The Journal of Open Source Software

Pyccel: a Python-to-X transpiler for scientific high-performance computing

Emily Bourne [1*], **Yaman Güçlü** [2*¶], **Said Hadjout** [2,3*], **and Ahmed Ratnani** [4*]

1 CEA, IRFM, F-13108 Saint-Paul-lez-Durance, France **2** NMPP division, Max-Planck-Institut für Plasmaphysik, Garching bei München, Germany **3** Dept. of Mathematics, Technische Universität München, Garching bei München, Germany **4** Lab. MSDA, Mohammed VI Polytechnic University, Benguerir, Morocco ¶ Corresponding author * These authors contributed equally.

DOI: 10.21105/joss.04991

Software

- Review
- Repository
- Archive

Editor: Daniel S. Katz

Reviewers:

- @aholmes
- @IgorBaratta
- @boegel

Submitted: 01 December 2022
Published: 09 March 2023

Summary

The Python programming language has gained significant popularity in scientific computing and data science, mainly because it is easy to learn and provides many scientific libraries, including parallel ones. While these libraries are very fast, they are usually written in compiled languages such as Fortran and C/C++. User code written in pure Python is usually much slower; because Python is a dynamically typed language which introduces overhead in many basic operations. Due to this limitation, one often needs to rewrite the computational parts of their Python code in a statically typed language, to take full advantage of optimization and acceleration techniques. This expensive process happens naturally during the transition from a prototype to a production code, which is the principal bottleneck in scientific computing. We believe that such a bottleneck can be resolved, or at least drastically reduced, through the use of automatic code generation tools.

In this work we present Pyccel, a Python library which acts as a transpiler by translating Python code to either Fortran or C code, and as an accelerator by making the generated code callable from Python once again. Not only is the Pyccel-generated Fortran or C code very fast, but it is human-readable; hence the expert programmer can easily profile the code on the target machine and further optimize it. Pyccel provides a variety of methods for the efficient usage of the available hardware resources, such as type annotations, function decorators, and OpenMP pragmas. Moreover, Pyccel allows the user to link their code to external libraries written in the target language.

Statement of need

Different approaches have been proposed to accelerate computation-intensive parts of Python code. Cython (Behnel et al., 2011), one of the first tools of this kind, allows the user to call the Python C API by introducing a static typing approach. However, the user must re-write their code into a hybrid Python-C language in order to remove expensive Python callbacks from the generated C code. As a result, the code can no longer be executed using the Python interpreter alone. A more recent tool is Pythran (Guelton et al., 2015), which allows dynamic Python code to be converted into static C++ code by providing types as comments. The HOPE (Akeret et al., 2015) library provides a just-in-time (JIT) compiler to convert Python code to C++, where the arguments' types are only known at execution time. Numba (T. Olifant et al., n.d.) follows the same idea of bringing JIT compiling to Python by generating machine code based on LLVM, which can run on either CPUs or GPUs. Both Numba and HOPE rely heavily on the use of simple decorators to instruct the Python package to compile

a given function. They also use the type information available at runtime to generate byte code. A different approach is given by PyPy (Bolz et al., 2009), a Python interpreter written in an internal language called RPython (which is a restricted subset of the Python language itself). The aim of PyPy is to provide speed and efficiency at runtime using a JIT compiler.

To the authors' knowledge, of all the different methods used to accelerate Python codes, nothing so far generates human readable code. In this work, we present a new Python static compiler named Pyccel which combines a transpiler with a Python/C API to create an accelerator. This approach has two main advantages. Firstly, it leaves the user the option of optimising the code further in the low-level language with the help of HPC specialists. Secondly, it allows the user to choose the language the most adapted to their problem or system. For example, Fortran is a language designed for scientific programming and is tailored for efficient runtime execution on a wide variety of processors. The compiler is therefore highly effective for array handling in the context of scientific programming. In contrast, the C compiler is more adapted to support GPU tools such as CUDA, and OpenACC.

Pyccel is designed for two different use cases: (1) accelerate Python code by converting it to Fortran and providing a CPython wrapper to interface between the low-level and high level languages, (2) generate low-level C or Fortran code from Python code. The latter case follows from the fact that the code is human-readable. This means that Pyccel can also be used to simplify the process of going from a prototype (which is often written in inefficient languages which are quick to write) to production code (written in a low-level language). To this end, Pyccel is designed to allow the use of low-level legacy codes and some Python scientific libraries such as numpy, scipy, etc.

Benchmarks

A few example codes are used to provide an indication of the performance of Pyccel as compared to the popular accelerators Numba and Pythran. The source code can be found in github.com/pyccel/pyccel-benchmarks. These examples, which illustrate several common scientific computing problems, are based on open-source code samples (Barba, n.d.; Burkardt, n.d.). All tests were run in single-threaded mode on a CPU compute node of the HPC system Raven (Max Planck Computing and Data Facility, n.d.), featuring an Intel Xeon IceLake-SP 8360Y processor with 72 cores and 256 GB RAM. The tests were run with Python 3.9.7 on Ubuntu SUSE Linux Enterprise Server 15 SP3, using Pyccel 1.7.2, Numba 0.56.4, and Pythran 0.12.1. The following flags were passed to GCC 12.1.0 via Pyccel and Pythran : `-O3 -march=native -mtune=native -mavx`. The Numba test cases were compiled using the `@njit` decorator.

Figure 1 shows the time required to execute the accelerated code for these test cases. We see that Pyccel is highly competitive in all cases, but unfortunately Pyccel's C printing is slightly less developed than the Fortran printer, leading to less performant code. The finite difference Laplace test case (FD-Laplace) relies heavily on Numpy vectorized expression, and is the hardest to optimize for all accelerators. In this test only Pyccel can provide a substantial speedup (about 68%) using Fortran as a backend language; Pythran and Pyccel (C) provide a marginal speedup of 13% and 4%, respectively, while Numba is slower than the original Python code.

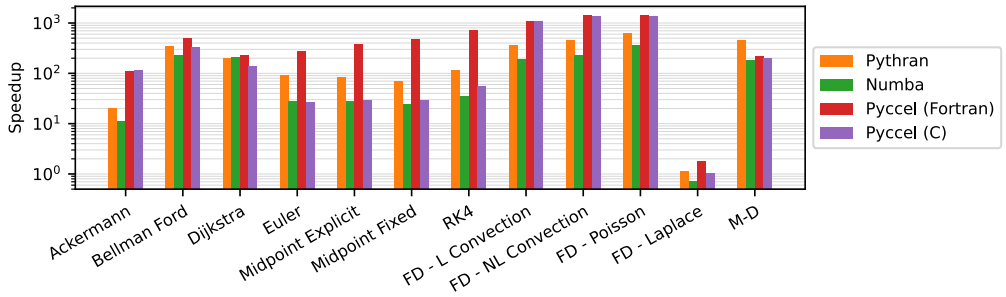

Figure 1: Comparison of speed-up compared to Python, obtained using accelerated code for various test cases executed with Python 3.9.7

Another important consideration is the time spent waiting for the accelerated version to be generated. This is shown in Figure 2, where Pyccel proves to be competitive with Numba while it outperforms Pythran significantly for large files.

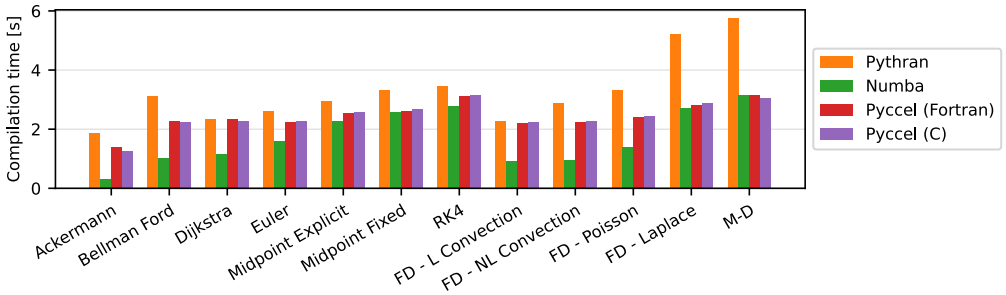

Figure 2: Comparison of times required to generate accelerated code for various test cases with Python 3.9.7

Acknowledgments

The authors would like to thank all the people who have contributed to Pyccel so far. The project has received funding from the European Union's Horizon 2020 Research and Innovation Program under Grant Agreement No. 800945 (Numerics PhD Program), and under Grant Agreement No. 676629 (Energy oriented Centre of Excellence for computing applications - EoCoE).

References

Akeret, J., Gamper, L., Amara, A., & Refregier, A. (2015). HOPE: A Python just-in-time compiler for astrophysical computations. *Astron. Comput.*, *10*, 1–8. https://doi.org/10.1016/j.ascom.2014.12.001

Barba, L. A. (n.d.). https://lorenabarba.com/blog/cfd-python-12-steps-to-navier-stokes/.

Behnel, S., Bradshaw, R., Citro, C., Dalcin, L., Seljebotn, D. S., & Smith, K. (2011). Cython: The best of both worlds. *Comput Sci Eng*, *13*(2), 31–39. https://doi.org/10.1109/MCSE.2010.118

Bolz, C. F., Cuni, A., Fijalkowski, M., & Rigo, A. (2009). Tracing the meta-level: PyPy's tracing JIT compiler. *Proceedings of the 4th Workshop on the Implementation, Compilation,*

Optimization of Object-Oriented Languages and Programming Systems, 18–25. https://doi.org/10.1145/1565824.1565827

Burkardt, J. (n.d.). https://people.sc.fsu.edu/~jburkardt/py_src/py_src.html.

Guelton, S., Brunet, P., Amini, M., Merlini, A., Corbillon, X., & Raynaud, A. (2015). Pythran: enabling static optimization of scientific Python programs. *Comput. Sci. Discov.*, *8*(1), 014001. https://doi.org/10.1088/1749-4680/8/1/014001

Max Planck Computing and Data Facility. (n.d.). *The supercomputer Raven*. https://www.mpcdf.mpg.de/services/supercomputing/raven.

T. Olifant et al. (n.d.). *Numba*. http://numba.pydata.org.