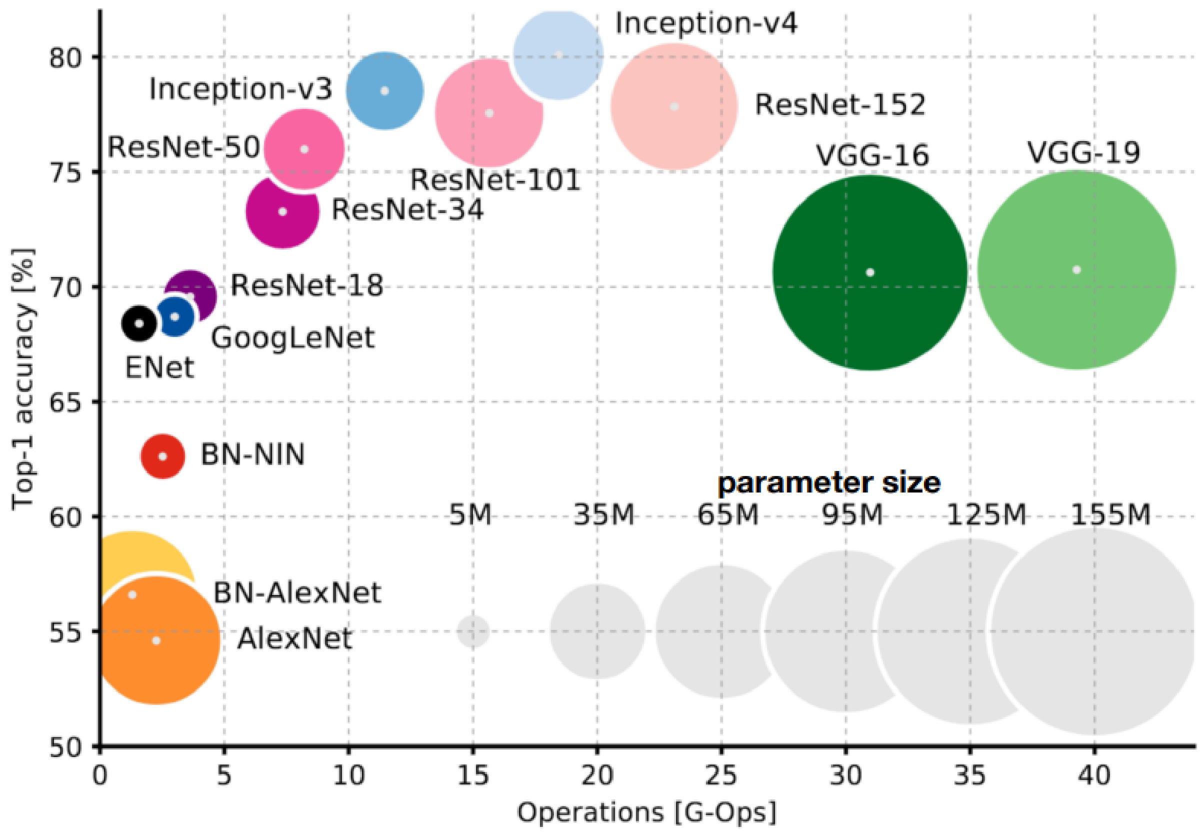
# Riptide: Fast End-to-End Binarized Neural Networks

Josh Fromm, Meghan Cowan, Matthai Philipose, Luis Ceze, and Shwetak Patel

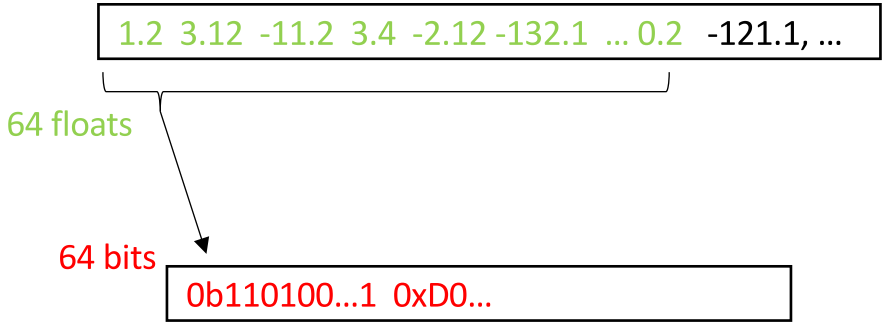



| | Device / Platform | GFLOPS | GFLOPS/W | Sec/Frame | WS/Frame | Cost |
|---|---|---|---|---|---|---|
| 1 | Raspberry Pi-zero | 0.32 | 0.24 | 56.42 | 75.01 | $22 |
| 2 | Raspberry Pi3 | 3.62 | 0.81 | 4.97 | 22.14 | $37 |
| 3 | Snapdragon 835 | 11.5 | 1.44 | 1.56 | 12.50 | $13 (bulk) |
| 4 | Haswell i7-4790 | 181 | 1.68 | 0.10 | 10.71 | $380 |
| 5 | Titan V | 15000 | 60 | 0.0012 | .3 | $3000 |

Canziani et al., "An analysis of deep neural network models for practical applications." 2016

# 1-bit Matrix Operations

- Quantize floats to +/-1
- 1.122 * -3.112 ==> 1 * -1
- Notice:
  - 1 * 1 = 1
  - 1 * -1 = -1
  - -1 * 1 = -1
  - -1 * -1 = 1
- Replacing -1 with 0, this is just XNOR
- Retrain model to convergence




$$A[:64] \,.\, W[:64] == \text{popc}(A_{/64} \text{ XNOR } W_{/64})$$

Rastegari et al., "Xnor-net: Imagenet classification using binary convolutional neural networks." 2016

# 1-bit Matrix Operations: Cost/Benefit

```
float x[], y[], w[];
...
for i in 1…N:
    y[j] += x[i] * w[i];
```

2N ops

⬇

```
unsigned long x[], y[], w[];
…
for i in 1…N/64:
    y[j] += 64 – 2*popc(not(x_b[i] xor w_b[i]));
```

3N/64 ops

~40x faster
32x smaller

**Typically, lose ~10% accuracy**

Rastegari et al., “Xnor-net: Imagenet classification using binary convolutional neural networks.” 2016

# 1-bit Matrix Operations: Cost/Benefit

```
float x[], y[], w[];
...
for i in 1…N:
    y[j] += x[i] * w[i];
```

2N ops

```
unsigned long x[], y[], w[];
…
for i in 1…N/64:
    y[j] += 64 – 2*popc(not(x_b[i] xor w_b[i]));
```

3N/64 ops

~40x faster
32x smaller

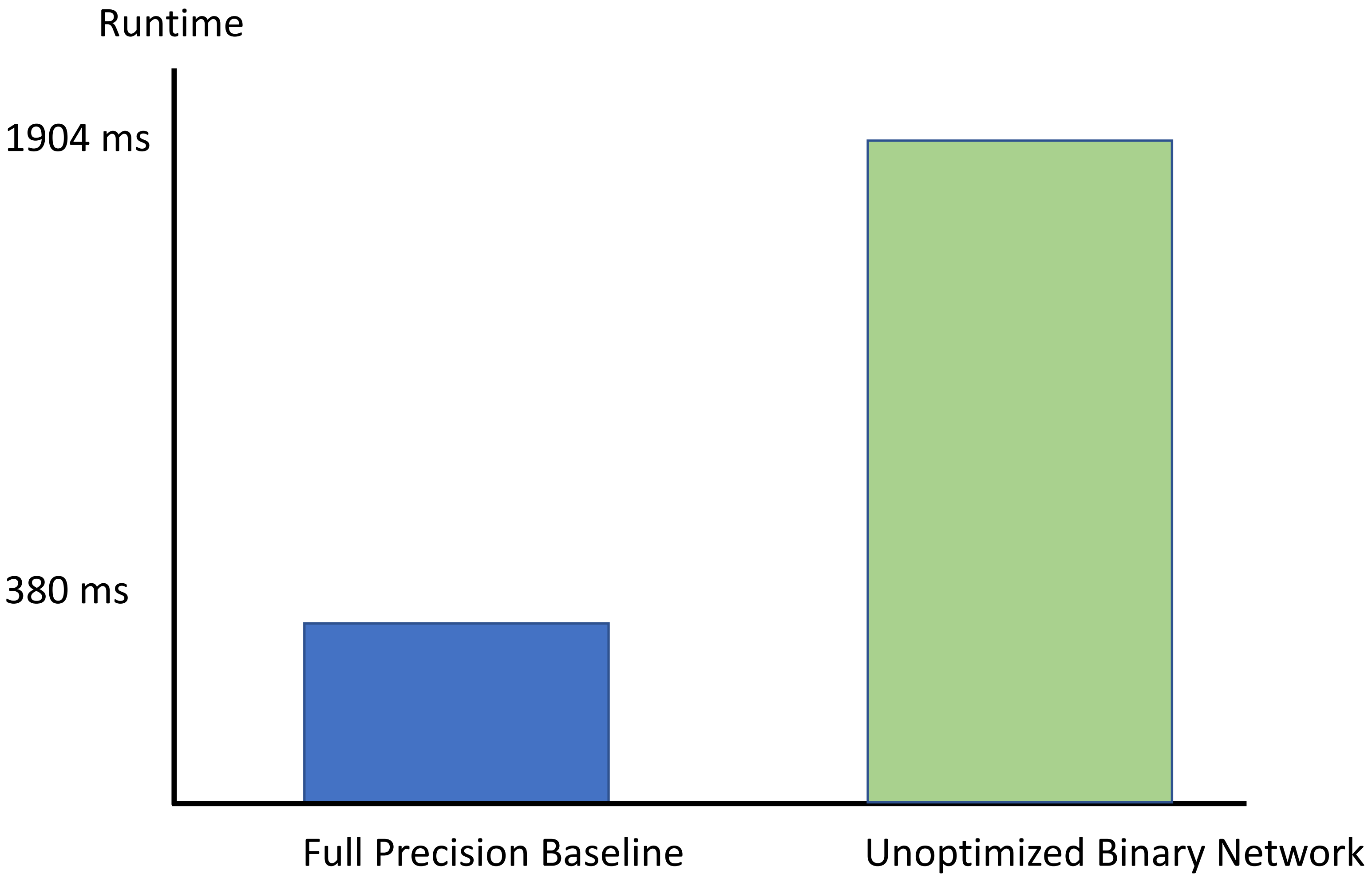
**Typically, lose ~10% accuracy**

Rastegari et al., “Xnor-net: Imagenet classification using binary convolutional neural networks.” 2016

# 1-bit Matrix Operations: Cost/Benefit

~~~40x faster~~
~~~

# 1-bit Matrix Operations: Cost/Benefit



# Implementation Challenges

**No optimized linear algebra libraries like BLAS to leverage**

~~uint1 uint2~~

Need to implement optimizations from scratch
Optimizations tuned for specific CPU

CPUs have no native support for low bit data types
Need to work on packed data

**Baselines incredibly well optimized**


Optimized linear algebra libraries
Hardware support for conventional deep learning

# Are Binary Networks Actually Fast?

Majority of work in binarization is simulated

- Which binarization techniques can be implemented efficiently?
- What are the runtime bottlenecks in a binary model?
- How do I deploy a fast binary model on my platform?

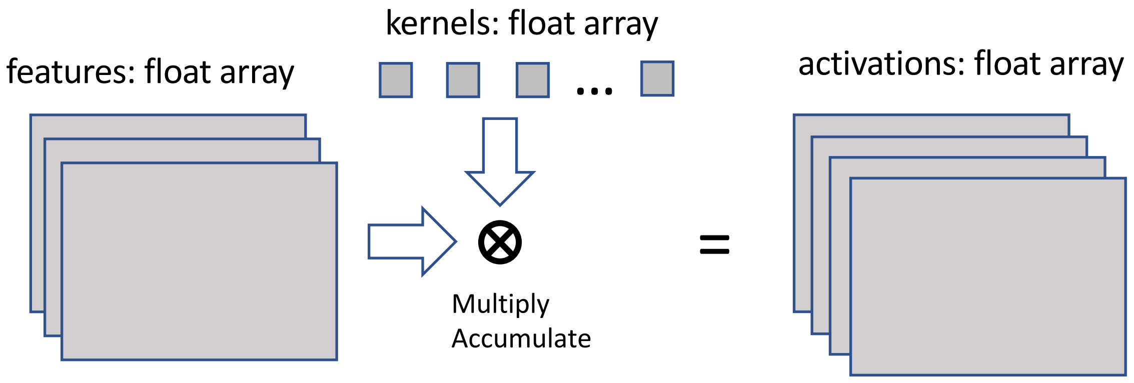
**To address these questions we introduce Riptide.**

# Riptide

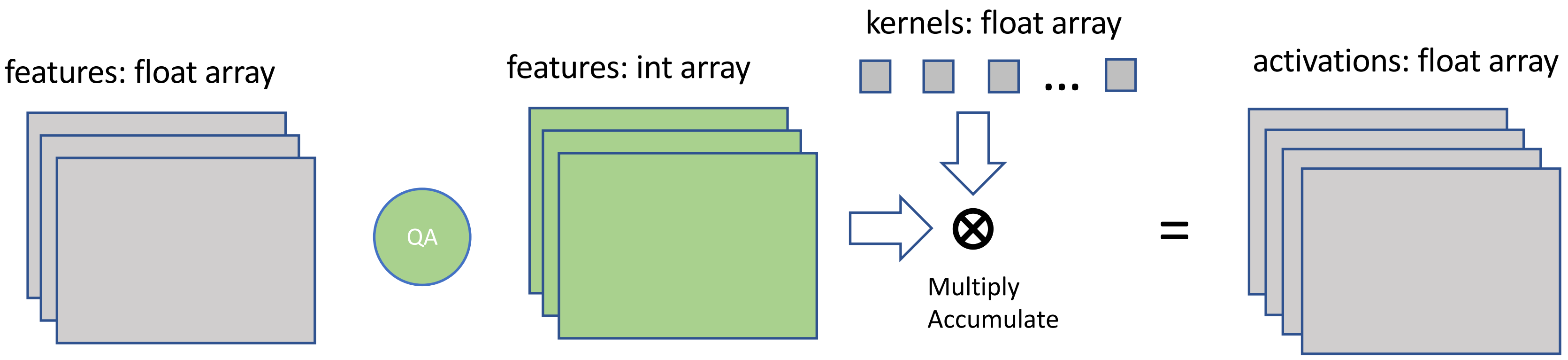
A one-stop solution to training and deploying fast binary networks on a variety of hardware platforms.

- Addresses implementation issues in mixed polarity quantization
- Introduces the Fused Glue operation, removing all floating-point arithmetic from binary models.
- Provides high-performance bitserial operators through TVM.
- Yields 4-12X speedups across various models and bitwidths while maintaining state-of-the-art accuracy.
- Available open-source today at github.com/jwfromm/Riptide

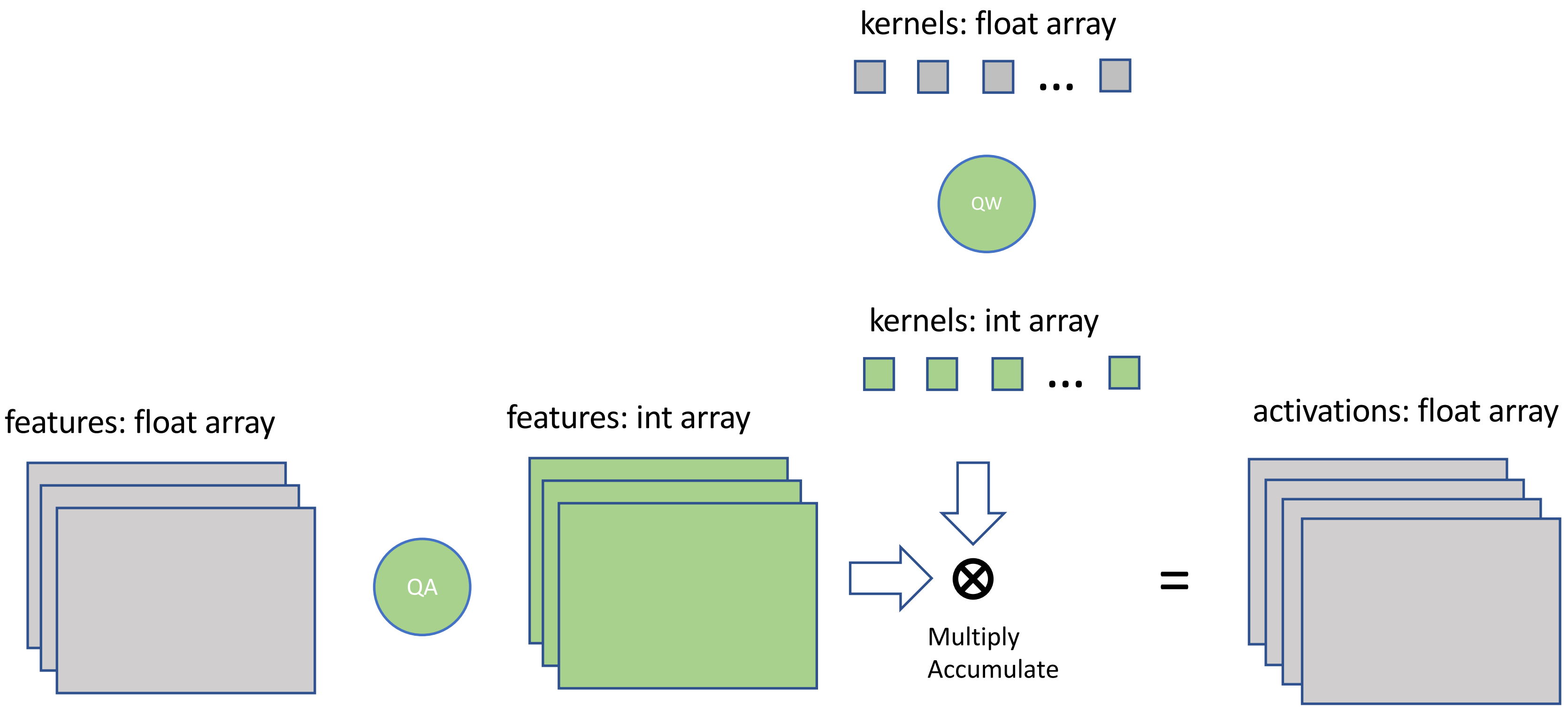
# Implementing Binary Layers



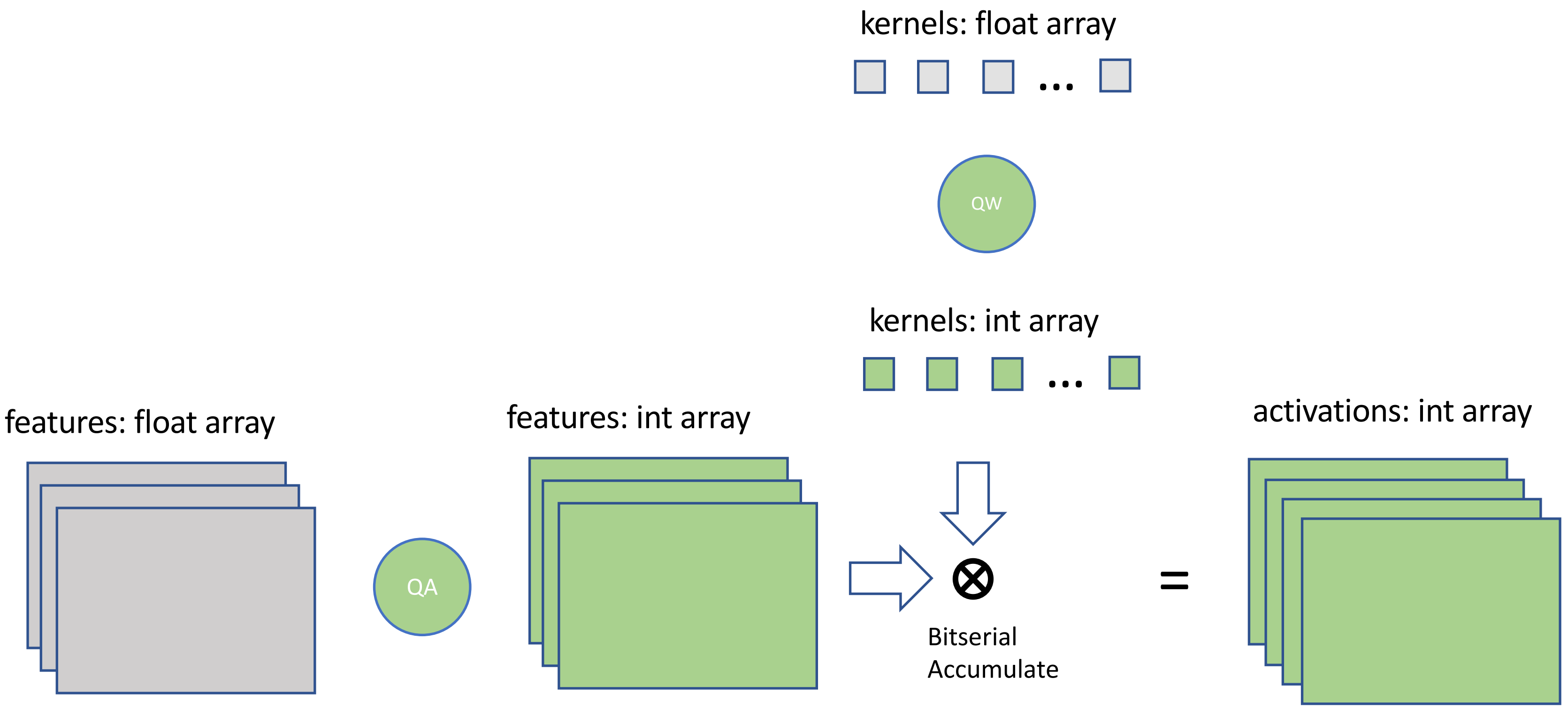
# Implementing Binary Layers



# Implementing Binary Layers

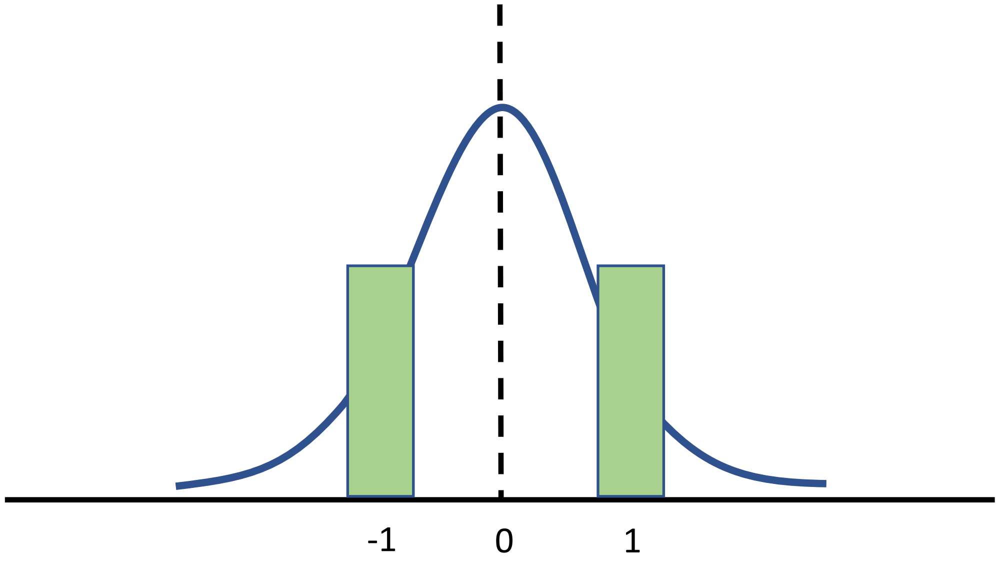


# Implementing Binary Layers



# Quantization Polarity

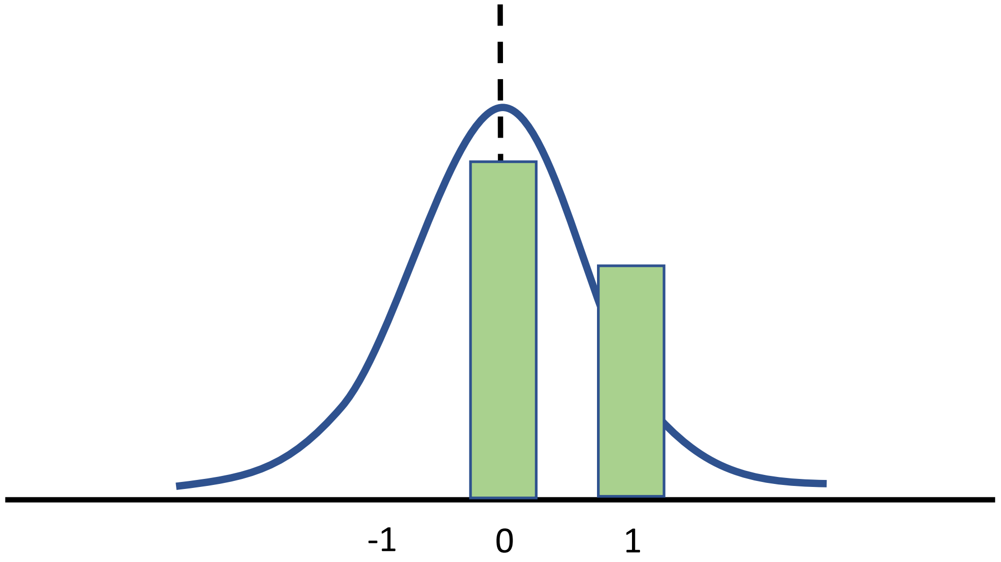
Bipolar Quantization




Quantization Function: $\tilde{x} = sign(x)$

- Implemented with bitwise-xnor and popcount
- Well-suited for weights, which represent correlation (1) or inverse-correlation (-1)

Unipolar Quantization




Quantization Function: $\tilde{x} = x > 0$

- Implemented with bitwise-and and popcount
- Well-suited for activations, which represent pattern-match (1) or no pattern-match (0)

# Quantization Polarity

- XnorNet (all bipolar) -> 44.2% accuracy
- DorefaNet (bipolar weights unipolar activations) -> 50.0% accuracy

| | |
|---|---|
| A (unipolar) | 1 1 0 1 0 1 … |
| | $\otimes$ |
| W (bipolar) | 0 1 0 0 1 0 … |
| | = |
| Expected | 0 1 0 -1 0 -1 … |

Multiple meanings of 0 bits causes mixed polarity to unimplementable

Zhou et al., “DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients.” 2016

# Mixed Polarity Operation

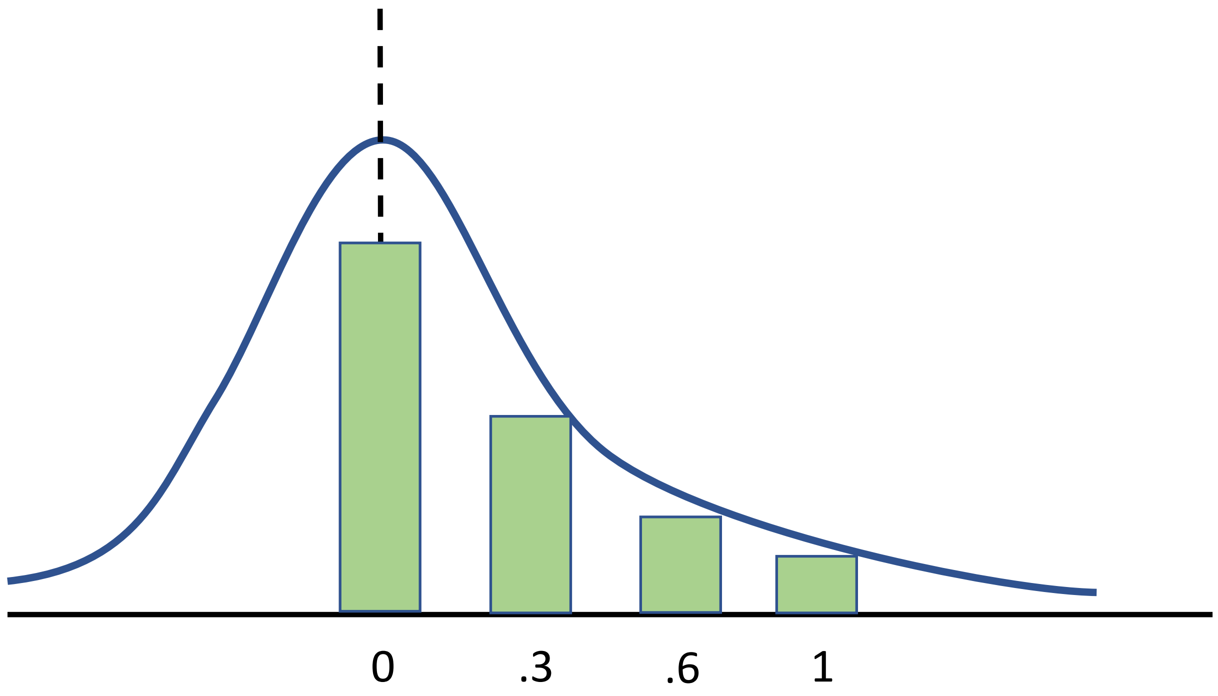
$$a \cdot w = \sum_{n=0}^{N-1} 2^N (popc(a_n \wedge w) - popc(a_n \wedge !w))$$

Count number of bit multiplications where output should be 1

Subtract cases where output should be -1

- Enables mixed polarity binary networks
- Doubles amount of inner loop compute but does not require additional memory operations
- Mixed polarity may offer compelling points on speedup to accuracy versus pure bipolar

# Multibit Quantization

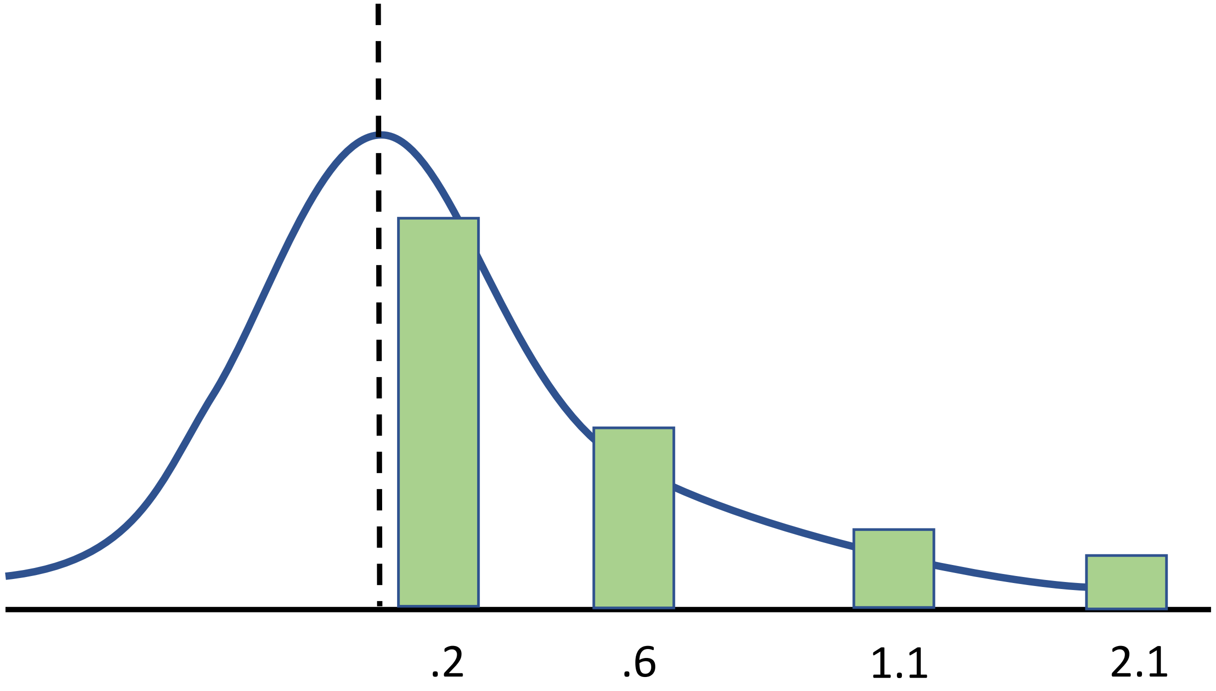



Quantization Function: $\tilde{x} = linear(x)$

- Translates naturally to integer representation
- Does not necessarily fit distribution

Zhou et al., "DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients." 2016

# Multibit Quantization

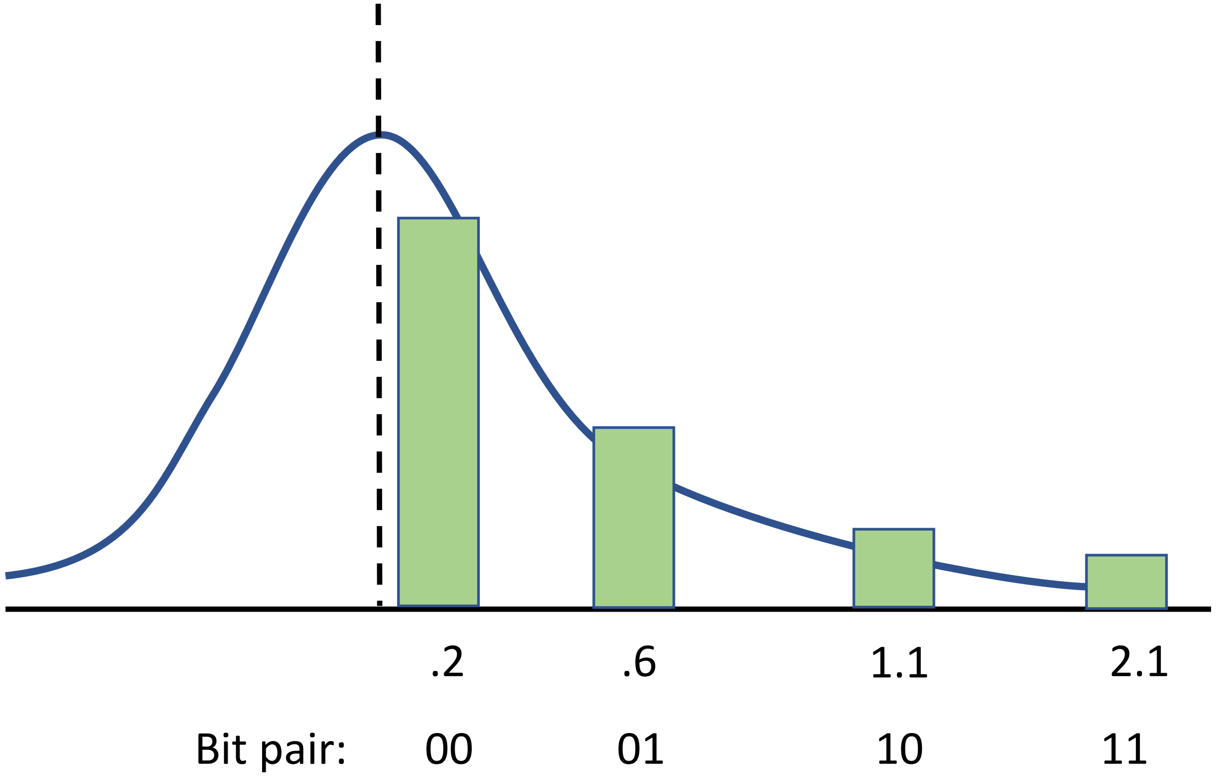



Quantization Function: $\tilde{x} = HWGQ(x)$

- Better fit for Gaussian distribution
- Not implementable

Cai et al., "Deep Learning with Low Precision by Half-wave Gaussian Quantization." 2017

# Multibit Quantization




Value is based on unique bit pair rather than combination of bits, $(01 + 10 \neq 11)$

Original Data uint32

| | | | |
|---|---|---|---|
| 2 | 0 | 1 | 3 |

| | | | |
|---|---|---|---|
| 0 | 0 | 1 | 1 |

Bitplanes uint1

| | | | |
|---|---|---|---|
| 0 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 |

| | | | |
|---|---|---|---|
| 0 | 0 | 1 | 1 |

Bitpacked Data uint4

0011 = 3
1001 = 9

0011 = 3

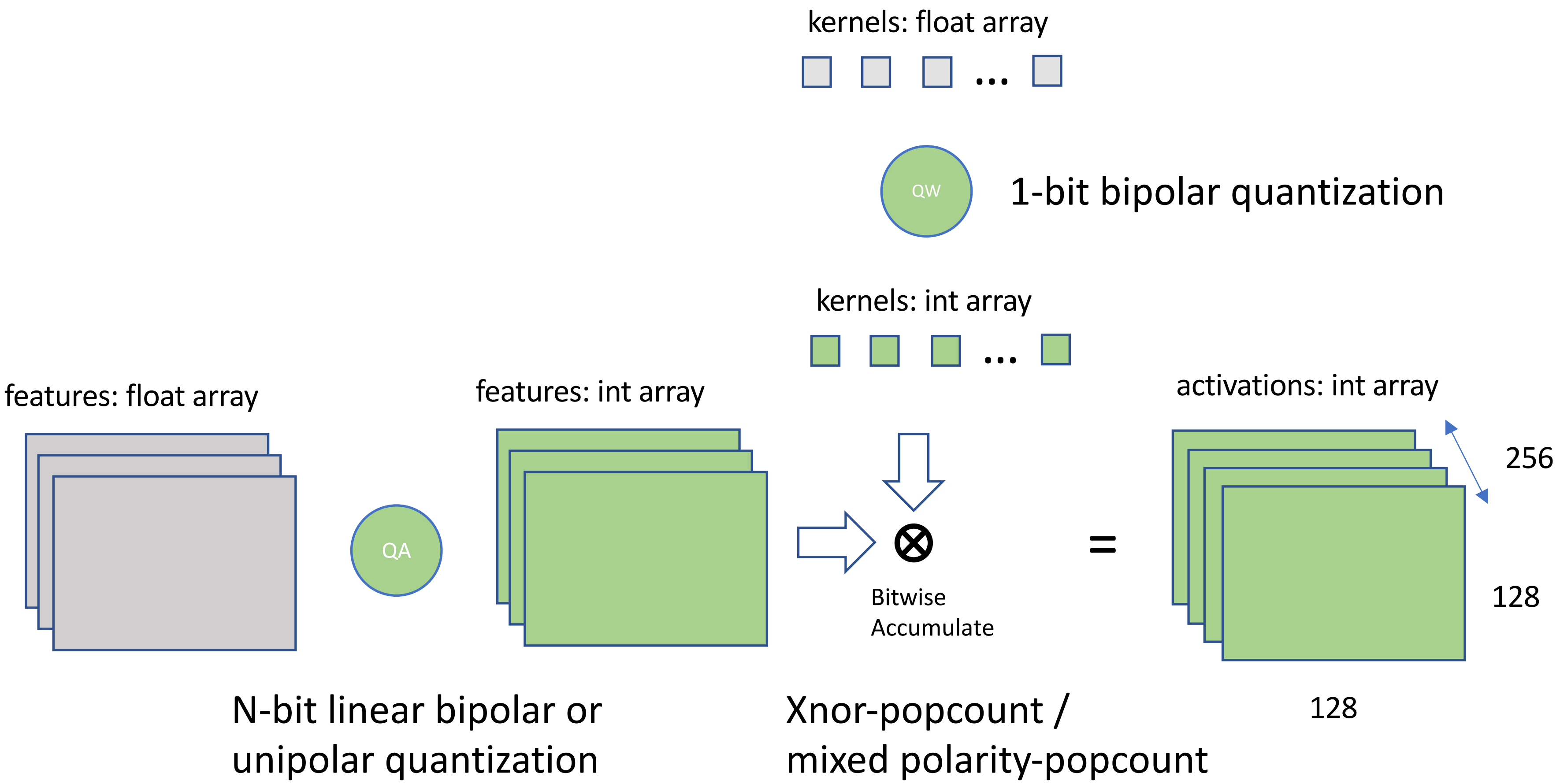
Bitserial Dot Product

$1\times \boxed{\text{popcount}(3\&3)} + 2\times \text{popcount}(3\&9) = 4$

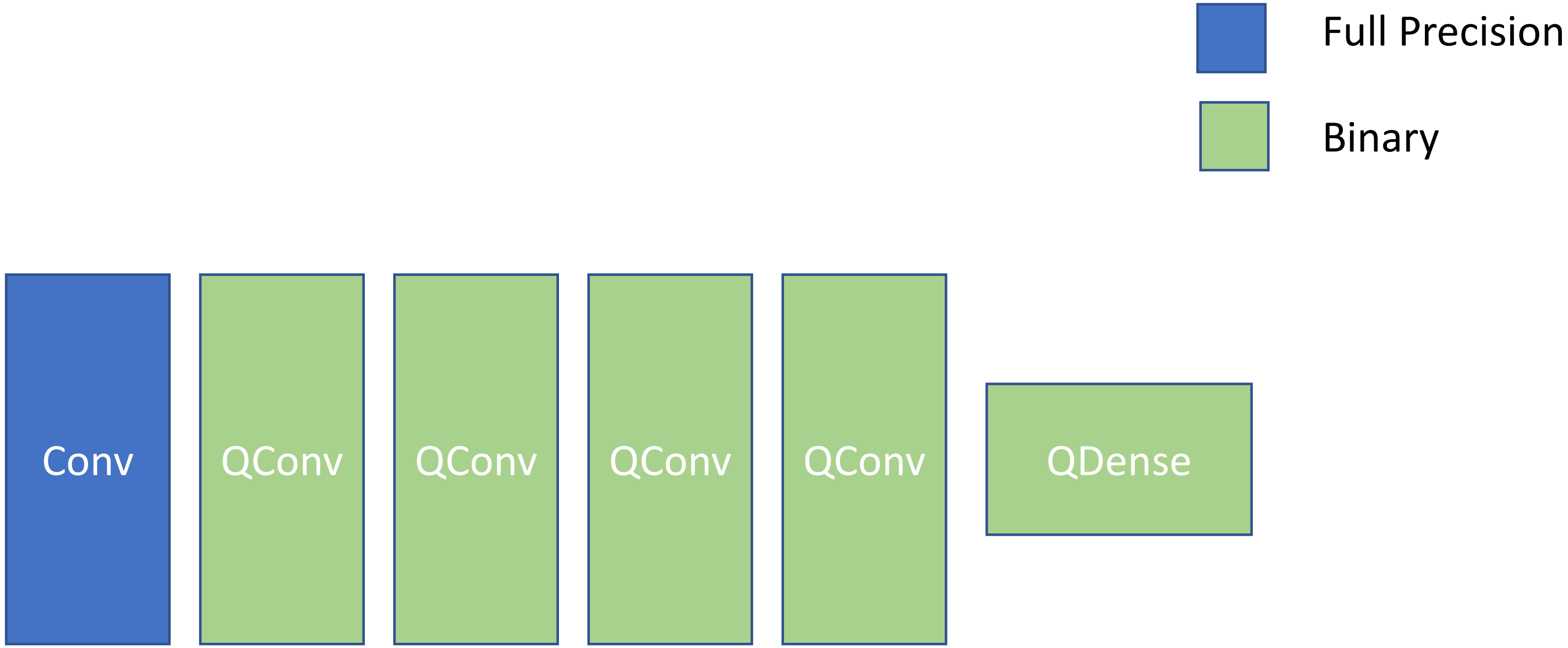
Unique bit combinations lost during popcount

Cai et al., “Deep Learning with Low Precision by Half-wave Gaussian Quantization.” 2017

# Implementing Binary Layers



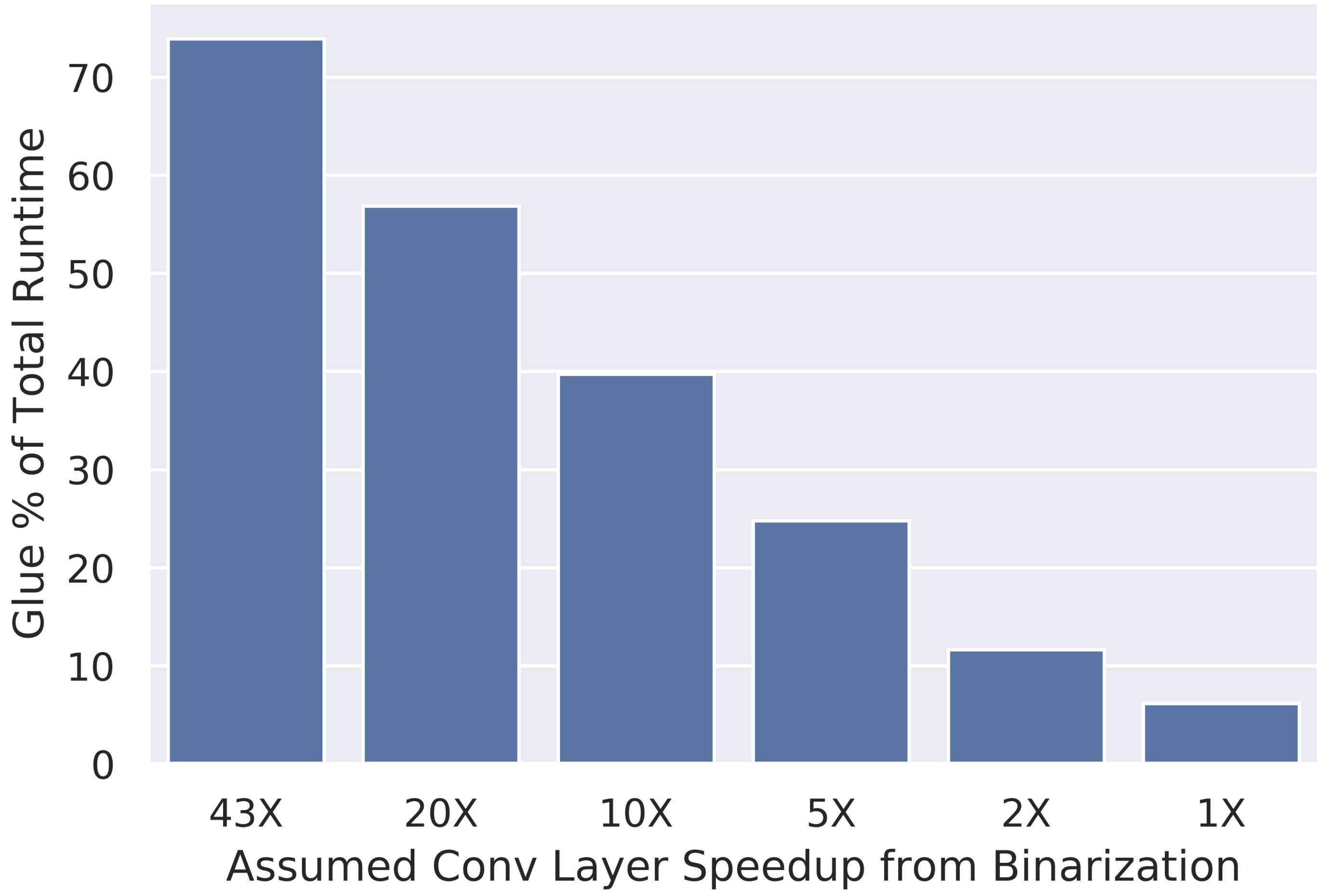
# Implementing Binary Models



# Implementing Binary Models

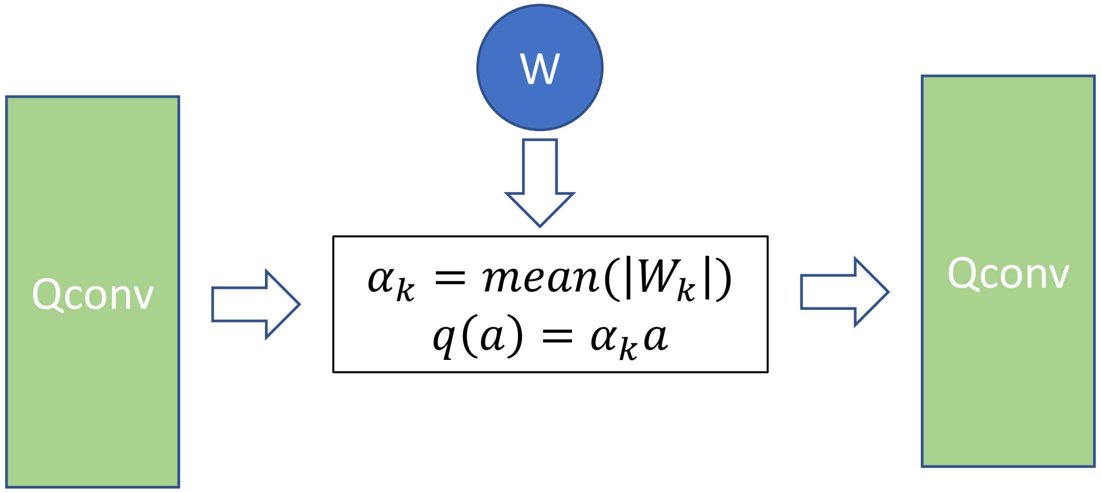

Computational Complexity: $4HWF$ $HWF$ $4HWF$ $HWF$ $5HWF$ $3HWF$ $\frac{NKKFHWC}{43}$

# Estimated Impact of Glue Layers




- Impact of glue layers is too high
- We must derive binarized glue for decent end-to-end speedups

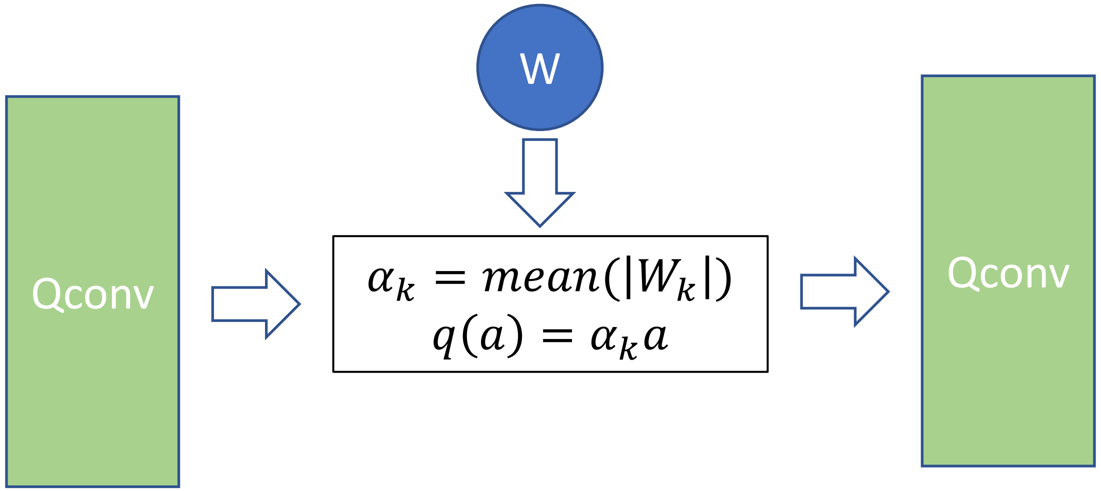
# Weight Scaling




- Introduced in XnorNet
- Allows scale of weights to be preserved
- Brought accuracy from 27% to 44%
- Now used ubiquitously

Rastegari et al., “Xnor-net: Imagenet classification using binary convolutional neural networks.” 2016

# Quantized Weight Scaling




$$AP2(x) = 2^{\text{round}(log_2(|x|))}$$

$$wb = -log_2(AP2(\alpha_k))$$

$$q(a) = (a + (1 \ll (wb - 1))) \gg wb$$

- Use approximate power of 2 (AP2)
- Replaces multiply with bitwise shift
- Constant at inference time
- Requires only a single instruction

Rastegari et al., "Xnor-net: Imagenet classification using binary convolutional neural networks." 2016

# BatchNormalization

- Centers and scales output activations
- Essential for quantization, used in all binary techniques
- Must derive quantized versions of both centering and scaling

$$\mu_k = \frac{1}{m}\sum_{i=1}^{m}(a_i) \qquad \sigma_k^2 = \frac{1}{m}\sum_{i=1}^{m}(a_i - \mu_k)^2 \qquad \hat{a}_i = \frac{a_i - \mu_k}{\sqrt{\sigma_k^2 + \epsilon}}$$

Ioffe et al., "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift." 2015

# Binary Scaling

- We can simply compute the AP2 of standard deviation

$$AP2(x) = 2^{\text{round}(log_2(|x|))}$$
$$wb = -log_2(AP2(\alpha_k))$$
$$q(a) = (a + (1 \ll (wb - 1))) \gg wb$$

$$\Rightarrow$$

$$AP2(x) = 2^{\text{round}(log_2(|x|))}$$
$$wb = -log_2(AP2(\alpha_k))$$
$$sb = log_2(AP2(\sqrt{\sigma_k^2 + \epsilon}))$$
$$q(a) = (a + (1 \ll (wb - 1))) \gg (wb + sb)$$

# Binary Center

To add a constant to a binarized tensor, we must use Fixed Point Quantization (FPQ) with the same bits and scale

**Algorithm 9:** Fixed point quantization (FPQ) function.

**Input:** a tensor $X$ to quantize to $B$ bits with scale $S$

**Output:** $B$-bit quantized tensor $Y$

$\hat{X} = \mathrm{clip}(X, -S, S)$

$g = \frac{S}{2^B - 1}$ `// Compute granularity`

$Y = \mathrm{round}(\frac{\hat{X}}{g})$

*N*-bit input to next layer

*wb* fractional bits

$\Rightarrow B = N + wb$

$$\overbrace{\underbrace{1\;1\;1\;1}_{1}}^{N\text{-bit input to next layer}}\;\overbrace{\underbrace{0\;1\;1\;0\ldots}_{\frac{1}{2}+\frac{1}{4}+\frac{1}{8}+\cdots}}^{wb\text{ fractional bits}}$$

$$\Rightarrow S = 1 + \sum_{i=1}^{wb} \frac{1}{2^N - 1} ({}^{1}/_{2})^i$$

$$= 1 + \frac{1}{2^N - 1}(1 - \frac{1}{2^{wb}})$$

$$\hat{\mu} = FPQ(\mu, B, S)$$

# Fused Glue Operation

$$AP2(x) = 2^{\mathrm{round}(log_2(|x|))}$$

$$wb = -log_2(AP2(\alpha_k))$$

$$sb = log_2(AP2(\sqrt{\sigma_k^2 + \epsilon}))$$

$$q(a) = (a + (1 \ll (wb - 1))) \gg (wb + sb)$$

$$\Rightarrow$$

$$AP2(x) = 2^{\mathrm{round}(log_2(|x|))}$$

$$wb = -log_2(AP2(\alpha_k))$$

$$sb = log_2(AP2(\sqrt{\sigma_k^2 + \epsilon}))$$

$$B = N + wb \qquad S = 1 + \frac{1}{2^N - 1}(1 - \frac{1}{2^{wb}})$$

$$\hat{\mu} = FPQ(\mu, B, S)$$

$$cb = (1 \ll (wb - 1)) - \hat{\mu}$$

$$\boxed{q(a) = (a + cb) \gg (wb + sb)}$$

This is the fused glue operation

All terms are constant at runtime except $a$

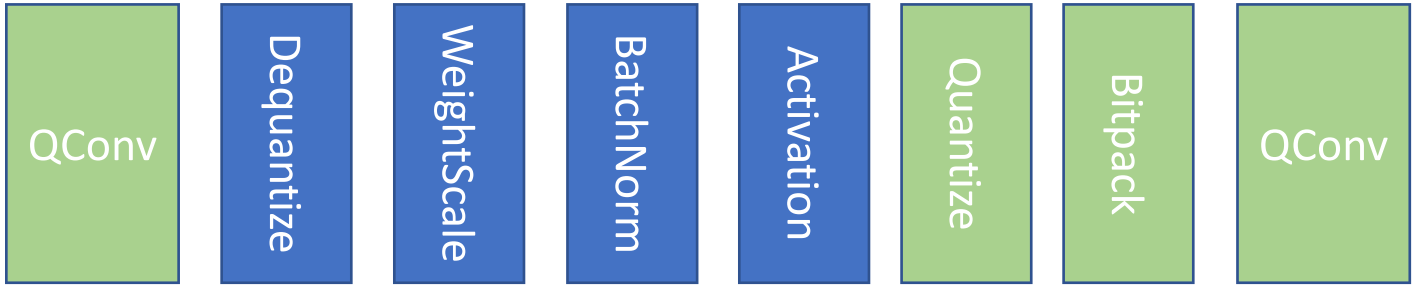
Only requires two integer operations

# Fully Binarized Network

Full Precision

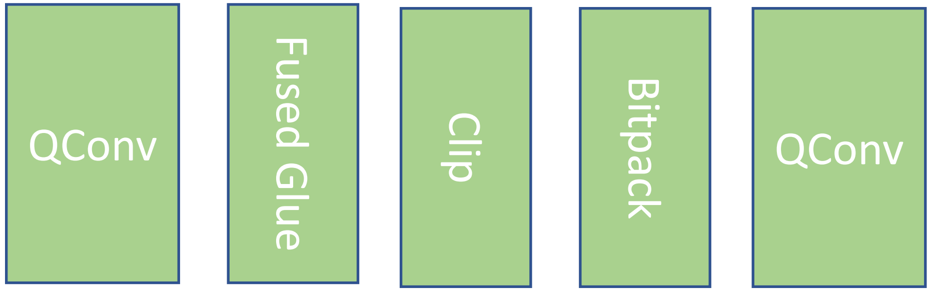
Binary

Traditional Binary Network




Computational Complexity: $4HWF$ $HWF$ $4HWF$ $HWF$ $5HWF$ $3HWF$ Total = $18HWF$

Fully Binarized Network




Computational Complexity: $2HWF$ $HWF$ $3HWF$ Total = $6HWF$

- 3X fewer glue operations
- No floating-point data
- No multiplication or division

# FBN Accuracy

| | Model | Name | 1-bit | 2-bit | 3-bit | full precision |
|---|---|---|---|---|---|---|
| | | ImageNet top-1 accuracy | | | | |
| 1 | AlexNet | Xnor-Net [48] | 44.2% | — | — | 56.6% |
| 2 | AlexNet | BNN [12] | 27.9% | — | — | — |
| 3 | AlexNet | DoReFaNet [63] | 43.6% | 49.8% | 48.4% | 55.9% |
| 4 | AlexNet | QNN [27] | 43.3% | 51.0% | — | 56.6% |
| 5 | AlexNet | HWGQ [4] | — | 52.7% | — | 58.5% |
| 6 | VGGNet | HWGQ [4] | — | 64.1% | — | 69.8% |
| 7 | AlexNet | Riptide-unipolar (ours) | 44.5% | 52.5% | 53.6% | 56.5% |
| 8 | AlexNet | Riptide-bipolar (ours) | 42.8% | 50.4% | 52.4% | 56.5% |
| 9 | VGGNet | Riptide-unipolar (ours) | 5[illegible] | 64.2% | 67.1% | 72.7% |
| 10 | VGGNet | Riptide-bipolar (ours) | 54.4% | 61.5% | 65.2% | 72.7% |
| 11 | ResNet18 | Riptide-unipolar (ours) | 47.9% | 58.4% | 61.8% | 70.9% |

- Our system is comparable to state-of-the-art techniques
- Unipolar quantization yields higher accuracies as expected
- Effective across various models

# Measurement Platform

Raspberry Pi
ARM Cortex-A53

- Widely available and inexpensive
- Representative of IoT devices
  - QualComm Snapdragons
  - Azure Sphere
- Resource constrained / in need of acceleration

Optimizing deep learning compiler

...

Tensor Expression Language

Schedule Optimization Space

AutoTVM: Optimize Tensor Operators

...

Separates compute and implementation into a declaration and schedule
Schedules contain knobs that are attuned for the backend

Chen et al., “TVM: An Automated End-to-End Optimizing Compiler for Deep Learning.” 2018

# TVM Schedule Intrinsics

- **Tiling:** Break computation into chunks for better locality
- **Vectorization:** Use hardware SIMD instructions for more efficient operation execution.
- **Parallelization:** Leverage MIMD facilities such as multiple cores.
- **Loop Unrolling:** Replicate the body of loops to reduce overhead.

Chen et al., "Learning to Optimize Tensor Programs." 2019

# Fast Popcount

LLVM 8.0

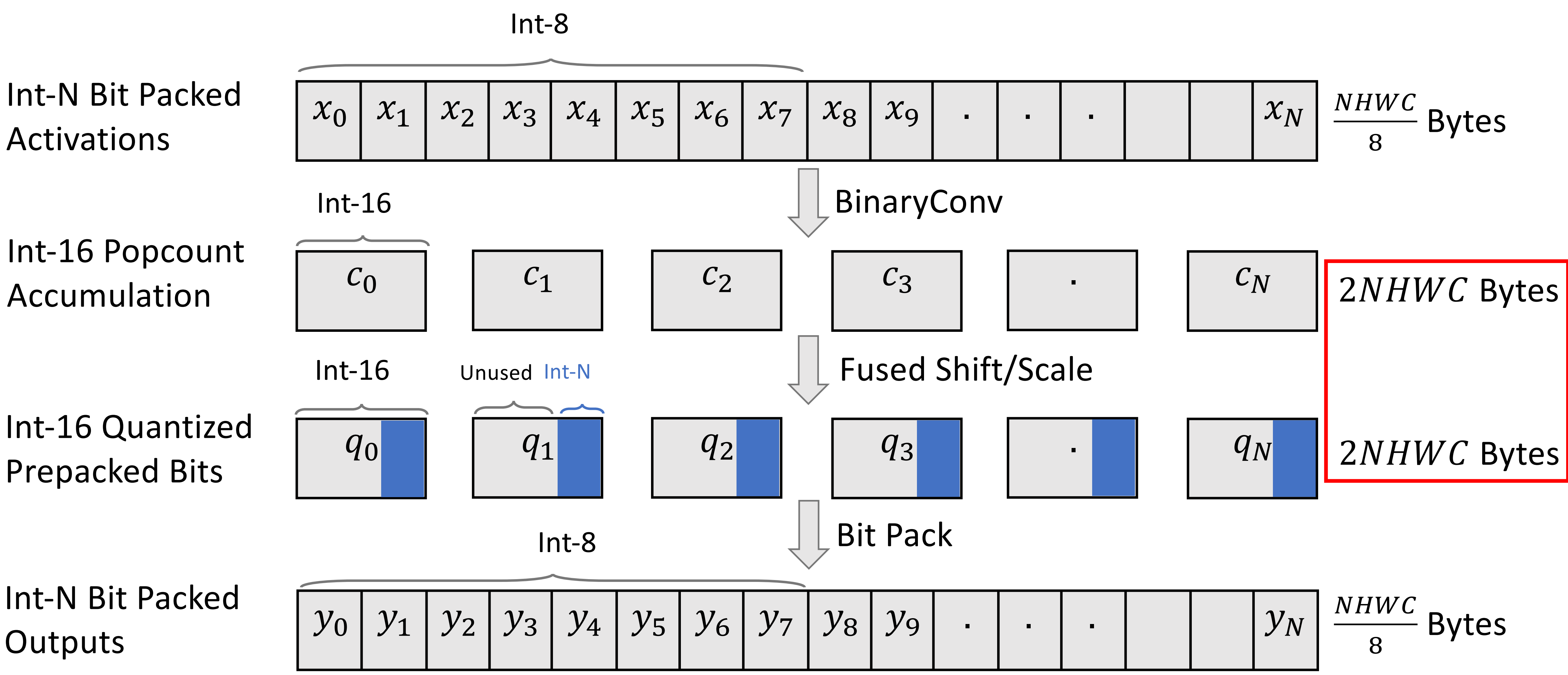
```
x8 | vmovl.8     q0, d0
   | vmovl.8     q2, d1
   | vand        q0, q0, q2
   | vcnt.8      q0, q0
   | vpaddl.8    q0, q0
   | vadd.16     q1, q0, q0

     vst1.16      q1, addr
```

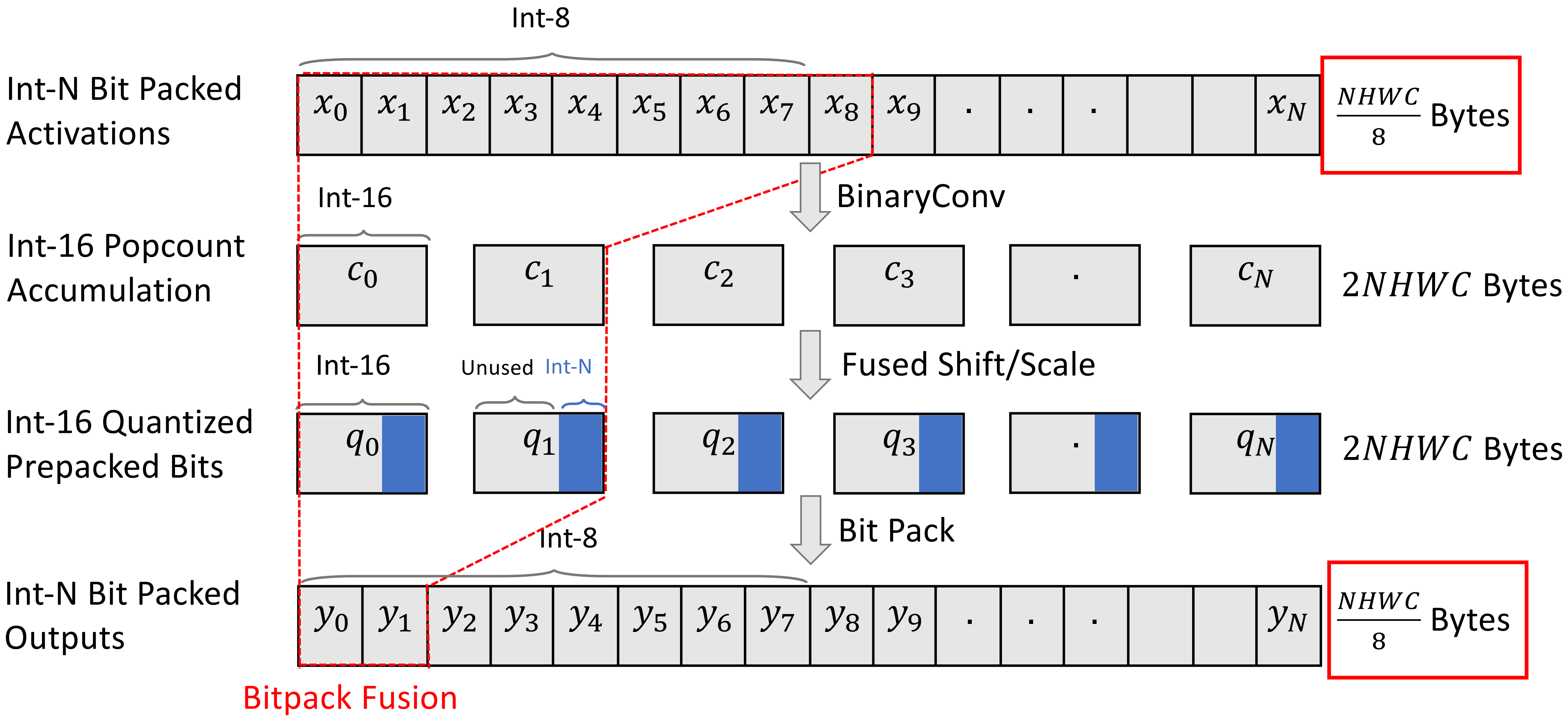
Total: 49

Synthesized

```
x8 | vand.8      d0, d0, d1
   | vcnt.8      d0, d0

     vadd.8      d0, d0, d1
     vadd.8      d0, d0, d1
     vadd.8      d0, d0, d1
     vadd.8      d0, d0, d1

     vpadd.8     d0, d0, d1
     vpadd.8     d0, d0, d1

     vpadal.8    q1, {d0,d1}

     vst1.16      q1, addr
```

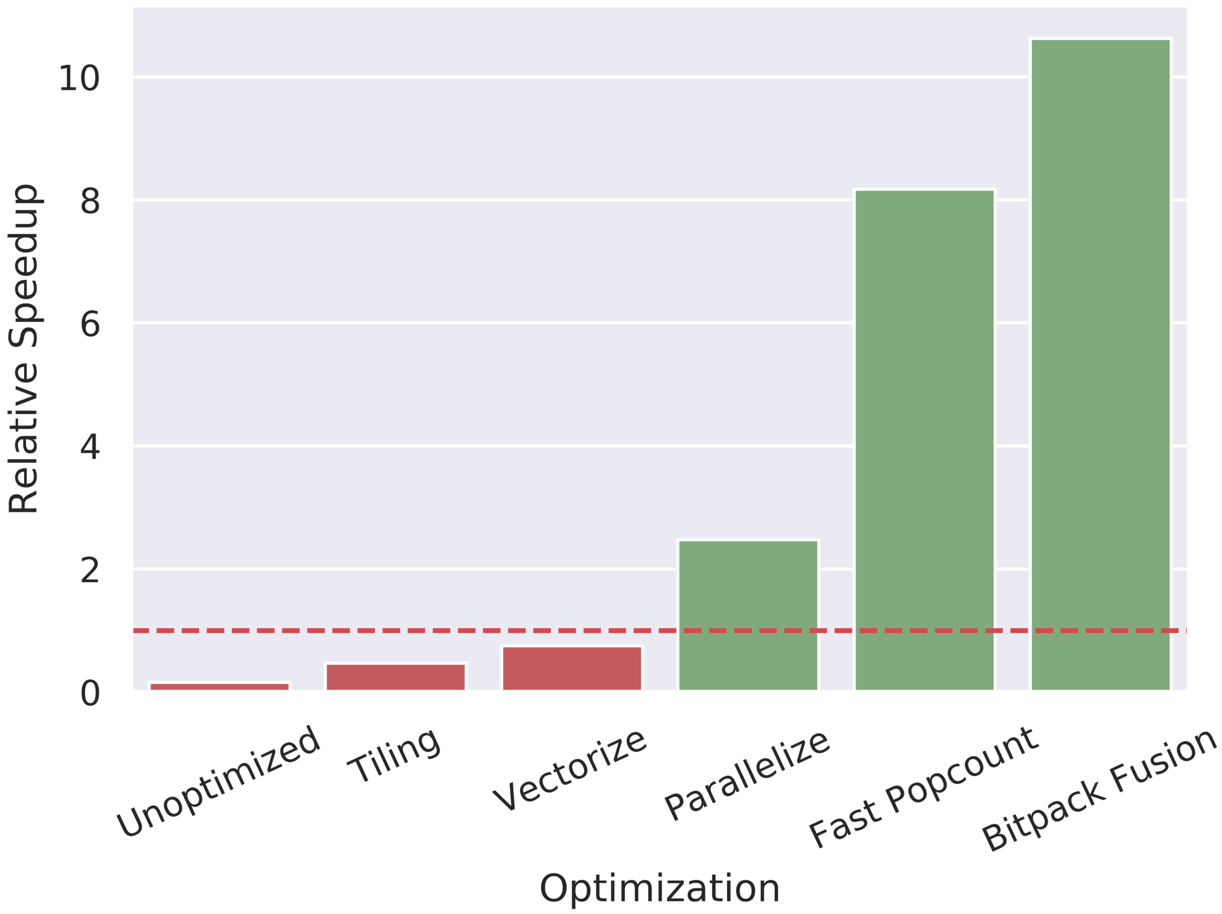
Total: 24


Int-8
Int-N Bit Packed Activations
$x_0$ $x_1$ $x_2$ $x_3$ $x_4$ $x_5$ $x_6$ $x_7$ $x_8$ $x_9$ . . . $x_N$
$\frac{NHWC}{8}$ Bytes
BinaryConv
Int-16
Int-16 Popcount Accumulation
$c_0$ $c_1$ $c_2$ $c_3$ . $c_N$
$2NHWC$ Bytes
Fused Shift/Scale
Int-16
Unused
Int-N
Int-16 Quantized Prepacked Bits
$q_0$ $q_1$ $q_2$ $q_3$ . $q_N$
$2NHWC$ Bytes
Bit Pack
Int-8
Int-N Bit Packed Outputs
$y_0$ $y_1$ $y_2$ $y_3$ $y_4$ $y_5$ $y_6$ $y_7$ $y_8$ $y_9$ . . . $y_N$
$\frac{NHWC}{8}$ Bytes



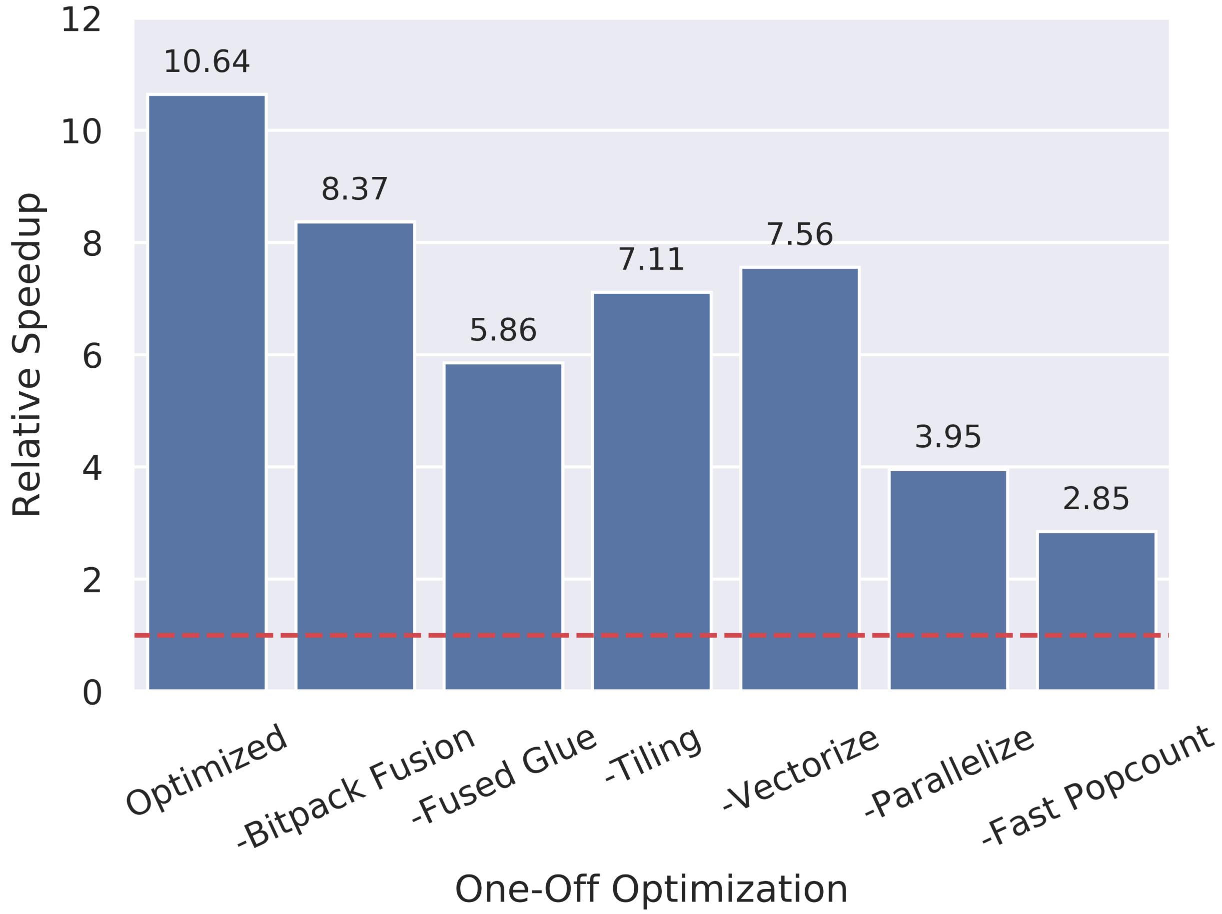
Int-8
Int-N Bit Packed Activations
$x_0$ $x_1$ $x_2$ $x_3$ $x_4$ $x_5$ $x_6$ $x_7$ $x_8$ $x_9$ . . . $x_N$
$\frac{NHWC}{8}$ Bytes
BinaryConv
Int-16
Int-16 Popcount Accumulation
$c_0$ $c_1$ $c_2$ $c_3$ . $c_N$
$2NHWC$ Bytes
Fused Shift/Scale
Int-16
Unused
Int-N
Int-16 Quantized Prepacked Bits
$q_0$ $q_1$ $q_2$ $q_3$ . $q_N$
$2NHWC$ Bytes
Bit Pack
Int-8
Int-N Bit Packed Outputs
$y_0$ $y_1$ $y_2$ $y_3$ $y_4$ $y_5$ $y_6$ $y_7$ $y_8$ $y_9$ . . . $y_N$
$\frac{NHWC}{8}$ Bytes
Bitpack Fusion

# Impact of Optimizations



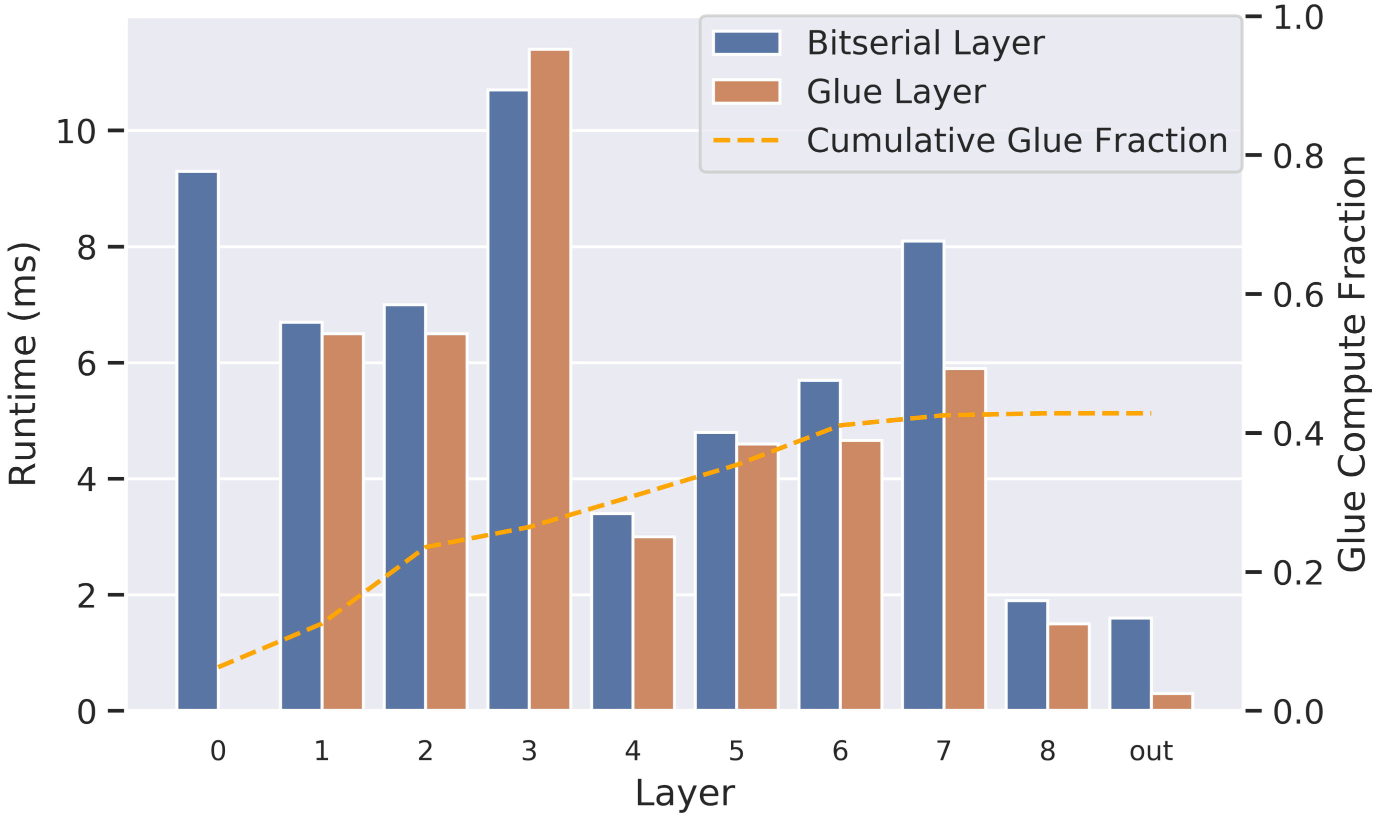

- Combination of TVM optimizations gives 12X Speedup over baseline
- Each optimization has a significant impact
- Speedups from bitpack fusion are due to fewer memory operations
- With a high-quality implementation, we can study our design choices

# Optimization Ablation Study



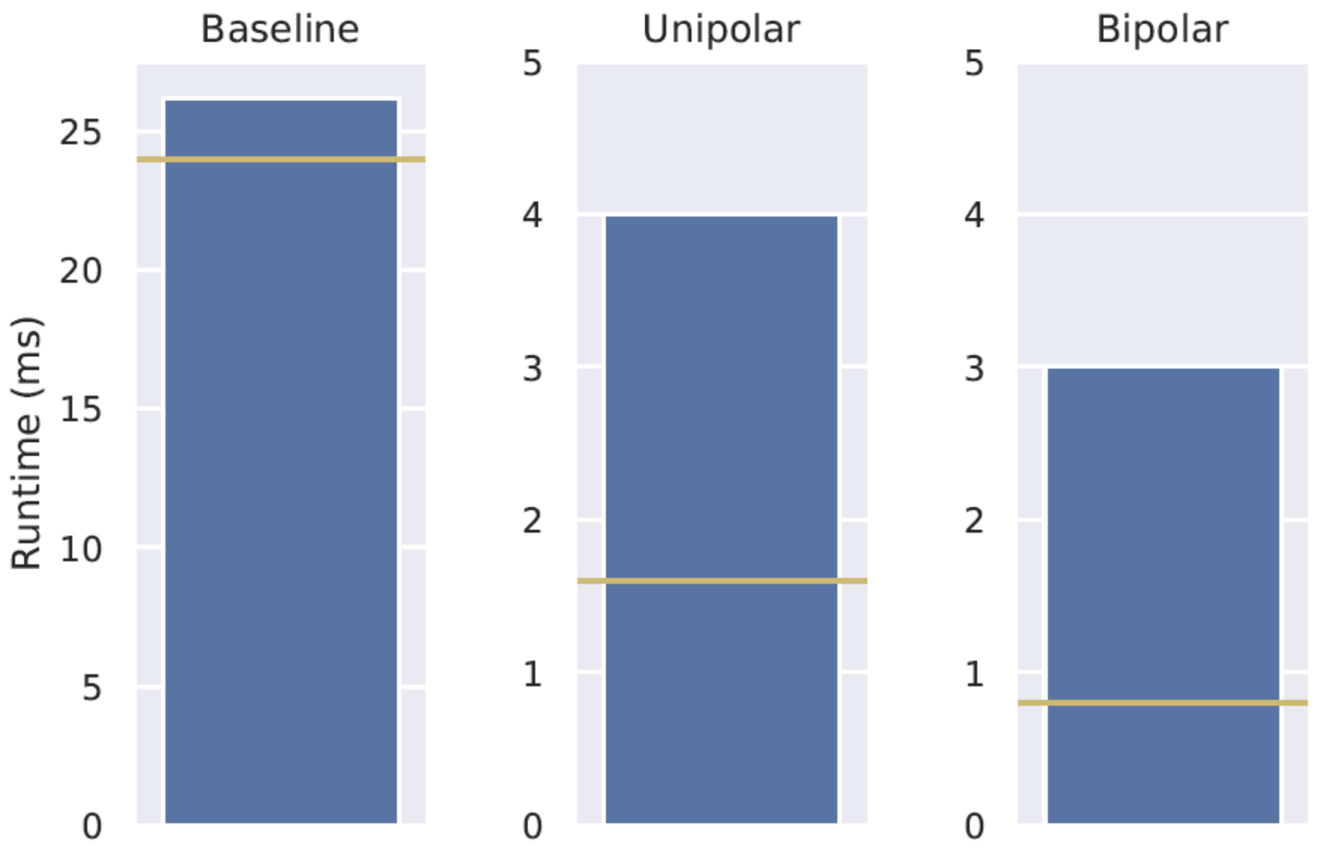

- Removing any optimization has a significant impact on performance
- Using fused glue gives nearly a 2X speedup, as predicted by opcount estimates

# Glue Layer Impact



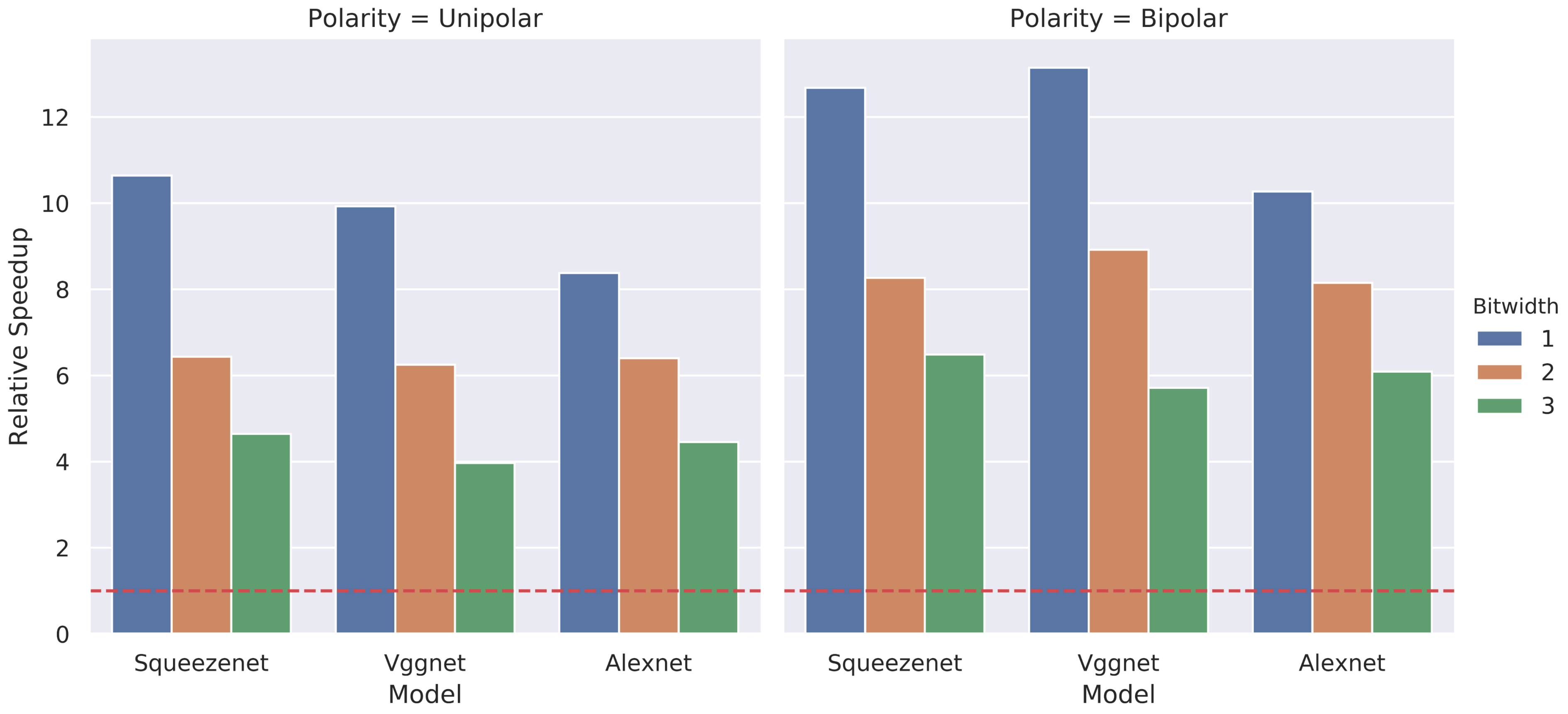

- Glue consistently takes a similar amount of time as core compute layers
- Our fused glue operation almost completely removes this cost

# Impact of Polarity

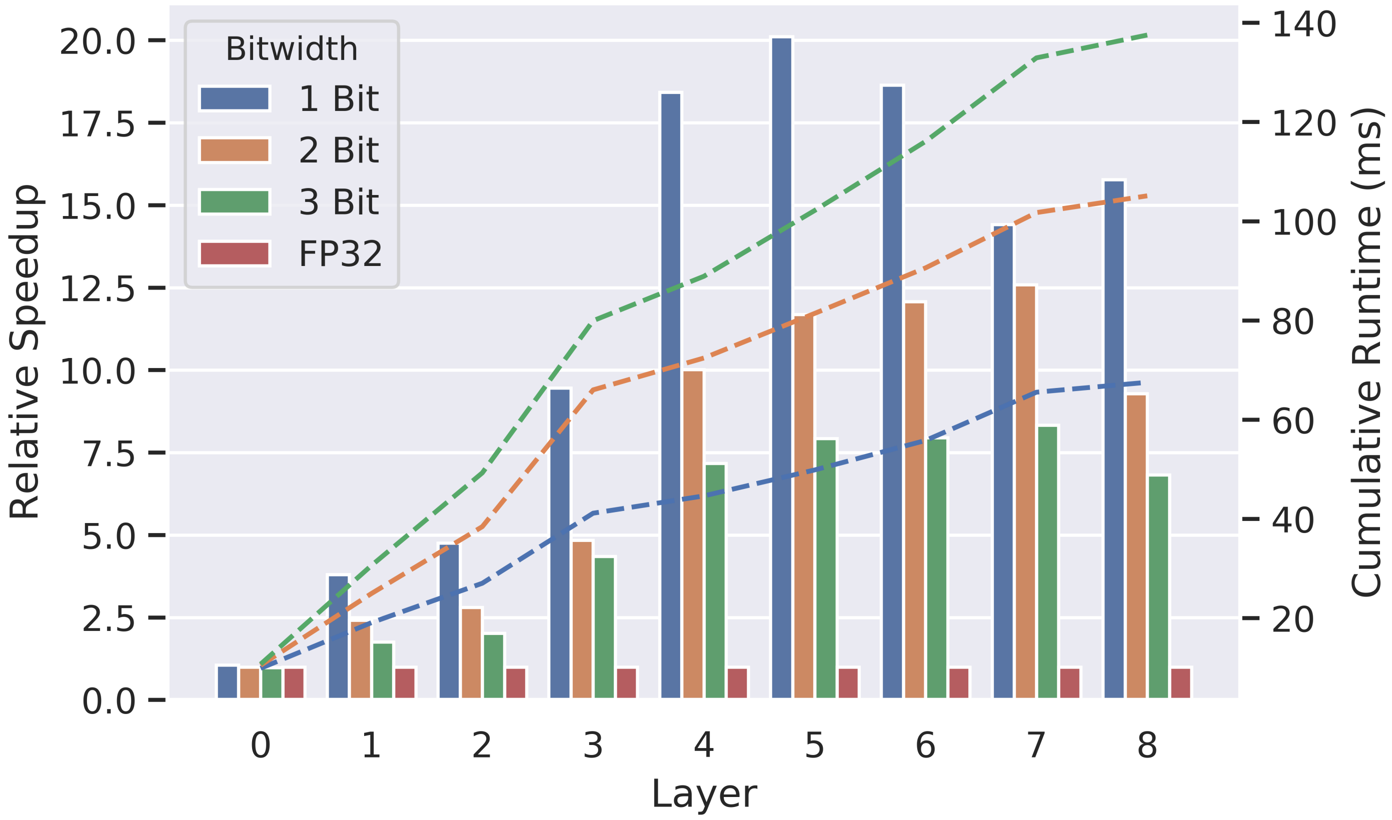



- Baseline is near optimal
- Quantized layers have much more memory overhead
- Although unipolar quantization has twice as many operations, it is only marginally slower than bipolar quantization

# Cumulative Speedup



# Layerwise Speedup




- Speedup is not consistent across layers
- May be possible to design a network of binarizable layers

# Thank You!

Code:

Paper:

# Backup Slides

| | Model | Name | 1-bit | 2-bit | 3-bit | full precision |
|---|---|---|---|---|---|---|
| | | ImageNet top-1 accuracy / Runtime (ms) | | | | |
| 1 | AlexNet | Xnor-Net [48] | 44.2% / — | — / — | — / — | 56.6% / — |
| 2 | AlexNet | BNN [12] | 27.9% / — | — / — | — / — | — / — |
| 3 | AlexNet | DoReFaNet [63] | 43.6% / — | 49.8% / — | 48.4% / — | 55.9% / — |
| 4 | AlexNet | QNN [27] | 43.3% / — | 51.0% / — | — / — | 56.6% / — |
| 5 | AlexNet | HWGQ [4] | — / — | 52.7% / — | — / — | 58.5% / — |
| 6 | VGGNet | HWGQ [4] | — / — | 64.1% / — | — / — | 69.8% / — |
| 7 | AlexNet | Riptide-unipolar (ours) | 44.5% / 150.4 | 52.5% / 196.8 | 53.6% / 282.8 | 56.5% / 1260.0 |
| 8 | AlexNet | Riptide-bipolar (ours) | 42.8% / 122.7 | 50.4% / 154.6 | 52.4% / 207.0 | 56.5% / 1260.0 |
| 9 | VGGNet | Riptide-unipolar (ours) | 56.8% / 243.8 | 64.2% / 387.2 | 67.1% / 610.0 | 72.7% / 2420.0 |
| 10 | VGGNet | Riptide-bipolar (ours) | 54.4% / 184.1 | 61.5% / 271.4 | 65.2% / 423.5 | 72.7% / 2420.0 |
| 11 | ResNet18 | Riptide-unipolar (ours) | 47.9% / 76.2 | 58.4% / 112.0 | 61.8% / 152.3 | 70.9% / 380.8 |