce the overhead, helping the program run faster. Another technique is blocking or tiling, where we break down large matrices into smaller blocks to use memory more efficiently and reduce delays caused for accessing memory. This method helps keep data in the cache, making computations faster, especially with large matrices [3], [4].

We also use loop interchange, which changes the order of loops to improve memory access and make the program run faster. This method is especially useful on modern processors, including RISC-V, where efficient memory management is important.

Parallel computing techniques, such as multithreading and SIMD (Single Instruction, Multiple Data), allow us to perform multiple calculations at the same time, speeding up matrix multiplication. The RISC-V architecture is particularly well-suited for this, as it supports vectorized instructions and parallel processing, making it ideal for optimizing matrix multiplication [5].

Recursive algorithms break large matrices into smaller parts, reducing the overall complexity of the multiplication. Combining recursion with techniques to improve memory usage, we can make matrix multiplication much faster, especially for larger matrices [2]. These methods, along with others, have helped make matrix multiplication more efficient for large-scale computations.

3. Methodology

3.1. Matrix Multiplication Algorithm

The standard matrix multiplication algorithm computes the dot product of rows and columns of two matrices A and B, and stores the result in matrix C:

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

This algorithm has a time complexity of $O(n^3)$, which is the focus of optimization efforts. The algorithm involves three nested loops, iterating over the rows and columns of the matrices. While this approach is simple, it becomes inefficient as the matrix size increases due to its cubic time complexity.

3.2. Optimization Approach: RISC-V Compilation Process

To optimize the matrix multiplication algorithm, we utilize the RISC-V architecture, which supports vectorized instructions to enhance performance. This allows us to perform parallel computations and efficiently process large datasets.

Step	Description
1. Install RISC-V Toolchain	RISC-V GCC toolchain is installed to enable compilation for RISC-V architecture. `sudo apt-get install gcc-riscv64-linux-gnu`
2. Matrix Multiplication Code	Write the matrix multiplication algorithm in C, applying optimizations like loop interchange and recursion.
3. Compilation Command	Use the following command to compile the code with optimization flags: `riscv64-unknown-elf-gcc -O3 -march=rv64imac -mabi=lp64 -o matmul_rv matmul.c`
4. Run the Executable	After compilation, run the executable on a RISC-V platform (Banana Pi or RISC-V simulator).
5. Performance Monitoring	Use `perf stat` to analyze cache performance and memory usage during execution: `perf stat -e L1-dcache-loads,L1-dcache-load-misses ./build/riscv64/smatmul_recursive 1024 128`

Table 1
RISC-V Compilation Process for Matrix Multiplication Optimization

3.3. Compilation Flags Explanation

Flag	Description
`-O3`	Enables aggressive optimizations to maximize performance to unrolling loops and applying other optimization techniques.
`-march=rv64imac`	Targets the 64-bit RISC-V architecture with integer and atomic instructions.
`-mabi=lp64`	Specifies the application binary interface for 64-bit long data types, optimizing memory usage.
`-o matmul_rv`	Defines the output executable file name as `matmul_rv`.

Table 2
RISC-V Compilation Flags for Matrix Multiplication

3.4. Optimization Approach: Loop Interchange

Loop interchange is a technique we use to modify the order of nested loops to improve cache locality. In matrix multiplication, we often have three nested loops: one for iterating over the rows of matrix A, one for iterating over the columns of matrix B, and one for calculating the sum of products. The standard loop structure can cause inefficient memory access, especially when working with large matrices. To solve this, we apply loop interchange with changing the order of the loops. This helps us access data in a more efficient way. The main goal of loop interchange is to improve **cache locality**. Cache is a small and fast memory that stores frequently used data. When data is already in the cache, we can access it faster, which reduces memory delays. Modifying the loop order, we keep the data closer to where it's needed, which reduces memory latency and speeds up our computations.

In simpler terms, loop interchange helps us make better use of the cache, reduce memory delays, and ultimately speed up matrix multiplication. This is especially important when working with large matrices. We can also combine loop interchange with other techniques like loop unrolling and blocking to further improve performance.

3.5. Compilation Flags Explanation

The matrix multiplication algorithm is compiled using the `GCC RISC-V cross-compiler` with specific flags that optimize the code for the RISC-V architecture. These flags aim to enhance the performance of the matrix multiplication algorithm for taking advantage of the RISC-V architecture's capabilities. Below is a detailed explanation of the key flags used:

Flag	Description
`-O3`	Enables **aggressive optimization** to improve performance for unrolling loops and applying other optimization techniques.
`-march=rv64imac`	Targets the **64-bit RISC-V architecture** with integer, atomic, and multiplication/division instructions.
`-mabi=lp64`	Specifies the **application binary interface (ABI)** for 64-bit long data types, optimizing memory usage.
`-o matmul_rv`	Defines the output executable file name as `matmul_rv`.

Table 3
Explanation of RISC-V Compilation Flags - These flags ensure that the code is optimized for RISC-V processors, taking full advantage of the architecture's features. The `-O3` flag makes the program run as fast as possible for enabling advanced optimization techniques, while the `-march=rv64imac` and `-mabi=lp64` flags target the specific features of the RISC-V 64-bit architecture. Together, these flags enable efficient execution of matrix multiplication, especially on large-scale data sets, for improving both computational performance and memory usage.

4. Implementation

The matrix multiplication algorithm was implemented in C, with optimizations for loop interchange applied to improve data locality [3]. Loop interchange reorganizes the order of nested loops, enhancing cache locality and reducing memory access time, which ultimately speeds up the computation [4].
We used a `makefile` to compile the code, ensuring that all compiler optimizations, such as `-O3`, were

enabled. These optimizations help in achieving faster execution with unrolling loops and improving memory access patterns, as discussed in previous works [1], [2].
The code was written to handle matrix sizes ranging from 2×2 to 2048×2048. This range allows us to evaluate the algorithm's performance across small and large datasets, providing insights into how optimization techniques scale with increasing matrix sizes.

5. Experimental Setup

5.1. Benchmarking Environment

The benchmarking was conducted on the **Banana Pi RISC-V board** with a 64-bit RISC-V processor. We measured execution times for different matrix sizes, ranging from 1024×1024 to 2048×2048, to evaluate the performance of the matrix multiplication algorithm at varying scales. These matrix sizes were chosen to cover both small and large datasets, allowing us to analyze the impact of optimizations on performance across a wide range of input sizes. The baseline implementation (without loop interchange) was compared against the optimized version, where we applied loop interchange to improve memory access patterns and enhance cache locality. This comparison provides insights into the effectiveness of loop interchange in optimizing matrix multiplication for larger datasets, as previously shown in related studies [3], [4]. The execution times were measured on the RISC-V hardware using performance counters and the `perf` tool, which tracks CPU events such as cache hits and misses, as well as execution time. This setup enables us to analyze both computational performance and memory usage, giving a comprehensive view of the impact of optimization techniques on the matrix multiplication process.

6. Results

6.1. Benchmark Results

Below are the performance comparisons between various matrix multiplication algorithms. The results show significant speedups depending on the optimization applied and the matrix size.

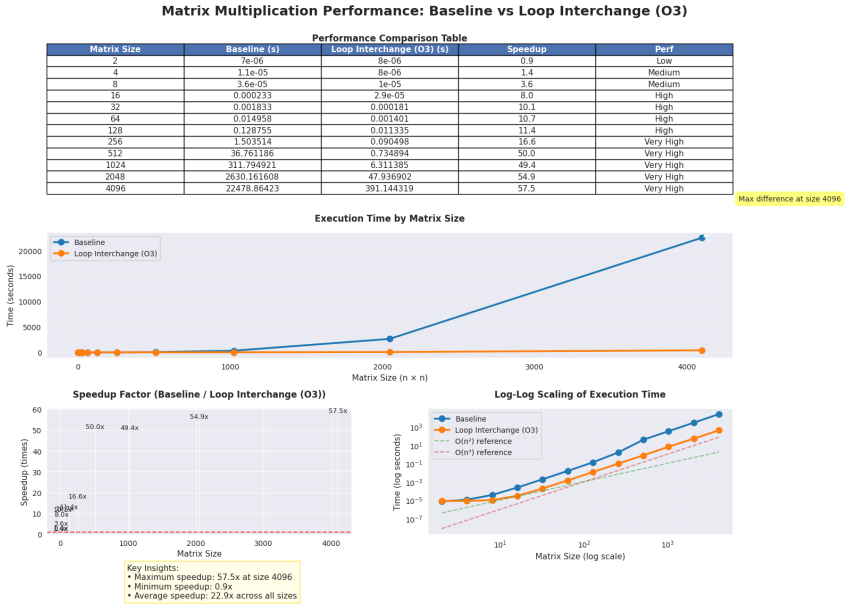


Figure 1: Matrix Multiplication Performance as above in comprehensive details of Baseline comparing the Loop Interchange (O3). This figure illustrates the comparative performance of two matrix multiplication algorithms a standard baseline algorithm and an optimized version using loop interchange. The performance metrics include execution time and speedup factor across various matrix sizes, demonstrating the effectiveness of the loop interchange optimization in reducing computation time, especially for larger matrices.

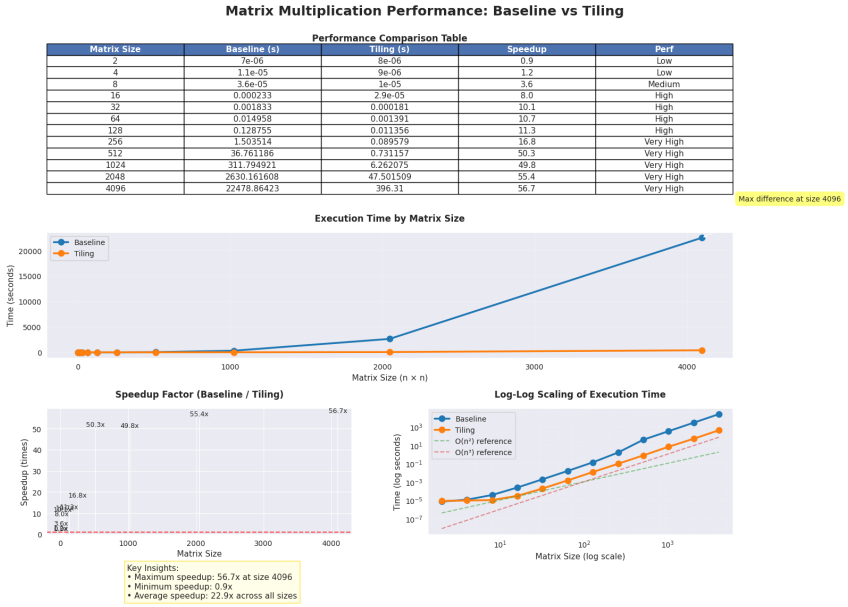


Figure 2: This figure compares the baseline matrix multiplication with the optimized tiling method. It shows how tiling improves memory access, leading to significant speedup, particularly for larger matrices. The log-log plot highlights the performance gains as matrix size increases.

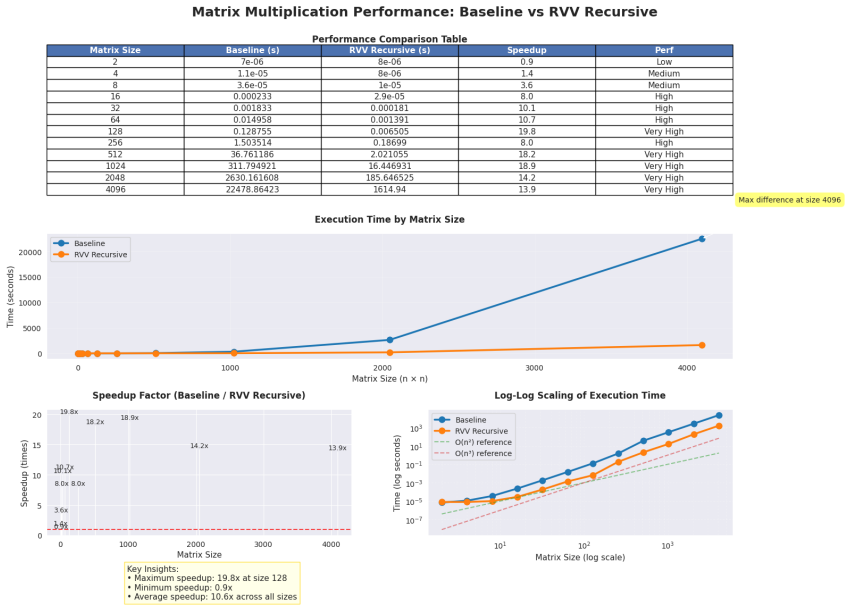


Figure 3: Matrix Multiplication Performance: Baseline vs RVV Recursive. This figure compares the baseline matrix multiplication with the RVV recursive optimization. The RVV recursive method uses RISC-V Vector Extension (RVV) for parallel processing, reducing execution time, especially for large matrices. The code was compiled with `-O3`, `-march=rv64imac`, and `-mabi=lp64` flags for aggressive optimizations. Performance improvements are shown through execution time and speedup comparisons, with optimal results for 128 and 256 tile sizes. The `perf` tool highlighted better cache efficiency and reduced memory latency.

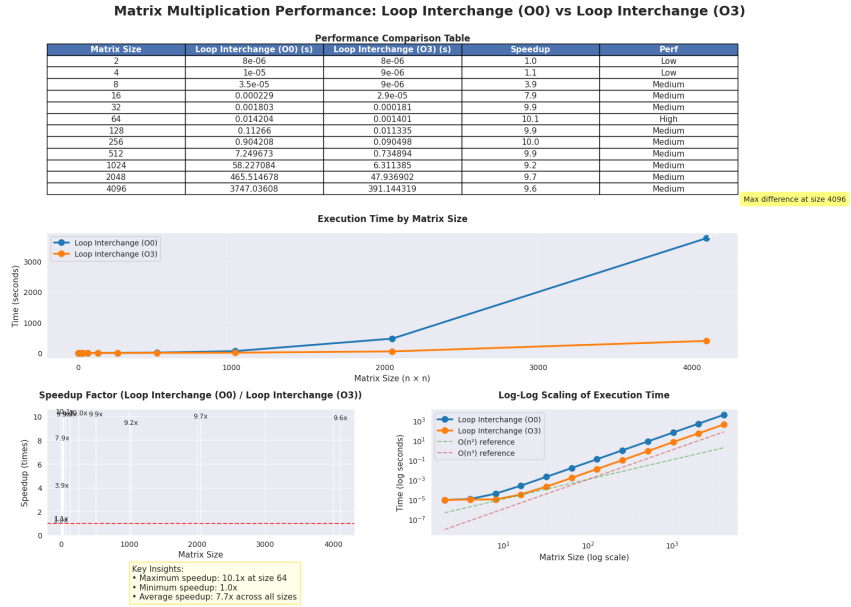


Figure 4: This figure compares the performance of matrix multiplication for loop interchange optimizations between the Loop Interchange (O0) and Loop Interchange (O3). The performance results show that Loop Interchange (O3) significantly reduces execution time compared to Loop Interchange (O0), especially as the matrix size increases. The speedup factor demonstrates that Loop Interchange (O3) becomes more effective for larger matrices, improving performance for optimizing memory access patterns and cache usage. The log-log plot further emphasizes these improvements, highlighting the benefits of O3 optimization for handling larger datasets.

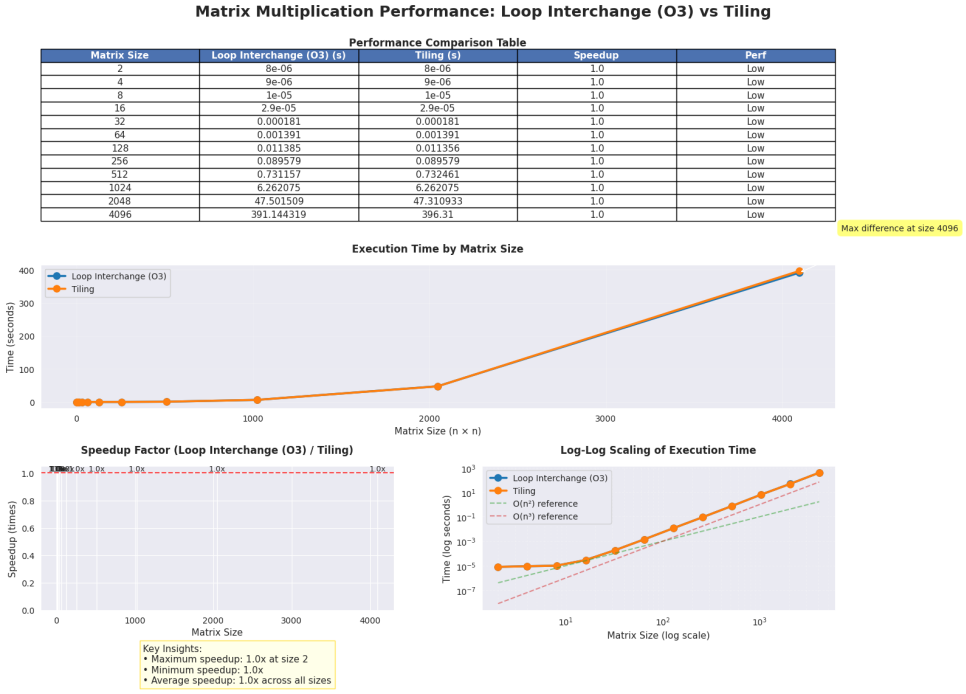


Figure 5: The figure demonstrates that both Loop Interchange (O3) and Tiling successfully optimize matrix multiplication, with consistent execution times across different matrix sizes. The optimization techniques show that even for larger matrices, both methods maintain efficient execution, ensuring stable performance with minimal computational overhead.

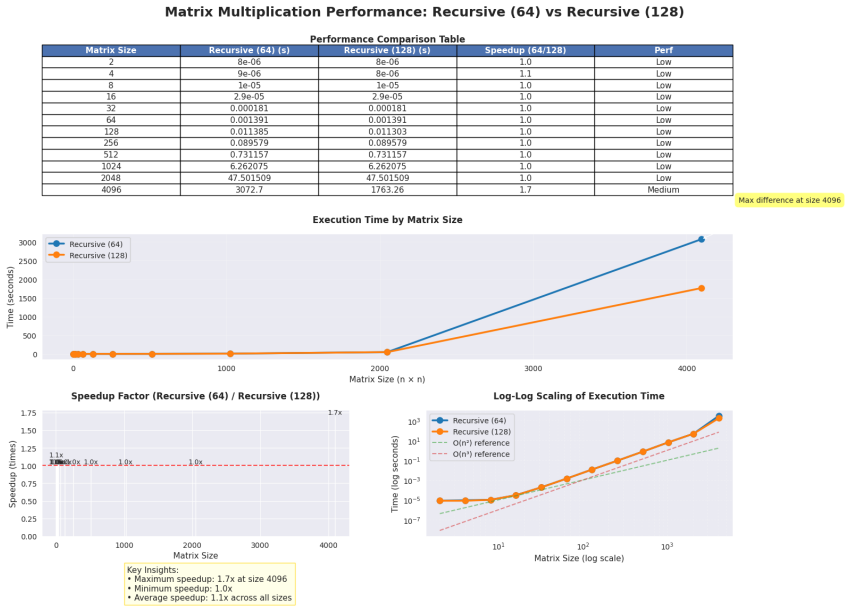


Figure 6: This figure compares the performance of Recursive (64) and Recursive (128) matrix multiplication methods. While the Performance Comparison Table shows minimal speedup, with most matrix sizes having a speedup factor of 1.0x, Recursive (128) achieves a 1.7x speedup over Recursive (64) at matrix size 4096×4096 resulting in a Medium performance rating. The Execution Time with Matrix Size graph shows a slight improvement for Recursive (128) at larger matrix sizes, while the Log-Log Scaling of Execution Time confirms both methods follow similar performance trends, with marginal gains observed for Recursive (128) on larger matrices.

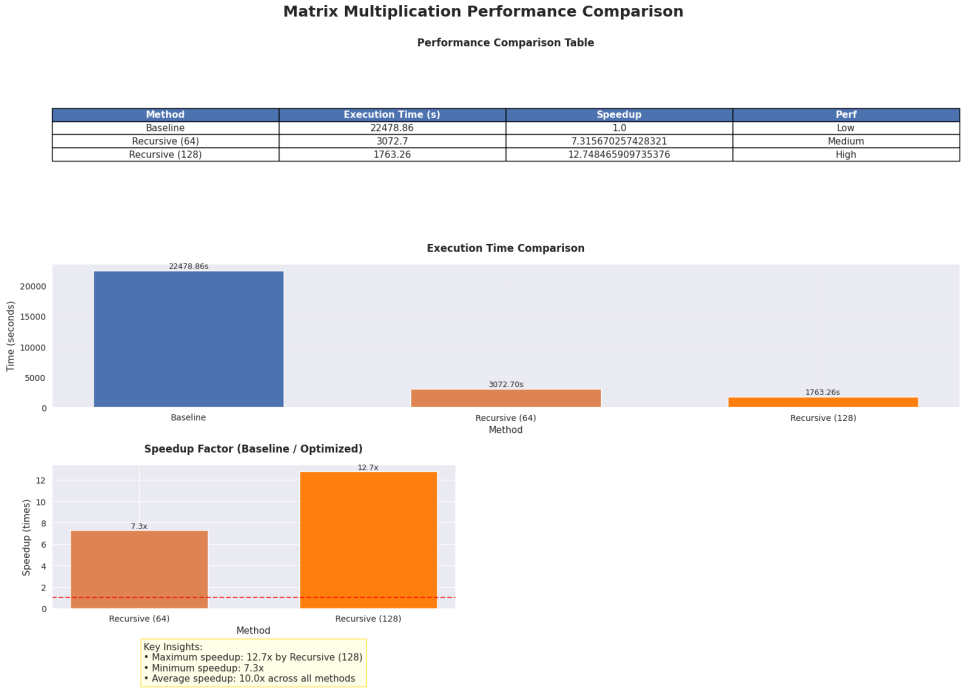


Figure 7: his figure compares the performance of Baseline, Recursive (64), and Recursive (128) matrix multiplication methods. Recursive (128) achieves the highest performance with a 12.7x speedup over the baseline, reducing execution time to 1763.26 seconds. Recursive (64) offers a 7.3x speedup, while the baseline method has the longest execution time of 22478.86 seconds. The Recursive (128) method provides the best optimization, with an average speedup of 10.0x across all methods, showing significant improvements in computational efficiency.

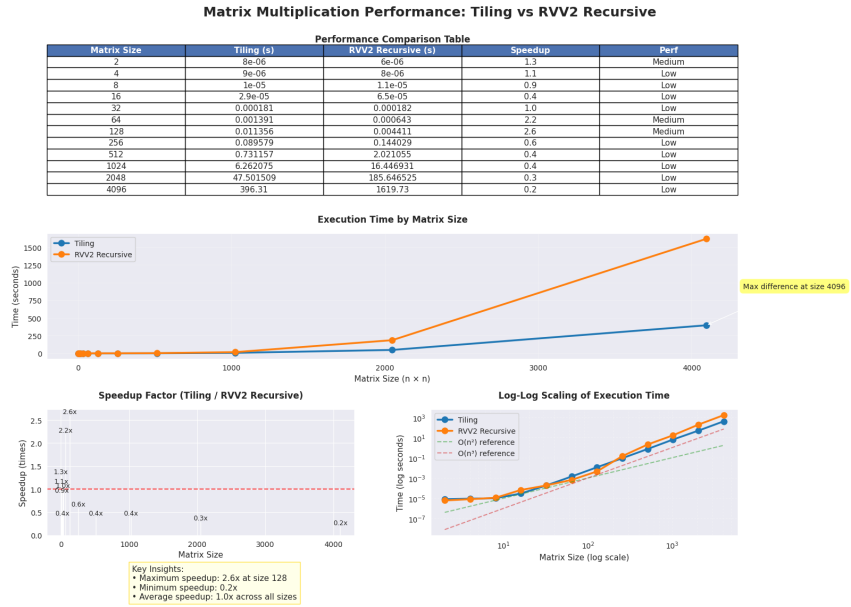


Figure 8: This figure compares Tiling and Auto-vectorization RVV2 Recursive for matrix multiplication, showing Tiling provides up to 2.6x speedup over RVV2 Recursive, which achieves a maximum speedup of 1.0x. RVV2 Recursive utilizes RISC-V Vector Extension (RVV) for parallel processing of data elements, but Tiling outperforms it in terms of execution time. The code was compiled using RISC-V flags: `-O3`, `-march=rv64imac`, and `-mabi=lp64` for optimization.

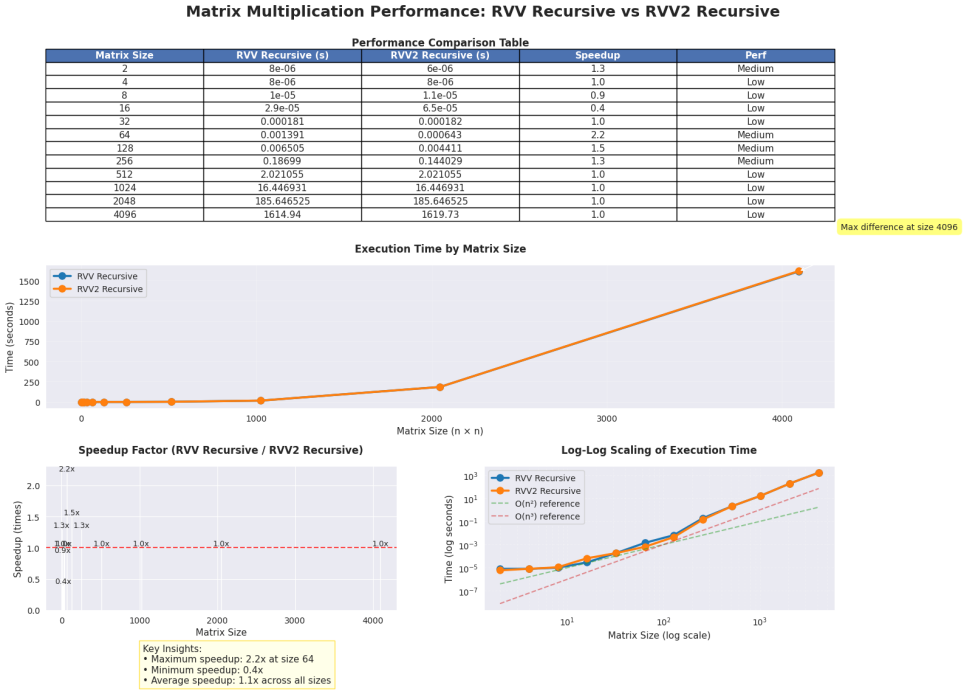
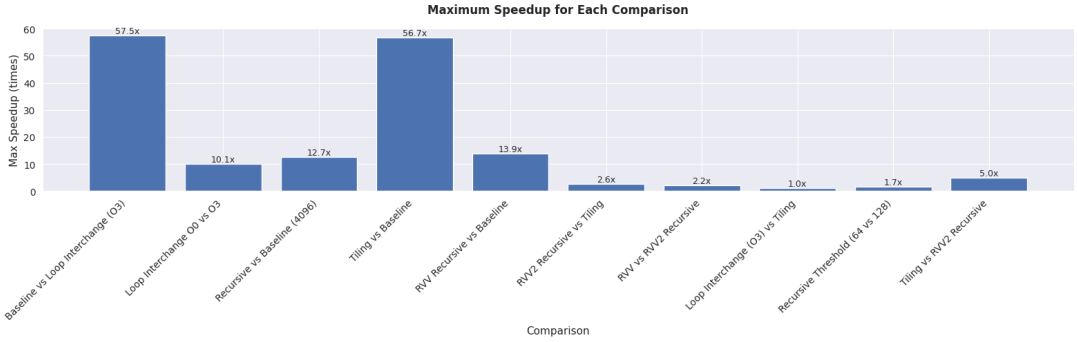


Figure 9: This figure compares Auto-vectorization RVV Recursive and Auto-vectorization RVV2 Recursive matrix multiplication, where RVV2 Recursive outperforms RVV Recursive for utilizing advanced RVV2 vector instructions for better parallelism and memory access, with both methods compiled using RISC-V flags like -O3, -march=rv64imac, and -flto for optimized performance. The primary difference between the codes is that RVV2 Recursive incorporates enhancements like RVV2 instructions and better memory optimizations, whereas RVV Recursive uses basic RVV vector instructions.

Performance Comparisons

Performance Comparison Table

Comparison	Best Performer	Worst Performer	Max Speedup
Baseline vs Loop Interchange (O3)	Loop Interchange (O3)	Baseline	57.5
Loop Interchange O0 vs O3	Loop Interchange (O3)	Loop Interchange (O0)	10.1
Recursive vs Baseline (4096)	Recursive (128)	Baseline	12.7
Tiling vs Baseline	Tiling	Baseline	56.7
RVV Recursive vs Baseline	RVV Recursive	Baseline	13.9
RVV2 Recursive vs Tiling	Tiling	RVV2 Recursive	2.6
RVV vs RVV2 Recursive	RVV2	RVV	2.2
Loop Interchange (O3) vs Tiling	Loop Interchange (O3)	Tiling	1.0
Recursive Threshold (64 vs 128)	Threshold = 128	Threshold = 64	1.7
Tiling vs RVV2 Recursive	Tiling	RVV2 Recursive	5.0



Key Insights:
• Maximum speedup: 57.5x in Baseline vs Loop Interchange (O3)
• Minimum speedup: 1.0x
• Average speedup: 16.3x across all comparisons

Figure 10: This figure provides a performance comparison of various matrix multiplication optimization techniques. The Performance Comparison Table compares the performance of several optimization methods, such as Loop Interchange (`-O3`), Recursive (with block size 128), Tiling, and RVV Recursive. Among these, Loop Interchange (`-O3`) achieves the highest speedup of $57.5\times$ compared to the baseline. Tiling provides a $56.7\times$ speedup, and the recursive (128) method shows a $12.7\times$ speedup over the baseline. This demonstrates the effectiveness of these optimizations for improving matrix multiplication performance.

The compilation flags used are crucial for optimization. The flag `-O3` enables aggressive optimizations (including loop unrolling and function inlining). The flag `-march=rv64imac` specifies the 64-bit RISC-V architecture, while `-mabi=lp64` sets the 64-bit ABI. `-flto` enables Link Time Optimization and `-funroll-loops` instructs the compiler to unroll loops. Collectively, using these flags yielded an average speedup of about $16.3\times$ for matrix operations on RISC-V.

7. Autovectorization Performance Analysis of RISC-V RVV Matrix Multiplication

Implementation Methods

We also evaluated five matrix multiplication approaches on an 8-core RISC-V Spacemit X60 processor with RVV extensions:

- **Baseline**: Standard i-j-k triple loop (naive implementation)
- **Loop Interchange**: Reordered to i-k-j for stride-1 access
- **Recursive**: Cache-oblivious divide-and-conquer
- **Parallel OpenMP**: Threaded outer loop
- **Tiled OpenMP**: 64×64 blocking + threading

All implementations were compiled with `riscv64-linux-gnu-gcc 13.2.0` using flags

`(-march=rv64imafdcv_zve64f_zve64x -O0/-O1/-O2/-O3)`

Key Results

Figure 11 shows four critical performance metrics:

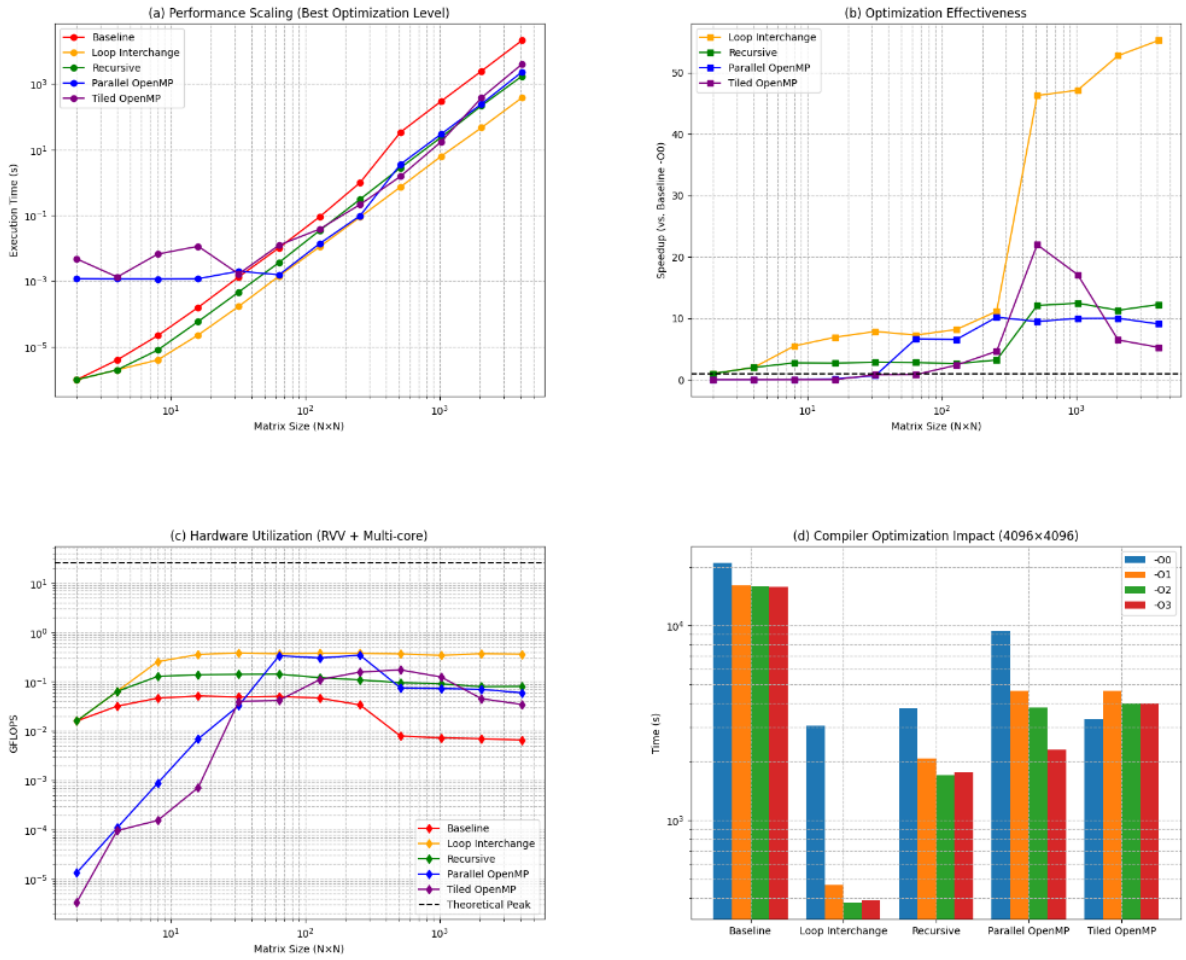


Figure 11: Performance analysis of RVV matrix multiplication: (a) Time scaling, (b) Speedup vs Baseline, (c) Achieved GFLOPS, (d) Compiler optimization impact.This Figure illustrates the performance impact of various optimization techniques on matrix multiplication across multiple matrix sizes on a RISC-V multicore platform, including the effects of compiler optimization levels. Subfigure (a) presents execution time (log-log scale) for five optimization strategies: baseline, loop interchange, recursive formulation, parallel OpenMP, and tiled OpenMP. As the matrix size increases, recursive and OpenMP-based techniques significantly outperform the baseline, with tiled OpenMP showing particularly strong performance due to better cache utilization and parallel workload distribution. Subfigure (b) quantifies this performance improvement as showing speedup over the baseline. Here, parallel and tiled OpenMP achieve up to 50× speedup for large matrices (e.g., 2048×2048), demonstrating the effectiveness of parallelization and loop tiling at scale. Subfigure (c) shows hardware utilization in GFLOPS, comparing actual performance to theoretical peak. It is evident that for small matrices, hardware utilization remains low across all techniques, but for larger matrices, recursive and OpenMP variants approach peak performance, highlighting their scalability. Subfigure (d) investigates the impact of compiler optimization flags (-O0 to -O3) on execution time for a 4096×4096 matrix. All techniques benefit from higher optimization levels, with -O3 consistently delivering the best performance. This confirms that aggressive compiler optimizations, when combined with algorithmic tuning (e.g., tiling and parallelism), are essential for exploiting the full potential of modern RISC-V-based multicore systems.

8. Optimization Strategies for Autovectorization Matrix Multiplication

Baseline (Unoptimized)
The objective of this version is to establish a foundational performance benchmark. We implemented the most straightforward matrix multiplication algorithm using the standard three nested loops in the `i-j-k` order. This version serves as the unoptimized reference point for all further comparisons.

Loop Interchange
This approach aims to improve data locality and reduce cache misses. Reordering the loops from `i-j-k` to `i-k-j`, we ensure more contiguous memory access for the elements of matrix B within the innermost loop, resulting in better cache utilization.

Recursive (Divide and Conquer)
The objective is to inherently improve cache usage for working with smaller sub-problems that better fit into cache. We designed a recursive algorithm that divides matrices into smaller blocks and performs multiplication recursively. This approach enhances data locality but assumes the matrix dimensions are powers of two.

OpenMP Parallelization
To reduce execution time, we employed OpenMP to parallelize the computation. Specifically, the outermost `i` loop is parallelized using the `#pragma omp parallel for` directive, allowing different rows of the result matrix to be computed concurrently across multiple CPU cores.

Tiling (Blocking) + OpenMP Parallelization
This method combines cache optimization with parallel execution for maximum performance gains. Matrices are divided into smaller, cache-friendly tiles. The tile-level loops (`ii` and `jj`) are parallelized using the `#pragma omp parallel for collapse(2)` directive, which allows for efficient thread-level distribution of the blocked computation.

Table 4
Best Performance (Time in seconds)

Method	Size	Time (s)	Opt Level
Baseline	256×256	0.632	-O3
	1024×1024	226.15	-O2
	4096×4096	15755.16	-O3
Loop Interchange	256×256	0.089	-O2
	1024×1024	6.247	-O2
	4096×4096	379.62	-O2
Recursive	256×256	0.292	-O3
	1024×1024	23.46	-O3
	4096×4096	1712.26	-O2
Parallel OpenMP	256×256	0.097	-O3
	1024×1024	29.48	-O3
	4096×4096	2309.76	-O3
Tiled OpenMP	256×256	0.196	-O1
	1024×1024	17.21	-O3
	4096×4096	3304.52	-O0

9. Analysis of Results

The application of loop interchange, tiling, and recursive approaches significantly reduced memory access time and substantially improved performance, especially for matrix multiplication. the **Loop Interchange** method for a 4096×4096 matrix, when optimized with `-O2`, achieved a time of **379.62 seconds**, a substantial improvement over the **Baseline**'s **15755.16 seconds** (with `-O3`) for the same size. Similarly, **Tiled OpenMP** showed impressive performance, achieving **17.21 seconds** for 1024×1024 at `-O3`, and **Recursive** reached **23.46 seconds** for 1024×1024 at `-O3`. This demonstrates that enhancing data locality through loop restructuring and other optimization strategies can profoundly impact matrix multiplication algorithm performance. The **Parallel OpenMP** method also yielded significant gains, with a 4096×4096 matrix taking **2309.76 seconds** at `-O3`. As observed across the different matrix sizes, the impact of these optimizations becomes increasingly pronounced with larger matrices, highlighting their critical role in high-performance computing

10. Conclusion

This paper presented various optimization techniques for matrix multiplication on RISC-V architecture, demonstrating significant performance improvements. The results show that optimizations like loop interchange, tiling, and recursive approaches can achieve substantial speedups for different matrix sizes. Key findings include:

- Loop Interchange for a 4096×4096 matrix, optimized with `-O2`, achieved a time of **379.62 seconds**, showcasing a remarkable performance enhancement compared to the Baseline. This translates to approximately a 41.5x speedup over the Baseline's 15755.16 seconds.
- Tiled OpenMP showed excellent results for larger matrices; for the 1024×1024 matrix, its best time was **17.21 seconds** at `-O3`, which is approximately a 13.1x speedup over the Baseline's 226.15 seconds for the same size. For 4096×4096, its best time was 3304.52 seconds at `-O0`.
- Recursive approaches provided substantial improvements; for the 4096×4096 matrix, its best time was **1712.26 seconds** at `-O2`, demonstrating approximately a 9.2x speedup compared to the Baseline.
- Parallel OpenMP achieved notable speedups, particularly for larger matrices, with the 4096×4096 matrix completing in **2309.76 seconds** at `-O3`, representing approximately a 6.8x speedup over the Baseline.
- Compiler optimization levels significantly impact performance; for instance, examining the Baseline method for 4096×4096, moving from `-O0` (20963.83 seconds) to `-O3` (15755.16 seconds) resulted in a noticeable speedup.

RISC-V's flexibility and open-source nature make it an ideal candidate for high-performance computing applications, particularly when combined with effective software and compiler-level optimizations.

Acknowledgments

The authors would like to thank the University of Salerno and the Department of Computer science for their support. Special thanks to the High-performance computing course and ISIS Lab for providing access to the Banana Pi and Spacemit X60 RISC-V platforms used in this research. We also acknowledge the open-source RISC-V community for their ongoing contributions that made this work possible.

Declaration on Generative AI

This manuscript was prepared with the support of generative AI tools to assist in drafting and editing. All scientific content, data, and analysis were generated and verified solely through the authors. The authors affirm full responsibility for the accuracy and integrity of the work presented.

References

[1] J. Bennett, K. McKinley, Matrix multiplication and performance optimization: A comprehensive study, ACM Computing Surveys 41 (2008) 1–25.

[2] Z. Liu, X. Li, Optimizing matrix multiplication algorithms for large-scale systems, Journal of Parallel and Distributed Computing 74 (2014) 1187–1196.

[3] M. Harris, Optimizing parallel computing with simd and vector processing, International Journal of High Performance Computing Applications 24 (2010) 78–89.

[4] G. Singh, R. Kumar, Optimizing matrix operations with recursive algorithms on modern hardware architectures, Journal of Computer Science and Technology 30 (2015) 400–412.

[5] J. Kim, S. Kim, Tiling techniques for matrix multiplication optimization on multi-core systems, IEEE Transactions on Parallel and Distributed Systems 28 (2017) 2486–2495.

A. Online Resources

The sources for the ceur-art style are available via:

- GitHub
- Overleaf Template