MILLIMAN REPORT

Python and Numba

Techniques for maximizing acceleration

April 2025

James Brackett
Chad Schuster, FRM
Donal McGinley, MSc, FSAI
Corey Grigg, FSA, MAAA, CERA
Kshitij Srivastava, MS

Milliman

Table of contents

THE PYTHON PROGRAMMING LANGUAGE: A BRIEF BACKGROUND **1**

NUMBA: A JIT COMPILER FOR PYTHON **2**

NUMBA PROCESSING OVERVIEW 2

BENEFITS OF NUMBA 3

NUMBA TRADE-OFFS 4

CHALLENGES IN NUMBA ADOPTION **5**

OBJECT-ORIENTED PROGRAMMING (OOP) 5

DEBUGGING 5

ERROR DETECTION AND REPORTING 6

COMPILER OPTIMIZATION 8

COMPILATION TIME 9

EXTENDING NUMBA **11**

EVENTS 11

EXTENSION APIS 12

COMPILER CUSTOMIZATION 13

PRACTICAL USE CASES FOR NUMBA EXTENSIONS **14**

COMPILE-TIME PROFILING 14

notify 14

on_start 14

on_end 15

EXECUTION-TIME PROFILING 15

OOP PARADIGMS 18

JitClasses 18

StructRefs 18

Milliman proof of concept (MPOC) 19

PERFORMANCE ANALYSIS 20

CONCLUSIONS **22**

The rise in popularity of Python means insurance companies, and vendors, have a growing body of actuaries, quantitative developers, and software engineers capable of building innovative and bespoke solutions for both data management and modeling. This comes at a time when many of these same organizations are leveraging cloud services to maintain data and to execute models, and may be challenged to improve efficiency and better manage cloud-related expenses. Additionally, economizing on cloud use can help organizations reduce carbon footprints, thereby aligning with sustainability goals. Accordingly, the runtime performance of such models can have a material impact not just on turnaround time and the quality of analytics but on operating costs and environmental stressors. We believe one potentially powerful tool in the effort to accelerate Python is Numba, a technology for transforming Python code to native machine code. Our experience with Numba has enabled Milliman to deliver increased value to our clients, through both collaborative development and improved service delivery, and to further enhance that value by innovating beyond *out-of-the-box* capabilities.

In this paper, we explore the background of Numba as a just-in-time (JIT) compiler to accelerate the increasingly popular Python programming language, identifying key benefits and trade-offs inherent in its use. A deeper dive into the underlying mechanics of Numba highlights a number of technical challenges, as well as techniques for customizing or extending Numba to navigate those challenges. We proceed with a high-level analysis of out-of-the-box and proprietary solutions for use of basic object-oriented programming (OOP) features in Numba, including Milliman-developed technology with potentially superior performance characteristics. Finally, we conclude with the thesis that extensive knowledge and expertise with the internal machinery of Numba afford opportunities to improve developer efficiency, maximize runtime performance, and minimize cloud-computing costs.

The Python programming language: A brief background

The first version of the Python programming language was released 30 years ago, in 1994, after development commenced in 1989. The intention of its creator, Guido van Rossum, was to prioritize readability, maintainability, simplicity, and consistency. He subscribed to the belief that Python should be easy to learn, free of unnecessary complexity, and fun to use.[1]

Fast forward to present day, and those founding principles have propelled Python to the top of some very compelling rankings. In 2014, it supplanted Java as the primary introductory computer programming language at top U.S. universities;[2] as of 2018, it is the most popular language for which online tutorials are searched, per the Popularity of Programming Language (PYPL) Index;[3] it has occupied the top spot in the TIOBE index of programming languages since 2022;[4] it ranks second behind only perennial front-runner Javascript in a 2024 Stack Overflow developer survey, where it also ranks first excluding respondents who are professional developers;[5] and in 2024, it overtook Javascript as the most popular language on GitHub.[6] While the methodology behind some of these studies may open the door to debate, year-over-year trends unambiguously support the thesis that adoption of Python has skyrocketed over the past 10 years.

The reasons behind such impressive growth in adoption include a relatively straightforward syntax, a feature-rich ecosystem of helpful libraries, expansive online documentation, and developer-friendly tooling—like notebooks that facilitate immediate interaction with code—that can reward beginning and intermediate developers with workable solutions given modest effort.

1. Quotes attributable to the creator of Python, Guido van Rossum, illuminate his motivation and principles, some of which are catalogued at https://www.bookey.app/quote-author/guido-van-rossum.
2. Guo, P. (2014, July 7). Python Is Now the Most Popular Introductory Teaching Language at Top U.S. Universities. Communications of the ACM. Retrieved April 7, 2025, from https://cacm.acm.org/blogcacm/python-is-now-the-most-popular-introductory-teaching-language-at-top-u-s-universities.
3. PYPL Index (2025, April). PYPL Popularity of Programming Language Index. PYPL. Retrieved April 7, 2025, from https://pypl.github.io/PYPL.html.
4. Jansen, P. (2025, April). TIOBE Index for April 2025. TIOBE. Retrieved April 7, 2025, from https://www.tiobe.com/tiobe-index.
5. Stack Overflow. (2024). 2024 Developer Survey. Retrieved April 7, 2025, from https://survey.stackoverflow.co/2024/technology#most-popular-technologies-language-learn. Note that our citation here excludes HTML and CSS rankings, as these are web presentation technologies often paired with Javascript but are not standalone languages for expression of computer logic.
6. GitHub. (2024, November 22). Octoverse 2024. Retrieved April 7, 2025, from https://github.blog/news-insights/octoverse/octoverse-2024.

Like C# or Java, Python offers developers the benefits of OOP design and access to a wealth of existing libraries that, to name only a few, include data analytics, machine learning, artificial intelligence, network communication, web APIs, database integration, and encryption and security. But like VBA or SAS, Python can be coded and run interactively, without the rigor of explicitly invoking compilers and other toolchains to transform human-readable source code into something runnable. This combination of support for modern development methodology and immediate accessibility enables Python developers to build fit-for-purpose solutions rapidly, without necessarily compromising good development practices (e.g., separation of concerns, use of polymorphism, rigorous unit testing, etc.) or re-inventing wheels.

The mainstream distribution of Python, known as conventional Python, or CPython, is implemented as an *interpreter*. This means as a Python program is run, the CPython analyzes the source code and then, following the program flow of control, looks at each Python instruction and executes a series of internal operations to achieve the correct interpreter state. A compiled program, on the other hand, is converted from source code into native machine-level operations *before* the program is run; by the time such a program is executed, the instructions needed to run it are already at machine level and require no additional interpretation. This distinction makes conventional Python (CPython considerably slower than compiled C or C++.

CPython divides the interpretation process into two parts. The first part analyzes Python source code and translates that code into a more convenient representation of the program called bytecode. Bytecode is a more granular expression of logic than, say, a single line of Python code, so it offers a more compact representation of simple operations than the original source code, which might carry programmer comments and other whitespace, and must be validated for correct syntax. Nevertheless, even though the original source code is replaced by more economical bytecode, such code must still be interpreted. The second part of the interpretation process, then, is the reading of bytecode and the translation of those lower-level Python instructions into machine-level operations as the program is run.

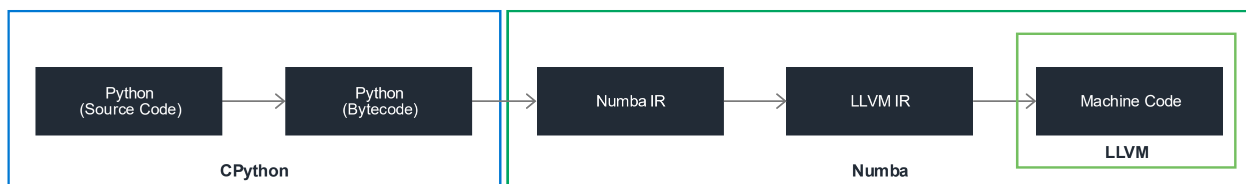
Languages like C# and Java are also compiled like C or C++, but rather than targeting machine-level instructions directly, a similar set of *virtual machine* instructions are used to represent the compiled program in a more portable way. These hardware-independent virtual machine instructions are then translated to hardware-specific machine instructions at runtime, in a process reminiscent of an interpreter; however, because the virtual machine instructions are at much closer parity to the native machine instruction set than the original source code, performance tends to be competitive with C or C++, and thus still much faster than interpreted Python.’

Numba: A JIT compiler for Python

To mitigate the performance penalty inherent in CPython, the Numba open-source project, initiated in 2012 by Travis Oliphant, attempted to apply the process of compilation (like with C or C++) to eligible parts of an otherwise-interpreted Python program. Because the compilation process is performed on demand, as the program is running, it is a flavor of compilation known as just-in-time, or JIT. JIT compilation is a widely employed technology across many programming languages, including Julia, Lua, and Rust. (For more about Julia, refer to the 2024 Milliman publication *An actuary’s guide to Julia*.)[7]

NUMBA PROCESSING OVERVIEW

FIGURE 1: NUMBA PROCESSING FLOW FROM PYTHON SOURCE CODE TO MACHINE CODE



7. See https://www.milliman.com/en/insight/an-actuary-guide-to-julia-use-cases-performance-benchmarking-insurance.

Numba supports JIT on a function-by-function (or, in some cases, class-by-class) basis using Python decorators. A Python decorator is a shorthand way of giving one body of Python code an opportunity to access and potentially modify another. By applying Numba decorators, a Python programmer can signify that select parts of the program should be handled by Numba rather than the default Python interpreter. This makes it relatively straightforward to undertake integration of Numba with an existing Python program.

Once Numba is aware of each function to be JIT-compiled, it first translates the CPython bytecode described in the previous section into an intermediate representation (IR) specific to Numba. This Numba IR is then modified and analyzed to infer the actual data types required for every function parameter and variable. Such inference is required because, while Python is flexible with respect to data types (which CPython resolves as the program is interpreted and run), machine-level compiled code must commit to specific data types in advance. If type inference succeeds, the corresponding *typed* Numba IR is further modified to leverage automatic parallelization and to apply a series of optimizations. At this point, the Numba IR is ready to be compiled to machine-level code.

To perform the final stage of compilation, Numba relies on an independent, open-source project dedicated to the development of tools useful for compilers. This suite of tools, called LLVM, includes a compiler library capable of translating an LLVM-specific IR, known as LLVM IR, into the final hardware-specific machine code that is ultimately executed by the operating system. This compiler library is also capable of applying a host of optimizations intended to boost performance by increasing efficiency without compromising behavior. To invoke the library, Numba must first translate, or *lower*, the typed Numba IR into LLVM IR. Once LLVM has compiled LLVM IR into machine code, Numba can substitute the resulting implementation for the original Python function.

Ideally, subsequent invocations of the same Python function would short-circuit the expense of compiling bytecode to machine code. However, recall that Python functions are flexible with respect to parameter data types, whereas machine-level code is bound to specific data types. To account for the possibility of different types passed to the same function, Numba effectively prepends logic to each JIT function that performs a runtime analysis of the types used in the current call. If this analysis is compatible with a type-specific version of the function that has already been compiled, then short-circuiting is indeed applied; otherwise, Numba must repeat the translation, type inference, and lowering steps described previously to obtain a new type-specific version. Numba support for compilation and reuse of different type-specific implementations of the same Python function is known as polymorphic dispatching.

LLVM, developed in 2000, is a broadly adopted technology. Organizations including AMD, Apple, IBM, Intel, and Nvidia leverage LLVM as the core of their C, C++, Objective-C, CUDA, OpenCL and Fortran compilers. LLVM is also used as the basis for JIT compilation in many programming languages other than Python, including Julia, Lua, Rust, Swift, and Mono. LLVM was originally an initialism for low-level virtual machine, but has distanced itself from the initialism and has evolved to represent the full name of the project. The use of LLVM in Numba is accomplished through a Numba subproject called llvmlite, which exposes a subset of the native LLVM API, implemented in C++, to Python.

BENEFITS OF NUMBA

The most obvious advantage of Numba is runtime performance. Numba-compiled code can run approximately one to two orders of magnitude faster than its interpreted counterpart. (See **Error! Reference source not found.** for some analysis of performance using a reference model developed for this paper.) Even taking into consideration the time required to invoke the Numba and LLVM compiler chain, plus the overhead of dynamic dispatching, this boost in performance is impressive. The value proposition is even more compelling when the original Python code is already compatible with Numba constraints, in which case only the trivial application of a decorator is required. (As noted subsequently, however, this is not always the case.)

A related advantage of Numba is support for compiling code to run on graphics processing units (GPUs). GPUs are specialized devices capable of running thousands of concurrent computational procedures in parallel, opening the door to significantly more throughput per unit time than possible on conventional central processing units (CPUs). CUDA, a technology developed by Nvidia to express GPU algorithms using the same concepts and structures as traditional CPU-based algorithms, is integrated with Numba proper and with the LLVM compiler suite. This integration exposes functions to Python programmers to explicitly move data to and from the device and allows JIT-decorated functions to run (as kernels) on the GPU. Well-designed kernels can accelerate processing by one or two orders of magnitude compared to their CPU counterparts.

Performance boosts attributable to Numba can dramatically reduce turnaround time for Python programs. This has the potential to deliver business value by obtaining results in less elapsed time and potentially freeing up computing resources for other use. Additionally, when contemplating use of Python in the cloud, where resources like CPU time are metered, Numba can also be viewed as a mechanism for reducing cloud expenditure by way of corresponding reductions in resource retention time.

The degree to which Numba can deliver reductions in time and cost is dependent on many factors, including how much Python code qualifies as *jittable*, what proportion of time is required for input/output (I/O), what efficiencies are already realized in the original Python logic, and the extent to which compiler optimizations can accelerate generated code. Some of these considerations are addressed in subsequent sections of this paper.

NUMBA TRADE-OFFS

It was noted previously that, when applicable, the realization of substantial performance gains in exchange for the minimal effort required to apply a decorator is a clear and obvious win. However, this best-case scenario is not always realistic. Numba simply cannot translate *all* possible Python code into machine code; instead, there are limits on what Python constructs are eligible for JIT compilation. These limits mean not all existing code can be accelerated.

To work around such limitations, it may be necessary in some cases to rewrite Python logic to accommodate Numba constraints or, in other cases, to abandon compilation altogether. There are also cases where Python code can be compiled only if experimental features of Numba are leveraged. Experimental features are those that are either not fully developed and tested or whose interfaces and behaviors are not fully decided, and thus subject to potentially radical change. In the former case, the risk of incorrect program behavior is elevated, while in the latter, future iterations of Numba may require significant rewrite or even a retreat to interpreted Python.

Another potential work-around beyond use of experimental Numba features is to develop extensions to Numba that can deliver capabilities not available *out of the box*. Numba, to its credit, contemplates this type of extensibility, and there is at least some official documentation available on how to leverage a number of these extension points. (In fact, most of the extension mechanisms available to Numba users are also leveraged within Numba itself to support its higher-level features.) However, the technical knowledge required to take advantage of Numba extensions is formidable and likely to be accessible to only a small subset of Numba users.

> The thesis that Numba extension development is limited to a small population of Numba users is supported by a search of public GitHub repositories. At the time of writing, results suggest Numba extension interfaces are imported at a rate of less than 2% across Python source files in which *any* Numba import occurs. Fewer than 1% include imports related to LLVM code generation. A cursory review of these results suggests Numba extension references are concentrated in projects specifically intended to integrate existing technologies with Numba, leading to a much smaller proportion of these extensions in the general population than file-count statistics indicate.

The remainder of this paper addresses some examples of challenges in the use of Numba and explores approaches to mitigating those potential impediments.

Challenges in Numba adoption

OBJECT-ORIENTED PROGRAMMING (OOP)

Python is an OOP language, and as such, code is commonly organized accordingly. This enables developers to leverage OOP principles like polymorphism, inheritance, and encapsulation to build solutions that can be easier to read and understand, are less prone to disruption when modified, and can facilitate efficient reuse of code.

Numba is not *entirely* without polymorphism. The types of arguments and variables for a particular Python function are permitted to change with each call to that function, but for each such call, the types must be known by Numba at the time the call is compiled. Numba will compile a unique version of the called function for each combination of applicable data types and then reference that particular version as it generates machine-level code. The relationship between a single CPython function and potentially many type-specific compiled versions of that function is known as polymorphic dispatching. This type of polymorphism is more closely related to the concept of *template* functions in C than to the OOP concept of *inheritance* between C classes. The former depends on the resolution of types at compile time, whereas the latter relies on compiler-generated tables of function pointers (called virtual tables or v-tables) to jump to different type-specific implementations at runtime. The distinction between the two in C++ is quite clear-cut, but because Numba compiles on the fly, the compile-time nature of polymorphic dispatching is still driven by runtime conditions in the Python interpreter. Nevertheless, setting aside *interpreter* runtime and contemplating the difference between compile time and *compiled-code* runtime can be a useful framework for understanding the nuances of Numba.

Unfortunately, Numba does not have seamless support for compiling Python code that makes use of classes. Instead, to preserve *some* class-like behavior, namely, the encapsulation of code and data into a single object, Numba offers an experimental feature called jitclasses. Applying the **jitclass** decorator to a Python class signifies that the associated methods are to be compiled and enables the programmer to declare the names and data types of associated member variables. This requirement to prescribe member and types differs from the traditional Python paradigm whereby attributes can be assigned to an object on-demand (customarily, concentrated in the **__init__** method) and with no constraints imposed on the data types assigned to those attributes.

Methods in jitclasses are subject to the same language limitations as standalone functions to be compiled by Numba. Additionally, Python classes to be compiled via Numba jitclass are forbidden from participating in subclassing, meaning that functionality from one class cannot be inherited in another. This limitation not only inhibits code reuse but also precludes traditional polymorphism, where two or more objects that share a common ancestor can be used interchangeably (and with distinct behavior) any place the ancestor may be used. This follows from the fact that no concept of ancestry can apply when inheritance is unsupported and from the requirement that the type of every variable must be resolved by Numba at compile time.

In addition to the language-related limitations of jitclasses, there is also an architecture-related limitation: jitclasses work (as an experimental feature) when compiling for CPU but are not yet supported when compiling for GPU.

DEBUGGING

Interpreted languages lend themselves to interactive debugging more readily than compiled languages. This is because an interpreter is required to explicitly track flow of control through program source code, as well as the scope and currently assigned value of all variables and arguments. The information needed by a debugger to expose the intermediate program state is already represented within the interpreter as it advances through program instructions.

Conversely, although compiled code is derived from source code, this derivation is typically far removed from the original structure of the program as expressed by the developer. For example, named variables are replaced by pointers to memory addresses or hardware registers, and constructs like function calls or loops may be transformed by the compiler to the point they are no longer identifiable.

However, for a debugging session to be useful to the developer, the debugger must be capable of exposing program state in terms of those original constructs, translating from the hardware domain back into the source code domain. For example, the compiler can export supplemental information as it transforms source code to machine code, and the debugger can then import that additional information for use in translating machine code back to source code. However, even with that supplemental information available, it can still be all but impossible to decipher modifications applied by compiler optimization; thus, either the debugging session lacks complete information (for example, some breakpoints cannot be applied to some lines of source code) or two versions of compiled output must be generated and maintained (one optimized for normal execution and one devoid of optimization to accommodate debugging).

Numba includes support for emitting supplemental information in a format that can be consumed by the GNU Project Debugger (GDB). The GNU Project is the open-source initiative under which the Linux operating system is developed, along with a myriad of related tools and programs, including the GNU Compiler Collection (GCC) and its companion debugger, GDB. While GCC and GDB are common to many compiled programming languages supported by the GNU Project, most prominent of which are C and C++, Python is not one of them; instead, interpreted Python is normally debugged using PDB, the Python Debugger. This means that two different debugging solutions are applicable when using Python and Numba: PDB for interpreted code and GDB for compiled code.

Use of GDB forfeits the Python-friendly features of PDB and instead requires the programmer to operate at a lower level of abstraction, closer to the machine-level code generated by Numba than the original Python source code. Although the use of supplemental debug information emitted by Numba facilitates some conveniences with respect to Python function and variable names, such as the ability to set breakpoints on named functions or to inspect values assigned to primitive Python variables, more complex data structures, like lists, are not trivial to inspect, and default optimization levels at compile time can undermine the integrity of the debug information. Additionally, the skill required to operate GDB is considerable, generally requiring command-line interaction and familiarity with concepts like stack frames and memory layout, concerns that would not apply to the use of PDB with interpreted Python.

Fortunately, in many cases, the technicalities of machine-level abstractions and use of GDB can be avoided. This is because the behavior of Numba-compiled code is usually faithful to the behavior of the Python source code to which it is applied. Accordingly, a developer can follow the dichotomy described previously of using separate execution and debugging programs by temporarily disabling JIT compilation, working with pure Python code during the debugging cycle (using PDB and possibly a front-end, like Microsoft Visual Studio Code), then restoring JIT compilation after the debugging exercise is complete. This enables GDB to be reserved for more esoteric cases, like debugging Numba proper or the by-product of Numba extensions (like intrinsic functions, described subsequently).

ERROR DETECTION AND REPORTING

To its credit, Numba is engineered to report any compilation problems it encounters with as much detail as possible, including reference to the applicable Python source code. Often, this error-reporting mechanism is both informative and accurate and is essential to quickly correcting source-level problems. However, there are cases where Numba error reporting falls short of the apparent intention, leading to messages that are obscure and unhelpful or, worse, to a red herring.

One area where compile-time errors can emerge without the benefit of helpful error messages is type inference. Recall that Numba must resolve every Python variable, function argument, and return value to a concrete data type when generating code for the embedded LLVM compiler. Because interpreted Python does not require the programmer to express this type information the same way a language like C, C++, Java, or C# would, the work required to achieve resolution can be quite extensive as type information is propagated and validated throughout potentially thousands of variables. If Numba encounters a particular case where type information is either unresolved or inconsistent, then compilation fails, and a corresponding error is emitted.

A simple example of a type-inference error is given in Figure 2.

FIGURE 2: EXAMPLE OF A TYPE-INFERENCE ERROR

```
from numba import njit

@njit
def foo(n):
    return 0.0 if n % 2 == 0 else 1

@njit
def bar(n):
    return (n, -n)[foo(n)]

print(bar(7))
```

In this contrived example, the return value from **foo** is either **0.0** or **1**, the former of which is a floating-point constant and the latter an integer. The actual return value is determined by whether the input argument **n** is even or odd. The calling function **bar** uses the return value to select between two elements of a tuple, the first of which is the input argument **n** and the second of which is its negative. Finally, the print statement calls **bar** with a value of **7**.

Interpreted Python would evaluate **7** in the call to **foo** as odd, return integer **1**, and compel **bar** to select the second (counting from zero) element of the local tuple, **-7**, for display via **print**.

With the Numba decorator **njit** applied to both **foo** and **bar**, however, an attempt to run the code in Figure 2 generates an error, shown in Figure 3.

FIGURE 3: EXAMPLE OF A TYPING ERROR AS A CONSEQUENCE OF TYPE INFERENCE

```
Traceback (most recent call last):
  File "numba_error_tests.py", line 11, in <module>
    print(bar(7))
  File "lib\site-packages\numba\core\dispatcher.py", line 468, in _compile_for_args
    error_rewrite(e, 'typing')
  File "lib\site-packages\numba\core\dispatcher.py", line 409, in error_rewrite
    raise e.with_traceback(None)
numba.core.errors.TypingError: Failed in nopython mode pipeline (step: nopython frontend)
No implementation of function Function(<built-in function getitem>) found for signature:

 >>> getitem(UniTuple(int64 x 2), float64)

There are 22 candidate implementations:
  - Of which 22 did not match due to:
  Overload of function 'getitem': File: <numerous>: Line N/A.
    With argument(s): '(UniTuple(int64 x 2), float64)':
   No match.

During: typing of intrinsic-call at numba_error_tests.py (9)

File "numba_error_tests.py", line 9:
def bar(n):
    return (n, -n)[foo(n)]
    ^
```

This typing error is a consequence of type inference. Specifically, Numba determines at compile time that the return type of **foo** should be floating point, as this accommodates both possible return values of zero and one by promoting the integer **1** to **1.0**. This determination is made *before* the specific value of **n** is known, unlike in interpreted Python, where no constraints on the return type of **foo** are imposed before it is called.

After Numba commits to a return type for **foo**, the tuple indexing logic in **bar** is forced to work with a floating-point index value. However, floating-point indices are not supported. At this point, Numba generates a typing error to reflect this.

Note that interpreted Python is also incapable of indexing a tuple by floating-point value. Calling **bar** with a value of **8**, for example, instead of **7**, induces the interpreter to select **0.0** as the return value from **foo** at runtime. In this case, Python raises a **TypeError** exception from **bar** to the effect of “tuple indices must be integers or slices, not float.”

In this very simple example, the contrived error is easily rectified by replacing the constant **0.0** with **0** in **foo**. (In fact, the entire dependency on **bar** is readily removed by indexing with the expression **n % 2** in **foo**, which resolves to integer **0** or **1** without an if-expression or use of any constants.)

However, the example in Figure 3 does illustrate that the *reach* of type inference crosses function boundaries; in the case of a more realistic code base, this could span dozens of nested function calls, rather than just the one illustrated in Figure 3. This example, while exceedingly simple, also highlights the relative complexity of Numba error messages: the message in Figure 3 includes a stack trace into Numba libraries rather than the program source code and makes reference to *built-ins*, **getitem**, *signatures*, **UniTuple**, *candidates*, and *intrinsic-call*. Understandably, developers unfamiliar with these terms may be challenged to draw meaningful conclusions about what the message actually conveys and, more importantly, in how to pursue a fix. In contrast, the fix for “tuple indices must be integers or slices, not float” error is easily conceived.

Another source of potentially unhelpful or misleading error messages is the *lowering* phase of compilation, during which Numba translates its intermediate representation (IR) of program code to that required by LLVM. In theory, any problems with source code or type inference should be resolved before lowering, but in practice, this is not always the case, particularly for code that is structurally complex or that leverages extensions to Numba that can effectively bypass the earlier guardrails. Although Numba maintains sufficient context to associate lowering errors with source code, the very nature of a lowering failure suggests a bug in Numba proper, defects in Numba-compatible libraries (like Numpy), or possible misuse of Numba extensions in the immediate code base.

In all such cases, the association with Python source code may be a helpful *hint* but is unlikely to be the *origin* of any root cause. Unless Numba extensions are in scope, a lowering error may not be fixable in code at-hand, leading instead to a frustrating cycle of internet searches and trial-and-error changes to Python code in the hope an alternate implementation can be found to skirt around, rather than rectify, the root cause.

COMPILER OPTIMIZATION

As noted previously, Numba compilation operates first at a Python-specific level, advancing Numba IR step by step, and then, after lowering to machine-specific level, engages the LLVM compiler to convert LLVM IR to optimized hardware instructions. This pipeline includes numerous touchpoints, some of which apply a similar concept at the two levels of representation (namely, Python versus machine).

An example of this duality is the process of function inlining, whereby a compiler might decide that, rather than preserving a call from one function to another, it will be more efficient to duplicate the code of the called function directly inside the calling function. Numba applies function inlining both at the Python level (where the intermediate representation is Numba IR) and at the machine level (where the intermediate representation is LLVM IR). The former is governed by options in Python source code expressed as decorator parameters, while the latter is performed entirely at the discretion of the LLVM compiler (based on a single high-level optimization setting, for which the maximum value of three is the default). Naturally, if Numba IR inlining is triggered for a particular function call, LLVM inlining will not apply, since the caller/callee relationship is already lost by the time LLVM IR processing commences.

Taken as a whole, transformations applied by both Numba and LLVM optimizers can lead to code that is more compact and/or faster than a naive, non-optimized translation. However, as noted in Debugging, these optimizations can also lead to morphological differences between source and compiled code. Moreover, small changes in that source code can, by virtue of complexities inherent in optimization, realize changes in final machine code that are effectively unpredictable. These differences in compiled code do *not* lead to different computational outcomes, as this would violate the principle of optimization in the first place, but *can* lead to variations in performance.

This opacity can hamper reasoning about the performance impact of source code changes, even to the point where modifications intended to reduce computational work actually increase runtime. This is presumably because the sequence of optimizations applied to the *heavier* workload achieves greater net efficiency than the sequence applied to the *lighter*.

In addition to the representation of machine-level instructions after *compile-time* optimization, performance may also be influenced by the order and conditions under which flow-of-control instructions are executed at *runtime*. This is because of the way in which CPUs endeavor to accelerate performance by developing *branch prediction* rules designed to minimize cycles for the more-frequently observed outcomes. (GPUs, on the other hand, do not employ branch prediction because the underlying architecture effectively anticipates that the cycle-cost of *both* branches will be incurred.)

The primary objectives of inlining at the Numba IR and LLVM IR levels differ. The former is intended to facilitate type inference, where types inferred for variables in the calling function can be used to disambiguate types for inlined variables that would otherwise be unresolvable. The latter is intended to accelerate runtime performance by reducing the overhead of a true function call, which typically involves copying argument values for use in the called function, copying back return values from the called function to the caller, and allocating and releasing memory for variables local to the callee. However, there are cases when inlining Numba IR yields better performance than deferring the possible inlining of LLVM IR.

Numba exposes functionality to programmatically access both the LLVM instructions Numba prepares during lowering and the ultimate machine-level output (in assembly language format). It also supports use of several environment variables that can trigger the printing of Numba and LLVM IRs as the compiler operates on source code. These facilities open the door to skilled analysis of the intermediate and final code. However, such analysis requires extensive subject matter expertise in Numba internals, LLVM, and/or assembly language and, as such, is of limited value to most Python developers. And even with such expertise, the volume of information to analyze can be quite formidable even for a relatively small Python code base, and for some transformations (like LLVM optimization) full reverse-engineering is impracticable. Detailed reasoning about the relationship between Python code and Numba-compiled machine code can, even under the best of circumstances, be tedious, time-consuming, and limited to the purview of those with compiler and hardware-level expertise.

Fortunately, the demands for this level of reasoning are likely limited to highly refined performance-tuning or simple curiosity; for the vast majority of Numba users, *fire-and-forget* application of JIT decorators is probably sufficient to achieve desirable outcomes.

COMPILATION TIME

JIT compilation, by its very nature, incurs the cost of compilation at the time an interpreted function is called. Accordingly, the time to translate source code into machine code, including multiple passes at IR transformation and aggressive optimization, is added to the execution time for the compiled function on the first call. This cost introduces a drag against the performance benefits of executing machine-level code instead of interpreting source code (or, more accurately, bytecode). However, for each distinct combination of data types for input arguments and return values, Numba pays the price only once; subsequent invocations of a jitted function after the first call bypass the expense of compilation and reuse already-generated machine code immediately.

Note that reuse is predicated on matching already-compiled versions of a function to the data types inferred for the call, since different data types require execution of different machine code. When calling a jitted function from interpreted Python, this means Numba must still perform type inference (described earlier in Numba Processing Overview) in order to determine whether an already-compiled version is available or not. The time to perform type inference alone is necessarily faster than a full compilation but can still represent a sizeable proportion of that time. Accordingly, every call from interpreted Python to compiled Python incurs a performance penalty, as the prevailing data types must be resolved, even when the corresponding type-specific function has already been compiled. (Every call from compiled Python to compiled Python, conversely, incurs the cost of type inference for the callee only one time for each type-distinct caller.)

Except in the most extreme edge cases, the performance benefit of running compiled code can be expected to significantly outweigh the costs associated with type inference and one-time compilation, particularly for jitted functions called in loops or from multiple sites.

One additional factor that can influence the trade-off between the time to compile versus accelerated execution is *inlining*. (See Compiler Optimization for more on inlining.) When one jitted function is called from another jitted function, and Numba inlining is enabled, the code (in Numba IR form) of the called function is essentially inserted directly into that of the calling function. This has the effect of increasing the size and complexity of the calling function, which in turn can amplify the time and resources required to compile it. Inlining also precludes reuse of already-compiled functions, since Numba inlining operates at a higher level than machine code, so called functions are effectively recompiled for each instance of an inlined call.

Numba inlining is intended[8] to help with type inference more than runtime performance (whereas LLVM inlining is, conversely, intended to help exclusively with runtime performance). However, there are cases where aggressive Numba inlining *does* indeed yield better runtime performance. Pursuit of performance improvements by way of Numba inlining can then inflate compilation time because called functions are subject to recompilation for every instance in which they are inlined. For code bases with many levels of nested function calls, a single top-level function call will lead to recompilation of *all* the functions called directly or indirectly from that top-level call.

The impact of compilation time is arguably most significant when it materially extends development time through longer coding and testing cycles; this slows development progress and incurs opportunity costs for skilled (and presumably limited) human resources. To strike a balance between accelerated development and accelerated runtime, it may be helpful to organize Python source code such that JIT functionality can be conveniently disabled while actively changing code. (This technique is also potentially useful in the context of debugging, as noted in Debugging.) Similarly, if compile times during development are acceptable with inlining disabled but excessive with inlining enabled, then JIT decorators can be modified to turn inlining on or off. For even more granular control, Python *cost functions* can be implemented to decide *programmatically* whether to inline; such functions can take into consideration some indicator as to whether compilation is running under a development context or not. Use of cost functions, however, requires significant knowledge about Numba internals, including how functions are represented using Numba IR.

If compile time is problematic more universally—that is, not just within development iterations—Numba currently offers an alternative to JIT, namely, ahead-of-time (AOT) compilation. AOT compilation is similar to how languages like C or C++ work, where translation from source code to machine code occurs *before* execution, and the outputs of compilation are saved to disk; these machine-level artifacts are then restored from disk at execution time, without any runtime performance penalty for compilation. Numba AOT applies the Numba compilation process to designated functions and then saves compiled versions as a Python extension module; this module can then be imported into future instances of the Python interpreter, potentially on different systems, to facilitate access to precompiled code.

8. See https://numba.readthedocs.io/en/stable/developer/inlining.html.

AOT compilation has the advantage that compile time is removed entirely from runtime, and it is possible for Python processes to benefit from Numba without requiring Numba to be installed on the runtime system. However, because type inference is not possible when code is compiled *a priori*, developers are obliged to decorate compiled functions with all required type information explicitly. Additionally, because compiled code may be run on a system different from that used to perform compilation, AOT compilation is compelled to generate more generic code than JIT, which—with knowledge that code will run where it is compiled—can optimize to specific hardware. This means that AOT-compiled Python code may not attain the same performance as JIT-compiled versions of the same code.

As of Numba version 0.57.0 (June 2023), the implementation of AOT functionality described previously is scheduled for deprecation. It will not, however, be removed until such time as an alternative implementation is made available (and until both implementations have coexisted for at least two Numba releases). This means work may be required to maintain AOT functionality as the details change, but some AOT capability is certain to be available for any foreseeable release.

Extending Numba

As noted previously, use of Numba can introduce a number of challenges in exchange for accelerated performance, at least some of which can be mitigated in part or in whole by work-arounds. Such work-arounds might include forgoing some features of the Python language, accepting risks associated with experimental features, or disabling JIT compilation (or specific optimizations, like inlining) on a temporary basis.

However, Numba offers another way to address challenges besides seeking work-arounds: customizing the way in which Numba itself works. This is accomplished by developing code that modifies Numba behavior by way of *extension points*. Implementing Numba extensions generally requires advanced knowledge of Numba internals, and not every challenge can be addressed via customization, so extensions are not a panacea. But, as demonstrated in the sections that follow, they can be a very useful resource in the developer toolbox.

The sections that follow describe some of the mechanisms Numba exposes to facilitate customization and identify how those mechanisms might be leveraged.

EVENTS

Numba exposes an *Event API* that enables user-supplied Python code to receive notification as the compilation process progresses. While use of this API doesn't empower user code to modify compilation—it is essentially a *read-only* mechanism—it can be useful for gaining insight into what steps are taken and, importantly, how long those steps take.

Consider the case of Python code calling into jitted functions. As described previously, Numba will undertake compilation on the first call to a jitted function (for a particular arrangement of argument and return types), then bypass compilation on subsequent calls (for the same argument and return types). By wrapping each such call with a timer, Python code can potentially calculate how long the called function requires to be compiled; subtracting the elapsed time for any call *after* the first from the elapsed time *of* the first should represent the *extra* time allocated to compilation alone.

There are, however, several deficiencies with this technique. First, it requires at least two calls into the function of interest, with distinct treatment for the first and subsequent calls. This means that the Python code base must be organized to isolate calls and that those calls be predictably ordered. Because this is not always trivial, a general approach is to insert an *extra* call early in the flow of control, specifically to isolate the first-call time. If the second call is not easily isolated, then this same approach may be applied again, inserting a *second* extra call to isolate the execution-only time. The result is a code base that incurs the expense of one or two function calls just to compute compilation time.

Secondly, wrapping calls in timers, and potentially isolating those calls with otherwise-unnecessary invocations, does not scale well. If multiple top-level functions are of interest, individual treatment and isolation can be unwieldy.

Furthermore, any approach that presupposes function-call hierarchy, or assumes *isolation calls* are free of side effects, is inherently fragile. Developers who make changes in the jitted code base must remember to revisit compilation-timers to either verify that isolation calls are properly ordered and do not change program behavior or, conversely, update those calls to restore proper timing logic.

Additional complexities with timer-wrapped calls include the possibility of type changes across calls (since *first-time* actually applies per function for specific input and return data types), variations in execution time between first and subsequent calls *not* attributable to compilation, and limits in granularity with respect to compilation steps (since only *overall* compilation time is inferred).

A more robust technique than introducing timers and isolating top-level function calls is to capitalize on the Numba event architecture. Under this paradigm, a Python *listener* object is registered with Numba early (before any jitted function calls), after which Numba calls methods on this object each time it initiates or finishes compilation (or a constituent compilation pass). By maintaining timer state inside the object, and by interrogating parameters passed by Numba to track compilation progress, compilation time can be calculated without isolation of calls, without *a priori* knowledge of call-order, without additional runtime allocated to *extra* calls, and without code maintenance obligations elsewhere (besides the class implementation and one-line registration). Additionally, because Numba events differentiate between different compilation steps, the event listener is empowered to attribute elapsed time in several different ways: it can track by individual function or by the program in totality, by individual compiler pass or compilation overall, or both.

Numba exposes yet another pathway to obtain compiler timing information. When a function is compiled, Numba internally measures the time allocated to each compiler pass and then saves those results as metadata attributes on the corresponding dispatch object. This approach obviates the developer from *any* code changes to activate timing (including even the one-line registration required for event listening) and, like the event-listening technique, avoids the overhead of isolation calls, eliminates risk of fragility, and operates at the granularity of individual functions and compiler passes. One differentiator, however, is *retrieval* of timing metrics. Using events, it is possible to accumulate information on the population of jitted functions as they are compiled and then report on those functions or derive aggregated timings in one place (i.e., the listener object). Use of function metadata, however, requires additional work to catalogue the functions actually compiled, such as using a hybrid approach (i.e., listening to events in order to inventory the population of jitted functions), maintaining an out-of-band list of contributing functions manually, or wrapping or modifying function decorators to harvest a record of compiled functions on the fly. Only after such a catalogue is available can reporting code then interrogate per-function metadata to generate detailed or aggregate timings.

EXTENSION APIS

Numba empowers developers to introduce functions and types not possible strictly through the JIT compilation of decorated Python functions. Python APIs to access these capabilities are exposed by a Numba extensions module.

One example of extending Numba is support for *overloading*, where one of multiple implementations of a function is selected by the compiler based on the types of arguments passed to that function. This behavior is common to many programming languages, including C++, Java, and C#, but is not generally applicable in languages like Python, where argument types are resolved by the interpreter only when needed and not at function call boundaries. However, since jitted functions in Numba require inference of concrete types for every variable, argument, and return value, overloading of functions in this context becomes possible.

A developer implements Numba function overloading by registering a custom Python function with Numba under the name of the function to be overloaded. During compilation, Numba then calls this registered function when it encounters a call to the overloaded function, passing information about the specific types used in that call. The developer-supplied function can inspect that type information to select a suitable implementation for the overloaded function or decline to select an implementation; in the latter case, Numba will continue to search for an implementation, possibly interrogating other registered functions for the same overload.

Numba overloading may also be used to implement *methods*, rather than just standalone *functions*, for types about which Numba already has knowledge. Similarly, developer-supplied functions may be registered with Numba to implement read-only attributes on such types.

The extensions described previously can be implemented with minimal working knowledge of Numba internals, using only Python code to implement both the type-checking and implementation-selection logic called by Numba at compile time, as well as the body of the function to be compiled and called at runtime. However, with additional knowledge of LLVM, including the instruction set used to express LLVM IR, a developer can also implement functions at the virtual-machine level directly, rather than starting with Python source code and relying on the Numba compiler to obtain LLVM IR. Such functions, callable from high-level source code but implemented using machine-level instructions, are called intrinsic functions (or just intrinsics). The concept of intrinsic functions is also leveraged in many C and C++ compilers, including those maintained by Microsoft, Intel, and the GNU Project, as well as in the Java HotSpot JIT compiler.

To implement an intrinsic function in Numba, a developer registers a custom function associated with the intrinsic function name. This function—as was the case for overloading, described previously—is responsible for supplying Numba with an implementation for each call encountered during compilation. However, unlike the overloading mechanism, the implementation is expressed as a sequence of LLVM instructions rather than a CPython function to be compiled. The registered function is implemented in Python, but its output is LLVM IR.

The technique of implementing Python code to generate LLVM IR can also be leveraged to introduce entirely new object types accessible to jitted code. This is accomplished by extending both the typing system, of which type inference is a part, and the lowering process, in which LLVM IR is generated. Numba exposes APIs that enable developer code to influence typing by registering new types, associating those types with functions (including object initializers), and mapping those types to one or more Python datatypes for purposes of type inference. Functions and read-only attributes related to custom types can then be implemented via developer-registered lowering functions responsible for generating LLVM IR directly.

Additional extension points accommodate custom type-casting (i.e., automatic conversion from one Numba datatype to another), boxing and unboxing (converting custom types to Python interpreter objects and vice versa), and assigning constant values to custom types.

COMPILER CUSTOMIZATION

The extension APIs described previously are useful for introducing new types and functions by providing pathways for developers to contribute machine code beyond decorating Python source code for compilation. These pathways, however, operate independently of the compilation steps undertaken by Numba when processing JIT-decorated functions. To facilitate customization of the latter, Numba also empowers developers to override or suppress default compiler behavior or inject entirely new compiler logic.

Overall compiler behavior is defined by its pipeline, a prescribed sequence of passes that operate successively on Numba IR to effect a transformation from the initial source code translation to the virtual machine instruction set. Each pass in the pipeline is configured to either *inspect* Numba IR (for analysis), *mutate* Numba IR (to modify implementation of the compiled function), or *lower* Numba IR (to translate to LLVM IR).

Numba offers developers a more streamlined approach to modify Numba IR than full-blown customization as described previously. Rather than subclassing the exposed compiler class, implementing a new compiler pass from scratch, inserting that pass into the pipeline for the alternate compiler, then explicitly selecting this alternate compiler at all JIT-decorator sites, a developer can instead *hook* code into a compiler pass already included in the default compiler pipeline. This *rewrite* pass is dedicated exclusively to interacting with developer code to analyze and potentially modify Numba IR. Specifically, a developer first derives from a Numba **Rewrite** class, implementing just two methods: **match** to examine Numba IR and decide if a modification is appropriate, and **apply** to actually effect that modification. Next, this class is registered with the Numba rewrite system; once registered, it will be consulted by the built-in rewrite pass during default pipeline execution.

Numba exposes a Python compiler class whose pipeline represents default Numba behavior. A developer may extend this base class to customize the pipeline it implements, thereby defining an alternate compiler. Such customization is typically achieved by adding, removing, or replacing passes expressed by the default pipeline. This allows a developer to *borrow* from the existing system and concentrate effort on just those behaviors that represent a departure from the default.

With one or more alternate compiler implementations available, a developer can then select which of those implementations to apply on a function-by-function (or, in the case of experimental jitclasses, class-by-class) basis.

Knowledge of both Numba internals and general compiler design is a prerequisite for customizing the Numba compiler. Knowledge of LLVM may be of value, too, particularly if implementing lowering compiler passes.

Practical use cases for Numba extensions

In the sections that follow, we describe several use cases for developing Numba extensions to mitigate some of the challenges identified previously (see Challenges in Numba Adoption).

COMPILE-TIME PROFILING

As noted in our exposition on Compilation Time, techniques to assess Numba compilation time include insertion of timer code around first-time and subsequent-time calls, use of the Numba Events API to track compiler progress, and interrogation of Numba-created metadata on individual compiled functions. The first approach is generally impractical, based on code maintenance requirements, accuracy issues, and assumption sensitivity; the third is seamless, accurate and robust, but—if a report on compile time across all functions is desired—insufficient.

To implement an automatic report on compile time per function and in-aggregate, we pursued the second technique, namely, use of the Numba Event system.

Integration with the Numba Event system begins with subclassing the **Listener** class exported from **numba.core.event**, shown in Figure 4.

FIGURE 4: SUBCLASSING THE LISTENER CLASS EXPORTED FROM NUMBA.CORE.EVENT

```
from numba.core.event import Listener

class ProfilerListener(Listener):
    def notify(self, event):
        super().notify(event)

    def on_start(self, event):
        pass

    def on_end(self, event):
        pass
```

The **Listener** subclass implements three related methods, all of which accept a single **event** argument in addition to **self**.

notify

This method is the primary entry point for Numba to propagate information about compilation events to developer-supplied code. The default implementation inspects the given event and passes it in turn to either **on_start** or **on_end** depending on which endpoint of the event lifetime is reported.

on_start

This abstract method is called by the base-class implementation of **notify** when the propagated event is at the start of its lifetime.

on_end

This abstract method is called by the base-class implementation of **notify** when the propagated event is at the end of its lifetime.

Each of these methods is also passed an argument with information about the function being compiled and, as appropriate, about the active compiler pass. We leveraged this interface to implement a **Listener** subclass that collaborates with helper classes to maintain state across all encountered functions and compiler passes, including snapshots of the high-resolution system clock at the start and end of each reported event.

The **Listener** subclass is registered with Numba before any JIT functions are compiled, using **numba.core.event.register**. At the end of the program, the helper classes responsible for maintaining state are queried to obtain a **DataFrame** into which the accumulated compiler metrics are exported. This query normalizes some of the collected data and also makes some inferences required to more accurately reflect inlining. This is because not all compiler passes are applied by Numba when a called function is inlined, including the top-level *compile* event, so some per-function *gaps* in data require special treatment to derive a comprehensive picture.

An example of the resulting **DataFrame** for a simple program with two jitted functions, **foo** and **bar**, is shown in Figure 5.

FIGURE 5: EXAMPLE OF DATAFRAME FOR A SIMPLE PROGRAM WITH TWO JITTED FUNCTIONS

	compile_count	pipeline_count	pass_time	compile_time
bar()	1	3	0.0357089	0.2223350
foo()	1	1	0.0435650	0.0495138
Total	2	4	0.0792739	0.2718490

EXECUTION-TIME PROFILING

Numba includes support for generating debug symbols compatible with the GNU Compiler Collection (GCC). These symbols can in turn be used with open-source project Profila,[9] which, at the time of writing, is in active development. Profila is a *sampling* profiler, meaning that it relies on period examinations of where the Python program is running at that point in time, and it uses these observations to develop a view as to which lines of compiled source code are consuming the most time. This approach minimizes the performance drag on running code, but, because results are obtained by sampling, they can also be imprecise. Profila is currently limited to single-threaded Numba code and is directly useable only on Linux.

An alternate approach to profiling is instrumentation. An *instrumenting* profiler modifies the program of interest at compile time, inserting code around each function body to record time spent there. (Because the overhead of code injection on a line-by-line basis would, in the general case, result in a significantly bloated output with more time invested in collecting timing data than actually executing original program logic, instrumentation is normally applied at the granularity of called functions.) This technique offers more accurate results than a sampling profiler but can introduce significant delays in execution time and runs a higher risk of unintentionally altering program behavior.

We explored development of an instrumenting profiler for Numba, using a customized compiler (as described in Compiler Customization). The strategy behind this proof of concept was to modify Numba IR to seamlessly insert logic for managing timer state per function, *activating* a timer when flow of control enters the corresponding function, and *deactivating* that timer when flow of control either exits the corresponding function or enters another. While simple in concept, pursuit of this strategy highlighted a number of challenges in practice.

First, identifying all cases for timer deactivation is non-trivial. This is in part because the compiler may generate multiple return instructions to exit a function based on the possibility of multiple pathways through the corresponding code, so all qualifying return instructions must be identified to catalog and modify all exit points.

A related challenge is the case where the function to be instrumented issues a call to another function, the lifetime of which *might* represent time *outside* the function proper. The decision whether to attribute this time to the caller or the

9. See https://pythonspeed.com/articles/numba-profiling/

callee is likely a question of whether the called function belongs to the same code base or represents a built-in function or an external library. Accordingly, the profiler must resolve this question at compile time in order to temporarily deactivate and then reactivate the calling-function timer for the former case or to bypass timer state changes altogether for the latter.

An alternative to temporarily suspending a function timer around a call to a function in the same code base is to instead suspend the timer of the caller inside the callee. This simplifies the need to discriminate sibling functions from built-in or external functions, since only sibling functions would include instrumentation code to suspend the timer of the caller. However, this also requires calling code to propagate information about its identity to the called function to enable that function to deactivate and reactivate the correct timer. Runtime propagation of this information proved to be more complex and potentially disruptive than a compile-time analysis of the call target.

Secondly, Python code that is re-entrant, or that Numba can automatically parallelize across multiple threads, complexifies per-function timer state. Minimal timer state can be expressed as the time at which the timer was last activated and a running total of time accumulated to date. In the simplest of cases, this is sufficient to support timer activation and deactivation; for the former, last activation time is updated to current time, and for the latter, last activation time is subtracted from the current time and that difference is added to accumulated time.

However, if a function can call itself, either directly or indirectly, then minimal state must be expanded to track last-activation times for each level of nesting. Similarly, if a function can be called from multiple threads such that concurrent lifetimes overlap, then last-activation times must be tracked on a per-thread basis. Support for multiple threads also requires that updates to accumulated time be atomic—that is, incapable of overlapping—to ensure race conditions do not potentially corrupt the running total.

The structures needed to maintain minimal timer state are dependent only on the total number of distinct functions and can be resolved at compile time. However, structures needed to track nesting and per-thread state must either be extensible at runtime, as the level of nesting and number of concurrent threads necessitate dynamic growth, or be defined at compile time with fixed capacity limits. The former can degrade performance by allocating memory on the fly, while the latter is potentially wasteful with respect to memory utilization. And both can fail at runtime: the former can experience a failure to allocate memory, while the latter may exceed fixed limits.

A third impediment to instrumentation is the risk that compiler optimizations applied *after* code injection subvert the intended behavior of that inserted code. While in theory, downstream optimization should not invalidate the sequence of Numba IR instructions inserted by the customized compiler, our research suggested reordering of code during optimization was sufficient in some cases to break timer activation and deactivation.

Recognizing these challenges, we set out to implement a proof-of-concept instrumenting profiler, conceding the following limitations:

- Re-entrant code is unsupported.
- Multi-threaded code is unsupported.
- Anomalies introduced by compiler optimization are tolerated.

The high-level algorithm for implementation is as follows:

1. A profiler activation function is called by Python source code early in process execution, before any JIT compilation happens. This activation can be accomplished simply by importing a dedicated profiler Python module whose *init*.py automatically calls the suitable function.
2. The profiler activation function is responsible for initializing a profiler-owned registry of jitted functions and for inserting a new *instrumenting* compiler pass into the default compiler pipeline.[10]
3. Each time the instrumenting compiler pass is applied to a function, a check is made if the function is part of the Numba or profiler package, and if so, no further action is taken. This ensures Numba itself is not subject to profiling, thereby circumventing circularity issues with the profiler trying to profile itself. Otherwise, the registry is consulted (and, as needed, updated) to obtain a unique ID for the compiled function, as well as a dedicated data structure in which counters and timer state can be maintained.
4. The start of the Numba IR for the compiled function is modified by the instrumentation pass to issue calls to profiler-implemented functions for incrementing the call-count and for activating both a global timer and a local timer. The global timer tracks time spent in the function, *including* calls to other functions; conversely, the local counter tracks time spent in the function, *excluding* calls to other functions. The profiler-implemented functions are called with the unique ID of the function as an argument.
5. The Numba IR for the compiled function is next searched for all *return* instructions, each of which represents an exit point from the function. The instrumentation pass inserts code immediately before each *return* to deactivate both the global and local timers. This inserted code issues more calls to profiler-implemented functions, passing the unique function ID as an argument.
6. The Numba IR for the compiled function is next searched for all *call* instructions, each of which represents a temporary change in flow of control. The instrumentation pass inserts code immediately before each *call* to deactivate the local timer and immediately after each *call* to reactivate it. Again, this inserted code issues more calls to profiler-implemented functions, passing the unique function ID as an argument.
7. Profiler-implemented functions for updating counters and timers are registered using the Numba low-level extension API. This means developer-supplied Python code is invoked by Numba at compile time (specifically, during the lowering stage) to generate the LLVM IR for each call site. This Python code first examines the unique function ID generated as an argument by the instrumentation pass, then consults with the registry to recover the data structure allocated to track counter and timer state for that function. Next, the LLVM IR generated for the call is synthesized to directly update that structure at the appropriate memory location. Because the instructions are generated in-line within the calling function, and because those instructions are bound in advance to the correct (constant) memory address, the overhead of instrumentation is kept to a minimum.

With the instrumentation mechanism realized as described previously, jitted functions are automatically compiled with the instructions needed to increment per-function counters and to activate and deactivate per-function timers. As the program executes, data structures allocated by the Python profiler code are updated by these compiled functions. Finally, after calls from Python code to jitted functions are complete, the registry and its data structures may be interrogated to report on performance metrics function by function.

An example of execution profiler output is given in Figure 6.

10. Because of an apparent bug in Numba at the time of writing, whereby the Numba default compiler pipeline is used for all inlined functions regardless of whether a custom pipeline was applied to the caller or to the callee, the instrumenting pass was forcibly injected into the default pipeline using the Python technique of monkey-patching.

FIGURE 6: EXAMPLE OF EXECUTION PROFILER OUTPUT

	call_count	local_time	total_time	%time
main.foo	10	1.42641	2.14886	75.8922
main.loiter	59	0.224952	0.224952	11.9686
main.call_target_3	29	0.113304	0.351926	6.02833
main.call_target_1	20	0.053101	0.218058	2.82524
main.bar	10	0.0447865	2.24661	2.38287
main.call_target_2	10	0.0169675	0.063025	0.902757

Note that techniques used to implement the proof-of-concept profiler draw on Numba capabilities covered in both Extension APIs and Compiler Customization. Additionally, techniques leveraging the **ctypes** Python module for accessing low-level C APIs to allocate and free memory, and to query high-resolution timers, were also employed. Numba intrinsic functions were developed to directly read and write memory managed via **ctypes**, enabling access to per-function state from both Python and instrumented code.

OOP PARADIGMS

JitClasses

We introduced the experimental jitclass feature in the section on OOP. This facility enables developers to organize code and data using class-like constructs similar to those in interpreted Python but with a number of restrictions, including the same language constraints imposed on jitted functions, no inheritance or polymorphism capabilities, and demands for strict typing of instance variables. (Jitclass support for CUDA is also unavailable at present time.)

StructRefs

As an alternative to jitclasses, Numba offers another experimental feature for aggregating data and methods: structrefs. A structref is a data structure to which methods can be attached, making it very similar in functionality to a Python class or Numba jitclass. The key difference between structrefs and other Numba datatypes, including jitclasses, is that a structref is passed as an argument to functions *by reference* rather than *by value*.

When a compiler generates code to pass by reference, the generated machine code uses a reference—typically, a memory address—as a proxy for the actual data. Accordingly, instructions to modify the associated data, whether in the caller or the callee, access those data through the same reference, meaning any changes applied by callee are reflected to the caller. Conversely, when a compiler generates code to pass by value, the generated machine code makes a copy of the data for the callee to access directly. Under this scenario, changes made by the callee are localized to the copy of data visible only to that function and do not modify the original argument accessible to the caller.

In addition to the capacity to mutate data across function calls, passing by reference also leads to faster machine code. This is because the single memory address used as a proxy for actual data translates to the absolute minimum number of instructions needed to implement argument-passing at machine level. The number of instructions required to pass by value, conversely, scales with the size of the associated data.

Applying these concepts to jitclasses and structrefs suggests the latter should consistently outperform the former, and indeed, empirical evidence confirms that expectation. It should be noted that, while jitclasses are passed by value, the compiler generates the necessary code for changes to jitclass data in a called function to be propagated back to the caller. This code ensures jitclasses *behave* as if they are passed by reference, but the generated machine code still includes instructions to perform by-value copies. The efficacy of the LLVM optimizer can potentially offset some of the computational expense of by-value copies when callee instructions are inlined to the caller, but in virtually all cases, use of a single memory address as a proxy for structref data will still outperform even highly optimized code for passing jitclass data.

While structrefs provide a generally faster alternative to jitclasses as a mechanism for realizing some object-oriented features in Numba, both facilities are currently experimental. (It also appears the shift in focus toward the newer and preferred structref mechanism has slowed investment in further development of jitclass support.) Additionally, the current state of structref support requires tedious implementation of boilerplate code on a field-by-field basis, leading to bloated source files and an elevated risk of human error.

Milliman proof of concept (MPOC)

As part of our investigations into Numba extensibility, we endeavored to build our own proof-of-concept object-oriented support in Numba, free of dependencies on Numba experimental features.[11] This exercise was pursued with the intent to compromise some conveniences of jitclasses and structrefs—such as access to objects in both interpreted and jitted code, support for datetime and string data types, and seamless integration with libraries like Numpy—in an effort to further increase performance and to minimize boilerplate code.

A simplified overview of the MPOC follows:

1. Implement an MPOC *type* decorator for introducing a class name and for defining the names and types of the associated fields. The decorator itself is parameterized to accept the class name, and the function it decorates is expected to return a list of tuples matching field names to field types for that class.
2. The type decorator then registers a custom type with Numba based on the class name parameter and calls its decorated function to obtain field definitions associated with that type.
3. The type decorator next analyzes the field definitions to construct a map of how those field values are to be organized in a block of contiguous memory and to calculate the size in bytes required to maintain such a block.
4. For each field, the type decorator registers an attribute setter and getter (associated with the registered custom type), using the name of the field as the attribute name. The decorator also registers implementations for the setter and getter, each of which generates LLVM code directly, given a base pointer as the *self* argument, and an implicit offset from that pointer determined by the field map, and knowledge of the field type.
5. The type decorator also registers a parameterless constructor function, using the same name as the custom type. The constructor implementation allocates a memory block whose size is based on the field analysis, zero-fills the block, and returns the address of that block. This return value is the *self* parameter for all associated attributes and methods.
6. Implement an MPOC *method* decorator for attaching a jitted Python function to the type introduced via the type decorator. This decorator accepts the type name as a parameter and registers the function to which it applies with the Numba extension system.

With the type and method decorators described previously, it is possible to define a class useable from jitted code by using the type decorator to define the class name and fields and by using the method decorator to define one or more methods. A jitted function can then instantiate the class by calling its parameterless constructor. (Note that no provision is made to initialize fields at instantiation time; instead, all fields take on default values consistent with zero-filled memory. However, because all fields are backed by settable attributes on the associated type, a separate initialization function can be implemented to override some or all default field values, potentially using arguments passed to that function.)

Because the custom Numba type associated with the class is configured to carry just a single value, namely, the memory address of the block used to maintain object state, the expense of passing that object to jitted functions is minimal, as described in the pass-by-reference treatment previous. Additionally, because attribute setters and getters resolve to inlined machine instructions that write or read memory relative to this memory address directly, the overhead to access class data is likewise minimal.

11. While bypassing direct use of experimental jitclasses and structrefs was straightforward, and most of the Numba extensions leveraged were described by official Numba documentation, a fraction of our work was derived from facilities exported from Numba modules but not documented outside of the Numba source tree. It is unclear whether these facilities are stable and production-ready but highlight the incompleteness of documentation by the Numba team or whether these facilities are intended to be opaque and are subject to change without notice.

One limitation of the MPOC implementation is that—in addition to memory addresses representing instances of other MPOC class types—only integer, floating-point, and boolean field types are supported. This leads to highly efficient LLVM code for setter and getter implementations but complicates use of more complex types supported elsewhere in Numba, most notably, Numpy arrays.

To facilitate access to Numpy arrays, a specialized custom type capable of conveying the structure and address of a Numpy array was developed, using the type and method decorators described previously. This class operates as a *wrapper* around Numpy arrays and exposes methods to assign or retrieve the underlying Numpy array. An unfortunate side effect of this approach is that functions and methods using this MPOC wrapper must make explicit calls to work natively with Numpy objects and methods.

To partially offset the inconvenience of the Numpy wrapper class, Numba methods were introduced on that wrapper to support basic indexing operations. Implementation of these methods bypass use of Numpy altogether, using the structure and address information maintained by the wrapper to read or write the appropriate memory address with minimal overhead. While more complex Numpy operations—like array slices or calls to methods that operate on multiple elements (e.g., **sum**)—still require explicit calls to retrieve the wrapped Numpy object, this shortcut for direct indexing of individual array elements leads to highly economical and performant code where applicable.

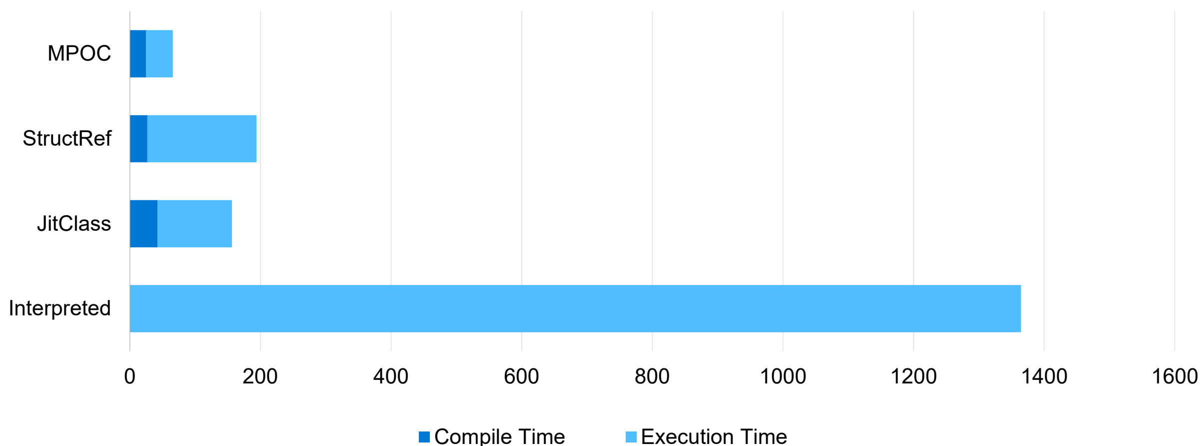
PERFORMANCE ANALYSIS

To assess relative performance of the various OOP techniques described here, we first developed a reasonably small but computationally heavy reference model in Python and then applied each technique to arrive at four different implementations: an interpreted Python model, a jitclass model, a structref model, and an MPOC model.

The reference model calculates the Solvency II (SII) best estimate liability (BEL) for unit-linked contract. This type of calculation involves projecting contract cashflows over multiple market scenarios and discounting the cashflows back to a given valuation date. In the reference model, 100,000 contracts were run with a monthly timestep for 50 years.

Performance metrics for the four models are given in Figure 7.

FIGURE 7: PERFORMANCE METRICS FOR THE FOUR MODELS – REFERENCE MODEL TIMING METRICS



One obvious takeaway from the results in Figure 7 is that all Numba-based implementations significantly outperform the interpreted Python model. This is of course not a surprising result, but it does underscore the performance benefits achievable through JIT compilation.

Another observation is that compile times for the Numba-based implementations vary, with jitclasses consuming 1.6 to 1.7 times the compile time of structrefs and MPOC, respectively. However, setting aside compile time, the jitclass implementation outperformed the structref solution by a factor of approximately 1.5. This highlights the possibility that the advantage of one over the other, strictly in terms of overall runtime, depends on how much *reuse* of compiled code applies during the execution phase. For example, the metrics in Figure 7 were based on running 100,000 model-point inputs. However, if that count is reduced by a factor of 10, the impact of jitclass compilation time relative to execution time is sufficient for the structref solution to exhibit an overall runtime faster than the jitclass counterpart.

We note that MPOC outperformed the experimental jitclass and structref implementations by a healthy measure (by more than a factor of two). Although both approaches pursue a similar strategy, namely, use of common data accessed from jitted code by a pointer (memory address) reference, it is difficult to pinpoint all of the differences that account for the performance disparity. Certainly, the MPOC solution imposed more concessions on model code than the structref implementation, including restrictions on data types, workarounds for Numpy arrays, and asymmetry in object representation between interpreted and jitted Python. Some of these adaptations are unique to the MPOC implementation and may themselves contribute to faster runtimes. There are also likely opaque differences in how this particular model was optimized by the LLVM compiler—after structref and MPOC implementations synthesized method calls and attribute accessors—which might further explain differences in performance.

Conclusions

Numba offers considerable promise as a mechanism for significantly accelerating Python code. In some cases, existing Python code may need to be heavily modified to comply with some of the language restrictions imposed by Numba. While such modification may be minimal or may be substantial, the knowledge and skills required are consistent with general knowledge of Python and basic familiarity with Numba.

To pursue more aggressive performance improvements and to leverage the benefits of OOP principles, more knowledge of Numba is required to take advantage of its experimental jitclass or structref features and, potentially, to entertain introduction of other potential accelerators, such as intrinsic functions. Knowledge of lower-level LLVM constructs can also be helpful in trying to decipher compiled code to better understand performance implications.

Pursuit of maximally efficient Numba code, as might be desirable when running computationally massive workloads on metered resources in the cloud, demands an elevated knowledge of Numba, LLVM, and machine-level code. As evidenced in this paper, such knowledge can be practically employed to minimize Numba compile times, analyze performance of compiled code at a useful level of granularity, and generate highly efficient machine code from higher-level sources organized in accordance with robust object-oriented design.

Solutions for a world at risk™

Milliman leverages deep expertise, actuarial rigor, and advanced technology to develop solutions for a world at risk. We help clients in the public and private sectors navigate urgent, complex challenges—from extreme weather and market volatility to financial insecurity and rising health costs—so they can meet their business, financial, and social objectives. Our solutions encompass insurance, financial services, healthcare, life sciences, and employee benefits. Founded in 1947, Milliman is an independent firm with offices in major cities around the globe.

milliman.com

CONTACT

Jim Brackett
jim.brackett@milliman.com

Donal McGinley
donal.mcginley@milliman.com

Chad Schuster
chad.schuster@milliman.com

© 2025 Milliman, Inc. All Rights Reserved. The materials in this document represent the opinion of the authors and are not representative of the views of Milliman, Inc. Milliman does not certify the information, nor does it guarantee the accuracy and completeness of such information. Use of such information is voluntary and should not be relied upon unless an independent review of its accuracy and completeness has been performed. Materials may not be reproduced without the express consent of Milliman.