Numba: A LLVM-based Python JIT Compiler

Group 16: John Oyer, Evan Gebo, Nathan Yap

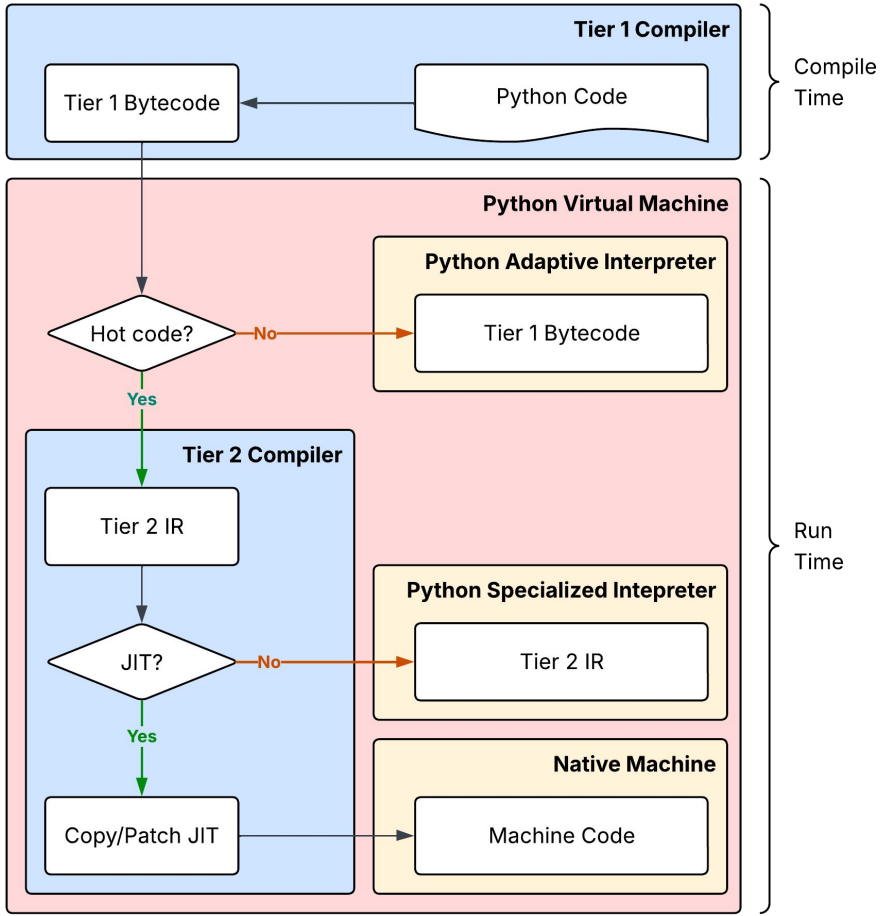
Python is a Very Slow Language

Python is Slow

- Python is a **dynamically typed** and **interpreted** language
- This is too slow for modern tasks (e.g. ML workflows)

But it Doesn't Have to Be!

- The default Python interpreter is **written in C** (the CPython interpreter)
- Other dynamic interpreted languages (e.g. JavaScript) **leverage JIT** to be fast
- Other Python implementations (e.g. PyPy) are faster than CPython
- Libraries like **NumPy, SciPy, or PyTorch** provide **efficient math operations**



Python Code

- Raw code written by user

Tier 1 Bytecode

- Parsed user code
- Very **minimal optimization** (dead code elimination and constant propagation)

Tier 2 Intermediate Representation

- Further **type-specialized bytecode**
- More optimizations applied

Copy/Patch JIT

- Applied to specific traces
- Generated with **LLVM**

Numba Is a JIT for Scientific Computing

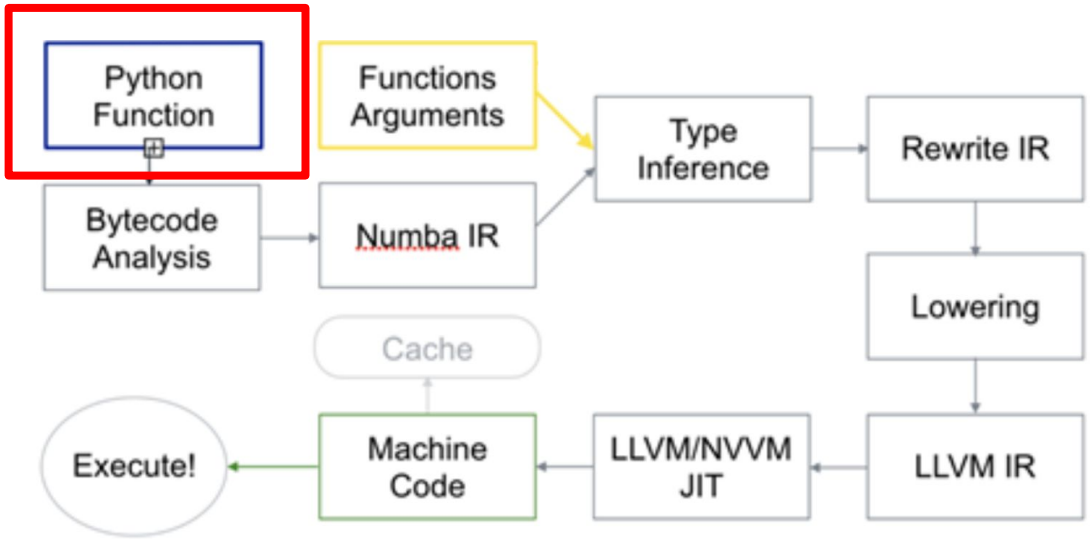
- One of the **first JITs for scientific computing** implemented for Python
- Requires **minimal code changes**
- **Integrates with the scientific Python** ecosystem (e.g. NumPy)

The User Calls Numba's JIT Compiler

```
@jit
def do_math(a, b):
    return a + b
```

Handled by Numba, **not** the Python VM in the flowchart

Numba Execution Diagram



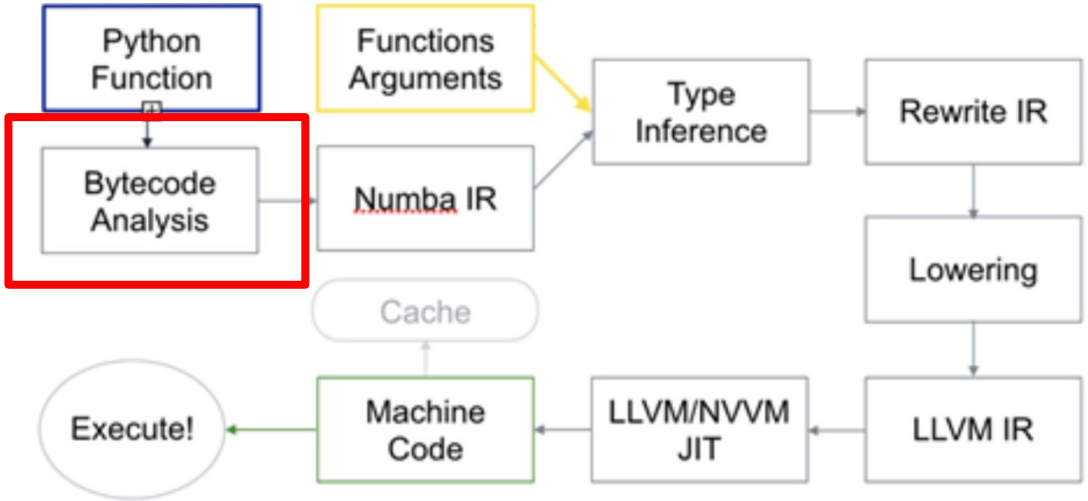
Code starts off as a Python function with the @jit decorator

Ex:

```
@jit
def do_math(a,b):
    return a + b
```

Image Source: https://www.nvidia.com/en-gb/glossary/numba/

Numba Execution Diagram



The Bytecode generated for this function has to be analyzed post-hoc during runtime.

Ex:

```
3           0 RESUME                   0

4           2 LOAD_FAST                0 (a)
            4 LOAD_FAST                1 (b)
            6 BINARY_OP                0 (+)
           10 RETURN_VALUE
```

Image Source: https://www.nvidia.com/en-gb/glossary/numba/

Disassembling ByteCode

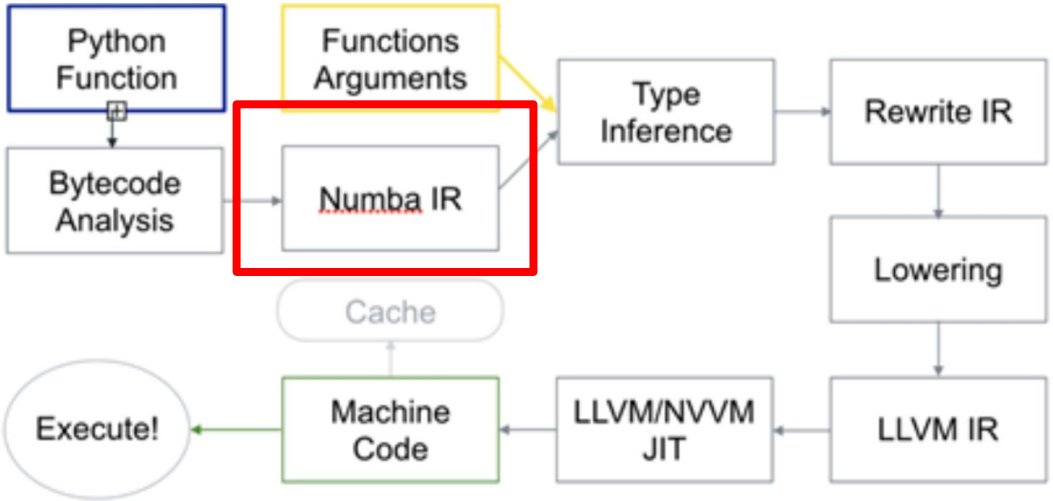
Python ByteCode has complex instructions that can modify the control-flow

```
12          35 LOAD_FAST                0 (n)
            38 LOAD_CONST               3 (10)
            41 COMPARE_OP               2 (==)
            44 POP_JUMP_IF_FALSE       19
```

Example ByteCode instruction which jumps to bytecode offset 19

Jump-targets are used to identify BB and create a CFG

Numba Execution Diagram



The Python Interpreter uses a stack machine representation that must be translated into a register machine repr.

TLDR: Registers need to be assigned to values

Image Source: https://www.nvidia.com/en-gb/glossary/numba/

Stack to Register Representation

Bytecode for a basic add function (return a + b)

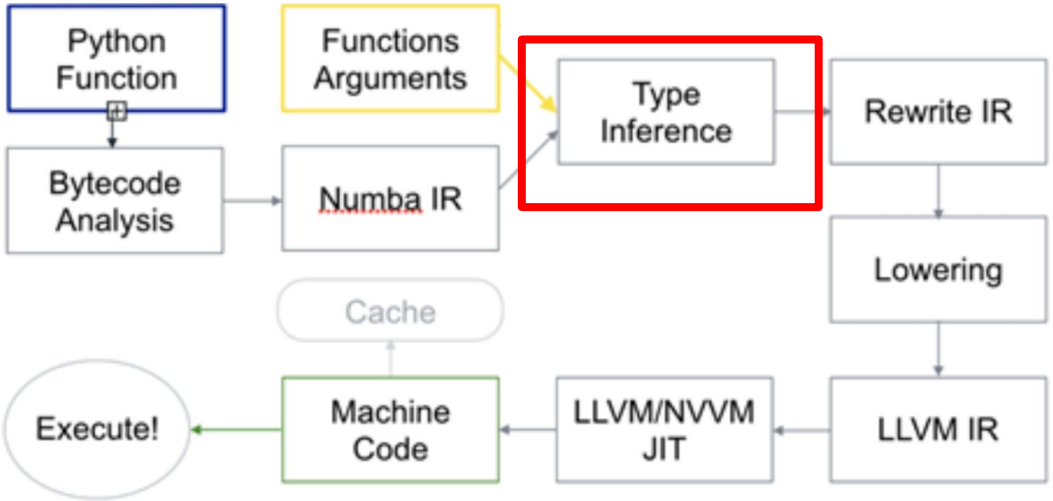
```
2           0 LOAD_FAST                0 (a)
            3 LOAD_FAST                1 (b)
            6 BINARY_ADD
            7 RETURN_VALUE
```

Numba IR

```
label 0:
    a = arg(0, name=a)                       ['a']
    b = arg(1, name=b)                       ['b']
    $0.3 = a + b                             ['$0.3', 'a', 'b']
    del b                                    []
    del a                                    []
    $0.4 = cast(value=$0.3)                  ['$0.3', '$0.4']
    del $0.3                                 []
    return $0.4                              ['$0.4']
```

Uses registers rather than a stack

Numba Execution Diagram



Intermediate var types need to be inferred during runtime by Numba

This is where Numba differentiates nopython mode and object mode

Image Source: https://www.nvidia.com/en-gb/glossary/numba/

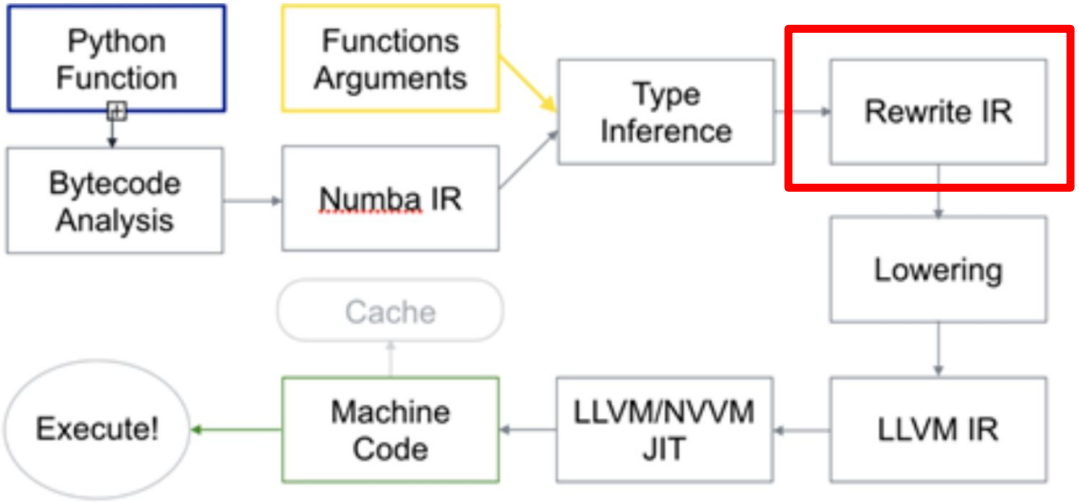
Nopython vs Object Modes

Nopython mode: Numba will fully compile a function into machine code (with no interpreter overhead)

Object mode: Numba will compile what it can within a function (such as loops, etc.)

If Numba cannot infer types or otherwise support certain variables, it will fallback to Object Mode

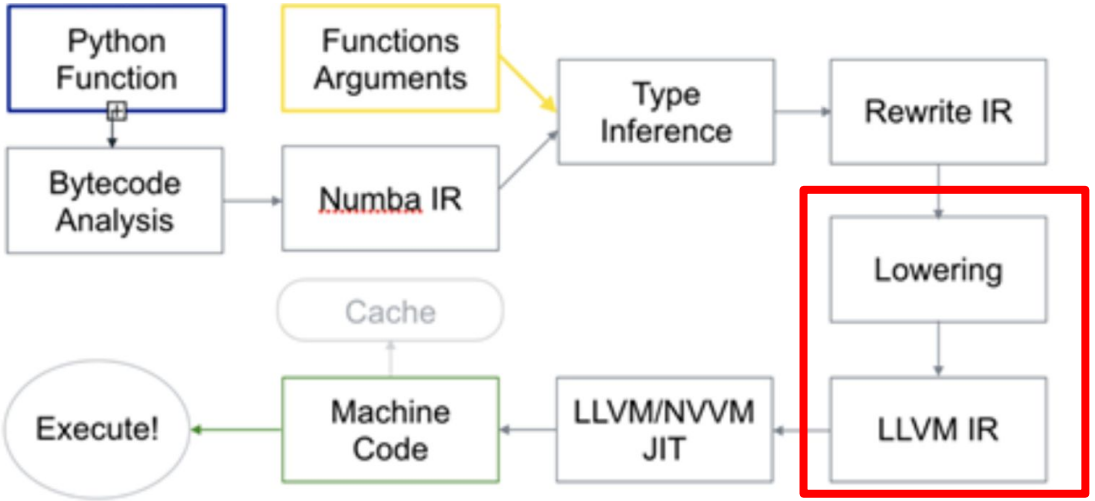
Numba Execution Diagram



High-level optimizations can be applied to the new rewritten IR

Image Source: https://www.nvidia.com/en-gb/glossary/numba/

Numba Execution Diagram



Based on the compilation context (CPU/GPU/etc.) the IR may be lowered into LLVM IR using a basic mapping

Image Source: https://www.nvidia.com/en-gb/glossary/numba/

Nopython LLVM IR

```
define i32 @"__main__.add$1.int64.int64"(i64* %"retptr",
                                         {i8*, i32}** %"excinfo",
                                         i8* %"env",
                                         i64 %"arg.a", i64 %"arg.b")
{
   entry:
     %"a" = alloca i64
     %"b" = alloca i64
     %"$0.3" = alloca i64
     %"$0.4" = alloca i64
     br label %"B0"
   B0:
     store i64 %"arg.a", i64* %"a"
     store i64 %"arg.b", i64* %"b"
     %".8" = load i64* %"a"
     %".9" = load i64* %"b"
     %".10" = add i64 %".8", %".9"
     store i64 %".10", i64* %"$0.3"
     %".12" = load i64* %"$0.3"
     store i64 %".12", i64* %"$0.4"
     %".14" = load i64* %"$0.4"
     store i64 %".14", i64* %"retptr"
     ret i32 0
}
```

*After optimization

```
define i32 @"__main__.add$1.int64.int64"(i64* nocapture %retptr,
                                         { i8*, i32 }** nocapture readnone %excinfo,
                                         i8* nocapture readnone %env,
                                         i64 %arg.a, i64 %arg.b)


   entry:
     %.10 = add i64 %arg.b, %arg.a
     store i64 %.10, i64* %retptr, align 8
     ret i32 0
}
```

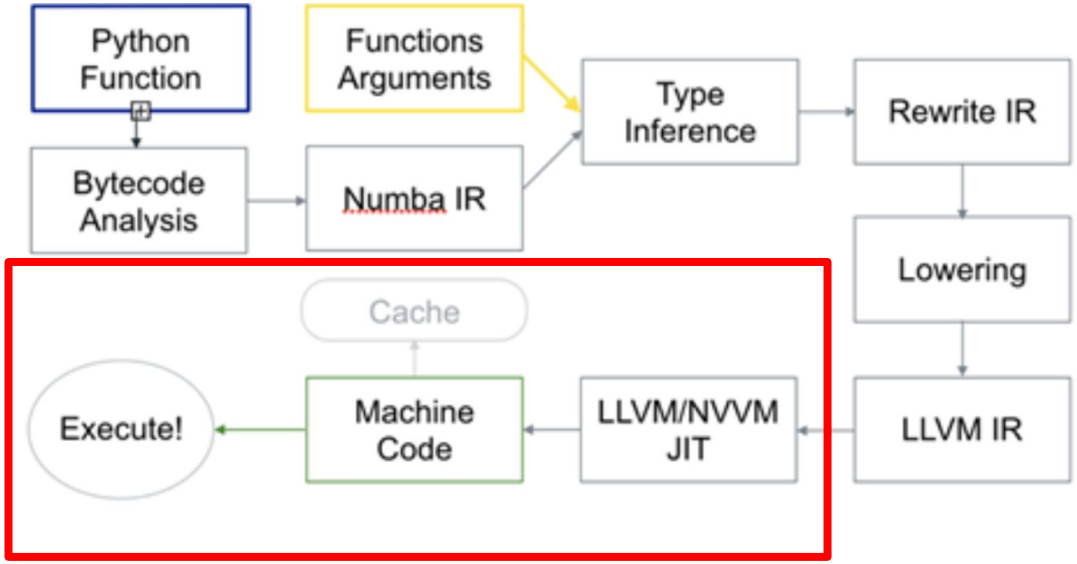
Object mode LLVM IR

```
@PyExc_SystemError = external global i8
@".const.Numba_internal_error:_object_mode_function_called_without_an_environment" = internal
@".const.name_'a'_is_not_defined" = internal constant [24 x i8] c"name 'a' is not defined\00"
@PyExc_NameError = external global i8
@".const.name_'b'_is_not_defined" = internal constant [24 x i8] c"name 'b' is not defined\00"

define i32 @"__main__.add$1.pyobject.pyobject"(i8** nocapture %retptr, { i8*, i32 }** nocaptu
entry:
  %.6 = icmp eq i8* %env, null
  br i1 %.6, label %entry.if, label %entry.endif, !prof !0

entry.if:                                         ; preds = %entry
  tail call void @PyErr_SetString(i8* @PyExc_SystemError, i8* getelementptr inbounds ([73 x i
  ret i32 -1

entry.endif:                                      ; preds = %entry
  tail call void @Py_IncRef(i8* %arg.a)
  tail call void @Py_IncRef(i8* %arg.b)
  %.21 = icmp eq i8* %arg.a, null
  br i1 %.21, label %B0.if, label %B0.endif, !prof !0

B0.if:                                            ; preds = %entry.endif
  tail call void @PyErr_SetString(i8* @PyExc_NameError, i8* getelementptr inbounds ([24 x i8]
  tail call void @Py_DecRef(i8* null)
  tail call void @Py_DecRef(i8* %arg.b)
  ret i32 -1

B0.endif:                                         ; preds = %entry.endif
  %.30 = icmp eq i8* %arg.b, null
  br i1 %.30, label %B0.endif1, label %B0.endif1.1, !prof !0
```

+A lot more code

The object mode LLVM IR will make many calls to the CPython API in order to maintain functionality

Numba Execution Diagram



All that's left is to compile using LLVM and route function calls to the optimized machine code

Image Source: https://www.nvidia.com/en-gb/glossary/numba/

Numba vs. Other Methods

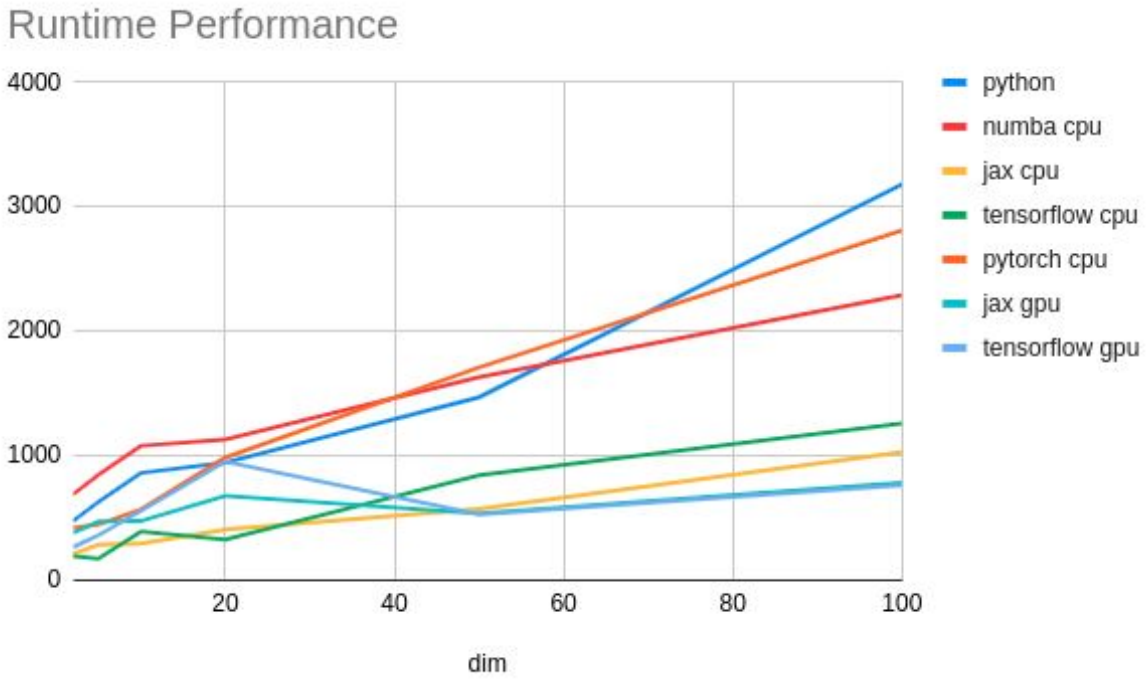
- **CPython:** Default Python interpreter executing Python bytecode
- **CPython w/ JIT:** Experimental version of CPython with JIT to accelerate hot paths
- **Cython:** Python-to-C compiler with optional static types
- **PyPy:** Alternative Python interpreter with a meta-tracing JIT for faster execution
- **JAX:** NumPy-like numerical library with automatic differentiation and hardware acceleration for machine learning and scientific computing
- **Cinder:** CPython fork developed by Meta for server-side performance
- **Numba:** JIT compiler library accelerating Python numerical code via LLVM

Numba vs. Other Methods

	CPython	CPython w/ JIT	Cython	PyPy	JAX	Cinder (Meta)	Numba
No Code Change Required?	✅	✅	❌ *(static typing)*	✅	❌	✅	❌ *(minimal)*
Efficient Scientific Python Integration?	❌	❌	✅ *(with c api)*	❌	✅	❌	✅
Only library? (no interpreter replacement)	–	–	❌ *(compiler/ language)*	❌	✅	❌	✅
Uses LLVM?	❌	✅ *(build time only)*	✅ *(clang/g++)*	❌	❌	❌	✅

Numba vs. Other Methods

- **Linear Regression** is trained via gradient descent
- **Runtime performance** is compared with varying model dimension (higher dimension -> more computations required)
- **Numba** shows better performance when dimension is large



Analysis Performed by Gavin Chan, 2022

Limitations

- JIT compilation introduces **overhead** to go from Python bytecode -> LLVM IR
- Requires **code change** (decorators/annotations) to see performance improvements, which requires user capability/knowledge
- Python language or library **updates** -> Numba may **break** (must sync with other libraries)
- Best for **numerical, loop-heavy** code; may slow down general-purpose Python

Thank You!