Pythran: Enabling Static Optimization of Scientific Python Programs

Serge Guelton[§*], Pierrick Brunet[‡], Alan Raynaud[‡], Adrien Merlini[‡], Mehdi Amini[¶]

http://www.youtube.com/watch?v=KT5-uGEpnGw

Abstract—Pythran is a young open source static compiler that turns modules written in a subset of Python into native ones. Based on the fact that scientific modules do not rely much on the dynamic features of the language, it trades them in favor of powerful, eventually inter procedural, optimizations. These include detection of pure functions, temporary allocation removal, constant folding, Numpy *ufunc* fusion and parallelization, explicit thread-level parallelism through OpenMP annotations, false variable polymorphism pruning, and automatic vector instruction generation such as AVX or SSE.

In addition to these compilation steps, Pythran provides a C++ runtime library that leverages the C++ STL to provide generic containers, and the Numeric Template Toolbox (NT2) for Numpy support. It takes advantage of modern C++11 features such as variadic templates, type inference, move semantics and perfect forwarding, as well as classical ones such as expression templates.

The input code remains compatible with the Python interpreter, and output code is generally as efficient as the annotated Cython equivalent, if not more, without the backward compatibility loss of Cython. Numpy expressions run faster than when compiled with numexpr, without any change of the original code.

Index Terms—static compilation, numpy, c++

Introduction

The Python language is growing in popularity as a language for scientific computing, mainly thanks to a concise syntax, a high level standard library and several scientific packages.

However, the overhead of running a scientific application written in Python compared to the same algorithm written in a statically compiled language such as C is high, due to numerous dynamic lookup and interpretation cost inherent in high level languages. Additionally, the Python compiler performs no optimization on the bytecode, while scientific applications are first-class candidates for many of them.

Following the saying that scientific applications spend 90% of their time in 10% of the code, it is natural to focus on computation-intensive piece of code. So the aim may not be to optimize the full Python application, but rather a small subset of the application.

Several tools have been proposed by an active community to fill the performance gap met when running these computation-intensive piece of code, either through static compilation or Just In Time (JIT) compilation.

* *Corresponding author: serge.guelton@telecom-bretagne.eu*
§ *ENS, Paris, France*
‡ *Télécom Bretagne, Plouzané, France*
¶ *SILKAN, Los Altos, USA*

An approach used by Cython [cython] is to suppress the interpretation overhead by translating Python Programs to C programs calling the Python C API [pythoncapi]. More recently, Nuitka [nuitka] has taken the same approach using C++ has a back-end. Going a step further Cython also uses an hybrid C/Python language that can efficiently be translated to C code, relying on the Python C API for some parts and on plain C for others. ShedSkin [shedskin] translates implicitly strongly typed Python program into C++, without any call to the Python C API.

The alternate approach consists in writing a Just In Time(JIT) compiler, embedded into the interpreter, to dynamically turn the computation intensive parts into native code. The *numexpr* module [numexpr] does so for Numpy expressions by JIT-compiling them from a string representation to native code. Numba [numba] extends this approach to Numpy-centric applications while PyPy [pypy] applies it to the whole language.

To the notable exception of PyPy, these compilers do not apply any of the static optimization techniques that have been known for decades and successfully applied to statically compiled language such as C or C++. Translators to statically compiled languages do take advantage of them indirectly, but the quality of generated code may prevent advanced optimizations, such as vectorization, while they are available at higher level, i.e. at the Python level. Taking into account the specificities of the Python language can unlock many new transformations. For instance, PyPy automates the conversion of the *range* builtin into *xrange* through the use of a dedicated structure called *range-list*.

This article presents Pythran, an optimizing compiler for a subset of the Python language that turns implicitly statically typed modules into parametric C++ code. It supports many high-level constructs of the 2.7 version of the Python language such as list comprehension, set comprehension, dict comprehension, generator expression, lambda functions, nested functions or polymorphic functions. It does *not* support global variables, user classes or any dynamic feature such as introspection, polymorphic variables.

Unlike existing alternatives, Pythran does not solely perform static typing of Python programs. It also performs various compiler optimizations such as detection of pure functions, temporary allocation removal or constant folding. These transformations are backed up by code analysis such as aliasing, inter-procedural memory effect computations or use-def chains.

The article is structured as follows: Section 1 introduces the Pythran compiler compilation flow and internal representation. Section 2 presents several code analysis while Section 3 focuses on code optimizations. Section 4 presents back-end optimizations

for the Numpy expressions. Section 5 briefly introduces OpenMP-like annotations for explicit parallelization of Python programs and section 6 presents the performance obtained on a few synthetic benchmarks and concludes.

Pythran Compiler Infrastructure

Pythran is a compiler for a subset of the Python language. In this paper, the name *Pythran* will be used indifferently to refer to the language or the associated compiler. The input of the Pythran compiler is a Python module —not a Python program— meant to be turned into a native module. Typically, computation-intensive parts of the program are moved to a module fed to Pythran.

Pythran maintains backward compatibility with CPython. In addition to language restrictions detailed in the following, Pythran understands special comments such as:

```
#pythran export foo(int list, float)
```

as optional module signature. One does not need to list all the module functions in an *export* directive, only the functions meant to be used outside of the module. Polymorphic functions can be listed several times with different types.

The Pythran compiler is built as a traditional static compiler: a front-end turns Python code into an Internal Representation (IR), a middle-end performs various code optimizations on this IR, and a back-end turns the IR into native code. The front-end performs two steps:

1) turn Python code into Python Abstract Syntax Tree(AST) thanks to the *ast* module from the standard library;
2) turn the Python AST into a type-agnostic Pythran IR, which remains a subset of the Python AST.

Pythran IR is similar to Python AST, as defined in the *ast* module, except that several nodes are forbidden (most notably Pythran does not support user-defined classes, or the *exec* instruction), and some nodes are converted to others to form a simpler AST easier to deal with for further analyses and optimizations. The transformations applied by Pythran on Python AST are the following:

- list/set/dict comprehension are expanded into loops wrapped into a function call;
- tuple unpacking is expanded into several variable assignments;
- lambda functions are turned into named nested functions;
- the closure of nested functions is statically computed to turn the nested function into a global function taking the closure as parameter;
- implicit *return None* are made explicit;
- all imports are fully expanded to make function access paths explicit
- method calls are turned into function calls;
- implicit *__builtin__* function calls are made explicit;
- *try ... finally* constructs are turned into nested *try ... except* blocks;
- identifiers whose name may clash with C++ keywords are renamed.

The back-end works in three steps:

1) turning Pythran IR into parametric C++ code;
2) instantiating the C++ code for the desired types;
3) compiling the generated C++ code into native code.

The first step requires to map polymorphic variables and polymorphic functions from the Python world to C++. Pythran only supports polymorphic variables for functions, i.e. a variable can hold several function pointers during its life time, but it cannot be assigned to a string if it has already been assigned to an integer. As shown later, it is possible to detect several false variable polymorphism cases using use-def chains. Function polymorphism is achieved through template parameters: a template function can be applied to several types as long as an implicit structural typing is respected, which is very similar to Python's duck typing, except that it is checked at compile time, as illustrated by the following implementation of a generic dot product in Python:

```
def dot(l0, l1):
    return sum(x*y for x,y in zip(l0,l1))
```

and in C++:

```
template<class T0, class T1>
    auto dot(T0&& l0, T1&& l1)
    -> decltype(/* skipped */)
    {
        return pythonic::sum(
            pythonic::map(
                operator_::multiply(),
                    pythonic::zip(
                        std::forward<T0>(l0),
                        std::forward<T1>(l1))
            )
        );
    }
```

Although far more verbose than the Python version, the C++ version also uses a form of structural typing : the only assumption these two version make are that *l0* and *l1* are iterable, their content can be multiplied and the result of the multiplication is accumulatable.

The second step only consists in the instantiation of the top-level functions of the module, using user-provided signatures. Template instantiation then triggers the different correctly typed instantiations for all functions written in the module. Note that the user only needs to provide the type of the functions exported outside the module. The possible types of all internal functions are then inferred from the call sites.

The last step involves a template library, called *pythonic* that contains a polymorphic implementation of many functions from the Python standard library in the form of C++ template functions. Several optimizations, most notably expression template, are delegated to this library. Pythran relies on the C++11 [cxx11] language, as it makes heavy use of recent features such as move semantics, type inference through *decltype(...)* and variadic templates. As a consequence it requires a compatible C++ compiler for the native code generation. Boost.Python [boost_python] is involved for the Python-to-C++ glue. Generated C++ code is compatible with g++ 4.7.2 and clang++ 3.2.

It is important to note that all Pythran analyses are type-agnostic, i.e. they do not assume any type for the variables manipulated by the program. Type specialization is only done in the back-end, right before native code generation. Said otherwise, the Pythran compiler analyzes polymorphic functions and polymorphic variables.

Figure 1 summarizes the compilation flow and the involved tools.

Code Analyses

A code analysis is a function that takes a part of the IR (or the whole module's IR) as input and returns aggregated high-

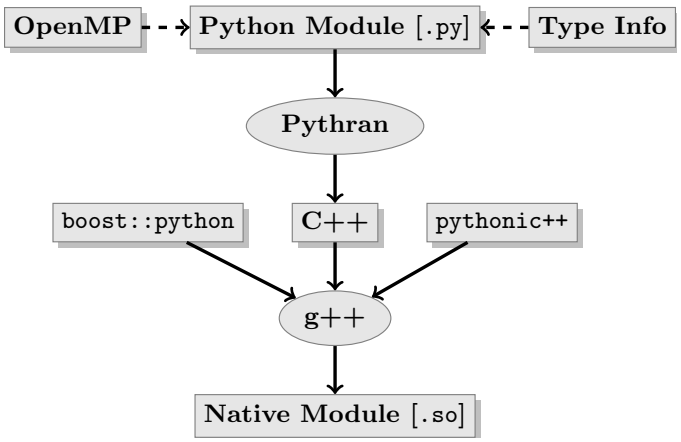

Fig. 1: *Pythran compilation flow.*

level information. For instance, a simple Pythran analysis called *Identifiers* gathers the set of all identifiers used throughout the program. This information is later used when the creation of new identifiers is required so that no conflict occurs with existing ones.

One of the most important analysis in Pythran is the *alias analysis*, sometimes referred as *points-to* analysis. For each identifiers, it computes an approximation of the set of locations this identifier may point to. For instance, let us consider the polymorphic function *foo* defined as follows:

```
def foo(a,b):
    c = a or b
    return c*2
```

The identifier *c* involved in the multiplication may refer to

- a fresh location if *a* and *b* are scalars
- the same location as *a* if *a* evaluates to *True*
- the same location as *b* otherwise.

As we do not specialise the analysis for different types and the true value of *a* is unknown at compilation time, the alias analysis yields the approximated result that *c* may point to a fresh location, *a* or *b*.

Without this kind of information, even a simple instruction like *sum(a)* would yield very few informations as there is no guarantee that the *sum* identifiers points to the *sum* built-in.

When turning Python AST to Pythran IR, nested functions are turned into global functions taking their closure as parameter. This closure is computed using the information provided by the *Globals* analysis that statically computes the state of the dictionary of globals, and *ImportedIds* that computes the set of identifiers used by an instruction but not declared in this instruction. For instance in the following snippet:

```
def outer(outer_argument):
    def inner(inner_argument):
        return cos(outer_argument) + inner_argument
    return inner
```

The *Globals* analysis called on the *inner* function definition marks *cos* as a global variable, and *ImportedIds* marks *outer_argument* and *cos* as imported identifiers.

A rather high-level analysis is the *PureFunctions* analysis, that computes the set of functions declared in the module that are pure, i.e. whose return value only depends from the value of their argument. This analysis depends on two other analyses, namely *GlobalEffects* that computes for each function whether this function modifies the global state (including I/O, random generators, etc.) and *ArgumentEffects* that computes for each argument of each function whether this argument may be updated in the function body. These three analyses work inter-procedurally, as illustrated by the following example:

```
def fibo(n):
    return n if n < 2 else fibo(n-1) + fibo(n-2)

def bar(l):
    return map(fibo, l)

def foo(l):
    return map(fibo, random.sample(l, 3))
```

The *fibo* function is pure as it has no global effects or argument effects and only calls itself. As a consequence the *bar* function is also pure as the *map* intrinsic is pure when its first argument is pure. However the *foo* function is not pure as it calls the *sample* function from the *random* module, which has a global effect (on the underlying random number generator internal state).

Several analyses depend on the *PureFunctions* analysis. *ParallelMaps* uses aliasing information to check if an identifier points to the *map* intrinsic, and checks if the first argument is a pure function using *PureFunctions*. In that case the *map* is added to the set of parallel maps, because it can be executed in any order. This is the case for the first *map* in the following snippet, but not for the second because the *print b* involves an *I/O*.

```
def pure(a):
    return a**2

def guilty(a):
    b = pure(a)
    print b
    return b

l = list(...)
map(pure, l)
map(guilty, l)
```

ConstantExpressions uses function purity to decide whether a given expression is constant, i.e. its value only depends on literals. For instance the expression *fibo(12)* is a constant expression because *fibo* is pure and its argument is a literal.

UseDefChains is a classical analysis from the static compilation world. For each variable defined in a function, it computes the chain of *use* and *def*. The result can be used to drive various code transformations, for instance to remove dead code, as a *def* followed by a *def* or nothing is useless. It is used in Pythran to avoid false polymorphism. An intuitive way to represent use-def chains is illustrated on next code snippet:

```
a = 1
if cond:
    a = a + 2
else:
    a = 3
print a
a = 4
```

In this example, there are two possible chains starting from the first assignment. Using *U* to denote *use* and *D* to denote *def*, one gets:

```
D U D U D
```

and:

```
D D U D
```

The fact that all chains finish by a *def* indicates that the last assignment can be removed (but not necessarily its right hand part that could have a side-effect).

All the above analyses are used by the Pythran developer to build code transformations that improve the execution time of the generated code.

Code Optimizations

One of the benefits of translating Python code to C++ code is that it removes most of the dynamic lookups. It also unveils all the optimizations available at C++ level. For instance, a function call is quite costly in Python, which advocates in favor of using inlining. This transformation comes at no cost when using C++ as the back-end language, as the C++ compiler does it.

However, there are some informations available at the Python level that cannot be recovered at the C++ level. For instance, Pythran uses functor with an internal state and a goto dispatch table to represent generators. Although effective, this approach is not very efficient, especially for trivial cases. Such trivial cases appear when a generator expression is converted, in the front-end, to a looping generator. To avoid this extra cost, Pythran turns generator expressions into call to *imap* and *ifilter* from the *itertools* module whenever possible, removing the unnecessary goto dispatching table. This kind of transformation cannot be made by the C++ compiler. For instance, the one-liner *len(set(vec[i]+i for i in cols))* extracted from the *nqueens* benchmarks from the Unladen Swallow project is rewritten as *len(set(itertools.imap(lambda i: vec[i]+i,cols)))*. This new form is less efficient in pure Python (it implies one extra function call per iteration), but can be compiled into C++ more efficiently than a general generator.

A similar optimization consists in turning *map*, *zip* or *filter* into their equivalent version from the *itertool* module. The benefit is double: first it removes a temporary allocation, second it gives an opportunity to the compiler to replace list accesses by scalar accesses. This transformation is not always valid, nor profitable. It is not valid if the content of the output list is written later on, and not profitable if the content of the output list is read several times, as each read implies the (re) computation, as illustrated in the following code:

```
def valid_conversion(n):
    # this map can be converted to imap
    l = map(math.cos, range(n))
    return sum(l) # sum iterates once on its input

def invalid_conversion(n):
    # this map cannot be converted to imap
    l = map(math.cos, range(n))
    l[0] = 1  # invalid assignment
    return sum(l) + max(l) # sum iterates once
```

The information concerning constant expressions is used to perform a classical transformation called *ConstantUnfolding*, which consists in the compile-time evaluation of constant expressions. The validity is guaranteed by the *ConstantExpressions* analysis, and the evaluation relies on Python ability to compile an AST into byte code and run it, benefiting from the fact that Pythran IR is a subset of Python AST. A typical illustration is the initialization of a cache at compile-time:

```
def esieve(n):
    candidates = range(2, n+1)
    return sorted(
        set(candidates) - set(p*i
                              for p in candidates
                              for i in range(p, n+1))
        )

cache = esieve(100)
```

Pythran automatically detects that *eseive* is a pure function and evaluates the *cache* variable value at compile time.

Sometimes, coders use the same variable in a function to represent value with different types, which leads to false polymorphism, as in:

```
a = cos(1)
a = str(a)
```

These instructions cannot be translated to C++ directly because *a* would have both *double* and *str* type. However, using *UsedDefChains* it is possible to assert the validity of the renaming of the instructions into:

```
a = cos(1)
a_ = str(a)
```

that does not have the same typing issue.

In addition to these python-level optimizations, the Pythran back end library, *pythonic*, uses several well known optimizations, especially for Numpy expressions.

Library Level Optimizations

Using the proper library, the C++ language provides an abstraction level close to what Python proposes. Pythran provides a wrapper library, *pythonic*, that leverage on the C++ Standard Template Library (STL), the GNU Multiple Precision Arithmetic Library (GMP) and the Numerical Template Toolbox (NT2) [nt2] to emulate Python standard library. The STL is used to provide a typed version of the standard containers (*list*, *set*, *dict* and *str*), as well as reference-based memory management through *shared_ptr*. Generic algorithms such as *accumulate* are used when possible. GMP is the natural pick to represent Python's *long* in C++. NT2 provides a generic vector library called *boost.simd* [boost_simd] that enables the vector instruction units of modern processors in a generic way. It is used to efficiently compile Numpy expressions.

Numpy expressions are the perfect candidates for library level optimizations. Pythran implements three optimizations on such expressions:

1) Expression templates [expression_templates] are used to avoid multiple iterations and the creation of intermediate arrays. Because they aggregates all *ufunc* into a single expression at compile time, they also increase the computation intensity of the loop body, which increases the impact of the two following optimizations.
2) Loop vectorization. All modern processors have vector instruction units capable of applying the same operation on a vector of data instead of a single data. For instance Intel Sandy Bridge can run 8 single-precision additions per instruction. One can directly use the vector instruction set assembly to use these vector units, or use C/C++ intrinsics. Pythran relies on *boost.simd* from NT2 that offers a generic vector implementation of all standard math functions to generate a vectorized version of Numpy expressions. Again, the aggregation of operators performed by the expression templates proves to be beneficial, as

it reduces the number of (costly) loads from the main memory to the vector unit.

3) Loop parallelization through OpenMP [openmp]. Numpy expression computation do not carry any loop-dependency. They are perfect candidates for loop parallelization, especially after the expression templates aggregation, as OpenMP generally performs better on loops with higher computation intensity that masks the scheduling overhead.

To illustrate the benefits of these three optimizations combined, let us consider the simple Numpy expression:

```
d = numpy.sqrt(b*b+c*c)
```

When benchmarked with the *timeit* module on an hyper-threaded quad-core i7, the pure Python execution yields:

```
>>> %timeit np.sqrt(b*b+c*c)
1000 loops, best of 3: 1.23 ms per loop
```

then after Pythran processing and using expression templates:

```
>>> %timeit my.pythranized(b,c)
1000 loops, best of 3: 621 us per loop
```

Expression templates replace 4 temporary array creations and 4 loops by a single allocation and a single loop.

Going a step further and vectorizing the generated loop yields an extra performance boost:

```
>>> %timeit my.pythranized(b,c)
1000 loops, best of 3: 418 us per loop
```

Although the AVX instruction set makes it possible to store 4 double precision floats, one does not get a 4x speed up because of the unaligned memory transfers to and from vector registers.

Finally, using both expression templates, vectorization and OpenMP:

```
>>> %timeit my.pythranized(b,c)
1000 loops, best of 3: 105 us per loop
```

The 4 hyper-threaded cores give an extra performance boost. Unfortunately, the load is not sufficient to get more than an average 4x speed up compared to the vectorized version. In the end, Pythran generates a native module that performs roughly 11 times faster than the original version.

As a reference, the *numexpr* module that performs JIT optimization of the expression yields the following timing:

```
>>> %timeit numexpr.evaluate("sqrt(b*b+c*c)")
1000 loops, best of 3: 395 us per loop
```

Next section performs an in-depth comparison of Pythran with three Python optimizers: PyPy, ShedSkin and numexpr.

Explicit Parallelization

Many scientific applications can benefit from the parallel execution of their kernels. As modern computers generally feature several processors and several cores per processor, it is critical for the scientific application developer to be able to take advantage of them.

As explained in the previous section, Pythran takes advantage of multiple cores when compiling Numpy expressions. However, when possible, it is often more profitable to parallelize the outermost loops rather than the inner loops —the Numpy expressions— because it avoids the synchronization barrier at the end of each parallel section, and generally offers more computation intensive computations.

Tool	CPython	Pythran	PyPy	ShedSkin
Timing	861ms	11.8ms	29.1ms	24.7ms
Speedup	x1	x72.9	x29.6	x34.8

TABLE 1: *Benchmarking result on the Pystone program.*

The OpenMP standard [openmp] is a widely used solution for Fortran, C and C++ to describe loop-based and task-based parallelism. It consists of a few directives attached to the code, that describe parallel loops and parallel code sections in a shared memory model.

Pythran makes this directives available at the Python level through string instructions. The semantic is roughly similar to the original semantics, assuming that all variables have function level scope.

The following listing gives a simple example of explicit loop-based parallelism. OpenMP 3.0 task-based parallelism form is also supported.

```
def pi_estimate(darts):
    hits = 0
    "omp parallel for private(x,y,dist), reduction(+:hits)"
    for i in xrange(darts):
        x,y = random(), random()
        dist = sqrt(pow(x, 2) + pow(y, 2))
        if dist <= 1.0:
            hits += 1.0
    pi = 4 * (hits / DARTS)
    return pi
```

The loop is flagged as parallel, performing a reduction using the *+* operator on the *hits* variable. Variable marked as *private* are local to a thread and not shared with other threads.

Benchmarks

All benchmarks presented in this section are ran on an hyper-threaded quad-core i7, using examples shipped along Pythran sources, available at https://github.com/serge-sans-paille/pythran in the *pythran/test/cases* directory. The Pythran version used is the *HEAD* of the *scipy2013* branch, ShedSkin 0.9.2, PyPy 2.0 compiled with the *-jit* flag, CPython 2.7.3, Cython 0.19.1 and Numexpr 2.0.1. All timings are made using the *timeit* module, taking the best of all runs. All C++ codes are compiled with g++ 4.7.3, using the tool default compiler option, generally *-O2* plus a few optimizing flags depending on the target.

Cython is not considered in most benchmarks, because to get an efficient binary, one needs to rewrite the original code, while all the considered tools are running the very same Python code that remains compatible with CPython. The experiment was only done to have a comparison with Numexpr.

Pystone is a Python translation of whetstone, a famous floating point number benchmarks that dates back to Algol60 and the 70's. Although non representative of real applications, it illustrates the general performance of floating point number manipulations. Table 1 illustrates the benchmark result for CPython, PyPy, ShedSkin and Pythran, using an input value of *10**3*. Note that the original version has been updated to replace the user class by a function call.

Tool	CPython	Pythran	PyPy	ShedSkin
Timing	1904.6ms	358.3ms	546.1ms	701.5ms
Speedup	x1	x5.31	x3.49	x2.71

TABLE 2: *Benchmarking result on the NQueen program.*

Tool	CPython	Pythran	PyPy	ShedSkin
Timing	1295.4ms	270.5ms	277.5ms	281.5ms
Speedup	x1	x4.79	x4.67	x4.60

TABLE 3: *Benchmarking result on the hyantes kernel, list version.*

It comes at no surprise that all tools get more than decent results on this benchmark. PyPy generates a code almost as efficient as ShedSkin. Altough both generate C++, Pythran outperforms ShedSkin thanks to a higher level generated code. For instance all arrays are represented in ShedSkin by pointers to arrays that likely disturbs the g++ optimizer, while Pythran uses a vector class wrapping shared pointers.

Nqueen is a benchmark extracted from the former Unladen Swallow* project. It is particularly interesting as it makes an intensive use of non-trivial generator expressions and integer sets. Table 2 illustrates the benchmark results for CPython, PyPy, ShedSkin and Pythran. The code had to be slightly updated to run with ShedSkin because type inference in ShedSkin does not support mixed scalar and *None* variables. The input value is *9*.

It seems that compilers have difficulties to take advantage of high level constructs such as generator expressions, as the overall speedup is not breathtaking. Pythran benefits from the conversion to *itertools.imap* here, while ShedSkin and PyPy rely on more costly constructs. A deeper look at the Pythran profiling trace shows that more than half of the execution time is spent allocating and deallocating a *set* used in the internal loop. There is a memory allocation invariant that could be taken advantage of there, but none of the compiler does.

Hyantes† is a geomatic application that exhibits typical usage of arrays using loops instead of generalized expressions. It is helpful to measure the performance of direct array indexing.

Table 3 illustrates the benchmark result for CPython, PyPy, ShedSkin and Pythran, when using lists as the data container. The output window used is *100x100*.

The speed ups are not amazing for a numerical application. there are two reasons for this poor speedups. First, the *hyantes* benchmark makes heavy usage of trigonometric functions, and there is not much gain there. Second, and most important, the benchmark produces a big 2D array stored as a list of list, so the application suffers from the heavy overhead of converting them from C++ to Python. Running the same benchmark using Numpy arrays as core containers confirms this assumption, as illustrated by Table 4. This table also demonstrates the benefits of manual parallelization using OpenMP.

Finally, *arc_distance*‡ presents a classical usage of Numpy expression. It is typically more efficient than its loop alternative as all the iterations are done directly in C. Its code is reproduced below:

```
def arc_distance(theta_1, phi_1, theta_2, phi_2):
    """
    Calculates the pairwise arc distance
    between all points in vector a and b.
    """
    temp = (np.sin((theta_2-theta_1)/2)**2
           + np.cos(theta_1)*np.cos(theta_2)
           * np.sin((phi_2-phi_1)/2)**2)
    distance_matrix = 2 * np.arctan2(
            sqrt(temp),sqrt(1-temp))
    return distance_matrix
```

Tool	CPython	Pythran	Pythran+OpenMP
Timing	450.0ms	4.8ms	2.3ms
Speedup	x1	x93.8	x195.7

TABLE 4: *Benchmarking result on the hyantes kernel, numpy version.*

Tool	CPython	Cython	Numexpr	Pythran
Timing	192.2ms	36.0ms	41.2ms	17.1ms
Speedup	x1	x5.33	x4.67	x11.23

TABLE 5: *Benchmarking result on the arc distance kernel.*

Figure 5 illustrates the benchmark result for CPython, Cython, Numexpr and Pythran, using random input arrays of *10**6* elements. Table 6 details the Pythran performance. Cython code is written using the *parallel.prange* feature and compiled with *-fopenmp -O2 -march=native*.

It shows a small benefit from using expression templates on their own, most certainly because the loop control overhead is negligible in front of the trigonometric functions. It gets a decent x2.5 speed-up when using AVX over not using it. The benefit of OpenMP, although related to the number of cores, makes a whole speedup greater than x11 over the original Numpy version, without changing the input code. Quite the opposite, Numexpr requires rewriting the input and does not achieve the same level of performance as Pythran when OpenMP and AVX are combined.

Writing efficient Cython code requires more work than just typing the variable declarations using Cython's specific syntax: it only takes advantage of parallelism because we made it explicit. Without explicit parallelization, the generated code runs around 176ms instead of 36ms. Cython does not generate vectorized code, and *gcc* does not vectorize the inner loop, which explains the better result obtained with Pythran.

Future Work

Although Pythran focuses on a subset of Python and its standard library, many optimizations opportunities are still possible. Using

*. http://code.google.com/p/unladen-swallow/

†. http://hyantes.gforge.inria.fr/

‡. The arc_distance test_bed is taken from to https://bitbucket.org/FedericoV/numpy-tip-complex-modeling

Pythran (raw)	Pythran (+AVX)	Pythran (+OMP)	Pythran (full)
186.3ms	75.4ms	41.1ms	17.1ms
x1.03	x2.54	x4.67	x11.23

TABLE 6: *Benchmarking result on the arc distance kernel, Pythran details.*

as Domain Specific Language(DSL) approach, one could use rewriting rules to optimize several Python idioms. For instance, *len(set(x))* could lead to an optimized *count_uniq* that would iterate only once on the input sequence.

There is naturally more work to be done at the Numpy level, for instance to support more functions from the original module. The extraction of Numpy expressions from *for loops* is also a natural optimization candidate, which shares similarities with code refactoring.

Numpy expressions also fit perfectly well in the polyhedral model. Exploring the coupling of polyhedral tools with the code generated from Pythran offers enthusiastic perspectives.

Conclusion

This paper presents the Pythran compiler, a translator, and an optimizer, that converts Python to C++. Unlike existing static compilers for Python, Pythran leverages several function-level or module-level analyses to provide several generic or Python-centric code optimizations. Additionally, it uses a C++ library that makes heavy usage of template programming to provide an efficient API similar to a subset of Python standard library. This library takes advantage of modern hardware capabilities —vector instruction units and multi-cores— in its implementation of parts of the *numpy* package.

This paper gives an overview of the compilation flow, the analyses involved and the optimizations used. It also compares the performance of compiled Pythran modules against CPython and other optimizers: ShedSkin, PyPy and numexpr.

To conclude, limiting Python to a statically typed subset does not hinders the expressivity when it comes to scientific or mathematic computations, but makes it possible to use a wide variety of classical optimizations to help Python match the performance of statically compiled language. Moreover, one can use high level information to generate efficient code that would be difficult to write for the average programmer.

Acknowledgments

This project has been partially funded by the CARP Project[§] and the SILKAN Company[¶].

References

[boost_python] D. Abrahams and R. W. Grosse-Kunstleve. *Building Hybrid Systems with Boost.Python*, C/C++ Users Journal, 21(7), July 2003.

[boost_simd] P. Estérie, M. Gaunard, J. Falcou, J. T. Lapresté, B. Rozoy. *Boost.SIMD: generic programming for portable SIMDization*, Proceedings of the 21st international conference on Parallel architectures and compilation techniques, 431-432, 2012.

[cython] 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, March 2011.

[cxx11] ISO, Geneva, Switzerland. *Programming Languages -- C++*, ISO/IEC 14882:2011.

[expression_templates] T. Veldhuizen. *Expression Templates*, C++ Report, 7:26-31, 1995.

[nt2] J. Falcou, J. Sérot, L. Pech, J. T. Lapresté *Metaprogramming applied to automatic SMP parallelization of linear algebra code*, Euro-Par, 729-738, January 2008, https://github.com/MetaScale/nt2.

[nuitka] K. Hayen. *Nuitka - The Python Compiler*, Talk at EuroPython2012.

[numba] T. Oliphant et al. *Numba*, http://numba.pydata.org/.

[numexpr] D. Cooke, T. Hochberg et al. *Numexpr - Fast numerical array expression evaluator for Python and NumPy*, http://code.google.com/p/numexpr/.

[openmp] *OpenMP Application Program Interface*, http://www.openmp.org/mp-documents/OpenMP3.1.pdf, July 2011.

[pypy] C. F. Bolz, A. Cuni, M. Fijalkowski and A. Rigo. *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, 2009.

[pythoncapi] G. v. Rossum and F. L. Jr. Drake. *Python/C API Reference Manual*, September 20012.

[shedskin] M. Dufour. *Shed skin: An optimizing python-to-c++ compiler*, Delft University of Technology, 2006.

§. http://carp.doc.ic.ac.uk/external/

¶. http://www.silkan.com/