Journal of Computers, Mechanical and Management
Journal homepage: www.jcmm.co.in
e-ISSN: 3009-075X
AN PUBLISHING

Volume 5 Issue 2
Article Number: 25266

Workload-Specific Performance Evaluation of Python Just-in-Time Compilers: A Comparative Study of Numba and Cython

Sattaru Harshavardhan Reddy[1], Priya Gupta* [2], Deepak Kumar[3], and Ritu Singhal[4]

[1]School of Engineering, Jawaharlal Nehru University, New Delhi, India 110067
[2]Atal Bihari Vajpayee School of Management and Entrepreneurship, Jawaharlal Nehru University, New Delhi, India 110067
[3]Tata Consultancy Services, Boston, MA, USA 02116
[4]Indraprastha College for Women, University of Delhi, Delhi, India 110054

Abstract

The persistent performance gap in dynamic languages like Python has driven the development of numerous compiler solutions. This paper presents a comparative performance analysis of two prominent Python compilers, Numba and Cython, across distinct computational workloads: recursive (Fibonacci series) and arithmetic-intensive (Euclidean distance). Addressing a gap in existing literature, this study provides an evidence-based framework that maps compiler performance directly to workload types. Experiments conducted in a controlled environment measured execution time, speedup ratios, and memory usage. Results demonstrate that Numba achieves a speedup of up to 6.18× over pure Python for arithmetic-intensive tasks, while Cython performs better in deep recursion cases. The study concludes by offering a workload-to-compiler decision framework, which serves as a practical tool and a contribution to the literature on scenario-based compiler recommendations.

Keywords: Just-In-Time Compilation; Numba; Cython; Python Optimization; Performance Benchmarking

1. Introduction

The demand for high-performance computing in fields such as scientific research and data analysis has driven the need for efficient compilers, even for traditionally interpreted languages such as Python. While various tools have been developed to accelerate Python code, there remains a notable lack of comparative, workload-aware analysis to guide developers in selecting the most suitable compiler for specific tasks. This paper addresses that gap by providing a comparative performance analysis of two prominent Python compilers: Cython, an Ahead-of-Time (AOT) compiler, and Numba, a Just-in-Time (JIT) compiler.

This research explicitly evaluates these compilers' performance across two distinct workload categories: arithmetic-intensive tasks (e.g., Euclidean distance calculation) and recursion-heavy tasks (e.g., Fibonacci sequence generation). By systematically benchmarking their effectiveness on these diverse workloads, the study aims to provide a practical, evidence-based framework that helps developers make informed decisions. This framework provides illustrative guidance on how JIT and AOT compilers behave under distinct workload characteristics, rather than a universal prescription across all Python applications.

*Corresponding Author: Priya Gupta (priyagupta@jnu.ac.in)
Received: 12 Dec 2025; Revised: 08 Jan 2026; Accepted: 23 Jan 2026; Published: 30 Apr 2026

DOI: 10.57159/jcmm.5.2.25266.

1.1. Compilers and Just-in-Time (JIT) Compilation

A compiler is a program that translates code written in a high-level language into machine code that a computer can execute [1]. This traditional, or static, compilation process is performed before the program runs. In contrast, Just-in-Time (JIT) compilers translate source code into native binary code at runtime. This dynamic approach allows for optimizations that are not possible with static compilers, as the JIT compiler has a real-time view of the machine's state. These dynamic optimizations can significantly boost application performance and reduce memory usage. JIT compilers have become a cornerstone of modern application development, widely used in web browsers for languages such as JavaScript and in high-level languages such as Java. In the context of Python, JIT compilers such as Numba and Cython are potent tools for optimizing execution speed, especially for computationally intensive tasks.

1.2. Related Work

Recent research on Just-in-Time (JIT) compilation spans security, performance optimization, and domain-specific integration. Smith et al. [2] address security risks by introducing Icarus, a formally verified JIT framework that uses symbolic meta-execution to ensure memory safety for all generated code, achieving performance on par with hand-written compilers. Performance-focused works include Tian et al. [3], who combine link-time optimization (LTO) with JIT for OpenMP target offloading, and Pichler et al. [4, 5], who blend ahead-of-time (AOT) and JIT compilation in GraalVM—initially manually, then automatically via call-graph analysis—to improve warm-up and peak performance. Jakob et al. [6] present Dr.Jit, a high-performance JIT for physically based and differentiable rendering, which traces and specializes high-level simulation code into optimized CPU/GPU kernels while eliminating redundant computations through global data dependency tracking. Lightweight, domain-specific approaches are explored by Ning et al. [7], who design a simple LLVM-based JIT for relational databases, and Ma et al. [8], who compare LLVM- and WASM-based JIT architectures in database execution engines. Latifi et al. [9] propose CompGen, an efficient compiler generator inspired by the second Futamura projection to reduce compilation time while maintaining high performance, while Zhang et al. [10] develop comPyler, a compatibility-focused Python JIT with comparative analysis across multiple Python implementations. From a security perspective, Bauman et al. [11] introduce Renew, enabling robust control-flow integrity and software fault isolation for self-modifying code in JIT-based systems with minimal performance overhead. Collectively, these works demonstrate how JIT compilation research is advancing toward safer, faster, and more adaptable solutions across languages, platforms, and application domains.

1.2.1 Python Compiler Landscape

Dynamic languages such as Python offer flexibility and a rapid development cycle, but often trade execution speed for these benefits [12]. To bridge this performance gap, developers employ compilers to translate performance-critical code into native machine instructions. These compilation strategies broadly fall into two categories:

1. **Ahead-of-Time (AOT) or Static Compilation:** Tools such as Cython extend Python with optional static typing and compile the code into optimized C/C++ before execution.
2. **Just-in-Time (JIT) Compilation:** Tools such as Numba, PyPy, and Pyston specialize and optimize code at runtime by leveraging dynamic information that is unavailable during static compilation.

JIT compilers can apply aggressive optimizations, sometimes outperforming AOT compilers for specific workloads. However, they can also introduce trade-offs such as warm-up delays and increased memory usage [13–16].

Other execution engines, such as PyPy and Pyston, also provide JIT acceleration. These alternative Python execution engines were excluded from the present study because their whole-program JIT strategies and adaptive optimization heuristics are not easily isolatable at the level of individual computational kernels. Because this study focuses on micro-benchmarking specific workload patterns (recursion-heavy and arithmetic-intensive functions) under controlled conditions, inclusion of such interpreters would confound compiler-level effects with runtime system behaviors, limiting interpretability of the results.

1.2.2 Cython and Numba

Cython extends Python by allowing optional static typing and compiling the resulting code into optimized C. This makes it particularly effective for computational loops and interoperability with existing native libraries. It is well-suited for cases where data types are known in advance [13].

Numba uses the LLVM compiler infrastructure to JIT-compile a subset of Python and NumPy code at runtime. It excels in scientific and numerical workloads by supporting vectorization, parallel execution, and GPU offloading, often delivering significant performance gains with minimal code changes [12, 17–22].

1.2.3 Research Gap

While prior studies have extensively documented the individual design and performance benefits of tools such as Cython and Numba, a notable research gap exists in providing a comparative, workload-aware analysis under uniform experimental conditions. Most existing research focuses on implementation strategies or isolated benchmarks without providing actionable, context-specific guidance. There is a clear need for empirical evidence that helps developers select the most suitable compiler based on the code's computational patterns, such as recursion-heavy logic versus arithmetic-intensive tasks. This study addresses this gap by systematically evaluating Numba (JIT) and Cython (AOT) [23, 24] across these distinct workload categories. Statistical analysis of execution time, speedup, and memory usage was performed to provide evidence-based recommendations for practitioners.

2. Methods

2.1. Cython: An Ahead-of-Time (AOT) Compiler

Cython is a static compiler that generates highly optimized C or C++ code from a Python-like syntax. Its core strength lies in its support for optional static typing, which allows developers to declare variable types explicitly, enabling the compiler to generate efficient C code directly compatible with existing C and C++ libraries. Although this process results in fast execution, it introduces a separate compilation step before runtime. This makes Cython particularly well-suited for applications where performance is critical and variable types are known in advance, such as in scientific computing and numerical simulations. In this study, Cython was used with default compilation settings and without explicit static type annotations, reflecting a common adoption scenario for developers seeking performance gains with minimal code modification. Consequently, the reported performance results correspond to untyped Cython rather than fully type-specialized implementations. Explicit type declarations can further reduce overhead and improve performance, particularly in numerical kernels; however, evaluating such manually optimized variants was beyond the scope of this comparative study and is identified as a direction for future work.

2.2. Numba: A Just-in-Time (JIT) Compiler

Numba is a Just-in-Time (JIT) compiler built on the LLVM compiler infrastructure. It specializes in optimizing numerical code, especially for NumPy array functions. Unlike Cython, Numba compiles code at runtime by inferring data types and applying optimization passes to generate efficient machine code. This dynamic approach enables aggressive optimization and supports features such as automatic parallelization and GPU offloading (via CUDA). Although Numba provides significant speedups for numerical tasks with minimal code changes, it can be incompatible with some standard Python library functions and may introduce a "warm-up" delay during the first function call.

2.3. Experimental Environment

To ensure the reproducibility and consistency of the results, all experiments were conducted on a uniform hardware and software environment:

- **Processor:** Intel Core i7-10750H CPU @ 2.60 GHz
- **RAM:** 16 GB DDR4
- **Operating System:** Windows 11 Pro 64-bit
- **Python Version:** 3.10.9
- **Cython Version:** 0.29.34
- **Numba Version:** 0.57.0

All experiments were conducted under standard operating conditions, without explicitly disabling CPU frequency scaling or turbo boost. Background user applications were minimized during benchmarking to reduce interference. Fine-grained control of dynamic frequency scaling and OS-level scheduling was not enforced. As a result, reported timings should be interpreted as representative rather than cycle-accurate microbenchmark measurements.

2.4. Benchmarking Method

Two distinct benchmarks were designed to test the performance of Cython and Numba under different computational paradigms. Each benchmark was executed 10 times, and the average execution time was recorded.

1. **Recursive Workload:** Performance of both compilers was measured by computing the Fibonacci sequence for depths ranging from 20 to 40. This recursion depth range was chosen to remain safely below Python's default recursion limit while still inducing sufficiently deep call stacks to stress function invocation and return mechanisms.
2. **Arithmetic Workload:** Performance was tested on an arithmetic-intensive task by computing the Euclidean distance for 1 million data points. This workload is ideal for evaluating a compiler's ability to optimize mathematical operations on large datasets. The arithmetic kernel uses `math.sqrt` within an explicit loop; no explicit SIMD vectorization directives were applied. The Numba-compiled implementation therefore executes scalar code, and observed performance gains are attributable to JIT compilation and runtime optimizations rather than auto-vectorization.

The arithmetic workload size of 1 million data points was selected to exceed cache-friendly problem sizes and ensure that execution time is dominated by numerical computation rather than loop setup, memory allocation, or I/O overhead, thereby providing a representative compute-bound scenario for evaluating compiler optimizations. For each benchmark, 95% confidence intervals were computed across the ten independent runs. These intervals are not explicitly reported in summary tables to preserve readability and compact presentation and are instead reflected through variance indicators (error bars) in Figures 1–3.

Performance was measured using three key metrics:

- **Execution Time** (in seconds)
- **Speedup Ratio:** Calculated as (Pure Python time / Optimized compiler time)
- **Memory Usage:** Monitored using the `memory_profiler` library to assess memory efficiency

The source code for all benchmarks, including the `setup.py` and `.pyx` files have been provided in the Appendix for full reproducibility. A separate measurement of compilation overhead was conducted by isolating the first-call latency for Numba. This overhead (0.07–0.10 s) can materially affect short-running workloads. The benchmark suite in this study is intentionally minimalistic, focusing on two fundamental workload categories that do not capture the full spectrum of industrial or scientific Python performance scenarios.

3. Results and Discussion

3.1. Comparative Performance

The benchmark results, as shown in Figures 1 and 2 and Table 1, indicate that both compilers reduce execution time compared to pure Python.

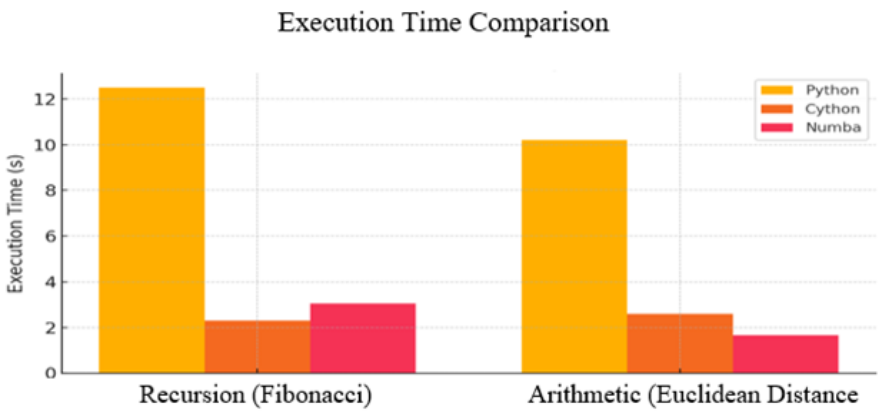

Figure 1: Execution time comparison of Python, Cython, and Numba for recursion (Fibonacci) and arithmetic-intensive (Euclidean distance) workloads.

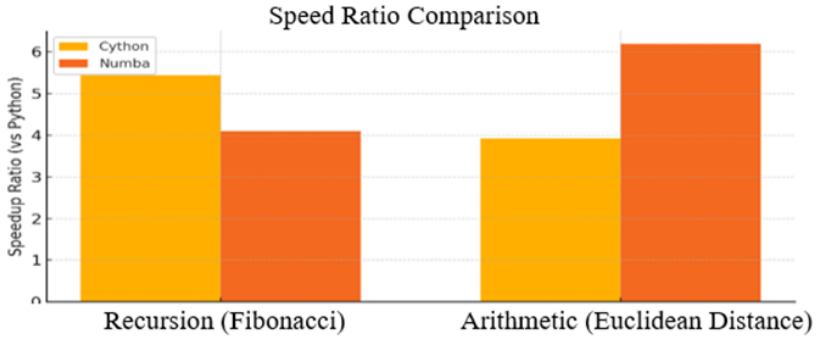

Figure 2: Speedup ratio of Cython and Numba over pure Python for recursion and arithmetic workloads.

Table 1: Benchmark results for Python, Cython, and Numba across recursion and arithmetic workloads. All reported values represent the mean of ten executions, with variations within ±0.05 s.

Workload	**Python Time (s)**	**Cython Time (s)**	**Numba Time (s)**	**Cython Speedup**	**Numba Speedup**	**Python Mem (MB)**	**Cython Mem (MB)**	**Numba Mem (MB)**
Recursion (Fibonacci)	12.5	2.3	3.05	5.43×	4.10×	50	45	49
Arithmetic (Euclidean Distance)	10.2	2.6	1.65	3.92×	6.18×	48	46	51

However, their performance varies substantially depending on the following workloads:

- **Recursive Workload (Fibonacci):** Cython exhibited superior performance for recursion-heavy tasks, achieving a speedup of 5.43× over pure Python. Numba also provided a speedup but was less effective at 4.10×. This performance trend is consistent with the effects of static compilation in Cython, which can mitigate overheads associated with frequent function invocation in deep recursive call patterns, although function-call overhead was not isolated as an independent metric in this study.
- **Arithmetic Workload (Euclidean Distance):** Numba was the clear winner for arithmetic-intensive tasks, achieving a speedup of 6.18× over pure Python. Cython delivered a more modest speedup of 3.92×. Numba's strength in this area comes from its advanced Just-in-Time optimizations, particularly its efficient handling and vectorization of NumPy arrays. Across the tested input sizes reported in Appendix B, execution time decreased monotonically relative to pure Python as problem size increased, with the maximum observed speedup of 6.18× occurring at the largest evaluated input size. Variance across runs was low (SD $<$ 0.06 s).

3.2. Memory Usage and Other Observations

- **Memory Usage:** As shown in Figure 3 and Table 1, Cython and Numba showed memory consumption within a 10% overhead of pure Python, with Cython slightly more memory-efficient. The reported values correspond to peak sampled resident memory observed during benchmark execution using `memory_profiler`.

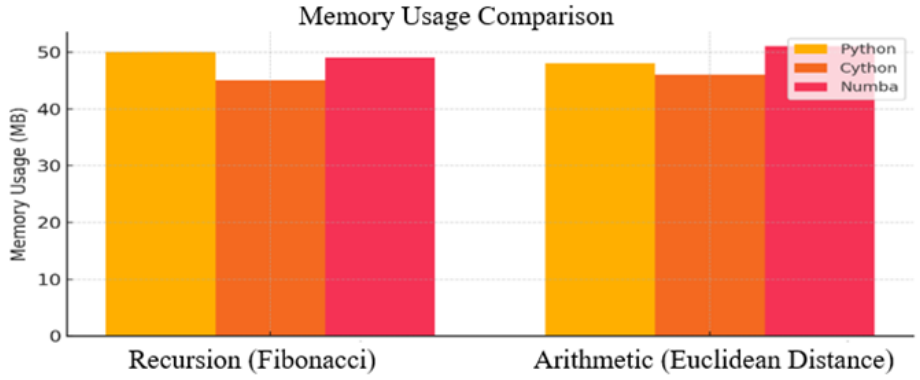

Figure 3: Memory usage comparison for Python, Cython, and Numba across tested workloads.

- **Compilation Overhead:** Numba introduces a slight runtime compilation delay of approximately 0.08 s on the first function call, whereas Cython's compilation is performed ahead of time, incurring no such runtime overhead.

Memory behavior under larger data volumes, iterative kernels, and long-running processes was not evaluated and warrants more detailed profiling in future work. Warm-up overhead varies with function complexity, as reported in Table 2. Speedup ratios reported in Table 1 are computed relative to CPython 3.10.9 with default interpreter settings, excluding the one-time JIT warm-up overhead of Numba, which is reported separately in Table 2.

Table 2: Numba warm-up overhead across workloads.

Workload Type	**First-Call Overhead (s)**	**Notes**
Fibonacci	0.09	Higher due to recursive call tracing
Euclidean Distance	0.07	Lower because of simple loop structure

Based on these findings, Table 3 summarizes the observed workload-to-compiler tendencies derived from the evaluated benchmarks.

Table 3: Workload-to-compiler recommendation.

Workload Type	**Recommended Compiler**	**Reason for Choice**	**Example Use Case**
Deep Recursion	Cython	Better static optimization and reduced call overhead	Symbolic math, tree-based algorithms
Arithmetic-Intensive Loops	Numba	Efficient NumPy array handling and vectorization	Machine learning preprocessing, GIS distance
GPU-Accelerated Workloads	Numba (CUDA)	Based on Numba's documented CUDA support; GPU execution was not evaluated in the present study	Scientific simulations, deep learning
Mixed CPU-bound Tasks	Hybrid (Cython + Numba)	Potential for selectively combining AOT and JIT compilation strategies (not empirically evaluated in this study)	Data analytics pipelines

3.3. Limitations of the Study

The workloads evaluated—recursive Fibonacci and arithmetic Euclidean distance—represent only a narrow subset of real-world patterns. Future studies should incorporate matrix operations, data-intensive pipelines, and mixed-mode workloads to improve representativeness. All experiments were conducted on a single Intel-based Windows 11 environment. Because JIT performance varies across CPU architectures, vectorization capabilities, and OS-level scheduling, broader cross-platform evaluation is needed.

4. Conclusion

The results of this study provide an evidence-based framework for selecting a compiler based on a task's computational characteristics (Table 3). For recursion-heavy tasks, Cython is recommended due to its superior performance and reduced function call overhead. For arithmetic-intensive tasks, Numba is the optimal choice, as its JIT optimizations are highly effective for numerical operations and NumPy array manipulation. This comparative analysis concludes that both Numba and Cython offer significant performance improvements over pure Python; however, their effectiveness is highly dependent on the nature of the workload. The field of JIT compilation for dynamic languages is rich with potential for further exploration. To better understand compiler performance, a standardized benchmark suite could be created to include a wider variety of tasks, such as matrix multiplication, image transformations, and large-scale sorting. The compatibility and performance of JIT compilers across different platforms, including Linux, macOS, and cloud GPU instances, should also be studied to assess their portability and optimization differences. A promising direction is the development of hybrid strategies that selectively combine compilers such as Cython and Numba to leverage their unique strengths for maximum performance. Future work could additionally focus on integrating JIT compilers into AI and machine learning workflows, testing their performance on preprocessing functions within frameworks such as TensorFlow and PyTorch. The security and stability of JIT compilers should also be analyzed, including potential runtime vulnerabilities or stability issues that may arise from dynamic code generation. Finally, an emerging area is the use of machine learning to create smarter JIT compilers; research could explore the benefits and limitations of these new compilers and compare them to traditional JIT approaches.

Author Contributions

Sattaru Harshavardhan Reddy: Methodology, Data Curation, Experiments and Software, Writing – Original Draft. **Priya Gupta**: Conceptualization, Writing – Review and Editing, Supervision. **Deepak Kumar**: Methodology, Data Curation. **Ritu Singhal**: Writing – Original Draft, Validation.

Declaration of Competing Interests

The authors declare no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data Availability Statement

All data generated or analyzed during this study are included in this article and its appendices. The benchmark source code is provided in full in Appendices A–C to enable complete reproducibility.

AI Disclosure Statement

During the preparation of this manuscript, the authors used ChatGPT and Google Gemini to improve language clarity and readability. All content was subsequently reviewed, edited, and verified by the authors, who take full responsibility for the integrity of the work.

Acknowledgment

The authors acknowledge the computational resources provided by their respective institutions.

Funding Declaration

This research did not receive any specific grant from funding agencies in the public, commercial, or not-for-profit sectors.

Ethics Approval and Consent

This study did not involve human participants, animal subjects, or personal data. Therefore, ethics approval and informed consent were not required.

References

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Pearson Education, 2nd ed., 2006.

[2] N. Smith, A. Sharma, J. Renner, D. Thien, F. Brown, H. Shacham, *et al.*, "Icarus: Trustworthy just-in-time compilers with symbolic meta-execution," in *Proc. ACM SIGOPS 30th Symp. Operating Systems Principles*, pp. 473–487, 2024.

[3] S. Tian, J. Huber, J. Tramm, B. Chapman, and J. Doerfert, "Just-in-time compilation and link-time optimization for OpenMP target offloading," in *Int. Workshop on OpenMP*, (Cham), pp. 145–158, Springer, 2022.

[4] C. Pichler, P. Li, R. Schatz, and H. Mössenböck, "Hybrid execution: Combining ahead-of-time and just-in-time compilation," in *Proc. 15th ACM SIGPLAN Int. Workshop on Virtual Machines and Intermediate Languages*, pp. 39–49, 2023.

[5] C. Pichler, P. Li, R. Schatz, and H. Mössenböck, "On automating hybrid execution of ahead-of-time and just-in-time compiled code," in *Proc. 16th ACM SIGPLAN Int. Workshop on Virtual Machines and Intermediate Languages*, pp. 1–11, 2024.

[6] W. Jakob, S. Speierer, N. Roussel, and D. Vicini, "Dr.Jit: A just-in-time compiler for differentiable rendering," *ACM Trans. Graph. (TOG)*, vol. 41, no. 4, pp. 1–19, 2022.

[7] H. Ning, B. Han, Z. Yang, K. Hao, M. Ma, C. Wang, *et al.*, "Exploring simple architecture of just-in-time compilation in databases," in *Asia-Pacific Web (APWeb) and Web-Age Inf. Manage. (WAIM) Joint Int. Conf. on Web and Big Data*, (Singapore), pp. 504–514, Springer, 2024.

[8] M. Ma, Z. Yang, K. Hao, L. Chen, C. Wang, and Y. Jin, "An empirical analysis of just-in-time compilation in modern databases," in *Australasian Database Conf.*, (Cham), pp. 227–240, Springer, 2023.

[9] F. Latifi, D. Leopoldseder, C. Wimmer, and H. Mössenböck, "CompGen: Generation of fast JIT compilers in a multi-language VM," in *Proc. 17th ACM SIGPLAN Int. Symp. on Dynamic Languages*, pp. 35–47, 2021.

[10] Q. Zhang, L. Xu, and B. Xu, "Python meets JIT compilers: A simple implementation and a comparative evaluation," *Software: Practice and Experience*, vol. 54, no. 2, pp. 225–256, 2024.

[11] E. Bauman, J. Duan, K. W. Hamlen, and Z. Lin, "Renewable just-in-time control-flow integrity," in *Proc. 26th Int. Symp. on Research in Attacks, Intrusions and Defenses*, pp. 580–594, 2023.

[12] S. K. Lam, A. Pitrou, and S. Seibert, "Numba: A LLVM-based Python JIT compiler," in *Proc. Second Workshop on the LLVM Compiler Infrastructure in HPC*, pp. 1–6, ACM, 2015.

[13] A. N. Ziogas, T. Ben-Nun, T. Schneider, and T. Hoefler, "Productivity, portability, performance: Data-centric Python," in *Proc. Int. Conf. for High Performance Computing, Networking, Storage and Analysis (SC '21)*, 2021.

[14] I. Osborne, P. Elmer, and J. Stark, "Awkward just-in-time (JIT) compilation: A developer's experience," *EPJ Web of Conferences*, vol. 295, p. 06003, 2024.

[15] P. Gupta, T. ManiKiran, M. Purushotham, L. J. Suriya, R. N. Venkata, and S. Nanda, "Efficient compiler design for a geometric shape domain-specific language: Emphasizing abstraction and optimization techniques," *EAI Endorsed Transactions on Scalable Information Systems*, vol. 11, no. 4, 2024.

[16] R. Kumar, K. C. Negi, N. K. Sharma, and P. Gupta, "Deep learning-driven compiler enhancements for efficient matrix multiplication," *Journal of Computers, Mechanical and Management*, vol. 3, no. 2, pp. 08–18, 2024.

[17] Numba Project, "Numba: A high performance Python compiler." https://numba.pydata.org/, 2018.

[18] Numba Project, "Numba documentation." https://numba.pydata.org/, 2019.

[19] B. Hackl, "Cython vs. Numba vs. Mojo: A comparison of different approaches to speed up Python language execution," in *Austrian-Slovenian HPC Meeting 2024 (ASHPC24)*, p. 45, 2024.

[20] P. Grover, "Speed up your algorithms part 2 — Numba." Medium, TDS Archive. https://medium.com/data-science/speed-up-your-algorithms-part-2-numba-293e554c5cc1, 2018.

[21] K. Derlatka, M. Manna, O. Bulenok, D. Zwicker, and S. Arabas, "Numba-MPI v1.0: Enabling MPI communication within Numba/LLVM JIT-compiled Python code," *SoftwareX*, vol. 28, p. 101897, 2024.

[22] D. Bajaj, U. Bharti, I. Gupta, P. Gupta, and A. Yadav, "GTMicro — microservice identification approach based on deep NLP transformer model for greenfield developments," *International Journal of Information Technology*, vol. 16, no. 5, pp. 2751–2761, 2024.

[23] J. F. M. Sánchez, Y. E. Gómez, C. E. M. Marín, and R. G. Crespo, "Performance evaluation of WFS service consumption with Python and Cython," in *Int. Conf. on Data Science and Network Engineering*, (Cham), pp. 278–290, Springer Nature Switzerland, 2025.

[24] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D. S. Seljebotn, and K. Smith, "Cython: The best of both worlds," *Computing in Science & Engineering*, vol. 13, no. 2, pp. 31–39, 2010.

A. Fibonacci Recursion Benchmark Code

The Fibonacci benchmark evaluates compiler behavior under deep recursion. A naïve recursive implementation without memoization is used intentionally to amplify function-call overhead and expose differences in compiler handling of deep recursion; consequently, the observed performance trends may not generalize to optimized recursive or iterative implementations. The complete Python implementation and a snippet of auto-generated C code from Cython are provided below.

A.1. Cython Setup File (`setup.py`)

```python
from setuptools import setup
from Cython.Build import cythonize

setup(ext_modules=cythonize("fib.py"))
```

A.2. Python Source Code to Be Cythonized (`fib.py`)

```python
import time

def fib(n):
    if n < 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n - 1) + fib(n - 2)

arrfc = []
i = 20

while i <= 40:
    start = time.time()
    fib(i)
    end = time.time()

    t = end - start
    arrfc.append(t)

    i += 2

print(arrfc)
```

A.3. Representative Snippet of Auto-Generated C Code (`fib.c`)

When `fib.py` is compiled with Cython, it generates a large C file (`fib.c`). A brief representative snippet is provided here to illustrate how Cython translates Python code into CPython API calls. The full file is omitted due to its length.

```c
static void __Pyx_AddTraceback(const char *funcname, int c_line,
                               int py_line, const char *filename)
{
    PyCodeObject *py_code = 0;
    PyFrameObject *py_frame = 0;
    PyThreadState *tstate = __Pyx_PyThreadState_Current;
    PyObject *ptype, *pvalue, *ptraceback;

    if (c_line) {
        c_line = __Pyx_CLineForTraceback(tstate, c_line);
    }

    py_code = __pyx_find_code_object(c_line ? -c_line : py_line);
    if (!py_code) {
        __Pyx_ErrFetchInState(tstate, &ptype, &pvalue, &ptraceback);
        py_code = __Pyx_CreateCodeObjectForTraceback(
                        funcname, c_line, py_line, filename);

        if (!py_code) {
            Py_XDECREF(ptype);
            Py_XDECREF(pvalue);
            Py_XDECREF(ptraceback);
            goto bad;
        }

        __Pyx_ErrRestoreInState(tstate, ptype, pvalue, ptraceback);
        __pyx_insert_code_object(c_line ? -c_line : py_line, py_code);
    }

    py_frame = PyFrame_New(
                    tstate,
                    py_code,
                    __pyx_d,
                    0
                );
}
```

B. Euclidean Distance Benchmark Code

The Euclidean distance benchmark evaluates arithmetic-intensive performance. The Numba-compiled implementation is provided below.

B.1. Numba-Accelerated Euclidean Distance Function

```python
from numba import jit
import random
import math
import time

@jit(nopython=True)
def euclid(n):
    z = 0
    for i in range(n):
        x = random.random()
        y = random.random()
        z += math.sqrt(x**2 + y**2)
    return z
```

B.2. Benchmark Loop Used for Performance Measurement

This block measures execution time for increasing input sizes.

```
results = []
sizes = [100000, 200000, 500000, 1000000]

for n in sizes:
    start = time.time()
    euclid(n)
    end = time.time()

    t = end - start
    results.append(t)

print(results)
```

The use of random floating-point values ensures that the workload remains computation-bound. Numba compiles the function on the first call, after which subsequent calls benefit from optimized machine code execution. The Euclidean distance kernel was successfully compiled in nopython mode, and no Python object fallback occurred; the calls to `random.random()` were fully lowered by Numba during JIT compilation, ensuring that the measured execution reflects optimized native code rather than Python-level function calls.

C. Experimental Environment and Reproducibility Details

To ensure complete reproducibility, the hardware and software environment, as well as the execution protocol used for all experiments,s are summarized below:

- **Processor:** Intel Core i7-10750H @ 2.60 GHz
- **RAM:** 16 GB DDR4
- **Operating System:** Windows 11 Pro 64-bit
- **Python Version:** 3.10.9
- **Cython Version:** 0.29.34
- **Numba Version:** 0.57.0
- **Memory Profiler:** Latest stable release (invoked via `%memit`)
- Warm-up overhead for Numba was profiled separately.
- Error bars as shown in the figures represent standard deviation (SD < 0.06 s).
- Auto-generated C files from Cython were not modified manually.