Runtime Code Generation & Multi-Stage Programming in Python

[VICTOR KAMEL]

Programmers often appreciate the convenience of writing code in high-level, interpreted languages such as Python. However, the high level of abstractions in these languages conspire against the programmer by making the resulting code slow. When performance is required, this leads to the so-called "Two-Language Problem," where part of the code is written in a slow-yet-flexible language such as Python, and the performance-critical sections are written in a fast-yet-rigid language like C.

Fortunately, techniques such as multi-stage programming and just-in-time (JIT) compilation allow code written in interpreted languages to reclaim some of the lost performance by partially compiling some of the code at runtime. While many Python implementation and libraries aim to provide JIT compilation, there are very few that provide facilities for multi-stage programming. In this project, we investigate the performance benefits of implementing a prototype of runtime code generation and multi-stage programming into Python, demonstrating a mean speedup of up to 25× over normal Python functions. In addition, we demonstrate how multi-stage programming can be used to create a compiler for a simple tape-based programming language.

1 Introduction

Multi-stage programming (MSP) allows—with the help of special annotations—programming languages to specialize (compile) code at runtime in a type-safe manner. Thus, we can gain the advantages of compiled and optimized code in an interpreted language.[1] This also allows for programming techniques such as generative programming, where we can use staged compilation to craft programs such as compilers for domain-specific languages (DSLs) that can avoid part of the performance loss incurred by using the abundance of abstractions available in high-level languages.

In this project, we implement a prototype that enables runtime compilation and multi-stage programming in Python in order to explore its performance benefits. Since Python is an incredibly popular language, there are many projects that aim to accelerate Python code via JIT compilation, beyond a simple foreign function interface (FFI), explored in §5. We leverage similar methods to achieve runtime compilation. However, unlike those approaches, we implement staged compilation in order to enable generative programming. Our prototype[2] supports the following features, enabled via various mechanisms:

(1) compilation and optimization of functions written in a subset of Python to native code at runtime;
(2) multistage programming with splices and quotes;
(3) integer, float, boolean, and multi-dimensional array datatypes;
(4) basic control flow, labels, and goto statements; and
(5) calling external C functions;

Although Python is dynamically typed, since compilation now occurs at runtime, we can generate properly-typed code that is interoperable with the running Python instance as needed. We show that the generated functions offer a significant speedup over pure Python, and demonstrate that MSP can be used to create a compiler for a DSL.

[1] If we can amortize the cost of compilation, and the compiled code comprises a significant proportion of the runtime (Amdahl's law).
[2] Source code available: https://github.com/HeroHFM/msp

Author's Contact Information: [Victor Kamel].

2 Methodology

In this section, we present the design of our Python MSP module, and details of the corresponding implementation.

2.1 Compilation

In order to compile a Python function at runtime, we provide the `compile` decorator. All function parameters as well as the return type must have a type annotation. In order to compile the function, the following must occur:

(1) the source code of the function is retrieved using the `inspect`[3] module's `getsource` function;
(2) the source code is converted to an abstract syntax tree (AST) representation using the `ast`[4] module's `parse` function;
(3) the AST is translated into LLVM IR using a custom `ast.NodeVistor` and Numba's `llvmlite`[5] binding layer;
(4) the AST is used to determine the signature of the function; and
(5) the compiled function along with its signature are saved in memory.

Subsequently, `ctypes`[6] can be used to run the function from Python using Python datatypes as input arguments. We chose to use LLVM intermediate representation (IR), this allows us to leverage LLVM's optimization passes to optimize the generated code automatically. The following is an example of compiling and running a simple function using our library:

```
from msp.msp import *
gen = CGEngine.create_engine()

@gen.compile()
def add3(n : int) -> int:
    return n + 1 + 1 + 1

# >>> add3(0)
# 3
```

The first 2 lines import our module, and create a code generation engine object that will be subsequently used to compile code. The `compile` decorator automatically performs all steps outlined above, and makes the function available to call in the current environment. The AST of `add3` and the unoptimized LLVM IR translation is available in Appendix A. The optimized LLVM code is as follows:

```
; Function Attrs: mustprogress nofree norecurse nosync nounwind readnone willreturn
define i32 @add3(i32 %n) local_unnamed_addr #0 {
entry:
  %.5 = add i32 %n, 3
  ret i32 %.5
}

attributes #0 = { mustprogress nofree norecurse nosync nounwind readnone willreturn }
```

Note that the three additions were collapsed into a single operation. In general, LLVM can perform much more sophisticated optimizations than this given more complex code.

[3] https://docs.python.org/3/library/inspect.html
[4] https://docs.python.org/3/library/ast.html
[5] https://github.com/numba/llvmlite
[6] https://docs.python.org/3/library/ctypes.html

2.2 Multi-stage programming

MSP[7] typically uses the following code annotations in order to enable staging: quotes, splices, and a mechanism for running code. Quotes allow us to represent code fragments as a unit that can be manipulated (e.g. returned from a function), splices allow us to "splice in" quoted code into our function, and finally we require a mechanism to compile and run the code.

2.2.1 Quote. To enable quotation, we provide the user with the function `q`, which takes the code in the form of a string as a parameter. We chose to make the code inside of the quotation a string, as it allows more flexible manipulation of the quotation using Python. The function evaluates to a Python AST node via the `eval` function.

2.2.2 Splice. To enable splices, we define the `splice` function[8]. This function takes a regular Python expression, and evaluates it using Python's `eval` keyword. If this produces an AST node, the node gets spliced into the AST of the calling function. If the splice operation contains a function call, we must also perform a rewrite pass in order to replace the parameters with the appropriate arguments it was called with.

2.2.3 Run. Any function that is defined using the `compile` decorator is automatically compiled. When this function is subsequently called, the compiled version of the function is executed. The following example demonstrates these mechanisms in action.

```
1  def power(n, x):
2      if n == 0: return q('1')
3      return q('splice(power(n - 1, x)) * splice(x)')
4
5  @gen.compile(env = globals())
6  def power10(x : int) -> int:
7      return splice(power(10, q('x')))
8
9  # >>> power10(2)
10 # 1024
```

This code specializes the `power` function at runtime. Note that the `power` function is a regular Python function that returns an AST node (since both returns contain a quote). Since this function is not compiled and runs only at runtime, there is no restriction on what code it may contain. In this instance, the `compile` decorator must take an additional argument—the environment—in order to "capture" the definition of the `power` function. The return value splices in the AST node retrieved from `power`, where the first argument is the value 10, and the second is the quotation of `x`. Once the first node is returned, the resulting expression is evaluated, and the subsequent splices are also evaluated. Finally, we note that the quotation of `x` is used in the call to `power`, since the value of `x` is not known at compile time, and thus we cannot evaluate `power` with `x` directly. Now that we have specialized the function, we can generate optimized code:

```
1  ; Function Attrs: mustprogress nofree norecurse nosync nounwind readnone willreturn
2  define i32 @power10(i32 %x) local_unnamed_addr #0 {
3  entry:
4    %0 = mul i32 %x, %x
5    %1 = mul i32 %0, %0
6    %2 = mul i32 %1, %x
7    %3 = mul i32 %2, %2
8    ret i32 %3
```

[7] For a more complete introduction to multi-stage programming, see [Taha 2004].
[8] Available within our compiled functions only.

```
}

attributes #0 = { mustprogress nofree norecurse nosync nounwind readnone willreturn }
```

2.3 Control Flow

We support several types of control flow, including If statements and While statements. More Python control flow structures can be added without much difficulty, but these were selected due to their versatility. In addition, `goto` and `label` were added as calls available in compiled functions, in order to aid with the implementation of the compiler in §3.2. In the following function definition, the `goto` call is used to break out of the While loop.

```python
@gen.compile()
def double_positive_numbers(n : int) -> int:
    i = 0
    acc = 0
    while True:
        if i >= n: goto("exit")
        else:      i = i + 1
        acc = acc + 2
    label("exit")
    return acc

# >>> double_positive_numbers(5)
# 10
```

2.4 Datatypes

Our implementation mostly has support for numeric datatypes, including integers, floats, booleans. This is due to the fact that `llvmlite` was developed for Numba[9], which is a JIT compiler for numerical computations. In addition, we have support for multi-dimensional arrays, as long as they are of a single type and their extents are known at compile time. In order to call these functions, the arrays must be created with Numpy[10]. The Numpy arrays are automatically marshalled by our backend (cast to a pointer type). We support loading from and storing to multi-dimensional arrays.

```python
import numpy as np

array = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]], dtype=np.int32)

@gen.compile(env = globals())
def sum_ndarray(arr : Int[3][3]) -> int:
    acc = 0
    i = 0
    while i < 3:
        j = 0
        while j < 3:
           acc = acc + arr[i, j]
           j = j + 1
        i = i + 1
    return acc

# >>> sum_ndarray(array)
# 45
```

[9] https://numba.pydata.org/
[10] https://numpy.org/

2.5 Calling external functions

We permit calling external functions (such as those from `libc`), provided that a stub of their function signiture is provided (passed to the `compile` decorator). For example, here we use the `putchar` function from the C standard library to print out a matrix of characters.

```
array = np.array([[72, 105, 32], [116, 104, 101], [114, 101, 33]], dtype=np.int32)

def putchar(c : int) -> int: pass

@gen.compile(env = globals(), external = {putchar,})
def sum_ndarray(arr : Int[3][3]) -> None:
    acc = 0
    i = 0
    while i < 3:
        j = 0
        while j < 3:
           putchar(arr[i, j])
           putchar(32) # " "
           j = j + 1
        putchar(10) # "\n"
        i = i + 1
    return

# >>> sum_ndarray(array)
# H i
# t h e
# r e !
```

2.6 Scope extrusion

It may be of some interest whether our implementation permits scope extrusion, such as that described in [Calcagno et al. 2003]. We can implement a similar example:

```
l = 1

def inner():
    global l
    l = q('x + 1')
    return q('2')

@gen.compile(env = globals())
def f(x : int) -> int:
    return splice(inner())

# >>> l
# <ast.Module at 0x7f78aa217a90>
```

Indeed, the quote has escaped[11]. However, we are not able to run `l` directly (unless we manually use Python's `compile` and `eval`), since we can only run compiled functions. We do not permit global functions in compiled functions either, so placing it in a function would fail to compile. However, this will not lead to a crash. At most, we will get a runtime exception (since compilation occurs at runtime), which can be handled.

[11]We could just as easily have simply assigned the quote to `l` directly to achieve the same effect. Python is dynamically typed, so we cannot rely on the type system here.

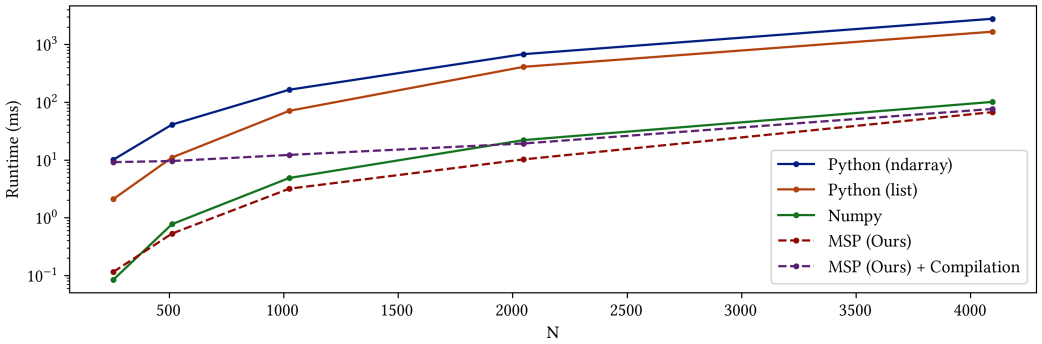

Fig. 1. Transpose running time on $N \times N$, $N \in \{2^8, 2^9, 2^{10}, 2^{11}, 2^{12}\}$ matrix.

3 Results

In this section, we will present some performance comparisons between normal Python functions and compiled functions, as well as to showcase an example of using MSP to compile a simple DSL, demonstrating the added benefit of MSP over plain JIT compilation.

3.1 Performance

It it clear that a compiled function will run faster than an interpreted function. However, since there is compilation overhead, a more detailed look at the performance breakdown is required. We will use a simple matrix transposition function to determine the speed advantage. Performance numbers were collected with the IPython `%timeit` line magic[12].

3.1.1 Matrix Transposition. We perform a transposition of a $N \times N$ matrix of floating point numbers. The functions used are available in Appendix B. Two pure-Python functions are compared. One uses the default `list` datatype, while the other uses an Numpy array. We observe that the naive use of the Numpy array does not improve the performance of the Python function. In addition, we test the use of a Numpy function to transpose the array[13], which achieves a significant speedup. Our implementation achieves an average speedup of 25× over the fastest Python version, and 1.5× over the Numpy implementation. However, compiling the MSP function takes an average of 9.03ms. Therefore, if the function call is used only once (or the function runs for too short a time), Numpy may be the better choice until N becomes sufficiently large.

3.2 DSL Compilation

§3.1 demonstrated that there can be a significant benefit to compiling Python functions to native code. However, we want to further demonstrate the usefulness of the MSP approach. To do so, we write a compiler for the `bf`[14] language, which is a Turing complete programming language resembling the operation of a Turing machine. In `bf`, we have a tape comprised of integer cells and a pointer. All cells initially hold the value zero, and the pointer begins at the leftmost cell. The `>` and `<` commands move the pointer right and left along the tape respectively, while the `+` and `-` commands increment or decrement the value of the current cell. The `.` command prints the value of the current cell in ASCII, and `,` prompts the user for a character and writes the corresponding

[12] https://ipython.readthedocs.io/en/stable/interactive/magics.html
[13] By default, Numpy's transposition creates a view and does not move any data. Therefore, we copy the array to ensure fairness between implementations.
[14] https://esolangs.org/wiki/Brainfuck

ASCII value to the cell. The [and] commands can be used to create loops. We can construct our compiler as follows, with source code that prints a greeting:

```
from msp.msp import *
import numpy as np

gen = CGEngine.create_engine()

source = "++++++++++[>+++++++>" \
         "++++++++++>+++>+<<<<" \
         "-]>++.>+.+++++++..++" \
         "+.>++.<<+++++++++++++" \
         "+++.>.+++.------.---" \
         "-----.>+.>."

def putchar(n : int) -> int: pass
def getchar() -> int: pass

def bf():
    program = []
    jmp = 0
    stack = []
    for command in source:
        if   command == '+': program.append('data[ptr] = data[ptr] + 1')
        elif command == "-": program.append('data[ptr] = data[ptr] - 1')
        elif command == ">": program.append('ptr = ptr + 1')
        elif command == "<": program.append('ptr = ptr - 1')
        elif command == ".": program.append('_ = putchar(data[ptr])')
        elif command == ",": program.append('data[ptr] = getchar()')
        elif command == "[":
            before, after = f"pre_{jmp}", f"post_{jmp}"
            stack.append((before, after))
            jmp += 1
            program.append("\n".join([
                f"label('{before}')",
                f"if data[ptr] == 0: goto('{after}')",
            ]))
        elif command == "]":
            before, after = stack.pop()
            program.append("\n".join([
                f"goto('{before}')",
                f"label('{after}')",
            ]))
    return q("\n".join(program))

    return body

times = []
@gen.compile(env = globals(), debug = False, external = {putchar, getchar})
def run(data : Int[256]) -> None:
    ptr = 0
    splice(bf())
    return

arr = np.zeros(30000, dtype=np.int32)

run(arr)
```

We can see that the `run` function simply splices in its body from the `bf` function, which generates a quote based on the `bf` source code. For each command, the corresponding code is added into the program. Where jumps are required, the `label` and `goto` statements are used. This allows the jumps to be pre-calculated in the resulting IR. While this produces quite long source code, LLVM can optimize it to a certain extent. Upon compilation, the `run` function contains the code required to print "Hello World!"

4 Discussion

The current implementation is still at a prototype stage, with many parts of Python syntax still left unimplemented. Adding additional control flow, datatypes, and other Python features, such as generators and list comprehensions, can be considered future work. However, since there are already several projects that aim to JIT-compile Python, a better approach may be to extend an existing JIT compiler with staging annotations. In addition, many of these implementations focus on numeric data (including our own), so it would be interesting to implement other datatypes such as strings, which would allow replicating Rompf and Amin [2015].

5 Related work

There are several projects that attempt compilation in Python, some of which integrate with CPython (the reference implementation of Python), and some that are substitutes for it. We organize them into several categories, and only describe a representative of each category for brevity.

Python compilers. Cython[15] is a Python-to-C compiler that can interoperate with CPython.
CPython alternatives. PyPy[16] is a faster implementation of Python that includes a tracing JIT-compiler.
Numerical accelerators. Libraries such as Numba[17] can compile and JIT-compile numerical Python at runtime.
JIT compilers. Pyjion[18] is a JIT extension for CPython that compiles to Microsoft's CIL.
DSLs. PyTorch[19], Triton[20] and others implement DSLs that can be compiled to optimized CPU/GPU code for machine learning. Other such DSLs exist for other fields as well, such as Seq[21] for genomics (now Codon [Shajii et al. 2023], a more general framework and compiler for high-performance DSLs).
Multi-stage programming. Scale[22] is an implementation of multi-stage programming, but it is not actively developed and has not been updated in many years.

Our approached aligns with that of Numba, as we take a Python function at runtime and lower it to LLVM IR. This is similar to the strategy employed by Scale. However, our splicing mechanism is different then theirs (we evaluate the function in Python, while they built their interpreter to do so). In addition, our multidimensional arrays are implemented differently. Our implementation is consistent with Numpy arrays, while theirs is implemented like Python lists, requiring expensive copy operations to move data between Python lists and their compiled function.

There is a wide literature describing the use of multi-staged compilation for programming languages, typically with functional languages such as ML [Taha and Sheard 2000], Haskell [Sheard

[15] https://github.com/python/cpython
[16] https://pypy.org/
[17] https://numba.pydata.org/
[18] https://github.com/tonybaloney/Pyjion
[19] https://pytorch.org/
[20] https://github.com/triton-lang/triton
[21] https://seq-lang.org/
[22] https://github.com/es1024/python-staged-programming

and Jones 2002] or Scala [Rompf and Odersky 2010]. In particular, Rompf and Amin [2015] claim that with their library-based Scala approach, an SQL compiler for in-memory data is competitive with a C implementation without needing to compromise on abstraction. They base their implementation on the idea of *Futamura Projections* [Futamura 1999], whereby a an interpreter that can specialize to a particular source code can be considered a compiler. This code specialization (in the form of AST manipulation followed by compilation) at runtime is exactly what we achieve in Python in this project.

6 Conclusion

In this project, we developed a prototype of a Python module that enables both runtime code generation through LLVM IR and multi-staged programming in Python. This allows us to effectively mitigate the the performance penalty of using Python compared to a more low-level language without giving up on any convenience, as well as enables easy creation of compilers for performant DSLs. We demonstrate the effectiveness of our approach by showing a 25× mean speedup over the pure Python version of a transpose function, as well as a 1.5× speedup over the Numpy version. In addition, we show that we can use the MSP functionality to create a compiler for a simple language.

References

C. Calcagno, E. Moggi, and T. Sheard. 2003. Closed types for a safe imperative MetaML. *J. Funct. Program.* 13, 3 (May 2003), 545–571. https://doi.org/10.1017/S0956796802004598

Yoshihiko Futamura. 1999. Partial Evaluation of Computation Process, Revisited. *Higher Order Symbol. Comput.* 12, 4 (Dec. 1999), 377–380. https://doi.org/10.1023/A:1010043619517

Tiark Rompf and Nada Amin. 2015. Functional pearl: a SQL to C compiler in 500 lines of code. *SIGPLAN Not.* 50, 9 (Aug. 2015), 2–9. https://doi.org/10.1145/2858949.2784760

Tiark Rompf and Martin Odersky. 2010. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. In *Proceedings of the Ninth International Conference on Generative Programming and Component Engineering* (Eindhoven, The Netherlands) *(GPCE '10)*. Association for Computing Machinery, New York, NY, USA, 127–136. https://doi.org/10.1145/1868294.1868314

Ariya Shajii, Gabriel Ramirez, Haris Smajlović, Jessica Ray, Bonnie Berger, Saman Amarasinghe, and Ibrahim Numanagić. 2023. Codon: A Compiler for High-Performance Pythonic Applications and DSLs. In *Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction* (Montréal, QC, Canada) *(CC 2023)*. Association for Computing Machinery, New York, NY, USA, 191–202. https://doi.org/10.1145/3578360.3580275

Tim Sheard and Simon Peyton Jones. 2002. Template meta-programming for Haskell. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell* (Pittsburgh, Pennsylvania) *(Haskell '02)*. Association for Computing Machinery, New York, NY, USA, 1–16. https://doi.org/10.1145/581690.581691

Walid Taha. 2004. *A Gentle Introduction to Multi-stage Programming*. Springer Berlin Heidelberg, Berlin, Heidelberg, 30–50. https://doi.org/10.1007/978-3-540-25935-0_3

Walid Taha and Tim Sheard. 2000. MetaML and multi-stage programming with explicit annotations. *Theoretical Computer Science* 248, 1 (2000), 211–242. https://doi.org/10.1016/S0304-3975(00)00053-0 PEPM'97.

A AST & LLVM Translation of add3

AST representation:

```
Module(
  body=[
    FunctionDef(
      name='add3',
      args=arguments(
        posonlyargs=[],
        args=[
          arg(
            arg='n',
            annotation=Name(id='int', ctx=Load()))],
        kwonlyargs=[],
        kw_defaults=[],
        defaults=[]),
      body=[
        Return(
          value=BinOp(
            left=BinOp(
              left=BinOp(
                left=Name(id='n', ctx=Load()),
                op=Add(),
                right=Constant(value=1)),
              op=Add(),
              right=Constant(value=1)),
            op=Add(),
            right=Constant(value=1)))],
      decorator_list=[],
      returns=Name(id='int', ctx=Load()),
      type_params=[])],
  type_ignores=[])
```

Unoptimized LLVM translation:

```
; ModuleID = ""
target triple = "unknown-unknown-unknown"
target datalayout = ""

define i32 @"add3"(i32 %"n")
{
entry:
  %".3" = add i32 %"n", 1
  %".4" = add i32 %".3", 1
  %".5" = add i32 %".4", 1
  ret i32 %".5"
}
```

B Transpose Functions

Python function with `np.ndarray` datatype:

```
%%timeit A, B = np.random.rand(n, n).astype(np.float32), np.zeros((n, n), dtype=np.float32)

# Transpose Python (Numpy array)
def transpose_py1(array, out):
    for i in range(n):
        for j in range(i + 1, n):
            out[i, j] = array[j, i]
            out[j, i] = array[i, j]

transpose_py1(A, B)
```

Python function with `list` datatype:

```
%%timeit A, B = np.random.rand(n, n).astype(np.float32).tolist(), np.zeros((n, n), dtype=np.
    ↪ float32).tolist()

# Transpose Python (Python list)
def transpose_py2(array, out):
    for i in range(n):
        for j in range(i + 1, n):
            out[i][j] = array[j][i]
            out[j][i] = array[i][j]

transpose_py2(A, B)
```

Numpy function:

```
%%timeit A, B = np.random.rand(n, n).astype(np.float32), np.zeros((n, n), dtype=np.float32)

B = A.T.copy()
```

MSP function:

```
@gen.compile(env = globals(), debug = False, add_time = times)
def transpose_msp(array : Float[n][n], out : Float[n][n], n : int) -> None:
    i = 0
    while i < n:
        j = i + 1
        while j < n:
            out[i, j] = array[j, i]
            out[j, i] = array[i, j]
            j = j + 1
        i = i + 1
    return
```

```
%%timeit A, B = np.random.rand(n, n).astype(np.float32), np.zeros((n, n), dtype=np.float32)

transpose_msp(A, B, n)
```