Seminar Report

Chess Engine Optimization in Python

Frederik Hennecke

MatrNr: 21765841

Supervisor: Lars Quentin

Georg-August-Universität Göttingen
Institute of Computer Science

April 11, 2025

Abstract

Python is one of the most widely used programming languages in scientific computing and data analysis, yet its performance limitations often hinder its use in compute-intensive applications. This report investigates the extent to which a Python program can be optimized using a chess engine as a case study. Chess serves as a well-understood and computationally demanding domain, offering a clear benchmark for evaluating performance. A custom chess engine was implemented from scratch in pure Python and successively optimized using various techniques and tools, including PyPy, Nuitka, Numba, Cython, and a reimplementation in C++. Benchmarking across these variants reveals trade-offs between performance gains, implementation effort, and maintainability. In addition to providing a comparative evaluation of modern Python optimization tools, this report contributes a detailed overview of their practical implications aimed at developers seeking efficient execution without fully departing from the Python environment.

Statement on the usage of ChatGPT and similar tools in the context of examinations

In this work, I have used ChatGPT or a similar AI-system as follows:

- ☐ Not at all
- ☐ In brainstorming
- ☐ In the creation of the outline
- ☐ To create individual passages, altogether to the extent of 0% of the whole text
- ☐ For proofreading
- ☑ Other, namely: Grammarly for wording, ChatGPT as coding copilot

I assure you that I have stated all uses in full.
Missing or incorrect information will be considered as an attempt to cheat.

Contents

List of Figures **iv**

List of Abbreviations **v**

1 Introduction **1**
1.1 Background and Motivation . . . 1
1.2 Objectives . . . 1
1.3 Structure . . . 2

2 Methodology **2**
2.1 Python . . . 2
2.2 Chess Engines and Computational Complexity . . . 2
2.3 Optimization Techniques Overview . . . 3

3 Chess Engine **4**
3.1 Overview . . . 4
3.2 Main Class . . . 4
3.3 Chess Board . . . 5
3.4 Move Generator . . . 6
3.5 Evaluation Function . . . 7

4 Performance Optimization Techniques **8**
4.1 Baseline - Python 3.10 . . . 8
4.2 Python 3.12 . . . 9
4.3 Pypy . . . 10
4.4 Nuitka . . . 12
4.5 Numba . . . 12
4.6 Cython . . . 13
4.6.1 Cython . . . 14
4.6.2 C++ . . . 16
4.7 Performance Expectations . . . 18

5 Performance Analysis **19**
5.1 Setup . . . 19
5.2 Evaluation Speed: Nodes per Move . . . 20
5.3 Playing Strength: Win Rates and Elo Ratings . . . 21
5.4 Interpretation . . . 23

6 Conclusion **23**
6.1 Summary of Findings . . . 23
6.2 Future Work and Outlook . . . 24

References **25**

List of Figures

1 Python versions download statistics from Pypistats[25] (February 2024). . . 9
2 Numpy download statistics from Pypistats[25] (February 2024). 9
3 Pypy architecture[27] 11
4 Numba architecture[30] 13
5 Cython architecture[33] 14
6 Number of processed Nodes in one second per engine 21
7 Wins/draws/losses of matches of the baseline engine vs the optimized engine 22
8 Elo differences for each matchup of the baseline engine vs the optimized engine 22

List of Abbreviations

API	Application Programming Interface
CPU	Central Processing Unit
FEN	Forsyth–Edwards Notation
GIL	Global Interpreter Lock
GPU	Graphics Processing Unit
JIT	Just-In-Time
MRO	Method Resolution Order
PEP	Python Enhancement Proposals
RAM	Random-Access Memory
UCI	Universal Chess Interface

1 Introduction

1.1 Background and Motivation

Python has become the de facto language for scientific computing, data analysis, and machine learning[1]. Its simplicity, readability, and extensive ecosystem have made it the go-to choice for researchers, students, and developers across disciplines. However, Python's main downside is its performance. As an interpreted, dynamically typed language, Python suffers from significant overhead in execution speed and memory management. While this is often tolerable in high-level scripting or prototyping, it becomes a major bottleneck in compute-intensive tasks.

One of the best ways to study and stress-test computational performance is through a domain that is highly demanding. Chess fits this role perfectly. Chess engines involve heavy computation: move-tree search, heuristic evaluation, and decision-making under strict time constraints. While the fastest chess engines worldwide are written in C++, Rust, or similar low-level languages, this project starts from a Python implementation. Why?

Because in many real-world applications, especially in academia and data science, switching to C++ is not always an option. The people writing simulation code, training machine learning models, or evaluating experiments often only know Python. They don't just write toy code: many research papers, experiments, and production pipelines hinge on Python code that could benefit from performance improvements if they are accessible.

The goal of this project is not to write the fastest chess engine. The goal is to explore how far we can push Python using modern optimization strategies and to assess how much speed-up is possible — and at what cost in complexity. We use a Python-based chess engine as a case study: a computationally expensive but conceptually manageable program representative of many real-world research scripts. By benchmarking various optimization tools, including JIT compilers, static compilers, and even a full C++ rewrite, we aim to understand the trade-offs between speed and development effort.

1.2 Objectives

This study focuses on evaluating the performance and usability of several Python optimization tools through the lens of a chess engine. Specifically, it aims to:

- Create a baseline chess engine using Python.
- Use various performance optimization tools, including Just-In-Time (JIT) compilation, static compilation, and alternative interpreters.
- Measure and analyze the impact of each optimization on execution speed and move evaluation efficiency.
- Assess trade-offs between ease of implementation and performance improvement.

By doing these evaluations, this study provides us insight into how well Python can be optimized with different tools.

1.3 Structure

The remainder of this report is structured as follows. Section 2 introduces the report's key concepts, such as Python, optimization techniques, and chess concepts. Section 3 presents the architecture of the chess engine, detailing its core components such as move generation, board representation, and evaluation. In Section 4, we discuss different Python performance optimization tools, explaining their implementation and their ease of use. Section 5 presents and analyzes the benchmark results, comparing execution speed across implementations. Finally, Section 6 concludes the report by summarizing the findings.

2 Methodology

2.1 Python

Python is widely used for its simplicity and ease of development, making it a popular choice for prototyping and research-oriented applications. However, its design as an interpreted, dynamically typed, and especially interpreted language introduces significant performance limitations.

One more bottleneck in Python is the Global Interpreter Lock (GIL). The GIL prevents multiple native threads from executing Python bytecode simultaneously, effectively limiting Python's ability to use multi-core processors for parallel execution. This makes computationally intensive tasks, such as searching large game trees in chess engines, less efficient compared to performance-oriented languages like C++.

Another bottleneck is dynamic typing. Unlike statically typed languages, Python variables can hold any data type, requiring the interpreter to maintain additional metadata to track and handle these types during execution. This results in slowed execution speed, as the interpreter necessitates boxing, where primitive types are wrapped in objects to standardize type handling, further adding overhead by requiring additional memory and processing for object management[2]. Furthermore, Python's Method Resolution Order (MRO) imposes additional performance costs in dynamic inheritance scenarios. MRO determines the sequence in which base classes are traversed when looking up attributes and methods, which can be computationally intensive if the class hierarchy is deep or complex[3].

The biggest bottleneck is that Python is an interpreted language. "Interpreted languages must be parsed, interpreted, and executed each time the program is run, thereby greatly adding to the cost of running the program. For this reason, interpreted programs are usually less efficient than compiled programs"[4].

Python also uses automatic memory management, including reference counting and garbage collection. While these features simplify programming, they introduce unpredictable performance overhead due to periodic garbage collection pauses, which can disrupt real-time applications like chess engines.

2.2 Chess Engines and Computational Complexity

A chess engine is a software program that analyzes chess positions and determines the best moves based on predefined evaluation criteria. The complexity of chess arises from its vast search space, exemplified by Shannon's number (approximately 10^{120}), Claude

Shannon's lower-bound estimate of the game-tree complexity of chess[5]; each move leads to exponentially growing possibilities, making brute-force computation infeasible beyond a few moves ahead.

Modern chess engines rely on tree search algorithms to navigate this complexity. One of the most common approaches is the Minimax algorithm[6], which simulates all possible moves and counter-moves up to a certain depth. As chess is a perfect information game[7], perfect competition[8] is assumed.

To improve efficiency, engines employ Alpha-Beta Pruning[9], which eliminates branches of the search tree that cannot affect the final decision. This significantly reduces the number of positions evaluated, making deeper searches possible within the same time constraints. The evaluation function is another critical component of a chess engine. It assigns numerical values to different board positions based on material advantage, piece activity, pawn structure, and other heuristics. By optimizing this function, the engine can make more accurate decisions while reducing computation time.

Chess engines must balance search depth and execution speed to achieve competitive performance. Traditional engines written in C or C++ are highly optimized to evaluate millions of positions per second, but Python-based implementations face challenges due to Python's execution overhead.

The most powerful modern chess engines use a combination of traditional search techniques and advanced machine learning. Some of the leading chess engines today include:

- Stockfish: One of the strongest open-source chess engines, Stockfish employs an extremely optimized alpha-beta search with advanced heuristics and bitboard representations for efficiency. It can evaluate millions of positions per second and is the dominant engine in competitive chess computing[10].
- AlphaZero: Developed by DeepMind, AlphaZero uses deep reinforcement learning to teach itself chess from scratch. Unlike traditional engines, it does not rely on human-designed heuristics but instead learns optimal strategies through self-play[11].
- Lc0 (Leela Chess Zero): Inspired by AlphaZero, Lc0 is an open-source neural-network-based engine that continuously improves by training on self-play games. It provides a strong alternative to engines like Stockfish, especially in complex positional play[12].

2.3 Optimization Techniques Overview

Given Python's performance constraints, several optimization techniques exist to improve the runtime performance:

- Alternative Interpreters: PyPy[13] provides JIT compilation to speed up Python execution without requiring code modifications.
- Ahead-of-time Compilation: Nuitka[14] and Cython[15] translate Python code into C, enabling more efficient execution.
- JIT Compilation: Numba[16] compiles Python functions into machine code at runtime, optimizing numerical computations.
- C++ Integration: Rewriting small performance-critical components in C++ provides maximum speed benefits while maintaining a Python interface for usability.

Each technique has its trade-offs in terms of performance gain, ease of implementation, and compatibility with existing Python code. The subsequent sections will analyze their impact on the chess engine's efficiency.

3 Chess Engine

3.1 Overview

The chess engine created for this project is split into four components. The main components are:

- Main Class: This class sets up all needed objects and manages the communication via the Universal Chess Interface (UCI)[17], allowing the engine to interact with other bots or external chess programs.
- Chess Board: Handles the representation of the board and chess pieces. The core functionalities include generating legal moves and determining whether a piece is under attack.
- Move Generator: Responsible for computing the best possible move using a pruning Minimax algorithm. It evaluates different board states by exploring possible moves and counter-moves.
- Evaluation Function: Assigns a score to different board states, helping the move generator decide the best move based on predefined heuristics.

For this project, we are using the NumPy library[18] for faster computation and to simulate real-world workloads. At the beginning of the project, we also used the chess library[19] for the board simulation, but we later implemented it ourselves due to some problems, which are described in subsection 4.5.

3.2 Main Class

The main class serves as the core of the chess engine. It is responsible for managing the game state, processing UCI commands, and responding with the appropriate moves.

For the communication protocol, we had the choice between UCI and winboard-/xboard[20]. We chose UCI because the protocol is newer, more widely supported, and stateless. Our engine does not support the complete UCI protocol, but the most basic commands are implemented. A UCI-compliant chess engine must support several key commands, including:

- `uci` : Signals that the engine is using the UCI protocol and provides metadata such as the engine's name and author.
- `setoption` : Options to send to the engine when the user wants to change the internal parameters of the engine. In our case, it is used to set different chess variants.
- `isready` : Used to check if the engine is ready for commands.

- `ucinewgame`: Notifies the engine to reset its state for a new game.
- `position`: Sets up the board position, either from the standard starting position or from a given FEN string.
- `go`: Instructs the engine to compute and return the best move.
- `quit`: Terminates the engine.

Upon initialization, the engine sets up the chessboard and processes the command-line arguments for engine customization. The class then listens for user input and processes the given UCI commands. When the engine receives the go command, it triggers the move generator, which uses the Minimax algorithm to determine the best move. The computed move is then returned in UCI notation (e.g., e2e4 for moving a pawn from e2 to e4).

3.3 Chess Board

The chessboard is used for modeling the game state and orchestrating move mechanics. The chessboard is stored in a 8×8 Numpy array, where each element encodes both piece type and color. Positive values denote white pieces, while negative values represent black counterparts.

```
-4, -2, -3, -5, -6, -3, -2, -4
-1, -1, -1, -1, -1, -1, -1, -1
0,  0,  0,  0,  0,  0,  0,  0
0,  0,  0,  0,  0,  0,  0,  0
0,  0,  0,  0,  0,  0,  0,  0
0,  0,  0,  0,  0,  0,  0,  0
1,  1,  1,  1,  1,  1,  1,  1
4,  2,  3,  5,  6,  3,  2,  4
```

Listing 1: A starting chessboard as decoded in our project.

The main features of this class are the following:

- Move Execution: The chess board class translates UCI formatted moves into coordinate transitions, handling edge cases such as castling and promotions.
- Move Validation: Legal move generation iterates over all squares, producing pseudo-legal moves for each piece. This method verifies move validity by simulating board states and ensuring the king remains unexposed to check.
- Forsyth–Edwards Notation (FEN) Compliance: The chess board class parses FEN[21] strings, enabling arbitrary position initialization. Many external chess programs use this notation.
- State Reversion: A Last-In-First-Out queue (event) tracks move history, allowing undo via a pop operation, which restores prior board configurations and captured pieces. This is used to backtrack the board so that finding further moves is possible.

3.4 Move Generator

The move generator is responsible for selecting the best chess move within the given time. In this engine, minimax with alpha-beta pruning is used to evaluate different moves. Additionally, iterative deepening is used to refine move selection within a given time constraint.

The main part of the move generator is the minimax algorithm, which evaluates future board states by considering all possible moves and counter-moves. The algorithm assumes that the opponent will always play optimally, meaning it tries to maximize the engine's advantage while assuming the opponent will minimize it. It stops evaluating a move when at least one possibility has been found that proves the move to be worse than a previously examined move. These moves will not be evaluated further[22]. Given a position P, the engine assigns a value

$$V(P) = \begin{cases} \max_{m \in M(P)} V(P_m) & \text{if it is the engine's turn} \\ \min_{m \in M(P)} V(P_m) & \text{if it is the opponent's turn} \end{cases}$$

where $M(P)$ is the set of legal moves from position P, and P_m represents the resulting position after move m is played.

The minimax function recursively explores moves up to a given depth and assigns evaluation scores based on the board position. To avoid redundant computations, Alpha-Beta Pruning is used to eliminate branches that cannot influence the final decision.

Rather than searching to a fixed depth, the chess engine uses iterative deepening[23], where it progressively searches deeper until the time limit is reached. This technique provides several advantages:

- If time runs out, the engine can still return the best move found so far.
- It improves move ordering, which benefits alpha-beta pruning by reducing unnecessary calculations.
- This helps slower engines to keep a higher score due to computing immediate advantageous/disadvantageous moves.

The engine starts with a shallow depth and increases iteratively while keeping track of the best move found. If time runs out, it returns the last best move found.

Move ordering is another component of the move generation process. Rather than evaluating moves arbitrarily, the engine sorts them based on heuristic values to prioritize the most promising ones. This is done using a move evaluation function that uses material gains, piece-square table adjustments, and positional improvements. Given a move m leading to position P_m, the evaluation score is computed as

$$S(m) = Material(P_m) + Position(P_m) - (Material(P_{old}) + Position(P_{old}))$$

where the material and positional components are derived from the evaluation function. Captures and promotions are prioritized, as they often lead to immediate gains. By searching higher-value moves first, alpha-beta pruning eliminates weaker branches earlier, making the search more efficient.

3.5 Evaluation Function

The evaluation function we are using is called 'Simplified Evaluation Function'[24].

The evaluation function is responsible for assessing a chess position and assigning a numerical value that estimates the advantage or disadvantage for the player to move. It has two primary components: material evaluation, which assigns static values to pieces based on their relative strength, and positional evaluation, which adjusts the score based on piece placement using precomputed piece-square tables.

Material evaluation is determined by summing the values of all pieces on the board. Each piece type is assigned a fixed value based on its relative strength, with the standard valuation: pawns worth 100, knights 320, bishops 330, rooks 500, queens 900, and the king assigned an arbitrarily high value of 20,000 to prevent accidental checkmates. Given a position P, let $M(P)$ be the material score, defined as:

$$M(P) = \sum_i V(p_i) - \sum_j V(p_j)$$

where $V(P)$ is the value of piece p and summations iterate over all white and black pieces, respectively.

The evaluation function also uses positional factors through piece-square tables. These tables provide predefined bonuses and penalties for each piece based on its location on the board. Positional adjustments encourage central control, stable pawn structures, and optimal piece coordination. The value of a piece at a given square is determined by a lookup function, where the table values for White and Black are mirrored to account for symmetry. For a given piece p at square s, the positional score is defined by the lookup function:

$$T(p, s) = \text{piece-square table value for } p \text{ at } s$$

This ensures that knights and bishops are encouraged to move toward active squares, rooks are prioritized on open files, and pawns follow structurally sound development patterns. The total positional score is given by:

$$S(P) = \sum_i (T(p_i, s_i)) - \sum_j (T(p_j, s_j))$$

where s_i, s_j are the squares occupied by white and black pieces, respectively.

When determining a move's value, the function considers not only material gain or loss but also the positional improvement or deterioration of the moved piece. Capturing an opponent's piece contributes directly to the score based on the standard piece values, while moving a piece to a stronger square adds a bonus corresponding to the difference in piece-square table values before and after the move. In cases of pawn promotion, a bonus equivalent to the value of a queen is assigned to reflect the transformation of a minor piece into the most powerful unit on the board. Formally, the move evaluation function is:

$$V(m) = C(m) + T(p, s') - T(p, s)$$

where $C(m)$ is the value of a captured piece, $T(p, s')$ is the new position score and $T(p, s)$ is the previous position score.

By combining the material and positional evaluation scores, we get the total board evaluation of

$$Eval(P) = M(P) + S(P)$$

where $M(P)$ is the material score, and P is the positional score, both computed with respect to the player to move.

Summarized, these formulas gives us a way to evaluates both moves and the current board. Both these values are used to compute the next best move.

4 Performance Optimization Techniques

Chess engines require significant computational resources to evaluate as many positions as possible. Optimizing performance is crucial to ensure the engine can search deeper within a limited timeframe, leading to better move selection. Various optimization techniques are used in this project to try to improve the execution speed. This section explores different optimization approaches, starting with a baseline implementation in Python 3.10 and progressively incorporating alternative methods such as Python 3.12, PyPy, Nukita, Numba, and Cython.

4.1 Baseline - Python 3.10

The initial implementation of the chess engine is written in Python; specifically, Python 3.10.8 will serve as the reference point for performance comparisons. Python comes with certain performance limitations due to its dynamic typing and interpreted nature.

Despite these limitations, Python provides a rich ecosystem of libraries, such as NumPy, for efficient numerical computations. The primary performance constraints in this implementation arise from:

- Interpreted language: Being an interpreted language naturally adds a certain performance overhead, due to needing to be parsed, interpreted, and finally executed each time the program is run.
- Function call overhead: Python's dynamic nature results in increased overhead when calling functions, particularly in recursive algorithms like minimax.
- Loop execution speed: Loops in Python are generally slower compared to lower-level languages due to interpretation overhead.
- Memory management inefficiencies: Automatic garbage collection can introduce unpredictable latency spikes.

For the baseline of our benchmark, we chose Python 3.10 as this is currently one of the most used Python versions as of the writing of this report. Subsequent sections will explore enhancements and alternative implementations aimed at overcoming these limitations.

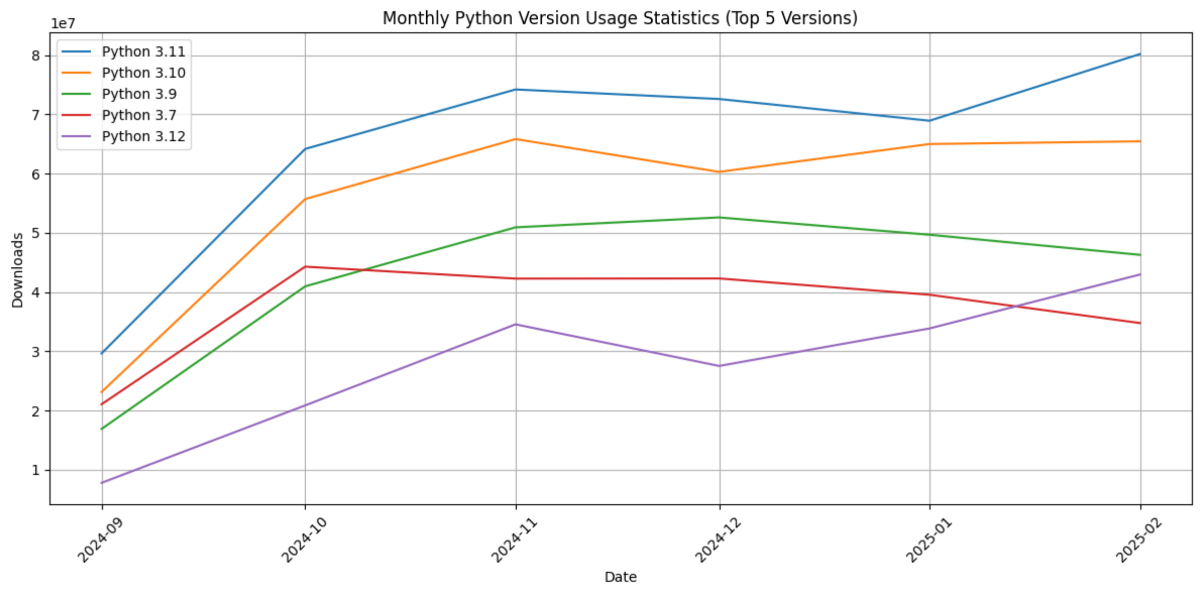

Figure 1: Python versions download statistics from Pypistats[25] (February 2024).

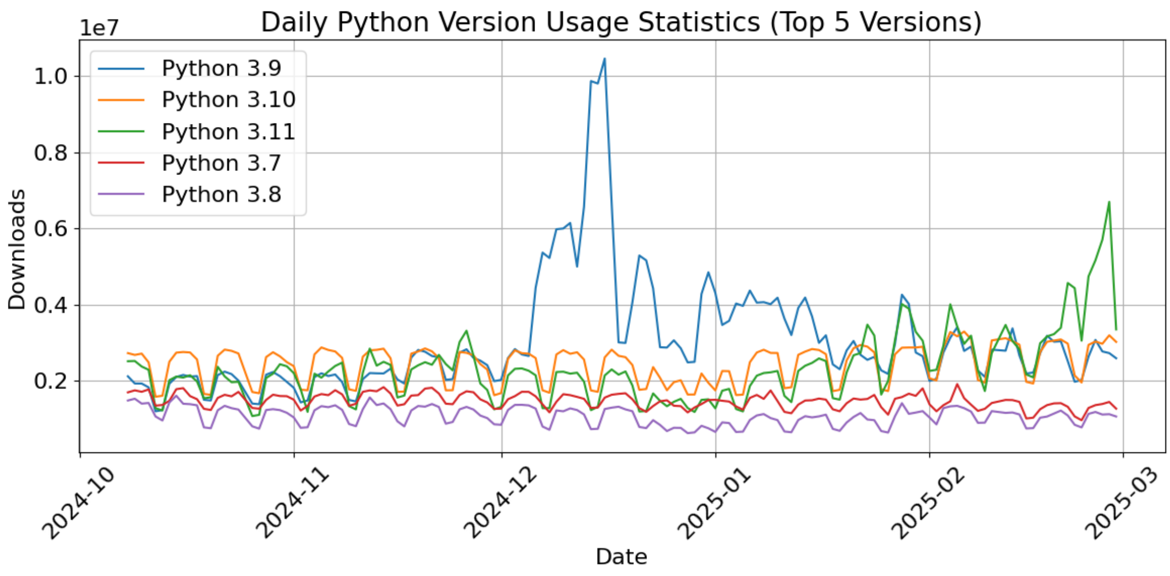

Figure 2: Numpy download statistics from Pypistats[25] (February 2024).

4.2 Python 3.12

Introduction: Python 3.12 has several improvements over Python 3.10, making it a promising upgrade for our chess engine without requiring any modifications to the code. For ease of use, we are using pyenv to switch between different versions of Python. This enables direct performance comparisons between different versions under identical conditions.

Possible Performance Improvements: Most performance improvements were done in Python 3.11[26]. However, we still chose Python 3.12 due to the increasing user base. The Python developers have made several optimizations, including:

- Better function inlining to reduce call overhead and improve recursive function performance.
- Faster attribute access and dictionary operations, which benefit internal data structures such as dictionaries and sets.
- More efficient memory management, reducing the frequency and impact of garbage collection pauses.
- Bytecode optimizations, leading to faster loop execution and improved overall efficiency.

How to port: The command to install a new Python version is:

```
pyenv install 3.12.8
```

Difficulty and Challenges: The easiest optimization method is upgrading to a newer Python version. Using pyenv, the process is as simple as running a command to install Python 3.12 and reinstalling the necessary libraries. This required no modifications to the code and was completed within minutes.

Performance Expectation: Since our engine relies heavily on recursive search (minimax with alpha-beta pruning) and data structure operations (move generation and evaluation), these improvements should contribute to faster execution and deeper search depths within the same time.

4.3 Pypy

Introduction: PyPy is an alternative implementation of Python that uses JIT compilation to improve performance.

Possible Performance Improvements: Unlike CPython, which interprets bytecode line by line, PyPy dynamically compiles frequently executed code paths into machine code, accelerating performance for pure-Python workloads. This makes it well-suited for operations such as recursive search algorithms and move generation in our chess engine.

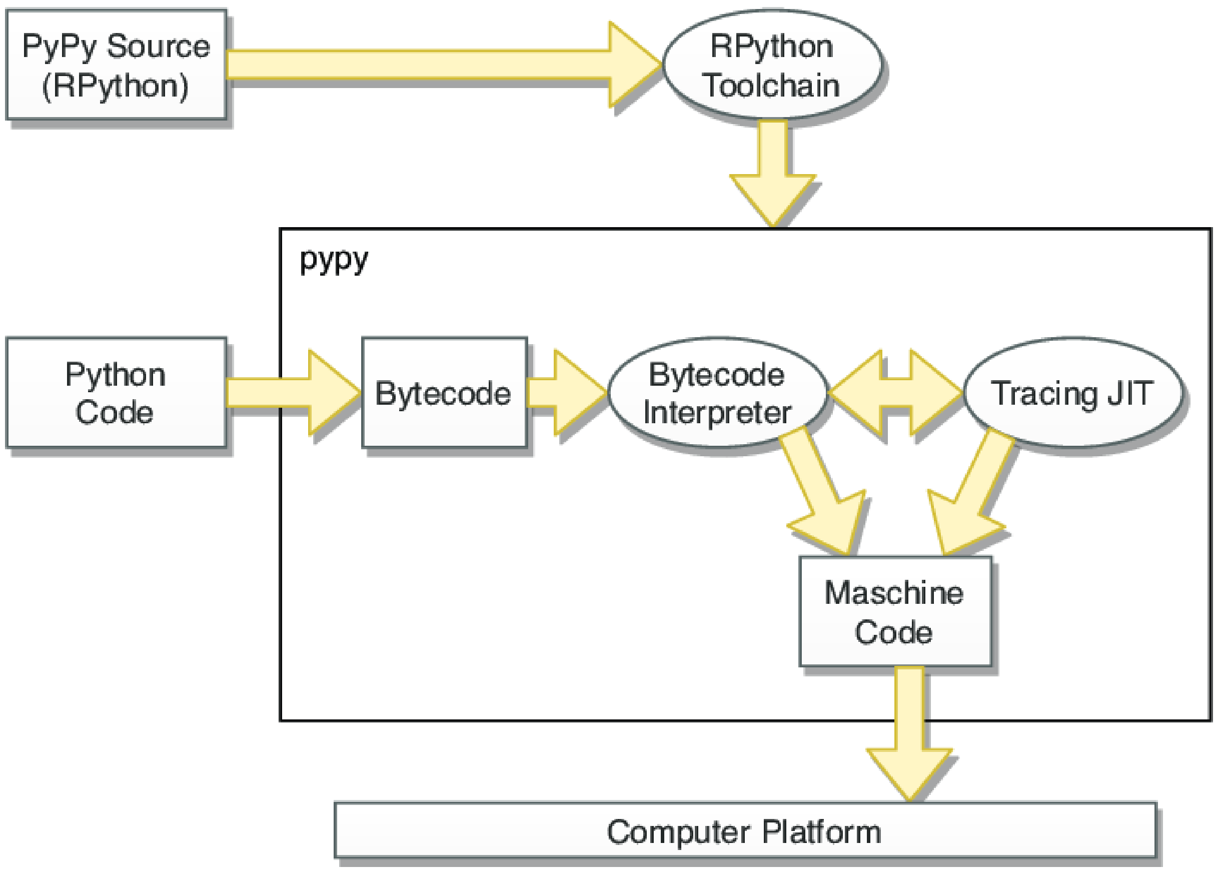

Figure 3: Pypy architecture[27]

How to port: Installing PyPy is straightforward, as it only requires downloading the interpreter. Once installed, we only had to set up NumPy. This can be done with the following commands:

```
pypy -m ensurepip
pypy -m pip install numpy
pypy chess.py # run python file
```

Difficulty and Challenges: Using PyPy was a straightforward process. While it involved downloading and installing a new Python interpreter, reinstalling the dependencies was the only additional step required. The process was slightly more manual compared to using pyenv, but it remained a quick and effortless optimization.

Performance Expectation: While PyPy is expected to improve performance for pure-Python components—such as the minimax search, move generation, and alpha-beta pruning—its compatibility with NumPy presents some challenges. The official PyPy documentation notes that its performance with NumPy is generally worse than that of CPython due to being run through the cpyext compatibility layer[13]. Given this, we expect a mixed impact on performance:

- Speedup for pure-Python code: Recursive functions, branching logic, and dynamic evaluations should benefit from JIT compilation.

- Potential slowdown for numpy-dependent code: Since NumPy is optimized for CPython's C extensions, performance may degrade when running under PyPy.

If the pure-Python components dominate execution time, the overall performance could still see a net improvement.

4.4 Nuitka

Introduction: Nuitka is a Python-to-C compiler that converts Python code into a standalone executable. Unlike tools like PyPy or Numba, which either replace the interpreter or optimize specific sections of code, Nuitka compiles the entire Python program into a C-level binary.

Possible Performance Improvements: Nuitka translates the Python modules into a C level program that then uses libpython and static C files of its own to execute in the same way as CPython does[28]. One of its advantages is that it fully supports all Python features and does not require modifications to the existing codebase, making it an easy-to-use optimization technique. While Nuitka does not perform JIT compilation, its ahead-of-time strategy ensures consistent and portable performance gains across systems.

How to port: We could compile the existing Python code without any modifications for our chess engine. The compilation process was straightforward and involved running the following command:

```
pip install nuitka
python -m nuitka ./default/main.py --follow-imports --output-dir=./nuitka
```

This command compiles the main file, follows all imports, and places the resulting executable and compiled files in the given directory.

Difficulty and Challenges: Nuitka was also easy to use. The compilation process was mostly automatic, with the only potential pitfall being the need to include the `--follow-imports` flag to ensure dependencies such as NumPy were properly compiled. The error messages provided clear guidance, making it easy to resolve issues quickly.

Performance Expectation: Once compiled, the engine can be executed as a standalone binary without requiring a Python interpreter, potentially improving performance. In terms of performance expectations, we expect a moderate improvement.

4.5 Numba

Introduction: Numba is a JIT compiler for Python that optimizes numerical computations by compiling Python code into highly efficient machine code.

Possible Performance Improvements: Numba translates Python functions to optimized machine code at runtime using the industry-standard LLVM compiler library. Numba-compiled numerical algorithms in Python can approach the speeds of C or FORTRAN[29].

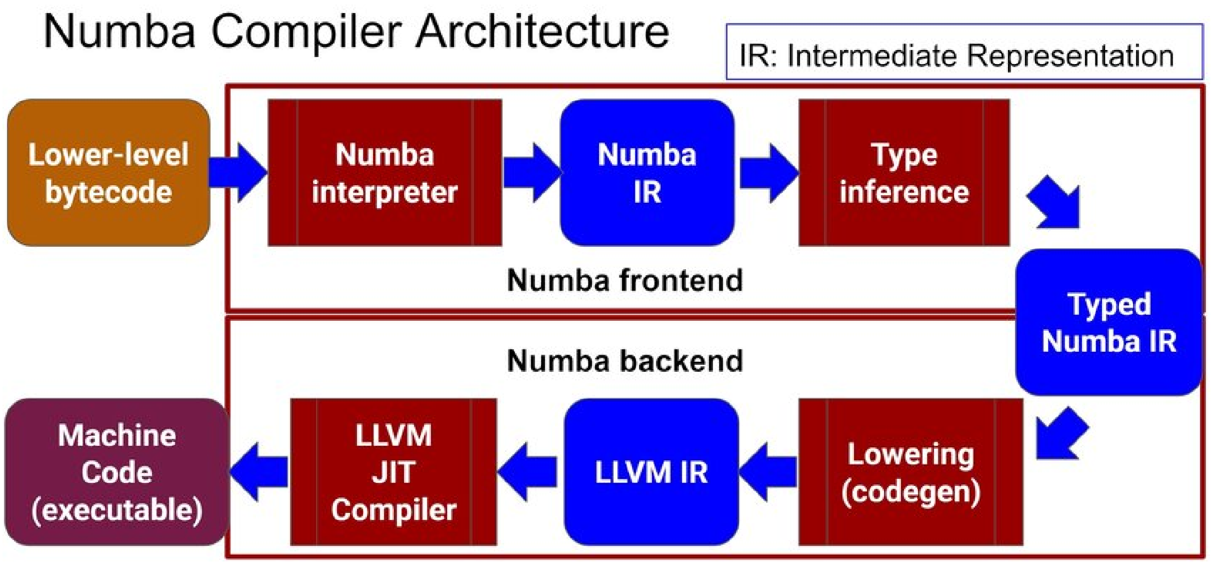

Figure 4: Numba architecture[30]

How to port: In this implementation, Numba optimized parts of the chess engine by accelerating functions that handle board evaluation and move validation. However, not all functions were suitable for Numba optimization. In particular, the move generation logic was left unchanged, as it primarily operates on a Board object, and Numba does not perform well with complex Python objects[31]. Instead, the focus was on optimizing numerical operations that worked with raw NumPy arrays.

To use Numba efficiently, some functions were rewritten to avoid object-oriented structures and instead only use NumPy arrays. For instance, functions that determine whether a square is attacked or whether a move is legal were rewritten, as they only needed the board as a NumPy array. These functions were then decorated with `@njit`, allowing them to be compiled with Numba.

Difficulty and Challenges: Numba required significantly more effort. Since Numba does not work well with dynamic Python objects, some functions had to be rewritten to avoid complex object structures. Functions that relied solely on NumPy arrays or primitive types were easy to optimize, but others required additional wrapper functions to separate numerical computations from object-oriented logic. At the beginning of this project, we used the Python `chess` library to compute moves, but integrating libraries with Numba was harder and less performant than just rewriting the library. The need to refactor the code made it more time-consuming for Numba to integrate than initially expected.

Performance Expectation: While we were not able to add Numba operations to the whole engine, we still expect some performance improvements in the benchmarks.

4.6 Cython

Cython is an optimizing static compiler for both the Python programming language and the extended Cython programming language[32]. It acts as an optimizing compiler that translates Python code into C, which is then compiled into a shared library or executable.

By allowing the use of both Python-like syntax and low-level C operations, Cython bridges the gap between high-level scripting and efficient, compiled execution.

For our chess engine, we explored two approaches to using Cython for optimization. The first approach involves using Cython in the "standard" way—adding type annotations and compiling the existing Python code to improve execution speed. The second approach takes a more direct route by implementing the chess engine in C++ and then exposing it to Python through Cython, leveraging the full power of compiled code while maintaining a Python interface.

Cython can be installed with the following command:

```
pip install cython
```

4.6.1 Cython

Introduction: Cython is a superset of Python designed to enable integration with C and C++. It allows Python code to be compiled into C code by adding static type declarations, leading to performance improvements.

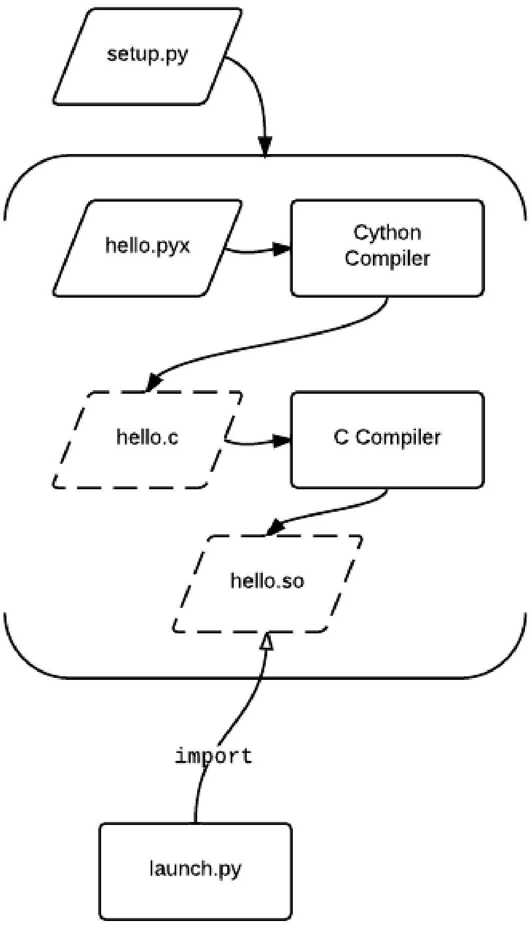

Figure 5: Cython architecture[33]

Possible Performance Improvements: The core optimization lies in Cython's static typing system. You minimize Python object overhead by declaring C types for all critical variables (`cdef int`, `DTYPE_t[:, ::1]` memory views). For instance, the board's internal state uses a NumPy memory view (`DTYPE_t[:, ::1]`) for direct access to contiguous memory blocks, avoiding Python list indirection. The Move class uses `cdef public int` fields, creating a C-struct-like object with a fixed layout for fast attribute access.

Memory views allow a hybrid approach for NumPy arrays. The memory view (`cdef public DTYPE_t[:, ::1]`) provides buffer protocol access for zero-copy interaction with NumPy while maintaining C-level speed. The `[:, ::1]` syntax enforces C-contiguous memory layout, enabling compiler optimizations like SIMD instructions in array operations[34].

Cython allows method declarations using `cpdef` to create dual Python/C interfaces. Internal methods like `push_uci` use `cpdef` for Python accessibility while maintaining optimized C paths internally. In contrast, with `cdef` methods like `_retract_move` exist purely in C-space for maximal speed during move unmaking.

For the move sorting in the move generation file, we chose to use the low-level `qsort` function. By marshalling Python `Move` objects into a C array of structs:

```
entries = <MoveOrderEntry *> malloc(...)
...
qsort(entries, ...)
```

We were able to achieve native-speed sorting ($O(nlogn)$) of move candidates before the alpha-beta search, which is critical for pruning efficiency. The `compare_entries` callback uses `noexcept nogil` to ensure the sort operates without Python GIL overhead.

How to port: For compiling the Cython files, we need a `setup.py` configuration file. It uses a modular build system architecture, utilizing Cython's transpilation infrastructure through the `cythonize` directive to convert .pyx source files into optimized C extension modules, with explicit compiler optimization directives (-O3), enabling processor-specific instruction set utilization and loop transformation techniques, while the inclusion of NumPy headers ensures low-level memory view compatibility critical for zero-copy array operations in board state representations.

```
from distutils.extension import Extension
from setuptools import setup
from Cython.Build import cythonize
import numpy as np

extensions = [
    Extension("chess", ["chess.pyx"],
              include_dirs=[np.get_include()],
              extra_compile_args=["-O3"]),
    Extension("movegeneration", ["movegeneration.pyx"],
              include_dirs=[np.get_include()],
              extra_compile_args=["-O3"]),
    Extension("evaluate", ["evaluate.pyx"],
              include_dirs=[np.get_include()],
              extra_compile_args=["-O3"])
]

setup(
    name="chess_engine",
    ext_modules=cythonize(extensions),
)
```

Listing 2: setup.py

Difficulty and Challenges: Cython proved to be the most challenging optimization to implement. While Cython syntax closely resembles Python, its strict typing requirements and various restrictions led to many trial-and-error iterations to resolve bugs. Certain functions, such as sorting the moves using `qsort`, were particularly difficult to implement due to IDEs not properly recognizing `noexcept nogil` keywords, causing misleading errors. Most of these errors stemmed from being written similar to C++ while using the Python syntax, so maybe it's mostly the experience that's missing. These challenges made Cython the hardest optimization to integrate.

Performance Expectation: Finally, we expect a performance boost due to the C transpiling and the compiler optimizations.

4.6.2 C++

Introduction: Instead of optimizing Python code directly, we rewrote the entire chess engine in C++ to leverage the performance benefits of a compiled language. The core logic remains unchanged, but the implementation now takes advantage of C++'s static typing and compiler optimizations and removes the overhead of the Python interpreter.

Possible Performance Improvements: By reimplementing the engine in C++, we get several performance benefits.:

- No interpreter overhead: C++ code is compiled directly to machine code, avoiding Python's runtime inefficiencies.
- Better memory locality: Using arrays and structs in C++ ensures faster memory access patterns.
- Predictable execution: Unlike dynamic Python objects, C++ variables have fixed sizes and types, allowing compilers to optimize aggressively.

How to port: To integrate the C++ implementation with Python, we used Cython to create a wrapper, allowing us to import the C++ classes into Python. This approach enables us to retain Python's flexibility while significantly boosting performance. A small excerpt of the wrapper can be seen in Listing 3.

```
# ===== C++ Interface Declarations =====
cdef extern from "chess.hpp":
    cdef cppclass CBoard "Board":
        CBoard() except +
        void reset()
        void set_fen(string fen)
        void push_uci(string move_str)
        void push(const CMove& move)
        void pop()
        vector[CMove] legal_moves()
        int turn
        cbool is_move_legal(const CMove& move, int color)
...
# ===== Python Wrappers =====
cdef class Board:
    cdef CBoard c_board

    def __init__(self, fen=None):
        if fen:
            self.set_fen(fen)
        else:
            self.reset()

    def set_fen(self, fen):
        self.c_board.set_fen(fen.encode('utf-8'))

    def push_uci(self, uci_str):
        self.c_board.push_uci(uci_str.encode('utf-8'))

    def legal_moves(self):
        cdef vector[CMove] moves = self.c_board.legal_moves()
        return [self._convert_move(m) for m in moves]

    cdef _convert_move(self, CMove cmove):
        py_move = Move()
        py_move.c_move = cmove
        return py_move
...
```

Listing 3: wrapper

The Cython wrapper consists of two main parts:

1. C++ Interface Declarations: These specify which C++ classes and functions should be accessible from Python. The `cdef extern` block declares C++ classes like `Board` and `Move`, ensuring that Cython can call them directly.

2. Python Wrappers: These define Python classes that wrap the C++ objects. The wrapper also adds compatibility functions, like decoding/encoding strings[35].

Using this wrapper, Python can interact with the chess engine as if it were a native Python module.

To execute this chess engine, we have to compile it with the following `setup.py`.

```
import numpy as np
from setuptools import setup, Extension
from Cython.Build import cythonize

extensions = [
    Extension("chess_engine",
              ["chess_engine.pyx", "evaluation.cpp", "chess.cpp", "
   movegeneration.cpp"],
              language="c++",
              extra_compile_args=["-std=c++17", "-O3"],
              include_dirs=[np.get_include()])
]

setup(
    name="chess_engine",
    ext_modules=cythonize(extensions),
)
```

Listing 4: setup.py

After compiling, we simply execute the `main.py` of the chess engine.

Difficulty and Challenges: Rewriting the chess engine in C++ took more time but was relatively straightforward. Many functions could be rewritten in less than a minute, while others required more attention, especially those involving pointer arithmetic, where extra care was needed to prevent memory errors. The full C++ reimplementation was completed within a few hours despite the increased complexity.

Performance Expectation: With this setup, we expect significant speedups in performance due to the implementation of C++.

4.7 Performance Expectations

Optimizing a chess engine involves balancing two competing objectives: minimizing development effort and maximizing execution speed. Some methods offer immediate speedups with minimal effort (e.g., upgrading Python versions), while others require substantial code rewrites and technical expertise (e.g., rewriting in C++ via Cython).

To evaluate the trade-offs, we rated each optimization method across two dimensions: time investment and ease of use. These are subjective but reflect real-world development constraints.

Method	**Time Effort (1–5)**	**Ease-of-Use (1–5)**	**Notes**
Python Version Upgrade	1	1	Drop-in replacement via `pyenv`
PyPy	1	1	Works out-of-the-box, best for pure Python code
Nuitka	1	2	Simple binary generation, needs flags for imports
Numba	3	3	Requires code restructuring around NumPy operations
Cython (Python)	5	5	Static typing, memoryviews, and qsort integration
C++ via Cython	5	4	Complete rewrite with better control and performance

Table 1: Optimization strategies rated by time investment and complexity (1 = lowest, 5 = highest)

From this table, we observe that:

- Lightweight methods like PyPy and Nuitka are excellent first steps, offering speedups with minimal friction.
- Numba is effective but demands architectural awareness, especially in separating numerical logic from object-oriented design.
- Cython, particularly with C++ integration, offers the highest performance ceiling but also requires the greatest effort.

The overall performance gain follows a trend of diminishing returns. Moving from Python 3.10 to PyPy or Nuitka can offer significant improvements with little effort, but subsequent gains require increasingly specialized implementations.

Our expectations for the performance are that we get more performance, likely linear, with more time investments.

5 Performance Analysis

In this section, we will show and analyze the performance results of the optimization strategies shown in section 4. The analysis is based on two primary metrics: the average number of nodes evaluated per move and the win rate against the unoptimized default Python engine, from which we also derive Elo rating differences.

5.1 Setup

Each chess matchup had 300 matches to ensure that there were no outliers.

Each chess match was executed on the following hardware:

- Central Processing Unit (CPU): Intel i7 10700k; 8 cores, 16 threads, 5.10 GHz frequency
- Random-Access Memory (RAM): 32GB DDR4 3000Mhz in dual-channel
- Storage: 2TB SATA SSD

We used Debian 12 as our operating system, ensuring that no background processes were active during the operation. Therefore, we maintain system performance and integrity.

5.2 Evaluation Speed: Nodes per Move

The average number of moves computed each round can tell us how fast each implementation is. The results are summarized below:

- Default (avg: 2,166): The baseline engine, running with Python 3.10.
- C++ (avg: 231,069): As expected, the C++ implementation outperformed all others by a large margin.
- Cython (avg: 9,284): Among the Python-adjacent tools, Cython demonstrated the highest node count, nearly quadrupling the performance of the default Python engine. Despite its challenging setup and debugging experience, the performance boost it provides is significant.
- Numba (avg: 4,520): Numba showed promising results when it could be effectively applied. It requires rewriting certain functions to avoid dynamic Python types, but for numerical operations (especially those using NumPy), the performance was nearly double that of the default engine.
- Nuitka (avg: 2,471): Nuitka compiled the Python code to C, resulting in modest but consistent gains in performance. Its ease of use, combined with reliable improvements, makes it a solid optimization tool.
- Python 3.12 (avg: 2,401): Upgrading to Python 3.12 provided a measurable speed-up over older versions. This highlights the ongoing improvements in CPython's performance in recent updates.
- PyPy (avg: 374): Surprisingly, PyPy underperformed in this benchmark. While its JIT compiler generally speeds up Python code, each NumPy operation slows down the performance due to the cpytext compatibility layer.

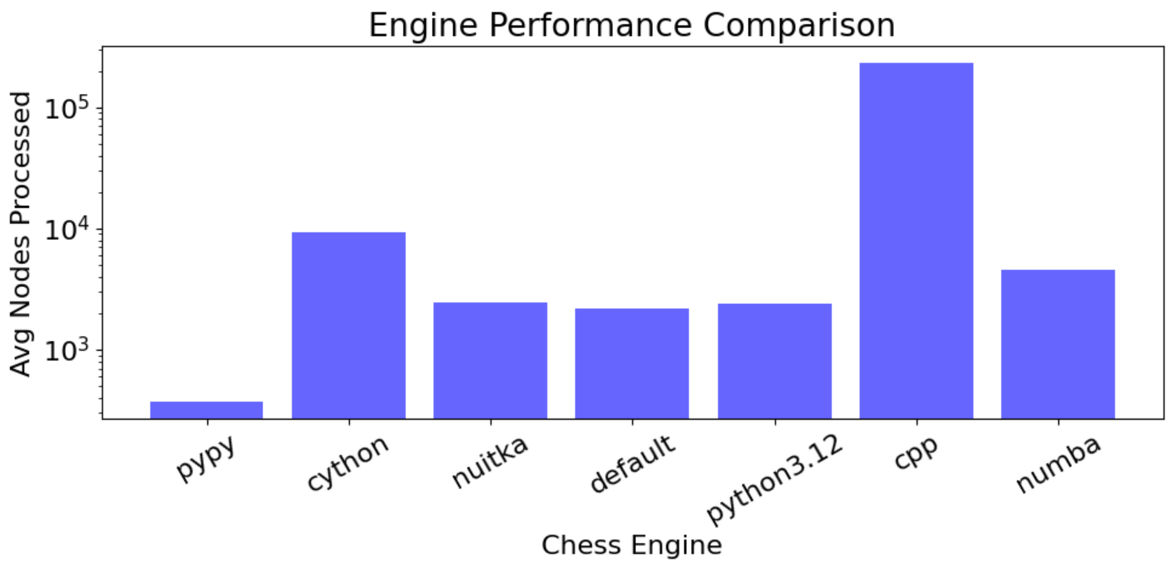

Figure 6: Number of processed Nodes in one second per engine

5.3 Playing Strength: Win Rates and Elo Ratings

The true test of a chess engine lies not only in its speed but in the quality of its decisions. To evaluate this, each optimized engine was pitted against the default implementation in 300 games. Figure 7 Figure 8 summarizes the win/draw/loss percentages and estimated Elo rating differences.

The Elo rating system in chess is a method for calculating the relative skill levels of players. It assigns each player a rating that rises or falls based on game outcomes against other players. This system helps match players of similar skills and tracks improvement over time. As a general rule of thumb, a player who is rated 100 points higher than their opponent is expected to win roughly five out of eight (64%) games. A player with a 200-point advantage will presumably win three out of four (75%) games[36].

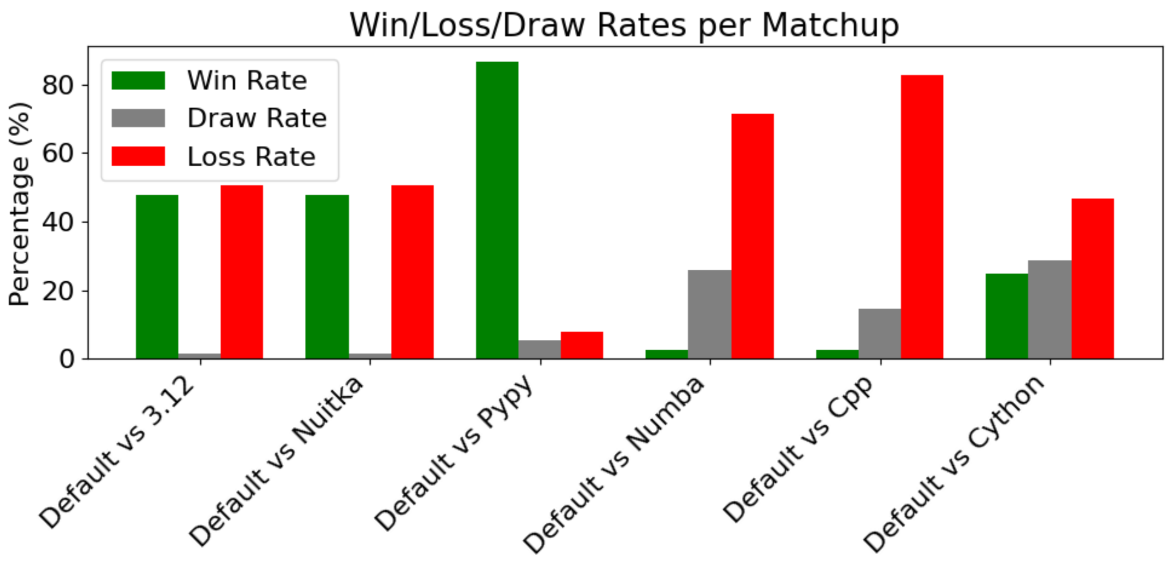

Figure 7: Wins/draws/losses of matches of the baseline engine vs the optimized engine

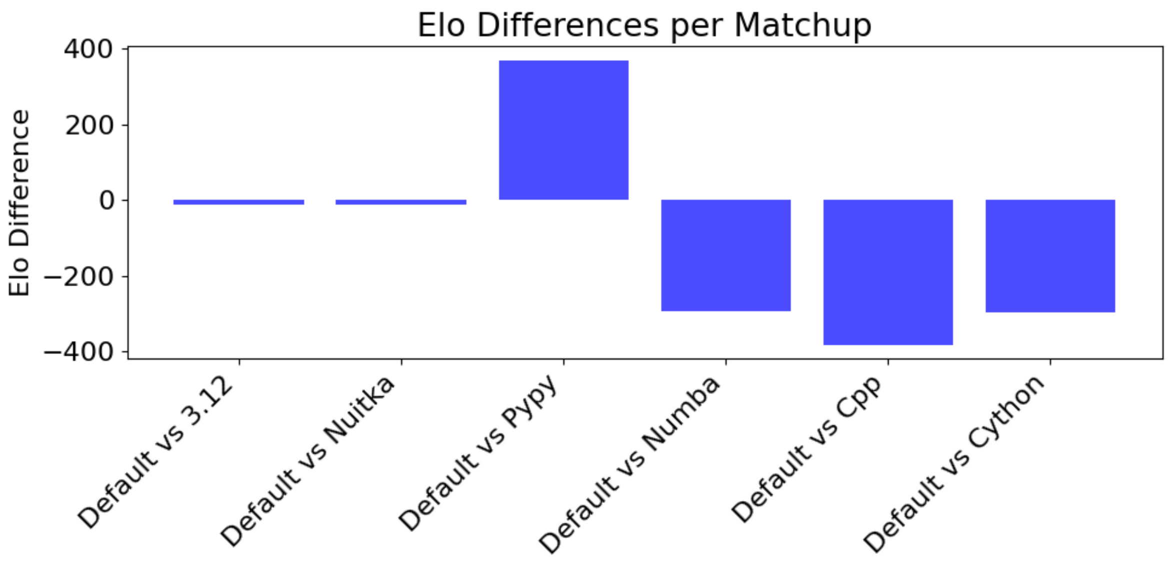

Figure 8: Elo differences for each matchup of the baseline engine vs the optimized engine

The C++ implementation stands out with the highest performance both in evaluation speed and game strength, defeating the default engine in over 80% of games and outperforming it by approximately 380 Elo points.

Numba and Cython follow as strong contenders, each surpassing the default engine by nearly 300 Elo points. These engines leveraged numerical optimization and static typing to improve both the speed and quality of move selection.

Nuitka's and Python 3.12's nearly even win/loss record and minimal Elo difference indicate that its optimizations improved speed but did not dramatically change the quality of play. Also, while Cython is faster than Numba, Numba performs better in chess games. It shows that higher performance does not always translate to stronger decision-making. This may be because while more moves are computed, it does not mean that better moves are computed. The engine first tries to evaluate moves that have a higher chance of getting a good score. Therefore, we may only see improvements if we reach a higher search depth.

In contrast, PyPy, despite being easy to use, severely underperformed in game strength, losing the vast majority of games and therefore also losing almost 400 Elo points.

5.4 Interpretation

The results clearly show a trade-off between ease of use, optimization effort, and performance gain. While C++ and Cython required more effort and setup complexity, they yielded the most significant improvements in speed and playing strength. Numba, with moderate implementation effort, also performed very well when applied correctly.

Conversely, PyPy's ease of use and minimal setup did not compensate for its poor performance in this specific use case, showing that not all optimization tools are equally effective for all workloads. Finally, techniques like Nuitka and Python 3.12 were easier to implement but provided only modest improvements.

Overall, this analysis supports the hypothesis that there is a linear scaling of performance vs ease of use for the tools we used.

6 Conclusion

6.1 Summary of Findings

This study explored the optimization potential of a Python-based chess engine using a range of performance enhancement tools and techniques. The goal was to push the limits of what is possible within the Python ecosystem and to see how close we could get to native-level performance through compilation, JIT optimizations, and language extensions.

Across the various tools evaluated, PyPy, Nuitka, Numba, Cython, Python 3.12, and a C++ reimplementation, several findings emerged:

- **Ease of Use:** Python 3.12 and PyPy were by far the easiest to adopt, requiring little to no code modification. Nuitka was also relatively straightforward, while Cython and Numba required significant refactoring. The C++ reimplementation was the most time-intensive but not prohibitively complex.

- **Performance:** In terms of average nodes per move, C++ outperformed all others by a large margin, followed by Cython and Numba. Nuitka and Python 3.12 showed modest improvements over the default interpreter, while PyPy lagged behind in raw speed.

- **Playing Strength:** More move evaluations did correlate with higher performance over many games.

In conclusion, Python can be made more performant with the right tooling and effort. However, reaching higher performance involves trade-offs in complexity, stability, and development time.

All implementations and scripts are on our GitHub repository[1].

[1] `https://github.com/FrederikHennecke/ChessOptimizationPython`

6.2 Future Work and Outlook

One direction for future performance improvements is the integration of parallelism into the chess engine. Since many parts of the move search process, such as evaluating multiple branches of the game tree, can be executed independently, parallel computing could significantly reduce computation time. Python's GIL poses a challenge for true multi-threaded execution, but alternatives such as multiprocessing, asynchronous task scheduling, or libraries can help use modern multi-core architectures more effectively.

Beyond the tools explored in this project, various other performance enhancement options could be investigated. Tools like PyO3 allow for the integration of Rust code into Python projects, offering performance improvements similar to Cython. Similarly, CFFI[37] or SWIG[38] could be used to bind C or C++ modules. Additionally, libraries such as TensorFlow or JAX might open the door to hybrid approaches where evaluation functions or heuristics are accelerated via machine learning or just-in-time compilation on a Graphics Processing Unit (GPU). Exploring these tools would provide a broader understanding of the performance landscape available to Python developers working on computationally intensive applications.

References

[1] Alexandros Nikolaos Ziogas et al. “Productivity, portability, performance: data-centric Python”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC ’21. St. Louis, Missouri: Association for Computing Machinery, 2021. ISBN: 9781450384421. DOI: `10.1145/3458817.3476176`. URL: `https://doi.org/10.1145/3458817.3476176`.

[2] *Boxing (computer programming)*. en. Page Version ID: 1270236391. Jan. 2025. URL: `https://en.wikipedia.org/w/index.php?title=Boxing_(computer_programming)&oldid=1270236391`.

[3] *The Python Method Resolution Order*. en. URL: `https://docs.python.org/3/howto/mro.html`.

[4] *Application programming on z/OS*. en-US. June 2023. URL: `https://www.ibm.com/docs/en/zos-basic-skills?topic=zos-compiled-versus-interpreted-languages`.

[5] Claude E. Shannon. “XXII. Programming a computer for playing chess”. In: *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 41.314 (Mar. 1950), pp. 256–275. ISSN: 1941-5990. DOI: `10.1080/14786445008521796`. URL: `http://dx.doi.org/10.1080/14786445008521796`.

[6] *Minimax*. en. Page Version ID: 1282084908. Mar. 2025. URL: `https://en.wikipedia.org/w/index.php?title=Minimax&oldid=1282084908`.

[7] *Perfect Information*. en. Page Version ID: 1283537762. Apr. 2025. URL: `https://en.wikipedia.org/w/index.php?title=Perfect_information&oldid=1283537762`.

[8] *Perfect Competition*. en. Page Version ID: 1270878961. Jan. 2025. URL: `https://en.wikipedia.org/w/index.php?title=Perfect_competition&oldid=1270878961`.

[9] D. J. Edwards and T. P. Hart. “The Alpha-Beta Heuristic”. en. In: (Dec. 1961). Accepted: 2004-10-04T14:39:10Z. URL: `https://dspace.mit.edu/handle/1721.1/6098`.

[10] The Stockfish developers (see AUTHORS file). *Stockfish*. URL: `https://github.com/official-stockfish/Stockfish`.

[11] David Silver et al. “Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm”. en. In: arXiv:1712.01815 (Dec. 2017). arXiv:1712.01815 [cs]. DOI: `10.48550/arXiv.1712.01815`. URL: `http://arxiv.org/abs/1712.01815`.

[12] The LCZero Authors. *LeelaChessZero*. URL: `https://github.com/LeelaChessZero/lc0`.

[13] *PyPy FAQ*. URL: `https://doc.pypy.org/en/latest/faq.html#should-i-install-numpy-or-numpypy`.

[14] *Nuitka the Python Compiler — Nuitka the Python Compiler*. URL: `https://nuitka.net/`.

[15] Stefan Behnel et al. “Cython: The Best of Both Worlds”. In: *Computing in Science & Engineering* 13.2 (Mar. 2011), pp. 31–39. ISSN: 1521-9615. DOI: `10.1109/MCSE.2010.118`.

[16] *Numba: A High Performance Python Compiler*. URL: https://numba.pydata.org/.

[17] 262588213843476. *Description of the Universal Chess Interface (UCI)*. en. URL: https://gist.github.com/DOBRO/2592c6dad754ba67e6dcaec8c90165bf.

[18] Charles R. Harris et al. "Array programming with NumPy". In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2. URL: https://doi.org/10.1038/s41586-020-2649-2.

[19] Niklas Fiekas. *niklasf/python-chess*. Python. Apr. 2025. URL: https://github.com/niklasf/python-chess.

[20] en. Page Version ID: 1235687097. July 2024. URL: https://en.wikipedia.org/w/index.php?title=XBoard&oldid=1235687097.

[21] *Standard: Portable Game Notation Specification and Implementation Guide*. URL: https://ia802908.us.archive.org/26/items/pgn-standard-1994-03-12/PGN_standard_1994-03-12.txt.

[22] Stuart Russell and Peter Norvig. *Artificial intelligence*. en. 4th ed. Upper Saddle River, NJ: Pearson, Nov. 2020, pp. 152–161.

[23] Richard E. Korf. "Depth-first iterative-deepening: An optimal admissible tree search". In: *Artificial Intelligence* 27.1 (1985), pp. 97–109. ISSN: 0004-3702. DOI: https://doi.org/10.1016/0004-3702(85)90084-0.

[24] Tomasz Michniewski. *Simplified Evaluation Function - Chessprogramming wiki*. 2021. URL: https://www.chessprogramming.org/Simplified_Evaluation_Function (visited on 04/08/2025).

[25] *PyPI Stats*. URL: https://pypistats.org/.

[26] *What's New In Python 3.11*. en. URL: https://docs.python.org/3/whatsnew/3.11.html.

[27] Lukas Razik. "High-Performance Computing Methods in Large-Scale Power System Simulation". PhD thesis. May 2020.

[28] *Nuitka User Manual*. URL: https://nuitka.net/doc/user-manual.pdf.

[29] *Numba: A High Performance Python Compiler*. URL: https://numba.pydata.org/.

[30] Omid Saedi. "Towards Eco-Conscious Python: A Comparative Analysis of Performance, Energy Efficiency and Carbon Emissions Between CPython and Alternative Implementations". en. In: (2024). DOI: 10.13140/RG.2.2.12314.04803. URL: https://rgdoi.net/10.13140/RG.2.2.12314.04803.

[31] *Compiling Python classes with @jitclass*. URL: https://numba.readthedocs.io/en/stable/user/jitclass.html#limitations.

[32] *Cython*. URL: https://cython.org/.

[33] DavidBrooksPokorny. *Plaques of Lambda Phages on E. coli XL1-Blue MRF*. File: LambdaPlaques.jpg. 2012. URL: https://en.wikipedia.org/wiki/Cython#/media/File:Cython_CPython_Ext_Module_Workflow.png.

[34] *Typed Memoryviews*. URL: https://cython.readthedocs.io/en/latest/src/userguide/memoryviews.html.

[35] *Unicode and passing strings*. URL: https://cython.readthedocs.io/en/latest/src/tutorial/strings.html.

[36] *Elo Rating System.* en-US. URL: `https://www.chess.com/terms/elo-rating-chess` (visited on 04/08/2025).

[37] *CFFI: Foreign Function Interface for Python calling C code.* Python. Apr. 2025. URL: `https://github.com/python-cffi/cffi`.

[38] *SWIG (Simplified Wrapper and Interface Generator).* C++. Apr. 2025. URL: `https://github.com/swig/swig`.