# Python Polybench Acceleration Using Compiler Techniques

Fatemeh Derakhshani
*McMaster University*
derakhsf@mcmaster.ca

Dr. Kazem Cheshmi
*McMaster University*
cheshmi@mcmaster.ca

***Abstract*—In the realm of high-performance computing, the quest for efficient computational methods remains a critical challenge. This study addresses the performance potential of Python, a language traditionally not synonymous with high computational efficiency, in the context of the Polyhedral Benchmark (Polybench) suite. We explore the application of Numba, a Just-In-Time (JIT) compiler, to optimize Python code through parallelization, vectorization and loop unrolling, and transformation techniques. Our methodology involves the implementation of selected Polybench benchmarks in Python, followed by their optimization using Numba's features. We systematically measure and compare the performance of the optimized Python code against its unoptimized counterparts and Corresponding C++ implementations, focusing on execution time and computational efficiency. The results indicate significant performance improvements, highlighting the effectiveness of Numba in bridging the gap between Python's ease of use and the demands of high-performance computing. This study not only showcases the feasibility of Python as a viable option in performance-critical applications but also contributes to the growing body of knowledge in Python-based scientific computing. Our findings open avenues for further research in optimizing Python for high-performance tasks, potentially expanding its adoption in computational-intensive domains.**

***Index Terms*—High-Performance-Computing, Polybench, Numba, Vectorization, Loop-Unrolling**

## I. Introduction

In the landscape of scientific and high-performance computing, efficiency and speed are paramount. While languages like C and Fortran have traditionally dominated this realm due to their low-level control and efficiency, Python has emerged as a popular language owing to its simplicity and readability. However, Python's interpreted nature often leads to performance bottlenecks, making it less suitable for computationally intensive tasks [1].

Polyhedral Benchmark (Polybench) is a collection of benchmarks widely used to evaluate the performance of programming languages and compilers in the context of numerical computations commonly found in scientific applications [2]. These benchmarks provide a comprehensive platform for assessing the efficiency of various optimization techniques.

Numba, a Just-In-Time (JIT) compiler for Python, presents an intriguing solution to Python's performance challenges. By converting Python functions to optimized machine code at runtime, Numba offers significant speedups [3]. Particularly, Numba's support for vectorization holds the potential to substantially enhance computational efficiency. Vectorization allows for the execution of operations on entire arrays rather than individual elements, exploiting modern processor capabilities. Similarly, we also implement loop unrolling as an optimization method which reduces the overhead of loop control and increases the granularity of operations.

This study aims to bridge the gap between Python's user-friendly syntax and the high-performance requirements of scientific computing. We investigate the application of parallelization, vectorization, loop unrolling (as well as different combinations of them), and transformation features to Python implementations of Polybench benchmarks. We aim to quantify the performance gains and assess whether these optimizations make Python a more viable candidate for high-performance computing tasks.

The rest of the paper is organized as follows: The background section provides an overview of Python in scientific computing, Polybench, and the Numba compiler. The methodology section details the implementation and optimization process. The results sections present and analyze the findings. Finally, we conclude with a summary of the study and potential directions for future research and related works.

## II. Background

**Polybench:** The Polyhedral Benchmark suite, commonly known as Polybench, is a collection of benchmarks designed for evaluating the performance of programming languages and compilers in the domain of numerical computations [2]. These benchmarks cover a wide range of computational patterns typical in scientific applications, providing a standard framework for assessing optimizations and enhancements in computational efficiency. Polybench is particularly relevant for studies aiming to enhance the performance of high-level languages in scientific computing. For this paper we have chosen three blas (basic linear algebra subprograms) benchmarks from Polybench which are as follows:

*1) Symm::* The Symm kernel performs a symmetric matrix-matrix multiplication. Given matrices $C$, $A$, and $B$ of dimensions $m \times n$, $m \times m$, and $m \times n$ respectively, and scalar coefficients $\alpha$ and $\beta$, the computation can be expressed as follows:

```
For $i = 0$ to $m - 1$ :
  For $j = 0$ to $n - 1$ :
    For $k = 0$ to $i - 1$ :
      $C[k][j] \mathrel{+}= \alpha \cdot B[i][j] \cdot A[i][k]$
      Update temporary variable temp2 $\mathrel{+}= B[k][j] \cdot A[i][k]$
  For $j = 0$ to $n - 1$ :
    $C[i][j] = \beta \cdot C[i][j] + \alpha \cdot B[i][j] \cdot A[i][i] + \alpha \cdot \text{temp2}$
```

*2) Syrk::* The Syrk kernel computes the symmetric rank-$k$ update. Given matrices $C$ and $A$ of dimensions $n \times n$ and $n \times m$ respectively, and scalar coefficients $\alpha$ and $\beta$, the computation can be expressed as follows:

```
For $i = 0$ to $n - 1$ :
  For $j = 0$ to $i$ :
    $C[i][j] *= \beta$
    For $k = 0$ to $m - 1$ :
      $C[i][j] \mathrel{+}= \alpha \cdot A[i][k] \cdot A[j][k]$
```

*3) Trmm::* The Trmm kernel performs triangular matrix multiplication. Given matrices $A$ and $B$ of dimensions $m \times m$ and $m \times n$ respectively, and scalar coefficient $\alpha$, the computation can be expressed as follows:

```
For $i = 0$ to $m - 1$ :
  For $j = 0$ to $n - 1$ :
    For $k = i + 1$ to $m - 1$ :
      $B[i][j] \mathrel{+}= A[k][i] \cdot B[k][j]$
    $B[i][j] = \alpha \cdot B[i][j]$
```

These kernels represent fundamental linear algebra operations, crucial in various numerical and scientific computing applications.

**Python and Performance in Scientific Computing:** Python's emergence as a major force in scientific computing is largely due to its simplicity, readability, and an extensive ecosystem of scientific libraries (e.g., NumPy, SciPy) [4]. However, the inherent performance limitations of Python, primarily due to its interpreted nature and dynamic typing, pose significant challenges in high-performance computing environments [5]. These challenges often necessitate the integration of Python with lower-level languages or the use of specialized tools to improve its computational efficiency.

**Numba:** Numba is an open-source JIT compiler that translates a subset of Python and NumPy code into fast machine code [3]. Numba is specifically designed for numerical functions and offers significant speedups by compiling Python code to optimize machine code at runtime. One of the key features of Numba is its ability to generate vectorized code and loop parallelization, optimizations that are beneficial for achieving high performance in computational tasks.

**Vectorization and Loop Unrolling:** Vectorization refers to the process of executing operations on entire arrays or vectors simultaneously, as opposed to performing element-wise operations. This approach is highly effective in exploiting the data parallelism capabilities (SIMD) of modern CPUs and GPUs [6]. Loop unrolling, on the other hand, involves duplicating the body of a loop multiple times in a sequence, reducing the loop overhead and improving cache performance. Both techniques are standard optimizations in high-performance computing, known for their ability to significantly boost computational efficiency.

## III. Motivation

The motivation behind this study stems from the growing intersection of Python's accessibility and the escalating demands of high-performance computing (HPC) in scientific research. While Python's ease of use and extensive library ecosystem has made it a language of choice for scientists and engineers, its performance limitations are a significant bottleneck in HPC applications [5]. This study is motivated by the potential to bridge this gap, leveraging Python's user-friendly nature while enhancing its computational efficiency.

**Python in Scientific Computing** Python's ascendance as a prominent tool in scientific computing is undeniable. Its readability, simplicity, and the rich array of scientific libraries (e.g., NumPy, SciPy) have democratized access to advanced computational tools [4]. However, Python's interpreted nature poses performance challenges, particularly in computationally intensive tasks that are common in scientific research. Besides, the increasing complexity and scale of scientific problems necessitate the use of high-performance computing resources. Traditionally, languages like C, C++, and Fortran have been preferred in HPC due to their efficiency and speed. Python's slower execution time, as a result of its high-level abstraction, poses a significant challenge when dealing with large-scale scientific computations. However, The development of Numba represents a significant step towards addressing Python's performance limitations [3]. By providing a JIT compiler that translates Python functions into optimized machine code, Numba opens up new possibilities for enhancing Python's performance in scientific computing.

**Polybench as a Benchmarking Tool**: The use of standard benchmarks like Polybench is crucial in evaluating and demonstrating the effectiveness of optimization techniques [2]. Polybench, with its suite of benchmarks designed for various computational patterns, provides an ideal platform for assessing the impact of performance optimizations in a controlled and systematic manner.

While there have been studies on optimizing Python code, there is a lack of comprehensive research focused on applying both vectorization and loop unrolling techniques using Numba in the context of Polybench benchmarks. This study aims to fill this gap by providing empirical insights into the effectiveness of these optimization strategies, contributing to the broader effort of making Python more viable for HPC applications.

## IV. Methodology

This study aims to evaluate the performance enhancements achieved by applying Numba's vectorization, parallelization, and loop unrolling features to Python implementations of specific Polybench benchmarks, with a focus on the three selected benchmarks from Polybench.

**Python Implementation and Optimization:** Among the various benchmarks available in Polybench, we selected the symm, trmm, and syrk which are linear algebra benchmarks for our study. These benchmarks are representative of typical computational patterns found in scientific computing, particularly those involving matrix operations. We have implemented these benchmarks in Python, adhering to the algorithmic structure provided by Polybench. This implementation serves as the baseline for our performance comparisons. The baseline Python implementation of the selected benchmarks was then optimized using Numba's JIT compiler. Three primary optimization techniques were employed:

### *A. Parallelization:*

In our quest for performance enhancement, we integrated parallelization techniques across our benchmarks. A parallelized version has been implemented for all selected benchmarks using a consistent methodology, extendable to other benchmarks as well. The parallelization is facilitated through the application of Numba's `@njit(parallel=True)` decorator. This strategic optimization enables the concurrent execution of specific operations, marked by the utilization of `prange` instead of `range` in for loops within the kernel functions. The result is a harnessing of the potent capabilities of parallel processing, contributing to significant performance gains.

### *B. Vectorization:*

To further augment performance, we employed vectorization. We defined a general vectorization function using Numba's `@vectorize` decorator. This function, exemplified by `vectorized_inner_loop`, compels the compiler to utilize Single-Instruction-Multiple-Data (SIMD) operations. Subsequently, we integrated this vectorization function to transform specific operations within the kernel functions of our benchmarks into accelerated, vectorized operations, maximizing the utilization of SIMD.

### *C. Loop Unrolling:*

The innermost loop of the kernel functions of the selected benchmarks was manually unrolled. The unrolling factor was chosen based on preliminary tests to determine the most effective unrolling degree. The unrolled version reduces loop control overhead and potentially improves cache utilization.

### *D. Combination:*

Drawing insights from the outcomes of applying the aforementioned optimizations to the test cases for each kernel function, we introduced two composite optimization methods (implemented as functions) to further refine the efficiency of our solutions. The initial combination integrates parallelization and loop unrolling, denoted as 'punroll' in codes and results. The second combination amalgamates all three optimizations (referenced as 'combined'). The implementation of these composite methods involves an amalgamation of the respective techniques, resulting in more nuanced and effective strategies for enhancing overall performance which will be discussed in the Conclusion section.

### *E. Automization:*

Despite its prowess in Just-In-Time (JIT) compilation for Python, Numba encounters challenges when it comes to automated parallelization, vectorization, and loop unrolling. The following elucidates these hurdles and delves into the limitations faced when attempting to implement user-defined run passes:

*Parallelization:* Numba primarily focuses on loop-level parallelism through the '@njit(parallel=True)' decorator. However, the absence of user-defined function passes for more intricate parallelization strategies hampers the automation of parallel constructs beyond loop structures. Advanced parallelization techniques may necessitate manual intervention or alternative methods.

*Vectorization:* in Numba confronts challenges due to the lack of direct support for user-defined function passes. In our experimentation, we implemented a vectorization pass encapsulated in the 'VectorizationPass' class. However, the intricacies arise from Numba's requirement for the 'vectorize' function to be Numba-compiled, adding complexity to the integration of vectorized functions into Intermediate Representation (IR) statements.

For this project, we opted for a workaround, utilizing the is_vectorizable function from the VectorizationPass class. This function checks for vectorizable IR statements within the code. While this approach allows the identification of vectorizable operations, seamlessly integrating automated vectorization into Numba's existing compilation pipeline presents challenges. The necessity for further refinement in the integration process underscores an area where Numba's functionality might benefit from additional flexibility.

*Loop Unrolling:* The loop unrolling capabilities within Numba's JIT compilation are not directly controllable by users. While Numba performs automatic loop optimizations, achieving precise control over loop unrolling factors requires substruction changes in Numba's predefined optimization pipeline.

User-Implemented Run Passes: In a bid to overcome these limitations, users may consider crafting custom run passes. The provided VectorizationPass is an attempt to identify and vectorize specific operations. However, the intricacies of adhering to Numba's compilation requirements may pose difficulties in applying this pass directly. Furthermore, the Dead Code Elimination pass, as an extra experiment has been added for each of the selected benchmarks to have more opportunities for comparison of our approaches.

To summarize, Numba stands out as an efficient Just-In-Time (JIT) compiler; however, the automation of specific optimizations and the introduction of user-defined run passes face challenges due to a lack of robust infrastructure. Exploring alternative solutions is recommended, and keeping an eye on future updates and additional features in Numba's 'Core' is advisable. The mentioned limitations pinpoint areas where the tool could benefit from further enhancements to accommodate advanced optimization strategies effectively.

### *F. C++:*

To facilitate comparison, we utilize both the original C++ kernels and our parallelized C++ implementations for each selected benchmark. The parallelization approach mirrors that of the Python implementation, employing OpenMP. The experiments in C++ are executed through a Python wrapper that utilizes the subprocess module. This approach allows seamless integration and consistent execution environments across both Python and C++ experiments.

**Performance Measurement:** To evaluate the impact of these optimizations, both the baseline and optimized versions of our benchmarks underwent testing across a range of matrix sizes, providing insights into their performance across different computational scales. Execution time was measured, with each version executed multiple times to account for runtime variations. The experiments were conducted on the Niagara server, ensuring a consistent testing environment for fair and accurate comparisons. Execution times of baseline and optimized versions were analyzed to quantify the performance gains achieved through parallelization, vectorization, loop unrolling, combined experiments, and the dead code eliminated transformed version and compare these to the naive version of Python and the above-mentioned C++ implementations. Additional scrutiny was applied to understand how these improvements vary concerning different matrix sizes, offering valuable insights into the scalability of our optimizations which we will discuss in the Conclusion section.

## V. Results

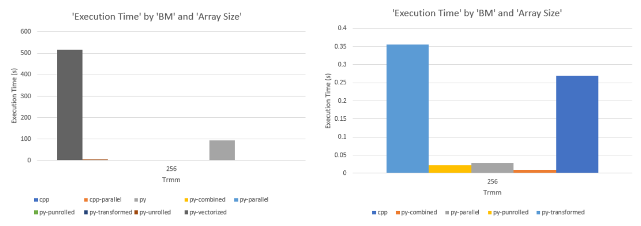

Fig. 1: Results of Trmm Experiments (Left Chart: Overview of All Discussed Experiments, Right Chart: Zoomed View Highlighting Optimal Setups) - Input Matrix Size: 256

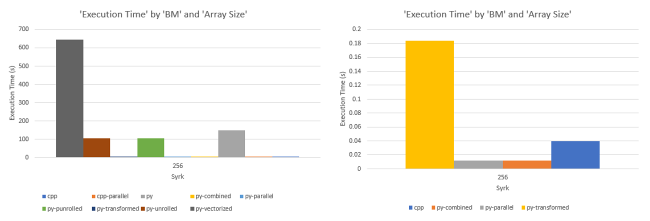

Fig. 2: Results of Syrk Experiments (Left Chart: Overview of All Discussed Experiments, Right Chart: Zoomed View Highlighting Optimal Setups) - Input Matrix Size: 256

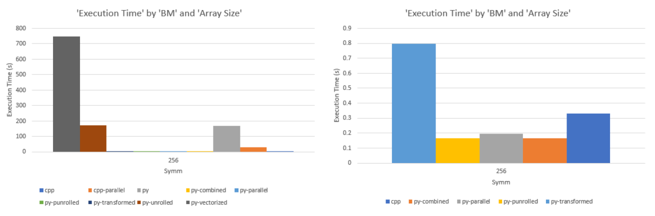

Fig. 3: Results of Symm Experiments (Left Chart: Overview of All Discussed Experiments, Right Chart: Zoomed View Highlighting Optimal Setups) - Input Matrix Size: 256

This section provides a detailed account of the outcomes stemming from our methodical application of Numba's transformation, vectorization, parallelization, and loop unrolling features to Python implementations of specific Polybench benchmarks. The primary focus is on the symm, trmm, and syrk benchmarks, and the results are structured to draw meaningful comparisons between the baseline Python implementation and its optimized counterparts. We will also compare these with baseline and parallelized C++ implementation.

**Specifications:** To obtain the results, we conducted experiments on the Niagara server. For these experiments, three essential modules need to be loaded on Niagara, as outlined below: intel/2019u3, python/3.11.5, cmake/3.21.4 (We need to ensure that Numba is installed on the server if it's not already).

**Performance Comparison:**

*1) Execution Time:* Figures 1, 2, and 3 present the experimental results for Trmm, Syrk, and Symm, respectively. Each figure comprises two charts: the left chart provides an overview of all discussed experiments in the methodology section, encompassing C++ naive and parallelized versions, Python naive, parallelized, unrolled, vectorized, 'punrolled' (parallelized-unrolled), combined (parallelized-vectorized-unrolled), and transformed (dead code eliminated) versions. the right chart contains only the ones that have smaller execution times and can't be understood vividly from the left chart. The input matrix size for these experiments is fixed at 256x256, although results for 128x128 and 512x512 matrices were also obtained, forming the basis of our observations.

Parallelization Discrepancy Between C++ and Python: The parallelized version of C++ exhibits comparable or even inferior results to its naive counterpart. In contrast, parallelization in Python shows performance improvements. Several factors contribute to this, including overhead, memory access patterns, synchronization mechanisms, compiler optimizations, and differences in underlying parallel libraries.

Efficacy of Combined Optimization Method: The combined optimization method, incorporating parallelization, vectorization, and loop unrolling, emerges as a consistently effective strategy. While vectorization and unrolling individually may not yield significant improvements, their synergistic integration in the combined method produces superior results. The 'punrolled' method, is specifically designed to explore the impact of excluding vectorization, which leads to the significance of the efficiency of combining optimization techniques including vectorization.

Transformation Method with Dead Code Elimination: The transformation method, emphasizing dead code elimination using the Numba Core library, demonstrates promising results. While slightly trailing the combined method in most cases, it offers competitive performance, particularly in one experiment where it excels. This observation shows the potential benefits of targeted transformations.

Overall Insights: The combined method emerges as the optimal choice in most scenarios, closely followed by the transformation method in specific cases. Notably, Python parallelization, the 'punrolled' method, and the transformation method showcase commendable results, providing viable alternatives when combined method optimization is not an option.

Key Takeaway: Despite Python's naive implementation initially lagging behind C++, the judicious application of optimization methods, notably parallelization, transforms Python's execution time to outperform even the C++ (naive) counterpart. This emphasizes the significance of effective optimization strategies in achieving competitive performance across languages.

*2) Speedup Factor:* We can compute two speedup factors in this analysis: one comparing the Python naive version to the combined method and another comparing the C++ naive version to the Python combined method. The speedup factor between the Python naive and Python combined versions exhibits dependency on the input size. In our experiments, this factor ranges approximately between 10 and 1000 times, calculated as (Python naive - Combined method) / Python naive. Surprisingly, we also observe a speedup between the naive C++ version and the combined method, amounting to around 50%, calculated as ((C++ naive - Combined method) / C++) * 100. This unexpected result underscores the efficacy of Python optimizations, even surpassing the performance of the original C++ implementation.

**Scalability Analysis:** As discussed, we conducted experiments across various input sizes, specifically 128x128, 256x256, and 512x512. Figure 4 illustrates the execution time of the 'Symm' benchmark for different input sizes, focusing on the combined method and the Python naive version. The increasing dominance of the combined method with larger input sizes is notable. This can be attributed to the growing visibility of the optimization methods' effectiveness as the input size increases. The combined method demonstrates superior performance, showcasing its ability to handle larger computational workloads more efficiently.

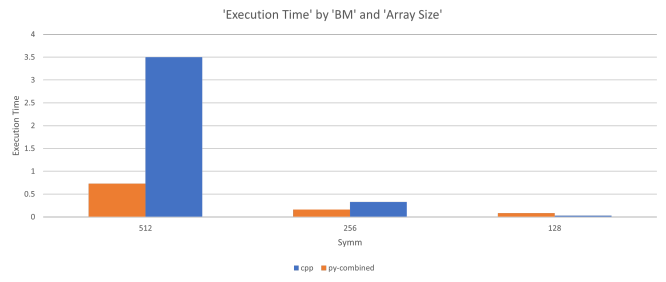

Fig. 4: A comparative analysis between the Python naive version and the combined version across various input sizes.

## VI. Conclusion

In this study, we embarked on a journey to explore the performance potential of Python, a language known for its simplicity and readability, in the domain of high-performance computing. Focusing on the Polyhedral Benchmark (Polybench) suite, we employed Numba, a Just-In-Time (JIT) compiler, to optimize Python code through parallelization, vectorization, loop unrolling, and transformation techniques.

Our systematic implementation and optimization of selected Polybench benchmarks, including Symm, Trmm, and Syrk, provided valuable insights into the efficiency of Python in comparison to traditional languages like C++. The results showcase significant performance improvements, emphasizing the effectiveness of Numba in narrowing the gap between Python's ease of use and the demands of high-performance computing.

Key observations include the unexpected performance gains in Python parallelization compared to its C++ counterpart. The combined optimization method, integrating parallelization, vectorization, and loop unrolling, emerged as a consistently effective strategy, outperforming individual optimization techniques. The transformation method with dead code elimination also demonstrated competitive performance, providing a viable alternative.

Our findings underscore the feasibility of Python as a viable option in performance-critical applications, particularly in scientific computing. Despite Python's inherent performance limitations, the judicious application of optimization methods, as demonstrated in this study, positions Python as a competitive language for high-performance tasks.

This study contributes to the growing body of knowledge in Python-based scientific computing and opens avenues for future research. Further exploration of optimization strategies, automated optimization techniques, and enhanced support for user-defined run passes in Numba could lead to even more robust solutions. The scalability analysis highlights the potential of Python in handling larger computational workloads efficiently, paving the way for its expanded adoption in computational-intensive domains.

## VII. Future Work

While our study provides valuable insights into Python's optimization for high-performance computing, there are avenues for future research to explore. Enhancements in Numba's functionality (or other tools), especially in automated parallelization, vectorization, and loop unrolling, would contribute to a more seamless optimization process. Additionally, further investigation into the impact of optimization techniques on a broader range of benchmarks and applications could provide a comprehensive understanding of Python's capabilities in high-performance computing.

## VIII. Related Works

The quest to enhance the performance of high-level programming languages in scientific computing has been a topic of significant interest in the research community. This section reviews some of the key studies that have explored various aspects of this challenge, particularly focusing on Python and its optimization through tools like Numba.

**Python in High-Performance Computing:** Several studies have highlighted Python's growing role in scientific computing, despite its inherent performance limitations compared to languages like C and Fortran. Millman and Aivazis [5] discuss Python's applicability in scientific contexts, emphasizing the need for performance optimizations to widen its usage in computationally intensive tasks.

**Optimization Techniques in Python:** The use of Just-In-Time (JIT) compilation as a means to optimize Python code has been the subject of extensive research. we provide an early look at the benefits of JIT compilation in improving Python's performance. More recent works by Lam et al. [3] delve into the specifics of Numba, demonstrating its effectiveness in compiling Python code to optimized machine code.

**Vectorization and Loop Unrolling:** The impact of vectorization and loop unrolling on performance has been explored in various studies. Frigo and Johnson [6] highlight the advantages of vectorized operations in exploiting data parallelism, while we discuss the benefits of loop unrolling in improving cache performance and reducing loop control overhead.

**Case Studies and Benchmarks:** Specific case studies using benchmarks like Polybench have been instrumental in demonstrating the practical implications of these optimizations. Pouchet [2] introduces Polybench as a comprehensive suite for evaluating compiler and language performance in numerical computations. Studies using Polybench often reveal insightful trends about the effectiveness of different optimization techniques across various computational patterns. We also have Polly [7], a project integrated into LLVM to enable polyhedral optimizations in a language-independent and transparent manner, the current study delves into optimizing Python programs using the Numba JIT compiler. Polly, similar to Numba, aims to automatically detect and transform relevant program parts, supporting various programming languages and constructs. However, Polly primarily targets LLVM-IR, ensuring language independence and allowing the optimization of code with features like C++ iterators and pointer arithmetic. In contrast, the current work concentrates on Python's numerical kernels and demonstrates performance improvements through parallelization, vectorization, and loop unrolling. Another relevant paper is PolyBench/Python [8] which introduces a benchmarking suite mirroring PolyBench/C, aiming to evaluate Python's performance in numerical kernels. In contrast, the current study explores Python's performance by optimizing Polybench benchmarks using the Numba JIT compiler.

**Comparative Studies:** Comparative studies of Python with other high-performance languages provide a broader perspective on its position in scientific computing. Behnel et al. [9] compare Python with Cython, discussing how the latter can bridge the gap between Python's ease of use and the performance of lower-level languages. Bolz et al. [10] offer insights into how different JIT compilers, including PyPy, impact Python's execution speed.

## IX. Artifact Evaluation

In order to recreate the graphs and delve into the specifics of methods applied across all three benchmarks, I've included both the C++ and Python code for each benchmark. Furthermore, I've provided three scripts tailored for the Niagara Server, each corresponding to a specific array size. These scripts facilitate the reproduction of experimental results. For each script, I've incorporated the output from Niagara, and to streamline analysis, an Excel file has been included. This file encompasses the table derived from the Niagara result files, along with a Pivot table and Pivot chart essential for generating the graphs. It's noteworthy that all graphs have been generated from the same Pivot table, leveraging various filtrations for distinct visualizations. The Excel file also includes a separate sheet that contains a summary of the main table; I have used this sheet to exploit the speed-up factor. (The evaluation file has been also included)

## References

[1] T. E. Oliphant, "Python for scientific computing," *Computing in Science Engineering*, vol. 9, no. 3, pp. 10–20, 2007.

[2] L.-N. Pouchet. (2012) Polybench benchmarks. [Online]. [Online]. Available: https://www.cs.colostate.edu/ pouchet/software/polybench/

[3] S. K. Lam, A. Pitrou, and S. Seibert, "Numba: A llvm-based python jit compiler," in *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, ser. LLVM '15. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: https://doi.org/10.1145/2833157.2833162

[4] F. Perez and B. E. Granger, "Ipython: A system for interactive scientific computing," *Computing in Science Engineering*, vol. 9, no. 3, pp. 21–29, 2007.

[5] K. J. Millman and M. Aivazis, "Python for scientists and engineers," *Computing in Science Engineering*, vol. 13, no. 2, pp. 9–12, 2011.

[6] M. Frigo and S. Johnson, "The design and implementation of fftw3," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005.

[7] L.-N. Pouchet, A. Größlinger, A. Simbürger, H. Zheng, and T. Grosser, "Polly-polyhedral optimization in llvm," vol. 2011, 01 2011.

[8] M. A. Abella-González, P. Carollo-Fernández, L.-N. Pouchet, F. Rastello, and G. Rodríguez, “Polybench/python: Benchmarking python environments with polyhedral optimizations,” in *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction*, ser. CC 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 59–70. [Online]. Available: https://doi.org/10.1145/3446804.3446842

[9] 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, 2011.

[10] C. F. Bolz, A. Cuni, M. Fijalkowski, and A. Rigo, “Tracing the meta-level: Pypy’s tracing jit compiler,” in *Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, ser. ICOOOLPS ’09. New York, NY, USA: Association for Computing Machinery, 2009, p. 18–25. [Online]. Available: https://doi.org/10.1145/1565824.1565827