The Impact of Matrix Multiplication on Computational Efficiency Across Programming Languages

Shayaan Hooda[1]

[1]Oakton High School Research Project

October 20, 2025

Abstract

Matrix multiplication is a foundational linear algebra operation, used computationally in machine learning, computer graphics, and various other applications. The efficiency of its implementation can have a profound impact on overall application performance. This research paper presents a theoretical framework to design an experiment, which will investigate the impact of programming language selection on the efficiency of matrix multiplication. We conduct a rigorous, first-principles analysis of two canonical algorithms: the naive (schoolbook) method and Strassen's divide-and-conquer algorithm. For each, we derive the time and space complexity from their recurrence relations (in various notations for rigor) and formulate precise, testable hypotheses for each selected programming language. We then outline a comprehensive research methodology to empirically validate these hypotheses across five distinct programming languages: C++, Rust, Java, Python, and JavaScript. This methodology is designed to isolate the performance characteristics inherent to each language's execution strategy – ranging from ahead-of-time (AOT) native compilation and manual memory management (C++ and Rust) to just-in-time (JIT) compilation and automatic garbage collection (Java and JavaScript), and interpretation (Python). By contrasting the theoretical complexities with a structured experimental plan, this work aims to quantify the performance trade-offs associated with different levels of programming abstraction and runtime environments for a computationally intensive kernel, while illustrating how algorithms, which are more complex for humans, may be handled efficiently by computers.

Contents

1 Introduction

Matrix multiplication serves as a cornerstone of linear algebra and is a computationally intensive kernel in a vast array of high-performance applications (such as commonly used LLMs today, including ChatGPT, Gemini, Ollama, etc.). The relentless growth of data-intensive fields has placed an ever-increasing demand on the efficiency of such fundamental algorithms. Take the race of AI currently. Speed and efficiency have significant impacts on an AI's capability, as humanity has started to use machine learning for autonomous tasks and has even considered to replace full jobs with AI. While highly optimized numerical libraries provide state-of-the-art performance and algorithms (which have been fine-tuned for time and space complexity to be readily available for developer use), the choice of programming language itself introduces a crucial layer of abstraction that dictates how powerful and capable a language itself can be. This means that, despite high-level languages offering benefits like garbage collection, readable code, and ease of method applicability, their abstractions can prevent direct control over hardware-level operations. As a result, abstraction may not just contribute to computational bottlenecks, but be a bottleneck itself, as it limits how finely algorithms can be optimized at a fundamental level (Colburn and Shute). Overall, abstraction can contribute to a choice programmers have to make: sacrificing an easier debugging process for raw speed and server resource consumption.

Compiled languages like C++ and Rust offer direct memory management and compile to native machine code, promising performance close to the hardware's theoretical limits. In contrast, languages with managed runtimes like Java and JavaScript rely on Just-In-Time (JIT) compilation and automatic garbage collection, which introduce runtime overhead but can achieve remarkable performance for hot code paths through dynamic optimization (Lin et al.). Interpreted languages such as Python prioritize ease of use but typically incur significant performance penalties for numerical loops, necessitating reliance on external libraries written in lower-level languages (Luo et al.).

The driving research questions this paper addresses are:

- How does the choice of programming language and its underlying execution model impact the empirical performance of matrix multiplication?
- How does this empirical performance relate to the algorithm's theoretical complexity?
- How can we design an experiment to apply and interpret these findings effectively?

To answer these questions, we decouple algorithmic complexity from library-specific optimizations by focusing on "pure" language implementations. We analyze the naive $O(n^3)$ algorithm and Strassen's sub-cubic $O(n^{\log_2 7})$ algorithm (Khan et al.). This dual-algorithm approach allows us to study not only the raw performance of each language but also how language overhead affects the practical crossover point where a theoretically superior but more complex algorithm becomes empirically advantageous.

This paper establishes the theoretical foundation by deriving the time and space complexities for both algorithms from first principles. Based on these derivations, we formulate hypotheses about their expected performance scaling and the relative efficiency ranking of C++, Rust, Java, Python, and JavaScript. We then propose a rigorous experimental methodology to test these hypotheses, detailing implementation strategies, performance measurement techniques, and a specific analysis of how language features are expected to cause divergence from theoretical ideals.

2 Theoretical Foundations

The efficiency of matrix multiplication is fundamentally governed by its algorithmic complexity. We analyze two seminal algorithms that represent different points in the complexity landscape: the standard schoolbook method and Strassen's recursive algorithm.

2.1 The Naive (Schoolbook) Algorithm

The naive algorithm is the direct implementation of the definition of matrix multiplication. For two $n \times n$ matrices A and B, their product $C = AB$ is an $n \times n$ matrix where each element C_{ij} is defined as:

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

2.1.1 Time Complexity Analysis

To compute a single element C_{ij} of the resulting matrix, we perform n multiplications ($A_{ik} \times B_{kj}$ for $k = 1, \ldots, n$) and $n - 1$ additions. The total number of elements in the matrix C is n^2. Hence, the total number of arithmetic operations is:

$$\begin{aligned} \text{Total Operations} &= (\text{Elements in } C) \times (\text{Operations per element}) \\ &= n^2 \times (n \text{ multiplications} + (n-1) \text{ additions}) \\ &= n^3 \text{ multiplications} + n^2(n-1) \text{ additions} \\ &= 2n^3 - n^2 \end{aligned}$$

The dominant term in this expression is $2n^3$. Asymptotically, the time complexity $T(n)$ is cubic in the dimension of the matrix.

$$T(n) = O(n^3)$$

Hypothesis 1. *The execution time of a pure, naive matrix multiplication implementation will exhibit cubic scaling with the matrix dimension n across all tested programming languages when plotted on a log-log scale.*

2.1.2 Space Complexity Analysis

To perform the multiplication, we must store the two input matrices, A and B, and the output matrix, C. Each matrix requires space for n^2 elements.

$$S(n) = \text{Space}(A) + \text{Space}(B) + \text{Space}(C) = n^2 + n^2 + n^2 = 3n^2$$

If the operation can be performed in-place for one of the inputs, space can be reduced to $2n^2$, but the asymptotic complexity remains quadratic.

$$S(n) = O(n^2)$$

Hypothesis 2. *The primary memory footprint of a naive matrix multiplication implementation will scale quadratically with the matrix dimension n.*

2.2 Strassen's Algorithm

In 1969, Volker Strassen discovered a recursive algorithm that performs matrix multiplication with a sub-cubic time complexity (Khan et al.). The algorithm divides the $n \times n$ matrices A, B, C into four $n/2 \times n/2$ sub-matrices:

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

Instead of the 8 recursive multiplications required by the naive divide-and-conquer approach, Strassen's method ingeniously computes the result with only 7 multiplications of $n/2 \times n/2$ matrices and a fixed number of matrix additions and subtractions. The 7 intermediate products are (Khan et al.):

$$\begin{aligned} M_1 &= (A_{11} + A_{22})(B_{11} + B_{22}) \\ M_2 &= (A_{21} + A_{22})B_{11} \\ M_3 &= A_{11}(B_{12} - B_{22}) \\ M_4 &= A_{22}(B_{21} - B_{11}) \\ M_5 &= (A_{11} + A_{12})B_{22} \\ M_6 &= (A_{21} - A_{11})(B_{11} + B_{12}) \\ M_7 &= (A_{12} - A_{22})(B_{21} + B_{22}) \end{aligned}$$

The sub-matrices of the result C are then formed by 18 additions/subtractions of these intermediate products.

2.2.1 Asymptotic notation: definitions (brief)

We will use the standard asymptotic notations. For two positive-valued functions f, g defined on sufficiently large integers:

- $f(n) = O(g(n))$ means $\exists\, C > 0,\ n_0$ such that $f(n) \le Cg(n)$ for all $n \ge n_0$.
- $f(n) = \Omega(g(n))$ means $\exists\, C' > 0,\ n_1$ such that $f(n) \ge C'g(n)$ for all $n \ge n_1$.
- $f(n) = \Theta(g(n))$ means $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.
- $f(n) = o(g(n))$ means $\lim_{n\to\infty} f(n)/g(n) = 0$.
- $f(n) = \omega(g(n))$ means $\lim_{n\to\infty} f(n)/g(n) = \infty$.

2.2.2 Time complexity: exact recurrence and rigorous solution

Let $T(n)$ denote the running time (measured as the number of basic operations, up to constant factors) required by Strassen's algorithm to multiply two $n \times n$ matrices, where n is a power of 2 and $T(1) = T_0 > 0$ is a constant. A straightforward model that counts the recursive multiplications and the cost of the matrix additions/subtractions at each level yields the recurrence

$$T(n) \;=\; 7\,T\Big(\frac{n}{2}\Big) \;+\; \beta n^2,$$

where $\beta > 0$ is a constant representing the cost of the 18 additions/subtractions per recursion level (we take the non-recursive cost per level to be an exact βn^2 for clarity; the same derivation carries through if that term is $\Theta(n^2)$).

Unrolling the recurrence. Unroll the recurrence for k levels:

$$\begin{aligned} T(n) &= 7\,T\Big(\frac{n}{2}\Big) + \beta n^2 \\ &= 7\Big(7\,T\Big(\frac{n}{2^2}\Big) + \beta\Big(\frac{n}{2}\Big)^2\Big) + \beta n^2 \\ &= 7^2 T\Big(\frac{n}{2^2}\Big) + \beta n^2\left(1 + \frac{7}{4}\right) \\ &\;\;\vdots \\ &= 7^k T\Big(\frac{n}{2^k}\Big) + \beta n^2 \sum_{i=0}^{k-1}\left(\frac{7}{4}\right)^i. \end{aligned}$$

Stop unrolling at $k = \log_2 n$ so that $n/2^k = 1$. Set $h = \log_2 n$ (assume n is an exact power of two for simplicity). Then

$$T(n) = 7^h T(1) + \beta n^2 \sum_{i=0}^{h-1}\left(\frac{7}{4}\right)^i.$$

Simplify the two terms. Use $7^h = 7^{\log_2 n} = n^{\log_2 7}$ (the identity $a^{\log_b n} = n^{\log_b a}$) and evaluate the geometric sum (ratio $r = \frac{7}{4} > 1$):

$$\begin{aligned} \sum_{i=0}^{h-1} r^i &= \frac{r^h - 1}{r - 1} \\ &= \frac{\left(\frac{7}{4}\right)^h - 1}{\frac{7}{4} - 1} = \frac{\left(\frac{7}{4}\right)^{\log_2 n} - 1}{3/4}. \end{aligned}$$

Now $\left(\frac{7}{4}\right)^{\log_2 n} = n^{\log_2(7/4)} = n^{\log_2 7 - 2}$. Multiply this by the prefactor n^2 to obtain

$$n^2\left(\tfrac{7}{4}\right)^{\log_2 n} = n^2 \cdot n^{\log_2 7 - 2} = n^{\log_2 7}.$$

Hence the sum contribution simplifies to

$$\beta n^2 \sum_{i=0}^{h-1}\left(\frac{7}{4}\right)^i = \beta \cdot \frac{4}{3}\Big(n^{\log_2 7} - n^2\Big).$$

Putting the two parts together gives the closed-form expression

$$T(n) \;=\; n^{\log_2 7}\,T_0 \;+\; \beta \cdot \frac{4}{3}\,n^{\log_2 7} \;-\; \beta \cdot \frac{4}{3}\,n^2.$$

Combine the $n^{\log_2 7}$ terms:

$$T(n) \;=\; \Big(T_0 + \tfrac{4\beta}{3}\Big)\,n^{\log_2 7} \;-\; \tfrac{4\beta}{3}\,n^2.$$

Leading term and limit. Since $\log_2 7 \approx 2.807 > 2$, the $n^{\log_2 7}$ term dominates the n^2 term as $n \to \infty$. Define the positive constant

$$C := T_0 + \frac{4\beta}{3} > 0.$$

Then

$$T(n) = C\,n^{\log_2 7} \;-\; \frac{4\beta}{3}\,n^2,$$

and therefore

$$\lim_{n\to\infty} \frac{T(n)}{n^{\log_2 7}} \;=\; C \quad \text{(a finite, positive constant)}.$$

From this limit we immediately obtain the exact asymptotic class:

$$T(n) = \Theta\big(n^{\log_2 7}\big).$$

2.2.3 Upper and lower bounds by induction

To make the Θ claim fully rigorous, one could also produce explicit O and Ω bounds by induction.

Upper bound ($T(n) = O(n^{\log_2 7})$). We prove $\exists\, A > 0,\ n_0$ such that $T(n) \le An^{\log_2 7}$ for all $n \ge n_0$. Choose $A \ge \max\{T_0,\ \frac{4\beta}{3} + T_0\}$ (this is one convenient choice). For $n = 1$ the inequality holds by choice of A. Assume it holds for all smaller powers of two; then

$$T(n) = 7T\Big(\frac{n}{2}\Big) + \beta n^2 \le 7A\left(\frac{n}{2}\right)^{\log_2 7} + \beta n^2 = An^{\log_2 7} + \beta n^2.$$

Since $\log_2 7 > 2$, for n large enough the βn^2 term is bounded by $(A - A)n^{\log_2 7}$ (with our concrete choice of A we can make the inequality hold for all $n \ge n_0$). Thus the inductive step completes and $T(n) \le An^{\log_2 7}$ for large n.

Lower bound ($T(n) = \Omega(n^{\log_2 7})$). Pick $B > 0$ small enough (e.g., $0 < B \le T_0$). Using induction one shows $T(n) \ge Bn^{\log_2 7}$ for all sufficiently large n because the recurrence's multiplicative factor 7 at each halving drives growth at rate $n^{\log_2 7}$ and the additive βn^2 cannot cancel that growth for large n. This again yields $T(n) \ge Bn^{\log_2 7}$ for all $n \ge n_1$.

Combining the two bounds gives $T(n) = \Theta(n^{\log_2 7})$.

2.2.4 Relations among the asymptotic classes

From $T(n) = \Theta(n^{\log_2 7})$ and since $n^2 = o(n^{\log_2 7})$ (because $2 < \log_2 7$), we obtain the following precise comparisons:

$$\begin{aligned}
&n^2 = o\big(n^{\log_2 7}\big), \qquad \text{so} \quad \beta n^2 = o\big(n^{\log_2 7}\big).\\
&T(n) = \Theta\big(n^{\log_2 7}\big), \quad \text{so} \quad T(n) = O\big(n^{\log_2 7}\big) \text{ and } T(n) = \Omega\big(n^{\log_2 7}\big).\\
&T(n) = \omega(n^2), \quad \text{and} \quad n^2 = o\big(T(n)\big).
\end{aligned}$$

Note that, because $T(n)$ is $\Theta(n^{\log_2 7})$, it is *not* the case that $T(n) = o\big(n^{\log_2 7}\big)$ or $T(n) = \omega\big(n^{\log_2 7}\big)$; rather $T(n)$ grows on the same scale as $n^{\log_2 7}$.

Hypothesis 3. *The execution time of a pure Strassen's algorithm implementation scales as*

$$T(n) = \Theta\big(n^{\log_2 7}\big) \approx \Theta\big(n^{2.807}\big),$$

and in particular $T(n) = \omega(n^2)$ (so Strassen is asymptotically faster than any n^2-time method but slower than any n^d for $d > \log_2 7$).

2.2.5 Remarks on the Master Theorem viewpoint

The Master Theorem classifies recurrences of the form $T(n) = aT(n/b) + f(n)$ by comparing $f(n)$ to $n^{\log_b a}$. Here $a = 7$, $b = 2$, and $f(n) = \Theta(n^2)$. Since $f(n) = o\big(n^{\log_2 7}\big)$ (because $2 < \log_2 7$), the recurrence falls under Case 1 of the Master Theorem and the solution is $T(n) = \Theta(n^{\log_2 7})$. The unrolled closed-form derivation above provides the explicit constant and therefore a fully rigorous justification beyond the shorthand Master-Theorem statement.

2.2.6 Space complexity: rigorous derivation

Model the auxiliary (temporary) space used by a naive recursive Strassen implementation. Let $S(n)$ be the extra space used (not counting the input matrices kept elsewhere). At one level we create k temporary matrices of size $(n/2) \times (n/2)$ (for some fixed integer k determined by the implementation), and we also need whatever space is used by a single recursive call on size $n/2$. A reasonable recurrence is

$$S(n) \;=\; S\Big(\frac{n}{2}\Big) \;+\; k\left(\frac{n}{2}\right)^2 = S\Big(\frac{n}{2}\Big) \;+\; \frac{k}{4}\,n^2.$$

Unroll for $h = \log_2 n$ levels:

$$S(n) \;=\; \frac{k}{4}n^2 \sum_{i=0}^{h-1} \left(\frac{1}{4}\right)^i = \frac{k}{4}n^2 \cdot \frac{1-(1/4)^h}{1-1/4} = \frac{k}{4}n^2 \cdot \frac{1-n^{-2}}{3/4}.$$

Simplify:

$$S(n) \;=\; \frac{k}{3}\,n^2\left(1-n^{-2}\right) \;=\; \frac{k}{3}\,n^2 - \frac{k}{3}.$$

Thus

$$S(n) = \Theta(n^2).$$

Consequences in asymptotic notation:

$$S(n) = O(n^2), \qquad S(n) = \Omega(n^2), \qquad S(n) = \Theta(n^2).$$

Additionally, for any $\varepsilon > 0$, $S(n) = o(n^{2+\varepsilon})$ and $S(n) = \omega(n^{2-\varepsilon})$.

Hypothesis 4. *The auxiliary memory usage of a straightforward recursive Strassen implementation is quadratic:*

$$S(n) = \Theta(n^2),$$

with a larger constant factor than the naive algorithm (the constant here equals $k/3$ under the model above), so empirically Strassen will consume noticeably more memory for the same n even though the asymptotic class matches the naive algorithm.

3 Research Methodology

To empirically test our hypotheses, we propose a rigorous methodology designed to measure the performance of matrix multiplication across a diverse set of programming languages, thereby isolating the impact of their respective execution models.

3.1 Implementation Strategy

The core of our methodology is the use of "pure" language implementations. For each target language, both the naive and Strassen's algorithms will be implemented using only standard language features: primitive data types, multi-dimensional arrays (or equivalent structures like vectors of vectors), and iterative loops or recursion. We will explicitly forbid the use of third-party numerical libraries (e.g., NumPy, Eigen, BLAS) or language-specific high-performance computing extensions, so that the algorithms are fully standardized, hence providing accurate findings and reducing confounding variables.

Justification: This strategy is critical for addressing our research question. Optimized libraries often use hand-tuned assembly, vectorization (SIMD instructions), and parallelization, which would obscure the performance characteristics of the language itself (Luo et al.). By enforcing a pure implementation, we create a controlled environment where performance differences can be more directly attributed to factors like compiler optimization quality, JIT compilation efficiency, garbage collection overhead, and the cost of fundamental operations within the language's runtime (Lin et al.).

Target Languages: The study will be conducted on five languages chosen to represent a wide range of execution models:

- **C++:** Compiled to native code, offering low-level memory control and high computational capability, providing low abstraction. Represents a performance baseline.
- **Rust:** Compiled to native code, with a strong focus on memory safety through its ownership and borrow checking system. Provides a modern comparison to C++ by offering similar computational capability while providing safer abstractions.
- **Java:** Compiled to bytecode and executed on the JVM with a JIT compiler and automatic garbage collection.
- **Python:** Primarily an interpreted, dynamically-typed language (using the CPython implementation) with automatic garbage collection. Serves as a baseline for high-level scripting languages, typically demonstrating slower execution times while providing high abstraction and a user-friendly development experience.
- **JavaScript:** Executed within the Node.js environment (V8 engine), featuring an advanced JIT compiler and garbage collection. While similar to Java but with different language semantics, it is primarily applied in backend web development, rather than computationally intensive tasks.

3.2 Experimental Setup

Experiments will be conducted on a single, dedicated machine to ensure consistency and eliminate variance from hardware differences.

- **Hardware:** A modern x86-64 laptop with a multi-core CPU (AMD Ryzen 7) and 16 GB of RAM to accommodate large matrices. Caches will be cleared between runs where possible.

- **Software:** A standardized Linux distribution (e.g., Ubuntu 22.04 LTS). Specific versions of compilers and runtimes will be documented (e.g., GCC 11, Cargo 1.92, Java 1.8, Python 3.10, Node.js 18). Compiler optimization flags will be set to a standard production level (e.g., `-O3` for C++/Rust).
- **Data:** Matrices will be square, with dimensions $n = 2^k$ for $k \in \{6, 7, \ldots, 12\}$ (i.e., sizes from 64x64 to 1024x1024). Using powers of two simplifies the implementation of Strassen's algorithm. Matrix elements will be populated with random 64-bit floating-point numbers.

3.3 Performance Measurement

To ensure accurate and reliable measurements, the following protocol will be observed:

1. **Warm-up Phase:** For JIT-compiled languages (Java, JavaScript), each program will execute the multiplication on a medium-sized matrix (e.g., 256x256) for several iterations (e.g., 10) without timing. This initial execution allows the JIT compiler to perform initial compilation and optimization of the "hot" code paths (reducing noise and providing additional standardization)(Lin et al.).
2. **Timing:** High-resolution monotonic clocks will be used in each language (e.g., Java `System.nanoTime()`, Python `time.perf_counter()`, Node.js `process.hrtime.bigint()`). Timing will only enclose the matrix multiplication function call, excluding matrix creation and initialization.
3. **Statistical Rigor:** For each matrix size n and each algorithm/language combination, the timed execution will be repeated at least 30 times. The mean, median, and standard deviation of the execution times will be recorded. This process mitigates the impact of system noise and garbage collection pauses.
4. **Memory Measurement:** Memory usage will be profiled using OS-level tools (e.g., `/usr/bin/time -v`) to measure the peak resident set size (RSS) of the process.

3.4 Variables

In this study, the following variables are defined to operationalize the relationship between programming language choice and computational efficiency:

Independent Variable: The programming language and algorithm used for the matrix multiplication implementation. Five categorical levels are considered: C++, Rust, Java, Python, and JavaScript. This variable represents differences in compilation model, memory management, and runtime execution strategies.

Dependent Variables:

1. **Execution Time** — The total time (in nanoseconds for extra precision) required to complete the matrix multiplication for varying matrix dimensions n.
2. **Memory Usage** — The peak resident memory consumption during execution, measured in megabytes.
3. **Scaling Behavior** — The empirical exponent p in the observed relation $T(n) \propto n^p$, derived from log–log regression of runtime data.

Controlled Variables: Hardware configuration, operating system, compiler/runtime version, and algorithmic implementation (naive or Strassen) are held constant to isolate the effect of the independent variable.

4 Expected Results and Hypotheses

This section outlines the anticipated outcomes of the experimental methodology and links them to the theoretical hypotheses, providing a deeper analysis of the expected language-specific behaviors.

4.1 H1 & H2 Validation: Algorithmic Scaling

We expect the empirical data to strongly support the theoretical time complexity scaling laws. By plotting the median execution time versus matrix size n on a log-log graph, we anticipate observing linear trends. The slope of the regression line for the naive algorithm should be approximately 3.0, while the slope for Strassen's algorithm should be approximately 2.807. Minor deviations may occur due to cache effects or language-specific overheads, but the overall trend should be clear.

4.2 H3: Language Performance Hierarchy

Based on the execution models of the selected languages, we hypothesize a distinct performance hierarchy for the absolute wall-clock time required to perform the multiplications.

- **Tier 1 (Highest Performance): C++ and Rust.** We expect C++ and Rust to yield the fastest execution times. Their AOT compilation to optimized machine code, lack of a managed runtime, and control over memory layout minimize overhead. We predict C++ may have a slight edge due to mature compilers, but Rust's performance should be very close, demonstrating that memory safety can be achieved with minimal performance cost.
- **Tier 2 (Intermediate Performance): Java and JavaScript (Node.js).** After the warm-up phase, we expect Java and JavaScript to be significantly faster than Python but slower than C++ and Rust. The performance of their JIT compilers is remarkable but may not fully match the static optimizations of AOT compilers for this type of numerical code. Furthermore, unpredictable pauses from their respective garbage collectors may increase the variance in measured times.
- **Tier 3 (Lowest Performance): Python.** We hypothesize that pure Python will be several orders of magnitude slower than the compiled languages. The overhead of interpreting each operation in a tight numerical loop, dynamic typing, and object model will result in poor performance. This result will underscore why the scientific computing ecosystem in Python relies heavily on native-code libraries.

4.3 H4: Strassen's Algorithm Crossover Point

Strassen's algorithm introduces overhead from recursion, increased memory traffic, and more complex logic. The naive algorithm, with its simple triple-loop structure, is highly amenable to compiler optimizations and CPU cache prefetching. Consequently, Strassen's algorithm is only faster for matrices larger than a certain crossover point, n_c. We hypothesize that this crossover point will vary significantly between languages.

- We expect n_c to be **lowest in C++ and Rust**. In these languages, function call overhead is minimal, and the compiler can efficiently manage the data for recursive calls.
- We expect n_c to be **intermediate in Java and JavaScript**. The JIT can optimize recursive calls, but object creation for sub-matrices might introduce overhead.

- We expect n_c to be **highest in Python**, or potentially not observed within our tested range. The high cost of function calls, recursion depth limits, and slow data manipulation in pure Python will create substantial overhead, pushing the crossover point to very large matrix sizes.

5 Conclusion and Future Work

This paper has proposed a structured research plan to dissect the performance of matrix multiplication, a critical computational kernel, across a representative set of modern programming languages. By grounding our investigation in the first-principles derivation of algorithmic complexity for both the naive and Strassen's methods, we have established a solid theoretical baseline. The outlined experimental methodology, which emphasizes pure language implementations and statistical rigor, is designed to generate empirical evidence that can be directly mapped back to these theoretical models.

The expected results aim to quantify the performance hierarchy dictated by different language execution models—from AOT-compiled native code to JIT-compiled managed runtimes and pure interpretation. By analyzing both algorithmic scaling and the practical crossover point for Strassen's algorithm, this research will provide valuable insights into the real-world cost of abstraction in programming languages for high-performance computing tasks.

Future work would extend this framework in several directions. We will need to prepare a standardized environment as modeled. This research paper has so far proposed hypotheses and data for the experiment, thus the natural next step is to execute the experiment.

6 Works Cited

Colburn, Timothy R., and Gary M. Shute. "Abstraction in Computer Science". *Minds & Machines*, vol. 17, no. 2, 2007, pp. 169–84. DOI: https://doi.org/10.1007/s11023-007-9061-7.

Khan, A. U. H., et al. "Optimizing the Matrix Multiplication Using Strassen and Winograd Algorithms with Limited Recursions on Many-Core". *International Journal of Parallel Programming*, vol. 45, no. 1, 2015, pp. 125–54. DOI: https://doi.org/10.1007/s10766-015-0378-1.

Lin, L., et al. "An Empirical Comparison of Implementation Efficiency of Iterative and Recursive Algorithms of Fast Fourier Transform". *Machine Learning and Intelligent Communications*. Springer, 2021, pp. 73–81, DOI: https://doi.org/10.1007/978-3-030-66785-6_9.

Luo, S., et al. "Automatic Optimization of Matrix Implementations for Distributed Machine Learning and Linear Algebra". *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21)*. ACM, 2021, DOI: https://doi.org/10.1145/3448016.3457317.