MLIR × Python: Building Core Compilation Pipelines in Python

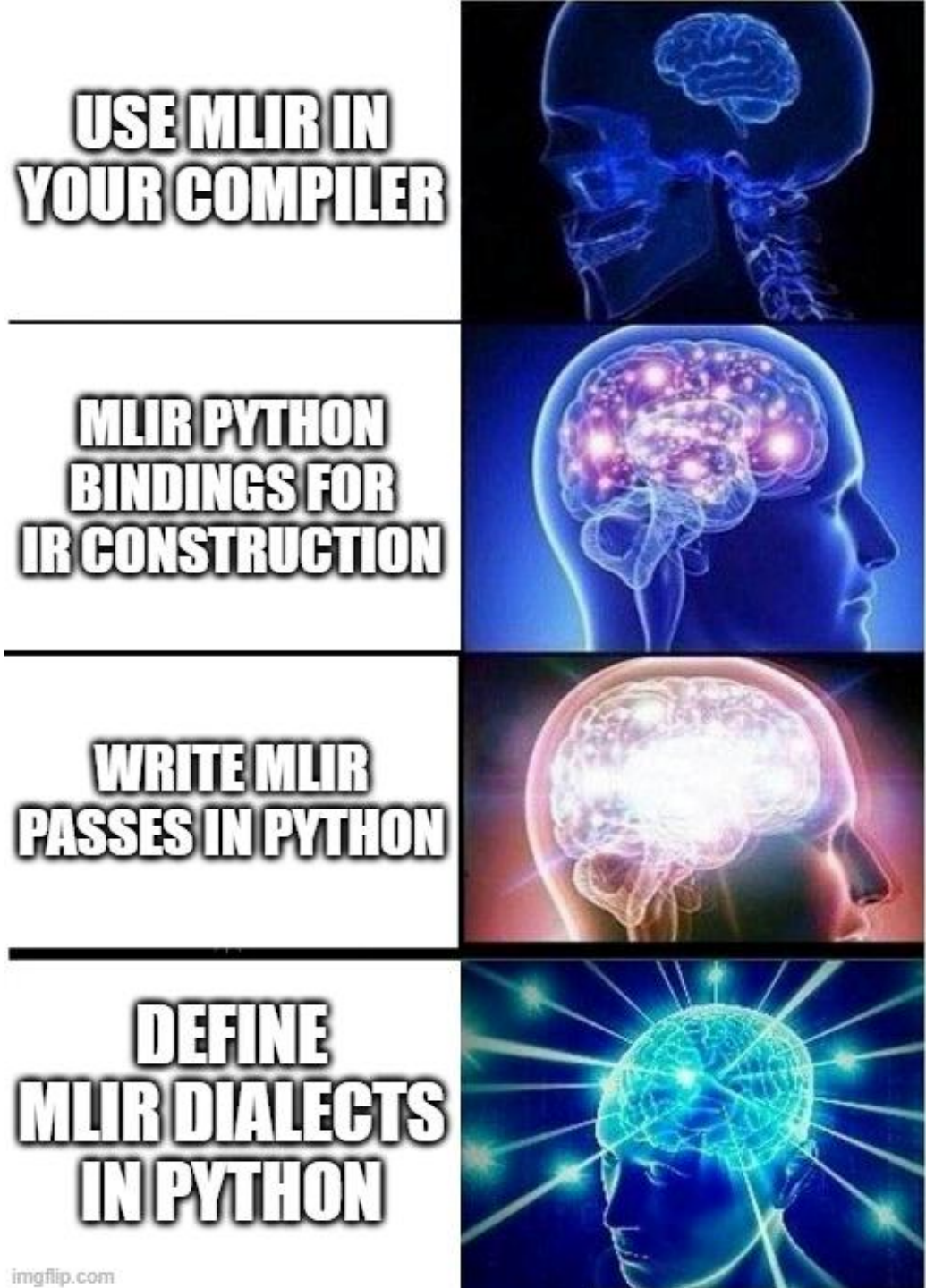
Mingyang Liu, twice@apache.org

https://github.com/PragmaTwice

Agenda

- Introduce MLIR Python
- Write Passes in Python
- Add Rewrite Patterns in Python
- Define Dialects in Python
- Python bindings API docs

These slides are mostly example-driven!



MLIR Python bindings

- Almost one-to-one mapping of MLIR C++ APIs, while (try to be) pythonic by design.
- Inside LLVM monorepo, DISABLE by default in CMake.
- Implemented via nanobind and MLIR C APIs (due to RTTI-related issues).
- Still experimental and incomplete.
- Used in IREE, CIRCT, Mojo, CUDA ..

MLIR Python bindings: IR Construction

```python
from mlir.ir import *
from mlir.dialects import func, arith

with Context(), Location.unknown():
    f32 = F32Type.get()

    module = Module.create()
    with InsertionPoint(module.body):

        @func.func(f32, f32, f32)
        def fma(x, y, z):
            return arith.addf(x, arith.mulf(y, z))

print(module)
```

MLIR Python bindings: IR Construction

```python
from mlir.ir import *
from mlir.dialects import func, arith

with Context(), Location.unknown():
    f32 = F32Type.get()

    module = Module.create()
    with InsertionPoint(module.body):

        @func.func(f32, f32, f32)
        def fma(x, y, z):
            return arith.addf(x, arith.mulf(y, z))

print(module)
```

MLIR Python bindings: IR Construction

```python
from mlir.ir import *
from mlir.dialects import func, arith

with Context(), Location.unknown():
    f32 = F32Type.get()

    module = Module.create()
    with InsertionPoint(module.body):

        @func.func(f32, f32, f32)
        def fma(x, y, z):
            return arith.addf(x, arith.mulf(y, z))

print(module)
```

```
module {
  func.func @fma(%arg0: f32, %arg1: f32,
%arg2: f32) -> f32 {
    %0 = arith.mulf %arg1, %arg2 : f32
    %1 = arith.addf %arg0, %0 : f32
    return %1 : f32
  }
}
```

MLIR Python bindings: Pass Manager

```
from mlir.passmanager import PassManager

pm = PassManager('any', module.context)
pm.add('convert-to-llvm')
pm.run(module.operation)

print(module)
```

MLIR Python bindings: Pass Manager

```python
from mlir.passmanager import PassManager

pm = PassManager('any', module.context)
pm.add('convert-to-llvm')
pm.run(module.operation)

print(module)
```

```
module {
  llvm.func @fma(%arg0: f32, %arg1: f32,
%arg2: f32) -> f32 {
    %0 = llvm.fmul %arg1, %arg2 : f32
    %1 = llvm.fadd %arg0, %0 : f32
    llvm.return %1 : f32
  }
}
```

MLIR Python bindings: Execution Engine

```python
from mlir.execution_engine import ExecutionEngine
import ctypes

with Context():
    ee = ExecutionEngine(module)
    c_float_p = ctypes.c_float * 1
    res = c_float_p(0)
    ee.invoke("fma", c_float_p(1), c_float_p(2), c_float_p(3), res)
    assert res[0] == 7
```

Note: `llvm.emit_c_interface` is omitted.

Pass in Python

- Could we just write passes in Python side?
- Don’t need to touch CMake, tablegen, C++ headers/sources, and recompile the whole MLIR.
- Prototyping a new pass in Python can be way faster.
- Via `pm.add(callable)`

Pass in Python: Example

```python
def custom_pass(op, pass_):
    # do something with `op`
    pass

pm = PassManager("any")

pm.add(custom_pass)
pm.add("canonicalize, cse, convert-to-llvm, ...")

pm.run(module)
```

Rewrite Pattern in Python

- Python passes are great. But wait.. how to define rewrite patterns?
- The greedy pattern rewrite driver is available in Python!
- (Driver for dialect conversion? Not yet.)
- Two ways to define patterns:
  - Define patterns as Python callables via PatternRewriter and RewritePatternSet;
  - Define patterns as constructing IR via PDL.

Rewrite Pattern in Python: Example

```
def to_muli(op, rewriter):
    with rewriter.ip:
        assert isinstance(op, arith.AddIOp)
        new_op = arith.muli(op.lhs, op.rhs, loc=op.location)
    rewriter.replace_op(op, new_op.owner)

patterns = RewritePatternSet()
patterns.add(arith.AddIOp, to_muli)
patterns.add(...)

apply_patterns_and_fold_greedily(module, patterns.freeze())
```

Rewrite Pattern in Python: via PDL

```python
m = Module.create()
with InsertionPoint(m.body):
    @pdl.pattern(benefit=1, sym_name="addi_to_mul")
    def pat():
        index_type = pdl.TypeOp(IndexType.get())
        operand0 = pdl.OperandOp(index_type)
        operand1 = pdl.OperandOp(index_type)
        op0 = pdl.OperationOp(
            name="arith.addi", args=[operand0, operand1], types=[index_type]
        )

        @pdl.rewrite()
        def rew():
            newOp = pdl.OperationOp(
                name="arith.muli", args=[operand0, operand1], types=[index_type]
            )
            pdl.ReplaceOp(op0, with_op=newOp)

frozen = PDLModule(m).freeze()
apply_patterns_and_fold_greedily(module, frozen)
```

Define Dialects in Python

- IRDL is available in MLIR Python.
- Define new dialects as constructing MLIR modules containing `irdl.dialect` ops.
- Load your dialects via `irdl.load_dialects(module)`

Define Dialects in Python: IRDL

```python
m = Module.create()
with InsertionPoint(m.body):
    myint = irdl.dialect("myint")
    with InsertionPoint(myint.body):
        constant = irdl.operation_("constant")
        with InsertionPoint(constant.body):
            iattr = irdl.base(base_name="#builtin.integer")
            i32 = irdl.is_(TypeAttr.get(IntegerType.get_signless(32)))
            irdl.attributes_([iattr], ["value"])
            irdl.results_([i32], ["cst"], [irdl.Variadicity.single])

        add = irdl.operation_("add")
        with InsertionPoint(add.body):
            i32 = irdl.is_(TypeAttr.get(IntegerType.get_signless(32)))
            irdl.operands_(
                [i32, i32],
                ["lhs", "rhs"],
                [irdl.Variadicity.single, irdl.Variadicity.single],
            )
            irdl.results_([i32], ["res"], [irdl.Variadicity.single])

irdl.load_dialects(m)
```

Define Dialects in Python: IRDSL?

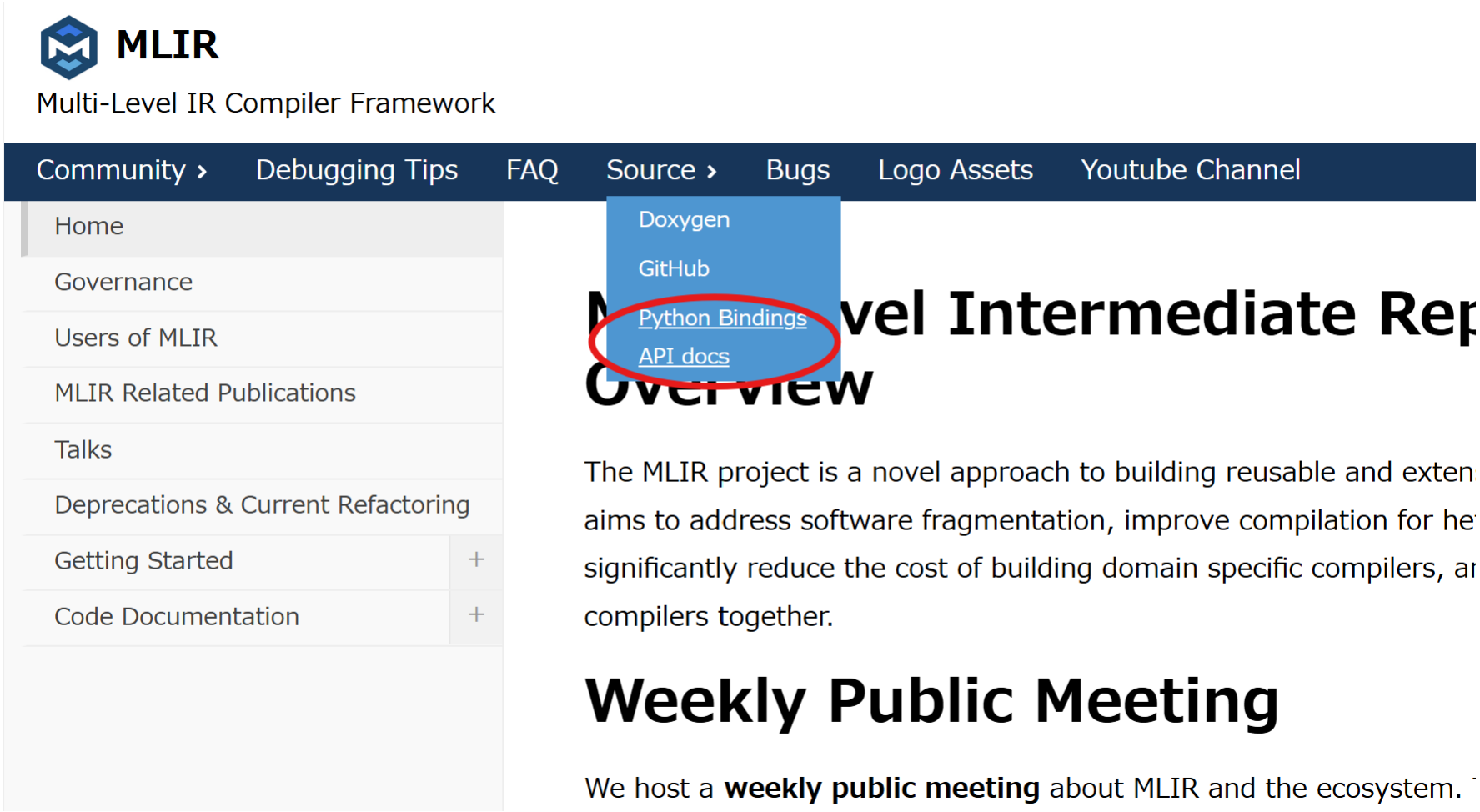
- A more intuitive and simpler interface to define dialects and ops
- Generate OpView subclasses and builder functions automatically
- Experimental and NOT yet get merged

Define Dialects in Python: IRDSL?

```python
myint = irdsl.Dialect("myint")
iattr = irdsl.BaseName("#builtin.integer")
i32 = irdsl.IsType(IntegerType.get_signless(32))

@myint.op("constant")
class ConstantOp:
    value = irdsl.Attribute(iattr)
    cst = irdsl.Result(i32)

@myint.op("add")
class AddOp:
    lhs = irdsl.Operand(i32)
    rhs = irdsl.Operand(i32)
    res = irdsl.Result(i32)

myint.load()
```

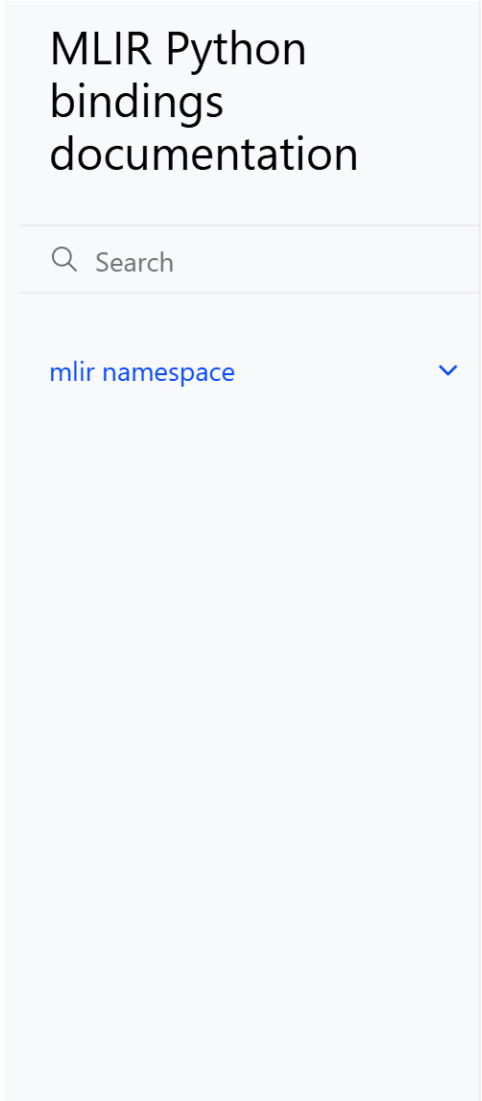
MLIR Python API docs

https://mlir.llvm.org



MLIR Python API docs

https://mlir.llvm.org/python-bindings



MLIR Python bindings API documentation

Welcome to the MLIR Python bindings API documentation. This website is generated from MLIR Python source code and type stubs.

- mlir namespace
  - mlir._mlir_libs._mlir.dialects.pdl
  - mlir._mlir_libs._mlir.dialects.quant
  - mlir._mlir_libs._mlir.dialects.transform
  - mlir._mlir_libs._mlir
  - mlir._mlir_libs._mlir.ir
  - mlir._mlir_libs._mlir.passmanager
  - mlir._mlir_libs._mlir.rewrite
  - mlir._mlir_libs._mlirExecutionEngine
  - mlir._mlir_libs._mlirPythonTestNanobind
  - mlir._mlir_libs
  - mlir.dialects._acc_ops_gen
  - mlir.dialects._affine_enum_gen
  - mlir.dialects._affine_ops_gen
  - mlir.dialects._amdgpu_enum_gen
  - mlir.dialects._amdgpu_ops_gen

MLIR Python API docs

https://mlir.llvm.org/python-bindings/...

```
print(large_elements_limit: int | None = None, large_resource_limit: int | None = None,
      enable_debug_info: bool = False, pretty_debug_info: bool = False,
      print_generic_op_form: bool = False, use_local_scope: bool = False,
      use_name_loc_as_prefix: bool = False, assume_verified: bool = False, file: object |
      None = None, binary: bool = False, skip_regions: bool = False) → None
```

Prints the assembly form of the operation to a file like object.

PARAMETERS:

- **large_elements_limit** – Whether to elide elements attributes above this number of elements. Defaults to None (no limit).
- **large_resource_limit** – Whether to elide resource attributes above this number of characters. Defaults to None (no limit). If large_elements_limit is set and this is None, the behavior will be to use large_elements_limit as large_resource_limit.
- **enable_debug_info** – Whether to print debug/location information. Defaults to False.
- **pretty_debug_info** – Whether to format debug information for easier reading by a human (warning: the result is unparseable). Defaults to False.
- **print_generic_op_form** – Whether to print the generic assembly forms of all ops. Defaults to False.

MLIR Python

- Try it now?

```
$ pip install mlir-python-bindings -f "https://llvm.github.io/eudsl"

$ python
>>> from mlir import ir
>>> with ir.Context():
...     print(ir.Module.parse("arith.constant 10"))
...
module {
  %c10_i64 = arith.constant 10 : i64
}
```

MLIR Python

- Try it in your browser? Without installation!

- https://llvm.github.io/eudsl/console - Pyodide REPL with MLIR Python bindings loaded
- https://llvm.github.io/eudsl/jupyter - JupyterLab packed with MLIR Python bindings

- via the Wasm version of MLIR python.

Thanks