Experimenting OpenMP/Python Coupling

Serge Guelton
École Normale Supérieure, Paris, France
serge.guelton@telecom-bretagne.eu

Pierrick Brunet
Télécom Bretagne, Plouzané,France
pierrick.brunet@telecom-bretagne.eu

Abstract

The Python language offers an interesting solution as a scientific computing language thanks to the SciPy package. However, the lack of fine-grained parallelism support remains an important shortcoming. This paper proposes to add OpenMP directives to the Python language and shows that it is possible to turn a module written in a large Python subset enhanced with these directives into a native parallel module that runs as fast as its naive C equivalent, while keeping the input code totally compatible with the Python interpreter. The approach is validated through the Pythran compiler.

Keywords Python, OpenMP, C++, static compilation, scripting language.

1. Introduction

Since its birth in December 1989, the Python language [19] has proved to be useful in various domains, ranging from system administration to web services, thanks to its dynamicity, expressiveness, rich ecosystem and battery-included standard library. It is also getting widely used in scientific computing [14], mainly thanks to the SciPy [10] project that adds MATLAB-like functionalities to the language.

The prohibitive performance overhead of the language implied by its interpreted and highly dynamic nature does not prevent its usage as a language for high performance computing. SciPy overcomes this issue through low-level routines written in C or Fortran and encapsulated in Python *native* modules. Moreover, its core data structure, the multi-dimensional array [18], has been designed so that the underlying data are available to both native modules and the Python interpreter without conversion cost.

To enable the user to write native functions, without having to write the glue code that turns Python object into native structures back and forth, the SciPy package provides the `weave` module that makes it possible to bundle C code snippet into Python code and compile then load them at runtime.

This hybrid approach, based on a mix of interpreted and native code, is getting widespread in the Python landscape. Section 2 studies the existing approaches and shows a critical lack of parallelism support, and emphasises on the need for a backward-compatible approach. Section 3 studies the validity of using OpenMP directives within the Python language in order to add fine-grain parallelism support to Python in the context of a Python subset called Pythran [9]. Static compilation of this language into C++ and the underlying runtime is described in Section 4. The Pythran compiler is benchmarked on several scientific kernels and compared to Cython in Section 5.

2. Parallel Computations in Python

In the many cores area, turning to parallelism to balance the performance limitations of scripting languages, as described in [7] in the context of the MATLAB language, or in [13] in the context of the R language, seems legitimate. In the context of Python, most approaches have focused on fork-based parallelism.

2.1 Python and Parallelism

Parallel computations are supported by the Python standard library through the `multiprocessing` module. It spawns several interpreters that can communicate through IPC, using Python built-in object serialization. This approach is only viable for computation intensive application, since the communication and synchronization overheads are much greater than a light-weight processes approach.

Although the standard `threading` module makes it possible to start several light-weight threads within the same interpreter, this approach is not applicable to HPC, because of a specificity of CPython, the *Global Interpreter Lock* [20].[1] This lock ensures that only one thread is active at a time in the interpreter. While it makes it possible to have cooperative threads, say for a GUI, it does not take advantage of multiple cores. However, there are two notable exceptions: the GIL is released on I/O, and the GIL does not prevent the use of threads inside native modules, where the user has full control.

To illustrate this behavior, we used the CPU-bound Buffon's needle algorithm to estimate the value of π. Its execution time using a multi-threaded implementation on 4 cores is $\times 1.46$ the execution time of the sequential version. GIL contention actually *increases* the execution time.

There have also been several approaches to replace the GIL by Transactional Memories [16, 17] but none of them made there way to the mainstream interpreters.

As a consequence, Python developers need to write multi-threaded native modules in order to fully benefit from multiple cores. This leads to a kind of computations referred as hybrid computations.

2.2 Hybrid Computations

In the context of interpreted languages, [12] defines a computation as *hybrid* when part of the code is interpreted, and part of it is executed natively.

[1] Other Python interpreters, such as IronPython or Jython, do not have a GIL.

It is now common for scripting language to have C bindings. To take advantage of compiled code, and to overcome the GIL limitations, Python developers must write parallel C/C++ functions and the associated boilerplate based on the Python C API [21]. Tools have been developed to relieve the user from writing the encapsulating code, notably SWIG [2] that relies on an enhanced interface specification, or `boost::python` [1] that relies on C++ facilities to guide translation.

An opposite approach consists in using the host language — herein Python— to describe both parts of the system, and to let an automated tool perform the translation to native code of a part of the application, generally the computation-intensive one where parallelism has been expressed in some ways. Thus, developers not familiar with lower level languages or not eager to invest the additional development time can still benefit from a fair performance boost. This approach is the subject of many studies that can be classified according to their compatibility with the host language.

2.2.1 Backward-Incompatible Approaches

A constraining (from the performance point of view) aspect of the Python language is its type system. It implies that each method call is resolved dynamically, even a simple add operation! It comes at no surprise that many approaches restrain the Python language to add a static typing overlay. Also, only two types of integers (64-bits integers and multi-precision integers) and one type of floating point type (double-precision floats) are available in Python, so using a type with the appropriate size may lead to significant performance boost.

Cython [3] is such a Python dialect. It extends the syntax with typing informations, calls to native functions from third party libraries, and a limited set of parallelism constructs, such as the possibility to define parallel loops, but no task parallelism. PLW [12] proposes a similar approach that mixes Python with C, using raw strings to hold the C code and the parallel directives, that are limited to parallel for loops. The numba [2] compiler uses additional type information to generate sequential LLVM bytecode. PyCUDA and PyOpenCL [11] also target accelerators by mixing Python with kernels embedded as raw strings containing accelerator code.

The main issue of these approaches is that they imply to modify, in a backward-incompatible way, the original code. They require to learn a new dialect, and the long-term preservation of this investment is not ensured. Moreover, there is no falling back if the code were to be deployed in an environment where the parallelizing tool/module is not available. For instance, once a code has been ported to PyCUDA and transformed into CUDA code embedded in Python strings, there is no way back and the developer has to maintain two versions of the algorithm.

2.2.2 Backward-Compatible Approaches

Most backward-compatible approaches also require to modify the input program. They do not extend the Python language, but restrict it. As a consequence, they remain compatible with the original language and do not suffer from the drawbacks of the previous approaches. They also benefit from existing tools associated to the language.

Copperhead [6] is a functional, data parallel language embedded in Python. It uses n-uplets, NumPy arrays and lists as its core data structure and prohibits usage of many control-flow operators such as loops, enforcing the use of the `map`, `filter` or `reduce` intrinsics to exhibit parallelism. But it can be efficiently compiled to either CUDA or C++ with calls to the Thrust[3] library. Python decorators are used to identify hot-spots that are JIT-compiled to native code.

[2] *cf.* `https://github.com/numba/numba`

[3] *cf.* `http://thrust.github.com/`

Tools such as PyPy [5], a Python interpreter with a tracing JIT, or Shed Skin [8], a Python to C++ compiler are also viable ways to turn regular Python codes into native ones, but they do not offer support for fine-grained parallelism beyond what the standard library proposes.

To be completely backward compatible, it must be possible to run the input code in an environment that is not aware of the existence of the parallelizing solution. This principle is not respected by parallel libraries, but code annotations partially satisfy it. This is where OpenMP [15] parallel annotations shine: the original code remains mostly compatible with a compiler that is not aware of OpenMP.

This article proposes to combine OpenMP [15] parallel annotations with a Python subset called Pythran to bring fine-grain parallelism to Python, While being backward compatible with both the Python language, and the sequential algorithm. It means that:

1. any Pythran code can be run (sequentially), with no module dependency or code change, by any Python interpreter;
2. parallelism is explicit and incrementally added to the original code through directives.

3. OpenMP Semantical Adaptations

OpenMP is a standard API for parallel programming for Fortran, C and C++. It consists of a set of parallelizing directives and a few runtime library calls. If OpenMP is not activated, the directives are ignored, thus enabling incremental parallelization of the original source code while keeping the original code structure. The languages targeted by OpenMP are statically compiled. This section studies the semantical adaptation required to use the same API for a scripting language such as Python.

3.1 Directives

OpenMP directives are held by C/C++ pragma, or by Fortran comments. Most of them apply to structured blocks, represented by an instruction in C/C++ and delimited by comments in Fortran. A few directives (e.g. `threadprivate`) are not attached to a specific instruction.

While Python has a decorator mechanism[4], it only applies to functions, methods and classes and does not make it possible to attach decorations to other statements. Pythran uses the path taken by PLW [12], that uses string instructions to hold such decorations. As Python does not have anonymous block, [5] one has to create a dummy `if 1:` instruction to apply an annotation to a whole block. Alternatively, the syntax `if 'my annotation':` is also supported.

Many OpenMP annotations are parametrized by clauses that list variables, specifying their behavior with respect to parallel regions, e.g. `private`, `shared`, `copyin`. They can only refer to variables that have already been declared. However, there is no variable declaration in Python, and all variables assigned in a function have the function scope. Consequently, all variables that are referenced in a function are considered when building such variable lists: there is no concept of variable local to a block.

The `reduction(`*`operator`*`: `*`list`*`)` directive is used to characterize some data dependencies when performing a parallel reduction. The list of supported operators depend on the input and

[4] *cf.* PEP 318, `http://www.python.org/dev/peps/pep-0318/` for a more detail explanation of Python decorators.

[5] *cf.* PEP340, `http://www.python.org/dev/peps/pep-0340/` for a discussion concerning anonymous block support in Python.

Listing 1. Parallel loop in Pythran with tuple unpacking.

```
'omp parallel for'
for i, v in enumerate([2, 3, 5, 7, 11]):
    print i, ':', v
```

backend languages: Python has `min`/`max` operators but C/C++ does not. C/C++ have `&&` or `||` while Python does not.[6]

3.1.1 Parallel For and Iterators

The core directive to handle data parallelism is the `for` directive that distributes the iteration space of the associated loop among the existing threads. To be compatible with OpenMP, the loop iteration space must be described by a random access iterator with a total order. Integers used as loop indices in Fortran and C satisfy this conditions, as well as C++ iterators with the `random_access_tag` trait. But a Python iterator only advances by a step of one until it is exhausted, in which case it raises an exception: it behaves as a forward iterator. To be compatible with OpenMP, these iterators must be turned into random access iterators.

The extension of Python iterator to random access iterators is direct for the standard containers: `list`[7], `set` or `dict`. Other iterators require more care.

Generators, Python objects that behave like iterators, are commonly used in Python. The simplest one, `xrange(`*start*`,` *stop*`,` *step*`)`, successively yields value starting from *start* to *stop* by a step of *step*. It is easily extended to support random access, but it generally does not make sense to use a generator as a loop iterator, as the relation between two random states of the iterator may be of an arbitrary complexity.

Generator expressions are generators whose content is built from an other iterator. For instance `(x*x for x in l)` successively yields the square of each element in `l`. They behave like adaptors: they apply a particular expression on each value of the iterator. It is also the case of the `enumerate` builtin, that yields each element of the enumerated iterator associated with its index. These generators are random access iterators only if their input iterator is a random iterator itself.

Finally, if the iterator yields a tuple, it is possible to unpack it inside the `for` construct, as shown in Listing 1. In that case all the unpacked variables are considered as iterators, especially with respect to default privatization rules: in the given example, `i` and `v` are private, and the parallel iteration is valid because the input of `enumerate` is a list, which allows random access iteration.

Pythran's runtime is aware of these three kind of iterators and supports parallel iteration over random access iterators and iterators adapting random ones.

3.2 Runtime Library

OpenMP provides a small runtime library that, for instance, makes it possible to retrieve the active thread id. All the functions are declared in the `<omp.h>` header, and have a default behavior when OpenMP is not activated.

Providing a binding to these library in Python as an `omp` module does not raise particular problems, as the signature of these functions only involves integers passed as parameters or return values, except for the mutex manipulation. In that case an opaque type is used to represent the native type.

The `_OPENMP` macro definition is always provided when OpenMP is activated, and can be used to detect when OpenMP is not available. Python does not have a preprocessor, but it is possible to catch the import exception if the `omp` module is not found.

[6] Although similar, the `and` and `or` keywords have a special meaning in Python.

[7] Python `lists` behave as C++ `vectors`.

Listing 2. Example of Python OpenMP validation test case.

```
def omp_parallel_for_if(loop_count):
  import omp
  using = sum = 0
  'omp parallel for if(using == 1)'
  for i in range(loop_count + 1):
    num_threads = omp.get_num_threads()
    sum += i
  known_sum = (loop_count *
               (loop_count + 1)) / 2
  return known_sum == sum and num_threads == 1
```

3.3 Validation

A validation suite for OpenMP is proposed in [22] for C and Fortran. We ported it to Python, and also extended it to validate the corner cases specific to Python described in this section. A typical test case is given in Listing 2.

We have used the Pythran tool described in the following section to turn each Python test function into a C++ function with the same directives and runtime calls. Apart from the `threadprivate` directive and the `collapse(`*n*`)` clause, all tests were successful. `threadprivate` directives were held by global variables not supported in Pythran yet ; and the C++ code generated by Pythran does not preserve the perfect loop nesting required by the `collapse` clause.

4. Static Translation of Python Programs

Pythran is a subset of Python designed for scientific computing. It is implicitly statically typed and supports most Python constructs except those that involve introspection (e.g. `getattr`) or runtime compilation (e.g. `eval`). A few standard modules are supported in addition to the core language (e.g. `math`, `random`). The `numpy` module is currently only partially supported. User classes are not supported. The associated compiler turns Pythran code, possibly annotated with OpenMP and a few function type annotations, into C++ code. Its processing is summarized in Figure 1 and relies on type inference to remove most dynamic behavior, combined with a high-level runtime library, `pythonic++`, to allow a one-to-one mapping between Python and C++ constructs. The purpose of this section is not to dig into the internals of the compiler, but rather to focus on the impact of the parallelism layer, especially on the runtime library.

4.1 Runtime Support

Pythran runtime is based on the STL. The standard library of C++11, unlike its C++03 version, is thread-safe, so no data dependencies are added to the original code.

A critical point of the design of the `pythonic++` runtime is memory management, the very same aspect that led to the use of a GIL in CPython. Memory management is implemented in Shed Skin through the general-purpose Boehm garbage collector [4], and Cython forbids usage of Python-managed objects inside parallel regions, thus making memory management explicit for these parts.

Pythran handles the problem by refusing recursive types, which makes it possible to solely rely on reference counting for memory management. It can be implemented through the thread-safe shared pointer mechanism provided by the C++11 standard library.

Using shared references simplifies memory management, but the counterpart is an extra atomic operation for each copy. As it

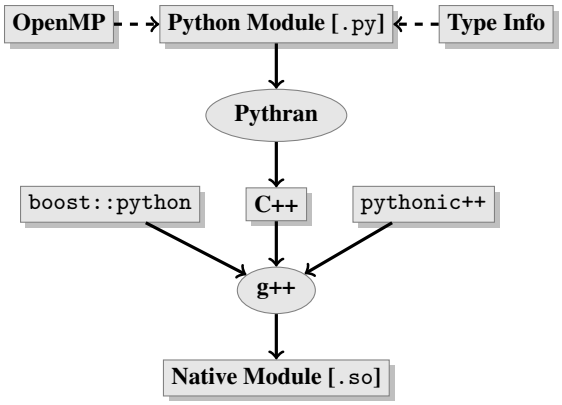


Figure 1. Pythran compiler workflow.

limits parallelism, reducing the number of copies becomes an important goal. The move semantic introduced in C++11 avoids a few copies when working on temporary objects, but argument passing still implies copies. To avoid a reference increment, one can pass parameters by reference. Using an out-of-scope interprocedural memory effect analysis, Pythran determines for each argument whether it must be passed using reference or const reference to prevent this overhead.

4.2 Directive Oblivious Translation

During Python to C++ translation, Pythran adopts a blind strategy: it does not understand the semantics of the annotations. Instead, it just splits each annotation into a context-sensitive part —the variable names— and a context-insensitive part —the clauses—, and attaches them to the proper instruction in the AST.

The Pythran compiler ensures a bijective translation between Python instruction and generated C++ instruction so that the OpenMP directive is regenerated on the proper instruction. The same approach is used at the expression level, to be compatible with the `if` clause.

At the AST level, it means that the Python AST is first reduced to a tree where all nodes not available in C++ are transformed into a compatible representation. For instance, list comprehension expressions are turned into function calls or tuple unpacking is turned in multiple assignments. During this transformation process, an expression is always transformed into a single expression and a statement is always transformed into a single statement. Then this AST is converted into a C++ AST meant to be pretty-printed.

4.3 Transfer Costs

When passing containers from Python to C back and forth, a copy of the whole container is made to turn the type agnostic, non-contiguous Python data into dense typed ones. This extra copy implies an extra cost that is not negligible: Passing two lists of float from Python to C++ requires as much as half the time to compute the dot product of the same lists directly in Python. Following Amdahl's law, this copy overhead greatly hinders the benefits of parallelization, and is a well known issue.

The traditional solution is to use a native type that exposes a Python interface and has a constant translation cost. NumPy's `ndarray` is a typical implementation of this concept and is the keystone of the Scipy and NumPy packages. Basically, a NumPy `ndarray` is a raw C pointer that is exposed at the Python level. As a tool for scientific computing, Pythran supports such a structure.

Table 1. Comparison of several versions of the Hyantes prototype.

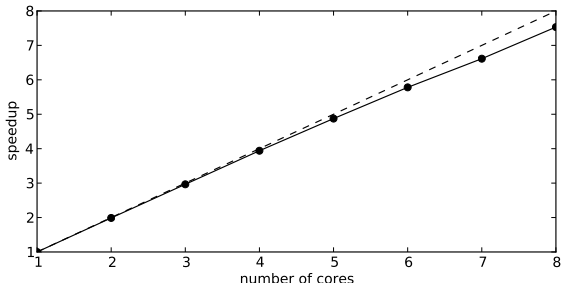
Language	SLOC	wall time (s) [gcc]	wall time (s) [clang]
C	102	25	22
Python	30	676	–
Pythran	30	38	22
Pythran+OMP	31	11	no OpenMP support

5. Validation

To validate the approach proposed in this paper, we first performed a simple experiment. Starting from the C code of a small geomatics application, Hyantes, we successively parallelized it with OpenMP, turned it into Python with a Pythran compatible kernel and parallelized the Python version using the approach described in this paper. We also measure the SLOC of the two versions. Table 1 summarizes the results of the experimentation made on a laptop with 4 non hyperthreaded cores using `-O2` compiler flag. It not only emphasises the use of a high level language prototyping, but also shows that it is possible to turn this prototype into reasonably efficient code through Pythran. It is also possible to prototype the parallel version while remaining at the Python level.

The Hyantes programs scales relatively well. Figure 2 shows the relative speedup of the application when increasing the number of OpenMP threads using up to 8 AMD Opteron 6176 SE.

Figure 2. Relative speedups of Pythran+OpenMP version of the Hyantes prototype.



We then compare Pythran with Cython. The approaches share some similarities: they both generate code written in a lower level language, with OpenMP directives. However, the inputs differ as Cython requires more typing information than Pythran to generate efficient code, and Cython is not backward-compatible with Python. They translate explicit fine-grained parallelism through code annotations or specific Python constructs, respectively.

Pythran exposes the full OpenMP interface to the user, thus enabling both data and task parallelism, as described in Section 3. It is not the case in Cython:

- Only loops can be made parallel, using a new `prange` generator.
- Reductions and variable privacy are inferred, but it chokes on reduction on private variables.
- The user is responsible from releasing the GIL inside parallel regions.
- It is impossible to use a function imported from a Python module in a parallel region, but it is still possible to use native C functions.

Listings 3 and 4 illustrate the difference between the two and illustrates the intrusive behavior of Cython.

The performance of the two approaches is shown in Figure 3. The benchmarked codes are typical mathematical, image-processing or linear algebra kernels. All these kernels have been

Listing 3. Cython implementation of a parallel reduction.

```
from libc.math cimport sqrt
from cython.parallel import parallel, prange
def sum_sqrt(double r):
    cdef int i
    with nogil, parallel():
        for i in prange(10000000):
            r += sqrt(i)
    return r
```

Listing 4. Pythran implementation of a parallel reduction.

```
#pythran export sum_sqrt(float)
import math
def sum_sqrt(r):
    "omp␣parallel␣for␣reduction(+:r)"
    for i in xrange(10000000):
        r += math.sqrt(i)
    return r
```

written in Cython and Python —compatible with Pythran— and annotated through the mechanism of each language to exhibit parallelism, then compiled into native code using GCC 4.7.2 and the `-O2 -fopenmp` flag. Their execution time when called from the Python interpreter is measured through the `timeit` module on a machine using 4 AMD Opteron 6176 SE. All results are normalized against Cython sequential execution time. They show that while handling code at a higher level than Cython, Pythran achieves comparable results.

The source codes used for these benchmarks are available on the Pythran repository.[8]

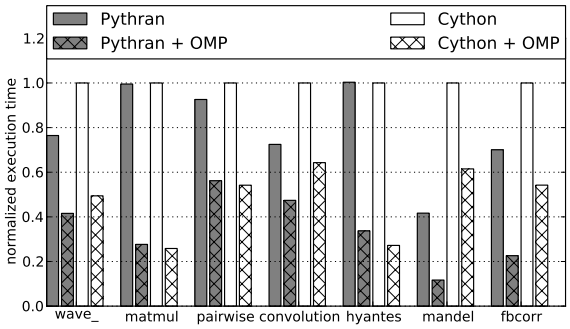


Figure 3. Comparison of Cython and Pythran generated code performance.

6. Conclusion and Future Work

This paper studies the addition of OpenMP annotations to Python. It shows that providing minor semantic adaptations, these annotations make it possible for Python code to benefit from multicores while retaining backward compatibility and without worrying about the Global Interpreter Lock.

To achieve this goal, a translator from a subset of the Python language to C++ has been designed. This translator turns regular Python modules annotated with OpenMP directives and a few type annotations into native parallel module. The input module remains compatible with the standard interpreter.

[8] *cf.* `https://github.com/serge-sans-paille/pythran` on `dls2013` branch.

The approach is compared with Cython, an extension of the Python language used to generate native module with an hybrid Python-C syntax that also provides means to exhibit fine grained parallelism. It shows that retaining Python compatibility does not prevent the achievement of comparable performances.

Future work includes the extension of the approach to OpenMP 4 and the `target` clause that should make it possible to target MIC from Python. There is also many vectorization possibility in Python, especially in the list comprehension construction, that needs to be explored.

Acknowledgments

This work has received fundings from SILKAN and the French ANR through the CARP project. The authors thank Adrien MERLINI and Alan RAYNAUD for their work on the runtime, Albert COHEN, Béatrice CREUSILLET, Fabien DAGNAT, Mehdi AMINI, Fran cois IRIGOIN and Ronan KERYELL for their valuable reviews.

References

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

[2] D. M. Beazley. Automated scientific software scripting with SWIG. *Future Generation Computer Systems*, 19(5):599–609, July 2003. ISSN 0167-739X.

[3] 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.

[4] H.-J. Boehm, A. J. Demers, and S. Shenker. Mostly parallel garbage collection. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation*, PLDI '91, pages 157–164, New York, NY, USA, 1991. ACM.

[5] 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.

[6] B. Catanzaro, M. Garland, and K. Keutzer. Copperhead: compiling an embedded data parallel language. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, pages 47–56, New York, NY, USA, 2011. ACM.

[7] R. Choy, A. Edelman, and C. M. Of. Parallel MATLAB: Doing it right. In *Proceedings of the IEEE*, pages 331–341, 2005.

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

[9] 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.

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

[11] A. Klöckner, N. Pinto, Y. Lee, B. C. Catanzaro, P. Ivanov, and A. Fasih. PyCUDA and PyOpenCL: A scripting-based approach to GPU runtime code generation. *Parallel Computing*, 38(3):157–174, 2012.

[12] P. Luszczek and J. Dongarra. High performance development for high end computing with Python language wrapper (PLW). *International Journal of High Performance Computing Applications*, 21(3):360–369, Aug. 2007. ISSN 1094-3420.

[13] X. Ma, J. Li, and N. F. Samatova. Automatic parallelization of scripting languages: Toward transparent desktop parallel computing. In *IPDPS*, pages 1–6. IEEE, 2007.

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

[15] openmp. OpenMP application program interface. http://www.openmp.org/mp-documents/OpenMP3.1.pdf, July 2011.

[16] N. Riley and C. Zilles. Hardware tansactional memory support for lightweight dynamic language evolution. In *Companion to the 21st*

ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications, OOPSLA '06, pages 998–1008, New York, NY, USA, 2006. ACM.

[17] F. Tabba. Adding concurrency in python using a commercial processor's hardware transactional memory support. *SIGARCH Computer Architecture News*, 38(5):12–19, Apr. 2010.

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

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

[20] G. van Rossum and F. L. J. Drake, editors. *Thread State and the Global Interpreter Lock*. In van Rossum and Drake [21], Sept. 2012.

[21] G. van Rossum and F. L. J. Drake, editors. *Python/C API Reference Manual*. Python Software Foundation, Sept. 2012.

[22] C. Wang, S. Chandrasekaran, and B. M. Chapman. An OpenMP 3.1 validation testsuite. In *8th International Workshop on OpenMP (IWOMP)*, volume 7312 of *Lecture Notes in Computer Science*, pages 237–249, Rome, Italy, June 2012. Springer.