REPORT ON OpenDreamKit DELIVERABLE D5.2

Facility to compile PYTHRAN compliant user kernels and SAGE code and automatically take advantage of multi-cores and SIMD instruction units in CYTHON

ADRIEN GUINET ET CLÉMENT PERNET

Due on	02/28/2017 (Month 18)
Delivered on	02/27/2017
Lead	Université Grenoble Alpes (UGA)
Progress on and finalization of this deliverable has been tracked publicly at: `https://github.com/OpenDreamKit/OpenDreamKit/issues/115`	

DELIVERABLE DESCRIPTION, AS TAKEN FROM GITHUB ISSUE #115 ON 2017-02-27

- **WP5:** High Performance Mathematical Computing
- **Lead Institution:** Université Joseph Fourier (UJF)
- **Due:** 2017-02-28 (month 18)
- **Nature:** Demonstrator
- **Task:** T5.7 (#105) Pythran
- **Proposal:** p.51
- **Final report**

The `Python` programming language is widely used in the development of computational mathematics systems like `SageMath` for its expressiveness and flexibility. Yet, as an interpreted language, it suffers from inherent inefficiencies.

Over the years several tools have been developed to overcome this barrier. A major player is `Cython`, which is both an extension of the Python language, and a compiler generating compilable `C` code. At the cost of additional work from the developer (e.g. type annotations), `Cython` can deliver performances similar to that of a compiled language. It's being used intensively in `SageMath`. Another emerging player is `Pythran`, a `Python` to `C++` compiler for a subset of the `Python` language, with a focus on scientific/numerical computing. It takes advantage of type inference features of `C++` as well as multi-cores and SIMD instruction units to deliver high performance without the need for additional work from the developer. In particular, it includes a `C++` implementation of a major subset of the `Numpy` API, optimized for speed, with support for expression templates that minimize the number of memory transfers needed to compute complex expressions (`Numpy` is the fundamental package for scientific computing with `Python`). However, unlike `Cython`, `Pythran` does not support user defined classes, a key feature in systems like `SageMath`.

This deliverable is a step toward taking the best of both worlds, and helping bridge the gap between numerical and exact computing. It proposes to incorporate `Pythran` support for `Numpy` within `Cython`, which consequently provides high performance numerical linear algebra to high level mathematical software developers, especially within `SageMath`.

As an illustration, the new `Pythran` backend in `Cython` achieves a speed-up of about 4 on the following typical `Numpy` based function:

```
def sqrt_sum (numpy.ndarray[FLOATTYPE_t, ndim=1] a,
              numpy.ndarray[FLOATTYPE_t, ndim=1] b):
    return numpy.sqrt(numpy.sqrt(a*a+b*b))
```

CONTENTS

1. INTRODUCTION

The aim of this project is to optimize the overall performance of CYTHON code that uses NumPy arrays.

CYTHON is a compiler for the PYTHON and CYTHON language. It basically converts PYTHON code to a C/C++ module that calls directly the C PYTHON API. The CYTHON language is an extension of PYTHON that allows to write C-like code within a PYTHON module. This code is directly inserted into the generated module. This allows to write C python module with the flexibility of a PYTHON script. This is used in numerous PYTHON project, with SageMath being one of them.

In CYTHON, when operations are done on NumPy arrays, CYTHON relies on the original NumPy package to compute them. This involves a fall back to the PYTHON interpreter. It thus misses several optimisation opportunities, espacially with complex expressions. Optimizing these NumPy calls can improve the overall performances of applications that rely on NumPy to do their computation work. The interest for the OpenDreamKit project is the overall performance gain that could be achieve within various SageMath workload.

The PYTHRAN project is a PYTHON to C++ compiler, that aims at optimizing scientific PYTHON. It thus supports only a subset of the PYTHON language. It includes a full C++ implementation of a major set of the NumPy API. Some of the advantage of this implementation is that it supports expression templates and SIMD instructions. Expression templates allow to "fuse" loops that can occurs when expressions with multiple operators are computed. For instance, the expression $a + b * c$ will originally be transformed by CYTHON in two calls: one for the multiplication of b by c, and one for the addition of the result of this multiplication and the addition by a. Each call will end-up in one loop, that will read memory, compute the operation and write back to memory. The second loop will have the same pattern. In nowadays architecture, memory bandwidth is often the limiting factor in this kind of operation. It is thus really interesting to merge these loops, and load/store the memory only once. Expression templating is a C++ technique that allows to evaluate expressions only when they are stored to memory. Thus, in this case, the two loops will be automatically "merged" by the C++ compiler, and we'll get an optimized version of this code. Note that this technique is used for instance by the C++ wrapper of the GMP library.

2. CYTHON AND PYTHRAN INTEGRATION

The project has been focused on using this PYTHRAN backend for NumPy arrays in CYTHON when possible. Indeed, PYTHRAN has a few limitations regarding the NumPy arrays it can handle:

- array "views" are not supported. That means that arrays must be stored in contiguous memory. Fortran and C-style formats are supported.
- the endianess of the integers must be the same as the one of the targeted architecture (note that CYTHON has the same limitation)

We thus need to be able to fall back to the CYTHON implementation if we are not in one of those cases.

The integration within CYTHON works this way:

- at the function level, for every argument that is a NumPy array and supported by PYTHRAN, we change its type by a fused type [1]. This fused type is either a PYTHRAN NumPy buffer or the original CYTHON buffer type
- for variables defined as NumPy array, we change them directly to a PYTHRAN buffer, if their type and endianess are supported.

[1] `http://cython.readthedocs.io/en/latest/src/userguide/fusedtypes.html`

CYTHON has a comprehensive suite test regarding the NumPy features it supports. This test suite is still valid after this integration.

3. USAGE

A new flag `--np-pythran` has been added to CYTHON that enables the usage of PYTHRAN for NumPy operations. It will generate a C++ file that uses the optimized NumPy functions that are in PYTHRAN. Then, when compiling the final extension, a path to an existing PYTHRAN installation must be provided.

Here is an example of usage using `distutils`:

```
from distutils.core import setup
from Cython.Build import cythonize

setup(
    name = "My hello app",
    ext_modules = cythonize('hello_pythran.pyx', np_pythran=True)
)
```

Basically, a flag `np_pythran` has been added to the `cythonize` call that will enable the usage of the PYTHRAN backend. It needs to have the PYTHRAN module installed.

4. BENCHMARKS

We did some benchmarks to see the benefits of the PYTHRAN integration for NumPy operations. These benchmarks have been done using an Intel Core i7-6700HQ.

4.1. Float computation

Here is a code snippet that does some computation on floating-point values:

```
import numpy
cimport numpy
def float_comp(numpy.ndarray[numpy.float_t, ndim=1] a, \
               numpy.ndarray[numpy.float_t, ndim=1] b):
    return numpy.sqrt(numpy.sqrt(a*a+b*b))
```

The figure 1 shows the compute time for various sizes for `a` and `b`, using the original CYTHON mode, then CYTHON with the PYTHRAN backend, and finally CYTHON with the PYTHRAN backend and SIMD instructions.

For instance, for $N = 1000000$, we have a speedup of 2.4x using the PYTHRAN backend (against the original CYTHON version), and 3.7x using SIMD instructions (still against the original CYTHON version).

4.2. Harris

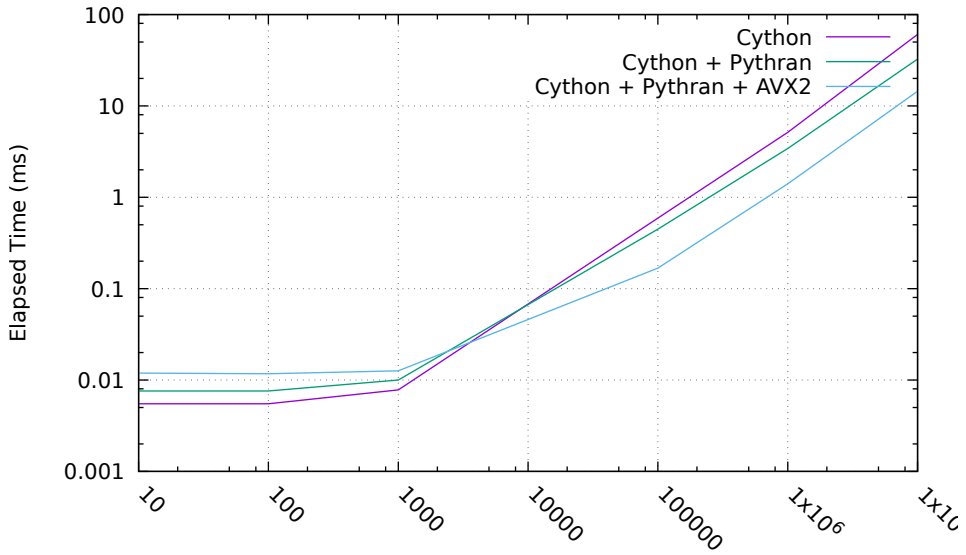
This benchmark is based on the code available here: `https://raw.githubusercontent.com/serge-sans-paille/numpy-benchmarks/master/benchmarks/harris.py`.

The adapted CYTHON code is the following:

```
import numpy
cimport numpy

def harris(numpy.ndarray[numpy.float_t, ndim=2] I):
  cdef int m = I.shape[0]
  cdef int n = I.shape[1]
```

FIGURE 1. Float computation benchmark (logarithmic scales)




```
cdef numpy.ndarray[numpy.float_t, ndim=2] dx = \
  (I[1:, :] - I[:m-1, :])[:, 1:]
cdef numpy.ndarray[numpy.float_t, ndim=2] dy = \
  (I[:, 1:] - I[:, :n-1])[1:, :]

cdef numpy.ndarray[numpy.float_t, ndim=2] A = dx * dx
cdef numpy.ndarray[numpy.float_t, ndim=2] B = dy * dy
cdef numpy.ndarray[numpy.float_t, ndim=2] C = dx * dy
cdef numpy.ndarray[numpy.float_t, ndim=2] tr = A + B
cdef numpy.ndarray[numpy.float_t, ndim=2] det = A * B - C * C
return det - tr * tr
```

The figure 2 shows the compute time for various sizes of the 2D array I ($N \times N$). We've got a 25 % improvement with an array of 4096×4096 floats as input using the PYTHRAN backend. Moreover, in this case, it seems that this code does not really benefits from SIMD instructions.

4.3. **Convolution**

We used this example from the CYTHON documentation: `http://cython.readthedocs.io/en/latest/src/tutorial/numpy.html`.

The input we use is the following:

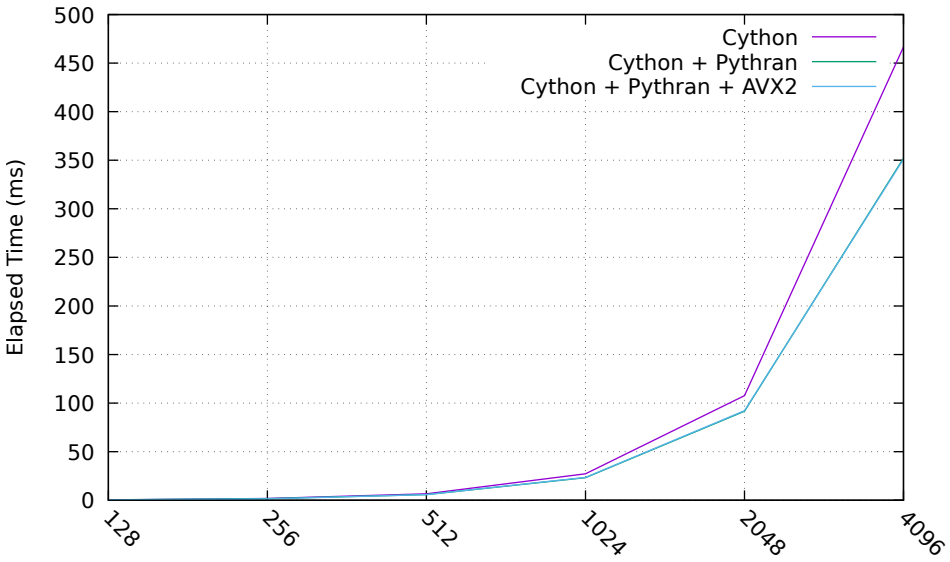
```
import numpy as np
N = 1000
f = np.arange(N*N, dtype=np.int).reshape((N,N))
g = np.arange(81, dtype=np.int).reshape((9, 9))
```

The results are the following:

- for the classical CYTHON version: 155ms
- for the CYTHON version using the PYTHRAN backend: 150ms
- for the CYTHON version using the PYTHRAN backend using SIMD instructions: 149ms

It seems the PYTHRAN backend don't manage to benefit a lot from SIMD instructions in this case. We thus still got an average speedup of 3 %.

FIGURE 2. Harris computation benchmark



5. LINKS

Modifications that were necessary to the PYTHRAN project have been accepted and merged into its master branch (see `https://github.com/serge-sans-paille/pythran/pull/629`, `https://github.com/serge-sans-paille/pythran/pull/628`, `https://github.com/serge-sans-paille/pythran/pull/616` and `https://github.com/serge-sans-paille/pythran/pull/614`).

The modifications necessary within the CYTHON project are currently being reviewed (see `https://github.com/cython/cython/pull/1607`).

Disclaimer: this report, together with its annexes and the reports for the earlier deliverables, is self contained for auditing and reviewing purposes. Hyperlinks to external resources are meant as a convenience for casual readers wishing to follow our progress; such links have been checked for correctness at the time of submission of the deliverable, but there is no guarantee implied that they will remain valid.