Numba: A LLVM-based Python JIT Compiler

Siu Kwan Lam
Continuum Analytics
Austin, Texas
slam@continuum.io

Antoine Pitrou
Continuum Analytics
Austin, Texas
apitrou@continuum.io

Stanley Seibert
Continuum Analytics
Austin, Texas
sseibert@continuum.io

ABSTRACT

Dynamic, interpreted languages, like Python, are attractive for domain-experts and scientists experimenting with new ideas. However, the performance of the interpreter is often a barrier when scaling to larger data sets. This paper presents a just-in-time compiler for Python that focuses in scientific and array-oriented computing. Starting with the simple syntax of Python, Numba compiles a subset of the language into efficient machine code that is comparable in performance to a traditional compiled language. In addition, we share our experience in building a JIT compiler using LLVM[1].

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Compilers, Optimization*

General Terms

Languages, Performance

Keywords

LLVM, Python, Compiler

1. INTRODUCTION

Python is a dynamically typed, interpreted language. It is regarded as a high productivity language due to its simple syntax, flexible semantics and the large number of libraries. As a result, it has gained popularity in the scientific computing community[2], which has driven the demand for high performance tools. Among many performance-focused libraries in Python, NumPy[3] is one of the important scientific libraries in the Python ecosystem as it provides a multi-dimensional array (ndarray) object that has become the foundation of efficient numeric computation in Python.

A ndarray is a homogeneously-typed data memory buffer. Data can be stored in either C contiguous, FORTRAN contiguous or have arbitrary strides at each dimension. This allows binding to high performance computing libraries that are traditionally written in C or FORTRAN, such as BLAS and LAPACK. The ndarray object also supports array expressions: using Python operators on ndarrays will trigger element-wise operations that are implemented in C efficiently. This provides a large speedup over interpreted Python, where looping over the elements in Python is inefficient due to the multiple layers of indirection inside interpreted code.

Numba[1] was initially developed to optimize the inefficient use-cases of NumPy. The layers of indirection for ndarray indexing in the interpreter are translated into direct loads and stores from pointers. Before Numba, NumPy users had to write Python C extensions (or use hybrid languages like Cython [4]) to implement any custom computation in an efficient way. This process can be error-prone due to the difficulty of manually managing the reference counts of Python objects, and generally requires a lot of boilerplate code even for simple use cases. Numba lets users annotate a compute-intensive Python function for compilation without rewriting the code in a low-level language.

2. A JIT FOR NUMERIC PYTHON

Numba is a function-at-a-time Just-in-Time (JIT) compiler for CPython [2], the most popular implementation of Python which, as the name suggests, is written in C. Numba is implemented as a library that can be loaded by programs running in the CPython interpreter and does not replace the interpreter itself. Its initial focus is to target a Python subset that makes heavy use of ndarrays and numeric scalars in loops so that users no longer need to rewrite their Python code in low-level languages for better performance. The ndarray provides important dimensionality, type and data layout information that allows Numba to generate specialized loops for arrays in machine code. By knowing the native in-memory structure of ndarray objects, Numba bypasses unnecessary indirection for accessing data in ndarrays. The basic structure of the ndarray consists of the data pointer to the base of the memory buffer and two integer arrays describing the dimensionality (aka shape) and the strides between elements along each dimension. Numba can directly access these fields for calculating the offset of an element given an index value. As a result, it can generate efficient loops that index into ndarrays with performance similar to equivalent code written in compiled language (see Table 1).

LLVM-HPC2015, November 15 - 20 2015, Austin, TX, USA
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-4005-2/15/11$15.00.
DOI: http://dx.doi.org/10.1145/2833157.2833162.

[1]`http://numba.pydata.org/`
[2]`https://www.python.org/`

Table 1: Numba (version 0.20) and pure C speedups over CPython (version 3.4) for a naive implementation of matrix multiplication.

Matrix Size	Numba	C
64 x 64	463x	453x
128 x 128	454x	407x
256 x 256	280x	263x
512 x 512	276x	268x

As Numba continues to develop, the supported subset of the Python language and standard library data types is expanding. For example, Numba can already generate efficient code for high-level Python/NumPy features such as generators and array expressions.

Unlike most JIT compilers for interpreted languages, Numba does not perform tracing nor replace the interpreter. Instead, it relies on the user actively transforming the Python functions that need compiling at runtime. In Python, this is done by applying a decorator to the function. The decorator replaces the original Python function with a special object that just-in-time compiles the function when it is first called with a new type signature. To relieve users from the burden of explicit type annotation, Numba inspects the types of the arguments and performs local type inference on the function. As a result, the compiler can have accurate type information for each value in the function without tracing the execution.

The semantics of the Numba-compiled code differ slightly from code interpreted by CPython. Numba only supports a subset of the language and violates some semantics of Python in a few specific areas that are generally acceptable for high performance numeric use cases. The compiler does not support the "big integer" representation used in Python to represent integers of arbitrary size. Instead, it is limited to fixed width integer types up to 64-bits in size, with semantics similar to many C platforms. Polymorphic types are not supported during execution, requiring the compiler to be able to resolve all polymorphism at compile time (when the function is called). Implicit coercion occurs when a type is assigned to a variable of a different type. In practice, these restrictions and deviations from Python semantics rarely affect the user because the ndarray object already has similar limitations. On the other hand, these limitations allow for aggressive optimization by the compiler. As Numba continues to develop, we aim to remove these limitations when practical, but leave an option for user permit the violation of Python semantics for optimization reason.

One novel feature of Numba is its support for targeting different hardware. It currently provides an NVIDIA CUDA[5] back-end using the NVVM[3] library, and an AMD HSA[6] back-end is also available on APU by using HLC[4]. Both NVVM and HLC are vendor-specific version of LLVM with additional support for their hardware. Rather than attempting to create a portable execution model supported by all targets, Numba directly exposes the execution model of each GPGPU architecture to Python. This allows the user to tailor their code to the specific features and performance characteristics of each architecture. Numba provides access to platform specific operations, such as thread barriers and atomic operations on GPGPU targets.

[3]http://docs.nvidia.com/cuda/nvvm-ir-spec/

[4]https://github.com/HSAFoundation/HLC-HSAIL-Development-LLVM

3. OTHER JUST-IN-TIME COMPILERS FOR PYTHON

There are or were several other efforts to build JIT compilers for Python. Closest to Numba perhaps was Psyco[5], an opt-in compiler layered on the CPython interpreter. Psyco featured its own x86-only code generator and was discontinued by its author, Armin Rigo, because its architecture made maintenance and evolution difficult. [7]

Another related effort was Unladen Swallow[6], a project sponsored by Google to produce an evolution of the CPython interpreter augmented with a LLVM-based Just-in-Time compiler. The objective of Unladen Swallow was to support the full range of existing Python code, and to make the generated code progressively faster by adding "proven" optimizations. The project was discontinued before it got significant results for reasons which were explained by one of its authors. [8]

Other Python JIT compilers have avoided building on CPython, either re-using a significant portion of the CPython runtime or building their own. Of the two main active projects today, PyPy [9] has built its own compilation infrastructure from scratch, whereas the other Pyston[7] is using LLVM. Both claim or intend to support the full range of features Python offers.

Compared to those, Numba is much more opportunistic and opinionated. It is opportunistic because of the focus on a narrow subset of Python's use-cases from which it aims to extract extremely good performance, comparable to what C or FORTRAN code would achieve (or even better, when running on the GPU). It is opinionated because it prefers and promotes the style of array-oriented programming via the use of NumPy ndarrays. Numba is also designed to support different execution models, allowing both CPU and GPU code to be handled by a common compiler framework.

4. IMPLEMENTATION

Numba relies on user annotation by using a Python decorator (@jit) on functions. The decorator substitutes the function object with a special object that triggers the compilation when called. When a decorated function is called, the call arguments are inspected. If the set of argument types have not appeared before, Numba will compile a specialized version of the function for the given types. Otherwise, the previously compiled version is called.

The following example illustrates the usage of the @jit decorator on a function:

```
from numba import jit

@jit
def foo(a):
  return a + a
```

The compilation starts by converting the Python bytecode into the an intermediate representation (IR) on which the type inference is performed. If the type of each value in the

[5]http://psyco.sourceforge.net/

[6]https://code.google.com/p/unladen-swallow/

[7]https://github.com/dropbox/pyston

IR can be inferred, the IR is lowered to LLVM, which then emits the final machine code. We call this the *nopython mode* because all operations can be lowered into efficient machine code without relying on the Python runtime for any operation. A *nopython mode* function can be executed without the *global interpreter lock* (GIL) and is able to run in parallel threads. If Numba cannot determine the type of one of the values in the IR, it assumes to all values in the function to be a Python object. In such situations, Numba must use the Python C-API and rely on the Python runtime for the execution. The generated code would be equivalent to unrolling the interpreter loop; thus, removing the interpreter overhead. This compilation mode, called *object mode*, serves as a fallback if Numba cannot infer the type.

4.1 Bytecode

When calling a function during normal CPython execution, the interpreter creates a new frame for the function and executes the function bytecode. The bytecode is an instruction stream similar to x86 assembly. Instructions are variable length. Branches are encoded as absolute or relative jump instructions. Some bytecode instructions perform multiple tasks. For instance, the JUMP_IF_TRUE_OR_POP jumps to the target address if the value on the top-of-stack (TOS) evaluates to true; otherwise, pops the TOS value. Another complex instruction is the FOR_ITER instruction. It is used in the encoding of the for-loop construct. This instruction asks for the next value of the iterator at TOS. If the iterator is exhausted, it pops the stack and jumps to the end of the loop indicated by the operand of the instruction. Otherwise, the next value of the iterator is pushed on to the stack and proceed to the next instruction, which is the first instruction of the loop body. Both of these instructions change the control-flow and have optional stack-effect. [10] Instructions like these are common and they cannot be mapped directly to the low-level representation used by LLVM IR. For this reason, the bytecode must pass through several stages of analysis before translation.

4.2 Disassembling the bytecode

Translation of the bytecode to LLVM IR is not trivial due to the complexity of many common bytecode instructions. We need to disassemble the bytecode to recover the basic-blocks and to convert the stack machine into a register machine to facilitate further analysis and lowering.

The first step of disassembling the bytecode is to recover the control flow information. The bytecode is scanned for jump targets. The jump targets indicate the start of the basic-blocks. Jump instructions mark the end of basic-blocks. In some cases, this can be a simple process but, as mentioned in previous section, some bytecode instructions can perform operations on one of the branch targets. Heuristics are used to reconstruct the basic-blocks properly. As a result, our bytecode disassembler must be tailored to the specific behavior of each CPython version. Different versions of CPython can change both the bytecode instruction set and the way certain syntactic constructs are encoded.

Once all instructions are grouped into basic-blocks. We can simulate stack operations to assign a virtual register to each value. Each basic-block is simulated individually. Therefore, it is possible that the stack is empty when a pop is encountered. In that case, a *phi* node is inserted. Its incoming values are later connected to the lingering stack value from the incoming basic blocks.

4.3 Internal Representation

With the control-flow information and the stack-to-register mapping of the bytecode, the bytecode is translated into an internal representation, called the Numba IR. The Numba IR is a higher-level representation of the function logic than the bytecode. It captures the control-flow as basic-blocks and values in variables.

During the translation to the Numba IR, a notion of scoping is introduced to minimize the effect of mono-typing for variables. An assignment to a variable is considered a new variable definition when the definition is visible by all subsequent basic-blocks. Otherwise, the assignment stores the new value to the previous definition. For example:

```
@jit
def foo():
  a = 1
  bar(a)
  a = a + 2.5
  bar(a)
  a = a + 2j
  bar(a)
  return a
```

The three assignments to *a* will create three definitions that have types integer, float and complex, respectively. As a result, the three calls to *bar* will be calling different overloaded versions.

4.4 Type Inference

Local type inference is applied to the Numba IR at call time so that the type of the argument is known. A data-dependency graph is built for propagating the type of each value. Each node of the graph is a function call (built-in operators, such as addition, are implicit function calls). Knowledge of function signatures is encoded in a registry. Given the argument types, the type inferencer looks up the corresponding entry for the function and gets the return type. If a different return type is obtained for a value, the two types are unified by coercion with preference to avoid information loss. Coercion is only available to numeric types and certain built-in types. The data-dependency graph can contain cycles due to loops, so the type inferencer runs until a fixed-point is reached. If the type inference process fails, all values are assigned a generic "Python object" type.

4.5 High-Level Optimizations

With type annotation on each variable in the Numba IR, several high-level optimizations are performed before lowering. These optimizations exploit high-level knowledge of Python semantics.

4.5.1 Deferred Loop Specialization

Since loops are likely to be compute-intensive, Numba will extract the loops from function compiled in *object mode* into a new function. The new function acts like a Numba *@jit* decorated function and defers compilation until its call site is reached. This provides a second chance for Numba to optimize any compute intensive loop to run in nopython mode.

Detecting loops with CPython bytecode is very simple. The *SETUP_LOOP* bytecode describes the span of each loop. Numba can simply copy all bytecodes marked by the *SETUP_LOOP* into a new function. The loop in the original function is replaced with a function call to the extracted loop.

4.5.2 *Array Expression*

Array expressions are formed when built-in Python operators, such as addition and exponentiation, are applied to ndarray objects. After type inference, an optimization pass searches for any array expression and rewrites them into a loop. This avoids allocating temporary arrays for intermediate results when an array expression contains multiple operations. For example, consider this function:

```
@jit
def axpy(a, x, y):
    return a * x + y
```

In the *axpy* function that takes three ndarrays, the normal Python/NumPy execution will allocate an array for the intermediate result of multiplication and one for the final result from addition. Numba will rewrite this expression into a loop over the three arrays with just one allocated output array. Each iteration computes both the multiplication and addition for each element in the array, ensuring that registers can be used to hold intermediate scalar results.

4.6 Lowering

The lowering phase is straightforward. At this point, we have type information for all values in the Numba IR. Each operation is translated to LLVM IR according to a implementation registry for each function known to the compiler (both built-in and user-defined). For each Python function, two functions are emitted in LLVM: one for the actual compiled function, and a wrapper that acts as a bridge between the interpreted Python runtime and the compiled Numba runtime. The wrapper unboxes Python objects into machine representation for use as arguments by the compiled function. The returned values is boxed from the machine representation back to the Python objects.

4.7 Supporting GPGPUs

Numba supports GPGPU backends by directly exposing the parallel execution model of the hardware in a way similar to CUDA-C and OpenCL. A Python function can be decorated as a GPU kernel or a device function. The advantage of this design is the simplicity of the implementation, since the GPGPU backends can share most of the code generation logic with the CPU backend. The disadvantage is in the altering of the language semantic to fit the parallel execution model of the GPGPUs. While existing GPGPU programmers will be familiar with execution model, regular Python programmers with no GPGPU experience may find the change unintuitive.

To support the CUDA GPU, Numba provides both automatic and manual memory management between the GPU deivce memory and the CPU host memory. When a ndarray is used as an argument, the memory is automatically transferred to the device and the copied back when the kernel is completed. The transfer is transparent to the user but may perform unnecessary transfer since Numba cannot determine whether a ndarray is never modified. For maximum control, users can explicitly request memory transfer using the *to_device* and *copy_to_host* functions. It is important to note that HSA APUs have cache coherent shared memory between the CPU and GPU. A HSA kernel can consume any ndarray directly.

The following is an example of a CUDA kernel written using Numba.:

```
from numba import cuda

@cuda.jit
def copy_array(inp, out):
  blksz = cuda.blockDim.x
  blkid = cuda.blockIdx.x
  tid = cuda.threadIdx.x
  i = blksz * blkid + tid
  if i < out.size:
    out[i] = inp[i]

# launch kernel
copy_array[griddim, blockdim](inp, out)
```

The code implements a simple kernel that copies array *inp* to array *out*. Target specific intrinsic for reading thread identity is exposed as *cuda.blockIdx.x*, *cuda.blockDim.x* and *cuda.threadIdx.x*. To launch a kernel, the indexing operator (square brackets) is overloaded to provide the launch configuration (number of threads and blocks). The syntax mirrors the <<<...>>> syntax in CUDA-C.

5. USING LLVM

In this section, we describe our experience with LLVM. LLVM provides a simple API into a high quality compiler back-end with JIT support readily available. We can focus the development effort on the front-end without immense knowledge of the processor instruction set. Once we had a working CPU back-end, porting Numba to support CUDA and HSA GPUs was not difficult, since we map directly to the execution model of the GPUs. The GPU backend shares most of the logic of the CPU backend with extra exposure of target specific intrinsics to the Python language.

LLVM is arguably the best library for compiler developers but there are still some issues and potential improvement. In the rest of this section, we share several challenges we encountered and our experience in trying to workaround them.

5.1 API Stability

LLVM development is fast and its C++ API changes frequently. In early days of Numba development, we maintained a Python binding, called llvmpy, to the C++ API and have tried to mirror the C++ interface in Python, but this was found to be very difficult. To simplify the binding and to minimize its exposure to the LLVM C++ API, we adopted the LLVM C API instead and added any missing C wrappers around parts of the LLVM C++ API that were needed. The new binding is called llvmlite[8]. It includes a pure Python implementation of the the LLVM C++ IR-Builder API and build a string representation of the LLVM IR.

5.2 Error Handling

Error handling inside LLVM does not follow a consistent convention. Some errors are ignored or left unpropagated,

[8]`http://llvmlite.pydata.org/`

which can lead to hard crashes later on [9]. Other errors are reported by the API as a status code. Other errors yet trigger an assert() in the C++ source code, and therefore create a controlled process crash at runtime. Crashes (either controlled or unpredictable) do not allow the Python binding to report errors to the user in the expected way, i.e. as Python exceptions.

Furthermore, since error conditions are not exercised as much by the test suite, regressions can sometimes happen. [10]

5.3 Managing ABI

LLVM does not provide an abstraction for handling *application binary interface*(ABI). It is left for the front-end to handle this architecture specific detail. While we understand that ABI can be part of a language design, a facility for support basic C ABI inside LLVM would simplify most of the ABI issues.

To handle the different ABIs of various architectures, including GPGPUs which have more restrictive ABIs, Numba avoids using any aggregate types as function arguments or return types in the low-level code. Instead, all aggregates are flattened to simple scalars, e.g. integers, float and pointers.

To handle Python exceptions in compiled code, Numba uses the return value of the compiled function for error code. The actual return value is passed via the first argument. This error code passing style is the same as the CPython API.

5.4 Cross Module Linkage

The legacy JIT in LLVM supports lazy compilation and allows function-at-a-time code generation. However, the new MCJIT does not. With the legacy JIT replaced by MCJIT, Numba must compile each function in a new LLVM module and finalize it immediately to trigger code generation. If function *foo* calls *bar*, we need to register the address of *bar* to the execution engine before the finalization of *foo*; otherwise, the finalization of *foo* will fail due to unresolved reference to *bar*. This makes handling of mutual recursion difficult.

Separating each function into different modules also makes inlining difficult. We have to maintain reference to all LLVM modules of every compiled function and perform cross-module linking manually. A problem for this is that LLVM functions can be duplicated in many modules; thus, increasing the memory use.

We will be investigating whether the new On Request Compilation JIT alleviates some of these issues.

5.5 Object Cache

The MCJIT features callback hooks for applications to implement a form of object caching. Before compiling a module, the MCJIT execution engine calls the user-provided getObject() method. That method can either return a pointer to an existing memory area, in which case that memory area is treated as an object file containing the module's generated code (for example a ELF file under Linux), or a null pointer, in which case compilation goes on as usual.

Then, whenever the MCJIT execution engine has finished compiling a module (specifically, this means the getObject() method has returned a null pointer), it calls the notifyObjectCompiled() method providing both the module reference and a pointer to the memory area containing the module's object code.

This callback-based model works well for a trivial form of caching, where an LLVM module's object code is cached as-is, without any additional information[11]. However, more complex forms of caching need to serialize additional information beside the object code for the module. In Numba, this ancillary information is two-fold. First, we need to store metadata (a timestamp, a version number, and potentially other data) in order to decide at runtime whether the cached module is fresh or stale. Second, we also need to store non-LLVM data that is necessary to call the module's functions to be called, information which is a akin to a closure and is composed of arbitrarily complex Python objects.

Both pieces of information are not known at the LLVM module layer; they reside at a higher level of abstraction. We therefore had to turn the callback model upside-down by keeping the object code inside some long-lived variables, in order to be able to save or load the object code at the place and time where it is natural to do so.

For our use case, and presumably for other non-trivial JIT use cases, it would have been better to rely on a simple imperative API, allowing both to fetch a module's object code (when compiled), and to feed object code to the execution engine for a module.

A callback-based API is essentially useful when events are produced from the outside, and/or in an uncontrolled way: for example, GUI or network events. Here, the events are predictable by the application itself (compilation is triggered deterministically by calls to the execution engine API), therefore an imperative API would have been well-suited to the task.

6. CONCLUSIONS

We presented Numba, a Python JIT compiler that focuses in numeric performance, which is critical for many scientific application. It offers deferred loop specialization, array expression rewrite and multiple back-end support. Finally, we shared our experience and feedback in using LLVM.

7. ACKNOWLEDGMENTS

We would like to acknowledge the support of the DARPA XDATA program for this work.

8. REFERENCES

[1] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis and transformation. pages 75–88, San Jose, CA, USA, Mar 2004.

[2] Travis E. Oliphant. Python for scientific computing. *Computing in Science Engineering*, 9(3):10–20, May 2007.

[3] S. van der Walt, S.C. Colbert, and G. Varoquaux. The numpy array: A structure for efficient numerical computation. *Computing in Science Engineering*, 13(2):22–30, March 2011.

[9] `https://llvm.org/bugs/show_bug.cgi?id=6701`

[10] `https://llvm.org/bugs/show_bug.cgi?id=22368`

[11] This is how object caching is demonstrated in the official Kaleidoscope tutorial [11]

[4] 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.

[5] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, March 2008.

[6] HSA platform system architecture specification, version 1.0 final, 2015.

[7] Armin Rigo. Representation-based just-in-time specialization and the psyco prototype for python. In *Proceedings of the 2004 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 15–26. ACM, 2004.

[8] Reid Kleckner. Unladen Swallow retrospective. `http://qinsb.blogspot.fr/2011/03/unladen-swallow-retrospective.html`, 2015.

[9] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. Tracing the meta-level: Pypy's tracing jit compiler. In *Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, ICOOOLPS '09, pages 18–25, New York, NY, USA, 2009. ACM.

[10] Python documentation: dis package. `https://docs.python.org/3.4/library/dis.html`, 2015.

[11] Andy Kaylor. Object caching with the kaleidoscope example program — LLVM blog. `http://blog.llvm.org/2013/08/object-caching-with-kaleidoscope.html`, 2015.