PyTorch-to-Quantized-C Compiler

Compiling Neural Networks for Microcontrollers

Ashitabh Misra, Joydeep Bhattacharyya, Tarek Abdelzaher

January 9, 2026

Outline

The Problem

Deploying quantized neural networks on microcontrollers is challenging.

Existing tools are inflexible and difficult to extend:

- **TFLite Micro:** Large binary size for microcontrollers usecases, not designed to be extended. As a result, extremely difficult to add new quantization techniques.

The Problem

Deploying quantized neural networks on microcontrollers is challenging.

Existing tools are inflexible and difficult to extend:

- **TFLite Micro:** Large binary size for microcontrollers usecases, not designed to be extended. As a result, extremely difficult to add new quantization techniques.
- **Glow:** Complex setup, designed for research not real-world embedded use, hard to add new features.

Comparison with Existing Tools

Feature	Tiny-NN-in-C	TFLite Micro	Glow

Comparison with Existing Tools

Feature	Tiny-NN-in-C	TFLite Micro	Glow
Framework	PyTorch	TensorFlow	ONNX/PyTorch*

Comparison with Existing Tools

Feature	Tiny-NN-in-C	TFLite Micro	Glow
Framework	PyTorch	TensorFlow	ONNX/PyTorch*
Output	Standalone C	.tflite + Interp.	Compiled binary

Comparison with Existing Tools

Feature	Tiny-NN-in-C	TFLite Micro	Glow
Framework	PyTorch	TensorFlow	ONNX/PyTorch*
Output	Standalone C	.tflite + Interp.	Compiled binary
Code Readability			

Comparison with Existing Tools

Feature	Tiny-NN-in-C	TFLite Micro	Glow
Framework	PyTorch	TensorFlow	ONNX/PyTorch*
Output	Standalone C	.tflite + Interp.	Compiled binary
Code Readability	✓	✗ Binary	✗ Opt. IR
Setup Complexity			

Comparison with Existing Tools

Feature	Tiny-NN-in-C	TFLite Micro	Glow
Framework	PyTorch	TensorFlow	ONNX/PyTorch*
Output	Standalone C	.tflite + Interp.	Compiled binary
Code Readability	✓	✗ Binary	✗ Opt. IR
Setup Complexity	Simple (pip)	Medium	Complex
Binary Size			

Comparison with Existing Tools

Feature	Tiny-NN-in-C	TFLite Micro	Glow
Framework	PyTorch	TensorFlow	ONNX/PyTorch*
Output	Standalone C	.tflite + Interp.	Compiled binary
Code Readability	✓	✗ Binary	✗ Opt. IR
Setup Complexity	Simple (pip)	Medium	Complex
Binary Size	∼150KB	∼390KB+	Varies
Quantization			

Comparison with Existing Tools

Feature	Tiny-NN-in-C	TFLite Micro	Glow
Framework	PyTorch	TensorFlow	ONNX/PyTorch*
Output	Standalone C	.tflite + Interp.	Compiled binary
Code Readability	✓	✗ Binary	✗ Opt. IR
Setup Complexity	Simple (pip)	Medium	Complex
Binary Size	∼150KB	∼390KB+	Varies
Quantization	int8/16, mixed	int8	int8
Custom Rules			

Comparison with Existing Tools

Feature	Tiny-NN-in-C	TFLite Micro	Glow
Framework	PyTorch	TensorFlow	ONNX/PyTorch*
Output	Standalone C	.tflite + Interp.	Compiled binary
Code Readability	✓	✗ Binary	✗ Opt. IR
Setup Complexity	Simple (pip)	Medium	Complex
Binary Size	∼150KB	∼390KB+	Varies
Quantization	int8/16, mixed	int8	int8
Custom Rules			
Extensibility			

Comparison with Existing Tools

Feature	Tiny-NN-in-C	TFLite Micro	Glow
Framework	PyTorch	TensorFlow	ONNX/PyTorch*
Output	Standalone C	.tflite + Interp.	Compiled binary
Code Readability	✓	✗ Binary	✗ Opt. IR
Setup Complexity	Simple (pip)	Medium	Complex
Binary Size	∼150KB	∼390KB+	Varies
Quantization	int8/16, mixed	int8	int8
Custom Rules	✓ Regex	✗ Per-layer	✗ Manual
Extensibility			
Transparency			

Comparison with Existing Tools

Feature	Tiny-NN-in-C	TFLite Micro	Glow
Framework	PyTorch	TensorFlow	ONNX/PyTorch*
Output	Standalone C	.tflite + Interp.	Compiled binary
Code Readability	✓	✗ Binary	✗ Opt. IR
Setup Complexity	Simple (pip)	Medium	Complex
Binary Size	∼150KB	∼390KB+	Varies
Quantization	int8/16, mixed	int8	int8
Custom Rules	✓ Regex	✗ Per-layer	✗ Manual
Extensibility			
Transparency			

Comparison with Existing Tools

Feature	Tiny-NN-in-C	TFLite Micro	Glow
Framework	PyTorch	TensorFlow	ONNX/PyTorch*
Output	Standalone C	.tflite + Interp.	Compiled binary
Code Readability	✓	✗ Binary	✗ Opt. IR
Setup Complexity	Simple (pip)	Medium	Complex
Binary Size	∼150KB	∼390KB+	Varies
Quantization	int8/16, mixed	int8	int8
Custom Rules	✓ Regex	✗ Per-layer	✗ Manual
Extensibility	✓ Easy	✗ C++	✗ LLVM
Transparency			

*Glow development has slowed; PyTorch support via ONNX conversion

Comparison with Existing Tools

Feature	Tiny-NN-in-C	TFLite Micro	Glow
Framework	PyTorch	TensorFlow	ONNX/PyTorch*
Output	Standalone C	.tflite + Interp.	Compiled binary
Code Readability	✓	✗ Binary	✗ Opt. IR
Setup Complexity	Simple (pip)	Medium	Complex
Binary Size	∼150KB	∼390KB+	Varies
Quantization	int8/16, mixed	int8	int8
Custom Rules	✓ Regex	✗ Per-layer	✗ Manual
Extensibility	✓ Easy	✗ C++	✗ LLVM
Transparency			

*Glow development has slowed; PyTorch support via ONNX conversion

Comparison with Existing Tools

Feature	Tiny-NN-in-C	TFLite Micro	Glow
Framework	PyTorch	TensorFlow	ONNX/PyTorch*
Output	Standalone C	.tflite + Interp.	Compiled binary
Code Readability	✓	✗ Binary	✗ Opt. IR
Setup Complexity	Simple (pip)	Medium	Complex
Binary Size	∼150KB	∼390KB+	Varies
Quantization	int8/16, mixed	int8	int8
Custom Rules	✓ Regex	✗ Per-layer	✗ Manual
Extensibility	✓ Easy	✗ C++	✗ LLVM
Transparency	✓ See code	✗ Black box	✗ Black box

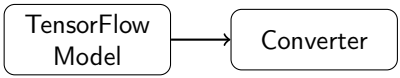
*Glow development has slowed; PyTorch support via ONNX conversion

Architecture Comparison

TFLite Micro:

Architecture Comparison

TFLite Micro:

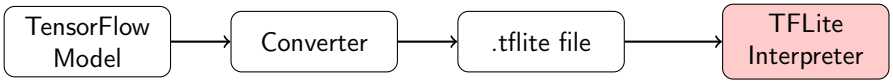


Architecture Comparison

TFLite Micro:

Architecture Comparison

TFLite Micro:



Architecture Comparison

TFLite Micro:

Architecture Comparison

TFLite Micro:

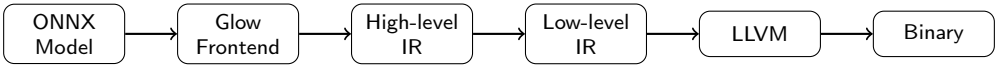
Runtime on device (∼390KB+)

Architecture Comparison

TFLite Micro:

Runtime on device (∼390KB+)

Glow:

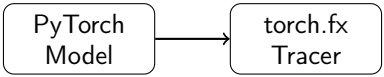


(Complex optimization pipeline)

Our Approach: Tiny-NN-in-C

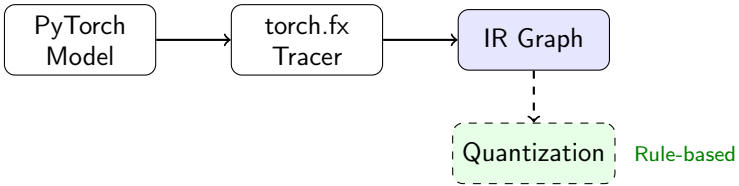
PyTorch Model

Our Approach: Tiny-NN-in-C

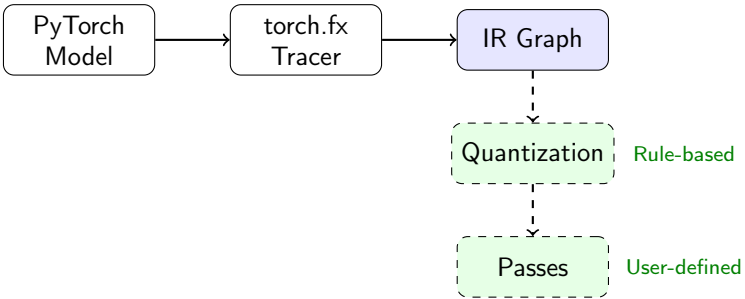


Our Approach: Tiny-NN-in-C

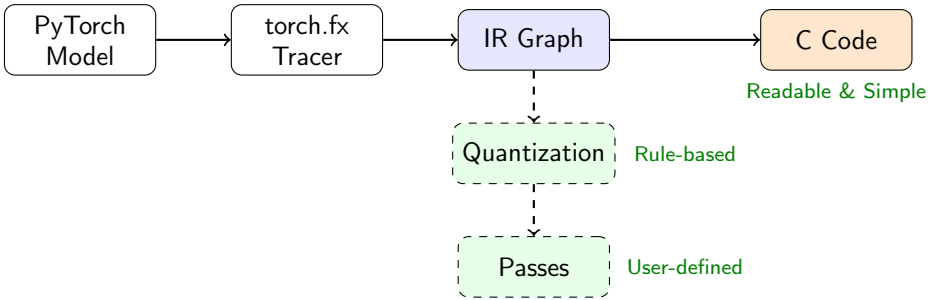
Our Approach: Tiny-NN-in-C



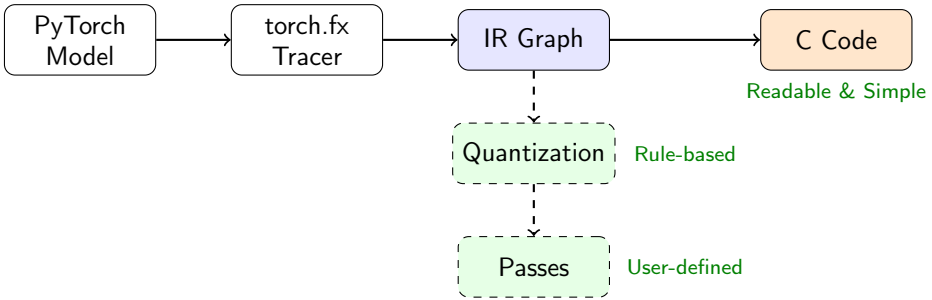
Our Approach: Tiny-NN-in-C



Our Approach: Tiny-NN-in-C



Our Approach: Tiny-NN-in-C



Key Insight

We achieve **transparency** and **simplicity** with minimal performance penalty.

Quick Start: The Core API

Three simple steps to compile a quantized model:

Step 1: Compile to C (float)

```
from src.pytorch_to_c.compiler import compile_model

ir_graph = compile_model(model, example_input, output_dir="out/")
```

Quick Start: The Core API

Three simple steps to compile a quantized model:

Step 1: Compile to C (float)

```
from src.pytorch_to_c.compiler import compile_model

ir_graph = compile_model(model, example_input, output_dir="out/")
```

Step 2: Define quantization rules

```
from src.pytorch_to_c.quantization import StaticQuantRule

rules = [StaticQuantRule(pattern=r'fc.*', dtype='int8', ...)]
```

Quick Start: The Core API

Three simple steps to compile a quantized model:

Step 1: Compile to C (float)

```
from src.pytorch_to_c.compiler import compile_model

ir_graph = compile_model(model, example_input, output_dir="out/")
```

Step 2: Define quantization rules

```
from src.pytorch_to_c.quantization import StaticQuantRule

rules = [StaticQuantRule(pattern=r'fc.*', dtype='int8', ...)]
```

Step 3: Apply quantization and generate

```
transform = QuantizationTransform(rules)
quant_ir = transform.apply(ir_graph)
CPrinter(quant_ir).generate_all("out/")
```

What is This?

A PyTorch-to-C compiler for microcontrollers

- Takes PyTorch models $\rightarrow$ Generates standalone C code

What is This?

A PyTorch-to-C compiler for microcontrollers

- Takes PyTorch models $\rightarrow$ Generates standalone C code
- Supports quantization (int8, int16) for embedded deployment

What is This?

A PyTorch-to-C compiler for microcontrollers

- Takes PyTorch models $\rightarrow$ Generates standalone C code
- Supports quantization (int8, int16) for embedded deployment
- Extensible rule-based system for selective quantization

What is This?

A PyTorch-to-C compiler for microcontrollers

- Takes PyTorch models $\rightarrow$ Generates standalone C code
- Supports quantization (int8, int16) for embedded deployment
- Extensible rule-based system for selective quantization
- IR-based architecture enables optimization passes

What is This?

A PyTorch-to-C compiler for microcontrollers

- Takes PyTorch models → Generates standalone C code
- Supports quantization (int8, int16) for embedded deployment
- Extensible rule-based system for selective quantization
- IR-based architecture enables optimization passes

```
python examples/tiny_resnet.py
```

Simple MLP Example

Model: 2-layer MLP (Linear → ReLU → Linear)

```
class SimpleMLP(nn.Module):
    def __init__(self, input_size=784, hidden_size=128, output_size=10):
        super().__init__()
```

Simple MLP Example

Model: 2-layer MLP (Linear → ReLU → Linear)

```
class SimpleMLP(nn.Module):
    def __init__(self, input_size=784, hidden_size=128, output_size=10):
        super().__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(hidden_size, output_size)
```

Simple MLP Example

Model: 2-layer MLP (Linear → ReLU → Linear)

```
class SimpleMLP(nn.Module):
    def __init__(self, input_size=784, hidden_size=128, output_size=10):
        super().__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        return x
```

Simple MLP Example

Compile to C:

```
model = SimpleMLP(input_size=784, hidden_size=128, output_size=10)
example_input = torch.randn(1, 784)

ir_graph = compile_model(
    model=model,
    example_input=example_input,
    output_dir="generated"
)
```

Simple MLP Example: Generated IR Graph

IR Graph structure:

```
IRGraph:
  Inputs: ['x']
  Outputs: ['fc2']
  Parameters: ['fc1_weight', 'fc1_bias', 'fc2_weight', 'fc2_bias']
  Nodes:
    x [input]
      inputs: []
      users: [fc1]
      shape: (1, 784), dtype: float32
    fc1 [linear]
      inputs: [x]
      users: [relu]
      shape: (1, 128), dtype: float32
    relu [relu]
      inputs: [fc1]
      users: [fc2]
      shape: (1, 128), dtype: float32
    fc2 [linear]
      inputs: [relu]
      users: []
      shape: (1, 10), dtype: float32
```

ResNet-Style Model (Skip Connections)

Architecture: Conv → BatchNorm → ReLU → Skip → Pool → FC

```
class ResNetBlock(nn.Module):
    def __init__(self, channels):
        super().__init__()
        self.conv1 = nn.Conv2d(channels, channels, 3, padding=1)
        self.bn1 = nn.BatchNorm2d(channels)
```

ResNet-Style Model (Skip Connections)

Architecture: Conv → BatchNorm → ReLU → Skip → Pool → FC

```
class ResNetBlock(nn.Module):
    def __init__(self, channels):
        super().__init__()
        self.conv1 = nn.Conv2d(channels, channels, 3, padding=1)
        self.bn1 = nn.BatchNorm2d(channels)

    def forward(self, x):
        identity = x
        out = torch.relu(self.bn1(self.conv1(x)))
        return out + identity
```

ResNet-Style Model (Skip Connections)

TinyResNet: Using ResNetBlock with skip connections

```
class TinyResNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv_init = nn.Conv2d(3, 32, 3, padding=1)
        self.block1 = ResNetBlock(32)
        self.fc = nn.Linear(32, 10)

    def forward(self, x):
        x = torch.relu(self.conv_init(x))
        x = self.block1(x)
        x = x.mean(dim=[2, 3])
        return self.fc(x)
```

Static Quantization

Key idea: Scales are known at compile time (from calibration)

Step 1: Define quantization rules

```
from src.pytorch_to_c.quantization import StaticQuantRule

rules = [
    StaticQuantRule(
        pattern=r'fc.*',
        dtype='int8',
```

Static Quantization

Key idea: Scales are known at compile time (from calibration)

Step 1: Define quantization rules

```
from src.pytorch_to_c.quantization import StaticQuantRule

rules = [
    StaticQuantRule(
        pattern=r'fc.*',
        dtype='int8',
        input_scale=0.01, weight_scale=0.01, output_scale=0.01
    )
]
```

Static Quantization

Key idea: Scales are known at compile time (from calibration)

Step 2: Apply to IR graph

```
ir_graph = compile_model(model, example_input, return_ir=True)
transform = QuantizationTransform(rules)
quant_ir = transform.apply(ir_graph)
```

Dynamic Quantization

Key idea: Input scale computed at runtime from actual data

Define dynamic rule (no calibration needed!):

```
from src.pytorch_to_c.quantization import DynamicQuantRuleMinMaxPerTensor

rules = [
    DynamicQuantRuleMinMaxPerTensor(
        pattern=r'fc.*',
        dtype='int8'
    )
]
```

Dynamic Quantization

Key idea: Input scale computed at runtime from actual data

Generated C code computes scale dynamically:

```
float scale = compute_dynamic_scale_int8(input, 784);
quantize_float_to_int8(input, 784, scale, 0, buf_quantized);
```

Advantage

No calibration dataset required—scales derived from weight statistics.

Mixed Precision Quantization

Selective quantization: Different rules for different layers

```
rules = [
    StaticQuantRule(pattern=r'.*encoder.*', dtype='int8', ...),
]
```

Mixed Precision Quantization

Selective quantization: Different rules for different layers

```
rules = [
    StaticQuantRule(pattern=r'.*encoder.*', dtype='int8', ...),

    StaticQuantRule(pattern=r'.*output.*', dtype='int16', ...),
]
```

Correctness Results (On TinyResNet model)

Comparing our implementation with Pytorch Float32 model.

Configuration	Max Error	Memory Savings

Correctness Results (On TinyResNet model)

Comparing our implementation with Pytorch Float32 model.

Configuration	Max Error	Memory Savings
Float32 (baseline)	1.19×10^{-7}	—

Correctness Results (On TinyResNet model)

Comparing our implementation with Pytorch Float32 model.

Configuration	Max Error	Memory Savings
Float32 (baseline)	1.19×10^{-7}	—
Static int16	0.07%	2×

Correctness Results (On TinyResNet model)

Comparing our implementation with Pytorch Float32 model.

Configuration	Max Error	Memory Savings
Float32 (baseline)	1.19×10^{-7}	—
Static int16	0.07%	2×
Static int8	1.63%	4×

Correctness Results (On TinyResNet model)

Comparing our implementation with Pytorch Float32 model.

Configuration	Max Error	Memory Savings
Float32 (baseline)	1.19×10^{-7}	—
Static int16	0.07%	2×
Static int8	1.63%	4×
Dynamic int8	2.95%	4×

Correctness Results (On TinyResNet model)

Comparing our implementation with Pytorch Float32 model.

Configuration	Max Error	Memory Savings
Float32 (baseline)	1.19×10^{-7}	—
Static int16	0.07%	2×
Static int8	1.63%	4×
Dynamic int8	2.95%	4×

Takeaway

Choose precision based on your accuracy requirements and memory constraints.

Optimization Pass: FuseDequantQuant

Problem: Consecutive quantized layers have redundant conversions

```
BEFORE: fc1(int8) -> dequant(float) -> quant(int8) -> fc2(int8)
```

Optimization Pass: FuseDequantQuant

Problem: Consecutive quantized layers have redundant conversions

```
BEFORE: fc1(int8) -> dequant(float) -> quant(int8) -> fc2(int8)
AFTER:  fc1(int8) -> fc2(int8)
```

Optimization Pass: FuseDequantQuant

Apply the optimization pass:

```
from src.passes import FuseDequantQuantPass

fuse_pass = FuseDequantQuantPass()
optimized_ir = fuse_pass.apply(quant_ir)
```

Optimization Pass: FuseDequantQuant

Apply the optimization pass:

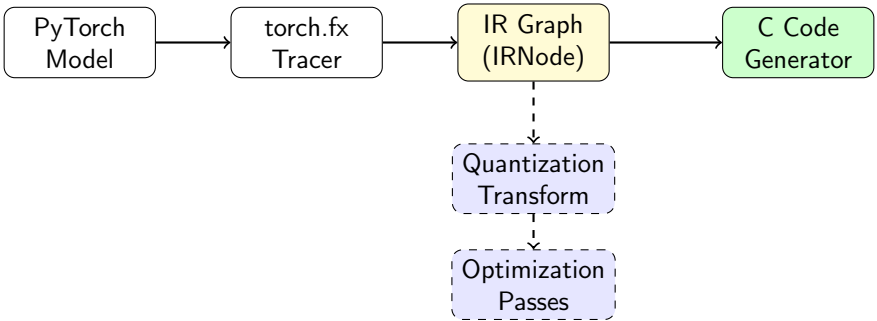
```
from src.passes import FuseDequantQuantPass

fuse_pass = FuseDequantQuantPass()
optimized_ir = fuse_pass.apply(quant_ir)
```

Result

When scales match, fusion is **bit-identical**: Max error = `0.00e+00`

Architecture Overview



Key files:

- `frontend/fx_tracer.py` – torch.fx tracing
- `lowering/lower.py` – FX $\rightarrow$ IR conversion
- `ir/graph.py` – IR graph structure
- `codegen/c_printer.py` – C code generation

IR Node Structure

Base class: `src/pytorch_to_c/ir/node.py`

```
class IRNode:
    def __init__(self, name, op_type, inputs=None, dtype='float32'):
        self.name = name            # Unique identifier
        self.op_type = op_type      # 'linear', 'conv2d', 'relu', etc.
        self.inputs = inputs or []  # List of input IRNodes
        self.users = []             # Nodes that use this output
        self.dtype = dtype          # 'float32', 'int8', 'int16'
        self.output_shape = None    # Inferred shape
        self.metadata = {}          # Extra info (weights, etc.)

    def get_c_dtype(self) -> str:
        """Returns C type: 'float', 'int8_t', 'int16_t'"""
```

Graph is doubly-linked: `inputs` + `users` enable traversal in both directions

IR Graph Structure

File: src/pytorch_to_c/ir/graph.py

```python
class IRGraph:
    def __init__(self):
        self.nodes = []
        self.inputs = []
        self.outputs = []
        self.parameters = {}

    def print_graph(self):
        """Pretty-print graph"""
```

Example IR output:

```
x [input] dtype=float32
  inputs: []
  users: [fc1]
fc1 [linear] dtype=float32
  inputs: [x]
  users: [relu]
  shape: (1, 8)
```

Quantization Rule System

Base class: `src/pytorch_to_c/quantization/rules.py`

```python
class QuantRule(ABC):
    def __init__(self, pattern: str, dtype: str):
        self.pattern = re.compile(pattern)  # Regex for layer names
        self.dtype = dtype                  # 'int8' or 'int16'

    def matches(self, node: IRNode) -> bool:
        """Check if rule applies to this node"""
        return self.pattern.search(node.name) is not None

    @abstractmethod
    def create_quant_node(self, float_node: IRNode) -> QuantIRNode:
        """Create quantized version of the node"""
```

Rule matching: First matching rule wins

```python
RuleMatcher([rule1, rule2, rule3])  # Checked in order
```

Writing a Custom Rule

Example: Static Quantization Rule

```
class StaticQuantRule(QuantRule):
    def __init__(self, pattern, dtype,
                 input_scale, weight_scale, output_scale, ...):
        super().__init__(pattern, dtype)
        self.input_scale = input_scale
        # ... store all scales/offsets
```

Writing a Custom Rule

Example: Static Quantization Rule

```
class StaticQuantRule(QuantRule):
    def __init__(self, pattern, dtype,
                 input_scale, weight_scale, output_scale, ...):
        super().__init__(pattern, dtype)
        self.input_scale = input_scale

    def create_quant_node(self, float_node: IRNode) -> QuantIRNode:
        if float_node.op_type == 'linear':
            return StaticQuantLinearNode(
                name=float_node.name, dtype=self.dtype,
                input_scale=self.input_scale, ...
            )
        elif float_node.op_type == 'conv2d':
            return StaticQuantConv2dNode(...)
```

Quantized Node Structure

Base class: `src/pytorch_to_c/ir/quant_node.py`

```python
class QuantIRNode(IRNode):
    """Base class for all quantized operations"""

    @abstractmethod
    def get_pre_nodes(self) -> List[IRNode]:
        """Nodes to insert BEFORE this op (e.g., QuantizeNode)"""

    @abstractmethod
    def get_post_nodes(self) -> List[IRNode]:
        """Nodes to insert AFTER this op (e.g., DequantizeNode)"""

    @abstractmethod
    def generate_c_code(self, c_printer) -> str:
        """Generate the C code for this operation"""
```

Key insight: Each QuantIRNode controls its own conversion nodes!

StaticQuantLinearNode Implementation

File: src/pytorch_to_c/quantization/ops/quant_linear.py

```
class StaticQuantLinearNode(QuantIRNode):
    def __init__(self, name, dtype, input_scale, weight_scale, ...):
        super().__init__(name, 'linear', dtype=dtype)
        self.input_scale = input_scale
        self.weight_scale = weight_scale
        self.output_scale = output_scale
```

StaticQuantLinearNode Implementation

File: src/pytorch_to_c/quantization/ops/quant_linear.py

```
    def get_pre_nodes(self):
        return [QuantizeNode(
            name=f"{self.name}_input_q",
            dtype=self.dtype,
            scale=self.input_scale,
            offset=self.input_offset
        )]

    def get_post_nodes(self):
        return [DequantizeNode(
            name=f"{self.name}_output_dq",
            scale=self.output_scale,
            offset=self.output_offset
        )]
```

StaticQuantLinearNode Implementation

File: src/pytorch_to_c/quantization/ops/quant_linear.py

```
    def generate_c_code(self, c_printer):
        return f"dense_int8({input}, {size}, {weight}, " \
               f"{bias}, {out_size}, {self.input_scale}f, " \
               f"{self.weight_scale}f, 0, {output});"
```

QuantizationTransform Pipeline

File: src/pytorch_to_c/quantization/graph_transform.py

```python
class QuantizationTransform:
    def __init__(self, rules: List[QuantRule]):
        self.matcher = RuleMatcher(rules)

    def apply(self, ir_graph: IRGraph) -> IRGraph:
        # Step 1: Find nodes matching rules
        nodes_to_quantize = self._find_nodes_to_quantize(ir_graph)

        # Step 2: Replace float nodes with QuantIRNodes
        self._replace_nodes(ir_graph, nodes_to_quantize)

        # Step 3: Insert pre/post nodes (Quantize/Dequantize)
        self._insert_node_controlled_conversions(ir_graph)

        # Step 4: Validate output is float32
        self._validate_float_output(ir_graph)

        # Step 5: Quantize weights
        self._quantize_weights(ir_graph, nodes_to_quantize)

        return ir_graph
```

Adding a New Quantized Operation

Steps to add `QuantizedSoftmax`:

1. **Create the node class:**

```
class StaticQuantSoftmaxNode(QuantIRNode):
    def get_pre_nodes(self): ...
    def get_post_nodes(self): ...
    def generate_c_code(self, c_printer): ...
```

Adding a New Quantized Operation

Steps to add `QuantizedSoftmax`:

1. **Create the node class:**

```
class StaticQuantSoftmaxNode(QuantIRNode):
    def get_pre_nodes(self): ...
    def get_post_nodes(self): ...
    def generate_c_code(self, c_printer): ...
```

2. **Add C implementation:**

```
void softmax_int8(const int8_t* input, int size,
                  float scale, int offset, int8_t* output);
```

Adding a New Quantized Operation

Steps to add `QuantizedSoftmax`:

1. **Create the node class:**

```
class StaticQuantSoftmaxNode(QuantIRNode):
    def get_pre_nodes(self): ...
    def get_post_nodes(self): ...
    def generate_c_code(self, c_printer): ...
```

2. **Add C implementation:**

```
void softmax_int8(const int8_t* input, int size,
                  float scale, int offset, int8_t* output);
```

3. **Update rule to create it**
4. **Export in `__init__.py`**

Pass Infrastructure

Base class: src/passes/base.py

```python
class IRPass(ABC):
    def __init__(self, verbose: bool = False):
        self.verbose = verbose
        self.stats = {}  # Track optimization statistics

    @abstractmethod
    def apply(self, ir_graph: IRGraph) -> IRGraph:
        """Transform the IR graph"""
        pass

    def get_stats(self) -> Dict[str, Any]:
        """Return statistics about optimizations applied"""
        return self.stats
```

FuseDequantQuantPass Implementation

File: `src/passes/fuse_dequant_quant.py`

```python
class FuseDequantQuantPass(IRPass):
    def apply(self, ir_graph: IRGraph) -> IRGraph:
        # 1. Find fusable pairs
        pairs = self._find_fusable_pairs(ir_graph)

        # 2. Fuse each pair
        for dequant, quant in reversed(pairs):
            self._fuse_pair(ir_graph, dequant, quant)

        return ir_graph
```

FuseDequantQuantPass Implementation

File: `src/passes/fuse_dequant_quant.py`

```
    def _find_fusable_pairs(self, ir_graph):
        pairs = []
        for node in ir_graph.nodes:
            if isinstance(node, DequantizeNode):
                if len(node.users) == 1:
                    user = node.users[0]
                    if isinstance(user, QuantizeNode):
                        # Check: same dtype AND same scale
                        if (node.inputs[0].dtype == user.dtype and
                            self._scales_match(node, user)):
                            pairs.append((node, user))
        return pairs
```

Graph Rewiring in Passes

Before fusion:

```
A -> dequant -> quant -> B
```

After fusion:

```
A -> B
```

Rewiring code:

```
def _fuse_pair(self, ir_graph, dequant_node, quant_node):
    source = dequant_node.inputs[0]
    targets = quant_node.users.copy()

    # Rewire: source -> targets
    source.users.remove(dequant_node)
    for target in targets:
        target.inputs = [source if x == quant_node else x
                         for x in target.inputs]
        source.users.append(target)

    # Remove nodes from graph
    ir_graph.nodes.remove(dequant_node)
    ir_graph.nodes.remove(quant_node)
```

Writing Your Own Pass

Example: Dead Code Elimination Pass

```python
class DeadCodeEliminationPass(IRPass):
    """Remove nodes whose outputs are never used"""

    def apply(self, ir_graph: IRGraph) -> IRGraph:
        changed = True
        while changed:
            changed = False
            for node in ir_graph.nodes.copy():
                # Skip output nodes
                if node in ir_graph.outputs:
                    continue
                # Skip input nodes
                if node in ir_graph.inputs:
                    continue
                # If no users, node is dead
                if len(node.users) == 0:
                    self._remove_node(ir_graph, node)
                    changed = True
        return ir_graph
```

Pass Composition

Chain multiple passes:

```python
from src.passes import FuseDequantQuantPass

# Apply quantization
transform = QuantizationTransform(rules)
quant_ir = transform.apply(ir_graph)

# Apply optimization passes
passes = [
    FuseDequantQuantPass(verbose=True),
    # DeadCodeEliminationPass(),
    # ConstantFoldingPass(),
]

optimized_ir = quant_ir
for p in passes:
    optimized_ir = p.apply(optimized_ir)
    print(f"{p.__class__.__name__}: {p.get_stats()}")

# Generate final C code
CPrinter(optimized_ir).generate_all("output/")
```

Test Commands Summary

```
# All tests (64 tests)
python -m pytest test/ -v

# Float model tests
python -m pytest test/test_integration.py -v -s

# Quantization unit tests
python -m pytest test/test_quantization.py -v -s

# Quantization end-to-end tests
python -m pytest test/test_quantization_e2e.py -v -s

# Optimization pass tests
python -m pytest test/test_passes.py -v -s
```

Commands for commit 9362ae0

Key Numerical Checks

Test	Max Error	Test File

Key Numerical Checks

Test	Max Error	Test File
Float MLP	$\sim 10^{-6}$	`test_integration.py`

Key Numerical Checks

Test	Max Error	Test File
Float MLP	$\sim 10^{-6}$	`test_integration.py`
Float ResNet	1.19×10^{-7}	`test_quantization_e2e.py`

Key Numerical Checks

Test	Max Error	Test File
Float MLP	$\sim 10^{-6}$	`test_integration.py`
Float ResNet	1.19×10^{-7}	`test_quantization_e2e.py`
Static int8	$\sim$1–2%	`test_quantization_e2e.py`

Key Numerical Checks

Test	Max Error	Test File
Float MLP	$\sim 10^{-6}$	`test_integration.py`
Float ResNet	1.19×10^{-7}	`test_quantization_e2e.py`
Static int8	$\sim$1–2%	`test_quantization_e2e.py`
Static int16	$\sim$0.1%	`test_quantization_e2e.py`

Key Numerical Checks

Test	Max Error	Test File
Float MLP	$\sim 10^{-6}$	`test_integration.py`
Float ResNet	1.19×10^{-7}	`test_quantization_e2e.py`
Static int8	$\sim$1–2%	`test_quantization_e2e.py`
Static int16	$\sim$0.1%	`test_quantization_e2e.py`
FusePass	**0.00e+00**	`test_passes.py`

Key Numerical Checks

Test	Max Error	Test File
Float MLP	$\sim 10^{-6}$	`test_integration.py`
Float ResNet	1.19×10^{-7}	`test_quantization_e2e.py`
Static int8	$\sim$1–2%	`test_quantization_e2e.py`
Static int16	$\sim$0.1%	`test_quantization_e2e.py`
FusePass	**0.00e+00**	`test_passes.py`

FusePass is Bit-Identical!

When scales match, the optimization produces exactly the same numerical results.

Project File Structure

```
src/
  pytorch_to_c/
    compiler.py              # Main entry point
    frontend/fx_tracer.py    # torch.fx tracing
    lowering/lower.py        # FX -> IR conversion
    ir/
      node.py                # IRNode base class
      graph.py               # IRGraph
      quant_node.py          # QuantIRNode base
    codegen/c_printer.py     # C code generation
    quantization/
      rules.py               # QuantRule, StaticQuantRule, etc.
      graph_transform.py     # QuantizationTransform
      ops/                   # Quantized operation implementations
  passes/
    base.py                  # IRPass base class
    fuse_dequant_quant.py    # FuseDequantQuantPass
  c_ops/
    nn_ops_float.h           # Float C kernels
    nn_ops_int8.h            # Int8 C kernels
    nn_ops_int16.h           # Int16 C kernels
```

Summary

What we built:

- PyTorch → C compiler with torch.fx frontend

Summary

What we built:

- PyTorch $\rightarrow$ C compiler with torch.fx frontend
- Clean IR with doubly-linked node graph

Summary

What we built:

- PyTorch → C compiler with torch.fx frontend
- Clean IR with doubly-linked node graph
- Independent from source rule-based quantization

Summary

What we built:

- PyTorch → C compiler with torch.fx frontend
- Clean IR with doubly-linked node graph
- Independent from source rule-based quantization
- Support for int8 and int16

Summary

What we built:

- PyTorch $\rightarrow$ C compiler with torch.fx frontend
- Clean IR with doubly-linked node graph
- Independent from source rule-based quantization
- Support for int8 and int16
- Mixed precision quantization

Summary

What we built:

- PyTorch → C compiler with torch.fx frontend
- Clean IR with doubly-linked node graph
- Independent from source rule-based quantization
- Support for int8 and int16
- Mixed precision quantization
- Optimization pass infrastructure

Summary

What we built:

- PyTorch $\rightarrow$ C compiler with torch.fx frontend
- Clean IR with doubly-linked node graph
- Independent from source rule-based quantization
- Support for int8 and int16
- Mixed precision quantization
- Optimization pass infrastructure
- FuseDequantQuantPass (bit-identical results!)

Extensibility Points

How to extend the compiler:

Add new quantization:

- New `QuantRule` subclass
- New `QuantIRNode` implementation

Add new optimization:

- New `IRPass` subclass
- Add to pass pipeline

Add new operations:

- Add to lowering rules
- Add C kernel implementation

Add new datatypes:

- Extend `get_c_dtype()`
- Add C kernels for dtype

Run all 64 tests: `python -m pytest test/ -v`

Future Directions

Extending the compiler for adaptive runtime behavior:

1. **Resource-Aware Precision Switching**
   Dynamically switch between quantization precisions at runtime

Future Directions

Extending the compiler for adaptive runtime behavior:

1. **Resource-Aware Precision Switching**
   Dynamically switch between quantization precisions at runtime
2. **Adaptive Operation Implementations**
   Choose optimal op implementations based on system state

Future Directions

Extending the compiler for adaptive runtime behavior:

1. **Resource-Aware Precision Switching**
   Dynamically switch between quantization precisions at runtime
2. **Adaptive Operation Implementations**
   Choose optimal op implementations based on system state
3. **User-Defined Switching Metrics**
   Custom policies for runtime adaptation decisions

Resource-Aware Precision Switching

Goal: Enable runtime precision switching based on available resources

Concept:

- Compile model with multiple precision paths (int8, int16, float32)
- Switch between precisions based on battery, thermal, latency constraints

Resource-Aware Precision Switching

Goal: Enable runtime precision switching based on available resources

Concept:

- Compile model with multiple precision paths (int8, int16, float32)
- Switch between precisions based on battery, thermal, latency constraints

Example API:

```
compile_model(
    model,
    multi_precision=True,
    precision_options=['int8', 'int16', 'float32'],
    switching_policy='battery_aware'
)
```

Resource-Aware Precision Switching

Goal: Enable runtime precision switching based on available resources

Concept:

- Compile model with multiple precision paths (int8, int16, float32)
- Switch between precisions based on battery, thermal, latency constraints

Example API:

```
compile_model(
    model,
    multi_precision=True,
    precision_options=['int8', 'int16', 'float32'],
    switching_policy='battery_aware'
)
```

Generated C code:

```
if (battery_level < 20) {
    model_forward_int8(input, output);  // Fastest, lowest power
} else if (accuracy_required > 0.95) {
    model_forward_float32(input, output);  // Highest accuracy
} else {
                                           Balanced
```

Adaptive Operation Implementations

Goal: Switch between operation implementations based on system state

Use cases:

- Multi-threaded vs. single-threaded based on CPU availability
- SIMD-optimized vs. scalar based on processor features
- Memory-optimized vs. speed-optimized based on RAM pressure

Adaptive Operation Implementations

Goal: Switch between operation implementations based on system state

Use cases:

- Multi-threaded vs. single-threaded based on CPU availability
- SIMD-optimized vs. scalar based on processor features
- Memory-optimized vs. speed-optimized based on RAM pressure

Example: Threading strategy

```
void dense_adaptive(float* input, int size, float* weight,
                    float* output, runtime_context_t* ctx) {
    if (ctx->cpu_cores_available > 2 && size > 1024) {
        dense_multithreaded(input, size, weight, output, ctx);
    } else {
        dense_singlethreaded(input, size, weight, output);
    }
}
```

Adaptive Operation Implementations

Goal: Switch between operation implementations based on system state

Use cases:

- Multi-threaded vs. single-threaded based on CPU availability
- SIMD-optimized vs. scalar based on processor features
- Memory-optimized vs. speed-optimized based on RAM pressure

Example: Threading strategy

```
void dense_adaptive(float* input, int size, float* weight,
                    float* output, runtime_context_t* ctx) {
    if (ctx->cpu_cores_available > 2 && size > 1024) {
        dense_multithreaded(input, size, weight, output, ctx);
    } else {
        dense_singlethreaded(input, size, weight, output);
    }
}
```

Compiler directive:

```
@adaptive_implementation(variants=['single', 'multi', 'simd'])
def forward(self, x):
```

User-Defined Switching Metrics

Goal: Allow users to define custom policies for runtime adaptation

Custom metric system:

```
class CustomSwitchingPolicy:
    def should_switch_precision(self, layer_name, current_metrics):
        latency = current_metrics['latency']
        accuracy = current_metrics['accuracy']

        if layer_name.startswith('critical_'):
            return 'float32' if accuracy < 0.99 else 'int16'

        return 'int8' if latency > 50 else 'int16'

    def should_switch_implementation(self, op_name, system_state):
        if system_state['temperature'] > 70:
            return 'low_power'
        return 'high_performance'
```

User-Defined Switching Metrics

Goal: Allow users to define custom policies for runtime adaptation

Custom metric system:

```
class CustomSwitchingPolicy:
    def should_switch_precision(self, layer_name, current_metrics):
        latency = current_metrics['latency']
        accuracy = current_metrics['accuracy']

        if layer_name.startswith('critical_'):
            return 'float32' if accuracy < 0.99 else 'int16'

        return 'int8' if latency > 50 else 'int16'

    def should_switch_implementation(self, op_name, system_state):
        if system_state['temperature'] > 70:
            return 'low_power'
        return 'high_performance'
```

Register policy:

```
compile_model(model, switching_policy=CustomSwitchingPolicy())
```

Benefits and Challenges

Benefits:

- **Adaptability:** Model adapts to runtime conditions

Challenges:

Benefits and Challenges

Benefits:

- **Adaptability:** Model adapts to runtime conditions
- **Efficiency:** Optimal resource usage

Challenges:

Benefits and Challenges

Benefits:

- **Adaptability:** Model adapts to runtime conditions
- **Efficiency:** Optimal resource usage
- **Flexibility:** User-defined policies

Challenges:

Benefits and Challenges

Benefits:

- **Adaptability:** Model adapts to runtime conditions
- **Efficiency:** Optimal resource usage
- **Flexibility:** User-defined policies
- **Performance:** Balance speed vs. accuracy

Challenges:

Benefits and Challenges

Benefits:

- **Adaptability:** Model adapts to runtime conditions
- **Efficiency:** Optimal resource usage
- **Flexibility:** User-defined policies
- **Performance:** Balance speed vs. accuracy

Challenges:

- Code size increases (multiple implementations)

Benefits and Challenges

Benefits:

- **Adaptability:** Model adapts to runtime conditions
- **Efficiency:** Optimal resource usage
- **Flexibility:** User-defined policies
- **Performance:** Balance speed vs. accuracy

Challenges:

- Code size increases (multiple implementations)
- Runtime overhead for decision logic

Benefits and Challenges

Benefits:

- **Adaptability:** Model adapts to runtime conditions
- **Efficiency:** Optimal resource usage
- **Flexibility:** User-defined policies
- **Performance:** Balance speed vs. accuracy

Challenges:

- Code size increases (multiple implementations)
- Runtime overhead for decision logic
- Complexity in debugging/testing

Benefits and Challenges

Benefits:

- **Adaptability:** Model adapts to runtime conditions
- **Efficiency:** Optimal resource usage
- **Flexibility:** User-defined policies
- **Performance:** Balance speed vs. accuracy

Challenges:

- Code size increases (multiple implementations)
- Runtime overhead for decision logic
- Complexity in debugging/testing
- Policy design requires domain knowledge

Thank You!

Questions?

https://github.com/ashitabh8/Tiny-NN-in-C.git