A practical introduction to MLIR for HPC using Python

Wash up from exercises and key takeaway points

ExCALIBUR

xDSL

Where we are

Time	Details	Type
8:30 - 8:35	Introduction, welcome and objectives	Presentation
8:35 - 8:55	An overview of MLIR and LLVM	Presentation
8:55 - 9:20	The xDSL framework	Presentation
9:20 - 9:25	Introduction to the hands-on activity	Presentation
9:25 - 10:00	Logging into ARCHER2 and hands-on practical activity	Hands-on
10:00 - 10:30	Morning break	
10:30 - 10:35	Welcome back and overview of second part	Presentation
10:35 - 11:45	Hands-on practical activity	Hands-on
11:45 - 11:55	Wash up from practical activities, highlighting key take-away points	Presentation
11:55 - 12:00	Conclusions & next steps to continue working with the technologies	Presentation

We wrapped up exercise one at the start of the second part, so here we focus on exercises two and three

Exercise two

- Extending our (very simple!) compiler to support loops

  1. *Support in the tiny py dialect for the loop construct*
  2. *Generate tiny py loop from python compiler*
  3. *Support in the tiny py to standard dialects lowering transformation to then lower this appropriately*

```
from python_compiler import python_compile

@python_compile
def ex_two():
    val=0.0
    add_val=88.2
    for a in range(0, 100000):
      val=val+add_val
    print(val)

ex_two()
```

We print the result at the end just so the compiler does not optimise the calculation out!

Remember, sample solutions are provided in the repository

Tiny py dialect support

The name of the loop variable is a string attribute

The from and to loop expressions are regions

Body of the loop is also a region

If a Python string is provide we wrap it in the xDSL/MLIR StringAttr

Builds the operation (note how we create a region and block that contains the operations

```
@irdl_op_definition
class Loop(IRDLOperation):
  name = "tiny_py.loop"

  variable: OpAttr[StringAttr]
  from_expr: Region
  to_expr: Region
  body: Region

  @staticmethod
  def get(variable: str | StringAttr,
          from_expr: Operation,
          to_expr: Operation,
          body: List[Operation],
          verify_op: bool = True) -> If:
      if isinstance(variable, str):
        # If variable is a string then wrap it in StringAttr
        variable=StringAttr(variable)

      res = Loop.build(attributes={"variable": variable}, regions=[Region([Block([from_expr])]),
        Region([Block([to_expr])]), Region([Block(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
```

This is what you needed to add, as we already provided the get method

ExCALIBUR

xDSL

Hooking this up….

The name of the loop variable is a string attribute

```
def visit_For(self, node):
  contents=[]

  for a in node.body:
    contents.append(self.visit(a))

  expr_from=self.visit(node.iter.args[0])
  expr_to=self.visit(node.iter.args[1])

  return tiny_py.Loop.get(node.target.id,
            expr_from, expr_to, contents)
```

```
tiny_py.module() {
  tiny_py.function() ["fn_name" = "ex_two",
        "return_var" = !empty, "args" = []] {
    ...
    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"]
        }
      }
    }
    ...
  }
}
```

The from and to loop expressions are regions

Body of the loop is also a region

ExCALIBUR

xDSL

Supporting this in the lowering

- We provided most of the lowering from the tiny py loop to the standard dialects, but there were a couple of missing parts that we walked you through completing
  - The sample solutions illustrate the completion of this, which should be obvious from the instructions (let us know if not!)

- We translate into the *for* operation of the *structured control flow* (*scf*) dialect
  - There is also a *control flow* (*cf*) dialect, and this contains building blocks such as branch and conditional branch.
  - A conditional branch is driven by a 1 bit integer (e.g. a Boolean), this is often generated by the *cmpi* and *cmpf* operations (compare integer, compare float) in the *arith* dialect
  - MLIR will lower *scf* into its *cf* counterpart, but means we work with higher level constructs

Lowering with MLIR

mlir-opt --pass-pipeline="builtin.module(loop-invariant-code-motion, convert-scf-to-cf, convert-cf-to-llvm{index-bitwidth=64}, convert-arith-to-llvm{index-bitwidth=64}, convert-func-to-llvm, reconcile-unrealized-casts)" ex_two.mlir

- The arguments provided to mlir-opt are quite a bit more complex than exercise one, but that's because we need to lower the loops.
  - Lowering transformation passes are in blue (see how we provide arguments to some of these)
  - Optimisation passes in red (this moves statements outside of the loop body where possible)
  - Instructions to MLIR in green (here to instruct MLIR to put in explicit operations for undertaking implicit data conversion)
- The result of this is then translated into LLVM-IR by *mlir-translate*

Exercise three

- Parallelisation of our loops

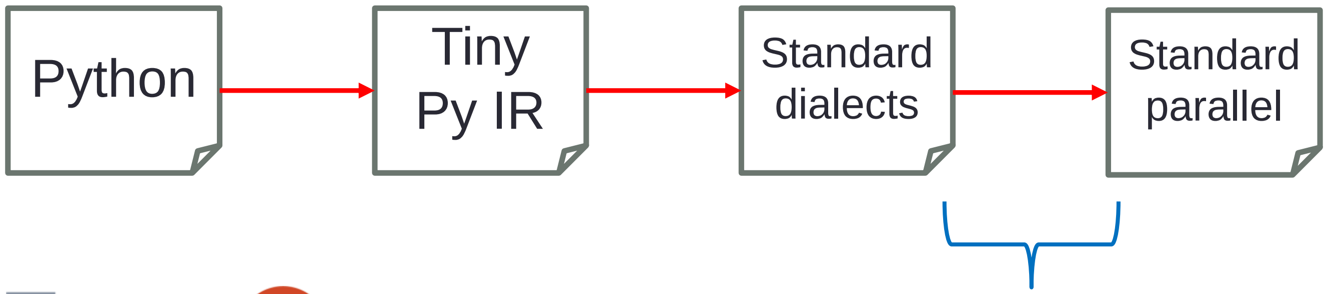
    1. *Exploit threaded parallelism*
    2. *Vectorise our calculations*

The user code remains unchanged, we are looking to apply an automatic transformation here to achieve these things

```
from python_compiler import python_compile

@python_compile
def ex_three():
    val=0.0
    add_val=88.2
    for a in range(0, 100000):
      val=val+add_val
    print(val)

ex_three()
```

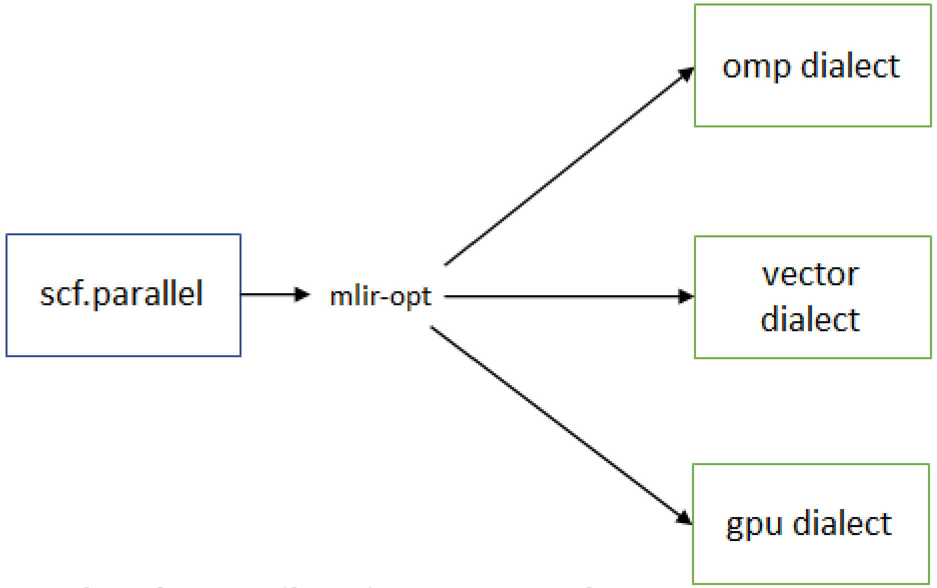


Adding this transformation

Remember, sample solutions are provided in the repository

Using scf.parallel

- We can leverage the *parallel* operation in the *scf* dialect
  - Which is an operation that represents a parallel loop



  - This really illustrates the benefit of MLIR, where we can use a high level construct such as this and then rely on existing transformations to lower it to a variety of targets

But this needs some thought…..

```
%7 = "scf.for"(%4, %5, %6, %0) ({
^0(%8 : index, %9 : f32):
  %10 = "arith.addf"(%9, %1) : (f32, f32) -> f32
  "scf.yield"(%10) : (f32) -> ()
}) : (index, index, index, f32) -> f32
```

```
%7 = "scf.parallel"(%4, %5, %6, %0) ({
^0(%8 : index, %9 : f32):
  %10 = "arith.addf"(%9, %1) : (f32, f32) -> f32
  "scf.yield"(%10) : (f32) -> ()
}) : (index, index, index, f32) -> f32
```

- Unfortunately it isn't as simple as swapping the *for* operation with *parallel*
  - Because we are updating the result of the *addf* operation each iteration (%10 which is yielded and becomes %9 in the next operation)
  - This adds a loop carried dependency that must be considered when parallelising the loop, as otherwise the result will be incorrect
  - The solution is to wrap the *arith.addf* operation in the *reduce* operation of the *scf* dialect
    - Which informs MLIR of this relationship

The transformation

```
%7 = "scf.parallel"(%4, %5, %6, %0) ({
^0(%8 : index):
  "scf.reduce"(%1) ({
    ^1(%lhs : f32, %rhs : f32):
      %11 = "arith.addf"(%lhs, %rhs) : (f32, f32) -> f32
      "scf.reduce.return"(%11) : (f32) -> ()
    }) : (f32) -> ()
  "scf.yield"() : () -> ()
}) {"operand_segment_sizes" = array<i32: 1, 1, 1, 1>} :
         (index, index, index, f32) -> f32
```

scf.parallel enables loops to be fused, where an arbitrary number of for and to bounds can be specified, along with their individual steps

Known as varadic operands and we need to specify the number of these so MLIR can interpret them (here have one loop with one input SSA value)

- Most of this pass is already provided, with you needing to fill in some missing parts as per the exercise sheet
  - See the sample solutions and ask us if this is unclear at all
- You can see how we are wrapping the *arith.addf* in the *scf.reduce* operation now, and also *scf.reduce.return* to return the result to the next iteration

Lowering with MLIR

mlir-opt --pass-pipeline="builtin.module(loop-invariant-code-motion, **convert-scf-to-openmp**, convert-scf-to-cf, convert-cf-to-llvm{index-bitwidth=64}, convert-arith-to-llvm{index-bitwidth=64}, **convert-openmp-to-llvm**, convert-func-to-llvm, reconcile-unrealized-casts)" ex_two.mlir

- Lowering transformation passes are in blue
  - Lowering scf to the *omp* dialect to add in threaded parallelism
  - Then lowering the *omp* dialect to the *llvm* dialect
- Optimisation passes in red (this moves statements outside of the loop body where possible)
- Instructions to MLIR in green (here to instruct MLIR to put in explicit operations for undertaking implicit data conversion)

scf-parallel-loop-specialization will lower to the vector dialect, can also lower to the *gpu* dialect (see the exercise sheet for how to do this)