NPBench: A Benchmarking Suite for High-Performance NumPy

Alexandros Nikolaos Ziogas, Tal Ben-Nun, Timo Schneider, and Torsten Hoefler
Department of Computer Science, ETH Zurich, Switzerland

ABSTRACT

Python, already one of the most popular languages for scientific computing, has made significant inroads in High Performance Computing (HPC). At the center of Python's ecosystem is NumPy, an efficient implementation of the multi-dimensional array (tensor) structure, together with basic arithmetic and linear algebra. Compared to traditional HPC languages, the relatively low performance of Python and NumPy has spawned significant research in compilers and frameworks that decouple Python's compact representation from the underlying implementation. However, it is challenging to compare language compatibility and performance among different frameworks and architectures without a standard set of benchmarks and metrics. To that end, we introduce NPBench, a set of NumPy code samples representing a large variety of HPC applications. We use NPBench to test popular NumPy-accelerating compilers and frameworks on a variety of metrics. NPBench will guide both end-users and framework developers focusing on performance and will drive further use of Python in the high-performance scientific domains.

CCS CONCEPTS

• **General and reference** → **Measurement**; **Metrics**; **Evaluation**; **Performance**.

KEYWORDS

High Performance Computing, Benchmark, Python, NumPy

ACM Reference Format:
Alexandros Nikolaos Ziogas, Tal Ben-Nun, Timo Schneider, and Torsten Hoefler. 2021. NPBench: A Benchmarking Suite for High-Performance NumPy. In *2021 International Conference on Supercomputing (ICS '21), June 14–17, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3447818.3460360

1 INTRODUCTION

Python is quickly developing towards being a dominant language in scientific applications ranging from diverse scientific domains such as molecular dynamics [33, 41] and climate codes [50] to machine learning [2, 37]. This accelerating adoption, which has pushed Python to be the second most-used language in open source projects in 2020 [19], is driven by its rich ecosystem of domain-specific libraries and frameworks but also its seamless integration into Jupyter notebooks [29]. The latter enables reproducible scientific workflows as simple lab notes but also most complex large-scale experiments. This high productivity and broad applicability of the core Python language also result in strong industry support, especially in big data, data science, and deep learning. More and more machine learning and artificial intelligence techniques are finding their way into scientific simulations by amending existing codes. Therefore, Python's use leads to strong synergies among industry, academia, and national labs, centered around its software ecosystem.

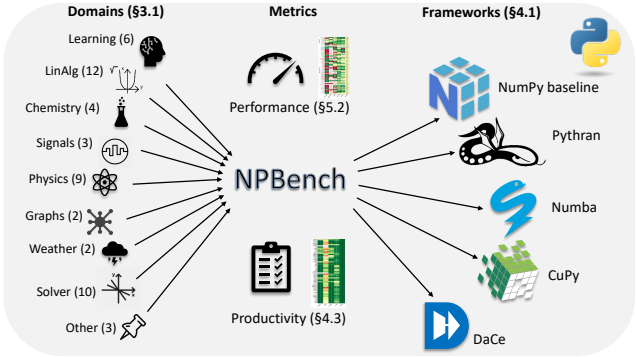

Figure 1: Overview of NPBench: represented scientific domains (number of example codes in brackets), metrics, and frameworks.

Apart from being the interface to data management, Python is also increasingly seen as a framework for scientific computing itself. Python's specific strength is the rich ecosystem of domain-specific libraries and frameworks, such as pandas, SciPy, Scikit-learn, PyTorch, TensorFlow, and Matplotlib, to name a few. The NumPy array library [23] forms a foundation for most, if not all, of those frameworks and many more. NumPy has been carefully optimized to circumvent many of Python's traditional weaknesses in performance. It uses careful low-overhead data storage schemes and utilizes optimized libraries for many operations. In fact, optimizing NumPy has been in its users' core interests, and several specializations exist for this reason [2, 37]. Its central position in performance-conscious scientific computing makes the NumPy library of paramount interest for high-performance, scientific computing, and machine learning workloads.

Techniques for achieving high performance are often specific to each domain. For example, machine learning frameworks use NumPy-like arrays and call operators from hand-optimized libraries (e.g., oneDNN [27] or cuDNN [13]) to achieve high performance. Nevertheless, operators that are not supported by the hand-picked libraries are often executed inefficiently. The state of scientific computing in Python is similar—if applications are using a set of predefined routines (e.g., BLAS or LAPACK) through the interfaces of NumPy or SciPy, then their execution is optimized. However, many high-performance codes require a large variety of functionality far beyond tuned scientific computing and machine learning operators. Furthermore, such codes often use series of small operator invocations that may not be efficient to accelerate individually. Thus, not

ICS '21, June 14–17, 2021, Virtual Event, USA

all NumPy operations can be accelerated manually, and higher-level optimizations play a crucial role in NumPy's ecosystem.

Several approaches exist to improve the performance of NumPy that go beyond merely calling optimized libraries. Most of those approaches analyze the NumPy source code in some way and generate more efficient code. Some, such as Numba [30], DaCe [11], Pythran [22], or CuPy [35] require relatively minor changes to the source code while others such as Cython [10] require programmers to write C-like code. We find that even though all those efforts focus exclusively on performance, it is hard, if not impossible, to compare their performance fairly.

However, even with all these efforts, NumPy performance is not always sufficient. To drive the high-performance Python ecosystem, we define NPBench, a set of representative high-performance computing benchmarks for NumPy implementations. NPBench includes 52 benchmarks from 8 scientific domains of performance-critical workloads in NumPy, ranging from linear algebra to full physics simulations and deep learning. Each benchmark is a carefully extracted performance-critical kernel from a larger application to focus on the critical pieces with a portable benchmark suite. NPBench allows us to establish a set of core features and principles of importance in NumPy codes that deserve special attention when tuning performance. Figure 1 shows an overview of NPBench and the structure of this paper.

NumPy and the Python language are at the pole position to become the main driver of the next generation of scientific computing applications. Python already took a large fraction of the scientific community by storm, and its strong industry support will boost developments further. High-performance computing is now ready to seize this opportunity and adopt a productive language into its core. With NPBench, we establish a clear set of targets to guide and accelerate high-performance frameworks' development. As such, NPBench's goals are similar to those of other impactful benchmarks such as the NAS Parallel Benchmarks [7], the Mantevo benchmark suite [25], or the ImageNet dataset from deep learning [44]. With its open and extensible structure, we expect NPBench to develop into a central catalyst for adopting Python by the high-performance computing community.

2 NUMPY

NumPy is a Python module that centers around the concept of *ndarray*, an efficient multidimensional array object. It includes a plethora of optimized routines that operate on those array objects. These range from basic arithmetic operations, through linear algebra, to statistics and simulations. With these features, NumPy facilitates writing *pythonic* code that is short and concise, readable, and potentially contains fewer bugs. Furthermore, by using array-wide operators, NumPy allows writing code in vectorized form, driving its adoption by the scientific community.

2.1 The NumPy ndarray

The data structure of the ndarray includes a pointer to the data itself and necessary metadata, such as the data type of the array elements, the *shape* of the array, and the access *strides*. The shape is an ordered tuple of the lengths of the array in each dimension. For example, an $M \times N$ matrix has shape (M, N). NumPy ndarrays have fixed size, and their elements must be of the same data type.

We can access the data of an ndarray object with a variety of different syntax. Indexing with slices of the form `start:stop:step` returns a *view* of the underlying data, avoiding unnecessary copies. On the other hand, indexing with scalar coordinates, masks, and boolean or integer arrays (also called *advanced indexing*) produces a copy of the data. A boolean indexing array has the same size as the data array, where a true or false value indicates whether the corresponding data element should be copied or updated. An integer indexing array consists of integer tuples that correspond to the data elements' coordinates to be accessed.

NumPy ndarrays support vectorization, the process of removing unnecessary indexing and loops, using standard mathematical notation. Typically, one must use loops to operate on a Python list structure. For example, the addition of two vectors x and y could be written as follows:

```
z = [None] * len(x)
for i, (a, b) in enumerate(zip(x, y)):
    z[i] = a + b
```

The above operation can be rewritten in vectorized form using NumPy ndarrays in a single line: `z = x + y`.

Another code-simplification concept supported by NumPy is broadcasting. NumPy can treat arrays of different shapes together, implicitly applying element-wise operations on their elements. To bring up another example, adding a scalar value to a Python list can be written in the following manner:

```
y = [None] * len(x)
for i, a in enumerate(x):
    y[i] = a + c  # c is a scalar value
```

With NumPy, the operation is simplified again to a single line: `z = x + c`. Broadcasting doesn't apply only in operations between arrays and scalars, but also among arrays of different shapes, as long as the smaller array can be "broadcasted" to the larger one's shape in an unambiguous way.

The NumPy ndarray class also contains methods that operate on the underlying data. Such routines include transposition and reductions or aggregations, e.g., *sum*, *min*, and *argmax*, to name a few. These routines are further enhanced with keyword arguments that modify their behavior and range of application. For example, reductions have an *axis* argument, which defines the dimensions where the reduction operator will be applied.

2.2 NumPy routines

Arrays can be allocated with undefined values or filled with zeros, ones, and any other constant. They can be "reshaped", i.e., have their shape adjusted while retaining their original total size and data. NumPy also provides methods for splitting an array or concatenating multiple arrays to a single one. Moreover, it implements all the standard Python unary and binary operations and the mathematical functions defined in the standard Python *math* module.

NumPy has sub-modules that implement linear algebra, Discrete Fourier Transformation, random number generation, and polynomial fitting, among others. For example, the linear algebra sub-module implements matrix and vector products (e.g., inner, outer,

and tensor dot product), decompositions (e.g., Cholesky and QR decompositions), and norms.

2.3 Memory Model and Interoperability

What enables flexibility and performance in NumPy is its memory model. Native Python objects, such as lists, are composed of linked lists and trees that must be traversed with multiple pointer indirections, prioritizing interpreter convenience over memory locality. The fixed-size NumPy arrays, on the other hand, are represented as contiguous buffers, which can be allocated by NumPy itself or other allocators (owing to their `__array_interface__` standard). This enables high-performance behavior — transpositions, for example, are not directly evaluated by default but simply swap the memory order. The memory model and vectorization not only allows NumPy to invoke external (e.g., BLAS) libraries directly but also promotes runtime optimizations, such as lazy evaluation, which some NumPy implementations use. Nearly all popular high-performance Python libraries, especially in the deep learning domain, followed suit and adopted similar memory models.

Interoperability goes beyond BLAS libraries. Due to the contiguous buffers and array interface standards, many libraries, including pre-compiled libraries and custom-written C/C++ extensions, could directly use NumPy buffers as inputs/outputs. The aforementioned model, combined with the simple, MATLAB-esque syntax, led to the wide adoption of NumPy by scientific computing library developers and users as one.

3 PRINCIPLES OF NPBENCH

A primary design goal of NPBench is to be representative of the current and potential future use of Python in HPC. To that end, we select micro-apps from a wide variety of scientific domains characterized by standard computation and communication patterns [6]. We also write and adapt benchmarks utilizing the expressive syntax of Python and the NumPy module. The above process leads to a collection of samples emulating the *pythonic* (or *numpythonic*) code style that domain scientists use in Python HPC applications.

3.1 Scientific Domains

We provide an exhaustive list of the NPBench code samples, sorted by their scientific domain. We also provide a short description for each sample.

3.1.1 Chemistry. We include two code samples from PyFAI [28], a Python library implementing azimuthal integration. The two samples are *azimnaiv* and *azimhist*, with the latter being a "code-reduction" of the former, down to two calls to the NumPy built-in method *histogram*. We also adapt the Polybench [38] implementations of the Nussinov algorithm (*nussinov*), which predicts nucleic acid structures, and the multiresolution analysis kernel *doitgen*.

3.1.2 Digital Signal Processing. In this domain, we include a code sample implementing the Stockham FFT algorithm (*sthamfft*) [12], a variation of the Cooley–Tukey algorithm allowing for greater utilization of SIMD architectures. Furthermore, we implement the *deriche* edge detector from Polybench [38], and we include a short sample rescaling and clipping the values of an array to a specified interval (*clipping*) [47].

3.1.3 Graph and Sparse Algorithms. We adapt the Polybench [38] implementation of the Floyd-Warshall shortest path algorithm (*floydwar*). We also implement sparse matrix-vector multiplication (*spmv*), which is used in GraphBLAS variants of breadth-first search.

3.1.4 Machine Learning. We include implementations of the basic deep learning operators 2D-convolution and softmax (*conv2d* and *softmax*). Furthermore, we provide implementations of a 3-layer Multilayer Perceptron (MLP), the LeNet-5 [31] Convolutional Neural Network (CNN), and the bottleneck residual block in ResNet-50 [24] CNN. Both CNNs operate in inference mode. The corresponding samples are *mlp*, *lenet*, and *resnet*. Furthermore, the kernels *correlat* and *covarian* are adapted from Polybench [38]. They represent the correlation and covariance statistical techniques used to compare different populations of data.

3.1.5 Physics. We adapt two code samples, *cavtflow* and *chanflow*, from CFD Python [9], a Jupyter-notebook tutorial for the Navier-Stokes equations using Python and NumPy. The micro-apps solve the cavity and channel flow equations in two dimensions. We adapt an N-Body simulation program (*nbody*) of star orbits according to Newton's Law of Gravity [34]. We adapt two samples, *coninteg* and *sselfeng* from the OMEN quantum transport simulator [48, 53]. The two samples represent computation patterns that appear in the simulation of nano-devices' thermal characteristics. We also adapt the stencils *jacobi1d*, *jacobi2d*, *heat3d*, and *fdtd_2d* from Polybench [38].

3.1.6 Linear Algebra. We implement a variety of linear algebra kernels. Most of them are adapted from Polybench [38] and can be classified into two groups:

(1) BLAS routines: *gemm*, *gemver*, *gesummv*, *symm*, *syr2k*, *syrk*, and *trmm*.
(2) Composite dense linear algebra kernels: Two and three matrix multiplications (*2mm*, *3mm*), A^TAx (*atax*), matrix-vector product and transpose (*mvt*), and trace computation (*npgo-fast*) [3].

3.1.7 Solvers and Matrix Decomposition. We include several solvers in the benchmark suite: Gauss-Seidel PDE solver (*seidel2d*), Toeplitz and triangular system solvers (*durbin*, *trisolv*), Alternating Direction Implicit solver (*adi*), and a kernel from Biconjugate Gradient Stabilized method (*bicg*), all adapted from Polybench [38]. Additionally, we adapt the Gram-Schmidt orthonormalization process (*gramschm*), LU (with and without pivoting), and Cholesky decomposition (vectorized and non-vectorized).

3.1.8 Weather Prediction and Climate Models. We include two micro-apps, *vadv* and *hdiff*, both adapted from the test suite of GT4Py [50]. The former represents vertical advection and the latter horizontal diffusion, both from the COSMO dynamical core [8, 14].

3.1.9 Other Domains and Basic Kernels. We further provide several code samples that do not belong to a major HPC scientific domain but exhibit interesting computation patterns or Python and NumPy syntax. We include two different implementations of the escape-time algorithm for generating Mandelbrot sets (*mandel1* and *mandel2)* [43]. We also adapt the CRC-16-CCITT algorithm (*crc16*) for cyclic redundancy check (CRC) [36].

3.1.10 Relation to Berkeley Parallel Dwarfs. Research in HPC and Parallel Computing, in general, has led to the identification of several computational and communication patterns [6] that are prevalent in scientific applications. With NPBench, we focus on those motifs that are amenable to array programming and, therefore, map naturally to NumPy syntax and operations. By adapting samples that cover an extensive range of those motifs, we provide a platform for exploring the expressibility of Python and NumPy, together with their performance potential. We cover dense linear algebra (BLAS and other linear algebra kernels and solvers), spectral (*sthamfft*) and N-Body (*nbody*) methods, structured grids (stencils, weather and CFD micro-apps), Monte-Carlo and other embarassingly parallel computations (*npgofast*), combinational logic (*crc16*), and dynamic programming (*floydwar*, *nussinov*). NPBench does not cover sparse linear algebra, apart from *spmv*, an implementation of sparse matrix-vector multiplication. The reason is that NumPy by itself does not provide support for sparse structures. SciPy [51], a module complementary to NumPy for scientific computing, does so but is out of the scope of this work. However, it can provide the basis for a future extension of NPBench.

3.2 NumPy Coverage

NPBench includes sample codes gathered from various sources of NumPy programming. Most are used as-is with minor modifications to facilitate benchmarking. For example, we convert global variables to method arguments and enforce consistent data types. To demonstrate the breadth of the feature-set covered in NPBench, we provide code samples below and discuss the Python language and relevant NumPy features.

3.2.1 Simple example codes. The *clipping* sample is taken from the Cython tutorial for NumPy users [47]. The method comprises simple array-scalar operations and a call to the NumPy built-in method `clip`, which clips (limits) the input array to a specified interval:

```
def clipping(array_1, array_2, a, b, c):
    return (np.clip(array_1, 2, 10) * a +
            array_2 * b + c)
```

The *resnet* sample consists of several methods, including the following implementation of the batch normalization operator. It consists of calls to the NumPy reductions for computing the mean and standard deviation of data. These calls utilize the *axis* and *keepdims* keyword arguments, which define the reduction dimensions and the shape of the output.

```
def batchnorm2d(x, eps=1e-5):
    mean = np.mean(x, axis=0, keepdims=True)
    std = np.std(x, axis=0, keepdims=True)
    return (x - mean) / np.sqrt(std + eps)
```

3.2.2 Adaptation of Polybench. Polybench is a collection of polyhedral algorithms, such as linear algebra kernels and solvers, written in C. To adapt its kernels, we first perform a straightforward translation of the loops and per-element array accesses to their equivalent Python syntax. As an example, the Cholesky decomposition is written as:

```
def cholesky(A):
    for i in range(A.shape[0]):
        for j in range(i):
            for k in range(j):
                A[i, j] -= A[i, k] * A[j, k]
            A[i, j] /= A[j, j]
        for k in range(i):
            A[i, i] -= A[i, k] * A[i, k]
        A[i, i] = np.sqrt(A[i, i])
```

However, the low-level code style used above misses an essential benefit of NumPy; the abstraction of array or matrix operations and basic linear algebra kernels to a single line of code. To simulate a NumPy user's code style, we attempt to minimize the code complexity by *vectorizing* loops, i.e., rewriting them as array expressions. For example, the two *k*-indexed loops in *cholesky* can be rewritten as dot products:

```
def cholesky(A):
    A[0, 0] = np.sqrt(A[0, 0])
    for i in range(1, A.shape[0]):
        for j in range(i):
            A[i, j] -= np.dot(A[i, :j], A[j, :j])
            A[i, j] /= A[j, j]
        A[i, i] -= np.dot(A[i, :i], A[i, :i])
        A[i, i] = np.sqrt(A[i, i])
```

In some cases, we can go one step further and provide an alternate, higher-level implementation, using NumPy built-in methods whenever appropriate. NumPy includes a routine for Cholesky decomposition, which we can use to simplify the code further. Therefore, an alternative implementation of the kernel that still returns the same result as the original code written in C is the following:

```
def cholesky(A):
    A[:] = (np.linalg.cholesky(A) +
            np.triu(A, k=1))
```

We note that NPBench includes codes similar in style to the two latter Cholesky examples (*cholesky*, *cholesky2*).

3.2.3 Basic Indexing. NumPy ndarrays can be accessed with a variety of syntaxes. Basic indexing (as defined in NumPy's documentation [1]) includes slices of the form `start:stop:step`, where any of the three indices may be missing or even be a negative integer. The following snippet from *cavtflow* showcases such indexing.

```
p[1:-1, 1:-1] = (
    ((pn[1:-1, 2:] + pn[1:-1, :-2]) * dy**2 +
     (pn[2:, 1:-1] + pn[:-2, 1:-1]) * dx**2) /
    (2 * (dx**2 + dy**2)) -
    dx**2 * dy**2 / (2 * (dx**2 + dy**2)) *
    b[1:-1,1:-1])
```

Another basic-indexing feature is the use of `newaxis` to add a dimension of length one to an array view. Python programmers can use it to manipulate broadcasting in array expressions and the shape of the output. We provide a code snippet from the 2D-convolution benchmark:

```
output[:, i, j, :] = np.sum(
    input[:, i:i + K, j:j + K, :, np.newaxis] *
    weights[np.newaxis, :, :, :],
```

```
    axis=(1, 2, 3),)
```

3.2.4 Advanced Indexing. NumPy ndarrays may also be accessed in an unstructured way using indexing arrays. For example, a boolean array of the same shape may indicate which elements of the data array must be extracted. A use-case of this feature can be observed in the mandelbrot samples (*mandel1*, *mandel2*):

```
for n in range(maxiter):
    I = np.less(abs(Z), horizon)
    N[I] = n
    Z[I] = Z[I]**2 + C[I]
```

3.2.5 NumPy Routines and Sub-Modules. Many samples utilize NumPy routines, including NumPy universal functions and methods from the NumPy sub-modules. For example, the *coninteg* sample uses the linear algebra sub-module for inverting a square matrix and solving a linear system of equations:

```
for z in int_pts:
    Tz = np.zeros((NR, NR), dtype=np.complex128)
    for n in range(slab_per_bc + 1):
        zz = np.power(z, slab_per_bc / 2 - n)
        Tz += zz * Ham[n]
    if NR == NM:
        X = np.linalg.inv(Tz)
    else:
        X = np.linalg.solve(Tz, Y)
```

4 MEASURING PRODUCTIVITY WITH NPBENCH

After designing NPBench, we use it to test state-of-the-art NumPy-accelerating compilers and frameworks. We opt to emphasize out-of-the-box performance in this work. Therefore, we select frameworks that can optimize *pythonic* code without requiring a performance-oriented rewrite from the user. For this reason, we exclude, for example, Cython [10] which, although able to handle vectorized NumPy code, will not offer any speedup unless the code is rewritten in a C-like manner in the extended Cython language. Furthermore, we limit our exploration in this work to the reference Python interpreter, CPython [40].

The frameworks that we test are, ordered by date of introduction, Pythran [22], Numba [30], CuPy [35], and DaCe [11]. We proceed with an introduction for each one of them.

4.1 The Contenders

4.1.1 Pythran. Pythran [22] is a static compiler for a subset of Python that includes NumPy ndarrays and routines. It converts supported Python code to a Python Abstract Syntax Tree (AST) based intermediate representation (IR), applies optimizations, and outputs C++ code. The generated code is then compiled with, e.g., g++ to produce an optimized Python module for CPU execution in the form of a shared library.

Pythran accepts as input a Python module with one or more methods. The routines to be exported may optionally include an annotation describing the data types of the arguments. This annotation takes the form of a comment:

```
# pythran export method_name(type1, type2, ...)
```

The annotated types can be Python built-in types, for example, `int` and `float`, or NumPy types, such as `int64` and `float32`. It is possible to define arrays and their dimensions.

After parsing the input Python module, Pythran converts it to an IR similar to regular Python AST. In this IR, it performs a variety of code transformations that in turn allow the application of optimizations in the generated C++ code. These optimizations include using C++ expression templates to avoid redundant intermediate arrays, loop vectorization to take advantage of vector instructions, and loop parallelization with OpenMP.

4.1.2 Numba. Numba [30] is an LLVM-based Just-in-Time (JIT) compiler for CPython [40], the Python language's reference implementation. It speeds up Python methods that utilize NumPy ndarrays and have been explicitly annotated by the user. It does so by generating efficient loops that access these arrays on a per-element basis with comparable performance to compiled languages. In addition, Numba supports single- and multi-core CPUs and offers the capability to write GPU kernels manually.

Numba performs optimizations during runtime on the methods explicitly annotated by the user with the `@numba.jit` decorator. Numba first receives as input the CPython bytecode of those methods and lowers them to the Numba intermediate representation (IR). It then builds a dependency graph and attempts to infer the types of all the values in the IR. If it succeeds for all of them, it proceeds in *nopython* mode (i.e., cannot call arbitrary Python), where it applies several high-level optimizations before lowering the methods further to LLVM IR. In cases where type inference fails, Numba falls back to *object* mode. Like NumPy, Numba considers all values to be Python objects in this mode, and it relies on the Python C-API and interpreter for code execution.

In *nopython* mode, Numba improves execution in three main ways, which complement each other. First, it removes unnecessary indirection overheads when accessing NumPy ndarrays with an index expression. Therefore, loops that perform operations among ndarrays on a per-element basis execute as fast as the equivalent array expressions. Second, it finds array expressions that consist of multiple ndarrays and Python built-in operators and rewrites them in loop-form. When regular NumPy executes such array expressions, it generates temporary arrays to hold the intermediate data. By converting these expressions to loops, Numba ensures that the intermediate data can be stored in registers, reducing the data movement. Lastly, Numba can perform loop fusion when applicable.

In *object* mode, the optimization potential is reduced. However, Numba can still split out loops (where type inference may succeed) and apply "deferred loop specialization" on them. Numba automatically transfers those loops to a separate method, which it attempts to compile in *nopython* mode.

Numba supports single- and multi-core CPUs with OpenMP, Intel TBB, and a simple built-in work-sharing task scheduler. Parallelization is enabled through the keyword argument `parallel` of the `@numba.jit` decorator. Furthermore, Numba has a custom loop iterator, `numba.prange`, that allows the user to annotate loops whose iterations can execute concurrently explicitly. Numba also supports execution on Nvidia and AMD GPUs, but not automatically. Instead, it exposes the parallel execution model of the hardware directly to the user, facilitating the writing of GPU kernels.

4.1.3 CuPy. CuPy [35] is an implementation of the NumPy module for Nvidia/AMD GPUs based on CUDA (or HIP for AMD). It performs on-the-fly kernel synthesis and uses the optimized CUDA libraries whenever possible.

CuPy does not optimize NumPy code. Instead, it implements its version of NumPy ndarray and NumPy methods for GPU execution. The CuPy API is compatible with NumPy, and in most cases, it is enough to change the module name from `numpy` to `cupy`. Furthermore, it also implements a subset of the SciPy methods. CuPy's API also contains methods for transferring data between the host and the GPU device.

CuPy synthesizes GPU kernels optimized for the exact shapes and data types of the arguments during runtime. These kernels are cached, and the synthesis overhead is amortized over subsequent executions. Moreover, CuPy allows the user to define their own element-wise and reduction kernels.

CuPy accelerates execution using the optimized CUDA libraries; cuBLAS, cuRAND, cuSOLVER, cuSPARSE, and NCCL. It can also improve performance by fusing kernels, according to user definition. Other optimizations include a custom memory allocator and a memory pool, which speed up and reduce (respectively) memory allocations and deallocations.

4.1.4 DaCe. DaCe [11] is a data-centric parallel programming framework. It accepts programs written in one of the supported front-end programming languages, including Python with NumPy ndarray operations. DaCe converts these programs into Stateful DataFlow multiGraphs (SDFGs), a dataflow-based IR. Subsequently, it optimizes the IR with graph transformations that are applied either automatically or with user-driven intervention. The optimized IR is then translated to one of the supported back-end programming languages and subsequently compiled into a shared library. DaCe supports this way single- and multi-core CPUs, Nvidia and AMD GPUs, as well as Intel and Xilinx FPGAs.

DaCe optimizes Python programs on a per-function basis, in a similar manner to Numba. The user must explicitly annotate Python methods with the `@dace.program` decorator. Furthermore, the method arguments must be type annotated using custom DaCe types that are wrappers around equivalent NumPy data types. Arguments that are NumPy ndarrays must also have their shape defined, either symbolically or with integer constants. `return` statements can have their type automatically inferred and do not need to be type annotated. Moreover, DaCe offers a parallel loop iterator, `dace.map`, which explicitly defines a parallel for-loop in Python code.

DaCe includes an extensive library of graph transformations that can be utilized to optimize the IR of a Python program. These transformations include tiling, loop (`dace.map`) fusion, vectorization, temporary storage for storing intermediate results in registers and caches, and elimination of redundant data transfers. The transformations can be applied either automatically or manually by the user. This can be achieved by environment variables, programmatically through the Python code, or interactively.

After optimizing the IR, DaCe generates code for the supported architectures; in C/C++ for CPUs, CUDA for Nvidia GPUs, and HIP for AMD GPUs. In addition, FPGAs are supported with High-Level Synthesis (HLS), with OpenCL for Intel FPGAs and C++ for Xilinx.

4.2 Other frameworks

Apart from those mentioned above, many other compilers, frameworks, and runtimes accelerate Python and NumPy code. PyPy [42] is an alternative Python interpreter which can speed up Python code via JIT compilation. Dask [16] allows Python programs that use NumPy or CuPy ndarrays to execute in multi-node machines. We also make a special note of the Cython [10] compiler below.

Cython is a compiler for Python but also an extended programming language between C and Python. It converts code written in a mix of Python and the Cython language into C. The user may then employ any C/C++ compiler to generate a shared library and import it in their Python code as a module. Cython provides significant speedups over regular Python code while supporting operations among NumPy ndarrays. However, the latter operations will not run faster than the execution with regular NumPy unless the user rewrites them with loops and per-element accesses. Therefore, any meaningful comparison of Cython against the other frameworks would require extensive rewriting of the benchmark samples and is out of the scope of this work.

4.3 Measuring framework-specific code adaptations

As part of NPBench, we provide a Python program for each of the samples described in Section 3.1. The user can execute these programs through a Python interpreter with NumPy. To accelerate program execution, we provide alternative implementations tailored to each of the frameworks introduced in Section 4.1. These code adaptations may differ from the reference NumPy program for two reasons; framework-specific annotations and rewriting of unsupported NumPy features.

4.3.1 Framework-specific annotations. Frameworks may require minimal changes, such as importing a module, adding special decorators to the Python methods, or explicitly annotating argument data types. We describe such framework-specific adaptations and their use in NPBench in detail below.

Pythran parses Python files and generates templated C++ code for the included methods, supporting arguments of different data types and sizes. To avoid unnecessary overheads due to the templates, the user may optionally place constraints and provide type hints to the Pythran compiler. These take the form of a comment beginning with the string `#pythran`:

```
# pythran export conv2d(float32[:,:,:,:],
#                       float32[:,:,:,:])
def conv2d(input, weights):
```

In the above code, we hint to the Pythran compiler that the method's arguments are 4-dimensional single-precision floating-point arrays.

To execute a Python function with Numba, we annotate it with the `@numba.jit` decorator. The decorator has several keyword arguments that may influence performance:

```
import numba as nb
@nb.jit(nopython=True, parallel=False,
                       fastmath=True)
def conv2d_nopython(input, weights):
```

The `nopython` argument allows selecting between *object* and *nopython* mode of execution. In cases where the original code cannot be executed as-is in *nopython* mode, we also test the performance in *object* mode:

```
@nb.jit(nopython=False, forceobj=True,
        parallel=False, fastmath=True)
def conv2d_object(input, weights):
```

The `fastmath` argument enables the LLVM fast-math flags. We always set this to true. The `parallel` argument enables automatic parallelization of operations. We test with `parallel` enabled and disabled. Furthermore, when `parallel` is enabled, it is possible to annotate for-loops (Python ranges) as parallelizable explicitly, by using the `numba.prange` iterator. When the original code contains Python ranges (for-loops) that can be executed in parallel, we also test Numba with a version using `numba.prange` iterators:

```
@nb.jit(nopython=True, parallel=True,
                       fastmath=True)
def conv2d_nopython_prange(input, weights):
    for i in nb.prange(H_out):
        for j in nb.prange(W_out):
```

CuPy provides a NumPy-compatible interface. For example, the hyberbolic tangent function, equivalent to `numpy.tanh`, is `cupy.tanh`. To minimize the number of required code changes, we rewrite the NumPy import statement as `import cupy as np`.

DaCe only compiles annotated `@dace.program` functions to shared libraries. Furthermore, DaCe requires the function arguments to be type annotated, including their shapes, which must be either integer constants or symbols. DaCe performs symbolic computations using SymPy [32], a Python library for symbolic mathematics:

```
import dace as dc
C_in, C_out, H, K, N, W = (
    dc.symbol(s, dc.int64) for s in (
    'C_in', 'C_out', 'H', 'K', 'N', 'W'))
@dc.program
def conv2d(
    input: dc.float32[N, H, W, C_in],
    weights: dc.float32[K, K, C_in, C_out]):
```

4.3.2 Rewriting unsupported NumPy features. In general, none of the frameworks support every single NumPy method or syntactic element. Whenever we encounter such codes, we write customized best-effort adaptations for each framework. With these adaptations, we aim to change as little as possible, while preserving the "spirit" of the original code. In this process, we utilize features supported by the frameworks but at the same time try to not optimize the code inadvertently. We demonstrate this approach using as an example the following code snippet from the 2D-convolution sample:

```
output[:, i, j, :] = np.sum(
    input[:, i:i + K, j:j + K, :, np.newaxis] *
    weights[np.newaxis, :, :, :],
    axis=(1, 2, 3))
```

Numba cannot execute the above code as-is in *nopython* mode for several reasons. A first issue is the lack of support for tuple of integers as value for the `axis` keyword. We overcome this problem by expanding the `sum` reduction to three nested ones; one for each of the three dimensions specified in the tuple:

```
output[:, i, j, :] = np.sum(np.sum(np.sum(
    input[:, i:i + K, j:j + K, :, np.newaxis] *
    weights[np.newaxis, :, :, :],
    axis=1), axis=1), axis=1)
```

Next, Numba does not support `newaxis` indexing (Section 3.2.3). We generate array views of the same shape as the original code using the NumPy `reshape` method instead:

```
output[:, i, j, :] = np.sum(np.sum(np.sum(
    np.reshape(input[:, i:i + K, j:j + K, :],
               (N, K, K, C_in, 1)) *
    np.reshape(weights, (1, K, K, C_in, C_out)),
    axis=1), axis=1), axis=1)
```

Finally, Numba cannot reshape `input[:, i:i + K, j:j + K, :]` because it is not contiguous in memory. We resolve this problem by forcing a copy of the above slice to a contiguous array:

```
inp = input[:, i:i + K, j:j + K, :].copy()
output[:, i, j, :] = np.sum(np.sum(np.sum(
    np.reshape(inp, (N, K, K, C_in, 1)) *
    np.reshape(weights, (1, K, K, C_in, C_out)),
    axis=1), axis=1), axis=1)
```

5 EVALUATING WITH NPBENCH

5.1 Code adaptation as productivity metric

We evaluate the NumPy coverage provided by the frameworks by extracting code-line metrics from the NPBench samples after adapting them according to Section 4.3. Our intention is to provide measurements of NumPy "compatiblity" and total effort required. To that end, we use two metrics; *total number of lines* and *code coverage* relative to the original code.

For the first metric, we count the total number of lines t for each sample and framework using SLOCcount [52]. To make a distinction between the frameworks, we use subscripts. For example, t_{np} is the number of lines for the original NumPy code, while t_{cp} is the total length of the CuPy adaptation. We use the Python Standard's (PEP8) 80 characters per line limitation [18] and count all code-lines, including import statements, decorators, and other framework-specific syntax. We note that SLOCcount ignores comments. However, for Pythran, we add the comment-lines that we use for type annotation.

For the second metric, we first count the total number of lines t_{np} in the original NumPy code. Then, we count how many of those lines l exist unmodified in the **adapted code**. We define *code coverage* as the ratio $\frac{l}{t_{np}}$. For example, the reference NumPy code for the *floydwar* benchmark is the following:

```
import numpy as np
def kernel(path):
    for k in range(path.shape[0]):
        path[:] = np.minimum(
            path[:], np.add.outer(path[:, k],
                                  path[k, :]))
```

The code is actually 4 lines long in the Python file (for presentation reasons, it is spread here over more lines). Therefore, t_{np} is equal to 4. To produce an adaptation for Numba, we first add two lines, one to `import` the module and another to annotate the method with the `@numba.jit` decorator. Numba cannot handle the call to `add.outer` in *nopython* mode. We remove the offending line and substitute it with two equivalent lines supported by Numba:

```
import numpy as np
import numba as nb
@nb.jit(nopython=True, parallel=False,
                       fastmath=True)
def nopython_mode(path):
    for k in range(path.shape[0]):
        for i in range(path.shape[0]):
            path[i, :] = np.minimum(path[i, :],
                path[i, k] + path[k, :])
```

t_{numba} is 7 lines long in the benchmark file. If we compare the two implementations, we see that they have 3 lines in common; the `import` statement for NumPy, the method signature, and the outer loop using the k index. In other words, l is equal to 3 and the code coverage is $\frac{l}{t_{np}} = 75\%$.

In Fig. 2 (best viewed in color and pdf zoom), we present the results of the NumPy coverage evaluation. Each row represents a benchmark, while each column corresponds to one of the frameworks. We plot in the rightmost column the code length (numbers of lines) for each benchmark written in NumPy. The other columns represent the frameworks annotated below them. The numbers in the cells are equal to the difference $t_{ct} - t_{np}$, where t_{ct} is the total number of lines for each of the contenders. Furthermore, each cell is color-coded to represent code coverage. It ranges from dark green (100%) to dark red (0%). Cells annotated with the "unsupported" label correspond to cases where it is infeasible to adapt the benchmark without violating its main purpose.

We discuss interesting observations based on Fig. 2. About half of the Pythran adaptations have one additional line compared with the original NumPy code, and they exhibit full (100%) coverage. These adaptations are identical to the reference benchmarks, apart from the Pythran-specific comment line that provides type annotations. In a similar manner, half of the Numba programs have two additional lines. These correspond to the import statement `import numba`, and the `@numba.jit` decorator used to annotate the Python methods. The CuPy samples exhibit the inverse behavior. Almost all of them have exactly the same code length as the reference NumPy codes. However, the corresponding cells are colored in various shades of green. This is because all CuPy adaptations edit at least a single line of the original code; they alter the import statement to `import cupy as np`.

We further elaborate on a couple of edge cases. The Numba adaptation of the *azimhist* benchmark contains 33 lines, compared with just five lines of reference code; almost seven times as long. However, the code coverage appears to be high. The original sample comprises the following five lines:

```
import numpy as np
def azimint_hist(data,radius,npt):
  histu = np.histogram(radius,npt)[0]
  histw = np.histogram(radius,npt,weights=data)[0]
```

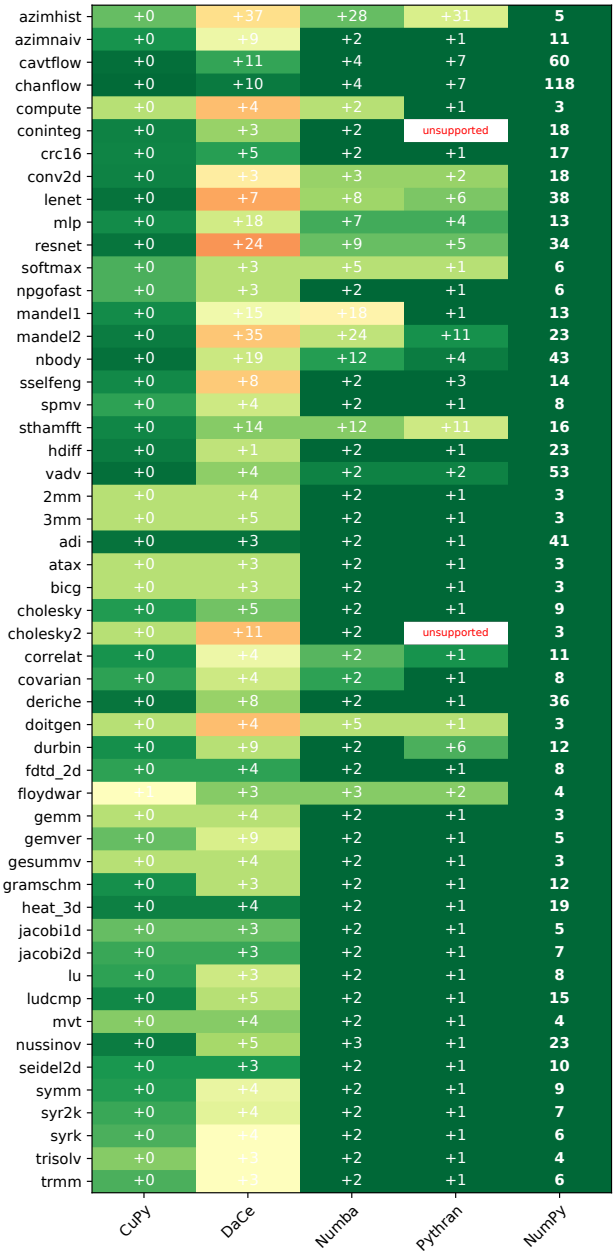

Figure 2: Code adaptation metrics.

```
  return histw / histu
```

Although Numba supports the use of the NumPy routine `histogram` in *nopython* mode, it does not accept the `weights` keyword argument. Therefore, we adapt the code by introducing a custom weighted-histogram implementation [5], which is 28 lines long. However, out of the five original lines of code, four of them remain in the adapted version as-is. Only the second `numpy.histogram` call is substituted by an invocation of the custom histogram method. For this reason, the coverage is 80%.

In *resnet*, a variable x is redefined multiple times as follows:

```
x = batchnorm2d(padded)
```

```
x = relu(x)
```

DaCe does not allow redefinition of variables, therefore the code must be adapted by renaming *x*, contributing to the low coverage (~26%):

```
x = batchnorm2d(padded)
x1 = relu(x)
```

5.2 Performance

We compare the performance of the frameworks in NPBench on a machine with two 16-core Intel Xeon Gold 6130 processors and an Nvidia V100 GPU with 32GB of memory, running CentOS 8. The CUDA version is 11.1. We use CPython version 3.8.5 as part of an Anaconda 3 environment.

We test NumPy version 1.19.2 with Intel MKL support, Numba version 0.51.2 with Intel SVML support, and CuPy 8.3.0. Furthermore, we test the latest versions of Pythran [21] (commit ID 09349c5) and DaCe [45] (commit ID 4c0429) from their respective GitHub repositories. We use the same backend compiler for both of them, GCC version 10.2.0. To benchmark Pythran, we invoke its compiler with the performance flags that enable vectorization with SIMD intrinsics and loop parallelization with OpenMP, as suggested by the developers in their tutorial [46]. For DaCe, we use the default compilation flags from the DaCe configuration file. Furthermore, we test it with different sets of automatically applied transformations that perform loop parallelization, loop tiling, loop fusion, and vectorization, among others. We ensure that both Pythran and DaCe compile with the *-ffast-math* flag to be consistent with Numba.

We use as a metric the runtime of execution, according to well-established scientific benchmarking practices [26]. We present the results for NPBench in Figure 3. Each benchmark is run ten times, and we compare the median runtime of each benchmark executed by each framework to the median runtime of that benchmark using NumPy. The numbers in each cell indicate how much faster/slower that framework executed the benchmark, except for NumPy, where we report the median runtime. Red colors (and numbers prefixed with a downward arrow) are used to show that a framework executed a benchmark slower than NumPy, and green (upward arrow) when it is faster. We classify our benchmarks into domains as described in Section 3.1, and indicate the domain of each benchmark by the color of the respective label. The overview over all domains serves as a legend. In the domain-overview plot on the top of this Figure, the numbers correspond to the geometric mean performance difference over NumPy across each domain. The total score is the geometric mean of the speedups of all benchmarks across all domains. In order to give an indication of dispersion of the ten measured runtimes around the median, we calculated the 95% confidence interval for the sample median using bootstrapping [17] and indicate the size of this interval in percent relative to the median in superscripts, omitting values smaller than 1%.

White areas in the Figure represent combinations for which we were not able to run the benchmark. We classify these failures into four categories and annotate them accordingly. If we are unable to adapt a code sample sufficiently to be accepted by a framework without parse errors, we mark it as "unsupported". If an adaptation is parsed by a framework but fails to compile, we classify it as a "compilation" error. If the sample compiles but does not finish execution due to, for example, a segmentation fault, then we mark it as an "execution" error. Finally, if a framework finishes executing a benchmark, but the result does not validate against the output of NumPy, then we record this as a "validation" error.

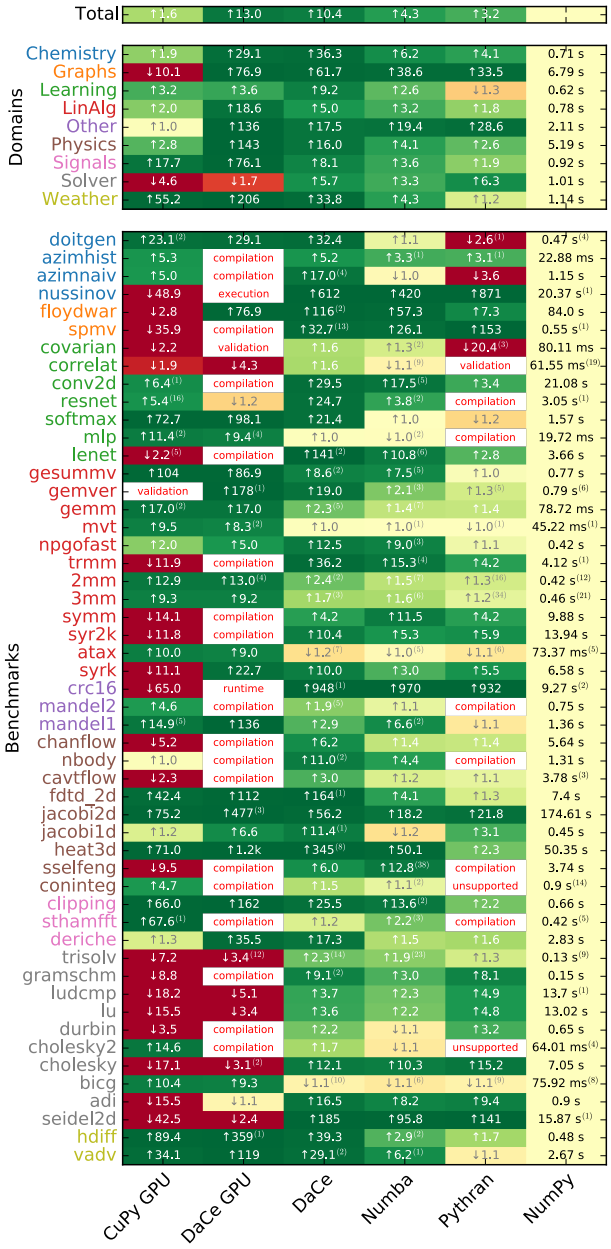

Figure 3: NPBench performance results.

We highlight select performance case studies below:
covarian computes a covariance matrix and performs worse than NumPy on all competitors except for DaCe (CPU) and Numba. Numba itself also runs slower in its native *nopython* mode and only runs with roughly the same performance in the fallback *object* mode. This can be attributed to the internal dot product in the

application, which is performed on non-contiguous vectors. This results in a performance warning from Numba, indicating a potential regression to a more naive implementation in use by Numba and Pythran. Based on a GPU kernel trace, CuPy utilizes the CUBLAS matrix-vector multiplication (GEMV) operation for this purpose but introduces extra memory copies. However, as the problem size is increased, the performance of CuPy approaches that of NumPy.
crc16, which computes a 16-bit Cyclic Redundancy Check digest on a buffer, works particularly well on DaCe (CPU), Numba, and Pythran, with 932–970× speedup. The slow baseline performance can primarily be attributed to the sequential nature of CRC. This causes considerable interpreter overhead, easily avoided by ahead-of-time compilation in DaCe, Numba, and Pythran. For the same reason, we observe a CuPy slowdown of 65×.
hdiff is a composition of stencils that is representative of many horizontal stencil programs found in weather models. The baseline NumPy version implements each stencil as a sequence of offsetted array slices added to each other to promote vectorization. Based on the generated Pythran code, Pythran and Numba follow the NumPy version and can thus reduce the Python interpreter overhead, but most of the computations remain element-wise vectorized array operations. For this reason, due to asynchronous (lazy) execution on the GPU, CuPy achieves 89.4× speedup. The automatically applied transformations in DaCe introduce high-level optimizations such as stencil fusion in both CPU and GPU versions, leading to 39.3× and 359× speedup, respectively.

Discussion. The results show that only Numba (and DaCe, but only on CPU) provides a complete set of correct results, compared with the baseline NumPy implementation. Numba exhibits the highest coverage due to its built-in fallbacks, but not the lowest code modification overhead, which can be found in Pythran for CPU and CuPy on GPU, the latter having no modifications at all. While DaCe provides portability between the two platforms and the highest CPU performance, its (GPU) coverage and code modifications are mostly lower than the other implementations. Overall, no single framework strikes a perfect balance between performance, portability, and productivity with NumPy.

5.3 Using and Extending NPBench

All of the results described in the previous section can be reproduced locally. We structure NPBench in folders, one for each benchmark. In each folder, we include the original NumPy code and the adaptations for the other frameworks. Furthermore, we provide a Python script that automates the execution of the benchmark with the different contenders, breakdown of compilation and first/subsequent execution times, validation against the reference NumPy output, and storage of the results into CSV files. The scripts can be extended to support other frameworks. Moreover, we provide visualization scripts that automatically parse the result files into performance and code coverage heatmaps.

6 RELATED WORK

There are a plethora of available benchmark suites for the Python language. We make a distinction between benchmarks aiming to test Python language implementations and those that test Python-accelerating frameworks.

In the first category, we find benchmarks such as the Python Performance Benchmark Suite (pyperformance) [49] and the PyPy benchmarks [39]. These aim to evaluate Python interpreters in a wide variety of common *every-day* use cases. For example, the Python Performance Benchmark Suite comprises tests for AES encryption, web applications, JSON serialization, message logging, compilation of regular expressions, and database management. Although such benchmarks provide almost complete coverage of the Python language, they do not represent HPC applications well.

Benchmarks belonging to the latter category are the NumPy test suites from the Pythran [20] and Numba [4] developers, as well as the Cython benchmarks [15]. These aim to test and compare the performance of Python and NumPy-accelerating frameworks in codes that fit scientific use-cases. Therefore, there is a significant overlap with NPBench regarding simple kernels, such as those belonging to linear algebra. However, NPBench emphasizes larger samples with higher code complexity that simulate realistic HPC workloads more accurately.

7 CONCLUSION

Python has taken Data Science and Machine Learning by storm and is expected to do the same for High-Performance Computing. With NPBench, we encapsulate both the performance and productivity aspects of high-performance Python programming. For the former, we include 37 micro-benchmarks and 15 micro-applications from 8 different scientific domains. For the latter, the benchmarks cover a wide range of NumPy features and quirks, such as vectorization, broadcasting, and advanced indexing, all of which can often be found in scientific computing applications.

To evaluate the benchmarks, we use four state-of-the-art NumPy frameworks and adapt the codes to run on them. These current contenders show that while NumPy implementations can achieve speedups on some workloads, sometimes of three orders of magnitude, there is still much to be desired in productivity (e.g., compatibility with Python features) and performance. NPBench can thus act as a guiding reference, both for the scientific community and for hardware vendors, to drive the development of next-generation high-performance Python frameworks.

8 ACKNOWLEDGEMENTS

This project received funding from the European Research Council (ERC) under the European Union's Horizon 2020 program (grant agreements DAPP, No. 678880 and DEEP-SEA, No. 955606). The Swiss National Science Foundation supports Tal Ben-Nun (Ambizione Project No. 185778). The authors would like to thank Mark Klein and the Swiss National Supercomputing Centre (CSCS) for access and support of the computational resources.

REFERENCES

[1] The SciPy community. [n.d.]. Ndarray Indexing. Retrieved 2021-02-04 from https://numpy.org/devdocs/reference/arrays.indexing.html
[2] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng.

2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. http://tensorflow.org/ Software available from tensorflow.org.
[3] Anaconda Inc. [n.d.]. A 5 minute guide to Numba. Retrieved 2021-02-04 from https://numba.readthedocs.io/en/stable/user/5minguide.html
[4] Anaconda Inc. [n.d.]. airspeed velocity of an unladen numba. Retrieved 2021-01-28 from http://numba.pydata.org/numba-benchmark/
[5] Anaconda Inc. [n.d.]. Example: Histogram. Retrieved 2021-02-04 from https://numba.pydata.org/numba-examples/examples/density_estimation/histogram/results.html
[6] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine Yelick. 2009. A View of the Parallel Computing Landscape. *Commun. ACM* 52, 10 (Oct. 2009), 56–67. https://doi.org/10.1145/1562764.1562783
[7] David H. Bailey. 2011. *NAS Parallel Benchmarks.* Springer US, Boston, MA, 1254–1259. https://doi.org/10.1007/978-0-387-09766-4_133
[8] M. Baldauf, A. Seifert, J. Förstner, D. Majewski, and M. Raschendorfer. 2011. Operational convective-scale numerical weather prediction with the COSMO model: Description and sensitivities. *Monthly Weather Review, 139:3387–3905* (2011).
[9] Lorena Barba and Gilbert Forsyth. 2019. CFD Python: the 12 steps to Navier-Stokes equations. *Journal of Open Source Education* 2, 16 (2019), 21. https://doi.org/10.21105/jose.00021
[10] Stefan Behnel, Robert Bradshaw, Craig Citro, Lisandro Dalcin, Dag Sverre Seljebotn, and Kurt Smith. 2011. Cython: The best of both worlds. *Computing in Science & Engineering* 13, 2 (2011), 31–39.
[11] Tal Ben-Nun, Johannes de Fine Licht, Alexandros Nikolaos Ziogas, Timo Schneider, and Torsten Hoefler. 2019. Stateful Dataflow Multigraphs: A Data-Centric Model for Performance Portability on Heterogeneous Architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '19)*.
[12] Gabriel Bengtsson. [n.d.]. *Development of Stockham Fast Fourier Transform using Data-Centric Parallel Programming.* Ph.D. Dissertation. http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-287731
[13] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cuDNN: Efficient Primitives for Deep Learning. *CoRR* abs/1410.0759 (2014). arXiv:1410.0759 http://arxiv.org/abs/1410.0759
[14] COSMO. 1998. Consortium for Small-scale Modeling. Retrieved 2021-02-04 from http://www.cosmo-model.org
[15] Cython. [n.d.]. Cython Demos/benchmarks. Retrieved 2021-01-28 from https://github.com/cython/cython/tree/master/Demos/benchmarks
[16] Dask Development Team. 2016. *Dask: Library for dynamic task scheduling.* https://dask.org
[17] Bradley Efron. 1992. Bootstrap methods: another look at the jackknife. In *Breakthroughs in statistics.* Springer, 569–593.
[18] Python Software Foundation. [n.d.]. PEP 8 – Style Guide for Python Code. https://www.python.org/dev/peps/pep-0008
[19] GitHub. 2020. The 2020 State of the Octoverse. https://octoverse.github.com/
[20] Serge Guelton. [n.d.]. Numpy Benchmarks. Retrieved 2021-01-28 from https://github.com/serge-sans-paille/numpy-benchmarks
[21] Serge Guelton. [n.d.]. Pythran. https://github.com/serge-sans-paille/pythran.
[22] Serge Guelton, Pierrick Brunet, Mehdi Amini, Adrien Merlini, Xavier Corbillon, and Alan Raynaud. 2015. Pythran: Enabling static optimization of scientific python programs. *Computational Science & Discovery* 8, 1 (2015), 014001.
[23] Charles R. Harris, K. Jarrod Millman, St'efan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fern'andez del R'io, Mark Wiebe, Pearu Peterson, Pierre G'erard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. 2020. Array programming with NumPy. *Nature* 585, 7825 (Sept. 2020), 357–362. https://doi.org/10.1038/s41586-020-2649-2
[24] K. He, X. Zhang, S. Ren, and J. Sun. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 770–778. https://doi.org/10.1109/CVPR.2016.90
[25] Michael A Heroux, Douglas W Doerfler, Paul S Crozier, James M Willenbring, H Carter Edwards, Alan Williams, Mahesh Rajan, Eric R Keiter, Heidi K Thornquist, and Robert W Numrich. 2009. *Improving Performance via Mini-applications.* Technical Report SAND2009-5574. Sandia National Laboratories.
[26] Torsten Hoefler and Roberto Belli. 2015. Scientific Benchmarking of Parallel Computing Systems: Twelve Ways to Tell the Masses When Reporting Performance Results. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Austin, Texas) *(SC '15)*. Association for Computing Machinery, New York, NY, USA, Article 73, 12 pages. https://doi.org/10.1145/2807591.2807644
[27] Intel Corporation. [n.d.]. oneAPI Deep Neural Network Library (oneDNN). Retrieved 2021-02-01 from https://github.com/oneapi-src/oneDNN
[28] Jérôme Kieffer and Giannis Ashiotis. 2014. PyFAI: a Python library for high performance azimuthal integration on GPU. In *Proceedings of the 7th European Conference on Python in Science (EuroSciPy 2014).* arXiv:1412.6367 [astro-ph.IM]
[29] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain Corlay, Paul Ivanov, Damián Avila, Safia Abdalla, Carol Willing, and Jupyter development team. 2016. Jupyter Notebooks - a publishing format for reproducible computational workflows. In *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, Fernando Loizides and Birgit Scmidt (Eds.). IOS Press, Netherlands, 87–90. https://eprints.soton.ac.uk/403913/
[30] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. 2015. Numba: A LLVM-Based Python JIT Compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC* (Austin, Texas) *(LLVM '15)*. Association for Computing Machinery, New York, NY, USA, Article 7, 6 pages. https://doi.org/10.1145/2833157.2833162
[31] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. 1989. Backpropagation Applied to Handwritten Zip Code Recognition. *Neural Computation* 1, 4 (1989), 541–551. https://doi.org/10.1162/neco.1989.1.4.541 arXiv:https://doi.org/10.1162/neco.1989.1.4.541
[32] Aaron Meurer, Christopher P. Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B. Kirpichev, Matthew Rocklin, AMiT Kumar, Sergiu Ivanov, Jason K. Moore, Sartaj Singh, Thilina Rathnayake, Sean Vig, Brian E. Granger, Richard P. Muller, Francesco Bonazzi, Harsh Gupta, Shivam Vats, Fredrik Johansson, Fabian Pedregosa, Matthew J. Curry, Andy R. Terrel, Štěpán Roučka, Ashutosh Saboo, Isuru Fernando, Sumith Kulal, Robert Cimrman, and Anthony Scopatz. 2017. SymPy: symbolic computing in Python. *PeerJ Computer Science* 3 (Jan. 2017), e103. https://doi.org/10.7717/peerj-cs.103
[33] Naveen Michaud-Agrawal, Elizabeth J. Denning, Thomas B. Woolf, and Oliver Beckstein. 2011. MDAnalysis: A toolkit for the analysis of molecular dynamics simulations. *Journal of Computational Chemistry* 32, 10 (2011), 2319–2327. https://doi.org/10.1002/jcc.21787 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/jcc.21787
[34] Philip Mocz. 2020. nbody-python: Create Your Own N-body Simulation (With Python). https://github.com/pmocz/nbody-python.
[35] Ryosuke Okuta, Yuya Unno, Daisuke Nishino, Shohei Hido, and Crissman Loomis. 2017. CuPy: A NumPy-Compatible Library for NVIDIA GPU Calculations. In *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Thirty-first Annual Conference on Neural Information Processing Systems (NIPS).* http://learningsys.org/nips17/assets/papers/paper_16.pdf
[36] Øystein Sture. [n.d.]. Implementation of crc16 (CRC-16-CCITT) in python. Retrieved 2021-02-04 from https://gist.github.com/oysstu/68072c44c02879a2abf94ef350d1c7c6
[37] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.). Curran Associates, Inc., 8024–8035. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf
[38] Louis-Noël Pouchet et al. 2012. Polybench: The polyhedral benchmark suite. *URL: http://www. cs. ucla. edu/pouchet/software/polybench* 437 (2012).
[39] PyPy. [n.d.]. Benchmarks. https://foss.heptapod.net/pypy/benchmarks.
[40] Python Software Foundation. [n.d.]. CPython. https://github.com/python/cpython.
[41] Richard J. Gowers, Max Linke, Jonathan Barnoud, Tyler J. E. Reddy, Manuel N. Melo, Sean L. Seyler, Jan Domański, David L. Dotson, Sébastien Buchoux, Ian M. Kenney, and Oliver Beckstein. 2016. MDAnalysis: A Python Package for the Rapid Analysis of Molecular Dynamics Simulations. In *Proceedings of the 15th Python in Science Conference*, Sebastian Benthall and Scott Rostrup (Eds.). 98 – 105. https://doi.org/10.25080/Majora-629e541a-00e
[42] Armin Rigo and Samuele Pedroni. 2006. PyPy's Approach to Virtual Machine Construction. In *Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications* (Portland, Oregon, USA) *(OOPSLA '06)*. Association for Computing Machinery, New York, NY, USA, 944–953. https://doi.org/10.1145/1176617.1176753
[43] Nicolas P. Rougier. 2016. *rougier/from-python-to-numpy: Version 1.1.* Zenodo. https://doi.org/10.5281/zenodo.225783
[44] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. 2015. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision* 115, 3 (01 Dec 2015), 211–252. https://doi.org/10.1007/s11263-015-0816-y
[45] Scalable Parallel Computing Lab. [n.d.]. DaCe - Data-Centric Parallel Programming. https://github.com/spcl/dace.

[46] Serge Guelton, Pierrick Brunet et al. [n.d.]. Pythran. Retrieved 2021-02-04 from https://pythran.readthedocs.io/
[47] Stefan Behnel, Robert Bradshaw, Dag Sverre Seljebotn, Greg Ewing, William Stein, Gabriel Gellner, et al. [n.d.]. Cython for NumPy users. Retrieved 2021-02-04 from https://cython.readthedocs.io/en/latest/src/userguide/numpy_tutorial.html
[48] Christian Stieger, Aron Szabo, Teutë Bunjaku, and Mathieu Luisier. 2017. Ab-initio quantum transport simulation of self-heating in single-layer 2-D materials. *Journal of Applied Physics* 122, 4 (2017), 045708. https://doi.org/10.1063/1.4990384 arXiv:https://doi.org/10.1063/1.4990384
[49] Victor Stinner. 2017. The Python Performance Benchmark Suite. Retrieved 2021-01-28 from https://pyperformance.readthedocs.io/
[50] Swiss National Supercomputing Centre (CSCS). [n.d.]. GT4Py. Retrieved 2021-02-01 from https://github.com/GridTools/gt4py
[51] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. 2020. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods* 17 (2020), 261–272. https://doi.org/10.1038/s41592-019-0686-2
[52] David Wheeler. [n.d.]. SLOCCount. Retrieved 2021-02-04 from https://dwheeler.com/sloccount/
[53] Alexandros Nikolaos Ziogas, Tal Ben-Nun, Guillermo Indalecio Fernández, Timo Schneider, Mathieu Luisier, and Torsten Hoefler. 2019. A Data-Centric Approach to Extreme-Scale Ab Initio Dissipative Quantum Transport Simulations. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '19)*. Association for Computing Machinery, New York, NY, USA, Article 1, 13 pages. https://doi.org/10.1145/3295500.3357156

A BENCHMARK DETAILS

The benchmark codes and the scripts to run them and produce Fig. (2, 3) can be found in https://github.com/spcl/npbench, the NPBench GitHub repository. The problem sizes used to measure the performance of the NumPy-accelerating frameworks are shown in Tab. 1.

Benchmark	Problem Size
doitgen	$NR = 220, NQ = 250, NP = 270$
azimhist	$N = 1000000, npt = 1000$
azimnaiv	$N = 1000000, npt = 1000$
nussinov	$N = 500$
floydwar	$N = 2800$
spmv	$M = N = 131072, nnz = 262144$
covarian	$M = 1200, N = 1400$
correlat	$M = 1200, N = 1400$
conv2d	$C_{in} = 3, C_{out} = 16, H = 256, K = 20, N = 8, W = 256$
resnet	$N = 8, W = 56, H = 56, C_1 = 256, C_2 = 64$
softmax	$N = 64, H = 16, SM = 512$
mlp	$C_{in} = 3, N = 8, S_0 = 30000, S_1 = 10000, S_2 = 1000$
lenet	$H = 256, N = 16, W = 256, C_{bfc1} = 59536$
gesummv	$N = 11200$
gemver	$N = 8000$
gemm	$NI = 2000, NJ = 2300, NK = 2600$
mvt	$N = 16000$
npgofast	$N = 12500$
trmm	$M = 1000, N = 1200$
2mm	$NI = 3200, NJ = 3600, NK = 4400, NL = 4800$
3mm	$NI = 3200, NJ = 3600, NK = 4000, NL = 4400, NM = 4800$
symm	$M = 1000, N = 1200$
syr2k	$M = 1000, N = 1200$
atax	$M = 18000, N = 22000$
syrk	$M = 1000, N = 1200$
crc16	$N = 1000000$
mandel2	$xn = yn = 1000, maxiter = 200$
mandel1	$xn = yn = 1000, maxiter = 200$
chanflow	$nx = ny = 101, nit = 50, dt = 0.001$
nbody	$N = 100, tEnd = 10.0, dt = 0.01$
cavtflow	$nx = ny = 101, nt = 700, nit = 50, dt = 0.001$
fdtd_2d	$TMAX = 500, NX = 1000, NY = 1200$
jacobi2d	$TSTEPS = 1000, N = 2800$
jacobi1d	$TSTEPS = 4000, N = 32000$
heat3d	$TSTEPS = 500, N = 120$
sselfeng	$N_{kz} = 4, N_E = 10, N_w = 3, N_A = 20, N_B = 4, N_{orb} = 4$
coninteg	$NR = 500, NM = 1000, slabs = 2, numpts = 32$
clipping	$M = N = 12500$
sthamfft	$R = 4, K = 10$
deriche	$W = 7680, N = 4320$
trisolv	$N = 16000$
gramschm	$M = 240, N = 200$
ludcmp	$N = 2000$
lu	$N = 2000$
durbin	$N = 16000$
cholesky2	$N = 2000$
cholesky	$N = 2000$
bicg	$M = 18000, N = 22000$
adi	$TSTEPS = 100, N = 200$
seidel2d	$TSTEPS = 100, N = 400$
hdiff	$I = 256, J = 256, K = 160$
vadv	$I = 256, J = 256, K = 160$

Table 1: The problem sizes used to measure the performance in Sec. 5.2.