Lecture two
The xDSL framework

MLIR is great, but….

- The MLIR framework has grown massively in popularity since it was introduced in 2020, with numerous projects relying upon it
  - Clearly it solves an important problem
- However, there are several disadvantages/challenges
  - Steep learning curve due to it being written in C++ and Cmake
  - Requirement to use esoteric Tablegen DSL to define dialects
  - Need to download and build the entirety of LLVM (it's big!)
  - API is fairly complex and not particularly well documented
  - A moving target, with lots of changes happening

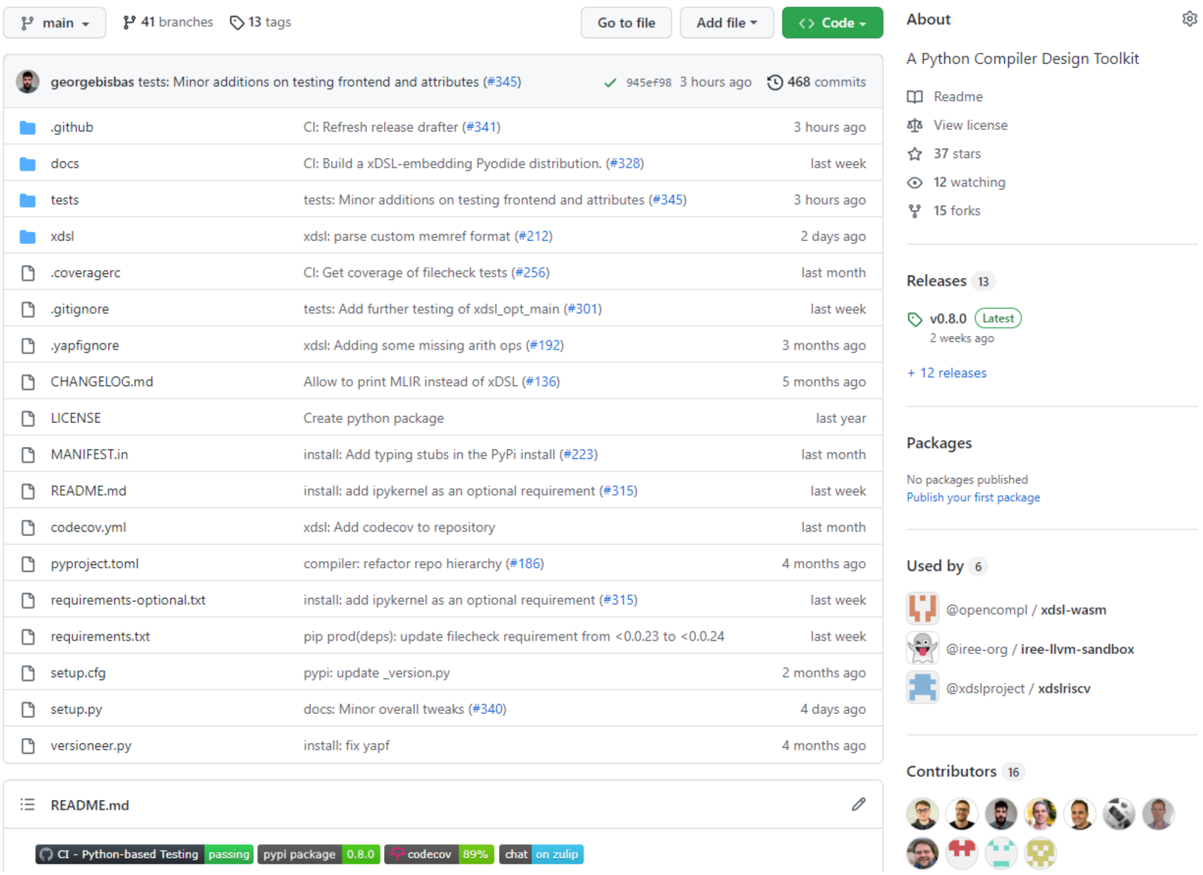
Step in xDSL……

- xDSL provides the ability to work with MLIR in Python
  - But crucially is far more than simply a Python wrapper!

- Contains the full MLIR functionality and concepts but with significant enhancements aimed at easing development of dialects and transformations
  - IRDL is our way of defining dialects, with tools to generate to/from Tablegen
    - Therefore, dialects are also written in Python and much more descriptive than in MLIR
  - Using a high productivity programming language one can easily experiment
  - No need to install LLVM, just need to grab the *xdsl* Python package from pip
  - Provides the standard MLIR dialects (and many others!)
  - Compatibility with MLIR, where output of MLIR can be consumed by xDSL and vica-versa

A burgeoning community

- BSD licenced and available at https://github.com/xdslproject/xdsl
- Alternatively, can install the latest release version via pip
  - Releases happen every few weeks
  - Currently version 0.8
- Lots of activity on the Github repository, and a chatty community on Zulip https://xdsl.zulipchat.com/



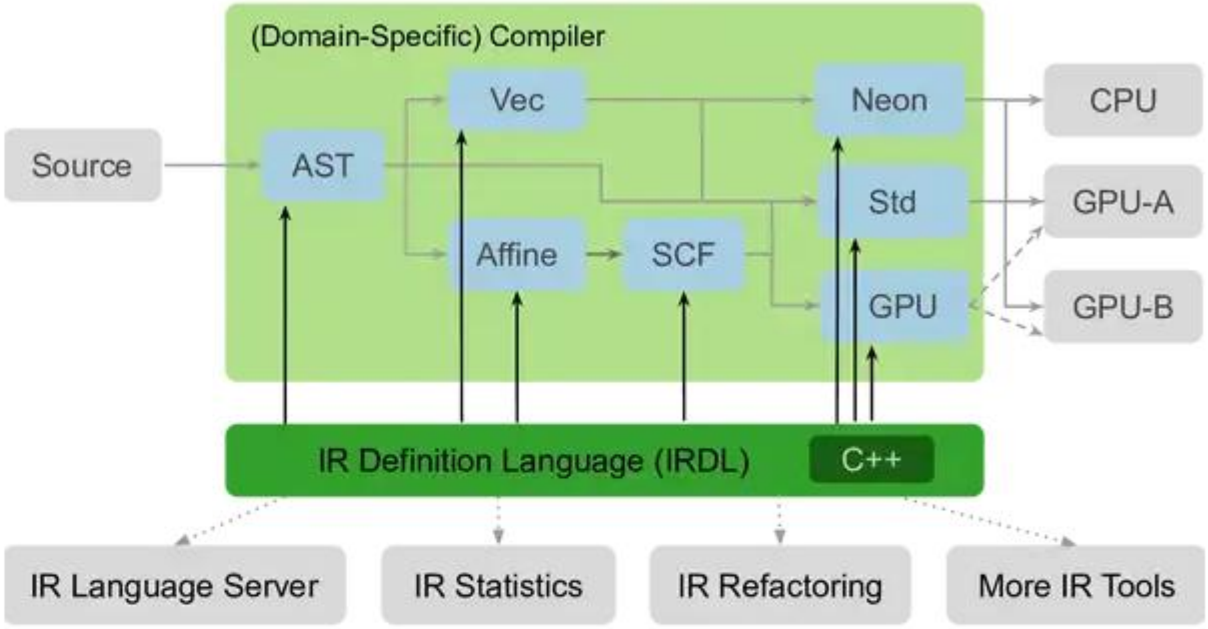
- Therefore, at the end of today’s session there is a community you can connect with and we will work with you help integrate xDSL with your code
  - We will share all the links at the end of the session

ExCALIBUR 10

xDSL

A key component: IRDL

- In MLIR one has to use the esoteric Tablegen to define dialects
- Instead, we provide IRDL which is a DSL for describing dialects using the same concepts (regions, blocks, operations) as MLIR but in a more straightforward manner



- Crucially, IRDL is fully compatible with MLIR via a C++ implementation, so the definition of dialects does not diverge from MLIR itself.
- We will not go into IRDL in detail, instead informally presenting this to you by looking at the Python code

If you are interested in more details then you can look at https://grosser.science/pub/fehr2022irdl and the associated paper

Defining a dialect

- Let's consider the IR node representing a function definition – this is the definition of the node from the hands-on exercise that we will look at soon

The name identifies this in the IR output

Decorator and inheritance defines the class to be an operator

```
@irdl_op_definition
class Function(Operation):
  name = "tiny_py.function"

  fn_name = AttributeDef(StringAttr)
  args = AttributeDef(ArrayAttr)
  return_var = AttributeDef(AnyAttr())
  body = SingleBlockRegionDef()

  @staticmethod
  def get(fn_name: Union[str, StringAttr],
          return_var,
          args: List[Operation],
          body: List[Operation],
          verify_op: bool = True) -> Routine:
      if return_var is None:
        return_var=EmptyToken()
      res = Function.build(attributes={"fn_name": fn_name, "return_var": return_var, "args": ArrayAttr.from_list(args)},
                           regions=[body])
      if verify_op:
          # We don't verify nested operations since they might have already been verified
          res.verify(verify_nested_ops=False)
      return res

  def verify_(self) -> None:
    pass
```

These are attributes associated with the IR node, you can see that the function name is a String, the arguments are an array and the return var is any type

This is a region (remember operations can nest regions)

Constructs the Function operation based on the parameters provided. Can also call *build* directly or the *create* which is similar

There is inbuilt verification to ensure that the operation has been correctly created

ExCALIBUR xDSL

Generating the IR

```python
def visit_FunctionDef(self, node):
    contents=[]
    for a in node.body:
        contents.append(self.visit(a))
    return tiny_py.Function.get(node.name, None, [], contents)
```

- Our, extremely simple, parser will create the IR node like so, where we provide it with an array representing the contents, and empty array for arguments and the name

```
builtin.module(){
  tiny_py.module() {
    tiny_py.function() ["fn_name" = "hello_world", "return_var" = !tiny_py.emptytoken, "args" = []] {
      ....
    }
  }
}
```

- Just as we have seen with MLIR in the previous lecture, the output generated from this is human readable
  - Can see we are using the EmptyToken to represent no return variable, this is defined within the dialect itself and the semantics of this is constrained within the dialects and aware transformations

The xDSL opt tool

- In the hands on exercise we will see the *tinypy-opt* tool which drives the transformations on IRs
  - These opt tools are standard in MLIR/LLVM, and xDSL provides Python infrastructure to make it trivial to write these for specific uses

```
user@login01:~$ ./tinypy-opt code.xdsl –t mlir
```

- We are instructing xDSL to parse *code.xdsl* contents & output it in MLIR format
  - This is different to the output on the previous slide, as that is in xDSL format (which we think is clearer). However, MLIR format is required to interoperate with MLIR.

```
"builtin.module"() ({
  "tiny_py.module"() ({
    "tiny_py.function"() ({
      ....
    }) {"fn_name" = "hello_world", "return_var" = #tiny_py.emptytoken, "args" = []} : () -> ()
  }) : () -> ()
}) : () -> ()
```

ExCALIBUR 10

xDSL

Undertaking transformations on the IR

- Transformations are written in Python and driven by the *opt* tool

```
user@login01:~$ ./tinypy-opt code.xdsl –p tiny-py-to-standard
```

- This transformation converts the tiny-py dialect to standard dialects
  - Here converting from tiny_py.function into func.func
  - This is an example of lowering from a dialect with a richer source of information to lower level standard dialects

```
builtin.module() {
  func.func() ["sym_name" = "hello_world", "function_type" = !fun<[], []>, "sym_visibility" = "public"] {
    ....
    func.return()
  }
}
```

ExCALIBUR
xDSL

Lifting the lid on transformations

Entry point of the transformation

```
def tiny_py_to_standard(ctx: MLContext, input_module: ModuleOp):
    res_module = translate_program(input_module)
    res_module.regions[0].move_blocks(input_module.regions[0])

def translate_fun_def(ctx: SSAValueCtx,
                      fn_def: tinypy.Function) -> Operation:
    routine_name = fn_def.attributes["fn_name"]

    body = Region()
    block = Block()

    c = SSAValueCtx(dictionary=dict(),
                    parent_scope=ctx)

    arg_types=[]
    arg_names=[]

    body_contents=[]
    for op in fn_def.body.blocks[0].ops:
      res=translate_def_or_stmt(c, op)
      if res is not None:
        body_contents.append(res)

    block.add_ops(flatten(body_contents))

    # A return is always needed at the end of the procedure
    block.add_op(func.Return.create())
    body.add_block(block)

    function_ir=func.FuncOp.from_region(routine_name, arg_types, [], body)
    function_ir.attributes["sym_visibility"]=StringAttr("public")

    return function_ir
```

Routine that translates the tiny_py function node to the standard func.func IR node

We create a Static Single Assignment (SSA) context to track variables so that they are private to the function

Visit all operations in the body of the function and translate these to the standard dialects

Add a return (required by MLIR)

Create func.func IR node

Manipulating the IR

- We also provide several ways of manipulating and rewriting the IR, these are based on MLIR standard approaches (but much easier in Python!)
  - This is registered with the opt tool and executed via *–p apply-my-analysis*

```
class ApplyRewriter(RewritePattern):
    @op_type_rewrite_pattern
    def match_and_rewrite(
            self, call_node: tiny_py.CallExpr, rewriter: PatternRewriter):

        block = call_node.parent

        idx = block.ops.index(call_node)

        call_node.detach()

        ...

        some_other_op = ....
        rewriter.insert_op_at_pos(some_other_op, block, idx)


def apply_my_analysis(ctx: psy_ir.MLContext, module: ModuleOp) -> ModuleOp:
    applyRewriter=ApplyRewriter()
    walker = PatternRewriteWalker(GreedyRewritePatternApplier([applyRewriter]), apply_recursively=False)
    walker.rewrite_module(module)

    return module
```

Will match (and execute this function) for all nodes in the IR that are of type *tiny_py.CallExpr*

Obtain the operation's parent block

Get the index of this node in the block's list of operations

Detach this node from the IR

Create some other operation, e.g. containing the *call_node* operation

Insert this new operation in the block at index *idx*

Entry point of the pass, we can apply many transformations here if we wish

- All these functions, and many more, to manipulate the IR are documented

Different levels of IR

```
builtin.module() {
  tiny_py.module() {
    tiny_py.function() ["fn_name" = "ex_two",
                        "return_var" = !empty, "args" = []] {
      tiny_py.assign() ["var_name" = "val"] {
        tiny_py.constant() ["value" = 0.0 : !f32]
      }
      tiny_py.assign() ["var_name" = "add_val"] {
        tiny_py.constant() ["value" = 88.2 : !f32]
      }
      tiny_py.loop() ["variable" = "a"] {
        tiny_py.constant() ["value" = 0 : !i32]
      } {
        tiny_py.constant() ["value" = 100000 : !i32]
      } {
        tiny_py.assign() ["var_name" = "val"] {
          tiny_py.binaryoperation() ["op" = "add"] {
            tiny_py.var() ["variable" = "val"]
          } {
            tiny_py.var() ["variable" = "add_val"]
          }
        }
      }
    }
  }
}
```

- Hierarchical
- Close representation to the original code
- Easy to reason about in transformation passes

Can transform between these, also possible to mix levels

```
"builtin.module"() ({
  "func.func"() ({
    %0 = "arith.constant"() {"value" = 0.0 : f32} : () -> f32
    %1 = "arith.constant"() {"value" = 88.2 : f32} : () -> f32
    %2 = "arith.constant"() {"value" = 0 : i32} : () -> i32
    %3 = "scf.while"(%2, %0) ({
      %4 = "arith.constant"() {"value" = 100000 : i32} : () -> i32
      %5 = "arith.cmpi"(%4, %2) {"predicate" = 1 : i64} : (i32, i32) -> i1
      "scf.condition"(%5, %2) : (i1, i32) -> ()
    }, {
    ^0(%6 : i32, %7 : f32):
      %8 = "arith.addf"(%0, %1) : (f32, f32) -> f32
      %9 = "arith.constant"() {"value" = 1 : i32} : () -> i32
      %10 = "arith.addi"(%2, %9) : (i32, i32) -> i32
      "scf.yield"(%10, %8) : (i32, f32) -> ()
    }) : (i32, f32) -> i32
    "func.return"() : () -> ()
  }) {"sym_name" = "ex_two", "function_type" = () -> (), "sym_visibility"
= "public"} : () -> ()
}) : () -> ()
```

- SSA form, much closer to concrete implementation
- Exposes lower level details and concerns
- More difficult to see how relates to original code

xDSL

Which level to run transformations on?

- It really depends on the transformation!
- Some will suit working on the higher level IR much more, for instance discovering parallelism or mapping code constructs to features such as stencils

```
builtin.module() {
  tiny_py.module() {
    tiny_py.function() ["fn_name" = "ex_two",
                        "return_var" = !empty, "args" = []] {
      tiny_py.assign() ["var_name" = "val"] {
        tiny_py.constant() ["value" = 0.0 : !f32]
      }
      tiny_py.assign() ["var_name" = "add_val"] {
        tiny_py.constant() ["value" = 88.2 : !f32]
      }
      tiny_py.loop() ["variable" = "a"] {
        tiny_py.constant() ["value" = 0 : !i32]
      } {
        tiny_py.constant() ["value" = 100000 : !i32]
      } {
        tiny_py.assign() ["var_name" = "val"] {
          tiny_py.binaryoperation() ["op" = "add"] {
            tiny_py.var() ["variable" = "val"]
          } {
            tiny_py.var() ["variable" = "add_val"]
          }
        }
      }
    }
  }
}
```

```
"builtin.module"() ({
  "func.func"() ({
    %0 = "arith.constant"() {"value" = 0.0 : f32} : () -> f32
    %1 = "arith.constant"() {"value" = 88.2 : f32} : () -> f32
    %2 = "arith.constant"() {"value" = 0 : i32} : () -> i32
    %3 = "scf.while"(%2, %0) ({
      %4 = "arith.constant"() {"value" = 100000 : i32} : () -> i32
      %5 = "arith.cmpi"(%4, %2) {"predicate" = 1 : i64} : (i32, i32) -> i1
      "scf.condition"(%5, %2) : (i1, i32) -> ()
    }, {
    ^0(%6 : i32, %7 : f32):
      %8 = "arith.addf"(%0, %1) : (f32, f32) -> f32
      %9 = "arith.constant"() {"value" = 1 : i32} : () -> i32
      %10 = "arith.addi"(%2, %9) : (i32, i32) -> i32
      "scf.yield"(%10, %8) : (i32, f32) -> ()
    }) : (i32, f32) -> i32
    "func.return"() : () -> ()
  }) {"sym_name" = "ex_two", "function_type" = () -> (), "sym_visibility"
= "public"} : () -> ()
}) : () -> ()
```

- Others suit working on the lower level IR, such as constant folding
- The point is xDSL/MLIR is very powerful and gives you the choice
  - Select what works best for your dialects and code!

In the hands-on exercises we have transformations that operate on both levels

Documentation

- In this lecture we have provided a general overview of how the different parts of xDSL fit together and can be used as part of the compilation flow
  - We will see these in the hands-on exercises and gain more experience with the different aspects

- However there is much more depth than we would want to cover in an introduction tutorial, at https://github.com/xdslproject/xdsl/tree/main/docs we have notebooks that you can explore afterwards to delve deeper
  - *irdl.ipynb* provides a deep exploration of how to define dialects and the different constructs and constraints that are available xDSL, for instance how to define bespoke types
  - *database_example.ipynb* walks you through the creation of a database DSL using xDSL alone (e.g. it never calls into MLIR or LLVM) that generates SQL for querying databases

Conclusions

- xDSL enables us to leverage MLIR constructs in a high productivity Python environment
- Don’t need MLIR or LLVM installed to use xDSL
  - Although will need it if you want to generate LLVM-IR and use the LLVM backends
- Lots of resources on xDSL that you can refer to
  - Website: https://xdsl.dev/
  - Zulip chat: https://xdsl.zulipchat.com/
  - Jupyter notebooks: https://github.com/xdslproject/xdsl/tree/main/docs
- A number of users of xDSL in the community
  - Devito
  - PSyclone