Efficient Compilation of High Level Python Numerical Programs with Pythran

Serge Guelton
Télécom Bretagne
serge.guelton@telecom-bretagne.eu

Pierrick Brunet
INRIA/MOAIS
pierrick.brunet@inria.fr

Mehdi Amini

Abstract

The Python language [5] has a rich ecosystem that now provides a full toolkit to carry out scientific experiments, from core scientific routines with the Numpy package[3, 4], to scientific packages with Scipy, plotting facilities with the Matplotlib package, enhanced terminal and notebooks with IPython. As a consequence, there has been a move from historical languages like Fortran to Python, as showcased by the success of the Scipy conference.

As Python based scientific tools get widely used, the question of High performance Computing naturally arises, and it is the focus of many recent research. Indeed, although there is a great gain in productivity when using these tools, there is also a performance gap that needs to be filled.

This extended abstract focuses on compilation techniques that are relevant for the optimization of high-level numerical kernels written in Python using the Numpy package, illustrated on a simple kernel.

1. Optimizations Opportunities in a Typical Numpy Kernel

This section briefly presents the core concepts of the Numpy package, then goes through all the optimization opportunities in a small kernel as a showcase of the optimization opportunities.

1.1 Numpy

The reference implementation of Numpy is a native Module, written mostly in C. It uses the BLAS API whenever possible and provides a relatively efficient array abstraction in the form of the `ndarray` data structure.

It also enforces a high-level programming style but it's very inefficient when explicit subscripts are used.

1.2 The Rosenbrock Function

We use the kernel illustrated in Listing 1 and adapted from the Scipy source code of `scipy.optimize.rosen` as a leading example. It uses Numpy's `sum` function, Python's square notation and Numpy's array slicing. It is a good example of high-level Python kernel, although using the original function directly would naturally make sense.

Note that due to dynamic typing, this function can take arrays of different shapes and types as input.

```
def rosen(x):
    t0 = 100 * (x[1:] - x[:-1] ** 2) ** 2
    t1 = (1 - x[:-1]) ** 2
    return numpy.sum(t0 + t1)
```

Listing 1: High-level implementation of the Rosenbrock function in Numpy.

1.3 Temporaries Elimination

In Numpy, any point-to-point array operation allocates a new array that holds the computation result. This behavior is consistent with many Python standard module, but it is a very inefficient design choice, as it keeps on polluting the cache with potentially large fresh storage and adds extra allocation/deallocation operations, that have a very bad caching effect. In the *rosen* function from Listing 1, 7 temporary arrays are allocated (slicing does not create a temporary array but a view) to hold intermediate steps. Had the expression been evaluated lazily, no temporary would have been needed.

1.4 Operator Fusion

As Numpy is a native library mostly written in C, each operator computation is performed by a function implemented as a loop performing a single operation, and the operator chaining is done at the interpreter level. This is a typical problem in library design: if only a small set of functions is provided, it prevents the optimization of merging multiple operators into a single specialized operator. On the contrary, providing many operator combinations as part of the library yields better performance to the price of API bloat. Listing 1 illustrates the use of a small set of functions: a loop is generated for each temporary computation, plus an extra loop for the `numpy.sum` reduction, whereas a single loop would have been necessary with operator fusion.

1.5 Loop Vectorization and Parallelization

Without operator fusion, there would be very little benefit to generate SIMD instructions for the respective array operations used by each operator, as the memory loads and stores would have dominated the execution time. This is even more important as Numpy typically operates on double precision floats, which means only two (SSE) to four (AVX) scalars per vector registers.

Parallelization would also suffer from the lack of operator fusion: a synchronization fence is needed between each temporary computation. Hence the loop computation intensity would be very low compared to the memory pressure implied by two array reads and one array write for a single binary operator.

On the opposite, if all computations were merged into a single loop using temporaries elimination and operator fusion, parallelization would be more effective as barriers between binary operations are no longer needed and loads and stores are counterbalanced by several operations.

2. Optimization with the Pythran Compiler

2.1 Pythran

The optimizations presented in the previous section have been implemented in the Pythran compiler [1], a translator from a subset of Python to C++11 [2]. The input Optimization a Python module written in the subset accepted by the compiler. Pythran translates it in its internal representation, a simplified Python AST. It performs various optimizations then outputs either Python code, in a source-to-source fashion, or C++ templated code calling the pythonic library that typically implements the `ndarray` interface. User annotations can be used to instantiate this code for the proper types and generate a native library. This library relies on Boost.Python library to match Python's C API.

The Pythran compiler is an open source project publicly released under the BSD license[1].

Compared to existing alternatives, Pythran keeps the static compilation approach used by Cython, when JIT compilation is mainly used to statically type kernels before code generation in Numba and Parakeet. Unlike Cython, it maintains full Python backward compatibility. Following the Parakeet approach, it focuses on high-level constructs, while still generating efficient code for explicit loop and subscripts.

2.2 Experiments

The experiments are run on an Intel(R) Xeon(R) CPU E5-2650 0 @ 2.00GHz for a total of 8 cores (multithreading is not used). Each node has access to 64 KB of L1 cache, 256 KB of L2 cache. Both L1 and L2 caches are private, while L3 cache is shared between the 8 cores. This configuration provides a total of 64 GB of main memory. It supports up to AVX. The backend compiler is GCC 4.9 (20140528) with libgomp.

To compare the different optimization effects, we evaluated different optimization combination for Listing 1. The input is a raw array of 1,000,000 single precision floating point elements. Figure 1 illustrates the results. We can notice

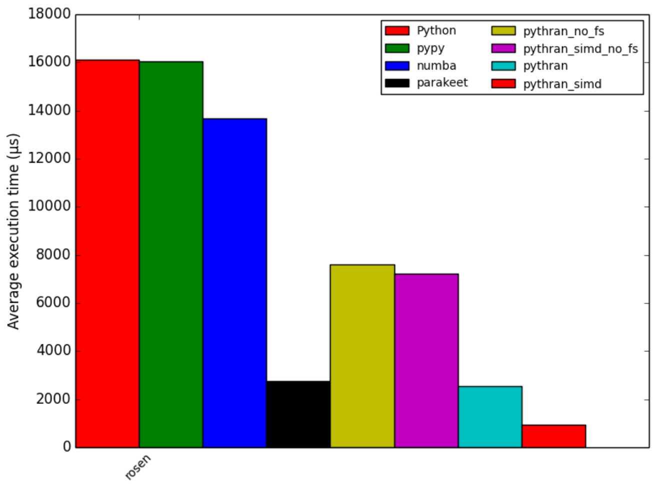

Figure 1: Execution time of the rosen kernel on Intel Xeon.

the importance of forward substitution on this benchmark as we avoid one extra loop, two temporary array assignments and some load/store for SIMD instructions. Vectorization is also extreemly profitable.

Conclusion

This extended abstract presents a short study of the optimization of Python/Numpy high level kernels in the context of high performance computing. It uses a real-worl synthetic kernel as leading example and focuses on the efficient implementation and optimization of array expressions within the ahead-of-time Pythran compiler, showing that a compilation step at Python level before generation of lower-level code makes it possible to generate vectorized, parallel C++ code. A comparison with existing JIT compilers for scientific Python validates the approach, showing significant performance improvements over the state of the art.

Acknowledgments

The Pythran project was partially funded by the CARP Project and the SILKAN Company.

References

[1] S. Guelton, P. Brunet, A. Raynaud, A. Merlini, and M. Amini. Pythran: Enabling static optimization of scientific python programs. In *Proceedings of the Python for Scientific Computing Conference (SciPy)*, June 2013.

[2] ISO/IEC 14882:2011. Programming languages – C++, 2011.

[3] T. E. Oliphant. Python for scientific computing. *Computing in Science and Engineering*, 9(3):10–20, May 2007.

[4] S. van der Walt, S. C. Colbert, and G. Varoquaux. The NumPy array: a structure for efficient numerical computation. *CoRR*, abs/1102.1523, 2011.

[5] G. van Rossum. A tour of the Python language. In *TOOLS (23)*, page 370, 1997.

[1] `http://pythonhosted.org/pythran/`