unPython: Converting Python Numerical Programs into C

Rahul Garg (garg1@cs.ualberta.ca) – *University of Alberta*, Canada
Jose Nelson Amaral (amaral@cs.ualberta.ca) – *University of Alberta*, Canada

unPython is a Python-to-C compiler intended for numerical Python programs. The compiler takes as input type-annotated Python source and produces C source code for an equivalent extension module. The compiler is NumPy-aware and can convert most NumPy indexing or slicing operations into C array accesses. Furthermore the compiler also allows annotating certain for-loops as parallel and can generate OpenMP code thus providing an easy way to take advantage of multicore architectures.

Introduction

Python and NumPy form an excellent environment for numerical applications. However often performance of pure Python code is not enough and the user is forced to rewrite some critical portions of the application in C. Rewriting in C requires writing glue code, manual reference count management and knowledge of Python and NumPy C APIs. This reduces the programmer productivity substantially. Moreover rewriting a module in C obscures the logic of the original Python module within a large amount of boilerplate. Thus extension modules written in C can often become very hard to maintain.

To relieve the programmer from writing C code, we present unPython. unPython is a Python to C compiler that takes as input annotated Python code and produces as output C code for an equivalent extension module. To compile a module with unPython, a programmer adds annotations, such as type declarations, to a module. The programmer then invokes unPython compiler and unPython converts the Python source into C. Annotations are added in a non-interfering way such that the annotated Python code still remains valid Python and thus can still run on CPython interpreter giving the same results as the original unannotated Python code.

The distinguishing feature of unPython is that unPython is focused on compiling numerical applications and knows about NumPy arrays. unPython therefore has knowledge of indexing and slicing operations on NumPy arrays and converts them into efficient C array accesses. The other distinguishing feature of unPython is its support for parallel loops. We have introduced a new parallel loop notation thus allowing Python programmers to take advantage of multicores and SMPs easily from within Python code. While the code runs as serial loop on the interpreter, unPython converts specially marked loops into parallel C loops. This feature is especially important since CPython has no built-in support for true concurrency and therefore all existing solutions for parallelism in Python are process based. Moreover since parallel loops are inspired from models such as OpenMP [openmp], the parallel loop will be familiar to many programmers and is easier to deal with than a general thread-and-lock-based model.

Features

1. unPython is focused on numerical applications and hence can deal with int, float, double and NumPy array datatypes. Arbitrary precision arithmetic is not supported and the basic numeric types are converted into their C counterparts. NumPy array accesses are converted into C array accesses. Currently "long" integers are not supported but will be added after transition to Python 3.

2. To compile a function, a user specifies the type signature of the function. The type signature is provided through a decorator. When running on the interpreter, the decorator simply returns the decorated function as-is. However when compiled with the unPython compiler, the decorator takes on special meaning and is seen as a type declaration. The types of all local variables are automatically inferred. To facilitate type inference, unPython requires that the type of a variable should remain constant. In Python 3, we aim to replace decorators with function annotations. An example of the current decorator-based syntax is as follows:

```
# 2 types for 2 parameters
# last type specified for return type
@unpython.type('int','int','int')
def f(x,y):
  #compiler infers type of temp to be int
  temp = x + y
  return temp
```

3. User-defined classes are supported. However multiple inheritance is not currently supported. The programmer declares the types of the member variables as well as member functions. Currently types of member variables are specified as a string just before a class declaration. Subclassing builtin types such as int, float, NumPy arrays, etc. is also not supported. Dynamic features of Python such as descriptors, properties, staticmethods, classmethods, and metaclasses are currently not supported.

4. unPython does not currently support dynamic facilities such as exceptions, iterators, generators, runtime code generation, etc.

5. Arbitrary for-loops are not supported. However simple for-loops over range or xrange are supported and are converted into efficient C counterparts.

6. Parallel loops are supported. Parallel loops are loops where each iteration of the loop can be executed independently and in any order. Thus such a loop can be speeded up if multiple cores are present. To support parallel loops, we introduce a function called prange. prange is just a normal Python function which behaves exactly like xrange on the interpreter. However, when compiling with unpython, the compiler treats it as a parallel range declaration and treats each iteration of the corresponding loop as independent. A parallel loop is converted into corresponding OpenMP declarations. OpenMP is a parallel computing industry standard supported by most modern C compilers on multicore and SMP architectures. An example of a parallel loop:

```
#assume that x is a NumPy array
#the following loop will execute in parallel
for i in unpython.prange(100):
   x[i] = x[i] + 2
```

   Under some conditions, prange loops cannot be converted to parallel C code because CPython is not thread safe. For example, if a method call is present inside a parallel loop body, then the loop is currently not parallelized and is instead compiled to a serial loop. However prange loops containing only operations on scalar numeric datatypes or NumPy arrays can usually be parallelized by the compiler.

7. Preliminary support for homogeneous lists, tuples, and dictionaries is present.

Implementation

unPython is a modular compiler implemented as multiple separate components. The compiler operates as follows:

1. A Python script uses CPython's compiler module to read a Python source file and converts the source file into an Abstract Syntax Tree (AST). AST, as the name implies, is a tree-based representation of source code. unPython uses AST throughout the compiler as the primary method of representing code.

2. The AST formed is preprocessed and dumped into a temporary file.

3. The temporary file is then read back by the core of the compiler. The core of the compiler is implemented in Java and Scala. To read the temporary file, the compiler uses a parser generated by ANTLR. The parser reads the temporary file and returns the AST read from the file.

4. Now the compiler walks over the AST to check the user-supplied type information and adds type information to each node.

5. The typed AST undergoes several transformations. The objective of each transformation is to either optimize the code represented by the AST or to convert the AST to a representation closer to C source code. Each phase is called a "lowering" of the AST because with each transformation, the AST generally becomes closer to low-level C code than high-level Python code. The term "lowering" is inspired from the Open64 [open64] compiler which also uses a tree like structure as the intermediate representation.

6. A final code generation pass takes the simplified AST and generates C code. We are looking to further split this phase so that the compiler will first generate a very low level representation before generating C code. Splitting the final code generation into two phases will allow us to easily add new backends.

Most of the compiler is implemented in Java and Scala [scala]. Scala is a statically-typed hybrid functional and object-oriented language that provides facilities such as type inference, pattern matching and higher order functions not present in Java. Scala compiles to JVM bytecodes and provides easy interoperability with Java. The choice of implementation language was affected by several factors. First, by using languages running on the JVM, we were able to utilize the standard JVM libraries like various data structures as well as third party libraries such as ANTLR and FreeMarker. Second, distribution of compiler binaries is simplified since binaries run on the JVM and are platform independent. Further, both Java and Scala usually perform much faster than Python. Finally, Scala provides language features such as pattern matching which were found to considerably simplify the code.

Experimental Results

The platform for our evaluations was AMD Phenom x4 9550 with 2 GB RAM running 32-bit Linux. GCC 4.3 was used as the backend C compiler and "-O2 -fopenmp" flags were passed to the compiler unless otherwise noted. The test codes are available at http://www.cs.ualberta.ca/~garg1/scipy08/

Recursive benchmark : Compiled vs Interpreted

The programming language shootout [shootout] is a popular benchmark suite often used to get a quick overview of speed of simple tasks in a programming language. We chose integer and floating point versions of Fibonacci and Tak functions from "recursive" benchmark as a test case. The inputs to the functions were the same as the inputs in the shootout. We chose a simple Python implementation and measured the time required by the Python interpreter to complete the benchmark. Then type annotations were

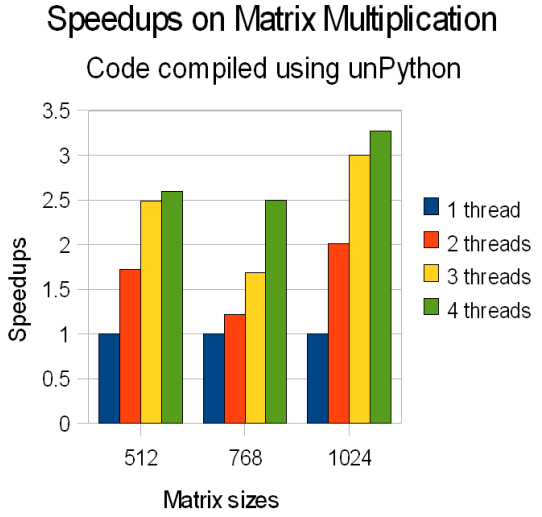
then added and the code was compiled to C using unPython.

The interpreted version finished the benchmark in 113.9 seconds while the compiled version finished in 0.77 seconds thus giving a speedup of 147x.

Matrix multiplication : Serial vs Parallel

We present experimental evaluation of the parallel loop construct "prange". We wrote a simple matrix multiplication function in Python to multiply two numpy arrays of doubles. The function was written as a 3-level loop nest with the outer loop parallelized using prange while the inner two loops were xrange.

We measured the performance of C + OpenMP code generated by unPython. For each matrix size, the number of threads was varied from 1 to 4 to obtain the execution time. The execution times for each matrix size were then divided by the execution time of 1 thread for that particular matrix size. The resulting speedups are shown in the following plot.



We also measured the execution time of a purely serial version of matrix multiplication with no parallel loops to measure the overhead of OpenMP on single thread performance. We found that the difference in execution time of the serial version and 1-thread OpenMP version was nearly zero in each case. Thus in this case we found no parallelization overhead over a serial version.

Related Work

Several other Python compilers are under development. Cython [cython] is a fork of Pyrex [pyrex] compiler. Cython takes as input a language similar to Python but with optional type declarations in a C like syntax. Pyrex/Cython produces C code for extension modules. Cython is a widely used tool and supports more Python features than unPython. Cython recently added support for efficient access to NumPy arrays using the Python buffer interface. Cython does not support parallel loops currently.

Shedskin [shedskin] is a Python to C++ compiler which aims to produce C++ code from Python code without any linking to Python interpreter. Shedskin relies on global type inference. Shedskin does not directly support numpy arrays but instead provides more efficient support for list datatype.

PyPy [pypy] is a project to implement Python in Python. PyPy project also includes a RPython to C compiler. RPython is a restricted statically typable subset of Python. PyPy has experimental support for NumPy.

Future Work

unPython is a young compiler and a work in progress. Several important changes are expected over the next year.

1. Broader support for NumPy is under development. We intend to support most methods and functions provided by the NumPy library. Support for user defined ufuncs is also planned.

2. Lack of support for exceptions is currently the weakest point of unPython. However exception support for Python is quite expensive to implement in terms of performance.

   NumPy array accesses can throw out-of-bounds exceptions. Similarly core datatypes, such as lists, can also throw many exceptions. Due to the dynamic nature of Python, even an object field access can throw an exception. Thus we are searching for a solution to deal with exceptions in a more selective manner where the user should be able to trade-off safety and performance. We are looking at prior work done in languages such as Java.

3. We intend to continue our work on parallel computing in three major directions. First we intend to investigate generation of more efficient OpenMP code. Second, we will investigate compilation to GPU architectures. Finally research is also being done on more general parallel loop support.

4. Support for the Python standard library module ctypes is also planned. ctypes allows constructing interfaces to C libraries in pure Python.

5. Research is also being conducted on more sophisticated compiler analysis such as dependence analysis.

Conclusion

The paper describes unPython, a modern compiler infrastructure for Python. unPython is a relatively young compiler infrastructure and has not yet reached its full potential. unPython has twin goals of great

performance and easily accessible parallel computing. The compiler has a long way to go but we believe with community participation, the compiler will achieve its goals over the next few years and will become a very important tool for the Python community. unPython is made available under GPLv3 at http://code.google.com/p/unpython.

References

[cython] http://cython.org
[open64] http://www.open64.net
[openmp] http://openmp.org
[pypy] http://codespeak.net/pypy
[pyrex] http://www.cosc.canterbury.ac.nz/greg.ewing/python/Pyrex/
[scala] http://www.scala-lang.org
[shedskin] http://code.google.com/p/shedskin
[shootout] http://shootout.alioth.debian.org