Quantization Flow

A quantization flow quantizes a floating point PyTorch model to a quantized model for faster inference speed and less resource consumption based on:

- Quantization configurations on how the model should be quantized
- Backend configurations that describes the supported quantization capabilities on the target backend

Advantages over FX Graph Mode Quantization

- Higher Model Coverage: FX Graph Mode Quantization relies on fx symbolic tracing to capture the graph, PyTorch 2.0 Export relies on torchdynamo for capturing which is expected to have a higher model coverage
- More Robust Quantization: With torchdynamo, we also have more information such as type information for each node in the model graph, which makes building a more robust quantization transformation possible

API Example

```python
example_inputs = (torch.randn(1, 5),)
m = M().eval()
# program capture
m = program_capture(m, example_inputs)
# we have a model with aten ops in Edge-ATen dialect
# quantization
qconfig_mapping = get_default_qconfig_mapping("executorch")
backend_config = get_executorch_backend_config()
m = prepare_pt2(m, qconfig_mapping, backend_config)
# calibration omitted
m = convert_to_reference_pt2(m)
# we have a model with aten ops + quantize/dequantize ops
# lowering
m = delegation_lowering(m)
# 1. quantization fusion
quantFusionPass = the pass that fuses reference quantized patterns into canonical quantized operator provided in executorch
m = quantFusionPass(m)
# we have a model with aten ops + quantized ops from canonical quantized operator library
```

How Modeling Developers Configure Quantization

1. Define a BackendConfig that represents the supported type of quantization for operators, for example, int8 static quantization for add operator, int8 activation and weight static quantization for linear operator.
2. For quantized operators that have state (e.g. fbgemm/qnnpack operators) or that need to execute through special runtime (e.g. GPU), lower through delegation
3. For other quantized operators, either implement the operator in a separate quantized operator library or in delegation
4. Implement lowering from reference quantized patterns to delegated or quantized ops

How Backend Developers Integrate with the Flow

1. Define a BackendConfig that represents the supported type of quantization for operators, for example, int8 static quantization for add operator, int8 activation and weight static quantization for linear operator.
2. For quantized operators that have state (e.g. fbgemm/qnnpack operators) or that need to execute through special runtime (e.g. GPU), lower through delegation
3. For other quantized operators, either implement the operator in a separate quantized operator library or in delegation
4. Implement lowering from reference quantized patterns to delegated or quantized ops

Timeline

- 12.2022: Early Prototype (available in PyTorch master)
- 06.2023: Prototype Release