Tutorial: Create an MLIR Pass for Triton

Haishan Zhu Meta

In this tutorial..

- Provide a taste of MLIR and its usage in the context of Triton compilation

- Assume
  - No compiler background
  - Some Triton experience

Agenda

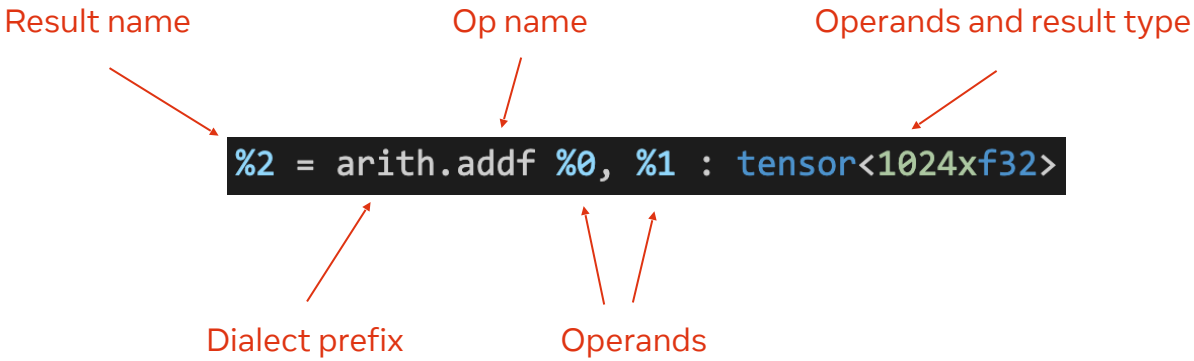
- MLIR Background
- Triton Kernel and ttir
- Create an MLIR Pass for Triton
  - Overview
  - Define a New Dialect
  - Define a New Pass
  - Putting It All Together
  - Other Considerations

MLIR Background

- Multi-Level Intermediate Representation
  - Started by Chris Lattner, Mehdi Amini, among others at Google
  - MLIR: A Compiler Infrastructure for the End of Moore's Law (2020 paper)
  - Goal: a reusable compiler infrastructure generally applicable to many use cases

- A framework to build compilers
  - Comes with existing dialects, types, transformations, analysis, etc.
  - Extensible with custom dialects, types, transformations, analysis, etc.
  - Pass infrastructure, diagnostics, testing tools
  - Easy integration into LLVM backend compiler for low-level code generation

IR Structure at a Glance



- MLIR uses the notion of operations
- Operations are “opaque functions” to MLIR
- IR is represented in SSA form
- Refer to these slides for more comprehensive examples

IR Structure at a Glance

- Dialects
  - Collection of operations, types, attributes, related transformation passes, analysis, etc.
  - Multiple dialects can co-exist within one module
  - E.g.: arith, math, func, llvm, memref, etc.
- Types
  - Supports both builtin and custom Dialect types
  - E.g.: i8, f32, memref, vector, tensor, etc.
- Attributes
  - In-place constant data on operations
  - E.g., comparison predicate, if perform boundary check on triton load, etc.

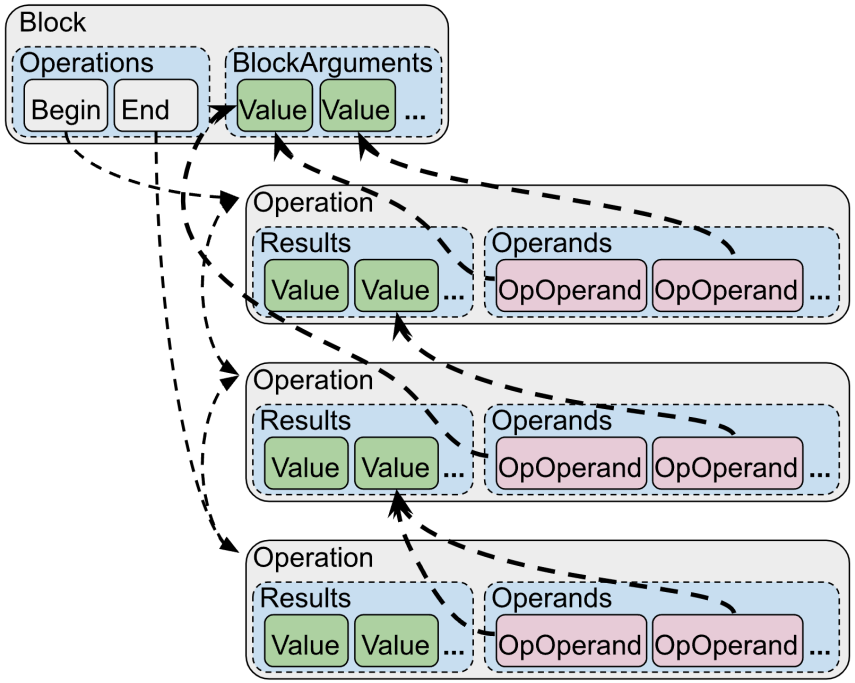
IR Structure at a Glance

Recursively Nested IR Structure

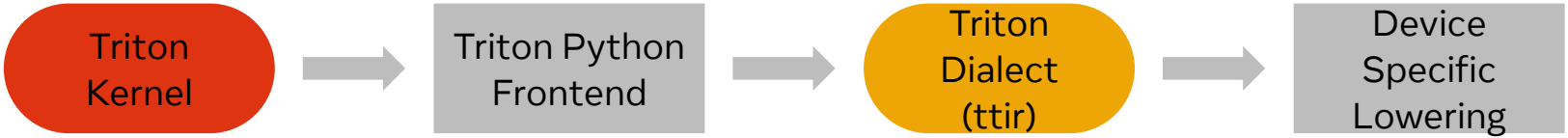
- Operation -> Regions -> Blocks -> Operations -> …
- E.g., a Triton reduction op can have multiple operations in its body

Navigate IR Structure

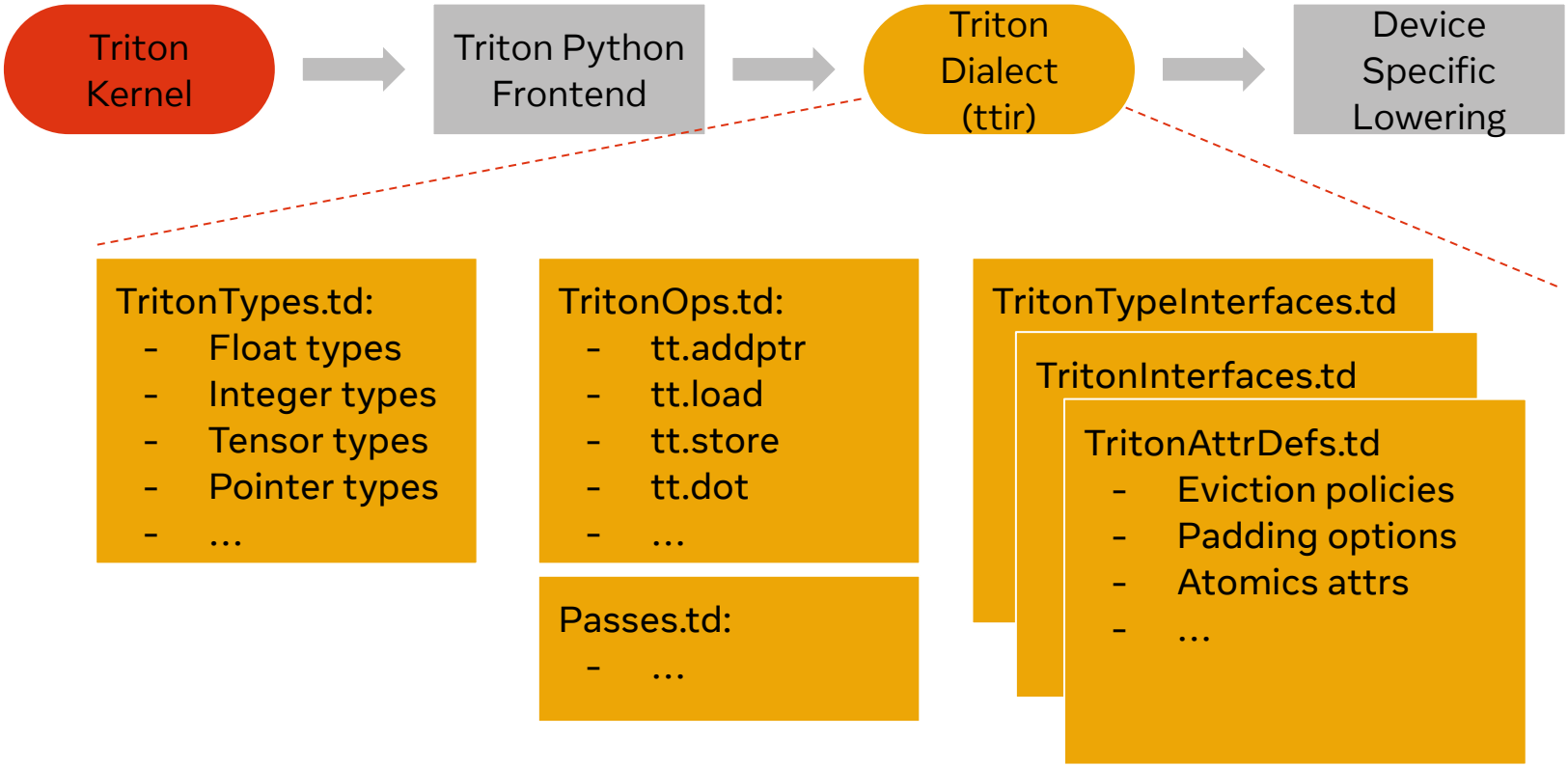
- MLIR provides mechanism to walk the recursive IR
- SSA nature of MLIR allows easy traversal of def-use chains



Triton Dialect



Triton Dialect



Triton Kernel and ttir

```
@triton.jit
def add_kernel(x_ptr, y, output_ptr, BLOCK_SIZE: tl.constexpr):
    pid = tl.program_id(axis=0)
    block_start = pid * BLOCK_SIZE
    offsets = block_start + tl.arange(0, BLOCK_SIZE)
    x = tl.load(x_ptr + offsets)
    output = x + y
    tl.store(output_ptr + offsets, output)
```

Triton Kernel and ttir

```
tt.func public @add_kernel_01234(%arg0: !tt.ptr<f32>, %arg1: f32, %arg2:
!tt.ptr<f32>) {
  %c1024_i32 = arith.constant 1024 : i32
  %0 = tt.get_program_id x : i32
  %1 = arith.muli %0, %c1024_i32 : i32
  %2 = tt.make_range {end = 1024 : i32, start = 0 : i32} : tensor<1024xi32>
  %3 = tt.splat %1 : i32 -> tensor<1024xi32>
  %4 = arith.addi %3, %2 : tensor<1024xi32>
  %5 = tt.splat %arg0 : !tt.ptr<f32> -> tensor<1024x!tt.ptr<f32>>
  %6 = tt.addptr %5, %4 : tensor<1024x!tt.ptr<f32>>, tensor<1024xi32>
  %7 = tt.load %6 : tensor<1024x!tt.ptr<f32>>
  %8 = tt.splat %arg1 : f32 -> tensor<1024xf32>
  %9 = arith.addf %7, %8 : tensor<1024xf32>
  %10 = tt.splat %arg2 : !tt.ptr<f32> -> tensor<1024x!tt.ptr<f32>>
  %11 = tt.addptr %10, %4 : tensor<1024x!tt.ptr<f32>>, tensor<1024xi32>
  tt.store %11, %9 : tensor<1024x!tt.ptr<f32>>
  tt.return
}
```

Create an MLIR Pass for Triton Dialect

- Replace tensor-tensor operation with tensor-scalar operation when possible

```
%8 = tt.splat %arg1 : f32 -> tensor<1024xf32>
%9 = arith.addf %7, %8 : tensor<1024xf32>
```

```
%9 = myarith.add_ts %7, %arg1 : tensor<1024xf32>, f32 -> tensor<1024xf32>
```

- Why
  - Many hardware supports vector-scalar operations natively, yet arith Dialect does not provide native support
  - Fewer instructions to execute
  - Avoid materializing the tensor of splatted scalar
- In this example
  - Define a custom myarith Dialect and a custom add_ts operation
  - Define a simple pass to rewrite splat-add operation sequence with the new add_ts

Define MyArithDialect

- We need to define a new Dialect before defining new operations
  - Dialect groups related operations, types, and attributes in a single namespace
  - New dialects can be defined in C++ by inheriting from mlir::Dialect or with tablegen

```
// MyArithDialect.td
def MyArith_Dialect : Dialect {
  let name = "myarith";

  let cppNamespace = "mlir::triton::myarith";

  let description = [{ My Arith Dialect. }];
}
```

Define myarith.add_ts Operation

- Define custom operation to model tensor-scalar add

```
// MyArithOps.td
class MyArith_Op<string mnemonic, list<Trait> traits = []> :
    Op<MyArith_Dialect, mnemonic, traits>;

def MyArith_AddTensorScalarOp : MyArith_Op<"add_ts"> {
  let summary = "perform tensor-scalar add.";

  let arguments = (ins RankedTensorOf<[AnyFloat, AnyInteger]> : $tensor,
                       AnyTypeOf<[AnyFloat, AnyInteger]> : $scalar);

  let results = (outs RankedTensorOf<[AnyFloat, AnyInteger]> : $result);

  let assemblyFormat = "operands attr-dict `:` type(operands) `->` type($result)";
}
```

Generated Code For Operation Definition

```
class AddTensorScalarOp : public ::mlir::Op<AddTensorScalarOp,
::mlir::OpTrait::ZeroRegions, ::mlir::OpTrait::OneResult,
::mlir::OpTrait::OneTypedResult<::mlir::RankedTensorType>::Impl,
::mlir::OpTrait::ZeroSuccessors, ::mlir::OpTrait::NOperands<2>::Impl,
::mlir::OpTrait::OpInvariants> {
```

```
::mlir::TypedValue<::mlir::RankedTensorType> getTensor() {
  return ::llvm::cast<::mlir::TypedValue<::mlir::RankedTensorType>>(*getODSOperands(0).
  begin());
}

::mlir::Value getScalar() {
  return ::llvm::cast<::mlir::Value>(*getODSOperands(1).begin());
}
```

```
static void build(::mlir::OpBuilder &odsBuilder, ::mlir::OperationState &odsState,
::mlir::Type result, ::mlir::Value tensor, ::mlir::Value scalar);
```

Declare convert-triton-to-myarith Pass

- We use tablegen to declare the pass

```
// Passes.td
def ConvertTritonToMyArith: Pass<"convert-triton-to-myarith", "mlir::ModuleOp"> {
    let summary = "Convert Triton to MyArith";
    let dependentDialects = ["mlir::triton::myarith::MyArithDialect"];
}
```

Define convert-triton-to-myarith Pass

- Leverage tablegen generated boilerplate code

```
#define GEN_PASS_DEF_CONVERTTRITONTOMYARITH
#include "triton/Conversion/TritonToMyArith/Passes.h.inc"
```

- The main logic walks the IR to manually pattern match and rewrite

```
// TritonToMyArithPass.cpp
struct ConvertTritonToMyArith
    : public impl::ConvertTritonToMyArithBase<ConvertTritonToMyArith> {
  using ConvertTritonToMyArithBase::ConvertTritonToMyArithBase;
  void runOnOperation() override {
    auto mod = getOperation();
    mod->walk([&](triton::FuncOp func) {
      rewriteSplatAddOp(func);
    });
  }
```

Declare convert-triton-to-myarith Pass

- Manually match the pattern of tt.splat followed by arith.addf

```
void rewriteSplatAddOp(triton::FuncOp func) {
  func->walk([&](triton::SplatOp splatOp) {
    auto src = splatOp.getSrc();
    auto res = splatOp.getResult();

    for (auto user : res.getUsers()) {
      if (auto addOp = dyn_cast<arith::AddFOp>(user)) {
        rewriteAddFOp(addOp, res, src);
      }
    }
  });
}
```

Declare convert-triton-to-myarith Pass

- Create new tensor-scalar operations and replace uses of original op

```
void rewriteAddFOp(arith::AddFOp addFOp, Value tensorSrc, Value scalarSrc) {
  OpBuilder b(addFOp);

  auto tensorOpnd =
      addFOp.getLhs() == tensorSrc ? addFOp.getRhs() : addFOp.getLhs();
  auto newOp = b.create<myarith::AddTensorScalarOp>(
      addFOp.getLoc(), tensorOpnd.getType(), tensorOpnd, scalarSrc);

  addFOp->replaceAllUsesWith(newOp);
};
```

Putting It All Together

- Recap
  - Defined new a dialect and a new operation
  - Created a new pass to manually pattern match splat-add sequence and rewrite them with newly defined operation

- Other changes required
  - Register corresponding dialects and passes in triton-opt (or custom pass driver)
  - CMake changes to build new files
  - Patch to enable this example

- See custom pass in action
  - Apply patch above to oss Triton, set up as usual
  - pip install -e python
  - triton-opt --convert-triton-to-myarith ./test.mlir > test.out0.mlir
  - Canonicalize the output to remove values with no uses
    - triton-opt --convert-triton-to-myarith --canonicalize ./test.mlir > test.out1.mlir

Other Considerations

- Define verifiers, concise assembly format, etc.
- Specify correct side effect and interfaces
- Use pattern rewriter
- Use tablegen to create rule-based pattern matching code
- Canonicalize within the pass
- Deciding best place to perform such transformation in the pipeline
- Extend such rewrite for pointer arithmetics

Reference

- MLIR official documentation
  - MLIR Language Reference
  - MLIR introduction slides and presentation
  - MLIR tutorial

Thank you!