On the Landscape of Scientific Computing Libraries in Python

Niteya Shah
Dept. of Computer Science
Virginia Tech
Blacksburg, Virginia, USA
niteya@vt.edu

Pi-Yueh Chuang
Mathematics & Computer Science
Argonne National Laboratory
Lemont, IL, USA
pchuang@anl.gov

Paul Sathre
Dept. of Computer Science
Virginia Tech
Blacksburg, Virginia, USA
sath6220@cs.vt.edu

Wu-chun Feng
Dept. of Computer Science
Virginia Tech
Blacksburg, Virginia, USA
wfeng@vt.edu

2025 IEEE High Performance Extreme Computing Conference (HPEC) | 979-8-3315-7844-2/25/$31.00 ©2025 IEEE | DOI: 10.1109/HPEC67600.2025.11196385

***Abstract*—Python has seen large-scale adoption as a highly productive language for scientific computing, primarily due to its rich ecosystem of libraries such as NumPy, PyTorch, and TensorFlow. These libraries claim to deliver scalable and portable performance without the low-level complexities associated with traditional high-performance compiled languages such as C, C++, and Fortran. However, they are predominantly designed and optimized for machine learning workloads.**

This work quantifies and characterizes the performance, productivity, and memory efficiency of these libraries for using four scientific computing workloads from the field of computational physics. In addition, we examine the influence of features offered by these libraries, including auto-parallelization, vectorization, and just-in-time (JIT) compilation. Using NumPy as the baseline and C++ as the performance upper bound, we analyze the compute and memory characteristics of each library, along with the associated runtime overhead within the Python ecosystem for solving these problems. Such a characterization enables an accurate quantification of the productivity-performance tradeoffs among the different libraries, both relative to each other and to C++. Finally, we leverage these insights to propose guidelines that can assist programmers and scientists in selecting the most suitable libraries for their work.

***Index Terms*—Python, NumPy, PyTorch, TensorFlow, Numba, Jax, C++, performance, memory utilization, top-down micro-architecture analysis, speedup, parallelism, JIT, vectorization, tensor operations**

I. Introduction

Python's ease of programmability has enabled the Python ecosystem to become widely adopted in the scientific computing community. Much of its success is due to NumPy, which provides accelerated linear algebra operations via an intuitive and productive interface. Building on this foundation, machine learning (ML) libraries, e.g., PyTorch, TensorFlow, and JAX, have emerged, offering not only the familiar NumPy API but also advanced features such as automatic parallelization over the multiple cores of a CPU, offloading work onto a GPU(s), automatic differentiation, and JIT compilation strategies to further reduce computational overhead. However, previous work in this field has focused on deep learning-based workloads, which, due to the prevalence of GEMM operations, lie deeply in the compute-bound region of the roofline model [1], [2], [3].

In this paper, we quantify, analyze, and characterize the productivity and performance of prominent scientific computing libraries using four mini-applications from computational physics: two from quantum chromodynamics (QCD) — a proxy theory and Duke & Owens — and two from physics simulations — lid-driven cavity, implemented using stencil-like operations, and n-body molecular simulation to capture dense pairwise computation. Specifically, we compare and contrast the productivity and performance of major scientific computing libraries in Python, using NumPy as our baseline for both performance and productivity. Performance is measured in floating-point operations per second (FLOPS) while productivity is measured using the commonly adopted metric of *source lines of code (SLOC)*. In addition, we investigate the performance of these libraries when employed in an unconventional manner — specifically, with no broadcasting — to assess how well each library adapts when the problem does not exhibit data-level parallelism.

Subsequently, this work seeks to provide guidance for end users considering advanced features such as vectorization, auto-parallelization, and JIT compilation, as well as to offer an overall performance characterization of major scientific computing libraries within this ecosystem. Our findings indicate that because productivity across these libraries is nearly identical, the feature set related to performance and memory efficiency become the key factors in selecting a library. However, both performance and memory efficiency are highly dependent on the algorithm and execution strategy employed.

The main contributions of this work are as follows:

- A characterization of the most suitable libraries and their associated features in the Python ecosystem, as shown in Figure 1, and categorized according to whether the algorithm exhibits data-level parallelism, as shown in Table I.
- Identification of issues and limitations present in these scientific computing libraries.
- A decision tree designed to guide users and developers in selecting the appropriate library for their specific problem.

This work was supported by the U.S. Department of Energy's Scientific Discovery through Advanced Computing (SciDAC) Program within the Office of Science via a partnership across the Office of Advanced Scientific Computing Research (ASCR) and Office of Nuclear Physics (NP) and the FASTMath Institute under the following contracts: DE-SC0023472 and DE-AC02-06CH11357. With respect to computational resources, this work was supported by Advanced Research Computing at Virginia Tech.

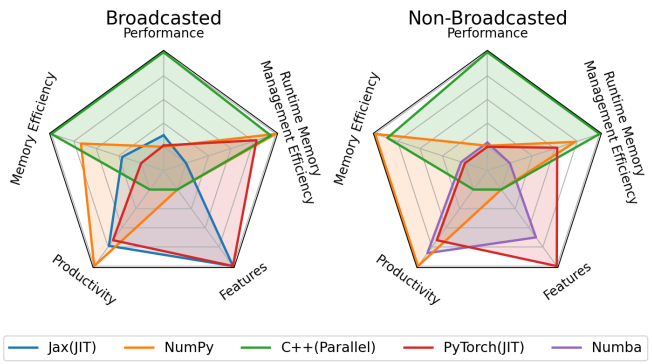


Fig. 1: Guidance radar plot for the different scientific computing libraries. Broadcasting enables vectorization for array arithmetic and vice versa for non-broadcasted. Productivity is measured as a geomean of SLoC normalized to NumPy. Memory efficiency is a measure of peak memory utilization normalized to NumPy for the same workload. Features is a enumeration of how many features the library supports from Table I

Library	JIT Compilation	Automatic Differentiation	Auto Parallelization	GPU Offloading
NumPy	No	No	No	No
DPNP	No	No	Yes	Yes
Numba	Yes	No	Yes	Limited
PyTorch	Yes	Yes	Yes	Yes
Tensorflow	Yes	Yes	Yes	Yes
Jax	Yes	Yes	Yes	Yes

TABLE I: Library Features

II. Background

In this section, we first provide background on the scientific computing ecosystem. Then, we discuss the workloads and profiling strategies used.

A. Python and the Scientific Computing Ecosystem

1) N-dimensional Arrays and Broadcasting in Python: A key feature of Python's scientific computing libraries is the N-dimensional array or *ndarray*. The *ndarray* provides a performant and productive abstraction for performing vector math by wrapping over BLAS libraries or highly optimized libraries written in C/C++/Fortran.

Broadcasting is the term used by NumPy to describe the way in which it treats arithmetic operations between arrays of different sizes by "broadcasting" both arrays to compatible shapes, as shown in the example in Table II. Performing this operation enables vectorization by performing arithmetic over loops in C rather than Python. However, broadcasting can introduce slowdowns or excessive memory utilization. In addition, because not all operations can be *broadcast*, the *non-broadcast* operations can be significantly slower than their counterparts.

2) Accelerating ndarray Operations with JIT Compilation and Parallelization: To further reduce the overhead of the Python layer as well as overcome the performance issues associated with Python's dynamic nature, many libraries ship with JIT compilers. These compilers aim to extract the maximum performance, or at least provide equivalent performance to hand-optimized compiled code. In particular, Numba transpiles Python and NumPy code to LLVM IR and compiles it to assembly. PyTorch supports compilation using backends, such as Torch Dynamo [4] and FX [5]. In our work, we focus on Dynamo. TensorFlow implements a tracing operation for its first call, constructing a compute graph and compiling it for faster execution. JAX works similarly underneath, relying on compiling operations via XLA to achieve high performance.

Broadcasted Operations	Non-Broadcasted Operations

`z = a * x`	`for i in range(len(x)):`
	`  for j in range(3):`
	`  z[i, j] = a[j] * x[i, j]`

TABLE II: In this example a is a vector of shape (3,) and x is a array of shape (N, 3). Because the sizes of a and x are different, a is broadcast to the shape of x and then multiplied.

Since these libraries are designed to solve numerically intensive deep-learning problems, many libraries also support auto-parallelization. This enables the library to leverage the multiple cores present on the CPU to speed up the computation. In addition, some libraries support GPU offloading. This allows the libraries to offload the computations onto the GPU. Because NumPy only supports parallel execution for a limited range of BLAS workloads, we use Intel's Data Parallel NumPy (DPNP) to represent parallel NumPy. Table I summarizes the libraries we consider and their supported features.

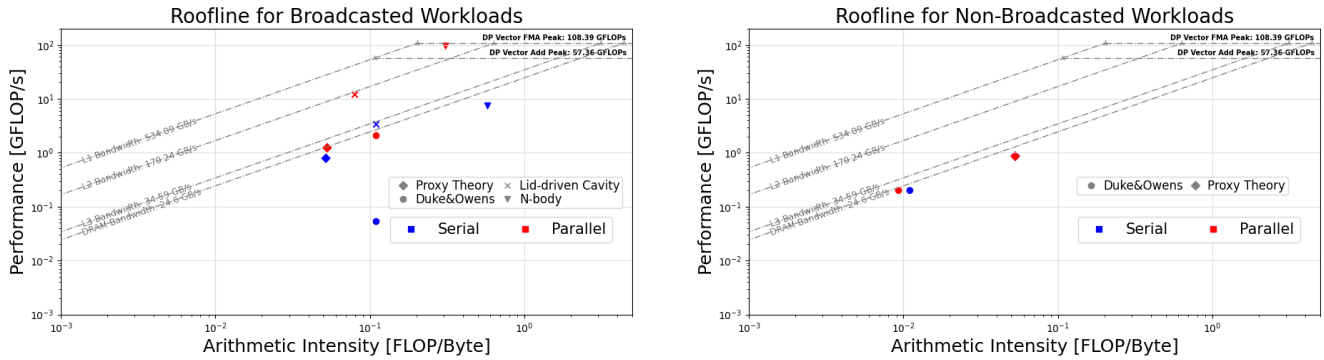
3) Memory Management in Python: Python is a garbage-collected language that employs two methods: (1) reference counting and (2) cyclical garbage collection. Reference counting is the main collection algorithm, which tracks an object's number of live references. When the number of references becomes zero, the object is destroyed and the memory is freed. While this algorithm is fast and easy to implement, it cannot handle cyclic references, where objects can hold references to themselves. The cyclical garbage collector handles the situations where the reference counting algorithm will never free the memory and the process will effectively leak this memory. Garbage collection is easier to use but adds overhead to the application [6], [7], [8].

B. Background of Computational Physics Workloads

We evaluate four distinct workloads to capture a range of computational challenges: two from quantum chromodynamics (QCD) — a proxy theory and Duke & Owens — and two from physics simulations — lid-driven cavity, implemented using stencil-like operations, and n-body molecular simulation to capture dense pairwise computation. Each workload has been selected to represent a different computational challenge, ranging from dense all-to-all interactions and structured stencil computations to computation of transcendental math and fused multiply-add operations.

Quantum correlation functions (QCFs) describe the longitudinal momentum of quarks and gluons (collectively known as "partons") inside moving hadrons such as protons and neutrons. These functions are not calculable from first principles

979-8-3315-7844-2/25/$31.00 ©2025 IEEE



(a) The rooflines for the broadcasted workloads of the C++ implementations show that they exhibit a variety of computational patterns. We can see that they represent workloads in the memory-bound and memory-compute bound regions.

(b) The rooflines for the non-broadcasted QCD workloads are, as expected, memory bound. We note the arithmetic intensity and performance of the serial and parallel Proxy Theory are almost identical, and thus are hard to differentiate in the plot.

Fig. 2: Roofline models 2(a), 2(b) for the different different styles of execution

but can be reconstructed from high-energy particle scattering data. A key bottleneck in this global QCD analysis is the evaluation of the phase space density (or the differential cross-section) for a particular formulation of the QCFs [9].

1) Proxy Theory: Proxy theory is a simple model of a QCF that directly resembles the differential cross-section but does not have the same complexity or computational intensity. This model is parameterized by five parameters and is used as a test case for prototyping. For the non-broadcasted cases, the computation is primarily dominated by the calculation of exponential functions and floating-point scalar multiplication and addition. For the broadcasted cases, the computation transforms into vector arithmetic.

2) Duke & Owens: Duke and Owens [10] is a model used to describe Q^2-dependent parton distributions to leading logarithm accuracy. This model requires 36 parameters to be fit. The calculation of the non-broadcasted cases for this model, like proxy theory, is dominated by scalar multiplication and the computation of exponentials and logarithms. Additionally, because it involves more parameters, it is more memory-bound. For the broadcasted cases, most of the computation transforms into vector arithmetic.

3) N-Body Physics Simulation: N-body simulations employing Newton's law of gravity exhibit distinct computational patterns characterized by repeated pairwise force calculations. For each time step, the gravitational force between every pair of bodies is computed, which directly influences the acceleration of each body. These accelerations are then used to update velocities and positions through sequential integration steps. The dominant factor impacting performance is the all-to-all evaluation of forces with a time complexity of $O(N^2)$, where N is the number of particles. It accounts for the majority of the application's runtime. In contrast, the subsequent updates to positions, velocities, and energy calculations are relatively inexpensive and scale linearly. As a result, the efficiency and scalability of the simulation are largely determined by the implementation and optimization of the core force computation loop. The simulation represents positions, velocities, and masses as arrays and expresses the pairwise interactions through matrix multiplication and array broadcasting, making it possible to be suitably implemented using our test libraries.

4) Stencil-Based Lid-driven Cavity (LDC) Flow: Stencil operations represent a computational pattern in which, given structured data layouts such as pixelated images or multi-dimensional arrays, updating the value of a data point depends only on its old value and the values of its nearby neighbors. This pattern is common in scientific computing, appearing in applications such as convolutional neural networks or heat-conduction simulations via explicit numerical schemes. Because of their importance, we include a benchmark in this work to represent applications that exhibit this pattern.

Our application here is the steady-state incompressible laminar lid-driven cavity flow, a standard validation benchmark in computational fluid dynamics. We adopt the implementation described in [11], which marches the unsteady incompressible Navier–Stokes equations to a steady state on a collocated uniform Cartesian grid. All spatial derivatives are approximated with second-order finite differences, except the convective terms, for which a first-order backward difference is applied. Time integration uses a first-order explicit Euler scheme, and the pressure Poisson equation is solved by Jacobi iterations.

An update at a given point uses only its own old value and its neighbors' old values from the previous iteration and does not use any freshly updated value. This property makes the algorithm embarrassingly parallel for the double loop of the array indexes and well-suited for SIMD execution. In Fortran, the pattern maps directly to built-in array arithmetic that compilers can auto-vectorize. In Python, array arithmetic syntax is provided by libraries such as NumPy and PyTorch, but the extent to which these operations translate into efficient SIMD instructions depends on how the libraries are implemented and built rather than on Python itself.

979-8-3315-7844-2/25/$31.00 ©2025 IEEE

	Proxy Theory		Duke & Owens		Lid-driven Cavity	N-Body	Scaled Geomean
	Broadcasted	Non-Broadcasted	Broadcasted	Non-Broadcasted	Broadcasted	Broadcasted	Productivity
NumPy	14	19	56	64	92	67	1.00
DPNP	14	19	56	64	92	68	1.00
Numba	18	22	57	68	94	74	0.91
PyTorch	21	26	71	75	102	70	0.81
Tensorflow	29	33	70	75	133	78	0.70
Jax	20	16	68	76	115	81	0.85
C++	50	40	139	94	176	176	0.43

TABLE III: Source lines of code for the core computation of the different workloads and libraries (and C++) for the broadcasted and non-broadcasted cases.

C. Computational Analysis of Workloads

To understand the computational behavior of the different workloads, we use roofline analysis, as shown in Fig. 2(a), and Fig. 2(b). We compare the arithmetic intensity (AI) and GFLOPS of the C++ kernels for different workloads. We can see that our workloads represent a breadth of computational patterns, spanning the regions of memory-bound and memory-compute bound.

III. Related Work

In addition to the previously discussed work, there has been a variety of related work that has been done with similar goals. Detlefs et al. [8] performed performance comparisons between Python, C, and FORTRAN and their composites, and Cai et al. [12] compared Python, NumPy, Numba, Cython, Fortran, and OpenACC for CPU and GPU codes in the context of computational fluid dynamics (CFD), whereas Langtangen et al. [13] looked at performing stencil updates for partial differential equations with NumPy, Fortran, C++, Numeric and auto-vectorization. However, these works rely on high-level metrics to explain behavior and do not look into system and hardware-level metrics. Additionally, work has been done that looks at the memory management behavior of different libraries. Zhou et al. [6] look at tracing memory management calls, and Basermann et al. [8] look at in-depth memory behavior for a C++ application. However, in that work they have not combined these metrics with performance metrics, particularly for memory-bound scientific applications. Moreover, Stefano et al. [14] look at the performance impact of using multiple threads with the Global Interpreter Lock (GIL), and Arthur et al. [15] compare JIT compilation with CPython and PyPy for the PyPerformance benchmarks suite. However, as this work discusses, much of scientific computing requires optimized numerical libraries to extract maximum performance from hardware.

IV. Experimental Setup

Table IV articulates the specifications of the compute node used in our evaluation. We built all the libraries, as shown in Table V. We tested the C++ implementation with fast-math optimizations, and found that we got a considerable speedup, while relative error in the output was within tolerances. We similarly built the other libraries with these flags, while ensuring that the numerical errors were within an acceptable range. We note that Tensorflow was built with Clang 15, as Clang 17 builds were unsuccessful. Additionally, Tensorflow disables compilation with fast-math via compile time checks, so it was not enabled in our suite. Also, all libraries that support BLAS backends were built using the Intel Math Kernel Library.

CPU	Intel Sapphire Rapids Xeon Platinum
SKU	8462Y+ 2.80GHz
Number of Cores	2x32 Cores (No Hyperthreading)
RAM	512 GB
Operating System	Rocky Linux 9.4
Linux Kernel	5.14
Intel VTune	2025.0.0
Python	Anaconda 3.10.15

TABLE IV: System Specifications

Library	Version	Compiler	Compiler Flags
NumPy	2.1.4	GCC 13.2	-march=native -Ofast -fno-finite-math-only
DPNP	0.18.0	oneAPI DPC++ 2025.0.4	-march=native -Ofast
Numba	0.60.0	GCC 13.2 llvmlite 0.44	-march=native -Ofast
PyTorch	2.5.0	GCC 13.2	-march=native -Ofast -fno-finite-math-only
Tensorflow	2.18.0	Clang 15	-march=native -O3
Jax	0.4.36	Clang 17	-march=native -Ofast -fno-finite-math-only
C++	std=c++17	GCC 13.2	-mavx512f -Ofast

TABLE V: Software specifications

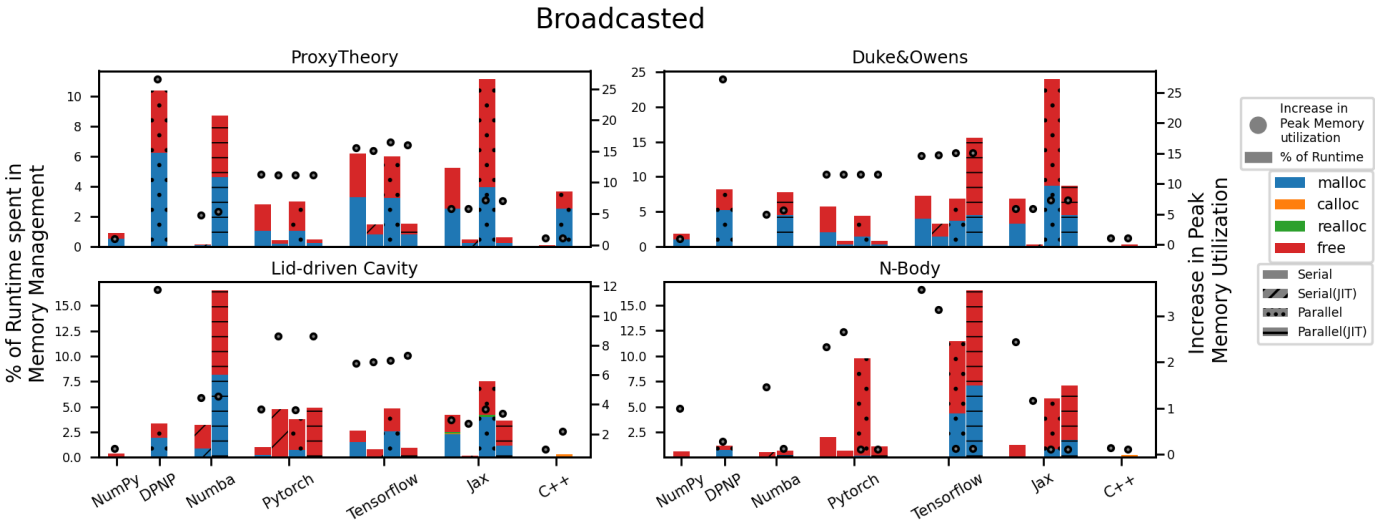
V. Results and Discussion

Here we explore the results of the different problems for productivity, computing, and memory tests. We then characterize the different libraries based on these results for their advantages and disadvantages. We can see the SLoC metric results in Table III, the memory behavior in Fig. 3 and the speedup results in Fig. 4. We first discuss the overall productivity trends for the different libraries. In the following section, we discuss the results C++ and NumPy to help provide users with the context for the problem by forming the baseline and modeling the peak performance. We then discuss DPNP, Numba, PyTorch, and finally Jax and Tensorflow.

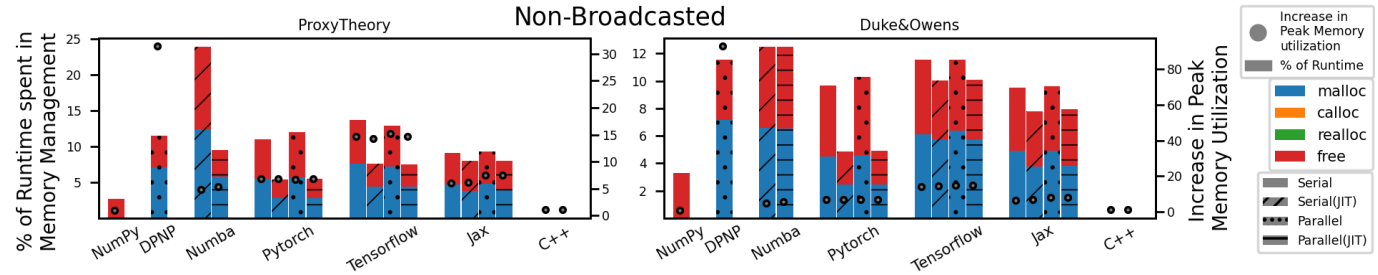
A. Productivity of Python Libraries

Table III shows the SLoC values for the different implementations using the various libraries. We compute the normalized geometric mean (geomean) of the the different libraries by computing the geomean of the SLoC of each of them and

979-8-3315-7844-2/25/$31.00 ©2025 IEEE



(a) Memory behavior of the different libraries and C++ for the broadcasted cases of the workloads. C++ and some libraries can have a minimal memory footprint, leading to them not being visible on the plot.



(b) Memory behavior of the different libraries and C++ for the broadcasted cases of the workloads. C++, similar to the broadcasted cases, shows a minimal memory footprint and isn't visible in the plots.

Fig. 3: Understanding memory behavior and overhead can give valuable insights into the behavior of a library, especially when compared against another, for the same workload.

normalizing them to the productivity of the NumPy baseline. We see that the productivity of the different Python libraries is very similar to the baseline. However, the productivity of C++ is much lower. This motivates our work further, as the key differentiating factor between these libraries is the performance, scalability memory efficiency of these libraries.

B. Characterizing Performance of C++ (Peak) and NumPy (Baseline)

C++, which we chose as the peak performance, consistently performs the best for all problems, providing a geomean speedup of 14× over the baseline. However, when we consider only the serial cases, C++ is 7.56× faster than NumPy. Additionally, C++ and NumPy, on average spend a manageable 0.41% and 1.62% of their runtime on memory management. This is summarized in Table VI.

1) Broadcasted Workloads: We see from Figure 4(a) that the Serial C++ implementation is 3.45× faster than the baseline for the broadcasted workloads. Enabling multi-threading gives C++ a 6.27× speedup. Additionally, we see from Figure 3(a) that C++ shows a 1.61× smaller peak memory consumption than the baselines'.

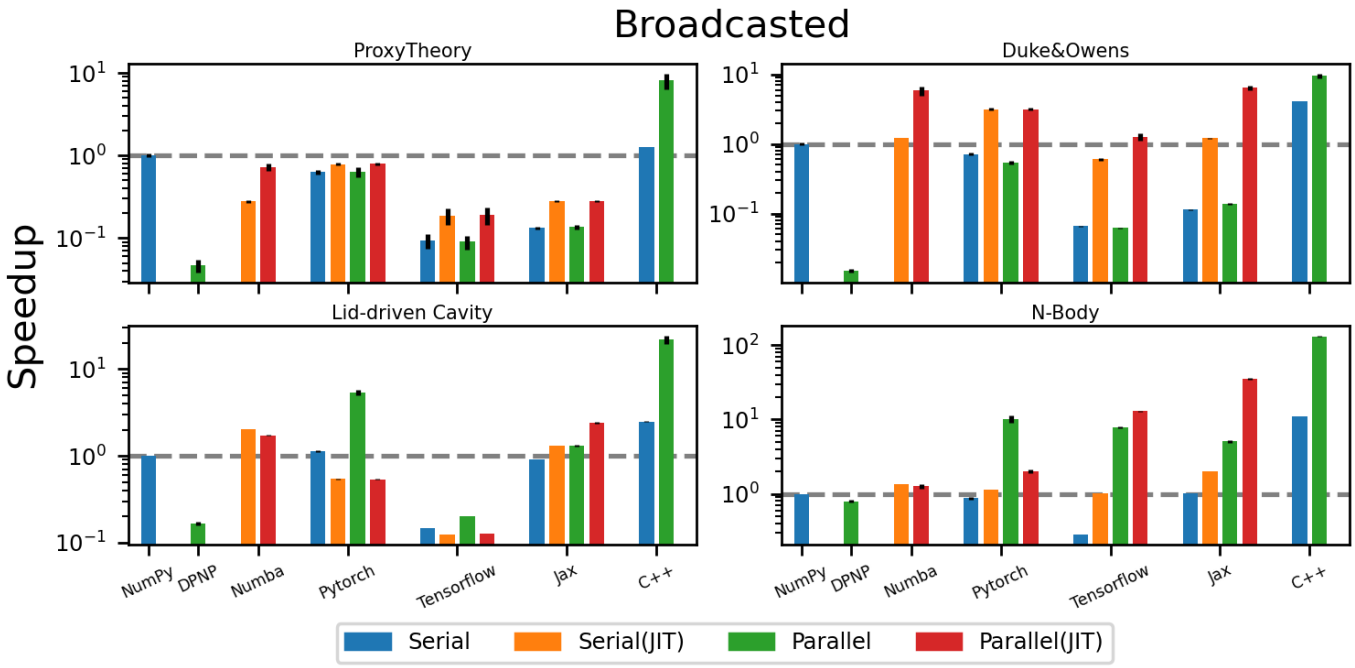
We see that for the proxy theory workload, both NumPy and C++ outperform the rest of the libraries. We identify the cause when looking at the instruction mix, where we find that only the NumPy and C++ implementations use the AVX-512 intrinsics for the exp function, a key bottleneck for this problem.

2) Non-Broadcasted Workloads: Serial C++ performs significantly better for the non-broadcasted workloads, having a 36.2× speedup over the baseline. However, multi-threading doesn't benefit C++ as much, only having a a speedup of 1.02×. Additionally, C++ consumes 1.1× more memory than NumPy for these workloads.

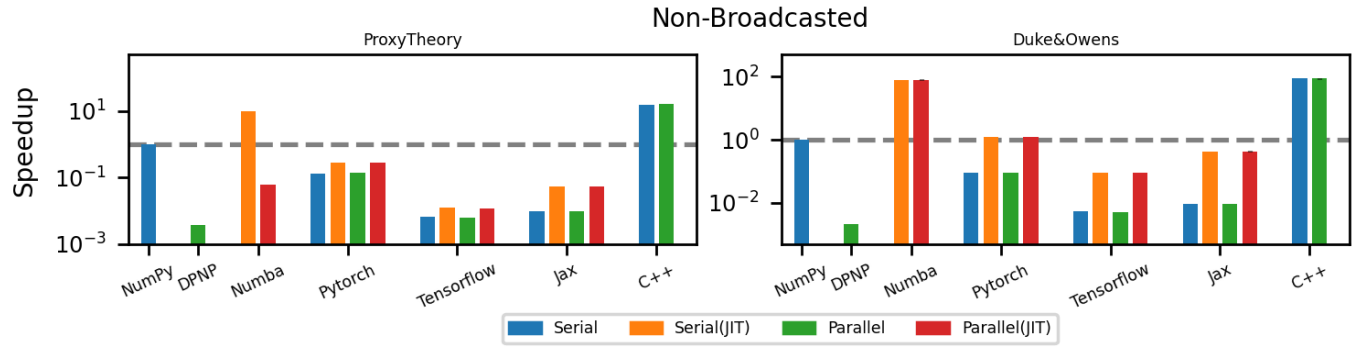
	C++		NumPy	
Metric	Bcast	Non-Bcast	Bcast	Non-Bcast
Performance	21.63	37.13	1.00	1.00
Memory Efficiency	1.37	0.88	1.00	1.00
% Runtime on Mem. Mgmt.	1.16	0.00	0.93	3.00

TABLE VI: C++ vs. NumPy: broadcasted and non-broadcasted cases.

979-8-3315-7844-2/25/$31.00 ©2025 IEEE



(a) Speedup and the corresponding standard deviations of the different libraries for the broadcasted workloads when compared with NumPy. We see that the performance a library is workload dependent, with Jax standing out in terms of performance and features, followed by PyTorch.



(b) Speedup and the corresponding standard deviations of the different libraries for the non-broadcasted workloads when compared with NumPy. We see poor performance across the board for the machine-learning based libraries. Numba stands out for this set of problems.

Fig. 4: The speedup plots show that when considering, C++ is unmatched. In many cases, we even see that it's serial performance is greater than the parallel performance of many libraries.

C. DPNP

Intel's Data Parallel NumPy offers a near drop in replacement for NumPy, enabling parallel execution on both CPUs and GPUs through its SYCL backend. It matches the productivity of the NumPy baseline, having a nearly perfect scaled geomean productivity score of 1. However we note that there were certain functions that had a slightly different signature than NumPy, and while these small changes cannot be captured by the SLoC metric, they significantly hampered productivity.

Additionally, it is inefficient, with a 33.24× slowdown over the serial baseline. It spends 7.69% of its runtime in memory management and has a peak memory utilization that is 13.83× times larger than the baseline. This points to issues in the execution model and memory access patterns. Its poor memory access patterns and the high overhead in its runtime significantly bottleneck the performance. DPNP is a poor choice for scientific computing on the CPU, especially considering that its parallel execution model using 64 cores is slower than the serial baseline of NumPy. This information summarized in Table VII.

1) Broadcasted Workloads: DPNP performs very poorly for this set of workloads, having a 10.16× slowdown over the *Serial* baseline and a 220× slowdown over the parallel C++. It only spends 5.77% of its runtime in managing memory, but consumes a staggering 7× larger amount of peak memory than the baseline. This again points to DPNP using memory in an inefficient way, leading to slowdowns.

979-8-3315-7844-2/25/$31.00 ©2025 IEEE

2) Non-Broadcasted Workloads: DPNP performs very poorly for this set of workloads as well, showing an average slowdown of a staggering 355×. While it spends a considerable 11.5% of its runtime on memory management, not even that can account for this magnitude of poor performance. This points to significant work needing to be done to enable DPNP to come on parity with the performance of the other libraries.

Metric	Bcast.	Non-Bcast.
Performance	0.10	0.002
Memory Efficiency	0.14	0.018
% of Runtime on Mem. Mgmt.	5.77	11.50

TABLE VII: DPNP: broadcasted and non-broadcasted cases.

D. Numba

Numba builds upon NumPy quite well in almost all problems, providing a geomean speedup of 3.01× over the baseline. Enabling multi-threading however, surprisingly slows it down by 1.6×. This points to issues in Numba's auto-parallelization implementation. We note that Numba doesn't support auto-parallelization when the broadcasting is implicit, leading to significant refactoring and thus reduced productivity. It spends a geomean 8.01% of its runtime on memory management, the largest of all the libraries, and has a peak memory utilization that is 3.31× larger than the baseline's. This information summarized in Table VIII.

1) Broadcasted Workloads: Interestingly, Serial Numba, which is supposed to compile NumPy, is a marginal 1.01× slower than the NumPy. Parallel Numba is just 1.74× faster than the baseline and 12.44× slower than the performance of parallel C++. It spends 4.71% of runtime on managing memory, and consumes 2.65× more peak memory than the baseline. Its poor parallelization efficiency combined with its slightly higher memory consumption points to issues both in how it handles memory as well as its ability to decompose vector operations into parallel loops.

2) Non-Broadcasted Workloads: Numba excels at the non-broadcasted workloads, performing 28× faster than the baseline and just 1.28× slower than the performance of C++. Interestingly, there are cases where we see Numba's performance degrade when using multi-threading, unlike with C++. This motivates for careful use of the feature as, just like with broadcasted cases, Numba's auto-parallelization feature can introduce slowdowns.

Metric	Bcast.	Non-Bcast.
Performance	1.74	28.00
Memory Efficiency	0.50	0.018
% of Runtime on Mem. Mgmt.	4.71	11.08

TABLE VIII: Numba: broadcasted and non-broadcasted cases.

E. PyTorch

Serial PyTorch without JIT compilation is 2.4× slower than the baseline and 18.14× slower than serial C++. Enabling JIT compilation gives it a 1.91× speedup, and enables serial PyTorch to being 1.11× slower than the baseline, and enabling multi-threading lets it match the performance of the baseline. PyTorch spends 6.34% of its runtime in memory management, and consumes 5.72× more peak memory than baseline. This information summarized in Table IX.

Metric	Bcast.	Non-Bcast.
Performance	1.27	1.01
Memory Efficiency	0.23	0.14
% of Runtime on Mem. Mgmt.	3.09	11.77

TABLE IX: Parallel PyTorch with JIT: broadcasted and non-broadcasted cases.

1) Broadcasted Workloads: Serial PyTorch with JIT compilation is 1.11× faster than the baseline. However, enabling multi-threading doesn't benefit performance, only giving it a 1.15× speedup. We see in workloads like Lid-driven Cavity and N-Body that enabling multi-threading leads to a performance degradation. This suggests there are overheads within PyTorch's auto-parallelization runtime causing slowdowns. This is further reinforced by 3(a), where we see that PyTorch spends just 3.09% of its runtime in memory management.

2) Non-Broadcasted Workloads: PyTorch performs better than most of its Machine learning oriented counterparts, for these set of workloads. With JIT compilation, we see that PyTorch is 1.71× slower than the baseline. Enabling multi-threading gives it a marginal 1.01× speedup. Additionally, it spends 11.77% of its runtime on memory management, significantly higher than the baselines.

F. Tensorflow and Jax

Serial Tensorflow and Jax perform relatively poorly, having slowdowns of 13.28× and 9.7× respectively over the baseline. However, when we enable JIT compilation and multi-threading, they are 3.66× slower and 1.23× faster than the baseline, respectively. Tensorflow and Jax, to support their functional paradigm, prohibit item assignment, significantly hampering productivity. This is visible in their lower productivity scores as well. Not all workloads can be transformed into functional forms, and this adds significant complexity when implementing in these languages. Jax and Tensorflow use 3.6× and 9.8× more peak memory than the baseline, and spend a considerable 6.51% and 7.74% of their runtime in memory management. This information summarized in Table X.

1) Broadcasted Workloads: When we consider Tensorflow and Jax with JIT compilation, they are 2.89× and 1.03× slower than the baseline. When using multi-threading Jax beats the performance of the baseline by 3.52×, while Tensorflow is 1.26× slower. Interestingly, for the Duke&Owens and N-Body workloads, Jax performs 15× faster than the baseline and just 2.34× slower than the peak performance of parallel C++.

2) Non-broadcasted Workloads: Jax and Tensorflow are not well suited for non-broadcasted workloads. We see that even with JIT compilation, they are 6.67× and 30.5× slower than baseline. Additionally, we see Jax and Tensorflow spend

979-8-3315-7844-2/25/$31.00 ©2025 IEEE

8.66% and 12.29% of their runtime on memory management, a very significant chunk.

Metric	TensorFlow		Jax	
	Bcast.	Non-Bcast.	Bcast.	Non-Bcast.
Performance	0.79	0.03	3.52	0.15
Memory Efficiency	0.16	0.06	0.477	0.128
% of Runt. on Mem. Mgmt.	8.63	12.29	5.01	8.66

TABLE X: Parallel TensorFlow and Jax with JIT: broadcasted and non-broadcasted cases.

VI. Future Work

This work looks into the performance of these libraries when run on a multi-core CPU. We plan on extending these experiments to use GPUs, as a lot of these libraries support offloading work to GPUs. Additionally, a key feature of these libraries is automatic differentiation. Future work will look into the efficiency of automatic differentiation with these libraries.

VII. Conclusion

This work evaluates several libraries in the Python scientific computing ecosystem, focusing on their performance in different execution flows. These libraries provide productive and high-performing alternatives to traditional HPC workflows, which are usually written in compiled languages such as C, C++, or Fortran. By investigating a range of problems in computational physics, we explore various computational patterns and assess how well these libraries handle them. To present a complete picture of the ecosystem, we also examine the impact of JIT compilation and automatic parallelization. Our evaluation considers each library's performance, memory efficiency, scalability, and overall productivity.

Our findings show that NumPy is both productive and performant for serial workloads. Serving as our baseline, it is flexible and delivers strong results across all tested problems and execution strategies. NumPy is also the most memory efficient among the Python libraries evaluated. However, its main limitation is that it is mostly restricted to serial execution, which reduces its applicability for parallel workloads. In comparison, a serial C++ implementation is 7.56× faster than NumPy, though it offers only half the productivity. It also uses just 0.65× the baseline's peak memory. For applications where performance and memory efficiency are critical, C++ is the preferred choice. Furthermore, C++ demonstrates the best scalability with multiple cores, achieving a 26× speedup over NumPy when multi-threading is enabled.

Numba is the ideal choice for problems that cannot be represented as tensor operations. It delivers a 28× speedup over NumPy and is just 1.28× slower than C++. However, it is important to note that Numba's auto-parallelization feature does not always guarantee performance improvements and can sometimes introduce slowdowns.

On the other hand, Intel DPNP consistently underperforms. Despite using multiple threads, it is 33.24× slower than the *serial* baseline and consumes much more peak memory. Much of its runtime is spent on memory management, suggesting inefficiencies in both scheduling and memory access patterns.

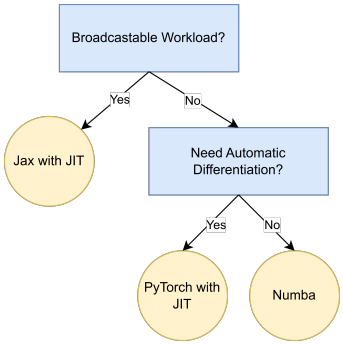


Fig. 5: Decision tree to help guide the decision process for choosing the most suitable library.

When evaluating broadcasted workloads, we find that Jax with JIT compilation stands out, performing 3.52× better than the baseline and outperforming both PyTorch and TensorFlow in terms of performance and productivity. However, Jax's lack of support for in-place array assignment can lead to additional overhead. Moreover, both TensorFlow and Jax perform poorly on workloads that cannot be expressed as tensor operations. Even with JIT compilation, they exhibit slowdowns of 31× and 6.67× over the baseline, respectively. PyTorch emerges as a suitable alternative to Jax and Numba, performing well across all workloads and execution patterns. It is particularly recommended when a workload contains both broadcasted and non-broadcasted operations, or when automatic differentiation is needed for non-broadcasted tasks. To help summaries our findings, we use the decision tree plot in Figure 5 to help guide users and developers when choosing the most suitable library for their workload.

Through this characterization study, our goal is to quantify the trade-offs between performance and productivity that are relevant to scientific applications, especially those incorporating machine learning. In conclusion, this work aims to guide users in selecting the most suitable library for their specific computational needs.

References

[1] M. H. Javed, K. Z. Ibrahim, and X. Lu, "Performance analysis of deep learning workloads using roofline trajectories," *CCF Transactions on High Performance Computing*, vol. 1, no. 3, pp. 224–239, 2019.

[2] C. Yang, Y. Wang, T. Kurth, S. Farrell, and S. Williams, "Hierarchical roofline performance analysis for deep learning applications," in *Intelligent Computing: Proceedings of the 2021 Computing Conference, Volume 2*. Springer, 2021, pp. 473–491.

[3] H. Prashanth and M. Rao, "Roofline performance analysis of dnn architectures on cpu and gpu systems," in *2024 25th International Symposium on Quality Electronic Design (ISQED)*. IEEE, 2024, pp. 1–8.

[4] J. Ansel, E. Yang, H. He, N. Gimelshein, A. Jain, M. Voznesensky, B. Bao, P. Bell, D. Berard, E. Burovski, G. Chauhan, A. Chourdia, W. Constable, A. Desmaison, Z. DeVito, E. Ellison, W. Feng, J. Gong, M. Gschwind, B. Hirsh, S. Huang, K. Kalambarkar, L. Kirsch, M. Lazos, M. Lezcano, Y. Liang, J. Liang, Y. Lu, C. K. Luk, B. Maher, Y. Pan, C. Puhrsch, M. Reso, M. Saroufim, M. Y. Siraichi, H. Suk, S. Zhang, M. Suo, P. Tillet, X. Zhao, E. Wang, K. Zhou, R. Zou, X. Wang, A. Mathews, W. Wen, G. Chanan, P. Wu, and S. Chintala,

979-8-3315-7844-2/25/$31.00 ©2025 IEEE

"Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ser. ASPLOS '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 929–947. [Online]. Available: https://doi.org/10.1145/3620665.3640366

[5] J. K. Reed, Z. DeVito, H. He, A. Ussery, and J. Ansel, "Torch.fx: Practical program capture and transformation for deep learning in python," 2022. [Online]. Available: https://arxiv.org/abs/2112.08429

[6] J. Zhou, D. Li, and T. Liu, "Profile dynamic memory allocation in autonomous driving software," in *2023 10th International Conference on Dependable Systems and Their Applications (DSA)*, 2023, pp. 905–914.

[7] J. Nanjekye, D. Bremner, and A. Micic, "Towards reliable memory management for python native extensions," in *Proceedings of the 18th ACM International Workshop on Implementation, Compilation, Optimization of OO Languages, Programs and Systems*, ser. ICOOOLPS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 15–26. [Online]. Available: https://doi.org/10.1145/3605158.3605849

[8] D. Detlefs, A. Dosser, and B. Zorn, "Memory allocation costs in large c and c++ programs," *Software: Practice and Experience*, vol. 24, no. 6, pp. 527–542, 1994. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.4380240602

[9] P.-Y. Chuang, N. Shah, P. Barry, I. Cloët, E. M. Constantinescu, N. Sato, J.-W. Qiu, and W.-c. Feng, "Characterization and optimization of the fitting of quantum correlation functions."

[10] D. W. Duke and J. F. Owens, "Q^2-dependent parametrizations of parton distribution functions," *Phys. Rev. D*, vol. 30, pp. 49–54, Jul 1984. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevD.30.49

[11] L. Barba and G. Forsyth, "Cfd python: the 12 steps to navier-stokes equations," *Journal of Open Source Education*, vol. 1, no. 9, p. 21, Nov. 2018. [Online]. Available: http://dx.doi.org/10.21105/jose.00021

[12] X. Cai, H. P. Langtangen, and H. Moe, "On the performance of the python programming language for serial and parallel scientific computations," *Scientific Programming*, vol. 13, no. 1, p. 619804, 2005. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1155/2005/619804

[13] H. P. Langtangen and X. Cai, "On the efficiency of python for high-performance computing: A case study involving stencil updates for partial differential equations," in *Modeling, Simulation and Optimization of Complex Processes*, H. G. Bock, E. Kostina, H. X. Phu, and R. Rannacher, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–357.

[14] S. Masini and P. Bientinesi, "High-performance parallel computations using python as high-level language," in *Euro-Par 2010 Parallel Processing Workshops*, M. R. Guarracino, F. Vivien, J. L. Träff, M. Cannatoro, M. Danelutto, A. Hast, F. Perla, A. Knüpfer, B. Di Martino, and M. Alexander, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 541–548.

[15] A. Crapé and L. Eeckhout, "A rigorous benchmarking and performance analysis methodology for python workloads," in *2020 IEEE International Symposium on Workload Characterization (IISWC)*, 2020, pp. 83–93.

979-8-3315-7844-2/25/$31.00 ©2025 IEEE