Streamlined Deployment for Quantized Neural Networks

Yaman Umuroglu, Magnus Jahre
Norwegian University of Science and Technology
Trondheim, Norway
yamanu@ntnu.no

ABSTRACT

Running Deep Neural Network (DNN) models on devices with limited computational capability is a challenge due to large compute and memory requirements. Quantized Neural Networks (QNNs) have emerged as a potential solution to this problem, promising to offer most of the DNN accuracy benefits with much lower computational cost. However, harvesting these benefits on existing mobile CPUs is a challenge since operations on highly quantized datatypes are not natively supported in most instruction set architectures (ISAs). In this work, we first describe a *streamlining* flow to convert all QNN inference operations to integer ones. Afterwards, we provide techniques based on processing one bit position at a time (bit-serial) to show how QNNs can be efficiently deployed using common bitwise operations. We demonstrate the potential of QNNs on mobile CPUs with microbenchmarks and on a quantized AlexNet, which is 3.5× faster than an optimized 8-bit baseline.

1 INTRODUCTION

From voice recognition to object detection, *Deep Neural Networks (DNNs)* are steadily getting better at extracting information from complex raw data. Combined with the popularity of mobile computing and the rise of the Internet-of-Things (IoT), there is enormous potential for widespread deployment of intelligent devices, but a computational challenge remains. A modern DNN can require billions of floating point operations to classify a single image, which is far too costly for energy-constrained mobile devices. Offloading DNNs to powerful servers in the cloud is only a limited solution, as it requires significant energy for data transfer and cannot address applications with real-time or low-latency requirements, such as augmented reality or navigation for autonomous drones.

Quantized Neural Networks (QNNs) have recently emerged as a potential solution to this problem. They contain convolutional, fully-connected, pooling and normalization layers similar to the floating point variants, but use a constrained set of values to represent each weight and activation in the network. We will use the notation $\mathbf{W^w A^a}$ to refer to a QNN with w-bit weights and a-bit activations, and focus on cases where they represent *few-bit integers* ($w, a \leq 4$). The computational advantages of such QNNs are two-fold:

(1) Each parameter and activation can be represented with a few bits. A greater portion of the working set can thus be kept in on-chip memory, enabling greater performance, reducing off-chip memory accesses and the energy cost of data movement.
(2) Most QNN operations are on few-bit integers, which are faster and more energy-efficient than floating-point.

While a quantized network will generally have reduced accuracy compared to an equivalent DNN using floating point, recent research has demonstrated significant progress in closing this accuracy gap. Courbariaux and Hubara et al. [5] first demonstrated that Binarized Neural Networks (BNNs), a QNN variant with $\mathbf{W^1 A^1}$, could achieve competitive accuracy on smaller image recognition benchmarks like CIFAR-10 and SVHN. XNOR-Net [7] improved upon this technique by adding scaling factors to better approximate the full-precision operations. Noting that more challenging classification tasks such as ImageNet could benefit from higher-precision activations, DoReFa-Net [9] used multi-bit activations and weights to further improve accuracy. Recently, Cai et al. [1] proposed Half-wave Gaussian Quantization (HWGQ) to take advantage of the Gaussian-like distribution of batch-normalized activations, demonstrating $\mathbf{W^1 A^2}$ networks with less than 5% top-5 accuracy drop compared to floating point DNNs on the challenging ImageNet dataset, as summarized in Table 1.

Table 1: Accuracy of a state-of-the-art QNN [1].

Dataset	Network	Floating Point top-1 (top-5)	$\mathbf{W^1 A^2}$ HWGQ [1] top-1 (top-5)
ImageNet	AlexNet	58.5% (81.5%)	52.7% (76.3%)
ImageNet	GoogLeNet	71.4% (90.5%)	63.0% (84.9%)
ImageNet	VGG-like	69.8% (89.3%)	64.1% (85.6%)
CIFAR-10	VGG-like	93.2%	92.5%

Despite the attractive accuracy and computational properties, there is a challenge in reaping the benefits on QNNs on mobile devices with commodity processors. Three outstanding issues limit the benefits of QNN deployment on existing mobile CPUs: floating point parameters inside and between quantized layers, lack of native support for efficient few-bit integer matrix multiplications, and overhead of bit-masking operations for convolution lowering on few-bit activations. In this work, we show how these problems can be addressed by absorbing floating point operations into thresholds (*streamlining*), using a bit-serial formulation for handling few-bit integer matrix multiplications and channel-interleaved lowering.

2 STREAMLINED QNNS

Even for layers with uniform-quantized input activations and weights, state-of-the-art QNN methods use some floating point computation in the forward pass to improve the accuracy. Although these layers do not typically contain a large amount of computation, they may still incur slowdowns on devices where floating point operations are expensive and increase the memory footprint of the QNN by adding floating point parameters. Three such examples from state-of-the-art QNN methods are:

(1) **Batch normalization.** Almost all state-of-the-art QNN techniques, including BinaryNet [5], XNOR-Net [7] and HWGQ [1], use batch normalization to obtain zero mean and unit variance prior to quantizing activations. The normalization

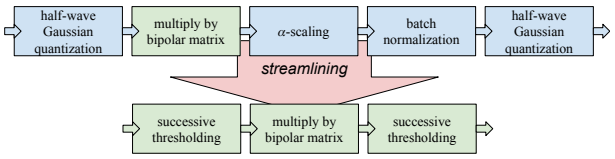

Figure 1: Streamlining an HWGQ network. Blue and green color indicate floating point and integer data, respectively.

parameters μ and i are floating point values obtained during network training.

(2) **α-scaling.** To better approximate the full-precision results using quantized operations, both XNOR-Net [7] and HWGQ [1] use α-scaling. This involves multiplying the quantized matrix multiplication result with α, which is a floating point vector containing the average L1-norm of each row of the weight matrix prior to quantization.

(3) **Non-integer quantization levels.** The chosen quantization levels in a QNN may be floating point values to best approximate the underlying value distribution. For instance, the state-of-the-art QNNs produced by HWGQ [1] use the following function for 2-bit uniform quantization:

$$\text{HWGQ}(x) = \left\{ \begin{array}{ll} 0, & \text{for } x \leq t_0 \\ 0.538, & \text{for } 0 < x \leq 0.807 \\ 1.076, & \text{for } 0.807 < x \leq 1.345 \\ 1.614, & \text{for } 1.345 < x \end{array} \right\}$$

2.1 The Streamlining Algorithm

Through a process we call *streamlining*, we show how the forward pass through any QNN layer with uniform-quantized activations and weights can be computed using only integer operations. This consists of the following three steps:

2.1.1 Quantization as successive thresholding. Given a set of threshold values $t = \{t_0, t_1 \dots t_n\}$, the successive thresholding function $T(x, t)$ maps any real number x to an integer in the interval $[0, n]$, where the returned integer is the number of thresholds that x is greater than or equal to:

$$T(x,t) = \left\{ \begin{array}{ll} 0, & \text{for } x \leq t_0 \\ 1, & \text{for } t_0 < x \leq t_1 \\ \dots & \dots \\ n-1, & \text{for } t_{n-2} < x \leq t_{n-1} \\ n, & \text{for } t_{n-1} < x \end{array} \right\}$$

Any uniform quantizer $Q(x)$ can be expressed as successive thresholding followed by a linear transformation such that $Q(x) = a \cdot T(x) + b$. As an example, the 2-bit uniform HWGQ quantizer can be expressed as $\text{HWGQ}(x) = 0.538 \cdot T(x, t)$ with $t_0 = 0, t_1 = 0.807, t_2 = 1.345$. It should be noted that this technique is only economical for few-bit activations, since the number of thresholds grows exponentially with the activation bitwidth.

2.1.2 Moving and collapsing linear transformations. Any sequence of linear transformations can be collapsed into a single linear transformation. We can first move all floating point linear operations to be positioned *between* the quantized matrix operation and the activation quantization, then collapse them into a single linear transformation. For the example in Figure 1, the linear transformation $ax+b$ for the previous layer's activation quantization can be moved past the bipolar matrix multiplication, since $W \cdot (ax+b) = a \cdot (Wx) + Wb$, forming a sequence together with the α-scaling and batch normalization. Afterwards, this sequence of three linear transformations can be reduced to a single linear transformation.

2.1.3 Absorbing linear operations into thresholds. The final step in the streamlining process is to update the threshold values as $t_i \leftarrow (t_i - b)/a$ using the parameters a, b of the linear transformation. Observe that in the inequality $t_0 < x \leq t_1$ we can substitute $ax + b$ as the variable, and rewrite it as $(t_0 - b)/a < x \leq (t_1 - b)/a$. By updating each threshold in this manner, we can remove the floating point linear transformations completely and feed the result of the quantized matrix operation directly into the successive thresholding layer. Furthermore, if the input to the quantized matrix operation is known to be integer (i.e. the previous layer's activations were also quantized), each threshold can be simply rounded up to the nearest integer without changing the produced results.

3 INFERENCE WITH FEW-BIT WEIGHTS AND ACTIVATIONS ON MOBILE CPUS

The dominating computation in QNN inference is convolutions between feature maps and kernels expressed as few-bit integers, which can be *lowered* [2] to matrix-matrix multiplication between few-bit integer matrices. Both the lowering and the matrix multiplications can be carried out by casting all operands to 8-bit integers, which are natively supported by most ISAs today. Libraries such as Google's gemmlowp [4], which has been used to deploy DNNs on mobile devices, offer high-performance 8-bit matrix multiplications. However, using 8-bit operands to carry out few-bit integer operations can be wasteful. For instance, using 8-bit operations to compute a $\mathbf{W^2A^2}$ matrix product would insert six zero bits into each operand, thus unnecessarily increasing the memory footprint by 4×. Here, we provide alternatives that take advantage of few-bit integers for both the lowering and matrix multiplication operations.

3.1 Few-Bit Integer Matrix Multiplication

To perform efficient few-bit integer matrix multiplication, we propose to use commonly-supported bitwise operations in *bit-serial* fashion. We will first describe how this is done for the $\mathbf{W^1A^1}$ case, then generalize the method to $\mathbf{W^wA^a}$.

The $\mathbf{W^1A^1}$ case. Binary matrix multiplication, which we refer to as BinaryGEMM, can be used for the case where each weight and activations can be represented using a single bit. Previous work [5–7] discussed how binary dot products can be implemented using bitwise XNOR followed by popcount (counting the number of set bits) operations. Most modern processors provide an instruction for popcount, which enables fast BinaryGEMM implementations even on mobile CPUs. Note that very high performance (in the trillion-operations per second range) on these operations can also be achieved with FPGAs [6, 8] and GPGPUs [5]. Although XNOR-popcount is only applicable for matrices with $\{-1, +1\}$ binary elements, it is possible to extend this idea to $\{0, 1\}$ binary elements by using bitwise AND instead of XNOR as shown in Algorithm 1.

```
function BinaryGEMM(W, A, res, α)
  for r ← 0 . . . rows − 1 do
    for c ← 0 . . . cols − 1 do
      for d ← 0 . . . ⌈depth/wordsize⌉ − 1 do
        res[r][c] += α · Popcount(W(r, d) & A(c, d))
```

Algorithm 1: $\mathbf{W^1A^1}$ GEMM using AND-popcount.

```
function BitSerialGEMM(W, A, res)
  for i ← 0 . . . w − 1 do
    for j ← 0 . . . a − 1 do
      sgnW ← (i == w − 1 ? −1 : 1)
      sgnA ← (j == a − 1 ? −1 : 1)
      BinaryGEMM(W[i], A[j], res, sgnW · sgnA · 2^(i+j))
```

Algorithm 2: Signed $\mathbf{W^wA^a}$ GEMM using BinaryGEMM.

The $\mathbf{W^wA^a}$ case. We now leverage BinaryGEMM as a building block for implementing few-bit integer matrix multiplication. We can rewrite the w-bit matrix W as a weighted sum $\sum_{i=0}^{w-1} 2^i \cdot W[i]$, where $W[i]$ is the binary matrix formed by taking bit i of each element of W, also referred to as a *bit plane*. In this manner, the product of two few-bit integer matrices can be written as a weighted sum of the pairwise products of their bit planes, i.e. $W \cdot A = \sum_{i=0}^{w-1} \sum_{j=0}^{a-1} 2^{i+j} \cdot W[i] \cdot A[j]$.

Algorithm 2 uses this observation to formulate few-bit integer matrix multiplication between the w-bit weight matrix W and the a-bit activation matrix A. This is a *vectorized bit-serial* operation, since the contributions to each result element are computed between two bit positions at a time, but the bitwise operations inside BinaryGEMM operate on vectors of bits. In this manner, we are able to take advantage of the full width of the processor datapath without introducing a large number of zero bits inside operations regardless of the values of w and a. The time taken by BitSerialGEMM will be proportional to $w \cdot a$, with $\mathbf{W^1A^1}$ executing fastest.

3.2 Lowering with Few-Bit Activations

Lowering convolutions [2] allows taking advantage of optimized matrix-matrix multiplications, which is the approach we take in this work. Memory usage is typically a concern for lowering due to duplicated pixels. In theory, QNNs do not suffer as much from this problem since quantized activations use much fewer bits per pixel, but taking advantage of this on a CPU can be tricky. Namely, the lowering process itself (often called *im2col*) requires accessing the feature map data in a "sliding window" fashion, which may require bit masking and shifting operations that decrease performance. Representing each few-bit activation as an 8-bit value avoids this problem, but introduces many unused zero padding bits.

We propose an alternative, which we refer to as *interleaved lowering*, that uses a bit-serial, channel-interleaved data layout as illustrated in Figure 2. Each pixel, which may span one or more CPU words, contains the bits from one bit position across activations from all channels, padded to the nearest word boundary. Afterwards, the lowering can be performed on the granularity of entire CPU words, with one im2col per activation bit.

4 EVALUATION

We implemented BitSerialGEMM using ARM NEON intrinsics in C++, with register blocking and L1 cache blocking to achieve higher performance. We compare against the gemmlowp library

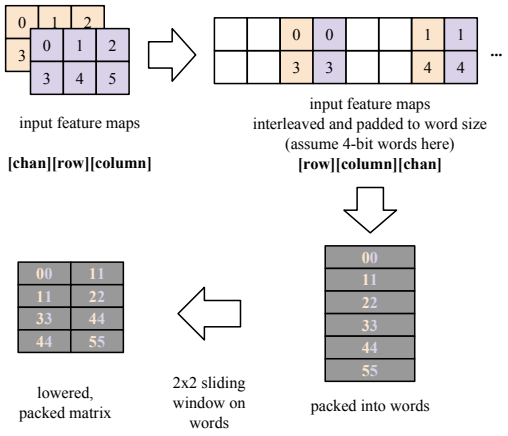

Figure 2: Interleaved lowering.

[4], which utilizes hand-optimized inline assembly for 8-bit matrix multiplications. All reported results are measured on a single ARM Cortex-A57 core running at 1.9 GHz on the Nvidia Jetson TX1 board. We use 8-bit native matrix multiplications provided by gemmlowp [4] as the baseline alternative to BitSerialGEMM.

4.1 Matrix Multiplication Microbenchmarks

As matrix multiplication accounts for the majority of time in neural network inference, we start by evaluating BitSerialGEMM on matrix multiplication microbenchmarks. For a (rows, depth, cols) operation that takes T nanoseconds, we report the performance in integer giga-operations per second (GOPS) measurement as $(2 \cdot \text{rows} \cdot \text{depth} \cdot \text{cols}) / T$ by averaging over a runtime of 10 s.

4.1.1 Compute-Bound Performance. To measure the maximum achievable performance with our implementation, we use the largest matrices that still fit into the L1 cache. For gemmlowp, we observed a peak performance of 22 GOPS. For BitSerialGEMM on $\mathbf{W^1A^1}$ (binary matrices), we observed a peak performance of 150 GOPS, which is 6.8× faster than using 8-bit operands. As expected, the performance linearly decreases with more bits of precision: 77 GOPS for $\mathbf{W^1A^2}$, 50 GOPS for $\mathbf{W^1A^3}$, 34 GOPS for $\mathbf{W^2A^2}$ and 23 GOPS for $\mathbf{W^2A^3}$. Thus, for this particular platform, BitSerialGEMM is faster than using 8-bit operations for $\mathbf{W^wA^a}$ with $w \cdot a \leq 6.8$ when working with in-cache matrices.

4.1.2 Performance vs Matrix Size. To investigate how performance is influenced by the dimensions of a $M \times N \times K$ matrix multiplication, we performed a sweep of different sizes between 2^6 and 2^{12} in each dimension using both gemmlowp and BitSerialGEMM with $\mathbf{W^1A^1}$. Figure 3 presents a scatter plot of the performance with increasing depth (K). We observe that BitSerialGEMM performance is sensitive to the depth (K) dimension. For small matrix sizes, there is little or no performance advantage over gemmlowp, which should be taken into consideration when choosing the execution method for each layer. BitSerialGEMM quickly becomes faster with increasing depth and becomes advantageous over gemmlowp, up to 6.6× faster than gemmlowp for a $64 \times 1024 \times 4096$ multiplication. With $K \geq 2048$, we observe decreased performance for larger M and N values in BitSerialGEMM

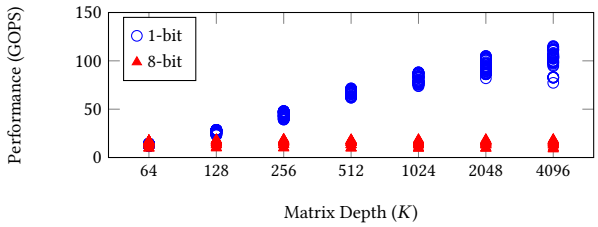

Figure 3: Log-linear plot of performance versus depth (K).

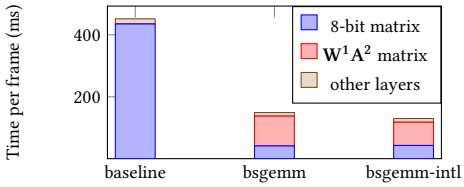

Figure 4: Overall performance for quantized AlexNet.

due to increased cache misses, which can be addressed by adding more levels of blocking to the implementation.

4.2 Quantized AlexNet

To assess the benefits of the techniques discussed for QNN deployment, we developed a version of Caffe with support for quantized layers. Each quantized layer can be configured individually to use either gemmlowp or BitSerialGEMM as the execution engine. We use a quantized $\mathbf{W^A A^1}$exNet from [1] with batch size 1 as a benchmark, with $\mathbf{W^8 A^8}$ for the first layer, $\mathbf{W^8 A^2}$ for the last layer, and $\mathbf{W^1 A^2}$ for all other matrix layers. The first and last layer are computed using gemmlowp in 8-bit precision to preserve accuracy. For non-matrix layers such as thresholding and max pooling which constitute a tiny portion of the total compute, we use regular floating point operations. We evaluated the performance for the following combinations of techniques:

- **baseline:** No streamlining, all layers using gemmlowp.
- **bsgemm:** Streamlining, all but the first and last layer using BitSerialGEMM, first and last layer in gemmlowp.
- **bsgemm-intl:** Streamlining and interleaving, all but the first and last layer using BitSerialGEMM .

4.2.1 Overall performance. Figure 4 compares the time per frame with the three techniques and presents a basic breakdown of time cost. Overall, bsgemm achieves a 3× speedup over the baseline, and bsgemm-intl further improves this to 3.5×. Speedups from bsgemm are limited by the presence of 8-bit first/last layers, which account for 33% of the execution time in bsgemm-intl. Also quantizing those layers further would bring further performance benefits. The current throughput is 2.2, 6.7 and 7.7 frames per second respectively for **baseline**, **bsgemm** and **bsgemm-intl**, and the performance can be further improved by multi-core parallelism and code optimization.

4.2.2 Detailed comparison. Table 2 presents a detailed breakdown of time spent in lowering and matrix multiplication operations across AlexNet convolutional and fully-connected layers with different optimizations. All matrix multiplications are indicated in parantheses as (rows, columns, depth), with the midline separating

Table 2: Time cost breakdown for $\mathbf{W^1 A^2}$ AlexNet with batch size 1. Best numbers for each row are highlighted.

	Operation	Time (ms)			Speedup
		baseline	bsgemm	bsgemm-intl	
convolutional	lowering	**6.7**	**6.7**	**6.7**	1×
	(96, 363, 3025)	**20.7**	**20.7**	**20.7**	1×
	lowering	8.7	15	**0.9**	10×
	(256, 2400, 729)	90.3	23.7	**23.7**	3×
	lowering	2.4	3.7	**0.2**	12×
	(384, 2304, 169)	32.2	8.3	**8.3**	4×
	lowering	3.5	5.7	**0.3**	10×
	(384, 3456, 169)	48	10.7	**10.7**	5×
	lowering	3.5	5.7	**0.3**	10×
	(256, 3456, 169)	35.8	7.3	**7.3**	5×
FC	(4096, 9216, 1)	114.7	2.3	**2.3**	50×
	(4096, 4096, 1)	52.9	1.1	**1.1**	50×
	(1000, 4096, 1)	**13**	**13**	**13**	1×

convolutional and fully-connected layers. The advantage of using quantized operations for fully-connected layers is especially prominent, with speedups of up to 50× owing to increased arithmetic intensity in matrix-vector multiplications. For matrix-matrix multiplications in convolutional layers, the advantage of using $\mathbf{W^1 A^2}$ BitSerialGEMM is around 4×. When the matrix multiplies become faster, the overhead of lowering and bit packing costs become substantial, almost as much as the matrix-matrix multiplication time for earlier layers with large filters. Fortunately, this can be remedied by interleaving, as indicated by the results in the **bsgemm-intl** column. By taking advantage of packing bits prior to lowering, interleaved lowering can offer a 10× speedup over the baseline.

5 CONCLUSION AND FUTURE WORK

We have presented methods for efficient processing of QNNs on mobile CPUs via absorbing scaling factors into thresholds, channel-interleaved lowering, and bit-serial matrix multiplication. Our results indicate that these methods can take better advantage of few-bit operations in QNNs, offering significant speedups over native 8-bit operations on mobile CPUs. As future work, we note that this approach can enable approximate computing by only considering the contributions from higher-order bits and taking advantage of bit-level sparsity. Another use for this technique would be on-device training DNNs using low-precision gradients [3], which also requires low-precision matrix operations.

REFERENCES

[1] Zhaowei Cai, Xiaodong He, Jian Sun, and Nuno Vasconcelos. 2017. Deep Learning with Low Precision by Half-wave Gaussian Quantization. In *CVPR*.

[2] Kumar Chellapilla, Sidd Puri, and Patrice Simard. 2006. High performance convolutional neural networks for document processing. In *Tenth International Workshop on Frontiers in Handwriting Recognition*. Suvisoft.

[3] Christopher De Sa, Michael Feldman, Kunle Olukotun, and Christopher Re. 2017. Understanding and Optimizing Asynchronous Low-Precision Stochastic Gradient Descent. In *Computer Architecture (ISCA), 2017 ACM/IEEE 44th Annual International Symposium on*. IEEE.

[4] Benoit Jacob et al. 2017. gemmlowp: a small self-contained low-precision GEMM library. (2017). https://github.com/google/gemmlowp.

[5] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2016. Binarized Neural Networks. In *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*.

[6] Eriko Nurvitadhi, David Sheffield, Jaewoong Sim, Asit Mishra, Ganesh Venkatesh, and Debbie Marr. 2016. Accelerating Binarized Neural Networks: Comparison

of FPGA, CPU, GPU, and ASIC. In *Field-Programmable Technology (FPT), 2016 International Conference on*. IEEE.

[7] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. 2016. XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks. In *ECCV*.

[8] Yaman Umuroglu, Nicholas J. Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. 2017. FINN: A Framework for Fast, Scalable Binarized Neural Network Inference. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '17)*. ACM.

[9] Shuchang Zhou, Zekun Ni, Xinyu Zhou, He Wen, Yuxin Wu, and Yuheng Zou. 2016. DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients. *CoRR* abs/1606.06160 (2016).