ML-Compiler Optimizations Assignment Report

Model: ResNet18 (torchvision)
Tools: PyTorch FX, torch-mlir
Date: November 5, 2025

Part 1: PyTorch Model and FX Graph Analysis

1.1 Model Selection

We use **ResNet18** from torchvision, consisting of basic layers:

- **Convolution (Conv2d):** 20 layers (17 main path + 3 downsample)
- **Batch Normalization (BatchNorm2d):** 20 layers
- **ReLU activations:** 17 instances
- **Residual Add operations:** 8 instances
- **Pooling:** MaxPool2d (1) + AdaptiveAvgPool2d (1)
- **Linear (FC):** 1 layer (512 → 1000)

1.2 FX Graph Tracing

```
import torch
import torchvision.models as models
import torch.fx

model = models.resnet18(weights=None)
model.eval()
traced = torch.fx.symbolic_trace(model)
traced.graph.print_tabular()
```

1.3 FX Graph Output (Excerpt)

```
opcode         name                  target                   args
-------------  --------------------  -----------------------  -----------
----------------------
placeholder    x                     x                        ()
call_module    conv1                 conv1                    (x,)
call_module    bn1                   bn1                      (conv1,)
call_module    relu                  relu                     (bn1,)
call_module    maxpool               maxpool                  (relu,)
call_module    layer1_0_conv1        layer1.0.conv1           (maxpool,)
call_module    layer1_0_bn1          layer1.0.bn1
(layer1_0_conv1,)
call_module    layer1_0_relu         layer1.0.relu
(layer1_0_bn1,)
call_module    layer1_0_conv2        layer1.0.conv2
```

```
(layer1_0_relu,)
call_module    layer1_0_bn2           layer1.0.bn2
(layer1_0_conv2,)
call_function  add                    <built-in function add>
(layer1_0_bn2, maxpool)
call_module    layer1_0_relu_1        layer1.0.relu              (add,)
...
call_module    avgpool                avgpool
(layer4_1_relu_1,)
call_function  flatten                <built-in method flatten> (avgpool,
1)
call_module    fc                     fc                         (flatten,)
output         output                 output                     (fc,)
```

Graph Statistics:

- Total nodes: 71
- call_module: 60
- call_function: 9
- placeholder: 1, output: 1

1.4 Structural Explanation (5-6 Sentences)

ResNet18 is a deep convolutional neural network consisting of an initial 7×7 convolution followed by batch normalization, ReLU activation, and max pooling. The network contains 4 sequential stages (layer1-layer4), each composed of BasicBlock residual units containing pairs of 3×3 convolutions with batch normalization and ReLU activations, plus skip connections. Each BasicBlock adds the input (identity or downsampled projection) to the output of the convolution path, enabling gradient flow through residual learning. The spatial dimensions are progressively reduced (112×112 → 56×56 → 28×28 → 14×14 → 7×7) while channels increase (64 → 128 → 256 → 512) through strided convolutions at layer transitions. Finally, adaptive average pooling reduces spatial dimensions to 1×1, followed by a fully connected layer mapping 512 features to 1000 ImageNet classes.

1.5 Redundant and Chainable Operations

Pattern	Count	Description
Conv → BN	17	Every Conv2d immediately followed by BatchNorm2d
BN → ReLU	9	BatchNorm followed by ReLU activation
Add → ReLU	8	Residual addition followed by ReLU
Conv → BN → ReLU	9	Triple chain fusible into single kernel
ReLU Module Reuse	8	Same ReLU module called twice per BasicBlock
Flatten (redundant)	1	Reshape with zero computation

Part 2: Torch-MLIR Compilation and Optimizations

2.1 Installation

```
conda create -n torch-mlir python=3.11 -y
conda activate torch-mlir
pip install torch torchvision
pip install torch-mlir -f https://github.com/llvm/torch-mlir/releases
```

2.2 MLIR Generation

```
from torch_mlir.fx import export_and_import, OutputType
import torch

model = models.resnet18(weights=None)
model.eval()
example_input = torch.randn(1, 3, 224, 224)

exported = torch.export.export(model, (example_input,))
mlir_module = export_and_import(exported,
output_type=OutputType.LINALG_ON_TENSORS)

with open("resnet18_mlir.mlir", "w") as f:
    f.write(str(mlir_module))
```

Output: `resnet18_mlir.mlir` (93,610,124 bytes)

2.3 MLIR Output Analysis

```
MLIR Operation Counts (actual from file):
  - linalg.conv_2d_nchw_fchw:  20  (convolutions)
  - linalg.generic:           167  (element-wise ops)
  - tensor.pad:                18  (padding operations)
  - linalg.fill:                8  (tensor initialization)
  - linalg.pooling:             2  (pooling layers)
  - tensor.expand_shape:       80  (reshape for broadcasting)

Arithmetic Ops inside linalg.generic blocks:
  - arith.subf:    20  (BN: subtract mean)
  - arith.mulf:    40  (BN: multiply by inv_std, gamma)
  - arith.addf:    49  (BN: add beta, residual add)
  - arith.select:  17  (ReLU: max(0, x))
```

2.4 MLIR Sample (First Conv + BN + ReLU)

```
// Convolution
%2 = linalg.conv_2d_nchw_fchw {dilations = dense<1>, strides = dense<2>}
```

```
      ins(%padded, %cst : tensor<1x3x230x230xf32>, tensor<64x3x7x7xf32>)
      outs(%1 : tensor<1x64x112x112xf32>) -> tensor<1x64x112x112xf32>

// BatchNorm decomposed into 4 linalg.generic ops:
%7 = linalg.generic ... { arith.subf %in, %mean }      // x - μ
%8 = linalg.generic ... { arith.mulf %7, %inv_std }    // (x-μ)/σ
%9 = linalg.generic ... { arith.mulf %8, %gamma }      // γ * normalized
%10 = linalg.generic ... { arith.addf %9, %beta }      // + β

// ReLU
%11 = linalg.generic ... { arith.select %cmp, %in, %zero }  // max(0, x)
```

2.5 Proposed Optimizations (Based on MLIR Analysis)

Optimization 1: Batch Normalization Folding

Observed: Each BatchNorm is lowered to 4 sequential `linalg.generic` operations (subf, mulf, mulf, addf).

Optimization: At inference time, fold BN parameters into Conv weights:

```
W_new = W × (γ / σ)
b_new = (b - μ) × (γ / σ) + β
```

Benefit: Eliminates 80 `linalg.generic` ops (4 × 20 BN layers), ~20% fewer operations.

Optimization 2: Element-wise Fusion

Observed: Separate `linalg.generic` ops for BN steps and ReLU.

Optimization: Apply `-linalg-fuse-elementwise-ops` MLIR pass.

Benefit: Reduces memory traffic by 67% per fusion (eliminates intermediate tensors).

Optimization 3: Pad-Conv Fusion

Observed: 18 explicit `tensor.pad` operations before convolutions.

Optimization: Fuse padding into convolution kernel (implicit padding).

Benefit: Eliminates 18 intermediate padded tensor allocations.

Optimization 4: Buffer Reuse

Observed: Many `tensor.empty()` calls create separate buffers.

Optimization: Apply `-buffer-deallocation`, `-buffer-hoisting` passes.

Benefit: 30-50% peak memory reduction.

Optimization 5: Loop Tiling and Vectorization

Optimization: Apply `-linalg-tile`, `-linalg-vectorize` passes.

Benefit: Better cache utilization, enables SIMD instructions.

Part 3: Fusion Opportunities and Hardware Recommendation

3.1 Fusion Pair 1: Conv2d → BatchNorm2d → ReLU

Evidence from FX Graph:

```
conv1 → bn1 → relu
layer1_0_conv1 → layer1_0_bn1 → layer1_0_relu
... (17 total Conv→BN chains, 9 BN→ReLU chains)
```

Evidence from MLIR:

- 20 `linalg.conv_2d_nchw_fchw` operations
- Each BN = 4 `linalg.generic` ops (subf → mulf → mulf → addf)
- 17 `arith.select` operations (ReLU)

Effect on Memory Access:

Metric	Before Fusion	After Fusion	Savings
Kernel operations	6	1	5 fewer
Memory accesses	6 tensor R/W	2 tensor R/W	**67%**
Traffic (112×112×64)	19.2 MB	6.4 MB	12.8 MB

Effect on Kernel Launch Overhead:

- Before: 6 kernel launches with GPU synchronization
- After: 1 kernel launch
- Savings: ~25-100 µs per block × 17 blocks = **~850 µs total**

3.2 Fusion Pair 2: Add (Residual) → ReLU

Evidence from FX Graph:

```
add → layer1_0_relu_1
add_1 → layer1_1_relu_1
... (8 total Add→ReLU pairs)
```

Evidence from MLIR:

```
%34 = linalg.generic ... { arith.addf %bn_out, %skip }  // residual add
%35 = linalg.generic ... { arith.select ... }           // ReLU
```

Effect on Memory Access:

Metric	Before Fusion	After Fusion	Savings
Memory ops	4 (2R + 2W)	3 (2R + 1W)	**25%**
Intermediate tensor	Required	Eliminated	100%

Effect on Kernel Launch Overhead:

- Before: 2 kernel launches
- After: 1 kernel launch
- Savings: ~5-20 µs × 8 blocks = **~80 µs total**

3.3 Additional Memory Optimizations

Optimization	MLIR Evidence	Benefit
In-place ReLU	17 `arith.select` ops	50% activation memory savings
Buffer Reuse	Many `tensor.empty()`	30-50% peak memory reduction
BN Constant Folding	80 BN `linalg.generic` ops	Eliminate all BN ops
Pad-Conv Fusion	18 `tensor.pad` ops	Eliminate padded tensor allocation

3.4 Hardware Backend Recommendation: **GPU (NVIDIA CUDA)**

Factor	GPU Advantage
Parallelism	Thousands of cores match conv parallelism (millions of MACs)
Memory Bandwidth	500-2000 GB/s vs CPU's 50-100 GB/s
Fusion Support	cuDNN provides fused Conv-BN-ReLU kernels natively
Tensor Cores	4-8× speedup with FP16/INT8

Why not CPU/TPU:

- **CPU:** Lower parallelism, viable only for edge/small batch inference
- **TPU:** Requires large batches (≥8), better suited for training than inference

Expected GPU Performance (RTX 3080):

- FP32: ~100-200 images/sec
- FP16: ~200-400 images/sec (Tensor Cores)
- INT8: ~400-800 images/sec (TensorRT)

Summary

Part	Key Findings
Part 1	FX traced ResNet18: 71 nodes, 60 modules, 9 functions. Identified 17 Conv-BN, 9 BN-ReLU, 8 Add-ReLU chainable patterns.
Part 2	Generated 93MB MLIR with 20 convs, 167 element-wise ops. Proposed 6 optimizations: BN folding, element-wise fusion, pad-conv fusion, buffer reuse, tiling, vectorization.
Part 3	Two fusion pairs (Conv-BN-ReLU: 67% memory savings; Add-ReLU: 25% savings). Recommended GPU for parallelism, bandwidth, and native fusion support.

Files

File	Description
`q1.py`	FX graph tracing and fusion analysis
`q2.py`	torch-mlir compilation and optimization proposals
`q3.py`	Fusion analysis from FX/MLIR with hardware recommendation
`resnet18_mlir.mlir`	Generated MLIR (linalg-on-tensors dialect)
`report.md`	This report

How to Run

```
conda activate torch-mlir
python q1.py   # Part 1: FX graph analysis
python q2.py   # Part 2: MLIR generation + optimizations
python q3.py   # Part 3: Fusion analysis + hardware recommendation
```