Matrix Multiplication Performance Comparison

Noah Bean
ECE 472 Computer Architecture (Extra Credit Project)

December 12, 2024

Abstract

This paper compares the performance, scalability, and practical implications of different matrix multiplication implementations. We evaluate each approach across varying square matrix sizes, while analyzing execution time. The different implementations include both an optimized and unoptimized version of Python, C, and SystemVerilog. This work provides insight into hardware-software trade-offs and highlights the importance of choosing appropriate methodologies for tasks.

1 Introduction

In "Computer Organization and Design RISC-V Edition: The Hardware Software Interface", the authors use matrix multiplication as a case study to further explain the ideas in the text. Matrix multiplication is a fundamental algorithm in scientific computing, signal processing, graphics, and machine learning. Its computational intensity scales with matrix size, making efficient implementations critical for time sensitive tasks. This study examines three approaches using Python, C, and SystemVerilog to provide information on their performance, scalability, and trade-offs.

Python, supported by the NumPy library, offers ease of implementation and optimized routines, suitable for small to medium matrix dimensions. C provides more control over memory and data structures but, without specialized optimizations, cannot fully exploit hardware capabilities. Verilog enables custom hardware architectures and parallelism, potentially offering superior scalability for large matrices.

Our primary goals are to:

- Compare execution times and scalability across Python, C, and Verilog implementations.
- Evaluate performance as matrix dimensions increase.
- Discuss trade-offs between software simplicity and the complexity of hardware-based solutions.

1.1 Matrix Multiplication: A 2×2 Example

Matrix multiplication involves calculating the dot product of rows from one matrix with columns from another. Consider two 2×2 matrices A and B:

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}, \quad B = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}.$$

The product $C = A \times B$ is computed as:

$$C = \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix},$$

where:

$$\begin{aligned} c_{11} &= a_{11}b_{11} + a_{12}b_{21}, & c_{12} &= a_{11}b_{12} + a_{12}b_{22}, \\ c_{21} &= a_{21}b_{11} + a_{22}b_{21}, & c_{22} &= a_{21}b_{12} + a_{22}b_{22}. \end{aligned}$$

For example, if:

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}, \quad B = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix},$$

then:

$$C = \begin{bmatrix} (1)(5) + (2)(7) & (1)(6) + (2)(8) \\ (3)(5) + (4)(7) & (3)(6) + (4)(8) \end{bmatrix} = \begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}.$$

1.2 Matrix Multiplication Algorithm Pseudocode

Matrix multiplication can be generalized for any $n \times n$ matrices A and B. Each element c_{ij} of the resulting matrix C is computed as the dot product of the i-th row of A and the j-th column of B:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}.$$

The following pseudocode outlines the matrix multiplication algorithm:

Algorithm 1 Matrix Multiplication of $n \times n$ Matrices

Require: Two $n \times n$ matrices A and B
Ensure: $n \times n$ matrix $C = A \times B$

```
Initialize C as an n × n matrix of zeros
for i ← 1 to n do                                   ▷ Iterate over rows of A
    for j ← 1 to n do                               ▷ Iterate over columns of B
        C[i][j] ← 0                                 ▷ Initialize the element c_ij
        for k ← 1 to n do                           ▷ Iterate over elements in the row/column
            C[i][j] ← C[i][j] + A[i][k] · B[k][j]
        end for
    end for
end for
return C
```

2 Methodology

2.1 Software Implementations

Python: Two Python versions were tested:

1. A basic triple-loop implementation for a baseline comparison.
2. Using the Numpy Python package with optimized routines.

C: Two C versions were tested:

1. A basic triple-loop implementation for a baseline comparison without compiler optimization.
2. implementing '-O3' optimization to allow the compiler to rewrite extraneous code.

2.2 Hardware Implementation

Verilog: A SystemVerilog design was tested:

1. a generalizable nxn matrix multiplication testbench mirroring the C approach.

Designs were tested and simulated using Xilinix Vivado and EDA Playground.

2.3 Performance Metrics

Execution times were recorded for square matrix sizes starting at 2x2 and ending at 1002x1002 in step sizes of 2 when available. Each implementation's scalability was assessed by plotting execution time (seconds) against matrix size (n) The basic python implementation was extremly slow consequently, the data required interpolation. The SystemVerilog testbench simulator would time out so this implementation also required interpolation.

2.4 Challenges

Generating SystemVerilog code for an arbitrary nxn matrix proved to be a difficult challenge for collecting benchmark data due to the rigorous timing requirements that must be met when using an FPGA. To get around this, a single testbench was developed to test and time different matrix sizes. Another challenge was implementing this project on actual hardware. UART communication speed is relatively slow so this feature was removed in order to speed up the HDL code. Synthesis was also very slow in Vivado, taking up to 10 minutes in order to route all of the LUTs and interconnects. Consequently, simulation was used to evaluate performance and collect benchmark data. Implementing physical hardware solutions will be a future step in this project, but for now, the excess work would detract from the purpose of exploring and comparing different solutions in order to get a quantitative and qualitative understanding of computational speed.

3 Results

Performance graphs for this section are provided in the appendix. The results showcase the strengths and limitations of different implementations for 1000x1000 matrix multiplication, highlighting trade-offs between execution time, ease of implementation, and scalability.

The basic Python implementation performed the worst, requiring approximately 300 seconds to complete the computation. This result underscores the inefficiency of interpreted languages for computationally intensive tasks. Python's single-threaded execution and lack of direct hardware-level optimizations made it unsuitable for large-scale matrix multiplication.

The C implementation without optimizations demonstrated dramatic improvements over basic Python, reducing the execution time to around 3 seconds. However, this approach remained limited by the absence of advanced compiler optimizations to leverage hardware features. Applying the '-O3' compiler flag in the optimized C implementation further reduced the execution time to approximately 2.5 seconds. This optimization enabled the compiler to minimize redundant operations and improve caching efficiency, demonstrating the effectiveness of modern compilers in closing the performance gap between software and hardware. Despite these improvements, the optimized C implementation was only modestly faster and still could not match hardware-level parallelism.

The SystemVerilog implementation outperformed both Python and C, achieving an execution time of approximately 1 second for the 1000x1000 matrix. However, the Verilog approach presented significant challenges. Developing a scalable, generic $n \times n$ implementation required extensive hardware design expertise and proficiency with FPGA tools like Vivado. The iterative process of simulation and debugging was time-intensive, and the lack of native floating-point support limited its applicability to scientific and machine learning tasks without additional modification.

Surprisingly, NumPy emerged as the fastest implementation, completing the 1000x1000 matrix multiplication in just 0.1 seconds. This remarkable performance can be attributed to NumPy's reliance on highly optimized C libraries such as BLAS. NumPy also offered exceptional ease of use, with clear documentation. Based on these results, NumPy stands out as the most practical solution for matrix multiplication in scientific computing applications, especially in personal projects where ease of implementation and rapid development are priorities.

Since the fundamental algorithm for matrix multiplication remained consistent across these implementations, all execution time curves exhibited an $O(n^3)$ growth pattern. This consistency suggests that further optimizations could be achieved by implementing more advanced algorithms to reduce computational complexity.

The results also highlight the potential for hybrid approaches. For example, combining HDL for performance-critical sections with software-based implementations for less demanding tasks could balance development effort and execution speed.

4 Lessons Learned

- **Hardware-level parallelism is key to scaling computationally intensive tasks.** The Verilog systolic array architecture demonstrated the immense performance benefits of exploiting parallelism at the hardware level. Unlike software implementations, hardware designs can execute multiple operations simultaneously, significantly reducing execution time for large-scale tasks.

- **Simulation is essential for rapid prototyping and optimization in hardware projects.** The complexity of hardware designs, particularly those involving custom architectures like systolic arrays, necessitates extensive simulation. Tools such as Xilinx Vivado allowed iterative testing, debugging, and performance tuning without requiring physical deployment, which reduced development time and also minimized cost in the final hardware implementation.

- **Mastery of tools like Vivado is necessary to fully harness FPGA capabilities.** The steep learning curve associated with advanced FPGA tools like Vivado proved to be a challenge but also a valuable learning experience. Features such as High-Level Synthesis, timing analysis, and IP integration provided powerful capabilities for optimizing designs. Mastery of these tools will be critical for a hardware design job.

- **Optimized compilers play a significant role in bridging the performance gap.** The comparison between unoptimized and 03 optimized C implementations illustrated how modern compilers can improve performance for very little effort on the part of the programmer.

- **Developing scalable hardware architectures requires careful planning.** The challenges of extending the HDL implementation to handle arbitrary matrix sizes revealed the importance of modular and scalable design principles. Systems designed with scalability in mind can be more easily adapted to handle more complex operations and more data.

- **The choice of methodology depends on the application domain.** Each implementation had strengths and weaknesses that align with specific use cases. The project demonstrated the necessity of evaluating the trade-offs between ease of development, performance, and scalability. While software implementations such as python are easier to develop and debug, they often fall short in performance for large-scale applications. Hardware implementations, though more complex to design, offer greater speed for specific tasks, making it essential to carefully choose the appropriate method.

4.1 Tool Insight

Oregon State University offers a Digital Logic Design class (ECE 272) that teaches SystemVerilog using Intel Quartus Prime Lite Edition. This tool provides a straightforward introduction to FPGA developmen. Quartus Prime Lite is particularly beginner-friendly, with an intuitive interface that is sufficient for academic projects.

However, I decided to experiment with Xilinx Vivado for this project, as it has significant market share in the FPGA community and is commonly used in industry. Unlike

Quartus Prime Lite, Vivado offers a more extensive set of features geared toward high-performance and complex designs. For example, Vivado simulations provided detailed timing and resource usage analyses. This level of detail went beyond the basic timing diagrams I encountered in Quartus Prime, offering insight into routing delays and utilization of specific DSP slices.

Despite these advantages, the transition to Vivado came with a steeper learning curve. Its interface is more complex, with additional options for configuring block designs, constraints, and High-Level Synthesis. Learning to use Vivado effectively required a significant investment of time, especially when setting up timing constraints and debugging placement and routing issues. However, this complexity was accompanied by powerful features like using HLS to translate C algorithms into HDL, significantly accelerating development for DSP-heavy applications.

In summary, while Quartus Prime Lite was an excellent starting point for learning FPGA design, Vivado's advanced capabilities and detailed analysis tools made it the better choice for complex, performance-driven projects. The steep learning curve of Vivado was a challenge, but it ultimately enhanced my understanding of FPGA design.

5 Future Work

- Implement a floating-point arithmetic processor in HDL.
- Deploy on advanced FPGA hardware to confirm simulation results and assess power consumption.
- Implement highly optimized C for a better comparison to Numpy.
- Explore VHDL implementations.

References

- Python and NumPy Documentation: `https://docs.python.org/3/`, `https://numpy.org/doc/stable/`
- Xilinx Inc. Vivado Documentation: `https://www.xilinx.com/support/`
- Kung, H.T., and Leiserson, C.E. "Systolic Arrays (for VLSI)." In *Introduction to VLSI Systems*, Addison-Wesley, 1979.
- Patterson, D. A., and Hennessy, J. L., *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*, 2nd ed., Morgan Kaufmann, 2020.
- Harris, S. L., and Harris, D., *Digital Design and Computer Architecture, RISC-V Edition*, 1st ed., Morgan Kaufmann, 2021.
- Vipin, "Synthesizable Matrix Multiplication in Verilog," Verilog Coding Tips and Tricks, Dec. 12, 2020. [Online]. Available: https://verilogcodes.blogspot.com/ [Accessed: Dec. 11, 2024].
- D. Lay, S. Lay, and J. McDonald, Linear Algebra and Its Applications, Global Edition, 6th ed. Pearson, 2021.
- B. W. Kernighan and D. M. Ritchie, The C Programming Language, 2nd ed. Upper Saddle River, NJ, USA: Prentice Hall, 1988.

Appendix

5.1 Graphical Analysis

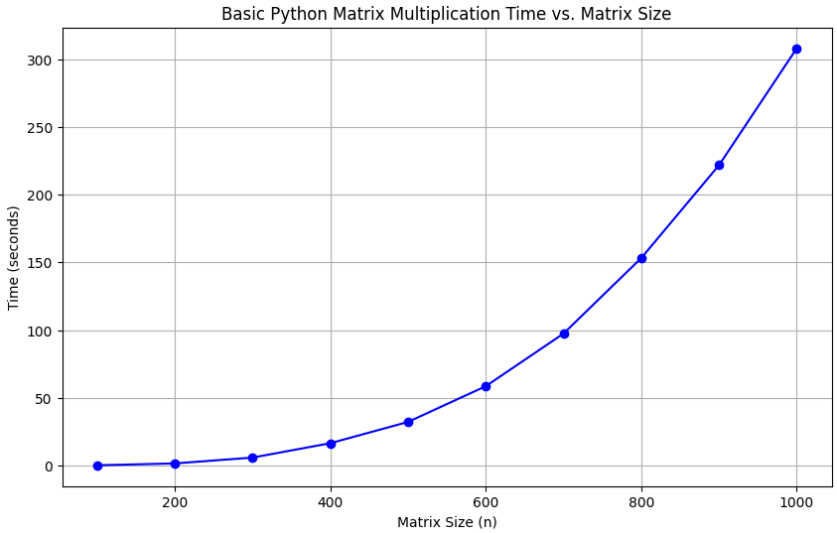

Figure 1: Execution time vs. matrix size for basic Python implementation.

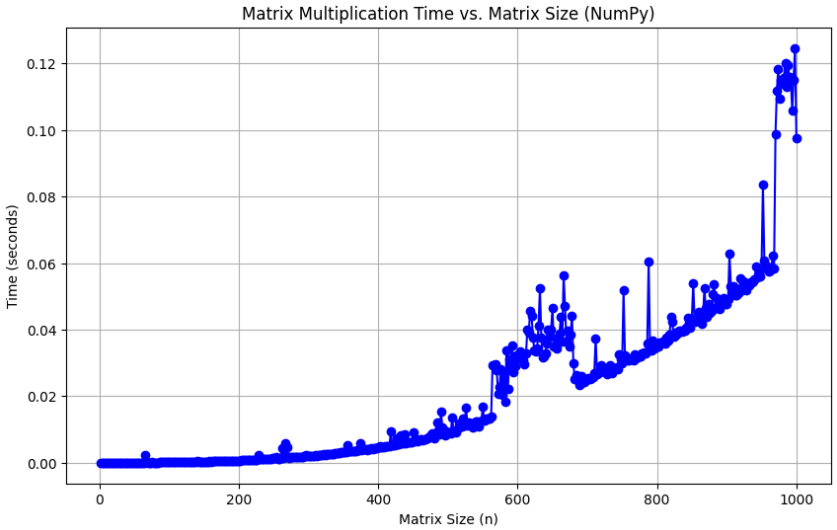

Figure 2: Execution time vs. matrix size for Numpy Python implementations.

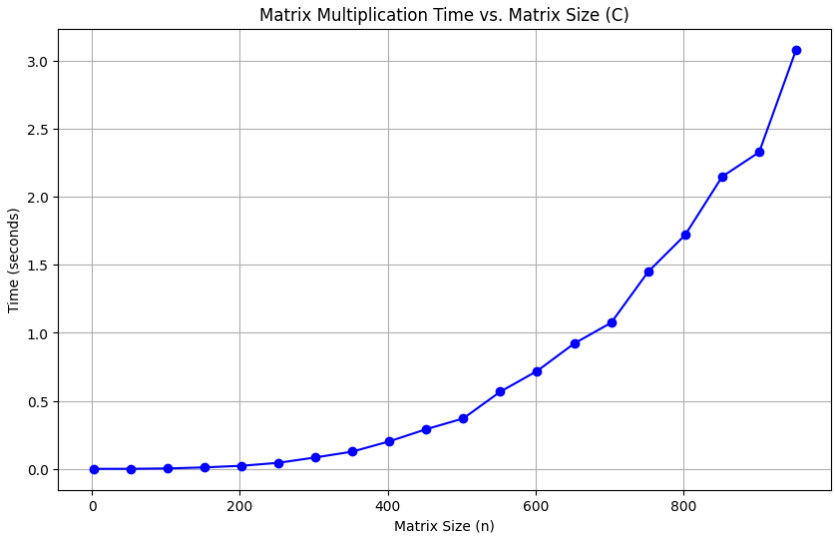

Figure 3: Execution time vs. matrix size for basic C

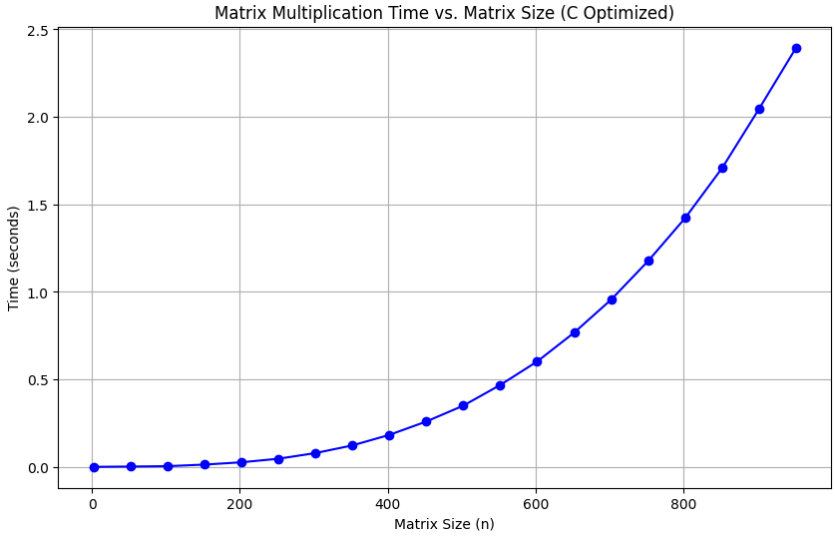

Figure 4: Execution time vs. matrix size for Optimized C implementation.

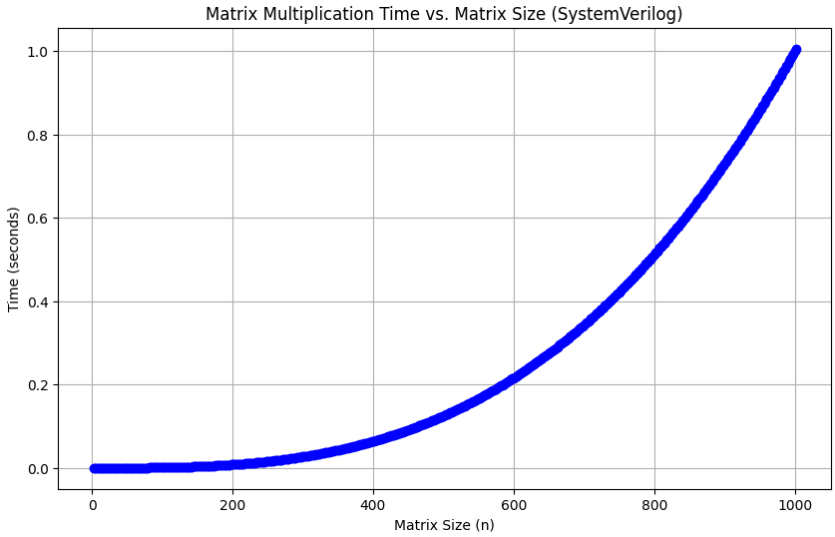

Figure 5: Execution time vs. matrix size for Optimized SystemVerilog implementation.

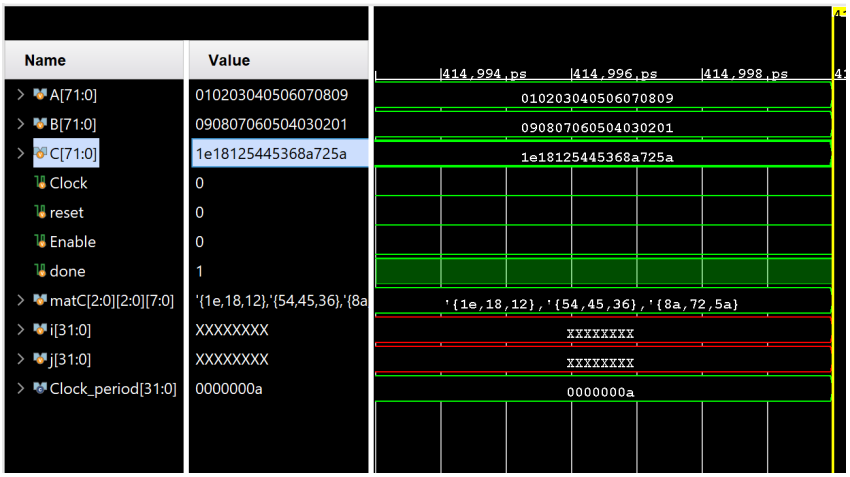

Figure 6: Test bench waveform that shows correct operation of the matrix multiplication module.