SOLVING COMPILER PUZZLES

Debug Methods in MLIR

CHRISTOPHER MCGIRR

roofline

COMPILER DESIGN FOR ML DEVELOPERS WITH LIMITED MLIR EXPERIENCE

What is this talk about?

- For developers who may not have had deep experience using the MLIR framework
- Focus on ML or Tensor Compilers
- Compiler inputs are non-source languages
- Inputs often contain large constants

Disclaimers

- A practical guide
- Not exhaustive or definitive
- Opinionated based on experience

DEBUGGING AI MODELS IN MLIR IS A PAIN

Case Example

AI Model

meta-llama/Llama-3.2-1B (f32)
~1000 operations that could be of interest
~4.7GB in byte code
~9.3GB in assembly form

linalg.batch_matmul	145
linalg.generic	906
linalg.transpose	193

Approach

Finding the root cause of a bug in an ML Compiler

1. Many different levels of abstraction (dialects)
   - Frontend: onnx-mlir, torch-mlir, TOSA
   - Middle-End: linalg, vector, memref
   - Backend: PTX, SPIRV, LLVM
2. Dealing with large files
3. Pass phase ordering
4. Shape Propagation
5. Pattern Matching

DEBUG METHODS

TODAY, LET’S FOCUS ON FIVE KEY DEBUG METHOS

1 IR Printing Mechanisms

2 Useful MLIR-Opt Arguments

3 Leveraging MLIR Reproducers

4 LLDB Integration

5 MLIR Reduce

FIRST STEP IS LOOKING AT THE IR

Advantages

- Easiest method to see how IR changes after every pass
- Finding where an operation changes

Disadvantages

- Lacks the arguments provided to the pass
- After-failure: does not work if you hit an assert
- Manageable with only the smallest of reproducers

Code and Insights

```
// -----// IR Dump After Canonicalizer (canonicalize) //----- //
// -----// IR Dump After ConvertElementwiseToLinalgPass (convert-elementwise-to-linalg) //----- //
// -----// IR Dump After ConvertLinalgToLoopsPass (convert-linalg-to-loops) //----- //
// -----// IR Dump After CSE (cse) //----- //
// -----// IR Dump After SCFToControlFlow (convert-scf-to-cf) //----- //
```

Arguments for printing around passes:

```
--mlir-print-ir-after=<pass-arg>
--mlir-print-ir-after-all
--mlir-print-ir-after-change
--mlir-print-ir-after-failure
--mlir-print-ir-before=<pass-arg>
--mlir-print-ir-before-all
```

ELIDING ARGUMENTS

Advantages

✓ Extremely useful when debugging large files

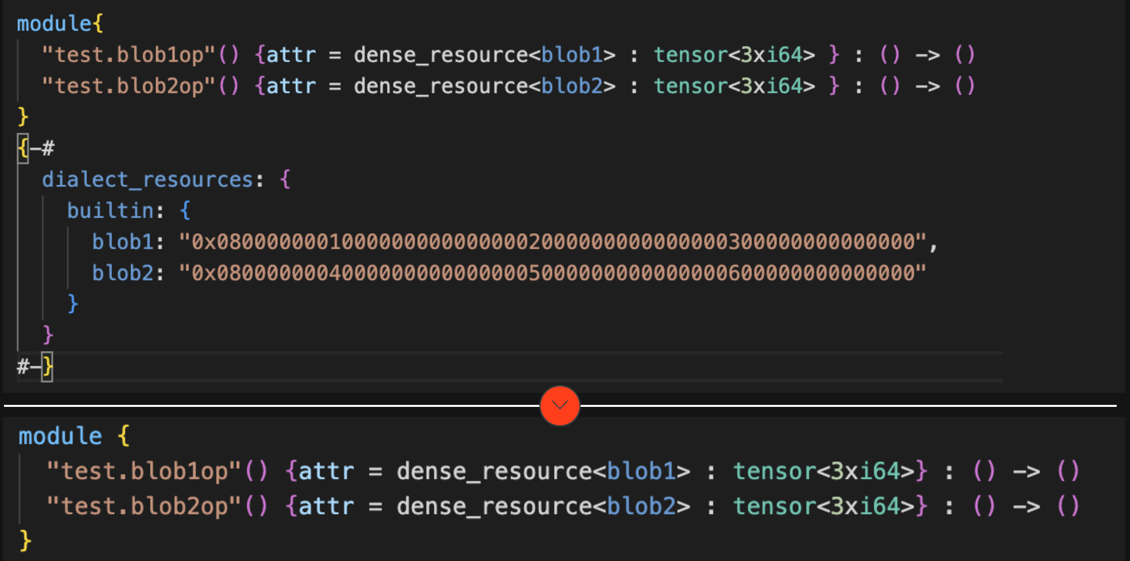
Disadvantages

✕ Constants are still in-lined in the IR for tensor compilers

✕ The only way to work with large models

Code and Insights

Eliding the constants replaces them with an external reference



```
module{
  "test.blob1op"() {attr = dense_resource<blob1> : tensor<3xi64> } : () -> ()
  "test.blob2op"() {attr = dense_resource<blob2> : tensor<3xi64> } : () -> ()
}
{-#
  dialect_resources: {
    builtin: {
      blob1: "0x08000000010000000000000002000000000000000300000000000000",
      blob2: "0x08000000040000000000000005000000000000000600000000000000"
    }
  }
#-}
```

```
module {
  "test.blob1op"() {attr = dense_resource<blob1> : tensor<3xi64>} : () -> ()
  "test.blob2op"() {attr = dense_resource<blob2> : tensor<3xi64>} : () -> ()
}
```

Arguments for printing around passes:

```
--mlir-elide-elementsattrs-if-larger=<uint>
--mlir-elide-resource-strings-if-larger=<uint>
```

OTHER NOTABLE ARGUMENTS

1. --mlir-print-ir-tree-dir=<string>

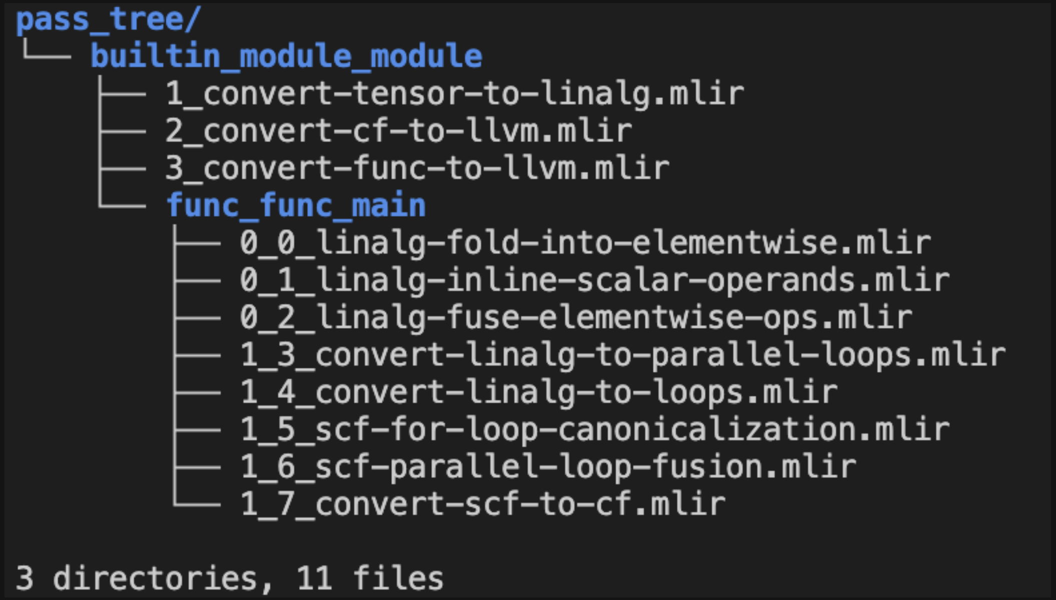
   Can be used to print IR to file after each pass in a directory structure that matches the pass manager nesting

2. --mlir-print-ir-module-scope

   Useful if your ML compiler deals with multiple, nested functions that have information attached to operations or regions in the parent level

3. --mlir-print-local-scope

   Prints all operations above the selected operation without the IsolatedFromAboveTrait. [1]



```
pass_tree/
└── builtin_module_module
    ├── 1_convert-tensor-to-linalg.mlir
    ├── 2_convert-cf-to-llvm.mlir
    ├── 3_convert-func-to-llvm.mlir
    └── func_func_main
        ├── 0_0_linalg-fold-into-elementwise.mlir
        ├── 0_1_linalg-inline-scalar-operands.mlir
        ├── 0_2_linalg-fuse-elementwise-ops.mlir
        ├── 1_3_convert-linalg-to-parallel-loops.mlir
        ├── 1_4_convert-linalg-to-loops.mlir
        ├── 1_5_scf-for-loop-canonicalization.mlir
        ├── 1_6_scf-parallel-loop-fusion.mlir
        └── 1_7_convert-scf-to-cf.mlir

3 directories, 11 files
```

[1] https://github.com/llvm/llvm-project/blob/main/mlir/lib/IR/AsmPrinter.cpp#L4065

NARROWING THE SCOPE

Advantages

- ✓ Useful for debugging pattern matching errors
- ✓ Some passes have great debugging messages

Disadvantages

- ✗ Prints ever pattern that did or did not match
- ✗ Only available in builds with Debug [1]
- ✗ Requires very small IR example which is good for single pass debugging

Code and Insights

Piping to a log file is very useful here

```
mlir-opt:
                --debug
                --debug-only=<pass>

//===-------------------------------------------===//
Processing operation : 'linalg.fill'(0x616758f17990) {

  * Pattern FoldTensorCastProducerOp : 'linalg.fill -> ()' {
Trying to match "FoldTensorCastProducerOp"
"FoldTensorCastProducerOp" result 0
  } -> failure : pattern failed to match

//===-------------------------------------------===//
```

Make sure to add these lines to your code

```
#define DEBUG_TYPE ”my-pass-name”
LLVM_DEBUG({
  llvm::dbgs() << ”My Debug Message\n";
});
```

[1] https://github.com/llvm/llvm-project/blob/6257621f41d1deb31cfbfcee993a75991a0bca13/llvm/lib/Support/Debug.cpp

REPLAY COMPILER PASSES

Advantages

- ✓ Great mechanism to generate IR after failures (excluding asserts)
- ✓ Great for reporting bugs to up-stream if IR is small enough

Disadvantages

- ✗ Each pipeline pass must support full CLI serialization of its options
- ✗ Potential odd behavior with the nested pass manager
- ✗ Requires multi-threading to be disabled when generating local reproducer

Code and Insights

MLIR-Pot Argument

```
--mlir-generate-reproducer=<filename>
--mlir-pass-pipeline-crash-reproducer=<string>
--mlir-pass-pipeline-local-reproducer
--run-reproducer
```

Stored as an external resource outside of the Module

```
{-#
  dialect_resources: {
    builtin: {}
  },
  external_resources: {
    mlir_reproducer: {
      pipeline: "builtin.module(...)"
      disable_threading: false,
      verify_each: true
    }
  }
#-}
```

Link to Implementation

PRETTY PRINTING

Advantages

- ✓ Works great with simple types: (SmallVector<int>)
- ✓ LLDB terminal is very useful here with “expr op->dump()”

Disadvantages

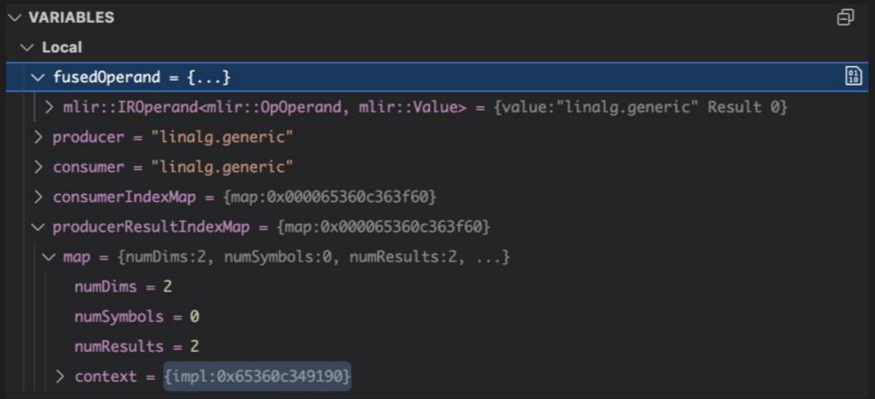
- ✗ Complex types are not resolved to a human-readable string
- ✗ Room for improvement with those scripts

Code and Insights

VSCode Debug Configuration

```
{
 . . .
 "initCommands": [
   "settings set target.disable-aslr false",
   "command script import ${workspaceFolder}/llvm/utils/lldbDataFormatters.py",
   "command script import ${workspaceFolder}/mlir/utils/lldb-
scripts/mlirDataFormatters.py",
 ]
}
```

VSCode Visualization



```
∨ VARIABLES
  ∨ Local
    ∨ fusedOperand = {...}
      > mlir::IROperand<mlir::OpOperand, mlir::Value> = {value:"linalg.generic" Result 0}
    > producer = "linalg.generic"
    > consumer = "linalg.generic"
    > consumerIndexMap = {map:0x000065360c363f60}
    ∨ producerResultIndexMap = {map:0x000065360c363f60}
      ∨ map = {numDims:2, numSymbols:0, numResults:2, ...}
          numDims = 2
          numSymbols = 0
          numResults = 2
        > context = {impl:0x65360c349190}
```

Pretty Printers Scripts: 1. Link 2. Link

ACTION DEBUGGING

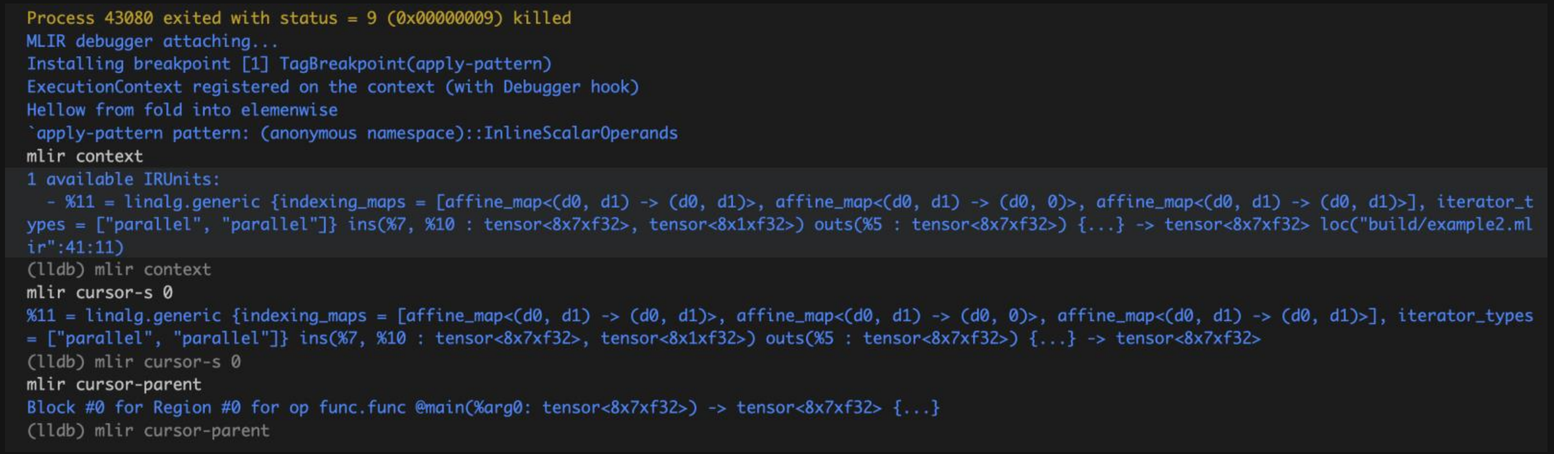
Advantages

- ✓ Action Debugging works great VSCode
- ✓ mlir break-on-tag: Useful for stop on patterns or pass executions
- ✓ mlir cursor-*: Navigate the IR at the current frame

Disadvantages

- ✕ Do not forget to add:
  --mlir-enable-debugger-hook [1]

Code and Insights



```
Process 43080 exited with status = 9 (0x00000009) killed
MLIR debugger attaching...
Installing breakpoint [1] TagBreakpoint(apply-pattern)
ExecutionContext registered on the context (with Debugger hook)
Hellow from fold into elemenwise
`apply-pattern pattern: (anonymous namespace)::InlineScalarOperands
mlir context
1 available IRUnits:
  - %11 = linalg.generic {indexing_maps = [affine_map<(d0, d1) -> (d0, d1)>, affine_map<(d0, d1) -> (d0, 0)>, affine_map<(d0, d1) -> (d0, d1)>], iterator_t
ypes = ["parallel", "parallel"]} ins(%7, %10 : tensor<8x7xf32>, tensor<8x1xf32>) outs(%5 : tensor<8x7xf32>) {...} -> tensor<8x7xf32> loc("build/example2.ml
ir":41:11)
(lldb) mlir context
mlir cursor-s 0
%11 = linalg.generic {indexing_maps = [affine_map<(d0, d1) -> (d0, d1)>, affine_map<(d0, d1) -> (d0, 0)>, affine_map<(d0, d1) -> (d0, d1)>], iterator_types
= ["parallel", "parallel"]} ins(%7, %10 : tensor<8x7xf32>, tensor<8x1xf32>) outs(%5 : tensor<8x7xf32>) {...} -> tensor<8x7xf32>
(lldb) mlir cursor-s 0
mlir cursor-parent
Block #0 for Region #0 for op func.func @main(%arg0: tensor<8x7xf32>) -> tensor<8x7xf32> {...}
(lldb) mlir cursor-parent
```

Useful Links:

1. Link to ODS Meeting Video
2. Documentation
3. Debugger Impl Source

[1] Link to debugging script

REDUCING THE PROBLEM SIZE

Advantages

- ✓ Enables automated bug reporting
- ✓ Well-documented usage instructions
- ✓ Implement your own reduction patterns

Additional Resources

MLIR Workshop (EuroLLVM2024)
MLIR-Reduce
IREE-Reduce
Circt-Reduce
LLVM-Reduce

Disadvantages

- ✕ Reducing real-world models involves searching a large design space
- ✕ Advanced knowledge is required.
- ✕ Manual reduction is significantly faster
- ✕ Downstream projects depend on customized versions of mlir-reduce

Further Reading

- Great Documentation for how to use
- Implement your own reduction patterns

CONCLUSION

FOUR KEY TAKEAWAYS ON COMPILER DEBUGGING IN MLIR

- Extensive toolkit already in place
- Still room for improvement – which presents opportunity
- LIT tests provide valuable insight into pass behavior
- Reading the code helps to understand how you can use the tools available to debug

Q&A

ANY QUESTIONS? I AM HAPPY TO CONNECT!

Lead Engineer
Christopher McGirr

mcgirr@roofline.ai

Christopher McGirr

Github Profile

www.roofline.ai