RIPTIDE: FAST END-TO-END BINARIZED NEURAL NETWORKS

Josh Fromm [1 2] **Meghan Cowan** [1] **Matthai Philipose** [3] **Luis Ceze** [1 2] **Shwetak Patel** [1]

ABSTRACT

Binarized neural networks have attracted much recent attention due to their promise of making convolutional neural networks fast and compact. However, these benefits have proven hard to realize in practice. In this paper, we identify the underlying barriers to high performance and propose solutions ranging from missing implementations for certain operations to carefully scheduled library support for binarized linear algebra operations. The combination of these innovations allows us to report the first measured end-to-end speedups for binarized networks. For instance, we show a 6.3× speedup over a standard VGGNet variant at state-of-the-art (64.2% for top-1 binarized classification of ImageNet) accuracy. More broadly, speedups range from 4-12× and the techniques we propose are crucial to achieving them.

1 INTRODUCTION

Binarized neural networks (BNNs) represent their parameters and activations using very low bitwidths (e.g., 1 to 3 bits). During inference, the resulting networks use much less memory than conventional (e.g., 32-bit) representations and execute dramatically fewer operations by converting operations on large floating point vectors into "bitserial" versions that apply bitwise operations on packed bit vectors. For instance (Rastegari et al., 2016) report convolution layers that use 58× fewer operations and 32 × less memory than the standard floating point versions. This increased efficiency typically comes at the cost of reduced inference accuracy, and a slew of recent work (Courbariaux et al., 2016; Zhou et al., 2016; Cai et al., 2017; Hubara et al., 2016; Tang et al., 2017; Dong et al., 2017; Fromm et al., 2018; Choi et al., 2019) has therefore focused on closing the accuracy gap. Despite considerable progress in developing more accurate models with low theoretical instruction counts, we are aware of no work that has realized measured performance gains on real-world processors. In this paper, we present Riptide, an end-to-end system for producing binarized versions of convolutional neural networks that yield significant speedups.

Measurable performance gains on binarized models are hard to achieve for three main reasons. First, for many recent higher-accuracy training techniques, no efficient bitserial implementation has been proposed. In some cases (e.g., where bits are scaled using non-linear scaling factors (Cai et al., 2017)), it is unclear that such implementations even exist. Second, current work has focused on schemes for binarizing the "core" convolutional and fully-connected layers. However, once the dramatic gains of binarization on these layers has been realized, the "glue" layers (batch normalization, scaling and (re-) quantization) become bottlenecks. Bitserial implementations of these layers have traditionally been ignored. Finally, existing floating-point implementations have been carefully scheduled for various processor architectures over many decades via libraries such as BLAS, MKL and CuDNN. No corresponding libraries exist for low-bitwidth implementations. The "number-of-operations" speedup above does not therefore translate to wallclock-time speedups.

The Riptide system presented in this paper addresses these issues. We carefully analyze the barriers to realizing bitserial implementations of recently proposed accuracy-enhancing techniques and suggest efficient options (Section 4.1). We show how to produce fully bitserial versions of glue layers, almost entirely eliminating their runtime overhead (Section 4.2). Finally, we show how to schedule binarized linear algebra routines by combining selected standard scheduling techniques (e.g., loop tiling, loop fusion and vectorization) with memory access optimization specific to packed representations (Section 4.3). Taken together, we show in our evaluation (Section 5.1) that these techniques yield binarized versions of standard networks (SqueezeNet, AlexNet, VGGNet, and Resnet) that show measured end-to-end speedups in the 4-12× range relative to optimized floating point implementations, while maintaining state-of-the-art accuracy. To our knowledge, these are the first reported numbers of measured speedup due to binarization.

[1]Department of Computer Science and Engineering, University of Washington, Seattle, USA [2]OctoML, Seattle, Washington, USA [3]Microsoft Research, Redmond, Washington, USA. Correspondence to: Josh Fromm <jwfromm@octoml.ai>, Matthai Philipose <matthaip@microsoft.com>.

Proceedings of the 2^{nd} SysML Conference, Palo Alto, CA, USA, 2019.

Algorithm 1 Typical CNN forward propagation. Lines marked $\star$ are glue layers that use floating point arithmetic.

1: **Input:** Image X, network with L convolutional layers.
2: $c_0 = \text{Conv}(X)$ {Input layer block.}
3: $a_0 = \text{BatchNorm}(c_0)$
4: **for** $k = 1$ **to** L **do**
5: $HWC = \text{shape}(a_{k-1})$ and $KKFC = \text{shape}(L_k)$
6: $c_k = \text{Conv}(a_{k-1})$ {$KKFHWC$ ops.}
7: $p_k = \text{Pooling}(c_k)$ {HWF ops.}
8: $b_k = \star\text{BatchNorm}(p_k)$ {$4HWF$ ops.}
9: $a_k = \text{Activate}(b_k)$ {HWF ops.}
10: **end for**
11: $Y = \text{Dense}(a_L)$

12: **Def. of Conv**, input A with F filters W, size KKC.
13: **for** $i = 0$ **to** K, $j = 0$ **to** K **do**
14: **for** $c = 0$ **to** C **do**
15: c_{hwf} += $A[h+i][w+j][c] * W_f[i][j][c]$
16: **end for**
17: **end for**

Algorithm 2 Convolution block that replaces line 6 in Algorithm 1 for an N-bit binary network.

1: **Input:** Activation tensor x.
2: $q = \star\text{Quantize}(x)$ {At least 5HWC ops.}
3: $b = \text{BitPack}(q)$ {At least 3HWC ops.}
4: $c = \text{BitserialConv}(b)$ {$\frac{NKKFHWC}{43}$ ops.}
5: $f = \star\text{Dequantize}(c)$ {4HWF ops.}
6: $y = \star\text{WeightScale}(f)$ {HWF ops.}

7: **Definition of BitserialConv** with F size K kernels on input Q, a set of N int64 bit packed tensors with original size HWC.
8: **for** $n = 0$ **to** N, $i = 0$ **to** K, $j = 0$ **to** K **do**
9: **for** $c = 0$ **to** $\frac{C}{64}$ **do**
10: $x_{hwf} = Q_n[h+i][w+j][c] \otimes W_f[i][j][c]$
11: y_{hwf} += $2^n \text{popc}(x_{hwf})$
12: **end for**
13: **end for**
14: $y_{hwf} = 2y_{hwf} - KKC$

2 Background and Related Work

2.1 Conventional Convolutional Networks

Algorithm 1 describes how a typical convolutional network processes an input image X. An initial convolution (line 2) extracts features from the color channels of X, then a stack of L convolutions (lines 4-10) is applied to progressively parse the activations of the previous layer into an embedding. An output dense layer (line 11) processes the embeddings to generate probabilities of what class the input image represents. Surrounding each convolution are intermediate "glue" layers (lines 7, 8, and 9) that apply some preprocessing to prepare the activation tensor for the next convolution. The number of operations in the network is completely dominated by the convolution layers, which are approximately KKC times (which is often two orders of magnitude) more operations than any glue layer. Noting the high concentration of operations in convolutions, many researchers have explored methods that optimize convolution to yield networks with superior inference times.

2.2 Binary Convolutional Networks

First introduced by Courbariaux et al. (2016), network binarization attempts to optimize networks by replacing full precision convolutions with the more efficient "bitserial" convolutions described in Algorithm 2. In a binary network, both weights and activations of the network are quantized to a single bit representing +1 or -1 ($q = \text{sign}(x)$). Not only does this give an immediate benefit of reducing the memory footprint of the model by 32×, it also allows a xnor-popcount operation (lines 10-11) to replace floating point multiply-accumulate. By packing bits into a larger datatype such as Int64 using Equation 1 (line 3), the amount of operations (and the theoretical runtime) in the inner loop of a bitserial convolution reduces from $2C$ to $\frac{3C}{64}$, a reduction of 43×.

$$b_{i,n} = \sum_{j=0,n=0}^{63,N-1} (Q_{64i+j} \wedge n) \ll (j - n) \tag{1}$$

The use of bitserial convolution requires additional glue layers (lines 2, 5, and 6). Because BatchNorm (Ioffe & Szegedy, 2015), which normalizes and centers activations, is used ubiquitously in binary networks, the integer output of a bitserial convolution must be converted to floating point (line 5) then converted back to integer for the next layer and rounded into 2^N bins, where N is the number of quantization bits used (line 2). Although the total number of operations spent in glue layers (8HWC + 9HWF) is sizeable relative to the bitserial convolution ($\frac{KKFHWC}{43}$) for typical model dimensions, their impact on runtime has not been well examined.

2.3 Binary Accuracy Improvement Techniques

Although 1-bit binary models promise significant performance benefits, the accuracy they have been shown capable of achieving on challenging datasets like ImageNet has been underwhelming. For example, the AlexNet (Krizhevsky et al., 2012) based BNN used by Courbariaux et al. (2016) was only able to reach a top-1 accuracy of 27.9% when trained on ImageNet compared to the full precision model's

56%. The significant accuracy loss that comes with network binarization has been the focus of research in the space, with most papers introducing modifications to the core algorithm or new training techniques.

Rastegari et al. (2016) introduced XNOR-Net, which improved the accuracy of single bit binary models by adding the WeightScale function on line 6 of Algorithm 2. The term $\alpha_k = \text{mean}(|W_k|)$ was multiplied into the binary convolution output, where W_k are the weights of one of the convolutional layer's filters. Weight scaling proved extremely useful for preserving both the magnitude and relative scale of weights. The authors additionally noted that applying batch normalization directly before quantization ensures maximum retention of information due to the centering around zero. These subtle but important changes allowed an XNOR-Net version of AlexNet to reach 44.2% accuracy on ImageNet.

Although XNOR-Net offered a substantial improvement to accuracy, follow-up works noted that even so, the accuracy achievable with 1-bit activations is simply not compelling and instead focus on using $N \geq 2$ bits. Hubara et al. (2016) and Zhou et al. (2016) introduce QNN and DoReFA-Net respectively, both of which use 2-bit activations to achieve higher accuracy on ImageNet. Both works used very similar techniques and had similar results, here we'll discuss DoReFa-Net's multi-bit implementation as it is more precisely defined. Like XNOR-Net, DoReFa-Net quantizes weights using $q = \text{sign}(x)$ and uses weight scale term α. Activations, on the other hand, are quantized into linearly spaced bins between zero and one (Equation 2). DoReFa-Net uses $\text{clip}(x, 0, 1)$ as activation function (line 9 in Algorithm 1), ensuring proper outputs from Equation 2. DoReFa-Net was able to reach an AlexNet top-1 accuracy of 50%, closing quite a bit of the gap between binary and floating point models.

$$\begin{aligned} q_{\text{bits}} &= \text{round}((2^N - 1) * x) \\ q_{\text{approx}} &= \frac{1}{2^N - 1} q_{\text{bits}} \end{aligned} \tag{2}$$

Cai et al. (2017) introduced Half Wave Gaussian Quantization (HWGQ), a new Quantize function that enables 2-bit activation binary networks to achieve the highest reported AlexNet accuracy (52.7%) to our knowledge. HWGQ uses the same weight quantization function as XNOR-Nets and DoReFa-Nets, quantizing to a single bit representing -1 or 1 and adding scale factor α. To quantize the networks activations, the authors note that the output of ReLU tends to fit a half Gaussian distribution. The authors suggest that the Quantize function should attempt to fit this distribution. To this end, HWGQ uses k-means clustering to find $k = 2^N$ quantization bins that best fit a half Gaussian distribution.

Although the original interest in binarization was due to its potential to enable high speed and low memory models without sacrificing too much accuracy, all follow up work has been focused on reducing the accuracy gap rather than the speedup itself. To our knowledge, no paper has reported an actual end-to-end speedup or described in detail the techniques required to yield one. In the following sections we examine the barriers that make such a measurement difficult and present Riptide, the first system to enable performant end-to-end binary models.

3 Challenges in Binarization

In this section we explore what it would take to create a highly performant end-to-end bitserial implementation. In doing so, we uncover multiple barriers that must be overcome. These challenges can be broken into three categories: choosing the proper binarization method from the many options discussed in Section 2, inefficiencies in glue layers, and generating efficient machine code that can compete against hand optimized floating point libraries.

3.1 Implementing core operations efficiently

The first step in building a binary network is choosing a quantization method. Although it may seem adequate to pick the method with the highest accuracy, it is often challenging to implement the most accurate models in a bitserial fashion (i.e., using logical operations on packed bit-vectors as described above). In particular, proposed algorithms often achieve higher accuracy by varying *bit consistency* and *polarity* so as to trade off accuracy for bitserial implementability.

Lines 10 and 11 of Algorithm 2 describe the inner loop of bitserial convolution when values are linearly quantized, as in Equation 2. For $n > 2$, the term 2^n (which can be implemented as a left shift) adds the scale of the current bit to the output of popcount before it is accumulated, this is possible because the spacing between incremental bits is naturally linear. Using a non-linear scale would require replacing the efficient shift operation with a floating point multiply.

Additionally, it is imperative that the values of bits are consistent, for example for $N = 2$, the value of the sum of bit pairs 01 and 10 must equal the value of bit pair 11. If this is not the case, values are effectively being assigned to bits that are conditional on their bit pairs. However, the use of popcount anonymizes bits by accumulating them before they can be scaled. Using a representation that does not have bit consistency would require multiplying each x_{hwf} by a scaling constant and prevent the use of popcount, removing any reduction in computation benefits that quantization otherwise offers. High accuracy binarization techniques that attempt to better fit non-linear distributions by dropping bit consistency such as HWGQ are thus difficult to implement

efficiently.

Quantization polarity describes what the bits of a quantized tensor represent. In **unipolar quantization**, bits with value 0 represent 0 and bits with value 1 represent 1. Conversely, in **bipolar quantization** bits with value 0 represent -1 and bits with value 1 represent 1. Early binarization models such as XNOR-Nets and QNNs use bipolar quantization for both weights and activations due to the ability of the xnor operation to elegantly replace multiplication in the inner loop of a binary convolution. Because bipolar quantization must be centered around zero, it is not possible to actually represent zero itself without breaking linearity. Not only does zero have some intrinsic significance to activations, but it also is ubiquitously used to pad convolutional layers. In fact, *this padding issue prevents QNNs and XNOR-Nets from being implemented as proposed by their authors*.

Methods that use unipolar quantization for activations such as DoReFa-Net and PACT-SAWB (Choi et al., 2019) are able to represent zeros but encounter other implementation issues. Because weights are always bipolar due to their need to be capable of representing inverse correlation (negative numbers), the unset bits in a quantized weight tensor represent -1 while the unset bits in quantized activation tensor represent 0. This polarity mismatch prevents a single bitwise operation and popcount from producing a correct result since the bits effectively represent three values instead of two. The current literature does not provide an answer to this issue and it is not clear how to efficiently and correctly implement mixed polarity models.

It is worth noting that there also exist Ternary Weight Networks (Li et al., 2016) that use bipolar quantization and a mask tensor that specifies some bits as representing 0. Although ternary quantization is able to represent both zero and negative numbers, it is effectively using an extra bit to do so. Instead of being able to represent 2^N unique values, ternary quantization can only represent $2^{N-1} + 1$ values. This loss of expressiveness leads to ternary networks not having competitive accuracy with state-of-the-art unipolar models.

Attempting to navigate these numerous complications and implement an end-to-end system could easily lead to poor performance or incorrect output values at inference time. In section 4.1 we examine the impact of polarity on runtime and describe the quantization scheme used in Riptide.

3.2 Binarizing glue layers

In a typical floating point model, the vast bulk of compute goes into the core convolutional and dense layers. The interlayer glue operations such as non-linear activations, MaxPooling and BatchNormalization are so minimally compute intensive compared to core layers that their impact on runtime is ignored. However, in BNNs the number of operations in core layers is so greatly reduced that the time spent in glue layers actually becomes a major bottleneck.

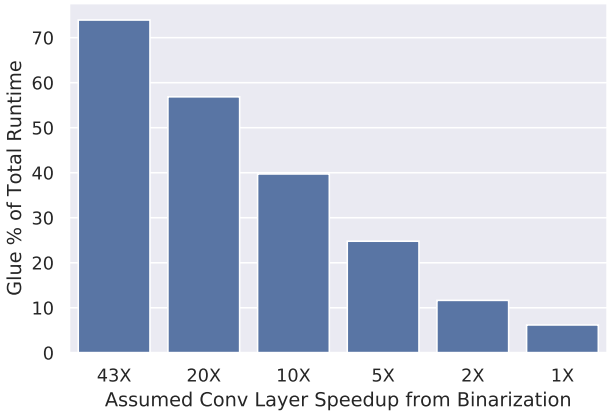

Figure 1. End-to-end speedup of SqueezeNet with fixed glue layer costs and theoretical speedups of convolution layers.

To demonstrate the effect of glue layers, we consider the total number of operations in a binarized SqueezeNet (Iandola et al., 2016). We count the number of operations in all bitserial convolution layers at various assumed speedups and compare those counts to the total number of glue operations. These estimates are visualized in Figure 1. We see glue layers make up a whopping 70% of the operations in a network given the optimal reduction in number of operations offered by binarization. Even assuming smaller speedups in practice, glue layers contribute a substantial fraction of the total estimated runtime at all scales where the speedups of binarization can justify its impact on accuracy.

Figure 1 makes it readily apparent that a high speed end-to-end implementation must minimize or all-together remove glue layers. However, all high accuracy binarization techniques today rely on BatchNormalization and weight scaling in the floating point domain. The centering and normalization effects of BatchNormalization are essential to generating consistent and accurate quantized activation representations and weight scaling has been shown to dramatically increase model accuracy by allowing the magnitude of weight tensors to be efficiently captured. Because these layers require floating point arithmetic, interlayer type casting and requantization must also be inserted. To address this bottleneck, we introduce a novel fusible operation that completely removes the cost of glue layers and yields a speedup of multiple factors without loss of accuracy in Section 4.2 .

3.3 Generating fast machine-specific code

One major benefit of binarization is the substantial compression of the amount of memory required to store weights.

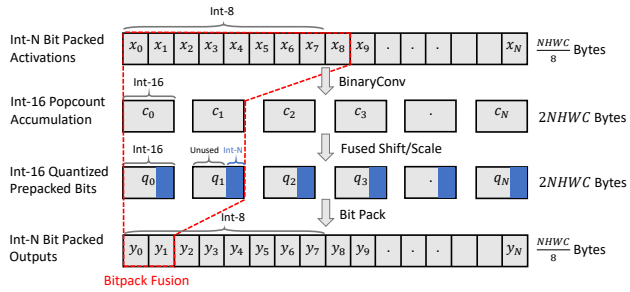

Figure 2. Scheduling of N-bit binary layer demonstrating intermediate memory used. By fusing computation within tiles, such as the region highlighted red, memory use can be reduced.

Ideally, this memory compression would also apply to the activation tensors of a model at runtime. However, as visualized in Figure 2 (ignore the dotted line for now), the intermediate memory is dominated by the output of popcount (2NHWC bytes) rather than the more efficient bitpacked tensor ($\frac{\text{NHWC}}{8}$ bytes). This is because each layer in Figure 2 is executed sequentially from top to bottom in a typical system. Not only does this increase the amount of memory shuffling required at inference time, but it also could prove to be a major challenge to running large models on resource constrained systems.

Even with all the barriers above resolved, a binary model is likely to be dramatically slower than its floating point variant. Floating point convolutions and dense layers use highly optimized kernels such as those found in OpenBLAS (Xianyi et al., 2014). By leveraging scheduling primitives such as tiling, vectorization, and parallelization, an optimized kernel can run orders of magnitude faster than a naive implementation. Unfortunately, no comparable hand optimized libraries exist for bitserial operations and developing one from scratch would be a challenging engineering effort that is outside the scope of most research projects. Recently, projects such as Halide (Ragan-Kelley et al., 2013) and TVM (Chen et al., 2018a) have arisen that attempt to simplify the process of creating optimized schedules by separating the definition of compute from the schedule itself, and in some cases supporting automated hyperparameter search to produce good schedules (Chen et al., 2018b). In Section 4.3, we describe how we extend TVM to support bitserial operations and produce machine code that allows significant speedups even when compared against highly optimized floating point libraries.

4 System Design

In this section we discuss the methods Riptide uses to overcome the challenges raised in Section 3, allowing it to generate fast end-to-end binary models. All the following described innovations are implemented and supported in both TensorFlow (Abadi et al., 2016) and TVM (Chen et al., 2018a). We use TensorFlow for training binary networks and TVM for compiling efficient machine code. The combination of these two halves makes Riptide an effective one-stop solution to training and deploying binary networks.

Algorithm 3 Riptide inference with N-bit activations.

Input: Input tensor X, binary layers L, weight scaling bits wb, shiftnorm scaling bits sb, and combined centering term cb.
$c_0 = \text{NormalConv}(X)$ {Full precision first block.}
$b_0 = \text{BatchNorm}(c_0)$
$q_0 = \text{LinearQuantize}(b_0)$
$a_0 = \text{BitPack}(q_0)$
for $k = 1$ **to** L **do**
  $c_k = \text{BinaryConv}(a_{k-1})$ {$\frac{NKKFHWC}{42}$ ops.}
  $q_k = (c_k + cb) \gg (wb + sb)$ {2HWF ops.}
  $l_k = \text{clip}(q_k, 0, 2^N - 1)$ {HWF ops.}
  $p_k = \text{Pooling}(l_k)$ {HWF ops.}
  $a_k = \text{BitPack}(p_k)$ {At least $3HWF$ ops.}
end for
$Y = \text{BinaryDense}(a_L)$

4.1 Quantization Technique and Polarity

As discussed, quantization methods that are not bit-consistent have fundamental issues being implemented in a bitserial way. As bitserial computation is essential to realizing speedups, we are forced to use one of the bit-consistent techniques. It remains an open question whether bit-inconsistent binarization can be implemented efficiently. We choose to use linear quantization in the style of Equation 2 as it does not require any floating point multiplication in its inner loop and has been shown to yield high accuracy models. However, there remain major barriers to supporting both it's bipolar and unipolar variants. To provide a deeper understanding of the impact of polarity, and offer as fine a granularity as possible in the trade-off between speed and accuracy, Riptide supports both unipolar and bipolar activation quantization.

Supporting unipolar activation quantization requires solving the polarity mismatch described in Section 3.1. There are a few possible solutions to this dilemma. Perhaps the most direct solution would be to get rid of the polarity mismatch by quantizing both activations *and* the weights unipolarly. Although this would allow a fast implementation by replacing bitwise-xnor with bitwise-and, it would also require that weight values be strictly positive. Because weights represent correlation with specific patterns, removing negative weights is similar to preventing a network from representing inverse correlation, which is highly destructive to accuracy.

Instead, we can treat the weight values *as if* they're unipolar.

Then, the bitwise-and operation between activations and weights is correct except when the activation bit is 1 and weight bit is 0. In this case, the product should have been -1 but is instead 0. To handle these cases, we count them and subtract it from the accumulation. This solution is given in Equation 3

$$a \cdot w = \sum_{n=0}^{N-1} 2^N (popc(a_n \wedge w) - popc(a_n \wedge !w)) \quad (3)$$

Here, we use two popcounts and bitwise-and operations and a bitwise invert (!) instead of the single popcount-xnor used in bipolar quantization. While the unipolar representation requires double the compute of the bipolar representation, the number of memory operations is the same.

4.2 Fused Binary Glue

Figure 1 demonstrates that the significant speedups (up to 43×) offered by binary layers pushes the cost of convolution and dense layers so low that it causes glue layers (lines marked with $\star$ in Algorithms 1 and 2) to become a major bottleneck at inference time. We seek to replace each such glue layer with bitserial operations. To this end, we introduce a novel operator that completely replaces all floating point glue layers while requiring only efficient bitserial addition and shifting. This new **Fused Glue** operator allows Riptide to simplify the forward pass of a binary model to the definition in Algorithm 3, where line 8 is our fused glue operation. Here we introduce the fused glue layer and explain how it works.

The glue layers in a traditional binary network perform three key functions: the WeightScale operation in line 6 of Algorithm 2 propagates the magnitude of weights into the activation tensor while the BatchNorm layer in line 8 of Algorithm 1 normalizes and centers the activations. Because these operations require floating point arithmetic, the remaining glue layers exist to cast and convert activations from integer to float and back again. If we could simply remove weight scaling, activation normalization, and activation centering, the rest of the glue layers wouldn't be required. Unfortunately, all three functions are essential to generating high quality quantizations. Instead, we seek to replace these floating point operations with efficient integer versions. Indeed, the three constants wb, sb, and cb in Algorithm 3 represent weight scaling, normalization, and centering terms respectively.

Weight Scaling: Multiplying the output of a bitserial operation by the scale term $a_k = \text{mean}(|W_k|)$ where k is the number of filters or units in weight tensor W has been a staple of BNNs since it was introduced in XNOR-Nets. This simple modification allows the relative magnitude of weight tensors to be preserved through quantization and gives a dramatic boost to accuracy while adding few operations. To maintain this functionality and preserve the integer domain, we replace weight scaling with an approximate power of two (AP2) bitwise shift. AP2 and its gradient g_x is defined in Equation 4.

$$\begin{aligned} AP2(x) &= 2^{\text{round}(log_2(|x|))} \\ g_x &= g_{AP2(x)} \end{aligned} \quad (4)$$

This allows us to approximate the multiplying of tensor A with weight scale α as $A \cdot \alpha_k \approx A \gg -log_2(AP2(\alpha_k))$ where $\gg$ is a bitwise right shift. Note that the term $-log_2(AP2(\alpha_k)$ is constant at inference time, so this scaling requires only a single shift operation, which is much more efficient than a floating point multiply on most hardware. However, right shifting is equivalent to a floor division when we'd really like a rounding division to preserve optimal quantization bins. Fortunately, $\text{round}(x) = \text{floor}(x+0.5)$ so we need only add the integer domain equivalent of 0.5 to a_k before shifting. Thus, Riptide's full weight scaling operation is defined in Equation 5.

$$\begin{aligned} wb &= -log_2(AP2(\alpha_k)) \\ q(a) &= (a + (1 \ll (wb - 1))) \gg wb \end{aligned} \quad (5)$$

Although the addition of the term $(1 \ll (wb-1))$ increases the amount of compute used, we will soon show that it can be fused with a centering constant without requiring any extra operations.

Normalization: We can extend Equation 5 to support activation normalization by approximating the variance of the activation tensor using AP2. Then, instead of dividing activation tensor A by its filter-wise variance σ_k, we can perform a right shift by sb bits, where sb is defined in Equation 6. Thus, we can perform a single right shift by $wb + sb$ bits to both propagate the magnitude of weights, and normalize activations. Equation 5 thus becomes Equation 6.

$$\begin{aligned} sb &= log_2(AP2(\sqrt{\sigma_k^2 + \epsilon})) \\ q(a) &= (a + (1 \ll (wb - 1))) \gg (wb + sb) \end{aligned} \quad (6)$$

We keep track of the running average of variance during train time so that the term $wb + sb$ is a constant during inference.

Centering: Finally we extend Equation 6 to center the mean of activations around zero. The simplest way of centering a tensor is by subtracting it's mean. Because this is a subtraction rather than a division or multiplication, we can not simply add more shift bits. Instead, we must quantize the mean of activation tensor A in an equivalent integer format so that it can be subtracted from the quantized activations. To this end, we use fixed point quantization (FPQ) as defined in Algorithm 4. The number of relevant bits in the output of an N-bit bitserial layer is $N + wb$, where the top

N bits form the quantized input to the next layer and the remaining wb bits are effectively fractional values. Thus we set $B = N + wb$ in Algorithm 4. Next we must determine the proper range, or scale, term to use in the quantization. This value should be equal to the floating point value that setting all $N + wb$ bits represents. By linear quantization's construction, setting the top N bits represents a value of 1 and the least significant of those N bits represents the value $\frac{1}{2^N-1}$. The value of setting all remaining wb bits is the geometric sum $\sum_{i=1}^{wb} \frac{1}{2^N-1}(\frac{1}{2})^i$ which simplifies to $\frac{1}{2^N-1}(1 - \frac{1}{2^{wb}})$. Thus, setting all $N + wb$ bits is equivalent to the floating point value $S = 1 + \frac{1}{2^N-1}(1 - \frac{1}{2^{wb}})$.

Algorithm 4 Fixed point quantization (FPQ) function.

1: **Input:** a tensor X to quantize to B bits with scale S.
2: $\hat{X} = \text{clip}(X, -S, S)$
3: $g = \frac{S}{2^B-1}$ {Compute granularity.}
4: $Y = \text{round}(\frac{\hat{X}}{g})$

With S and B properly defined, we can compute a quantized mean $\hat{\mu}$ from a floating point mean μ as $\hat{\mu} = FPQ(\mu, B, S)$ and directly subtract the result from binary activations. Conveniently, $\hat{\mu}$ can be subtracted from $(1 \ll (wb-1))$ to create a new centering constant. Equation 7 is thus the final form of Equation 6 and allows weight scaling, normalization, and centering in just two integer operations.

$$\begin{aligned} cb &= (1 \ll (wb - 1)) - \hat{\mu} \\ q(a) &= (a + cb) \gg (wb + sb) \end{aligned} \tag{7}$$

As in the case of variance, we keep track of the running mean of activations during train time so that during inference cb is a constant.

We thus have fully derived the fused glue operation used in Algorithm 3. The only other point worth noting is that we use $\text{clip}(q_k, 0, 2^N - 1)$ as the activation for our network. This has a similar effect as a saturating ReLU that bunches large activations into the highest quantization bin. We demonstrate in Section 5 that Riptide's fused glue layers not only are dramatically faster than floating point glue, but also do not negatively impact a binarized model's accuracy.

4.3 Generating Efficient Code

To compile our described algorithms to efficient machine code, we extend TVM (Chen et al., 2018a) to support bitserial operations. This allows Riptide to directly convert its TensorFlow training graph to a TVM based representation that can leverage LLVM to compile to multiple backends. Additionally, supporting bitserial operations in TVM allows Riptide to apply TVM's scheduling primitives to bitserial operations. These scheduling primitives include:

- **Tiling**, which splits loops over a tensor into small regions that can better fit into the cache, thereby reducing memory traffic and increasing compute intensity.
- **Vectorization**, which enables the use of hardware SIMD instructions to operate on multiple tensor elements simultaneously.
- **Parallelization**, which allows loops to be executed on multiple cores via threading.

Although these primitives require well chosen hyperparameters to maximize performance, we leverage AutoTVM (Chen et al., 2018b) to automatically search and find high quality settings. In addition to TVM scheduling primitives, we replace the default LLVM ARM popcount kernel with a more efficient **Fast Popcount** following the recommendations of Cowan et al. (2018).

To address the memory bottleneck shown in Figure 2, we introduce an optimization we call **Bitpack Fusion**, visualized by the dotted red line. By folding our fused glue operation and bitpacking into an outer loop of the preceding bitserial convolution, we need only store a few instances of the integer output before bitpacking back into a more compact representation. By storing only a small number of integer outputs at a time, we can reduce the total amount of memory used to store activations by a factor of 16×. This memory reduction is not only potentially essential to running models on resource constrained platforms, but also increases execution speed by reducing the time spent on memory operations.

5 Evaluation

In our evaluation of Riptide, we consider two primary objectives.

1. Demonstrate that Riptide's optimizations do not cause accuracy loss relative to state-of-the-art binarization results.
2. Show that Riptide can produce high speed binary models and explore the impact of its various optimizations.

Most previous work in binarization has been evaluated on AlexNet (Krizhevsky et al., 2012), VGGNet (He et al., 2015), and Resnets (He et al., 2016). To directly compare against these results, we train these three models with multiple bitwidth and polarity configurations. In these comparisons, we consider HWGQ (Cai et al., 2017) the current state-of-the-art for high accuracy binary AlexNets and VGGNets and PACT-SAWB (Choi et al., 2019) the state-of-the-art for binarizing Resnets. For all models (including SqueezeNet), we binarize all layers except the input layer

Table 1. Accuracy and speed of related binarization work and our results

	Model	Name	1-bit	2-bit	3-bit	full precision
			ImageNet top-1 accuracy / Runtime (ms)			
1	AlexNet	Xnor-Net (Rastegari et al., 2016)	44.2% / —	— / —	— / —	56.6% / —
2	AlexNet	BNN (Courbariaux et al., 2015)	27.9% / —	— / —	— / —	— / —
3	AlexNet	DoReFaNet (Zhou et al., 2016)	43.6% / —	49.8% / —	48.4% / —	55.9% / —
4	AlexNet	QNN (Hubara et al., 2016)	43.3% / —	51.0% / —	— / —	56.6% / —
5	AlexNet	HWGQ (Cai et al., 2017)	— / —	52.7% / —	— / —	58.5% / —
6	VGGNet	HWGQ (Cai et al., 2017)	— / —	64.1% / —	— / —	69.8% / —
7	Resnet18	PACT-SAWB (Choi et al., 2019)	— / —	62.8% / —	— / —	70.4% / —
8	Resnet50	PACT-SAWB (Choi et al., 2019)	— / —	67.4% / —	— / —	76.9% / —
9	AlexNet	Riptide-unipolar (ours)	44.5% / 150.4	52.5% / 196.8	53.6% / 282.8	56.5% / 1260.0
10	AlexNet	Riptide-bipolar (ours)	42.8% / 122.7	50.4% / 154.6	52.4% / 207.0	56.5% / 1260.0
11	VGGNet	Riptide-unipolar (ours)	56.8% / 243.8	64.2% / 387.2	67.1% / 610.0	72.7% / 2420.0
12	VGGNet	Riptide-bipolar (ours)	54.4% / 184.1	61.5% / 271.4	65.2% / 423.5	72.7% / 2420.0
13	Resnet18	Riptide-unipolar (ours)	54.9% / 82.0	62.2% / 177.2	— / —	70.4% / 385.5
14	Resnet50	Riptide-unipolar (ours)	59.1% / 171.9	66.9% / 340.3	— / —	76.9% / 771.8

as is common practice in the literature. Notably, however, we find that binarizing the output dense layer does not negatively impact the accuracy of Riptide models.

Binarized models are most attractive in resource constrained environments where full precision models are too slow and memory hungry. To this end, all time measurements are made on a Raspberry Pi 3B (RPi) (Pi, 2015). The RPi has an ARM Cortex-A53 CPU with 4 cores clocked at 1.2 GHz. This processor (and RPi's other hardware) is similar to many embedded systems, and results measured here are likely to be representative of other platforms. Due to the precedent of previous binarization, all Riptide accuracy measurements are made using AlexNet, VGGNet, and Resnet. However in practice, these architectures are bulky enough that they would not be deployed to an RPi class device. In our detailed runtime analysis, we instead examine quantized versions of SqueezeNet (Iandola et al., 2016), a highly parameter and runtime efficient model that is commonly used in embedded applications. Although we do not provide extensive accuracy validations of SqueezeNet, we have confirmed that training SqueezeNet with Riptide's optimizations achieves a top-1 accuracy within 1% of SqueezeNet trained with other state-of-the-art approaches.

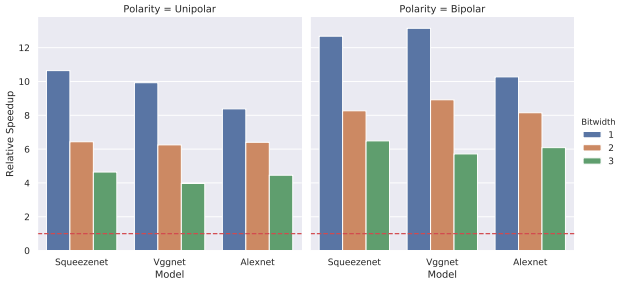

Figure 3. Visualization of end to end speedups of all explored models and bitwidths compared to floating point baselines.

5.1 End-to-End Results

The results of our training experiments and the accuracy reported by previous binarization works are reported in Table 1. Models are binarized using all Riptide optimizations and trained on the ImageNet dataset for 100 epochs using SGD with an initial learning rate of 0.1 that is decreased by 10× every 30 epochs. We train variants of AlexNet and VGGNet with 1-bit, 2-bit, and 3-bit activations, in all cases weights are quantized bipolarly with 1-bit except Resnets which use 2 bits. For baselines we train full precision versions of Alexnet, VGGNet, and Resnet using the same settings as above. We report the runtime of these baseline models when optimized using TVM. The speedup results for each model and bitwidth is visualized in Figure 3.

Although the author's of PACT-SAWB reported impressive accuracies of 67.0% and 72.2% for 2-bit Resnet18 and 2-bit Resnet50, we were unable to replicate these results. In our implementation, we instead reached the top-1 accuracies reported in Table 1, which are only marginally higher than those reported in DoReFaNet. Although it is possible that

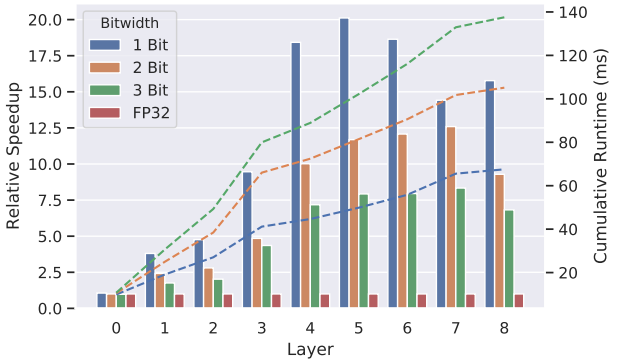

Figure 4. Layerwise speedup of SqueezeNet quantized with varying bitwidth versus the floating point baseline model.

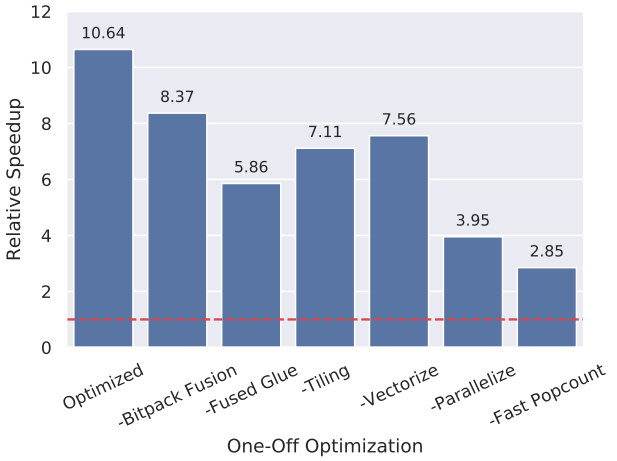

Figure 5. Ablation study of the effect of Riptide optimizations versus the baseline floating point model.

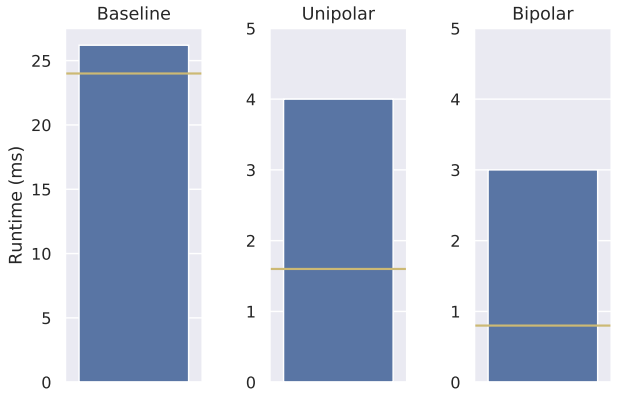

Figure 6. Effect of quantization polarity on the runtime of the first fire layer in SqueezeNet. The horizontal yellow line indicates the runtime of the layer if it were run with perfect efficiency.

the lower accuracy is due to an implementation mistake, it is difficult to verify as there is no open source PACT-SAWB implementation. However, it is worth noting that the techniques used in PACT-SAWB are entirely compatible with those used in Riptide, so it may be possible to improve Resnet accuracies by combining the two works.

There are three key takeaways from these results:

- Riptide is able to generate the first ever reported end-to-end speedups of a binary model and achieves accuracy comparable to the state-of-the-art across all configurations, confirming that Riptide's optimizations do not cause a drop in accuracy.
- We observe end-to-end speedups across all models and bitwidths and have a wide range of points in terms of the speed to accuracy trade-off; ranging from high accuracy 3-bit unipolar models with 4× speedup to high speed bipolar models with 12× speedup.
- Although unipolar models yield higher accuracy and slower runtimes than bipolar models as expected, they are only about 25% slower despite having twice the number of operations. This suggests our mixed polarity implementation (Equation 3) is quite efficient.

Taking these points together, we are confident that Riptide provides a high quality implementation of bitserial networks.

5.2 Layerwise Analysis

We measure the per-layer runtime of SqueezeNet unipolarly quantized at each bitwidth and visualize the results in Figure 4. The bars indicate the relative speedup versus a floating point baseline and the dotted lines tracks the cumulative runtime over the layers. There are a few interesting takeaways from this measurement. We see that not all layers benefit equally from quantization; those towards the end of the model have speedup of up to 20× compared to early layers' 3×. Early layers tend to be spatially larger but have fewer channels than later layers, suggesting that binarization scales better with the number of channel than it does spatially. Leveraging this knowledge, it may be possible to design architectures that are able to achieve higher speedups when binarized even if they are less efficient in full precision. We also note that the output dense layer achieves speedups inline with convolutional layers, suggesting our techniques apply well to both types of layer. Although we leave the input layer in full precision, we see that it takes a relatively small amount of the total runtime, suggesting that this is not a significant bottleneck.

5.3 Optimization Ablation

We perform a one-off ablation study of each of Riptide's optimizations. The results of this study are shown in Figure 5. Although not all optimizations contribute equally, its clear that they are all essential to our final highly performant model, with the smallest reduction lowering the end-to-end speedup from 10.6× to 8.4× and the largest reduction lowering speedups to only 2.9×. In the subsequent sections we drill down further into these optimizations to better understand their impact.

5.4 Polarity

To better examine the impact of polarity, we consider the runtime of the first layer of SqueezeNet for a baseline FP32 model, a unipolar model, and a bipolar model with the latter two being quantized unipolarly with 1-bit in Figure 6. Here, we have added a yellow line to indicate the runtime if the layer were entirely compute bound (calculated by dividing the number of operations by the RPi's frequency × core

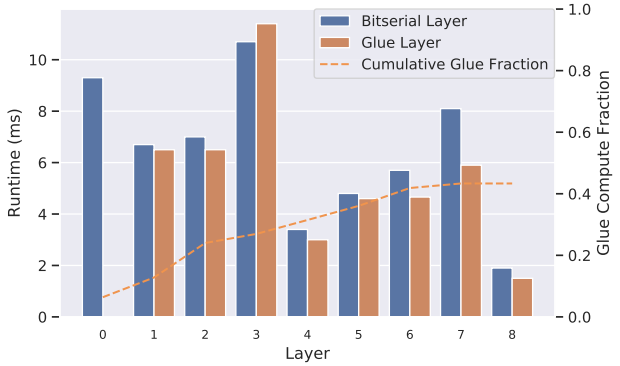

Figure 7. Layerwise runtime breakdown of SqueezeNet quantized with 1-bit weights and activations.

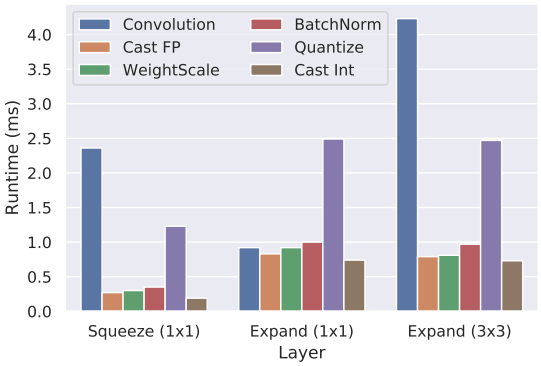

Figure 9. Runtime of each operation in the first fire layer of SqueezeNet quantized with 1-bit weights and activations.

count). We see that the baseline model is extremely close to its hypothetical ceiling, suggesting it is in fact quite compute bound. The binarized models on the other hand are far from compute bound. Because Riptide's unipolar quantization requires no more memory operations than bipolar, it is only marginally slower. Thus in general, unipolar quantization tends to give better accuracy to speedup trade-offs. However, in situations where speed is more important than accuracy, bipolar models may be more attractive.

5.5 Scheduling

To demonstrate the impact of proper machine code generation, we take a SqueezeNet binarized unipolarly with 1-bit and compile it with no optimizations then measure its end-to-end runtime. From that unoptimized starting point, we incrementally apply the optimizations discussed in Section 4.3 and measure the resulting runtimes. The speedups of these measurements are shown in Figure 8. We note that

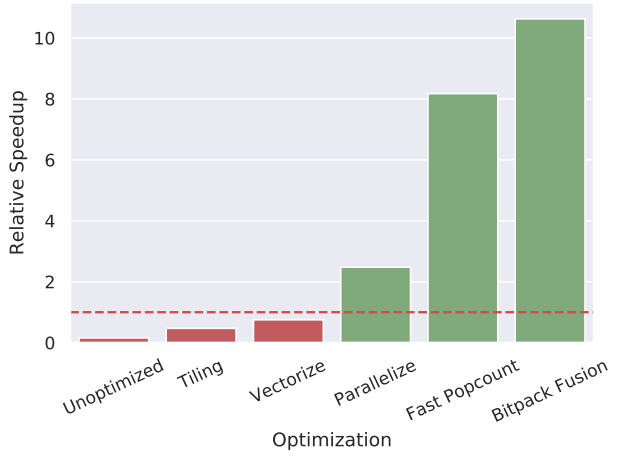

Figure 8. End-to-end speedup of quantized SqueezeNet versus the baseline floating point model when scheduling optimizations are incrementally applied.

an unoptimized binary model is actually about 7× **slower** than the full precision baseline. We can see that each optimization contributes significantly to generating performant machine code. Notably, we find that bitpack fusion gives a speed boost of about 30%, all of which comes from the reduction to memory operations that it contributes.

5.6 Glue Layers

To understand the impact of our fused glue operation, we examine the runtime of SqueezeNet when all optimizations except fused glue are applied. We first measure the runtime of each bitserial layer and its corresponding glue and visualize the results in Figure 7. We find that glue layers make up a considerable portion of each layer and cumulatively contribute 44% of the total inference time. This confirms that glue layers are a major bottleneck at all points in the model. Next, we examine each individual operation in the first quantized layer of SqueezeNet and visualize the runtimes in Figure 9. Here we find that although the Quantize operation is the largest of the glue operations, all contribute non-trivially. This leads us to conclude that optimizing only a portion of the glue layers would be insufficient and give us confidence that our fused glue operation is essential and highly performant.

6 Conclusion

In this paper we introduce Riptide, an end-to-end system for training and deploying high speed binarized networks. In our development of Riptide, we encounter and address numerous design flaws in existing binary networks. We propose a solution to unipolar quantization implementability, derive a novel "Fused Glue" interlayer operation that completely removes floating point arithmetic from binary networks, and demonstrate how to generate efficient bitserial machine code. We show that Riptide's optimization techniques lead to order of magnitude speedups and do not sacrifice accuracy relative to state-of-the-art approaches.

REFERENCES

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. Tensorflow: a system for large-scale machine learning. In *OSDI*, volume 16, pp. 265–283, 2016.

Cai, Z., He, X., Sun, J., and Vasconcelos, N. Deep learning with low precision by half-wave gaussian quantization. *arXiv preprint arXiv:1702.00953*, 2017.

Chen, T., Moreau, T., Jiang, Z., Shen, H., Yan, E., Wang, L., Hu, Y., Ceze, L., Guestrin, C., and Krishnamurthy, A. Tvm: End-to-end optimization stack for deep learning. *arXiv preprint arXiv:1802.04799*, 2018a.

Chen, T., Zheng, L., Yan, E., Jiang, Z., Moreau, T., Ceze, L., Guestrin, C., and Krishnamurthy, A. Learning to optimize tensor programs. *arXiv preprint arXiv:1805.08166*, 2018b.

Choi, J., Chuang, P. I.-J., Wang, Z., Venkataramani, S., Srinivasan, V., and Gopalakrishnan, K. Bridging the accuracy gap for 2-bit quantized neural networks (qnn). *SysML*, 2019.

Courbariaux, M., Bengio, Y., and David, J.-P. Binaryconnect: Training deep neural networks with binary weights during propagations. In *Advances in Neural Information Processing Systems*, pp. 3123–3131, 2015.

Courbariaux, M., Hubara, I., Soudry, D., El-Yaniv, R., and Bengio, Y. Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. *arXiv preprint arXiv:1602.02830*, 2016.

Cowan, M., Moreau, T., Chen, T., and Ceze, L. Automating generation of low precision deep learning operators. *arXiv preprint arXiv:1810.11066*, 2018.

Dong, Y., Ni, R., Li, J., Chen, Y., Zhu, J., and Su, H. Learning accurate low-bit deep neural networks with stochastic quantization. *arXiv preprint arXiv:1708.01001*, 2017.

Fromm, J., Patel, S., and Philipose, M. Heterogeneous bitwidth binarization in convolutional neural networks. *arXiv preprint arXiv:1805.10368*, 2018.

He, K., Zhang, X., Ren, S., and Sun, J. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pp. 1026–1034, 2015.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

Hubara, I., Courbariaux, M., Soudry, D., El-Yaniv, R., and Bengio, Y. Quantized neural networks: Training neural networks with low precision weights and activations. *arXiv preprint arXiv:1609.07061*, 2016.

Iandola, F. N., Han, S., Moskewicz, M. W., Ashraf, K., Dally, W. J., and Keutzer, K. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and¡ 0.5 mb model size. *arXiv preprint arXiv:1602.07360*, 2016.

Ioffe, S. and Szegedy, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International Conference on Machine Learning*, pp. 448–456, 2015.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pp. 1097–1105, 2012.

Li, F., Zhang, B., and Liu, B. Ternary weight networks. *arXiv preprint arXiv:1605.04711*, 2016.

Pi, R. Raspberry pi model b, 2015.

Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F., and Amarasinghe, S. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *ACM SIGPLAN Notices*, 48 (6):519–530, 2013.

Rastegari, M., Ordonez, V., Redmon, J., and Farhadi, A. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision*, pp. 525–542. Springer, 2016.

Tang, W., Hua, G., and Wang, L. How to train a compact binary neural network with high accuracy? In *AAAI*, pp. 2625–2631, 2017.

Xianyi, Z., Qian, W., and Chothia, Z. Openblas. *URL: http://xianyi. github. io/OpenBLAS*, 2014.

Zhou, S., Wu, Y., Ni, Z., Zhou, X., Wen, H., and Zou, Y. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv preprint arXiv:1606.06160*, 2016.