How to bring your Neural Network into Upstream MLIR Dialects

A Practical Guide for Beginners

MAXIMILIAN BARTEL

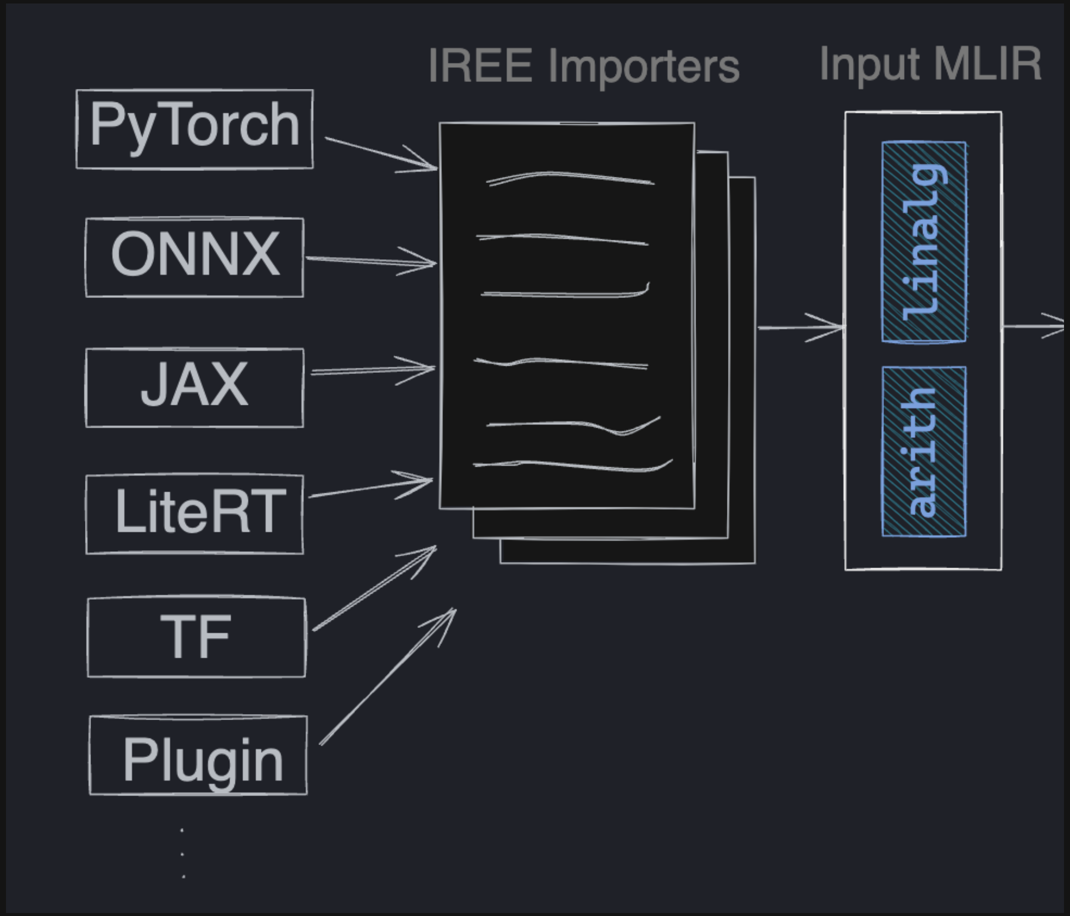
roofline

AGENDA

roofline

BEGINNERS OFTEN STRUGGLE TO GET THEIR MODEL INTO UPSTREAM MLIR DIALECTS

- Tensor compiler is a big part of upstream MLIR
- Newcomers want to get their model into upstream MLIR to play around with passes and transformations
- The importing is not part of upstream MLIR



HOW TO CONVERT YOUR MODEL TO THE LINALG OR TOSA – FRAMEWORK BY FRAMEWORK

- The projects which you can use to get your model into upstream MLIR
- Where to find the code and examples
- Tips and tricks on how to use them
- My personal experiences with the projects

QUICK EXAMPLES OF LINALG AND TOSA

Linalg Dialect

- Two flavors:
  - Named ops
  - Generic ops
- Main Codegen dialect in MLIR
- Frameworks try to lower to named ops to preserve information

TOSA Dialect

- Tensor Operator Set Architecture
- Minimal and stable set of tensor-level operators
- Only ML graph dialect upstream
- It is the output format for some accelerators

Code and Insights

```
func.func @matmul_divisible(%A: tensor<1024x1024xf32>,
                            %B: tensor<1024x1024xf32>,
                            %C: tensor<1024x1024xf32>)
    -> tensor<1024x1024xf32>
{
  %cst = arith.constant 0.000000e+00 : f32
  %0 = linalg.fill ins(%cst : f32)
                   outs(%C : tensor<1024x1024xf32>)
      -> tensor<1024x1024xf32>
  %1 = linalg.matmul ins(%A, %B : tensor<1024x1024xf32>, tensor<1024x1024xf32>)
                     outs(%0 : tensor<1024x1024xf32>)
      -> tensor<1024x1024xf32>
  return %1 : tensor<1024x1024xf32>
}
```

```
func.func @test_erf(%arg0: tensor<13x21x3xf32>) -> tensor<13x21x3xf32> {
  %0 = tosa.erf %arg0 : (tensor<13x21x3xf32>) -> tensor<13x21x3xf32>
  return %0 : tensor<13x21x3xf32>
}
```


https://github.com/llvm/llvm-project/blob/d1fd97737e9064431983e48c2105d1c54d5feacc/mlir/test/Dialect/Linalg/matmul-shared-memory-padding.mlir, https://github.com/llvm/llvm-project/blob/d1fd97737e9064431983e48c2105d1c54d5feacc/mlir/test/Dialect/Tosa/ops.mlir

LITERT (FORMELY TENSORFLOW LITE)

Overview

- Lightweight ML runtime optimized for mobile/edge devices
- Flatbuffer file format (.tflite) with small footprint
- Supports quantization and model optimization
- Key operators for CNN/RNN architectures
- Used widely in Android, iOS, embedded Linux devices

LOWERING LITERT TO TOSA VIA TENSORFLOW

Steps

TFLite Flatbuffer → TFLite Dialect → TOSA Dialect

Code and Insights

flatbuffer_translate --tflite-flatbuffer-to-mlir model.tflite -o – | tf-opt --tfl-to-tosa-pipeline

```
%350 = tosa.add %345, %349 : (tensor<?x1xi32>, tensor<?x1xi32>) -> tensor<?x1xi32>
%351 = tosa.mul %350, %343 {shift = 31 : i8} : (tensor<?x1xi32>, tensor<?x1xi32>) -> tensor<?x1xi32>
%352 = tosa.sub %19, %351 : (tensor<1x1xi32>, tensor<?x1xi32>) -> tensor<?x1xi32>
%353 = tosa.mul %350, %352 {shift = 31 : i8} : (tensor<?x1xi32>, tensor<?x1xi32>) -> tensor<?x1xi32>
%354 = tosa.mul %353, %18 {shift = 0 : i8} : (tensor<?x1xi32>, tensor<1x1xi32>) -> tensor<?x1xi32>
%355 = tosa.add %350, %354 : (tensor<?x1xi32>, tensor<?x1xi32>) -> tensor<?x1xi32>
%356 = tosa.mul %355, %343 {shift = 31 : i8} : (tensor<?x1xi32>, tensor<?x1xi32>) -> tensor<?x1xi32>
%357 = tosa.sub %19, %356 : (tensor<1x1xi32>, tensor<?x1xi32>) -> tensor<?x1xi32>
%358 = tosa.mul %355, %357 {shift = 31 : i8} : (tensor<?x1xi32>, tensor<?x1xi32>) -> tensor<?x1xi32>
```

Problem

There is no Python API to convert tfl dialect to tosa anymore

Maybe some of you know more here?

MY OWN EXPERIENCE

1. If the Tensorflow package and your MLIR branch are in sync this is quite stable
2. The removal of the Python API is quite inconvenient, as it now requires building from source
3. Depending on TensorFlow for just a few pieces is annoying
4. This is a nice way to test full integer networks

ONNX

Overview

- Open Neural Network Exchange format
- Protocol buffer (.onnx) file format
- Supports static computation graphs
- Enables framework interoperability
- Used as bridge between training and deployment

IMPORTING ONNX VIA ONNX-MLIR

Overview

- Original way of importing ONNX models
- Build by IBM mainly to support their own platforms
- Backend for IBM Telum accelerator is available
- There is an incomplete conversion to TOSA
- The StableHLO conversion has way better coverage and can lower to Linalg

Code and Insights

```
onnx-mlir --EmitONNXIR model.onnx | onnx-mlir-opt --convert-onnx-to-tosa
```

```
func.func private @test_conv_no_bias_no_pad(%arg0 : tensor<1x2x32x64xf32>, %arg1
  %cst = "onnx.NoValue"() {value} : () -> none
  %0 = "onnx.Conv"(%arg0, %arg1, %cst) {auto_pad = "NOTSET", group = 1 : si64} :
  "func.return"(%0) : (tensor<*xf32>) -> ()
```

IMPORTING ONNX VIA TORCH-MLIR

Process

ONNX Model → "ONNX" Torch dialect → Torch dialect → Linalg/TOSA

Code and Insights

- "New kid on the block"
- ONNX and Aten are quite similar
- People didn't want to add yet another MLIR project as a dependency

```
torch-mlir-import-onnx --opset-version 18 model.onnx | torch-mlir-opt --convert-torch-onnx-to-torch --torch-backend-to-linalg-on-tensors-backend-pipeline
```

```
  // CHECK: torch.aten.avg_pool2d %arg0, %[[KERNELSIZE]], %[[STRIDE]], %[[PADDING]], %[[F
  %0 = torch.operator "onnx.GlobalAveragePool"(%arg0) : (!torch.vtensor<[1,3,5,5],f32>) -
  return %0 : !torch.vtensor<[1,3,1,1],f32>
}
```

MY OWN EXPERIENCES

1. Both paths are not stable (It is getting better though!)
2. For ONNX-MLIR the path through StableHLO might be more successful
3. You can get Torch and ONNX support with one dependency
4. The ONNX representation in Torch-MLIR is a bit weird, one would expect a separate dialect for that

TORCH

Overview

- Python-first approach with C++ backend
- Multiple export formats: TorchScript, ONNX, FX Graph
-
- torch.export introduced in PyTorch 2.0+
- Popular in research and production deployments
- No stable serialization format (yet)

CURRENT STATE OF PYTORCH IMPORTING – TORCH EXPORT AND FXIMPORTER

In Torch

- Torch Export is now the default way of exporting a model
- It gives a lot more freedom and expressiveness to the user
- Torch Export uses tracing, control flow on tensor elements break this (graph break)

In MLIR

- FXImporter in Torch-MLIR can take this ExportedProgram and generate the torch dialect
- It has hooks for advanced features like buffers
- This is nontrivial, because torch allows to have mutable buffers

EXAMPLE OF FXIMPORTER

```
@run
# CHECK-LABEL: test_import_frozen_exported_program_with_dynamic_shapes
# CHECK:     func.func @test_net(%[[ARG0:[a-zA-Z0-9]+]]: !torch.vtensor<[?,?,5],f32>) -> !torch.vtensor<[?,?,5],f32>
# CHECK:     %[[S0:.*]] = torch.symbolic_int "s0" {min_val = {{[0-9]+}}, max_val = {{[0-9]+}}} : !torch.int
# CHECK:     %[[S1:.*]] = torch.symbolic_int "s1" {min_val = 2, max_val = {{[0-9]+}}} : !torch.int
# CHECK:     torch.bind_symbolic_shape %[[ARG0]], [%[[S0]], %[[S1]]], affine_map<()[s0, s1] -> (s0, s1, 5)> : !torch.vtensor<[?,?,5],f32>
# CHECK:     %[[TANH:.*]] = torch.aten.tanh %[[ARG0]] : !torch.vtensor<[?,?,5],f32> -> !torch.vtensor<[?,?,5],f32>
# CHECK:     torch.bind_symbolic_shape %[[TANH]], [%[[S0]], %[[S1]]], affine_map<()[s0, s1] -> (s0, s1, 5)> : !torch.vtensor<[?,?,5],f32>
# CHECK:     return %[[TANH]] : !torch.vtensor<[?,?,5],f32>
def test_import_frozen_exported_program_with_dynamic_shapes():
    class Basic(nn.Module):
        def __init__(self):
            super().__init__()

        def forward(self, x):
            return torch.tanh(x)

    batch = Dim("batch", max=10)
    channel = Dim("channel", min=2)
    dynamic_shapes = {"x": {0: batch, 1: channel}}
    m = fx.export_and_import(
        Basic(),
        torch.randn(3, 4, 5),
        dynamic_shapes=dynamic_shapes,
        func_name="test_net",
        import_symbolic_shape_expressions=True,
    )
    print(m)
```


https://github.com/llvm/torch-mlir/blob/cccb0c030c0bafad43b5b3a55863dff1cf616210/test/python/fx_importer/basic_test.py

LOWERING TORCH TO UPSTREAM DIALECTS

File: aten_example.mlir

```
func.func @torch.aten.matmul.2d(%arg0: !torch.vtensor<[8,16],f32>, %arg1: !torch.vtensor<[16,8],f32>) -> !torch.vtensor<[8,8],f32> {
  %0 = torch.aten.matmul %arg0, %arg1 : !torch.vtensor<[8,16],f32>, !torch.vtensor<[16,8],f32> -> !torch.vtensor<[8,8],f32>
  return %0 : !torch.vtensor<[8,8],f32>
}
```

```
torch-mlir-opt --torch-backend-to-tosa-backend-pipeline file.mlir
```

STDIN

```
module {
  func.func @torch.aten.matmul.2d(%arg0: tensor<8x16xf32>, %arg1: tensor<16x8xf32>) -> tensor<8x8xf32> {
    %0 = tosa.reshape %arg0 {new_shape = array<i64: 1, 8, 16>} : (tensor<8x16xf32>) -> tensor<1x8x16xf32>
    %1 = tosa.reshape %arg1 {new_shape = array<i64: 1, 16, 8>} : (tensor<16x8xf32>) -> tensor<1x16x8xf32>
    %2 = tosa.matmul %0, %1 : (tensor<1x8x16xf32>, tensor<1x16x8xf32>) -> tensor<1x8x8xf32>
    %3 = tosa.reshape %2 {new_shape = array<i64: 8, 8>} : (tensor<1x8x8xf32>) -> tensor<8x8xf32>
    return %3 : tensor<8x8xf32>
  }
}
```

LOWERING TORCH TO UPSTREAM DIALECTS

File: aten_example.mlir

```
func.func @torch.aten.matmul.2d(%arg0: !torch.vtensor<[8,16],f32>, %arg1: !torch.vtensor<[16,8],f32>) -> !torch.vtensor<[8,8],f32> {
  %0 = torch.aten.matmul %arg0, %arg1 : !torch.vtensor<[8,16],f32>, !torch.vtensor<[16,8],f32> -> !torch.vtensor<[8,8],f32>
  return %0 : !torch.vtensor<[8,8],f32>
}
```

```
torch-mlir-opt --torch-backend-to-linalg-on-tensors-backend-pipeline file.mlir
```

STDIN

```
module {
  func.func @torch.aten.matmul.2d(%arg0: tensor<8x16xf32>, %arg1: tensor<16x8xf32>) -> tensor<8x8xf32> {
    %cst = arith.constant 0.000000e+00 : f32
    %0 = tensor.empty() : tensor<8x8xf32>
    %1 = linalg.fill ins(%cst : f32) outs(%0 : tensor<8x8xf32>) -> tensor<8x8xf32>
    %2 = linalg.matmul ins(%arg0, %arg1 : tensor<8x16xf32>, tensor<16x8xf32>) outs(%1 : tensor<8x8xf32>) -> tensor<8x8xf32>
    return %2 : tensor<8x8xf32>
  }
}
```

MY OWN EXPERIENCES

1. Torch Export is extremely powerful in what it can capture
2. Torch frontends need to be more complicated by design
3. FXImporter is quite nice, and the hooks expose things like mutable buffers
4. Torch to Linalg has more coverage than Torch to TOSA
5. Code quality of some parts makes it hard to add new features or fix bugs
   -> Most of the time it is good enough
6. The project got redesigned into a “frontend” a few years back and not everything is adapted yet
7. TOSA and Torch have a shape mismatch for convolutions and pooling ops
   -> You will see a lot of transposes in the IR

JAX

Overview

- Functional approach to numerical computing and ML
- XLA-based acceleration for NumPy operation
- Auto-differentiation
- Used in research and large-scale transformers
- Native integration with MLIR via XLA/MHLO

https://docs.jax.dev/en/latest/

HOW JAX HANDLES MLIR EXPORT

Code

```python
from jax._src.interpreters import mlir as jax_mlir
from jax._src.lib.mlir import ir


# Returns prettyprint of StableHLO module without large constants
def get_stablehlo_asm(module_str):
    with jax_mlir.make_ir_context():
        stablehlo_module = ir.Module.parse(
            module_str, context=jax_mlir.make_ir_context()
        )
        return stablehlo_module.operation.get_asm(large_elements_limit=20)


# Disable logging for better tutorial rendering
import logging

logging.disable(logging.WARNING)


from transformers import AutoImageProcessor, FlaxResNetModel
import jax
from jax import export
import numpy as np

# Construct jit-transformed flax model with sample inputs
resnet18 = FlaxResNetModel.from_pretrained("microsoft/resnet-18", return_dict=False)
resnet18_jit = jax.jit(resnet18)
sample_input = np.random.randn(1, 3, 224, 224)
input_shape = jax.ShapeDtypeStruct(sample_input.shape, sample_input.dtype)

# Export to StableHLO
stablehlo_resnet18_export = export.export(resnet18_jit)(input_shape)
resnet18_stablehlo = get_stablehlo_asm(stablehlo_resnet18_export.mlir_module())
print(resnet18_stablehlo)
```

Command

```
python jax_example.py | stablehlo-opt --stablehlo-legalize-to-linalg
```

Insight

- Exporting from their example on was straightforward
- To lower to TOSA or linalg you need to build Stablehlo
- I don't have too much experience with it, but the coverage looks good

IN SUMMARY: I HAVE MY ML MODEL, WHAT NOW?

Each framework has its project

- LiteRT -> Tensorflow
- ONNX -> ONNX-MLIR or Torch-MLIR
- Torch -> Torch-MLIR
- JAX -> StableHLO

Insights

- Easy cases most often work
- Devil is often in the detail, especially for Torch
- You can often choose TOSA or Linalg, but overall Linalg has better support IMO
- It is a bit annoying that you need to build some of the projects from source

ANY QUESTIONS? I AM HAPPY TO CONNECT!

CTO
Maximilian Bartel

bartel@roofline.ai

https://www.linkedin.com/in/maximilianbartel97/

maxbartel

www.roofline.ai