Exploring the Vectorization of Python Constructs Using Pythran and Boost SIMD

Serge Guelton
QuarksLab, Télécom Bretagne
sguelton@quarkslab.com

Joël Falcou
LRI, Université Paris-Sud
joel.falcou@lri.fr

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

Abstract

The Python language is highly dynamic, most notably due to late binding. As a consequence, programs using Python typically run an order of magnitude slower than their C counterpart. It is also a high level language whose semantic can be made more static without much change from a user point of view in the case of mathematical applications. In that case, the language provides several vectorization opportunities that are studied in this paper, and evaluated in the context of Pythran, an ahead-of-time compiler that turns Python module into C++ meta-programs.

Keywords Vectorization, Meta-Programming, Python, C++

1. Python, Unboxing and Pythran

The Python language [14] has grown in audience for the past ten years, even reaching the world of scientific computations [11] thanks to the `numpy` module [13], a module that provides an efficient multi-dimensional array type, and the `scipy` package [7] that provides a MATLAB-like API. As a consequence, more and more code is being written either in pure Python, generally to prototype an application, or as a Python and native code mix when performance matters. However, Python trades performance for dynamicity and does not particularly shines in terms of performance.

1.1 Python Performance Shortcomings

To illustrate the relatively bad performance of Python for numerical computations, let us consider the sixth problem from the Project Euler: "Find the difference between the sum of the squares of the first one hundred natural numbers and the square of the sum". A straight-forward solution can be coded as in Listing 1 in Python or as in Listing 2 in C. A comparison of the performance of the Python function and the C function called through the `ctypes` module shows that the C version runs more than $\times 250$ faster than the Python version. Enabling compiler auto-vectorization even makes the C version $\times 400$ faster than the Python version (Note that to perform more reliable time measurements, the size of the problem has been increased to 100001 instead of 101).

There are two main reasons for the poor performance of Python:

dynamic binding each variable access is made through a dictionary look-up that binds variable names to variable instances;

boxing each variable value is encapsulated into a heap-allocated generic object —the `PyObject`— which implies extra indirections for each operation.

In that context, it does not make sense to speak about vectorization: the nature of an operation is unknown to the interpreter until its execution, and data are not even contiguous in memory.

```
#pythran export solve(int)
def solve(n):
  r = range(1, n)
  a = sum(r)
  return a * a - sum(i * i for i in r)
```

Listing 1. Solution to the Project Euler Sixth Problem in Python

```
unsigned int solve(long n) {
  long a = 0, s = 0;
  for(long i = 0; i < n; ++i) {
    a += i;
    s += i * i;
  }
  return a * a - s;
}
```

Listing 2. Solution to the Project Euler Sixth Problem in C

1.2 Compiling Python Code

Several approaches have been proposed to solve the performance issue of Python: using native modules, relying on Ahead-Of-Time (AOT) compilation or Just-In-Time (JIT) compilation.

A native module is a shared library whose interface makes it callable from the Python interpreter using the regular `import` mechanism. It makes it possible to call native code from Python in a transparent way. Typically Python objects are unboxed when transfered from Python to C, where efficient computations can happen. Then the result is boxed when getting back to the Python world. In between, vectorization can happen. This is the approach taken by the `numpy` module: a native type, the `ndarray` contains contiguous unboxed elements, say single precision floating point numbers, and provides a great deal of high-level functions to manipulate them. All the computation-intensive part is done in pure C. Vectorization can eventually happen at this level, for instance when performing a point-to-point operation like the sum of two arrays. However the granularity of the operations limits the ratio of `load/store` versus operation. For instance, if `a`, `b` and `c` are 1-D arrays, the sum `a + b + c` is computed using two loops and an intermediate array. It means there are 4 `load/store` and only two additions per loop iteration and vectorization is unlikely to be profitable.

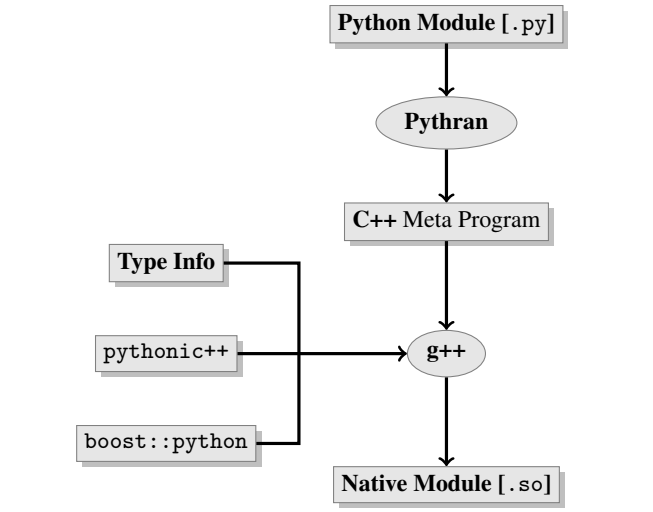

Figure 1. Pythran compiler work-flow.

AOT compilation uses type inference or type annotation to statically compute the type of all (like in Shedskin [4], Parakeet [12]) or part of (like in Cython [1] or Numba [10]) variables, effectively unboxing them to speed-up computations. Using JIT compilation, type information is available at runtime and the unboxing decision is based on profitability, e.g. according to a trace analysis. This is the approach of the PyPy project [3]. In both cases, generation of vector instruction is possible, but none of the cited project uses it. Indeed automatic vectorization is a complex topic, and as stated in [9], many cases are not yet correctly supported even for mainstream C/C++/FORTRAN compilers.

1.3 The Pythran Compiler

Pythran [5, 6] is also an AOT compiler, but it holds the unique property of turning Python modules into C++11 *meta-programs*. It does not need any type annotations during translation and keeps the polymorphism of original functions, respecting Python's duck typing. Its compilation work-flow is described in Figure 1: Python source is turned into generic C++11 code that is then instantiated for the given types provided as extra annotations. Like other AOT compilers, it unboxes all variables for better performance, but also performs a wide variety of optimizations such as lazy evaluation, constant folding, expression fusion etc. It supports core `numpy` constructs and most Python constructs, to the notable exception of user classes. Instructions such as `eval`, `exec` or `getattr` are not supported either.

This paper explores the vectorization of Python programs based on the idea that many Python constructs already exhibit good vectorization opportunities. Thus the goal of the compiler switches from automatic vectorization to vectorization of high-level Python intrinsics. One of the difficulties of vectorizing Python functions is linked to function polymorphism: vectorization in the context of C++ meta-programs is first examined in Section 2, and Python constructs that can benefit from vectorization are then described in Section 3. The approach is validated using the Pythran compiler on several kernels presented in Section 4.

2. Vectorization of a Meta-Program

Single Instruction Multiple Data (SIMD) extensions have been a feature of choice for processor manufacturers for a couple of decades. However, programming applications that take advantage of the SIMD extension available on the current target remains a complex task. Programmers that use low-level intrinsics have to deal with a verbose programming style due to the fact that SIMD instructions sets cover a few common functionalities, requiring to bury the initial algorithms in architecture specific implementation details. Furthermore, these efforts have to be repeated for every different extension that one may want to support, making design and maintenance of such applications very time consuming.

BOOST.SIMD is a high-level C++ library to program SIMD architectures. Designed as an *Embedded Domain Specific Language*, BOOST.SIMD provides both expressiveness and performance by using generic programming to handle vectorization. BOOST.SIMD does not only provide a portable way but also enables the use of common programming idioms when designing SIMD-aware algorithms. Two components are provided:

- **An abstraction of SIMD registers** for portable algorithm design, coupled with a large set of functions covering the classical set of operators along with a sensible amount (200+) of mathematical functions and utility functions,
- **A support for C++ standard components** like `Iterator` over SIMD `Range`, SIMD-aware allocators and SIMD-aware STL algorithms.

2.1 SIMD register abstraction

The first level of abstraction introduced by BOOST. SIMD is the `pack` class. For a given type `T` and a given static integral value `N` (`N` being a power of 2), a `pack` encapsulates the best type able to store a sequence of `N` elements of type `T`. For arbitrary `T` and `N`, this type is simply `std::array<T,N>` but when `T` and `N` matches the type and width of a SIMD register, the architecture-specific type used to represent this register is used instead. This semantic provides a way to use arbitrarily large SIMD registers on any system and let the library select the best vectorizable type to handle them. By default, if `N` is not provided, `pack` will automatically select a value that will trigger the selection of the native SIMD register type. Moreover, by carrying informations about its underlying scalar type, `pack` enables proper instruction selection even when used on extensions (like SSE2 and above) that map all integral type to a single SIMD type (`__m128i` for SSE2).

`pack` handles these low-level SIMD register type as regular objects with value semantics, which includes the ability to be constructed or copied from a single scalar value, list of scalar values, iterator or range. In each case, the proper register loading strategy (splat, set, load or gather) will be issued. `pack` also takes care of issues like boolean predicates support and provides both a range and a tuple-like interface.

2.2 Interaction with Standard Algorithm

Modern C++ programming style based on Generic Programming usually leads to an intensive use of various STL components like Iterators. BOOST.SIMD provides iterator adaptors that turn regular random access iterators into iterators suitable for SIMD processing. These adaptors act as free functions taking regular iterators as parameters and return iterators that output `pack` whenever dereferenced. These iterators are then usable directly in usual STL algorithms such as `transform` or `fold` (Listing 3).

```
std::vector<int,allocator<int>> v(N),r(N);

std::transform( simd::begin(v.begin())
              , simd::end(v.end())
              , simd::begin(r.begin())
              , [](pack<int> const& p)
                {
```

```
            return p*p;
        }
    );
```

Listing 3. SIMD Iterator with STL algorithm

Previous examples of integration within standard algorithm are still limited. Applying `transform` or `fold` algorithm on SIMD aware data requires the size of the data to be an exact multiple of the SIMD register width and, in the case of fold, to potentially perform additional operations at the end of its call. To alleviate this limitation, BOOST.SIMD provides its own overload for both `transform` and `fold` that take care of potential trailing data and performs proper completion of `fold`. Moreover, as BOOST.SIMD provides default implementation of all its function for scalar values, we can rely on polymorphic callable object to write a mixed scalar/SIMD call to algorithm like `transform`. By using the `boost::simd::transform` algorithm, code handling alignment, scalar prologue and epilogue is generated.

```
std::vector<int,allocator<int>> v(128), r(128);

struct f
{
  template<class T>
  auto operator()(T const& p)
  -> decltype(p * p)
  {
    return p * p;
  }
};

simd::transform( v.begin(), v.end()
               , r.begin()
               , f()
               );
```

Listing 4. Polymorphic SIMD algorithm call

Code written this way keeps a conventional structure and facilitate an usage of template functors for both scalar and SIMD cases also helps maximizing code reuse.

2.3 Interaction with Pythran

The first attempt to have Pythran generate vectorized code directly relied on SSE intrinsics. This approach quickly proved to be inappropriate for several reasons:

portability Only SSE was supported, moving to AVX or another vector instruction set would have required a great deal of engineering;

completness Many basic mathematic functions such as cos or sin were not directly available in the target instruction sets;

typing Pythran manipulates meta-functions with parametric types, while a given instruction set only provides concretes type.

The BOOST.SIMD abstraction layer solves these three issues to the expense of an increase in compilation time, which still remains acceptable for an AOT compiler.

Pythran uses function objects to represent any Python functions, either user-functions or functions from the standard library. The former are automatically generated from the user code analysis, and the latter are hand-coded in a library called `pythonic++` provided with the compiler. As an example, the Python function `def foo(x): return 3 * x` is compiled into a code similar to the one in Listing 5. The actual is slightly more complicated to handle type inference across several statements, but the structure remains valid.

Thanks to this representation, both the Python and C++ function can be called on scalar types, `list` type or any other type that supports multiplication by an integer constant. The only difference is that the Python version uses *dynamic polymorphism* while the C++ version uses *static polymorphism*.

```
struct foo {
    template<class T>
    auto operator()(T&& x)
    -> decltype(3 * x)
    {
        return x * 3;
    }
};
```

Listing 5. Simplified Pythran Translation of a Polymorphic Function

A critical point is that this representation perfectly matches the requirements of BOOST.SIMD, as presented in Listing 9. The `foo` function can be used indifferently on scalar types and vector types. This paves the way for automatic vectorization of high level idioms.

3. Using Python Semantic for Vectorization

Automatic vectorization generally relies on explicit loop-level vectorization, as extensively described in [2], or on block-level vectorization, also called Superword Level Parallelism (SLP) as introduced in [8]. Modern compilers for C/C++/FORTRAN generally implement a mixture of both.

However, a high-level language such as Python exhibits another level of vectorization opportunity: implicit loops. Implicit loops can be found in many places, but this paper focuses on *list comprehension* and *set comprehension*, the *map* and *sum* intrinsics, and *array operations* through the `numpy` module.

3.1 Intrinsics

3.1.1 The `sum` Intrinsic

The `sum` intrinsic is a typical case of potential vectorization of an implicit loop. Its prototype is: `sum(sequence[, start])` $\rightarrow$ `value` and its behavior is equivalent to the Python code in Listing 6. Depending on the type of `sequence`, and also whether one needs to strictly conforms to the IEEE Standard for Floating-Point Arithmetic (IEEE 754), vectorization of this reduction is a valid application of BOOST.SIMD's `fold`.

```
def sum(sequence, value=0):
  for e in sequence:
    value += e
  return value
```

Listing 6. Pseudo code of the `sum` intrinsic.

Apart from standard compliance, to be able to efficiently vectorize this code in C++, one must check that:

1. All element of `sequence` have the same types;
2. `sequence` contains scalar types;
3. `sequence` contains contiguous elements;
4. `sequence` provides a random access iterator. In particular one can computes its size in $\mathcal{O}(1)$.

Condition 1 is a prerequisite for static typing and efficient unboxing. For instance, any code that does not match it will fail to compile using Pythran or Shed-Skin. It is always satisfied for the

`ndarray` container as it wraps unboxed arrays. Condition 2 can be checked at compile time. Python supports two native scalar types candidates for vectorization: `int` (64 bits integer) and `float` (double-precision floating point numbers). The `numpy` module adds support for a wide range of signed and unsigned integers as well as single, double or quadruple precision float. Depending on the target architecture, only some of them can fit into vector registers. Finally condition 3 also is a static property of a container. It is verified for C++ alternative to Python's `list`, `str` and `ndarrays`, but generally not for `set` or `dict`. Condition 4 is matched by the C++ alternative of contiguous container but it is not matched by *generator*s, the Python incarnation of co-routines.

As function have to be generic but some containers do not match required properties, it is necessary to select a vectorized implementation of fall back to a generic one using Substitution Failure Is Not An Error (SFINAE) [15]. Vectorized versions are implement in this manner.

A possible C++ meta-implementation of `sum` in that context is presented in listing 7. It makes use of the parametric vector register introduced by Boost.SIMD and presented in Section 2. There is nothing new in the vectorization scheme itself. However, the possibility to write a meta-function that provides a vectorized implementation of the `sum` intrinsic makes it possible to efficiently turn a Python call into a C++ call.

```
template<class T, class V=long>
typename std::enable_if<
  is_vectorizable<T>::value,
  typename T::value_type
>::type
sum(T const& sequence, V value=0L)
{
  // E is the type of an element of T
  typedef typename T::value_type E;
  // vE is a vector type of vsize elements
  typedef typename simd::native<E,
    BOOST_SIMD_DEFAULT_EXTENSION> vE;
  static const size_t vsize =
    simd::meta::cardinal_of<vE>::value;

  // parallel reduction
  vE vector_value = simd::splat<vE>(E());
  const size_t n = sequence.size();
  const size_t bound = n / vsize * vsize;
  for(size_t i = 0; i < bound; i+= vsize)
    vp += sequence.load(i);
  // inner sum of the vector
  value += boost::simd::sum(vp);

  // handle the tail
  for(size_t i = bound; i < n; ++i)
    value += sequence.at(i);
  return value;
}
```

Listing 7. Meta-implementation of the `sum` intrinsic in C++11.

3.1.2 `map`

The `map` intrinsic simply creates a new `list` and fills it by repeatedly applying a function to the items of its argument sequence(s). Its prototype is: `map(function, sequence[, sequence, ...])` → `list`. Again, this is a typical case of vectorizable implicit loops, providing some conditions are met:

- `function` must be vectorizable;
- all `sequence`s verify the same constraints as stated for the `sum` intrinsic.

Deciding if a given function is vectorizable can be approximated through a composition rules. First, some functions are known to be vectorizable. This is generally the case for the addition, subtraction etc. Depending on the target vector instruction set, some operations may or may not be available as vector instructions, e.g. trigonometric functions, but they can be provided as composite instructions, as done by BOOST.SIMD. Some are just not available, as quadruple precision float addition. Starting from this set of vectorizable function, one can use the `;` as a closure binary relation to compute a subset of all vectorizable functions. Said otherwise, any sequence of operation that only involve vector operations is a vector operation and a function whose body is a sequence of vector operation or vectorizable functions is a vectorizable function itself.

The preceding statement is only valid if the body of the function only references vector functions and formal parameters, as the function must be valid for both scalar types and vector types. Note that this perfectly matches the translation of polymorphic Python functions into C++ meta-functions, as detailed in Listings 8 and 9. The core idea is that the `square` function can be called indifferently on floating point values or on vector of floating point values. Through (automatic) template instantiation, the C++ compiler generates the actual scalar and vector version from the same meta-function. As Pythran always translates Python's polymorphic function in that form, writing a vector version of the `map` intrinsic becomes possible. It is slightly more technical than for the `sum` intrinsic due to its variadic number of arguments, but it relies on the same concepts.

```
def square(x):
  return x * x
```

Listing 8. Python implementation of the square function.

```
struct square {
    template<class T>
    auto operator()(T const& x)
    -> decltype(x * x)
    {
      return x * x;
    }
};
```

Listing 9. C++ meta-implementation of the square function.

3.2 List Comprehension

Python provides *list comprehension* as a way to create `list` from any iterable. For instance `[abs(x) for x in l]` creates a new `list` that holds the absolute value of each element of `l`. A filter can be set up as in `[abs(x) for x in l if x % 3]`, where only the elements non multiple of 3 are taken into account. Let us call *comprehension rule* the expression on the left of the `for`.

List comprehension can be translated into a combination of `map` and (optionally) `filter`, by transformation of the comprehension rule into a lambda function. For instance the previous example can be re-written as `map(abs, filter(lambda x: x% 3, l))`. Once in this form, the same vectorization opportunities as in the previous section arise.

Python also provide same features for set and dictionary. Set comprehension can be rewritten in the form of generator expression using the eponym function: `{ x + 3 for x in l}` is equivalent to `set(x + 3 for x in l)`.

Dictionary comprehension, generator expressions and multi-level comprehension (i.e. containing multiple `for` clauses) are not considered in this paper.

3.3 The Numpy Module

The `numpy` module played a major role in the adoption of Python by the scientific community. In a nutshell, it is a thin Python wrapper above a raw array that provides an array language embedded into Python. An example of `numpy` code is show on Listing 10.

```
import numpy as np
def rosen(x):
  return np.sum(100*(x[1:] - x[:-1] ** 2) ** 2
                + (1 - x[:-1]) ** 2)
```

Listing 10. Sample `numpy` code.

Common features of array language include indexing, slicing, point-to-point operation etc. Point-to-point operations are traditionally a good source of vectorization. However, because of the mandatory module layer, `numpy` always create a new array to hold intermediate result. For instance in the expression `(1 - x[:-1]) ** 2)` two array are created —slicing does not create an array but a view. The implied extra cost would hide the benefits of vectorization.

The typical way to solve this issue in C++ is to use *expression templates*, a technique described in [16] that uses the C++ type system to perform lazy evaluation of an expression. Yet, this approach is limited to a single expression, as illustrated by Listing 11: in that case expression templates would not prevent the creation of temporary arrays `t0` and `t1`. Fortunately, it is possible to apply *forward substitution* to fall-back to the expression from Listing 10 and avoid the creation of any temporary based on static analysis of the source code.

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

Listing 11. Sample `numpy` code requiring forward subsitution.

It is possible to hook in the expression template code to add vectorization support. Indeed the expression from Listing 10 hides a single vector loop, that can be explicitly vectorized using BOOST.SIMDin a similar manner as the `map` and `sum` intrinsics.

One potential drawback of the approach would be the generation of multiple load from the same location, as in `1. / a + a`, where `a` is loaded twice on each loop iteration. Fortunately, the back-end C++ compiler is capable of detecting the redundant load and perform *redundant load store elimination*, effectively issuing a single load.

4. Validation of the Approach using the Pythran Compiler

To validate the approach, a number of code snippets presenting vectorization opportunities have been selected and benchmarked. The original Python version is used as a reference, then code is compiled with Pythran without vectorization support and finally with Pythran and vectorization support. Time measurements are made using the `timeit` module, which runs the piece of code a number of time and measure its average execution time. It does this three times and picks the best average execution time among those.

Experiments are run on a computer running a Linux kernel version 3.10. The processors are Intel Core i7, so the AVX instruction set is available. Python version is 2.7.5, Pythran version is the `HEAD` of the `wpmvp14` branch of the official Pythran repository `https://github.com/serge-sans-paille/pythran`. The back-end C++ compiler is Clang version 3.4-1. The compilation flags used are `-O2` without vectorisation support and `-O2 -march=native -DUSE_BOOST_SIMD` with vectorisation support. Using `-O3` or `-Ofast` actually degraded performances.

All the benchmarks use Numpy arrays of *double precision* random floats as input, which means a maximum theoritical speedup of $\times 4$ using AVX.

The scripts used to perform the measures as well as all other experimental data used in this paper are available online at `https://github.com/serge-sans-paille/pythran/tree/wpmvp14/doc/papers/wpmvp14/experiments`.

4.1 Vectorization of an Euler Problem

This experiment considers the code from Listing 1. This code uses *generator expression* that have not been explored in this paper, so it has been modified to use *list comprehension* instead. As a result, a lot of temporary array are used, which leads to the performance show on Figure 2. Although Pythran does a decent job at optimizing the code, vectorization does not provide significant speedup. This is due to the poor computation over memory transfer ratio, and the lack of optimization done by Pythran on that aspect for `lists`. A similar setup is evaluated in the context of `numpy` arrays in § 4.4 that use temporary array elimination.

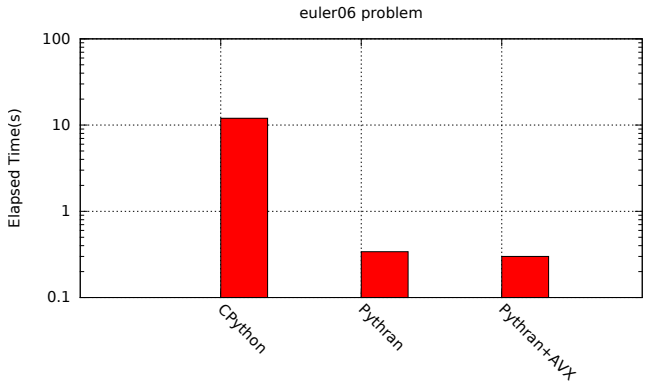

Figure 2. Timings of euler06 problem using CPython, Pythran, and Pythran with vectorization.

4.2 Vectorization of the `map` intrinsic

To evaluate the vectorization of the `map` intrinsic, two expressions are used: `map(abs, r)` and `map(lambda x: 5 * x + 3, r)`, where `r` is a `ndarray` of random double-precision floating point numbers. Figure 3 summarizes the result. The execution time for the CPython version is extremely long due to the `ndarray` indexing and function call overheads in CPython. Thanks to static typing and aggressive inlining, Pythran does not suffer from these issues, as shown by its sequential execution time. Vectorization is beneficial and yields a $\times 2.93$ speedup.

4.3 Vectorization of the `sum` intrinsic

To evaluate the vectorization of the `sum` intrinsic, two expressions are used: `sum(r)` and `sum(r * r)`, where `r` is a `ndarray` of random double-precision floating point numbers. Figure 4 summarizes the result. Again, CPython is extremely slow. It is interesting to note that the extra array multiplication does not impacts the execution time much, simply because it is an operation performed at

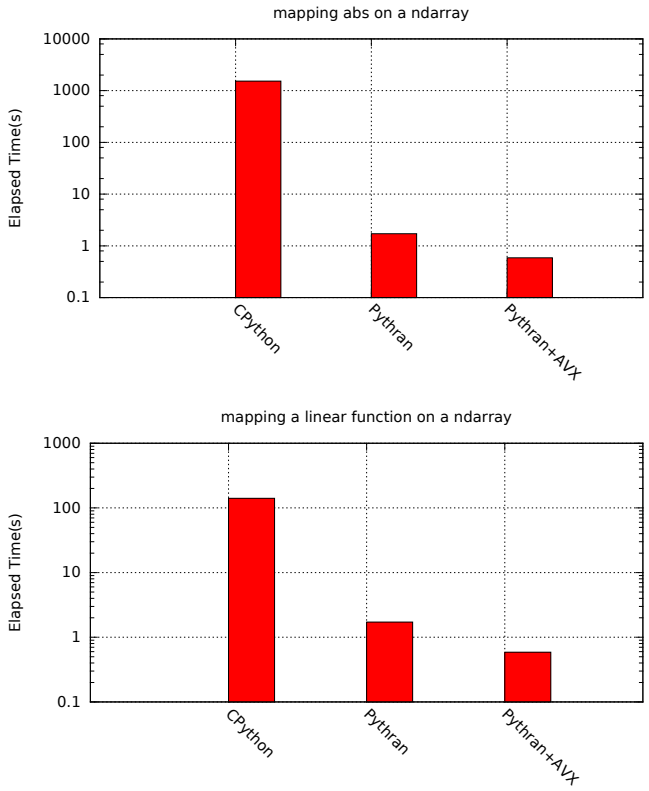

Figure 3. Timings of various `map` calls using CPython, Pythran, and Pythran with vectorization.

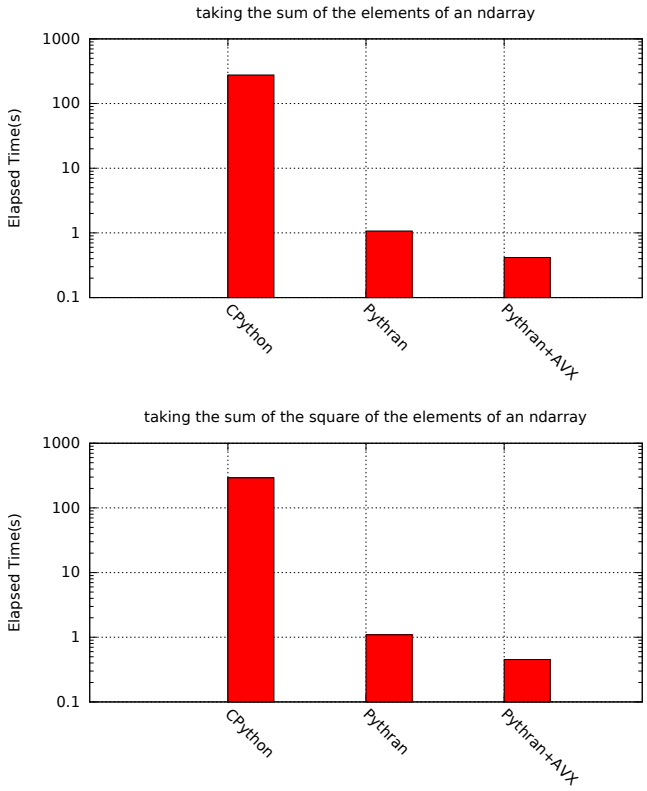

Figure 4. Timings of various `sum` calls using CPython, Pythran, and Pythran with vectorization.

the native level by the `numpy` module, while the `sum` is fully interpreted. The vectorized reductions yields a decent $\times 2.41$ speedup over the sequential version.

4.4 Vectorization of `ndarray` point-to-point operations

To evaluate the vectorization of `ndarray` point-to-point operations, three expressions are used: `1. / a + a`, the example from Listing 10 and a computation of the point-to-point arc distance between random values. Again, all inputs are large randomized `ndarray` of double precision floating point numbers. Figure 5 summarizes the results. Because all computations are done in native space, there is not much speedup when running them through Pythran. The only benefits comes from the elimination of temporary arrays and loop fusion. As a consequence, the more elements in the `numpy` expression, the more Pythran accelerates the code. The vectorization speedups follow the same process, as larger expression tend to increase the operation over memory movement ratio.

4.5 Vectorization of `ndarray` point-to-point operations with intermediate array

To have good performance even with intermediate array, Pythran uses forward substitution. This is not always beneficial if the substituted variable is used multiple time —it would lead to the computation of the expression several times— and it is only legal if no dependency are violated. This detection transforms three SIMD loop computations in one intensive computation in the case of Listing 10. Figure 6 shows the benefits of the approach.

Conclusion

This paper explores the design and implementation of a vectorizer for the Python language. It targets high-level constructs from the language and from the `numpy` module that already have a vector semantic, efficiently vectorizing them thanks to the combination of Pythran, a Python to C++ compiler that turns Python modules into C++ meta-programs and BOOST.SIMD, a C++ library to program SIMD architectures.

Common Python constructions such as *list comprehension*, the `map` and `sum` intrinsics and array operations from `numpy` are considered and the approach is validated on several kernels, showing promising results.

Future works include the case of Python generators, that have the benefit of not creating intermediate container to hold the data, but require more analyse in the context of vectorization. Handling `list` operations in an efficient manner as done for `numpy` arrays is also a required development axis.

Acknowledgments

Several students from Télécom Bretagne have been working on this project, the authors would like to thank Alan Raynaud, Adrien Merlini and Yuanchang Peng for their contribution to the Pythran project. They are also in debt to Mehdi Amini for his careful code reviews.

References

[1] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D. S. Seljebotn, and K. Smith. Cython: The best of both worlds. *Computing in Science Engineering*, 13(2):31–39, 2011. ISSN 1521-9615.

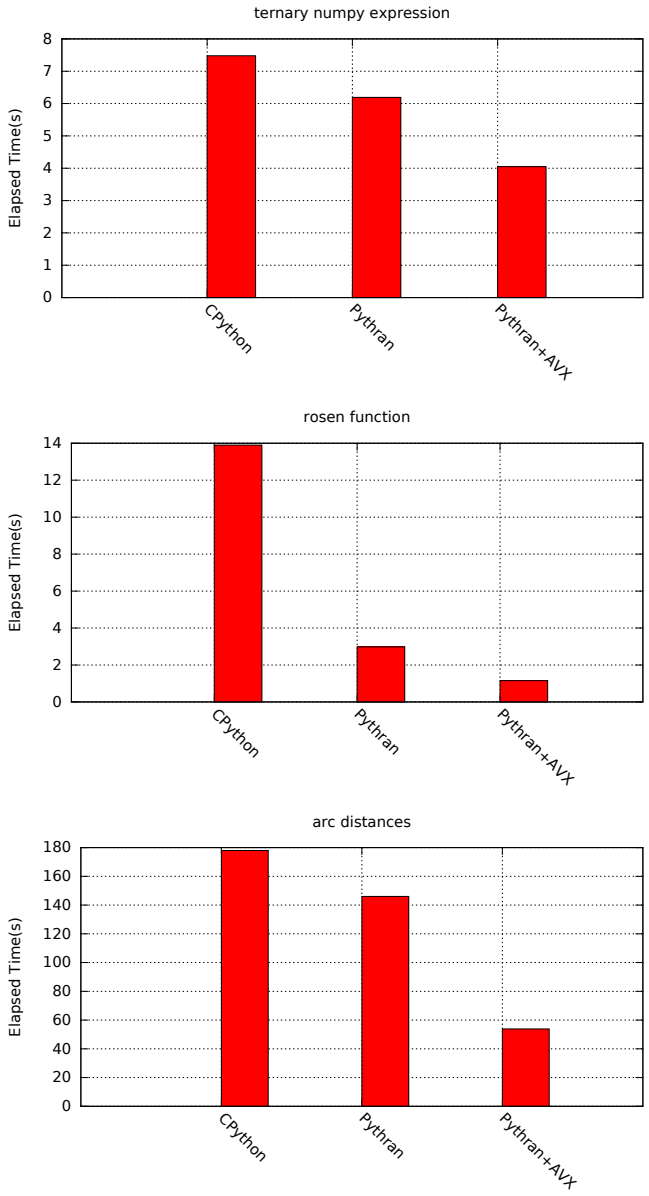

Figure 5. Timings of various `numpy` expressions using CPython, Pythran, and Pythran with vectorization.

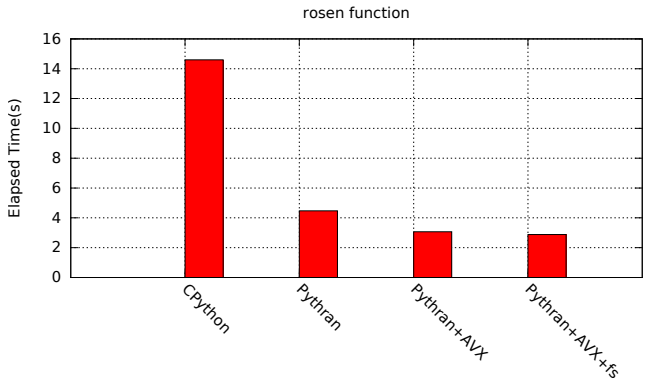

Figure 6. Timings of rosen computation with multiple assignment using CPython, Pythran, Pythran with vectorization, Pythran with vectorization and forward substitution.

June 2013.

[7] E. Jones, T. Oliphant, P. Peterson, et al. SciPy: Open source scientific tools for Python. http://www.scipy.org/, 2001.

[8] S. Larsen and S. P. Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In *Programming Language Design and Implementation*, PLDI, pages 145–156, 2000.

[9] S. Maleki, Y. Gao, M. J. Garzarán, T. Wong, and D. A. Padua. An evaluation of vectorizing compilers. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, PACT '11, pages 372–382, Washington, DC, USA, 2011. IEEE Computer Society.

[10] numba. numba. `http://numba.pydata.org`, 2013.

[11] T. E. Oliphant. Python for scientific computing. *Computing in Science and Engineering*, 9(3):10–20, May 2007. ISSN 1521-9615.

[12] A. Rubinsteyn, E. Hielscher, N. Weinman, and D. Shasha. Parakeet: a just-in-time parallel accelerator for python. In *Proceedings of the 4th USENIX conference on Hot Topics in Parallelism*, HotPar'12, pages 14–14, Berkeley, CA, USA, 2012. USENIX Association.

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

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

[15] D. Vandevoorde and N. M. Josuttis. *C++ Templates: The Complete Guide*. Addison-Wesley Professional, 1 edition, Nov. 2002.

[16] T. Veldhuizen. Expression templates. *C++ Report*, 7:26–31, 1995.

[2] A. J. C. Bik. *The Software Vectorization Handbook: Applying Intel Multimedia Extensions for Maximum Performance*. Intel Press, 2004.

[3] C. F. Bolz, A. Cuni, M. Fijalkowski, and A. Rigo. Tracing the meta-level: PyPy's tracing JIT compiler. In *Proceedings of the 4th ICOOOLPS workshop*, pages 18–25, New York, NY, USA, 2009. ACM.

[4] M. Dufour. Shed skin: An optimizing python-to-c++ compiler. Master's thesis, Delft University of Technology, 2006.

[5] S. guelton, P. Brunet, and M. Amini. Compiling python modules to native parallel modules using Pythran and OpenMP annotations. In *Proceedings of Workshop on Python for High Performance and Scientific Computing*. PyHPC 2013, Nov. 2013.

[6] 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)*,