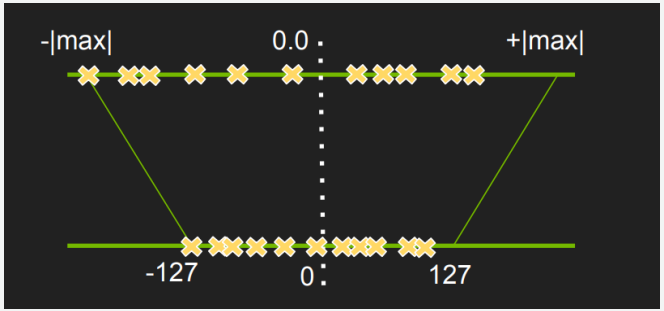
Compilation of Quantized Models in TVM

Animesh Jain
Amazon SageMaker Neo
AWS AI

Quantization Overview

- Represent FP32 numbers with a lower-precision INT8 numbers
- Integer number stands as a proxy for FP32 number (not a downcast)



- Quantized tensor is represented with **a scale and a zero point**

$$real\_value = scale * (quantized\_value - zero\_point)$$

http://on-demand.gputechconf.com/gtc/2017/presentation/s7310-8-bit-inference-with-tensorrt.pdf

aws

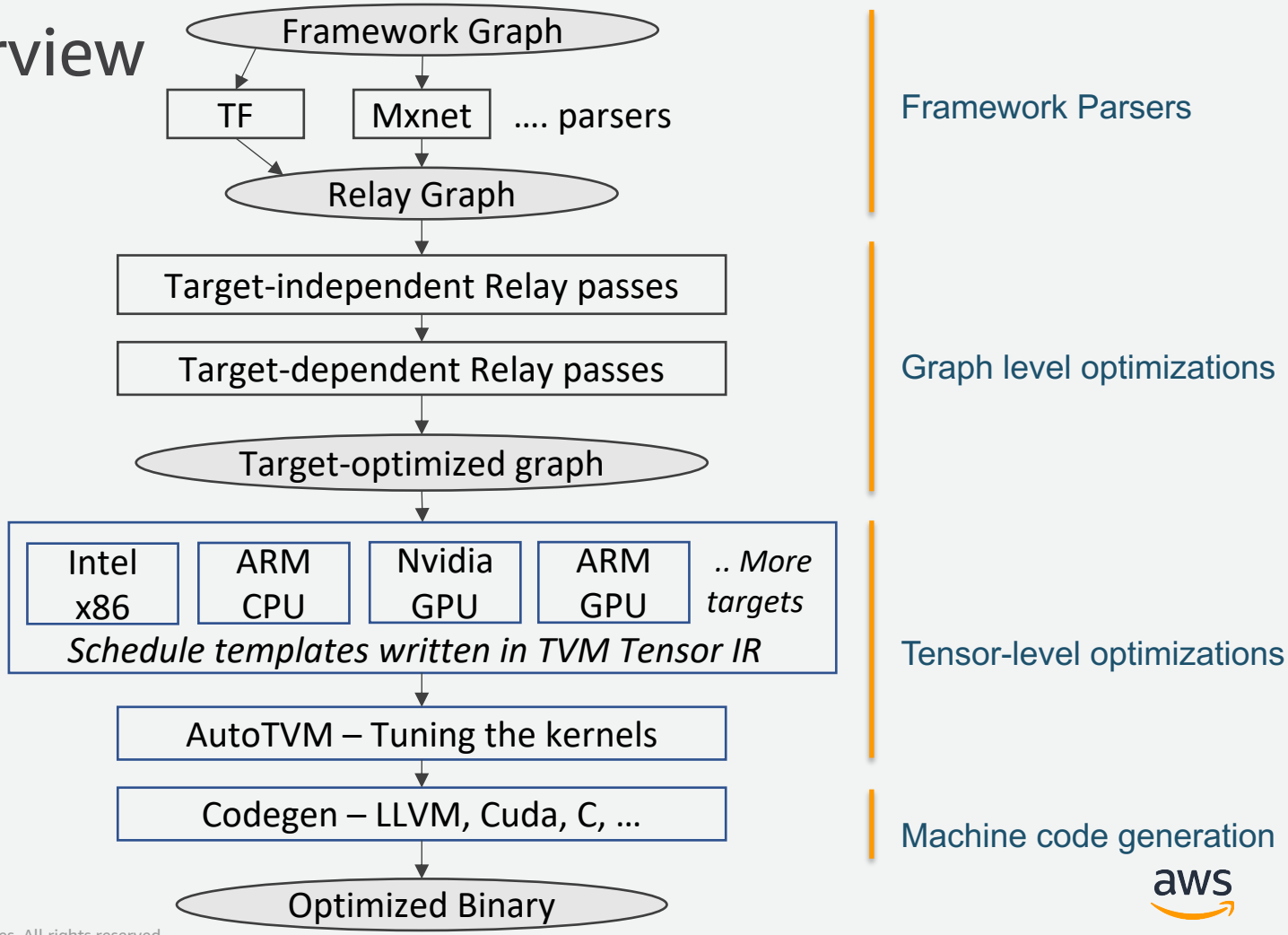
Quantization in TVM

- Quantization within TVM - Automatic Quantization
  - TVM stack ingests a FP32 graph and a small dataset
  - Finds suitable quantization scale
  - Produces a quantized graph

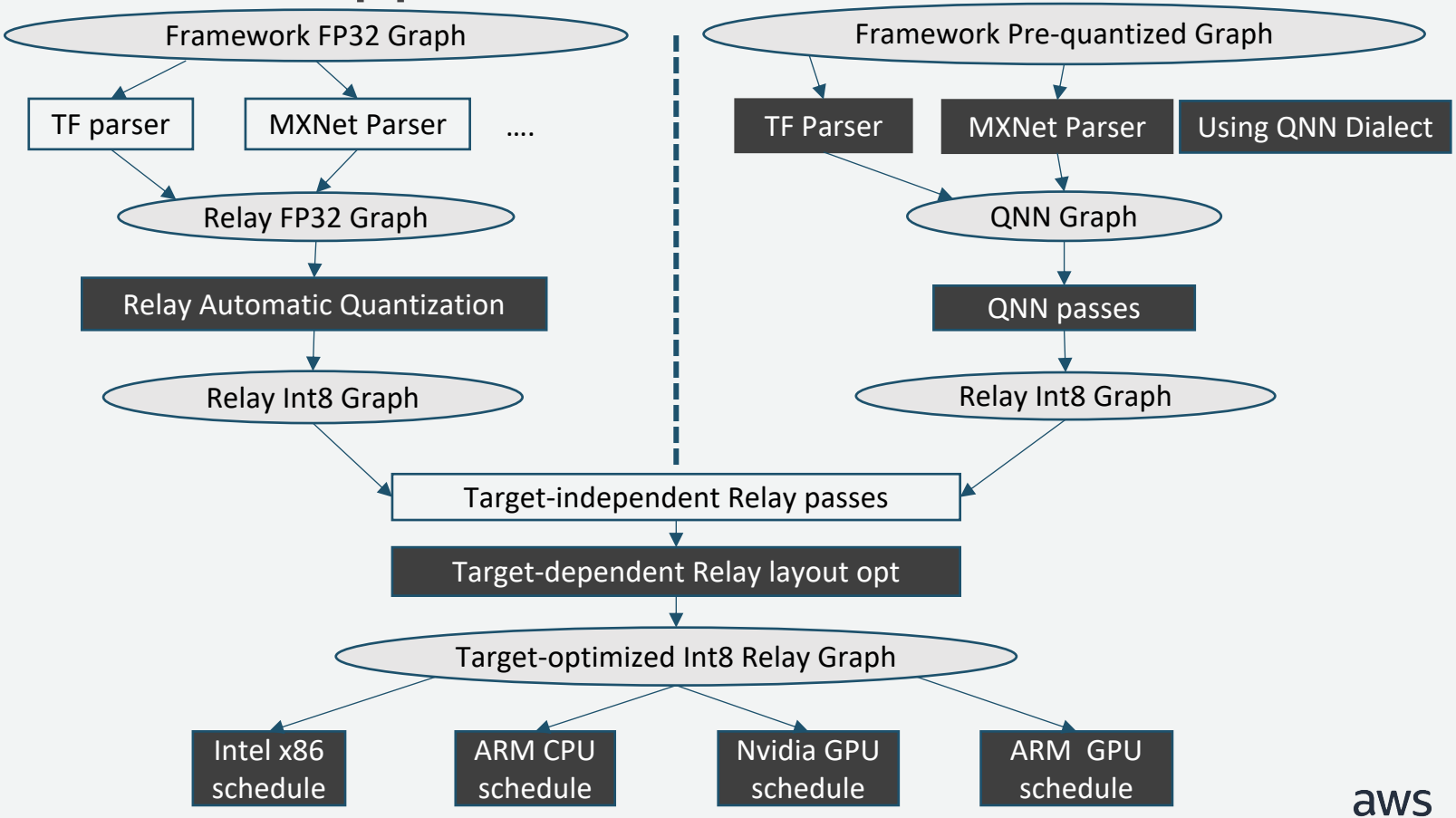
- Compiling Pre-quantized models – QNN Dialect
  - TVM ingests a pre-quantized graph in TFLite or MxNet
  - Use high-level wrapper ops of QNN dialect

aws

TVM Overview



Quantization Appraoches in TVM



Outline

- QNN Dialect
  - Design
  - Operators
- Results on Intel Cascade Lake

Quantized Operators in Framework

- New operators like TF quantized_conv2d
  - Underlying calculations are different than FP32 conv2d

$$real\_value = \boldsymbol{scale} * (quantized\_value - \boldsymbol{zero\_point})$$

- Sometimes operators are aggressively fused
  - TFLite fuses quantized_conv2d, bias, relu and requantize

How to Support Framework Quantized Operators?

Option 1 – Completely add new ops from scratch

- New Relay passes and TVM schedules required
  - AlterOpLayout, Graph Fusion etc require work/operator
- No reuse of existing Relay and TVM infrastructure.

Option 2 – Lower to a sequence of existing Relay operators

- We introduced a new Relay dialect – QNN to encapsulate this work
- Complete reuse of Relay pass infrastructure
- Possible reuse of TVM schedules (only to some extent)

aws

QNN Dialect

- Design operators that satisfy many framework operators
  - qnn.quantize, qnn.dequantize, qnn.requantize
  - qnn.conv2d, qnn.dense
  - qnn.concatenate
  - qnn.add, qnn.mul
- QNN operators will be lowered to Relay operators
- QNN Optimization passes
  - Some optimizations are easier at QNN level
  - Intel x86 VNNI requires conv input dtypes to uint8 x int8

aws

Lowering of QNN Quantize Operator

```
fn (%input_data: Tensor[(2, 5), float32]) {
  qnn.quantize(%input_data, out_dtype="uint8", output_zero_point=127, output_scale=0.5f)
}
```

```
def @main(%input_data: Tensor[(2, 5), float32]) -> Tensor[(2, 5), uint8] {
  %0 = divide(%input_data, 0.5f /* ty=float32 */) /* ty=Tensor[(2, 5), float32] */;
  %1 = round(%0) /* ty=Tensor[(2, 5), float32] */;
  %2 = cast(%1, dtype="int32") /* ty=Tensor[(2, 5), int32] */;
  %3 = add(%2, 127 /* ty=int32 */) /* ty=Tensor[(2, 5), int32] */;
  %4 = clip(%3, a_min=0f, a_max=255f) /* ty=Tensor[(2, 5), int32] */;
  cast(%4, dtype="uint8") /* ty=Tensor[(2, 5), uint8] */
}
```

QNN Conv2D Operator

- Calculations are different from FP32 Conv2D

$$real\_value = \boldsymbol{scale} * (quantized\_value - \boldsymbol{zero\_point})$$

$$\Sigma c,r,s\ (Q_W(k, c, r, s) - zp\_w) \times (Q_A(n, c, h + r, w + s) - zp\_a)$$

$$\begin{aligned}
&\Sigma c,r,s\ Q_W(k, c, r, s) \times Q_A(n, c, h + r, w + s) && \text{// Term 1}\\
&- \Sigma c,r,s\ zp_a \times Q_W(k, c, r, s) && \text{// Term 2}\\
&- \Sigma c,r,s\ zp_w \times Q_A(n, c, h + r, w + s) && \text{// Term 3}\\
&+ \Sigma c,r,s\ zp_a \times zp_w && \text{// Term 4}
\end{aligned}$$

https://discuss.tvm.ai/t/tf-lite-quantized-conv2d-operator-conversion/2651/8

Lowering of QNN Conv2D Operator

```
fn (%data: Tensor[(1, 3, 2, 3), uint8], %weight: Tensor[(3, 3, 2, 2), uint8]) {
  qnn.conv2d(%data, %weight, … , out_dtype="int32", input_zero_point=1, kernel_zero_point=1)}
```

```
def @main(%data: Tensor[(1, 3, 2, 3), uint8], %weight: Tensor[(3, 3, 2, 2), uint8]) -> Tensor[(1, 3, 1, 2), int32] {
  %0 = nn.conv2d(%data, %weight, … , out_dtype="int32") /* ty=Tensor[(1, 3, 1, 2), int32] */;
  %1 = cast(%data, dtype="int32") /* ty=Tensor[(1, 3, 2, 3), int32] */;
  %2 = multiply(%1, 4 /* ty=int32 */) /* ty=Tensor[(1, 3, 2, 3), int32] */;
  %3 = nn.avg_pool2d(%2, pool_size=[2, 2]) /* ty=Tensor[(1, 3, 1, 2), int32] */;
  %4 = sum(%3, axis=[1], keepdims=True) /* ty=Tensor[(1, 1, 1, 2), int32] */;
  %5 = multiply(1 /* ty=int32 */, %4) /* ty=Tensor[(1, 1, 1, 2), int32] */;
  %6 = subtract(%0, %5) /* ty=Tensor[(1, 3, 1, 2), int32] */;
  %7 = cast(%weight, dtype="int32") /* ty=Tensor[(3, 3, 2, 2), int32] */;
  %8 = sum(%7, axis=[1, 2, 3]) /* ty=Tensor[(3), int32] */;
  %9 = reshape(%8, newshape=[1, 3, 1, 1]) /* ty=Tensor[(1, 3, 1, 1), int32] */;
  %10 = multiply(1 /* ty=int32 */, %9) /* ty=Tensor[(1, 3, 1, 1), int32] */;
  %11 = subtract(12 /* ty=int32 */, %10) /* ty=Tensor[(1, 3, 1, 1), int32] */;
  add(%6, %11) /* ty=Tensor[(1, 3, 1, 2), int32] */}
```

Asymmetric

For zero-centered zero point, the lowering will have just nn.conv2d

aws

Frontend Parsers

- TFLite Pre-quantized Models
  - In good shape
  - Supports all Image Classification PreQuantized hosted models
- MXNet Pre-quantized Models
  - Tested internally with MxNet + MKLDNN path
  - Will open RFC in a month

Evaluation

- Intel Cascade Lake 12-core Server
- TFLite Pre-quantized Hosted Models

Accuracy

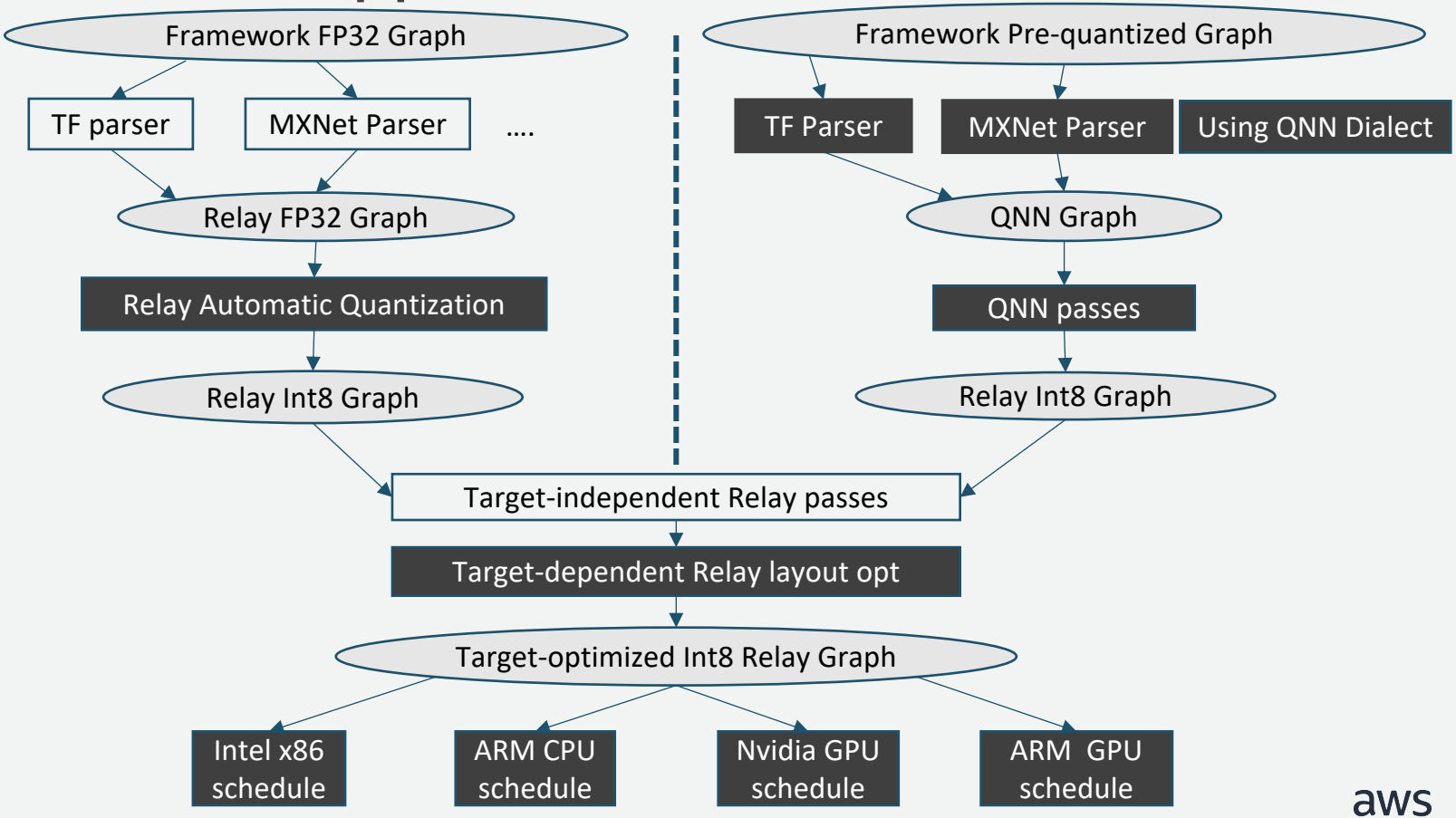
Quantized models	TFLite graph via TVM	Reported TFLite accuracy
Inception V1	69.6%/89.5%	70.1%/89.8%
Inception V2	73.3%/91.3%	73.5%/91.4%
Inception V3	77.3%/93.6%	77.5%/93.7%
Inception V4	79.6%/94.2%	79.5%/93.90%
Mobilenet V1	70.1%/89.0%	70.0%/89.0%
Mobilenet V2	70.9%/90.1%	70.8%/89.9%

Performance Comparison

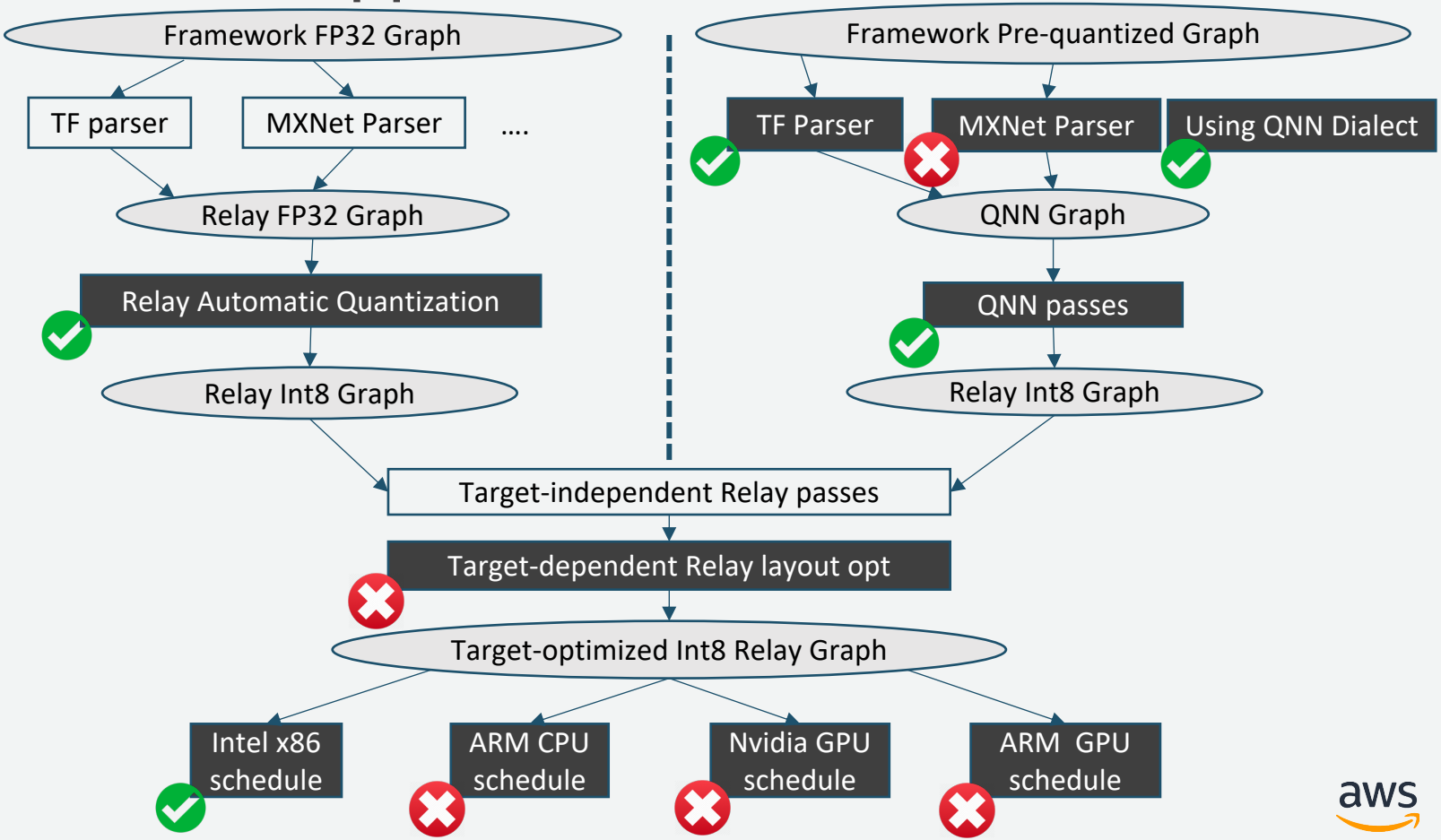
	Float32	Quantized	Speedup
Inception V1	NA	2.143	NA
Inception V2	NA	8.919	NA
Inception V3	10.373	6.115	1.7
Inception V4	21.233	12.315	1.72
Mobilenet V1	5.578	7.328	0.76
Mobilenet V2	6.424	8.798	0.73

- Metric – Latency in ms for batch size = 1
- 1.7x speedup on Inception **asymmetric** quantized model
- Mobilenet requires depthwise convolution VNNI schedule
- **Symmetric** model improves the speedup to 2.8x

Quantization Approaches in TVM



Quantization Approaches in TVM



Conclusion

- TVM community is pursuing both Automatic- and Pre-quantized model support. Contributions are welcomed.
- We need new/tuned TVM schedules using fast Integer operations like Intel VNNI, ARM Dot, Nvidia DP4A
- Full pipeline is available. Please try it and give suggestions.
- Open-source discussions formed the foundations of both the approaches.

aws