
# PolyBench/Python: Benchmarking Python Environments with Polyhedral Optimizations

Miguel Á. Abella-González
Universidade da Coruña
Spain
miguel.abella@udc.es

Pedro Carollo-Fernández
Universidade da Coruña
Spain
pedro.carollo@udc.es

Louis-Noël Pouchet
Colorado State University
USA
pouchet@colostate.edu

Fabrice Rastello
INRIA Grenoble Rhône-Alpes
France
fabrice.rastello@inria.fr

Gabriel Rodríguez
Universidade da Coruña, CITIC
Spain
gabriel.rodriguez@udc.es

## Abstract

Python has become one of the most used and taught languages nowadays. Its expressiveness, cross-compatibility and ease of use have made it popular in areas as diverse as finance, bioinformatics or machine learning. However, Python programs are often significantly slower to execute than an equivalent native C implementation, especially for computation-intensive numerical kernels.

This work presents PolyBench/Python, implementing the 30 kernels in PolyBench/C, one of the standard benchmark suites for polyhedral optimization, in Python. In addition to the benchmark kernels, a functional wrapper including mechanisms for performance measurement, testing, and execution configuration has been developed. The framework includes support for different ways to translate C-array codes into Python, offering insight into the tradeoffs of Python lists and NumPy arrays. The benchmark performance is thoroughly evaluated on different Python interpreters, and compared against its PolyBench/C counterpart to highlight the profitability (or lack thereof) of using Python for regular numerical codes.

***CCS Concepts:*** • **Software and its engineering → Compilers**; **Interpreters**.

***Keywords:*** Python, Benchmarking, JIT Optimization, Polyhedral Compilation


*CC '21, March 2–3, 2021, Virtual, Republic of Korea*

ACM ISBN 978-1-4503-8325-7/21/03…$15.00
https://doi.org/10.1145/3446804.3446842

**ACM Reference Format:**
Miguel Á. Abella-González, Pedro Carollo-Fernández, Louis-Noël Pouchet, Fabrice Rastello, and Gabriel Rodríguez. 2021. PolyBench/-Python: Benchmarking Python Environments with Polyhedral Optimizations. In *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction (CC '21), March 2–3, 2021, Virtual, Republic of Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3446804.3446842

## 1 Introduction

Python has become one of the most used and taught languages nowadays, popularized well beyond its first uses as a scripting language in e.g., bioinformatics [5] into a go-to language to implement full object-oriented complex applications such as for deep learning approaches, e.g. [10, 22]. Its benefits in terms of flexibility and ease-of-programming come in large part from dynamic typing and runtime interpretation. However, this typically comes with a performance penalty when comparing to e.g. a native C implementation compiled and ran on the target platform.

As Python has become ubiquitous, its ecosystem is rapidly growing, including compiler analyses and optimizations for Python programs being developed. There is a compelling need to provide benchmarks and tools to help characterize the performance profile of various approaches to implement Python programs, recognizing the software ecosystem ranges from classical runtime interpreters [7], just-in-time compilation for improved performance [29] or even optimizing compilers for Python programs [20].

In this work, we aim to enable the evaluation of current and upcoming Python ecosystems for scientific programming, focusing specifically on numerical kernels that are typically implemented using dense arrays (lists). The kernels we consider are all polyhedral programs by design [15], that is, they involve only static control-flow (Static Control Parts, or SCoP). In a SCoP, no operation may be conditionally executed depending on the input data: the execution is deterministic and reproducible irrespective of the actual

input data values. This includes numerous dense linear algebra methods (e.g., matrix product, tensor contractions), dynamic programming, stencil computations for image processing or physics simulation, and equation solvers. Polyhedral programs can be represented at compile-time using affine functions and integer lattices, enabling exact array dataflow analysis to be computed [11]. Consequently, it is possible to design highly advanced automatic optimizing compilation algorithms for polyhedral programs, where aggressive restructuring including loop parallelization and tiling can be seamlessly implemented, e.g. [4, 12, 21, 23, 27]. Production compilers like GCC [26] and LLVM [17] integrate already polyhedral optimizers, as well as numerous research tools such as Pluto [4] or ISL [32].

We introduce PolyBench/Python, a self-contained benchmarking suite made of 30 numerical *polyhedral* kernels, for which polyhedral compilers can be designed and evaluated. PolyBench/Python is on purpose an exact mirror of the 30 kernels in PolyBench/C 4.2 [28] in terms of computation performed, problem sizes and data types. This feature is key to enable fair side-by-side comparison of Python-based implementations with clean, native C implementations optimized with state-of-the-art optimizing compilers.

PolyBench/C provides a single reference implementation for each kernel, and a variety of schemes to allocate data along with mechanisms to ensure reproducible evaluations. PolyBench/Python offers similar features, but it also includes three Python implementations for each benchmark: a simple, C-like implementation using multidimensional arrays; another simple C-like implementation using linearized arrays; and a NumPy [31] implementation. As we demonstrate in this paper, different types of implementation may deliver the best performance, depending on the benchmark and execution environment.

We developed an automated polyhedral optimizer for PolyBench/Python, automatically extracting the polyhedral representation of the kernel, and optimizing it with off-the-shelf polyhedral compilers. We present extensive experimental studies of the impact of several classical polyhedral loop transformations when applied to PolyBench/Python, including complex loop fusion to improve data reuse (using the Pluto algorithm [4]), and loop permutations to further expose parallel inner loops. We make the following contributions:

- We introduce PolyBench/Python, a suite of 30 numerical kernels, for the purpose of benchmarking Python runtime environments and (just-in-time) Python compilers.[1]
- We developed an automated polyhedral compilation flow for PolyBench/Python, and present extensive experiments on the profitability (or lack thereof) of implementing loop transformations for improved data reuse for these benchmarks.
- We present extensive experimental results characterizing PolyBench kernels, contrasting their execution profile in native C with their Python equivalent, to characterize the profitability and overhead of using Python for such numerical computations.
- We present several insights on Python programming styles and good practice to expose solid performance for Python-based implementations of numerical kernels.

The rest of the paper is organized as follows. Section 2 motivates the work. Section 3 introduces PolyBench/Python. Section 4 presents extensive experimental results. Section 5 presents related work, before concluding in Sec. 6.

## 2 Implementing Numerical Kernels in Python

***The Python Ecosystem.*** Python is a high-level, object-oriented programming language that is well established with a very large community in both academia and industry. It is a general-purpose language which is extremely easy to learn due to its very clean syntax and great readability. One of the best characteristics of Python is its productivity. It is a dynamically (but strongly!) typed and interpreted language, with elegant syntax that makes it a very good option for scripting and rapid application development.

NumPy [31] includes both a C array-like storage format, and high performance operations on arrays for Python. Nowadays, many Python codes are efficient enough for production use. However, due to Python's interpreted nature and the lackluster performance of "pure Python" (i.e., not using NumPy or external libraries) codes in the reference CPython interpreter, part of the community dismisses Python as a low performance language in itself.

***Execution Profiles with Different Implementations.*** Figure 1 illustrates the wide ranging performance profiles that can be obtained with various types of dense numerical kernels. It presents the normalized performance, in CPU execution cycles, achieved by different C and Python versions of five selected PolyBench kernels with and without polyhedral optimizations using the Pluto algorithm [4][2]

We observe that Python implementations may end up competing with native C implementations (e.g., `gramschmidt`, `seidel-2d` and `syr2k`); but may also massively degrade performance, e.g., by nearly 10x for the Python versions of

[1] PolyBench/Python is free software, available at https://github.com/UDC-GAC/polybench-python.

[2] Additional and/or different polyhedral optimizations may provide higher performance gain than reported here, no tuning of these optimizations was implemented: we limit here to evaluating the benefit of loop fusion/distribution (possibly including loop skewing/shifting to make the fusion possible).

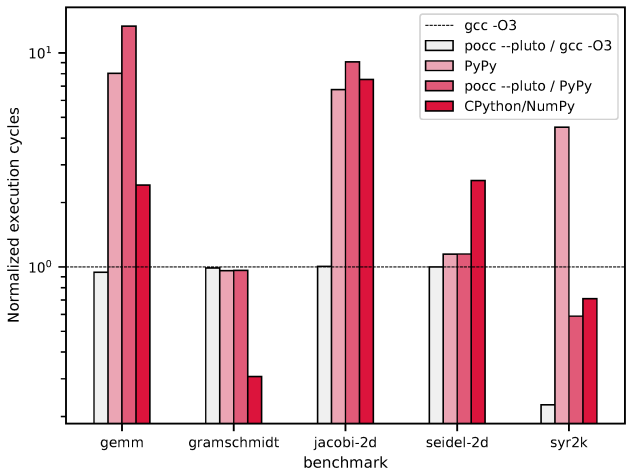


**Figure 1.** Performance (CPU cycles elapsed) of selected C/Python PolyBench benchmarks. The plot presents two C versions, with and without polyhedral fusion to reduce the data reuse distance (through `pocc -pluto`); and three Python versions, with and without polyhedral fusion, and an additional manually vectorized version using NumPy. The Y axis is normalized to the values of `gcc -O3`. The full experimental setup is described in Sec. 4.

`gemm` and `jacobi-2d`. We extensively analyze the reasons for this execution profile differences in Sec. 4, pointing to an extremely significant overhead that can be added by the Python JIT in terms of branching and memory movement instructions for certain codes.

We also observe that the NumPy implementations, the typical method of choice to implement numerical kernels in Python, do not necessarily outperfom the other Python versions, as shown for the two stencils, `seidel-2d` and to a lesser extent `jacobi-2d`. Finally, the impact of polyhedral loop transformations for data reuse appears limited here, as is confirmed by experiments presented later: there is a massive impact for the C and Python versions of `syr2k`, but mostly no benefit elsewhere, pointing to the fact that the performance bottleneck of these codes is not the memory traffic. All this is carefully analyzed and explained later in Sec. 4.

***Evaluating PolyBench/Python.*** This paper targets the Python benchmarking of small computational kernels. The 30 PolyBench/C kernels are translated to Python, supporting the exploration of the optimization space, including parameters such as how arrays are implemented or what is the comparative performance of different Python interpreters for different types of codes. It provides support for automatically measuring performance counters through the PAPI library, enabling seamless performance analysis. Furthermore we have developed a tool to generate a ScopLib representation [8] of a polyhedral Python kernel, and conversely, to generate a Python code from a ScopLib representation. This allows to integrate Python with off-the-shelf polyhedral compilers such as PoCC [8] and PLUTO [1, 4].

## 3 PolyBench/Python

We now introduce PolyBench/Python, which is based on PolyBench/C, and present the approach for the various kernel implementations we provide for each benchmark.

### 3.1 PolyBench

PolyBench is a benchmark suite of 30 numerical computations with static control flow, extracted from operations in various application domains (linear algebra computations, image processing, physics simulation, dynamic programming, statistics, etc.). Its original objective is to offer a set of representative numerical computations that are amenable to polyhedral optimizations, and to offer a platform to ease reproducibility of experiments, including numerous features for e.g. cache flushing, highly accurate timing, support for PAPI hardware counters, and non-random initialization data. PolyBench/C was developed by Pouchet, with significant contributions from Yuki starting from PolyBench/C 4.0 [28].

PolyBench/Python, based on PolyBench/C 4.2.1 by Pouchet and Yuki [28] implements, by design, *exactly* the same computation as in the equivalent PolyBench/C program: the same data types, exact same algorithm (including the same execution order for the operations, for loop-based Python implementations), and exact same dataset sizes. The objective is to provide implementations in different languages of the exact same computation to allow "apple-to-apple" comparisons between native C implementations and JIT/interpreted implementations in Python, thereby enabling to observe the overhead of the Python techniques compared to the execution of only-necessary instructions in the native C program.

Table 1 presents the 30 kernels in PolyBench and their descriptions, for completeness. We report the order of data reuse, that is roughly the order of magnitude of times a data element is being reused by different computations in the kernel. Programs with $O(N)$ reuse, where N represents the problem size (e.g., the number of rows or columns of an input square matrix $N \times N$) are typically viewed as "compute-bound", while those with $O(1)$ reuse as "memory-bound". Stencil computations time-iterated $T$ times have their data reused $O(T)$ times.

### 3.2 PolyBench/Python: General design

The PolyBench/Python benchmarks have been implemented following the Python 3 standard. Note no support was considered for Python 2 as it was discontinued in early 2020 [14].

The actual benchmark implementation is built around the `PolyBench` abstract class. It includes routines that support benchmarking and array allocation, and defines abstract handles that must be filled by implementing subclasses. In particular, a benchmark will extend the `PolyBench` class and

**Table 1.** 30 kernels in PolyBench

| Benchmark | Reuse | Description |
|---|---|---|
| 2mm | $O(N)$ | 2 Matrix Mult. ($\alpha ABC + \beta D$) |
| 3mm | $O(N)$ | 3 Matrix Mult. ($(AB)(CD)$) |
| adi | $O(T)$ | Alternating Direction Implicit solver |
| atax | $O(1)$ | Matrix transpose and vector mult. |
| bicg | $O(1)$ | BiCG sub-kernel of BiCGStab solver |
| cholesky | $O(N)$ | Cholesky decomposition |
| correlation | $O(N)$ | Correlation computation |
| covariance | $O(N)$ | Covariance computation |
| deriche | $O(1)$ | Edge detection filter |
| doitgen | $O(N)$ | Multi-res. analysis kernel (MADNESS) |
| durbin | $O(N)$ | Toeplitz system solver |
| fdtd-2d | $O(T)$ | 2-D Dinite Diff. Time Domain kernel |
| floyd-warshall | $O(N)$ | Graph shortest path length |
| gemm | $O(N)$ | Matrix-multiply ($C = \alpha AB + \beta C$) |
| gemver | $O(1)$ | Vector mult. and matrix add. |
| gesummv | $O(1)$ | Scalar, vector and matrix mult. |
| gramschmidt | $O(N)$ | Gram-Schmidt decomposition |
| head-3d | $O(T)$ | Heat equation over 3D data domain |
| jacobi-1D | $O(T)$ | 1-D Jacobi stencil computation |
| jacobi-2D | $O(T)$ | 2-D Jacobi stencil computation |
| lu | $O(N)$ | LU decomposition |
| ludcmp | $O(N)$ | LU decomposition + Forward Subst. |
| mvt | $O(1)$ | Matrix Vector product and Transpose |
| nussinov | $O(N)$ | Dyn. programming for seq. alignment |
| seidel | $O(T)$ | 2-D seidel stencil computation |
| symm | $O(N)$ | Symmetric matrix-mult. |
| syr2k | $O(N)$ | Symmetric rank-2k update |
| syrk | $O(N)$ | Symmetric rank-k update |
| trisolv | $O(1)$ | Triangular solver |
| trmm | $O(N)$ | Triangular matrix-mult. |

define the abstract methods in Fig. 2. These implement the actual benchmark functionality, including array initialization, the kernel code itself, and printing the results in a standardized format which is readily compatible with the outputs produced by PolyBench/C, allowing for cross-language validation. The `run_benchmark()` method is similar to `main()` in C codes. It is in charge of defining the input and output structures of the benchmark, and initializing them and running the kernel via calls to the appropriate abstract methods.

```
def initialize_array(self, *args, **kwargs): ...
def print_array_custom(self, array: list, dump_message: str = ''): ...
def kernel(self, *args, **kwargs): ...
def run_benchmark(self) -> list[tuple]: ...
```

**Figure 2.** Abstract functions to be implemented by a PolyBench/Python benchmark.

PolyBench/Python provides two different ways to measure performance. The first is to measure the execution time of the kernel using the Timestamp Counter (TSC register). Its contents are directly accessed using assembly code executed through the `inlineasm` library. The second way is to measure performance counters using the PAPI library [24, 25] through the `python_papi` module.

**3.2.1 Available Implementations.** The process of translating PolyBench/C to Python requires a number of design decisions to deal with the intrinsic differences between both languages. One of the critical differences is related to data representation. Polyhedral codes in general, and the PolyBench benchmarks in particular, manipulate arrays and scalar variables of basic types. In C, these are stored sequentially in memory in row-major order. There is no equivalent representation in pure Python, where everything is an object and there are no basic datatypes, in contrast to other interpreted languages, such as Java, where both concepts coexist. Since everything is an object, lists of `int` values are not a collection of contiguously stored 32- or 64-bit basic values, but rather a collection of contiguously stored 64-bit pointers to `int` objects. This creates an additional level of indirection which degrades performance when traversing the array.

Furthermore, when considering multidimensional structures, there is a choice between implementing them as a sequence of nested lists, similar to how a cascade of pointers to pointers would work in C, or flattening the structure and linearizing the accesses, as automatically done by C compilers with multidimensional array allocations. One can envision how the flattening should be more efficient, in the same way that in C using cascaded pointers introduces an additional level of indirection for each dimension in the data structure, degrading memory performance.

One additional alternative for array implementation is to directly use NumPy arrays [31]. This looks like a good design alternative, since NumPy arrays are, by design, C-like objects, with homogeneously-typed data, and contiguous in memory. This has the potential to greatly improve the memory behavior, and therefore performance, but it comes with its own performance pitfalls that need to be carefully studied.

In PolyBench/Python, the abstract `PolyBench` class that all benchmarks must extend implements all these different strategies for array allocation, exemplified in Fig. 3. The user must select the desired implementation using runtime knobs. These alternatives will be studied and compared in the experimental analysis in Sec. 4.

**3.2.2 Control Structures.** The PolyBench/C benchmarks prominently feature two control structures: `if` statements and `for` loops. The conditionals have a direct translation to Python, with no semantic and minimal syntactic variations. However, `for` loops in Python are *foreach* style loops that traverse a collection of objects. In order to implement them efficiently, a C `for` loop is translated to a Python `for` traversing a `range` expression. This is a special type of collection

```
for i in range(0, self.NI):
  for j in range(0, self.NJ):
    C[i][j] *= beta
  for k in range(0, self.NK):
    for j in range(0, self.NJ):
      C[i][j] += alpha * A[i][k]
                      * B[k][j]
```

**(a)** List

```
C *= beta
C += alpha * np.dot( A, B )
```

**(b)** NumPy

```
for i in range(0, self.NI):
  for j in range(0, self.NJ):
    C[self.NJ * i + j] *= beta
  for k in range(0, self.NK):
    for j in range(0, self.NJ):
      C[self.NJ * i + j] += alpha * A[self.NK * i + k]
                                  * B[self.NJ * k + j]
```

**(c)** Flattened List

**Figure 3.** List, flattened list, and NumPy alternative array implementations for the `gemm` kernel.

spawned by a Python `generator` object. This kind of collections are populated on demand, one object at a time, avoiding the memory overhead of instantiating the full collection.

### 3.3 Support for Polyhedral Optimizations

We have implemented an analysis and translation layer capable of reading Python kernels and generating their ScopLib [8] representation, as well as back-generating Python codes from the ScopLib. The input to this tool is not the Python source code, but the bytecode generated by CPython. This allows the optimizations to be performed in a just-in-time fashion, upon execution of the code, although this approach has not been used in the current work. However, no search or isolation of the static control part (SCoP) is performed at this time. The tool receives a Python function and assumes its entire body to be a valid SCoP.

### 3.4 Python Interpreters

Similar to using different compilers for C codes, a collection of different interpreters exist for Python. CPython [13] is developed by the Python Software Foundation. Its goal is not to achieve high performance, but to provide a multi-platform environment that serves as the reference interpreter for other projects.

PyPy [29] is a performance-oriented interpreter developed using the RPython [2] tool-chain. The Python code is translated to RPython, which is then translated to flow graphs, and then to C. The RPython layer includes a tracing just-in-time layer including an optimizer and a back-end that generates machine code.

Intel Distribution for Python [7] is designed to make Intel libraries such as the Math Kernel Library [33] and the Data Analytics Acceleration Library [6] usable from Python. It does not intend to provide fast pure Python code, but to bridge the technological gap between Python libraries such as NumPy and Intel products.

The performance of these interpreters will be compared in the next section.

## 4 Experimental Results

Experiments with the 30 PolyBench kernels were executed on an Intel Core i7 8700K with 64 GB of RAM memory. The CPU frequency was fixed at the base frequency of 3.7 GHz to prevent thermal constraints affecting experimental variability. We first analyze the performance of the Python version of the benchmarks, comparing different interpreters and using both nested lists and flattened lists to implement arrays. Afterwards, we focus on PyPy and flattened lists to assess the relative performance of Python and C codes. Then, we study the performance impact of basic polyhedral optimization on both C and Python codes. Finally, we study the potential performance improvements to be gained from using NumPy on CPython.

Our experimental study analyzes the performance of PolyBench/C 4.2.1-beta and PolyBench/Python. The tool-chain includes GCC 10.2.0, PoCC 1.5.0-beta, CPython 3.9.1, PyPy 7.3.2, and NumPy 1.19.4. All benchmarks are configured to use the default data types and dataset sizes, that is, the `LARGE` dataset size in PolyBench [28], where problem sizes typically far exceed L2 cache size. For each benchmark-configuration pair, selected PAPI hardware performance counters are collected during execution of the kernel, including those measuring hits and misses to each level of the memory hierarchy, execution cycles, total instructions executed, and stalled cycles. Instruction count is further broken down into several different instruction types, including memory instructions, branches, and floating point operations. For all experiments, we report the average of 5 runs. In all figures, performance counters are plotted relative to a baseline for simplicity and space. Complete experimental data, including absolute values for all measurements in the plots, are available as auxiliary material.

### 4.1 Relative Performance of Scalar Pure Python Implementations

We first focus on pure Python codes, and more specifically on evaluating the performance of the different Python interpreters considered in our experimental setup. We found that for pure Python (i.e., not NumPy codes, which will be covered later), the performance metrics for PyPy are an order of magnitude better than those of CPython and the Intel Distribution for Python. This includes memory performance, CPU stalls, and branches and branch mispredictions. The average speedup over the full PolyBench/Python suite is 20x. For the sake of space saving, we do not report these results, and in the following we default to PyPy unless otherwise noted.

Next, we study the tradeoffs of using nested lists versus flattened lists. Figure 4 summarizes key performance counters of the nested list-based implementation as compared to the flattened list-based one.

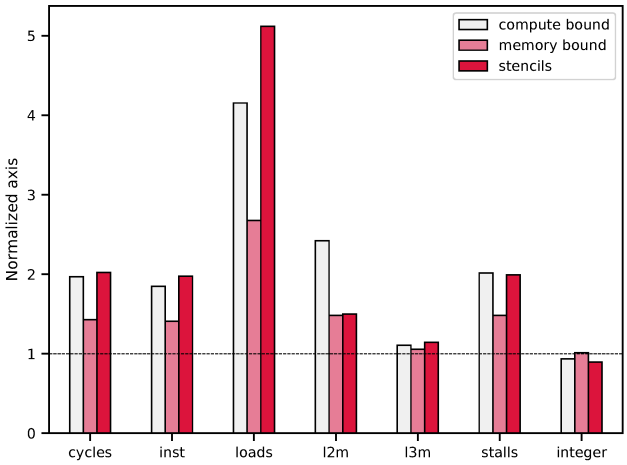


**Figure 4.** Selected performance counters of the implementation with nested lists, executed using PyPy, normalized to the values of the implementation with flattened lists.:

Using flattened lists (i.e., linearized arrays) always offers the best performance, with an average speedup of 2x. Codes using flattened lists execute, on average, 1.9x fewer instructions, distributed as 1.7x fewer branches (20% of the total instruction count reduction), and 4.4x fewer load instructions (80% of the total instruction count reduction). This large reduction in the number of L1 accesses is caused by the removal of the intermediate levels of indirection when employing nested lists, but it does not have a large fundamental impact on the number of off-chip accesses, as the cache hierarchy is capable of absorbing the excess loads. L2 misses decrease by 2.2x. Misses to L3, where the actual data reside, and consequently main memory accesses, are decreased by only 1.1x, an indication that the actual benefit from the memory point of view is the elimination of repeated accesses to intermediate indirection levels in nested lists. If we categorize the benchmarks into compute-bound, memory-bound, and stencils following their data reuse profiles as shown in Table 1, we see a stark difference between memory-bound and compute-bound benchmarks. While for memory-bound benchmarks the average speedup is 1.4x, it goes up to 2.0x for the compute-bound and stencil kernels. The reason is that many compute-bound and stencil benchmarks become memory-bound as the number of memory accesses is multiplied by a factor of up to five when using nested lists. By flattening the lists the reduction in memory accesses directly translates into a halving of the pipeline stalls.

PyPy provides an additional performance advantage when using flattened lists. Since PyPy 1.8, lists containing a collection of homogeneous objects, i.e, objects of the same Python type which corresponds to a C native type, will be internally stored as lists of native types, without a wrapping object [30]. Although this optimization is not identical to a native C array, as it still requires additional helper objects to implement operations on the list, a level of indirection is saved on the access to the data, improving overall performance. In the remainder of this section, all Python codes will be implemented through flattened lists, except where otherwise noted.

### 4.2 C vs Python

This subsection focuses on the relative performance of C codes with different optimization levels and pure Python codes executed using PyPy and flattened lists. Since there is wide variability in the performance characteristics of different benchmarks in the PolyBench set, we break the 30 kernels down into smaller, more easily analyzable subsets that present common traits. We find that, for this comparison, the classical categorization into compute-bound, memory-bound and stencil benchmarks is not appropriate. In this case a categorization which follows the relative success of the C benchmarks depending on the optimization levels applied is more useful to determine the reasons for performance differences between Python and C, and it gives insight into how the Python-to-machine translation process can be further improved to speed up executions. For the remainder of this subsection, we will use the term "baseline" to refer to C codes compiled with `gcc -O3 -fno-tree-vectorize`. Indeed, we aim to isolate the effect of SIMD vectorization in our experiments for fair comparison, as PyPy does not appear to fully exploit SIMD vectorization in the generated programs.

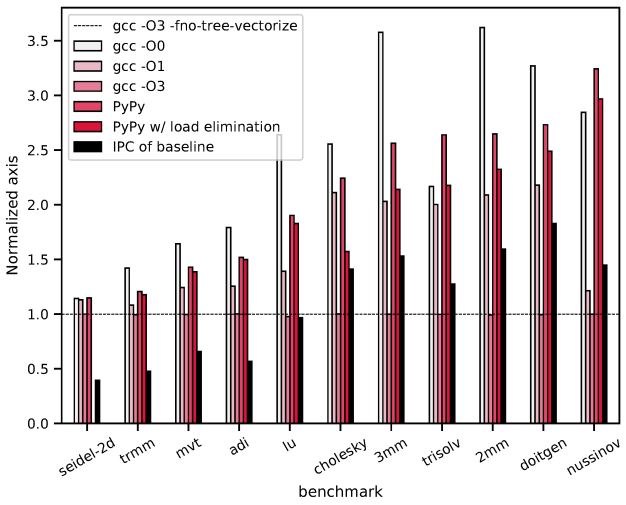


**Figure 5.** Performance, in execution cycles, of the benchmarks which benefit from `-O3` but not from SIMD instructions. Results are normalized to those of the baseline, and the rightmost column is its IPC.

Figure 5 shows the performance obtained by the subset of codes that benefit from `-O3` optimizations, but not from vectorization. As for the Python performance, we can see that

it is closely related to the performance of `gcc -O1`, except for `nussinov`, which will be isolated and studied later. The average speedup of `-O3` with respect to `-O1` in this subset is 1.4. This improvement comes from a reduction in the number of load instructions performed by the kernel. Applying differential analysis of performance optimizations, the culprit turns out to be load elimination on the innermost reductions, as exemplified in Fig. 6. The only exceptions are `adi`, where load elimination accounts for 30% of the improvement while `-fexpensive-optimizations`, a relatively opaque set of program analyses is responsible for the remaining 70%; and `seidel-2d`, where all the performance improvement comes from `-fpredictive-commoning`, which tries to reuse computations from previous iterations of a loop.

```
for c1 in range (N):
  for c2 in range ((c1-1)+1):
    x[c1] -= L[c1][c2] * x[c2]
  x[c1] = x[c1] / L[c1][c1]
```

**(a)**

```
for c1 in range (N):
  tmp_x = x[c1]
  for c2 in range ((c1-1)+1):
    tmp -= L[c1][c2] * x[c2]
  x[c1] = tmp / L[c1][c1]
```

**(b)**

**Figure 6.** Example of load elimination for a fragment of `trisolv`.

We manually applied load elimination to this set of Python benchmarks, finding an average reduction in the number of both load and store instructions of approximately 25%, and in the number of total instructions executed of 11%, for a total average speedup of 1.1.

Regarding performance, the Python version is, on average, 1.6x slower than the baseline, and only 1.2x slower than `-O1`. With the exception of `nussinov`, the execution cycles of all benchmarks present a correlation of 0.99 with the number of executed instructions. This is not at all the case for C codes, where this correlation is approximately 0.60. This signals that, for an interpreted language such as Python, where executing each instruction includes expensive operations such as types and dimensionality checks, the number of executed instructions determines, to a very large extent, the attainable performance. In fact, the relative performance of Python codes is closely related to the IPC (Instructions Per Cycle) of the C codes, as shown in the IPC column of Figure 5. When the IPC is low, the slack available to the Python VM to execute each instruction is larger, and the effect on total execution time of the interpreting overhead is hidden. As the number of instructions per cycle increases, this overhead starts to become the bottleneck of the system. In the `nussinov` case, the number of L3 misses for this benchmark increases by 13x, an anomaly compared to the average 1.3x of this set of benchmarks. This signals that this benchmark, which has a very low L3 miss rate of 1.6% in the C version, becomes memory bound due to conflict caused by the extra memory pressure due to the interpreting process. The L3 miss rate in the Python version scales up to 8.3%.

The second subset of analyzed codes benefits from both `-O3` optimizations and the use of SIMD instructions, as detailed in Fig. 7. With the exception of `deriche`, `gemver`, and `atax`, the kernels in this group are either stencils or compute-bound.

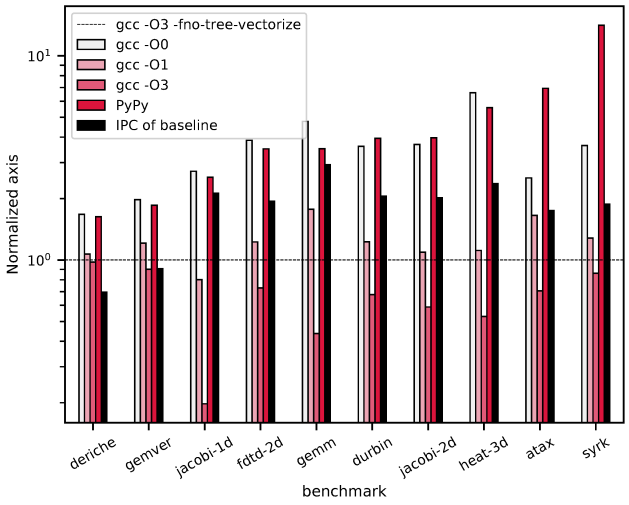


**Figure 7.** Performance, in execution cycles, of the benchmarks which benefit from `-O3` and the generation of SIMD instructions. Results are normalized to those of the baseline, and the rightmost column is its IPC. Note that the Y axis is logarithmic.

The Python VM overhead manifests more clearly for these codes. The average slowdown with respect to the baseline goes up to 5.2, and to 4.3 with respect to `-O1`. The total executed instructions increase by 7.1x. When compared to the `-O3` versions generating SIMD operations, the number of executed instructions scales up to 25.3x, and the slowdown to 8.5. The evolution of the overhead introduced by PyPy still has a high correlation to the number of instructions per cycle (IPC) of the baseline. Some exceptions appear, related to particular compiler optimizations being introduced by GCC which are not done by PyPy. For instance, in the worst performance case, `syrk`, GCC is performing a loop interchange, followed by loop fusion, and load elimination, as shown in Fig. 8. When manually applied to the Python code, these optimizations achieve a reduction in the amount of executed instructions of 5.2x, and a combined speedup of 7.4. The slowdown with respect to the baseline is limited to 2x, showing that, when similarly optimized, the performance of PyPy usually comes within the same order of magnitude as the performance of equivalent C codes.

Finally, we study the subset of codes for which GCC optimizations do not significantly improve the execution time. These are shown in Fig. 9, and include `gramschmidt`, the only benchmark for which the performance of PyPy is higher, if only by 5%, than the performance of the baseline.

In the absence of advanced GCC optimizations, a few benchmarks, in particular `gesummv`, `covariance`, `syr2k`, and

```
for i in range(self.N):
  for j in range(i + 1):
    C[i][j] *= beta
  for k in range(self.M):
    for j in range(i + 1):
      C[i][j] += alpha * A[i][k]
                      * A[j][k]
```

(a)

```
for i in range(self.N):
  for j in range(i + 1):
    tmp_c = C[i][j] * beta
    for k in range(self.M):
      tmp_c += alpha * A[i][k]
                     * A[j][k]
    C[i][j] = tmp_c
```

(b)

**Figure 8.** `syrk` code: (a) original, (b) after loop interchange and fusion, and load elimination.

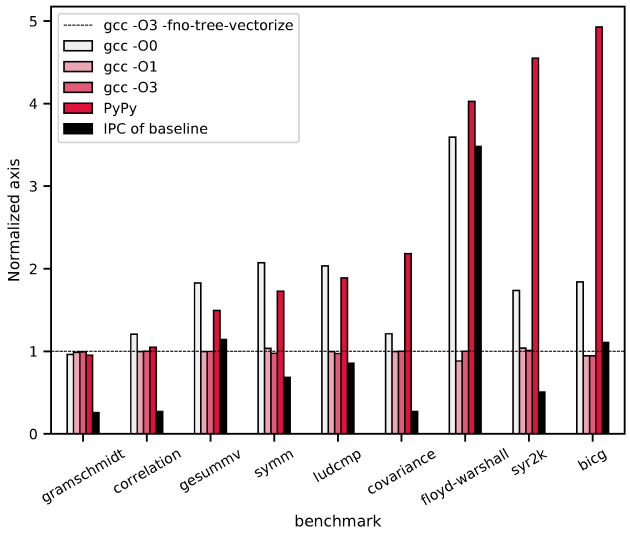


**Figure 9.** Performance, in execution cycles, of the benchmarks which do not benefit from advanced GCC optimizations. Results are normalized to those of the baseline, and the rightmost column is its IPC.

`bicg` do not seem to follow the rule of thumb that the overhead of Python follows the IPC of the C code. The problem with `bicg` is related to JIT misoptimization. In fact, if the inner loop of the kernel is interchanged, counter-intuitively traversing the 2-dimensional array through its columns, performance is increased by 1.7x. This anomalous behavior disappears as soon as either the `unroll` or `heap` JIT optimizations in PyPy are turned off. In the case of `syr2k`, the problem lies in an abnormal increase in the number of L3 misses of 4.9x. The benchmark, which was compute-bound in the C version, has become memory bound due to conflict misses in L1 caused by the extra memory pressure due to the interpreting process.

### 4.3 Polyhedral Optimizations

Figure 10 presents the relative performances, per benchmark, of 4 different PyPy implementations. The base implementation in PolyBench/Python (`PyPy`) is contrasted with three optimization approaches in the PoCC compiler: `maxfuse` implements the original Pluto algorithm [4] with maximal loop fusion for the kernel, resulting in possibly complex loop nest expressions when e.g. loop skewing is required to enable fusion. `fusion` is fusing statements surrounded by the same number of loops only, that is the legacy "smartfuse" heuristic of Pluto [4]. `vectorizer` implements also smart fusion, but adds additional loop permutations to expose when possible parallel inner loops with the most stride-1 accesses, i.e., favors spatial locality.

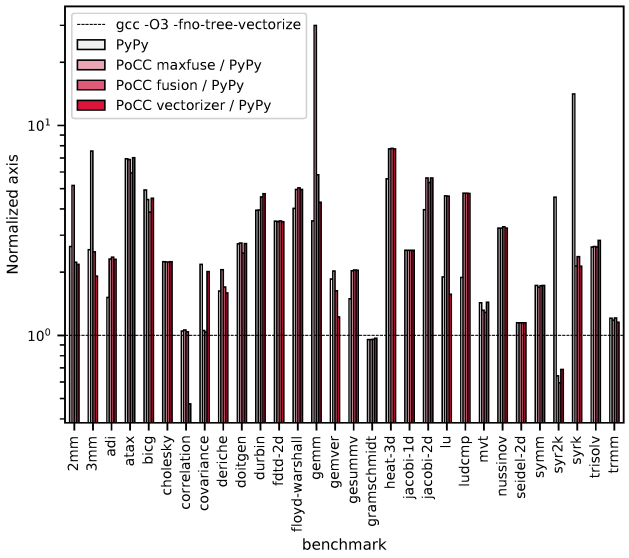


**Figure 10.** Performance, in execution cycles, of different polyhedral optimizations.

While we observe numerous instances where the base PyPy version is outperformed by one or more versions using polyhedral transformations, such as for `gemver`, in most cases these optimizations do not provide additional performance improvement. For numerous benchmarks, this is to be expected: we did not implement any tiling in these experiments, and datasets typically exceed the L3 cache size, preventing to fully implement data reuse in the cache, e.g. as is observed for the various stencil computations. However attempting to derive a simple explanation based on the computational and data reuse patterns of the benchmarks would fail: we must instead take into account the overhead of PyPy that is added to the benchmarks, as discussed previously. Figure 11 displays aggregated hardware counter metrics over all 30 benchmarks, illustrating the PyPy overhead in terms of number of instructions executed often makes the code "worse" than GCC -O0, which implements a spill-everywhere approach.

While instruction count is on average increased by 6x when using PyPy flattened lists (that is exactly the PyPy bar in Fig. 10), there is a wide ranging increase over the benchmarking suite. For example for `gemver`, the number of executed instructions grows by 7x and the number of branches grows by 9x. The instruction type distribution is altered for the PyPy version: loads represent 32% of instructions executed in the baseline C implementation, but only

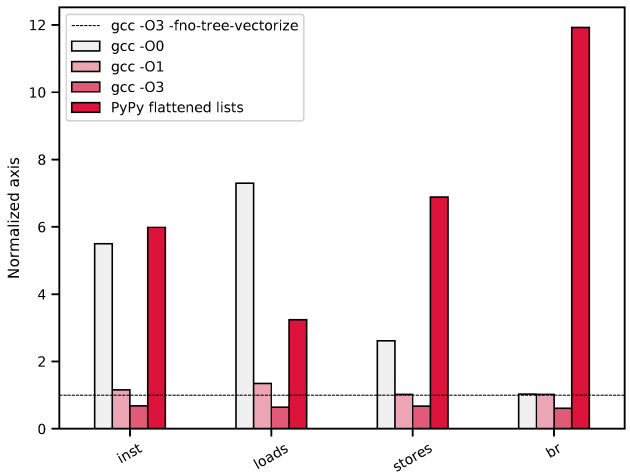


**Figure 11.** Geometric mean of the ratio of different types of instructions for different versions of the benchmarks, normalized to that of the baseline.

20% in the PyPy version. But looking at `jacobi-2d`, instruction count increases by 26x, branches by 75x, loads by 20x and stores by 24x. Such overhead cannot be compensated by traditional loop transformations, irrespective of which loop order/fusion is implemented. Across all benchmarks, instruction count increases always by 4x or more (`gesummv` being the outlier, at 3.3x). Surprising differences occur, e.g., for `gemm` instructions increase by 20x, but for `2mm` by 4x only. This increase alone explains the difficulty to reach performance comparable to the native C implementation in many cases, and the differences across benchmarks with similar compute and reuse patterns (e.g., `gemm` vs. `2mm`) prevent from finding easily a performance model to predict the PyPy overhead. In general, of the various instances where polyhedral transformations have improved performance, we observed mostly a reduction in instruction count and overall PyPy overhead added in these variants compared to the base PyPy version.

### 4.4 NumPy: Loop-based vs. Vectorized Operation

The core of the NumPy routines is directly implemented in C, both for performance reasons and to allow to easily pass data between applications. While the interfacing of CPython and C subroutines is seamless and provides native performance, interfacing PyPy with C codes requires ad-hoc extensions to bridge the gap between the C representation and the internal RPython representation used by PyPy. In particular, this requires a module, called `cpyext`, to make RPython's garbage collector aware of the objects managed by the C layer [3]. Due to this, PyPy is on average 3x slower than CPython on NumPy codes and, for this reason, we will use CPython as the default interpreter in this section.

The performance of NumPy codes is heavily conditioned by the coding style, and more specifically, on what type of operations are performed on the NumPy data. Although simply writing C-like code using the `ndarray` class provided by NumPy could seem like an easy way to improve execution performance, exactly the opposite is true. When NumPy arrays are traversed using regular loops accessing individual scalar elements, the basic data types stored in the `ndarray` object need to be promoted to first-class Python objects, which is the only kind of data that can actually be manipulated by the Python VM.

In order to improve NumPy performance, it is necessary to write code in a "vectorized" fashion. In the context of NumPy, the term vectorization does not refer to issuing SIMD instructions, but rather to transferring control of a particular operation (e.g., matrix-matrix addition or multiplication) to a native C, highly efficient implementation. Given that NumPy arrays are homogeneous, there is no need to call the VM interpreter while inside the C implementations, and consequently fewer instructions are executed, efficiently issuing SIMD operations and even allowing for multi-threaded implementations (although NumPy developers have opted not to do so for design reasons). Figure 12 presents the NumPy performance compared to PyPy using flattened lists, and with the `-O3` versions. As can be observed, for many benchmarks the performance offered by the NumPy codes is largely superior to that of the C codes. Such is the case with `gramschmidt`, where the speedup for the NumPy version with respect to the best C version is 3.3. However, for some other benchmarks the performance of NumPy lags far behind, such as with `heat-3d`, in which it delivers a 17.3x slowdown with respect to the `-O3` execution.

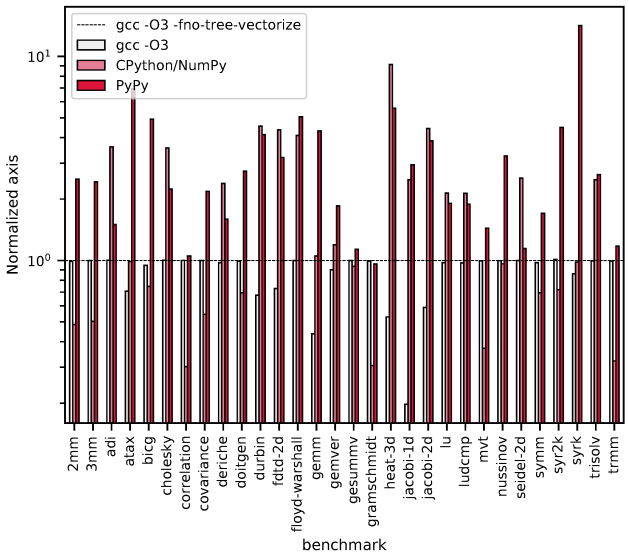


**Figure 12.** Performance, in execution cycles, of NumPy benchmarks compared to C and pure Python versions. Note that the Y axis is logarithmic.

The reasons for these performance differences are varied. In the case of `heat-3d`, `jacobi-2d` and other similar stencil codes, the problem lies within the memory management performed by NumPy. Upon vectorizing a stencil operation, NumPy will replicate the data in the original buffer to achieve efficient operation. For large matrices, this might cause a significant degradation of their memory performance. For example, for `jacobi-1d`, a 3-point stencil, the number of L1 misses is increased by 20.4x. However, since its footprint comfortably fits L3, this increase does not scale to lower levels of the hierarchy and the large reduction in total number of instructions (1.64x), driven by the use of 256-bit packed floating point operations, achieves a net 1.2 speedup. When working with larger stencils this behavior degrades. For `jacobi-2d`, a 5-point stencil illustrated in Fig. 13, the number of L3 misses increases by 4.6x, and for `heat-3d`, a 9-point stencil, it increases by 10x. This seems to indicate that NumPy is replicating the original buffer for each of the shifted matrices in the stencil computation, causing slowdowns of 1.1 and 1.6, respectively. However, the total number of instructions executed decreases by a factor of 4.2x for `jacobi-1d`, and 1.8x for `heat-3d`. In both cases, all floating-point operations are executed using 256-bit packed SIMD instructions.

```
for t in range( self.TSTEPS ):
  B[1:self.N-1, 1:self.N-1] = 0.2 * (A[1:self.N-1, 1:self.N-1]
                                   + A[1:self.N-1, 0:self.N-2]
                                   + A[1:self.N-1, 2:self.N]
                                   + A[2:self.N,   1:self.N-1]
                                   + A[0:self.N-2, 1:self.N-1])

  A[1:self.N-1, 1:self.N-1] = 0.2 * (B[1:self.N-1, 1:self.N-1]
                                   + B[1:self.N-1, 0:self.N-2]
                                   + B[1:self.N-1, 2:self.N]
                                   + B[2:self.N,   1:self.N-1]
                                   + B[0:self.N-2, 1:self.N-1])
```

**Figure 13.** NumPy version of `jacobi-2d`.

Other benchmarks require special code transformations to be vectorized, as they present loop-carried dependences. For instance, it is necessary to perform loop interchanges in order to vectorize the innermost loop of `adi` and `deriche`. For other benchmarks with more complex dependences, such as `seidel-2d` or `nussinov`, index-set splitting [16] may be employed to expose parallelism. These transformations modify the original sequential memory access of the kernels, consequently damaging both temporal and spatial locality, and may even cause performance to degrade with respect to the non-vectorized PyPy version. They are, however, fundamental to achieving performance with NumPy. For example, if non-vectorized, `adi` will execute 135x more instructions, 50% of them loads and stores. The memory behavior is improved at the L2 level, as array traversals are in row-major order. However, the final performance is degraded by a factor of 80 with respect to the vectorized version. For benchmarks which can be vectorized without hindering locality, and that do not operate on a large number of array views at the same time, the performance of NumPy matches, or even beats, the performance of the best C version.

## 5 Related Work

There is an immense body of work relating to Python performance measurement and optimizations, we limit below to highlighting several key software and publications, and their differences with the present work.

***Python Benchmarking Suites.*** There is a plethora of benchmarking suites written for Python, including specifically to evaluate the quality and performance of interpreters. The Python Performance Benchmarking Suite [34] integrates numerous applications and kernels, including (some synthetic) kernels to measure float and integer-heavy operations. However, nearly none of the algorithms implemented in PolyBench kernels are available. The PyPy benchmarking suite [9] similarly includes a high number of applications and some numerical computation programs, but does not provide the type of implementations we offer in PolyBench/Python. The Pythran project released Numpy-style implementations of numerous scientific kernels [19], several also available in PolyBench. While these benchmark suites tend to focus on full applications, PolyBench/Python has been designed to specifically cover a spectrum of regular numerical kernels that are amenable to polyhedral optimizations *and* systematically provide 3 implementation flavors for each benchmark, including Numpy-style.

***Python Environments and Compilers.*** Similar as to using different compilers for C codes, a collection of different interpreters exist for Python. CPython [13] is developed by the Python Software Foundation. The Intel Distribution for Python [7] is designed to make Intel libraries such as the Math Kernel Library [33] and the Data Analytics Acceleration Library [6] usable from Python. Its aim is to bridge the technological gap between Python libraries such as NumPy and Intel products. PyPy [29] is a performance-oriented interpreter developed using the RPython [2] tool-chain. The Python code is translated to RPython, which is then translated to flow graphs, and then to C. The RPython layer includes a tracing just-in-time layer including an optimizer and a back-end that generates machine code.

The Pythran compiler [18, 20] is a powerful optimizing compilation flow for (a subset of) Python that compiles programs into a C++ implementation for subsequent native execution. Pythran is an ahead-of-time compilation approach, which supports a variety of Python and Numpy concepts. It supports multi-threading.

## 6 Concluding Remarks

This paper introduced PolyBench/Python, a polyhedral benchmarking suite for Python environments, and presented extensive experimental analysis. Our experiments show that the performance of PyPy is usually an order of magnitude better than that of CPython. In fact, PyPy provides performance of the same order of magnitude than C for most kernels, when

disabling static compiler optimizations (i.e., using `-O0`). We observed high correlation between the relative performance of Python codes and the IPC of the C baselines. When the IPC is low, the slack available to the Python VM to execute each instruction is larger, and the effect on total execution time of the interpreting overhead is hidden, bringing Python performance closer to C. Besides the overhead introduced by the interpreting process itself, another important factor to explain the relative performance of Python and C codes is the absence of static or dynamic optimizations in the Python execution stack. Even though scientific-oriented interpreters, such as PyPy, perform high amounts of JIT optimizations, these usually target type inference and reducing the number of calls to the Python interpreter. Very few high-level code optimization is performed, specially given that most Python interpreters, including CPython and PyPy, are stack-based. While this reduces the complexity of the interpreter, which can just execute isolated pieces of code, it complicates the introduction of even very simple optimizations such as loop-invariant code motion or load elimination. Our results have shown how these very simple optimizations have potential to significantly improve the performance of Python codes, and suggest that these could be implemented in modern interpreters in a just-in-time fashion. For instance, load elimination could be implemented by detecting that the same array position is being repeatedly loaded from memory inside a loop, and dynamically enabling its lowering to a scalar during runtime. SIMDization is another optimization that could be dynamically enabled in a speculative fashion, by detecting floating point operations to consecutive positions of an array at runtime and fusing them together after a given number of iterations. By adding advanced SIMDization capabilities to performance-oriented Python interpreters such as PyPy, we can expect a substantial performance increase. The vectorization itself compounds with the reduction in the number of issued instructions to be interpreted, fundamentally in the number of memory accesses and branches.

We have also shown how NumPy can achieve performance superior to that of native C codes, provided that no locality tradeoffs are required in order to vectorize the code. These tradeoffs manifest when loop-carried dependences prevent row-order vectorization, requiring column or even wavefront traversals of the data.

## Acknowledgments

This research was supported in part by the Ministry of Science and Innovation of Spain (PID2019-104184RB-I00 / AEI / 10.13039/501100011033), and by the U.S. National Science Foundation (award CCF-1750399). CITIC is funded by Xunta de Galicia and FEDER funds of the EU (Centro de Investigación de Galicia accreditation, grant ED431G 2019/01).

## References

[1] [n.d.]. *PLUTO - An automatic parallelizer and locality optimizer for multicores.* http://pluto-compiler.sourceforge.net

[2] D. Ancona, M. Ancona, A. Cuni, and N.D. Matsakis. 2007. RPython: a step towards reconciling dynamically and statically typed OO languages. In *Proceedings of the 2007 Symposium on Dynamic Languages (DLS'07)*. 53–64.

[3] S. Beyer. 2020. *Efficient cycle detection on a partially reference counted heap.* Master's thesis. Technische Universität Wien.

[4] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. 2008. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, PLDI*. 101–113.

[5] Brad Chapman and Jeffrey Chang. 2000. Biopython: Python tools for computational biology. *ACM Sigbio Newsletter* 20, 2 (2000), 15–19.

[6] Intel Corporation. [n.d.]. *Intel Data Analytics Acceleration Library.* https://software.intel.com/content/www/us/en/develop/tools/data-analytics-acceleration-library.html

[7] Intel Corporation. [n.d.]. *Intel Distribution for Python.* https://software.intel.com/content/www/us/en/develop/tools/distribution-for-python.html

[8] Louis-Noel Pouchet et al. [n.d.]. PoCC, the Polyhedral Compiler Collection 1.5. http://pocc.sourceforge.net.

[9] Maciej Fijalkowski et al. 2020. PyPy Performance Benchmarks. https://foss.heptapod.net/pypy/benchmarks.

[10] Inc. Facebook. [n.d.]. PyTorch, an open source machine learning framework. https://pytorch.org.

[11] Paul Feautrier. 1991. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming* 20, 1 (1991), 23–53.

[12] P. Feautrier. 1992. Some efficient solutions to the affine scheduling problem, part II: multidimensional time. *International Journal of Parallel Programming* 21, 6 (1992), 389–420.

[13] Python Software Foundation. [n.d.]. *The Python programming language.* https://github.com/python/cpython

[14] Python Software Foundation. 2020. *Sunsetting Python 2.* https://www.python.org/doc/sunset-python-2/

[15] Sylvain Girbal, Nicolas Vasilache, Cédric Bastoul, Albert Cohen, David Parello, Marc Sigler, and Olivier Temam. 2006. Semi-Automatic Composition of Loop Transformations. *International Journal of Parallel Programming* 34, 3 (June 2006), 261–317.

[16] Martin Griebl, Paul Feautrier, and Christian Lengauer. 2000. Index set splitting. *International Journal of Parallel Programming* 28, 6 (2000), 607–631.

[17] Tobias Grosser, Armin Groesslinger, and Christian Lengauer. 2012. Polly—performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters* 22, 04 (2012), 1250010.

[18] Serge Guelton. 2018. Pythran: Crossing the Python Frontier. *Computing in Science & Engineering* 20, 2 (2018), 83–89.

[19] Serge Guelton. 2020. Pythran Numpy Benchmarks. https://github.com/serge-sans-paille/numpy-benchmarks/.

[20] 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.

[21] François Irigoin and Rémi Triolet. 1988. Supernode partitioning. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 319–329.

[22] Nikhil Ketkar. 2017. Introduction to pytorch. In *Deep learning with python*. Springer, 195–208.

[23] Martin Kong, Richard Veras, Kevin Stock, Franz Franchetti, Louis-Noël Pouchet, and Ponnuswamy Sadayappan. 2013. When polyhedral transformations meet SIMD code generation. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*. 127–138.

[24] P.J. Mucci, S. Browne, C. Deane, and G. Ho. 1999. PAPI: A Portable Interface to Hardware Performance Counters. In *Proceedings of the*

*department of defense HPCMP users group conference*, Vol. 710.

[25] P. Mucci and The ICL Group. [n.d.]. Performance Application Programming Interface. http://icl.cs.utk.edu/papi/people/index.html..

[26] Sébastian Pop, Albert Cohen, Cédric Bastoul, Sylvain Girbal, Geogres-André Silber, and Nicolas Vasilache. 2006. GRAPHITE: Loop optimizations based on the polyhedral model for GCC. In *Proc. of the 4þ GCC Developper's Summit*.

[27] Louis-Noël Pouchet, Uday Bondhugula, Cédric Bastoul, Albert Cohen, J. Ramanujam, P. Sadayappan, and Nicolas Vasilache. 2011. Loop Transformations: Convexity, Pruning and Optimization. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '11)*. ACM, New York, NY, USA, 549–562. https://doi.org/10.1145/1926385.1926449

[28] Louis-Noel Pouchet and Tomofumi Yuki. [n.d.]. PolyBench/C 4.2.1. http://polybench.sourceforge.net.

[29] The PyPy Team. [n.d.]. *PyPy*. https://www.pypy.org

[30] The PyPy Team. 2012. *PyPy 1.8 release notes*. https://doc.pypy.org/en/latest/release-1.8.0.html

[31] S. van der Walt, S.C. Colbert, and G. Varoquax. 2011. The NumPy array: a structure for efficient numerical computation. *Computing in Science & Engineering* 13, 2 (2011), 22–30.

[32] Sven Verdoolaege. 2010. ISL: An integer set library for the polyhedral model. In *The 3rd International Congress on Mathematical Software (ICMS'10)*. Springer.

[33] E. Wang, Q. Zhang, B. Shen amd G. Zhang, X. Lu, Q. Wu, and Y. Wang. 2014. *Intel Math Kernel Library*. Springer International Publishing, 167–188.

[34] Collin Winter and Jeffrey Yasskin. 2020. The Python Performance Benchmark Suite. https://pyperformance.readthedocs.io.