HPC Project 1: Optimization of the Matrix Product

Addison Olstad, Joe Ruse, Reyhaneh Bahranifard

February 4, 2026

Contents

1 Introduction

This project serves as an introduction to the basic concepts behind high performance computing (HPC) using the relatively simple task of computing the product of two square matrices. We discuss fundamental ideas such as order of complexity and performance metrics, and explore the roles that programming language, compiler choice, and optimizers play in a program's performance.

High performance computing is a highly interdisciplinary field that involves mathematics, network administration, computer architecture, and more. In simple terms, HPC is about managing the computational cost of a program. For every program, there is a price to be paid in three forms: time, space, and energy. The time-cost is simply the time taken for the program to complete its task, the spatial-cost of a program refers to the physical space used in the computer's memory by a program, and the energy-cost of a program is the electrical power consumed by the computer.

The importance of memory and energy cost has waxed and waned over the years as improvements to hardware and infrastructure have made both resources abundantly available for most small-scale projects. However, these resources have become especially relevant in recent times as the demands of large-scale projects like AI data centers have grown to a level that they impact entire electrical grids.

The time-cost, or wall time, tends to be the first performance metric to be considered when evaluating a program. It is the only cost out of the three that a user directly experiences alongside the computer; time spent waiting is the cost paid. While it is the simplest to understand, it is also unique in how it is managed. Unlike memory and energy usage, the time-cost cannot be balanced by simply making more of the resource available. Instead, it must be balanced by performing computations faster. For a long time, improvements to CPU clock speeds were the answer to this problem. As clock speeds have plateaued, the solution has shifted from doing one thing faster to doing many things at once, resulting in parallelization becoming synonymous with HPC.

This project focuses on the wall time as a performance metric for completing the matrix multiplication. To compare the performance between different programming languages, a naive three-loop algorithm was written in C, Fortran, and Python and the wall times were recorded at various matrix sizes. Additionally, we explore the impact of compiler choice and optimization level for the compiled languages C and Fortran. Finally, we compare the performance of the three-loop algorithm against built-in functions from libraries like BLAS, LAPACK, and NumPy.

1.1 Matrix Multiplication Algorithm

For a matrix product $C = AB$, where A and B are square $n \times n$ matrices, the definition of matrix multiplication states tells us that each i, j entry of C is the dot product of the i^{th} row of A and the j^{th} column of B. Since a vector dot product is simply the sum of the entry-wise products of the vectors, we can compute the entries of C as

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

A simple approach to computing the product $C = AB$ is a three-loop algorithm that iterates over the rows of A and the columns of B to populate the entries of C using Eq. (1).

Algorithm 1 Naive Three-Loop Iterative Algorithm

Require: $N \geq 1$ ▷ Size of the square matrices
  Input: $A_{N\times N}, B_{N\times N}$, square matrices filled with values.
  Let $C_{N\times N}$ be an empty array.
  for i in $1, N$ **do** ▷ Row Selection
    for j in $1, N$ **do** ▷ Column Selection
      $c_{ij} = 0.0$
      for k in $1, N$ **do** ▷ Dot Product Calculation
        $c_{ij} \leftarrow a_{ik} b_{kj}$
      end for
    end for
  end for

This algorithm can be thought of in two parts: row/column selection, and the dot product. The two outer loops iterate over the entries to be calculated, and the innermost loop calculates the value that belongs in each entry.

1.2 Time Complexity

As N increases, the number of floating-point operations (FLOPs) required to perform the multiplication increases as well. Assuming each FLOP takes approximately the same amount of time, we can use the rate at which FLOP count grows with N to determine how the wall time will increase with N. This is known as time complexity.

To determine the number of FLOPs required to perform our matrix multiplication, we start by counting the number that occur in a single iteration of the three-loop. Looking back to algorithm 1, we can see that the only floating-point operations happen in the line $c_{ij} \leftarrow a_{ik} b_{kj}$, where the product of $a_{ik} b_{kj}$ is added to the current value of c_{ij}. So a single iteration performs two floating-point operations.

To get the total number of FLOPs, we must now count the number of iterations in which they are performed. Starting with the innermost loop where k increases from 1 to N, this loop will perform

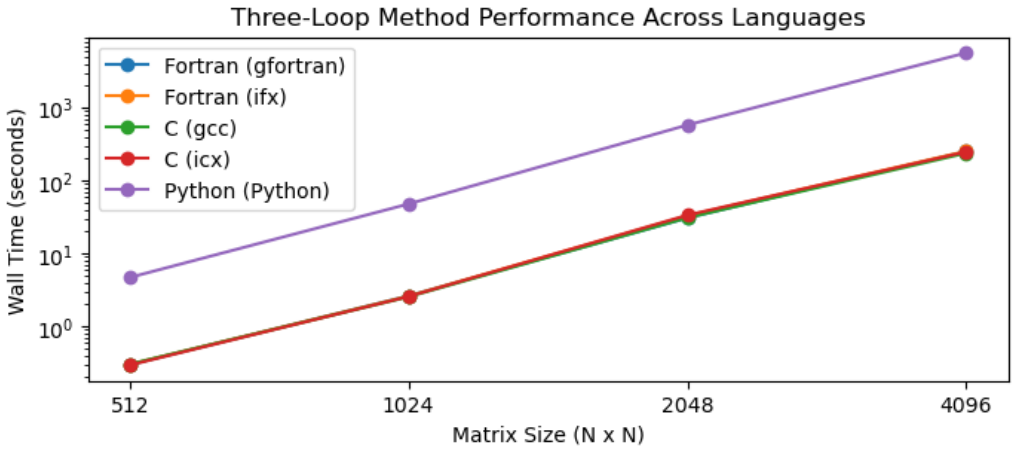

Figure 1: Wall time vs. Matrix Size for each language and compiler. C and Fortran are compiled without optimization ($-O0$).

the two FLOPs each iteration, meaning the total number of FLOPs for the k-loop is

$$\text{FLOP}_k = \sum_{k=1}^{N} 2 = 2N.$$

Repeating the idea used to determine FLOP_k to calculate the total FLOP count, we can calculate the total number of FLOPs as

$$\text{FLOP}_{total} = \sum_{i=1}^{N}\sum_{j=1}^{N}\sum_{k=1}^{N} 2 = NNN2 = 2N^3.$$

Thus, the number of floating-point operations, and therefore the wall time, increases as the cube of the matrix size N. So we say that this algorithm has a "Big-O" of $O(N^3)$, or the walltime w is

$$w(N) = O(N^3).$$

2 Results and Analysis

This section presents a comparison between wall times of different programming languages, compilers, and optimization levels using the matrix multiplication experiment. To compare the raw performance of each language, we measured the wall time of the naive three-loop matrix multiplication algorithm to increase the values of N. All tests were run for matrix sizes

$$N = [512, 1024, 2048, 4096].$$

$N = 512$ was chosen as the baseline because smaller matrix sizes result in wall times that are so fast that some outputs were rounding to zero.

2.1 Three-Loop Method Across Languages

To explore the role of programming language in performance, the three-loop program was written in Python, C and Fortran. In order to evaluate the raw performance of each language, C and Fortran were compiled with optimization disabled (`-O0`).

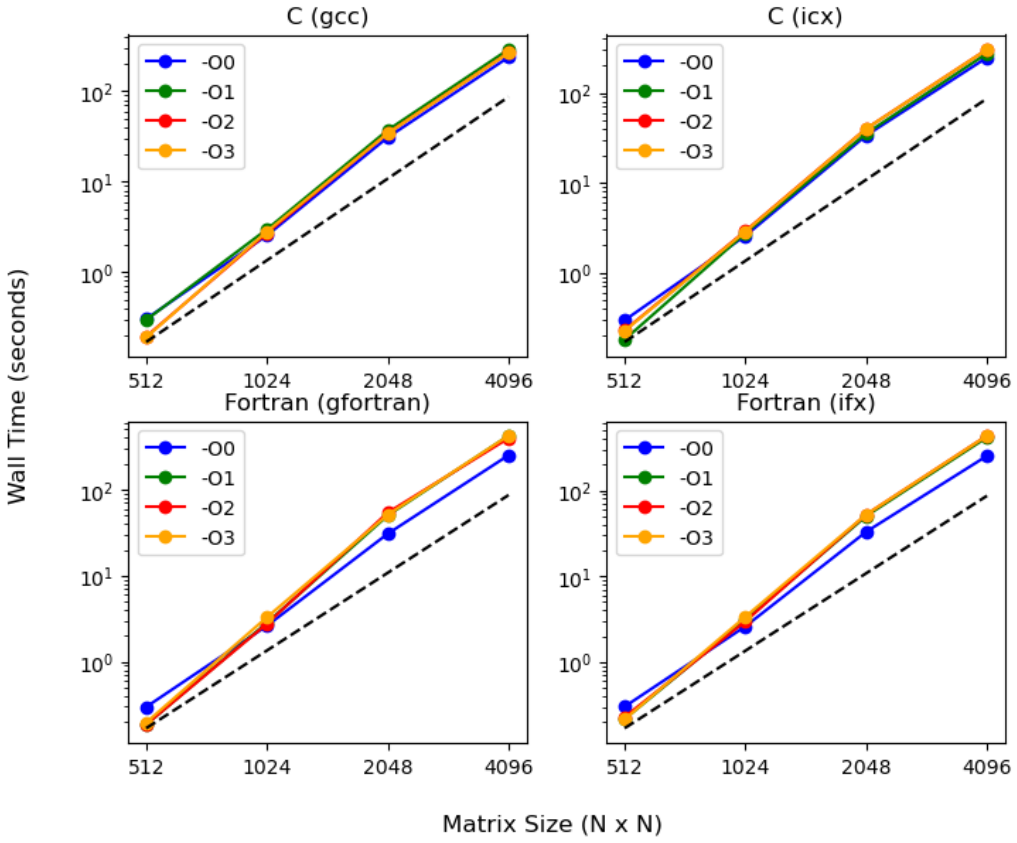

Figure 2: Wall time vs. Matrix Size at different optimization levels for C (top row) and Fortran (bottom row) using the GNU compiler (left column) and Intel compiler (right column).

In figure 1, the wall times of each language and respective compilers are compared. All three languages follow the expected $O(N^3)$ scaling, meaning the runtime grows at the same rate in all three languages, however the measured wall times differ significantly between the compiled languages and Python. Even though the same algorithm is used in all cases, Python's wall times are almost an order of magnitude higher.

Another notable result from figure 1 is that Fortran and C have almost identical performance when using either compiler. This could be due to the fact that the matrix multiplication is a relatively simple numerical operation, so with the lack of optimization it results in both languages compiling to the same assembly code. It is possible that a more complex task would highlight the differences between the two languages.

2.2 Compiler and Optimization Level

Programs written in languages like C or Fortran can be compiled with different optimization levels using the `-O` flag. Each level of optimization gives the compiler permission to use increasingly aggressive methods to improve the programs performance. While lower levels might allow the compiler to remove redundancies or minimize function calls, the higher optimization levels allow the compiler to improve the code by effectively deconstructing the program and rebuilding it as it sees fit.

The plots in figure 2 show the wall times at different optimization levels of the two compilers used for each language. While there are slight differences between the GNU and Intel compilers, neither option provides a significant boost to performance over the other. Another interesting detail to note

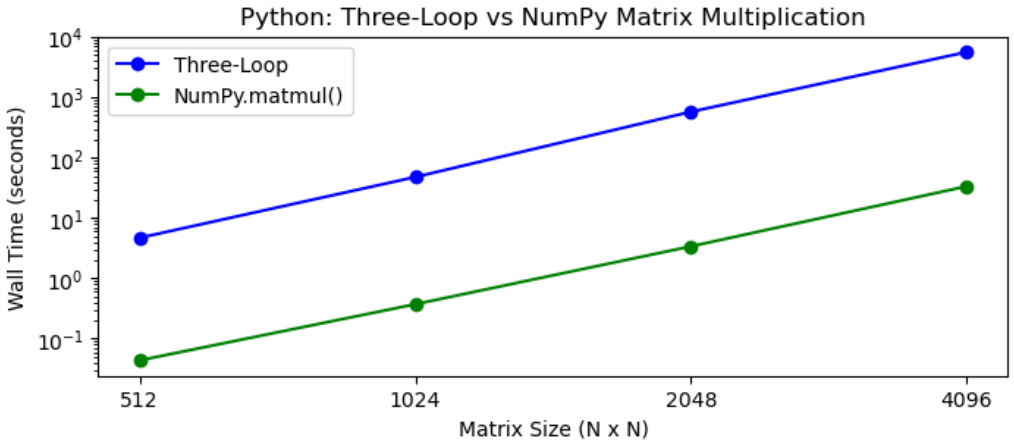

Figure 3: Matrix Multiplication wall times in Python using three-loop (blue) and using NumPy (blue).

is that the `-OO` wall times have the shortest $N = 512$ wall times for all four plots, but is overtaken by the higher `-O` levels as N increases. This suggests that the higher optimization levels create an additional overhead cost that is only beneficial at higher matrix sizes.

For the higher N values, we can see that the performance of C is fairly consistent across optimization levels with no significant boost from optimization. Fortran, on the other hand, shows a noticeable performance gain when optimization is enabled, but further increases to the optimization level provide little to no further improvement. This difference might be due to the compiler, or it might indicate that Fortran is more readily optimizable than C under the hood.

2.3 Three-Loop vs Built-In Functions

It is true that Python's performance was not ideal, but it is definitely the most popular. Fortunately, packages like `NumPy` provide optimized functions like `numpy.matmul()` which uses LAPACK in the backend to boost performance. Here, we compare `numpy.matmul()` with the three-loop method to see how much it improves Python's performance.

In figure 3, we can see that `numpy.matmul()` has provided a massive performance increase, dropping the wall times by almost two orders of magnitude. This result very clearly shows the benefit of using built-in functions in a language like Python. By offloading the work to modules written in faster language, Python can achieve speeds that rival raw C and Fortran.

2.4 Python Overhead

Early testing for the Python three-loop using small matrix sizes brought an interesting behaviour to light. In figure 4, it can be seen that for matrices of size $N < 32$, the wall time diverges from the expected N^3 scaling and appears to approach a minimum wall time of $\approx 10^{-6}$.

This effect can be attributed to the fact that Python is a high-level interpreted language. Each time a Python program is executed, it must be parsed by the interpreter and converted to bytecode, which is then executed by the machine. Additionally, Python has several quality of life features such as garbage collection and dynamic typing which make things easier on the developer's side of things at the cost of performance. The added cost from these factors means that even simple operations like addition or subtraction can take several CPU cycles to complete, resulting in an overhead time for any operation.

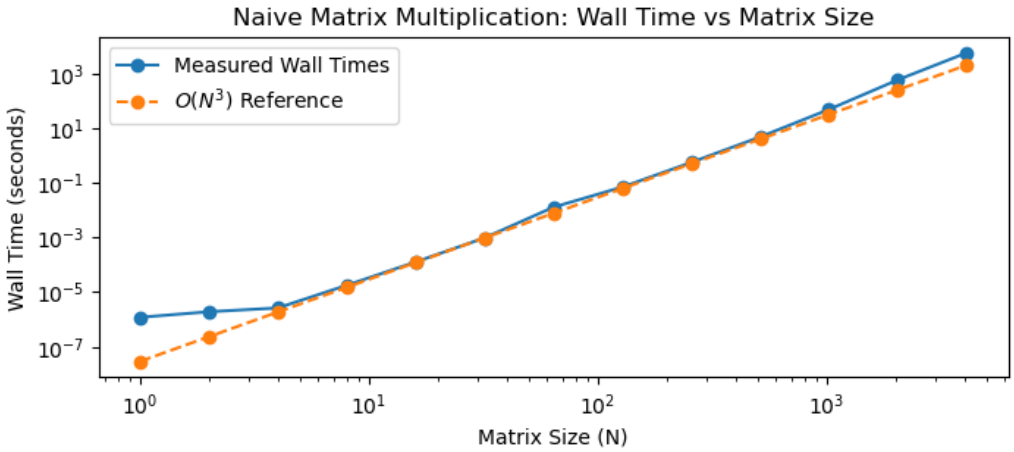

Figure 4: Wall times for python three-loop at low N.

At $N = 1$, the algorithm will go through the triple loop only once, and will perform one addition and one scalar multiplication. From figure 4, we see that the $N = 1$ wall time is $w_{N=1} \approx 10^{-6}$, meaning that the overhead of one iteration through the loop is $h_{\text{iteration}} \approx 10^{-6}$ seconds.

3 Conclusion

Compiler choice, optimization flags, and language each play a significant role in a program's performance, even if the complexity with N stays the same. From our experiments we can conclude that compiled languages like C and Fortran are much faster than interpreted ones like Python, but packages like `NumPy` can boost Python's performance to the level of these compiled languages.

The roles of compiler choice and optimization level were not as significant as expected, however they each still had noticeable effects. Changing from the GNU to Intel compiler did not notably boost performance, but a change in performance was still seen by the change in the plots. Increasing the optimization level only had a significant effect in Fortran when going from `-O0` to `-O1`, but further levels had no effect. This likely suggests that the algorithm used was not optimizable beyond basic the improvements given by `-O1`. Performing a similar analysis using a more demanding algorithm would most likely let compiler and optimization levels play a larger role in performance.