Scripting Language Performance Through Interoperability

Simon A. F. Lund
Niels Bohr Institute, University of Copenhagen
safl@nbi.dk

Bradford L. Chamberlain
Cray, Inc.
bradc@cray.com

Brian Vinter
Niels Bohr Institute, University of Copenhagen
vinter@nbi.dk

1. Introduction

Attractiveness of programming in scripting languages can arguably be attributed to language features, removal of responsibilities from the programmer, and a rich execution environment. Attractive language features include dynamic typing and type inference supporting generic and less verbose code. Dynamic, managed memory, and garbage collection removes the error-prone tasks of allocating, reallocating and freeing memory from the programmer. Interactive interpreters facilitates experimentation and rapid application development. The lack of features such as parallel language constructs gives the programmer straightforward sequential semantics without concern to the hazards related with parallel execution. Scripting languages are often labeled as high-level since they remove these responsibilities from the programmer, allowing for code to evolve around manipulating abstractions closer to the application domain.

The price for these conveniences is often paid with lowered hardware utilization since the programmer only expresses *what* is to be computed not *how* to compute it. With a lack of control and no means of obtaining it, it becomes the task of the interpreter to map high-level application code to hardware efficiently.

One approach is to rely on language interoperability to increase application performance. Using Python as an example, the CPython interpreter allows for interoperability with C/C++, either via language extensions or by providing access to libraries.

Python has seen widespread use and popularity using this approach, specifically the NumPy/SciPy/iPython software stack gives the programmer a rich interactive environment for scientific computing. This is achieved by maintaining high-level abstractions and enabling significant performance improvements by implementing the computationally demanding portions in C and providing access to them via high-level data structures and operations upon them.

This becomes more challenging (or important) as efficient utilization of hardware becomes increasingly complex with continuing developments in hardware architectures such as increasing core counts on CPUs with NUMA architectures, distinct address spaces in accelerators such as GPUs, MICs, FPGAs. Interpreters are rarely able to keep up with the developments in hardware, and the abstractions provided by scripting languages will fail to deliver the potential use of available hardware. At this point, the need arises to enable the programmer to peel off layers of the language abstractions and take control.

Current approaches for doing so in Python include writing Python extensions, using SWIG or ctypes. In addition low-level APIs such as pyCUDA, pyOpenCL, pyMIC, and MPI4PY can be used.

These approaches successfully provide the means of recovering the majority of the lost performance and have contributed to Python's popularity in HPC environments serving as a steering-language for scientific computing. Language interoperability is the driver for the realization of these tools and libraries.

However, they come at the price of sacrificing all of the conveniences described earlier. The following sections introduce a different approach to foreign-function interfaces and interface generation which leverages the advantages of interoperability without sacrificing the attractive qualities of the scripting language.

2. Approach

The general idea is to provide a less verbose foreign function interface (FFI) by using introspection on the function definition and function decoration to construct a function prototype which can either be implemented inline, from file, or mapped to a library.

In order to maintain the high-level abstractions, the foreign language must support them. C does not meet this requirement, however, Chapel provides an ideal target. It supports high-level operations on arrays and also supports peeling off layers of the abstraction allowing the programmer to take control if they have sufficient motivation for doing so.

```
1  def simulation(tsteps, spin, decay, velocity):
2      """Simulation of some physical phenomenon on a dataset."""
3
4      delta = velocity
5      for _ in xrange(0, tsteps):
6          delta = (spin * decay + velocity) / delta
7
8      return np.add.reduce(delta)
```

Figure 1. Computationally expensive Python function.

```
1  @Chapel()
2  def simulation(tsteps=int, spin=np.ndarray, decay=np.ndarray, velocity=np.ndarray):
3      """
4      var delta = velocity;
5      for timestep in 1..tsteps {
6          delta = (spin * decay + velocity) / delta;
7      }
8      return +reduce(delta);
9      """
10     return float
```

Figure 2. Python function rewritten in Chapel using an inlined foreign function body.

PPOPP '14, Month 0–0, 2014, San Francisco, California, United States of America.

http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

A prototype implementation named *pyChapel*[1] serves to illustrate the approach. pyChapel consists of a foreign function interface and a module compiler / interface generator.

The idea is that a computationally expensive function such as the one in figure 1 can be rewritten in a foreign language or mapped to a library.

The raw foreign function body can be provided inline, from file or by mapping it to an existing library. When using pyChapel in this manner, Chapel code will be dynamically generated and compiled at runtime. Machinery within pyChapel lowers compilation overhead by re-using previously compiled code.

PyChapel also provides a module-compiler, compiling Chapel modules into Python modules as illustrated in figure 3.

```
module HelloLib {
    export
    proc hello_caller() {
        writeln("Hi Caller, I am Chapel, pleased to meet you.");
    }

    export
    proc add_ints(x: int, y: int) : int {
        return add_everything(x, y);
    }

    proc add_everything(x, y) {
        return x + y;
    }
}
```

```
 # pych --compile hellolib.chpl
```

Figure 3. Source code of the Chapel *HelloLib* module and command to compile it into a Python module.

The module can then be accessed from Python as illustrated in figure 4.

```
from hellolib import hello_caller, add_ints

if __name__ == "__main__":
  hello_caller()
  print add_ints(2, 3)
```

Figure 4. Accessing Chapel module *HelloLib* from Python.

When using the pyChapel module compiler, code will be generated and compiled prior to runtime.

3. Results

Two synthetic Python applications, hereafter referred to as *finance* and *scicomp*, were used to get an initial impression of the performance gained by mapping the computationally expensive portion of the application code to Chapel. The code can inspected in the pyChapel online documentation[2]. Two versions of each application were implemented, a reference implementation and a modification mapping the expensive functions (`simulation` and `quant`) to Chapel using techniques illustrated in figure 1 and figure 2. The rewrites were implemented without any explicit parallelism in order to maintain the abstraction-level of the code.

The machine executing was a laptop with 6GB of memory with an Intel i5-2410M CPU @ 2.3Ghz CPU with two physical cores. *finance* ran in $26.7s$ without pyChapel and $4.2s$ with. *scicomp* ran in $28.3s$ without pyChapel and $8.8s$ with. Resulting in speedups of $6.3\times$ and $3.2\times$, respectfully.

A third implementation of *scicomp* was written, replacing the `@Chapel` decorator with `@FromC`. Mapping the Python function `simulation` to a C implementation. Running the C-targeted implementation showed only a marginal decrease in wall-clock time compared to the Chapel-targeted implementation.

4. Future Work

The pyChapel module was created as a means of providing interoperability between Python and Chapel. However, it can with little effort be expanded to support any language capable of interoperating with C including but not limited to Fortran and Haskell and thereby providing a generic and simplified approach to FFIs in Python.

The dynamic compilation machinery in pyChapel opens up exploration within staged computation[1]. This can be done by manipulating inlined foreign function bodies at runtime as a means of generating optimized code based on run-time values of inputs, such as specializing input-sensitive BLAS and FFT routines.

Current work focuses on maintaining the attractiveness of Python by implementing pyChapel as a target for the npbackend[2] module. The array operations of Python/NumPy can thereby be mapped transparently to Chapel. Thereby, effectively increasing the performance of the Python/NumPy program without changing a single line of code.

5. Conclusion

The experimental prototype pyChapel proposes a new approach to foreign function interfaces and interface generation for Python. The primary target for the prototype is Chapel, however, C is currently also supported, and further work can with little effort expand support for any language capable of interoperating with C. Chapel is the primary target as it maintains the attractive qualities of the scripting languages such as high-level array operations for scientific computing. Initial results indicate that the performance improvement of targeting Chapel is equivalent to that of targeting C.

Acknowledgments

The Chapel Team at Cray, Inc. provided valuable insights during the development of PyChapel especially Elliot Ronaghan, Ben Harshbarger, Thomas Van Doren, and Lydia Duncan.

This research has been partially supported by the Danish Strategic Research Council, Program Committee for Strategic Growth Technologies, for the research center 'HIPERFIT: Functional High Performance Computing for Financial Information Technology' (hiperfit.dk) under contract number 10-092299.

This research has been partially supported by the Danish Strategic Research Council, Program Committee for Strategic Growth Technologies, for the 'High Performance High Productivity' project under contract number 09-067060.

References

[1] O. Kiselyov, C.-c. Shan, and Y. Kameyama, "Bridging the theory of staged programming languages and the practice of highperformance computing," Tech. Rep., 2012.

[2] M. R. B. Kristensen, S. A. F. Lund, T. Blum, and K. Skovhede, *Separating NumPy API from Implementation*, 2014.

[1] `http://pychapel.rtfd.org`

[2] `http://pychapel.readthedocs.org/usage_examples.html#accelerate-your-numpy-code`