Precision Batching: Bitserial Decomposition for Efficient Neural Network Inference on GPUs

Maximilian Lam
Harvard University
Cambridge, Massachusetts, U.S
maxlam@g.harvard.edu

Zachary Yedidia
Harvard University
Cambridge, Massachusetts, U.S
zyedidia@college.harvard.edu

Colby R Banbury
Harvard University
Cambridge, Massachusetts, U.S
cbanbury@g.harvard.edu

Vijay Janapa Reddi
Harvard University
Cambridge, Massachusetts, U.S
vj@eecs.harvard.edu

***Abstract*—We present *PrecisionBatching*, a quantized inference algorithm for speeding up neural network inference on traditional hardware platforms at low bitwidths. *PrecisionBatching* is based on the following insights: 1) neural network inference with low batch sizes on traditional hardware architectures (e.g: GPUs) is memory bound, 2) activation precision is critical to improving quantized model quality and 3) matrix-vector multiplication can be decomposed into binary matrix-matrix multiplications, enabling quantized inference with higher precision activations at the cost of more arithmetic operations. Combining these three insights, *PrecisionBatching* enables inference at extreme quantization levels (< 8 bits) by shifting a memory bound problem to a compute bound problem and achieves higher compute efficiency and runtime speedup at fixed accuracy thresholds against standard quantized inference methods. Across a variety of applications (MNIST, language modeling, natural language inference, reinforcement learning) and neural network architectures (fully connected, RNN, LSTM), *PrecisionBatching* yields end-to-end speedups of over $8\times$ on a GPU within a $< 1-5\%$ error margin of the full precision baseline, outperforming traditional 8-bit quantized inference by over $1.5\times-2\times$ at the same error tolerance.**

I. Introduction

Recent advances in deep learning have demonstrated the wide range of the applications of neural networks [1]–[8], however, neural network execution remains computationally expensive. In the context of inference, where a trained neural network is executed to make predictions, these computational costs are even more significant as the quality of user facing products is often highly sensitive to the application's responsiveness. In these use cases, slow inference times not only degrade the usability of these applications (e.g: a robot arm must quickly be able to identify and manipulate an object; a text recognition system must react fast enough to ensure quality user experience; a voice recognition system must recognize speech quickly enough to enable real time interaction), but in some extreme cases may even restrict deployment (e.g: a drone must execute a neural network policy quickly to adapt to a changing environment, or risk crashing).

Research in quantization aims to reduce the computational costs of neural network inference by reducing the precision of neural network weights and activations [9]–[11], [11]–[14], however, this technique incurs an increasingly larger accuracy penalty when quantizing to lower bitwidths due to quantization error [10], [15]. Traditionally, without retraining, neural networks suffer significant accuracy degradation beyond 8 bit quantization [16], [17], limiting speedups to ˜$4\times$ the speed of the original network. With retraining, research has shown that networks may be quantized beyond 8 bits [9], [10], [18], however, retraining for quantization is computationally expensive, requires architectural changes to the network and converges slower [9]. Thus, in the context of quantization without retraining, it remains challenging to enable < 8 bit quantization without significantly degrading model quality, and, in the context of quantization with retraining, convergence time continues to be a major issue.

In this paper, we develop *PrecisionBatching*, a quantized inference algorithm for traditional hardware platforms to speed up low batch neural network inference. *PrecisionBatching* decomposes network weights and activations into 1 bit tensors, batches the 1 bit activations together, and performs quantized inference with low precision weights and high precision activations (see Figure 1). This attains speedups by reducing the precision of weight layers, maintains accuracy by keeping activations at higher precision, and utilizes the compute platform's higher arithmetic intensity to absorb the extra computation. Besides speedup, *PrecisionBatching* enables finer granularity control over the weight and activation precision of quantized inference, which may yield further speedups at a given accuracy. *PrecisionBatching* is a quantized inference method, a kernel, which specifies how a traditional compute platform (like a GPU) may efficiently perform quantized inference. This is unlike quantization algorithms such as [15], [17], [19] which specify how to quantize the network, but not how to execute over the lower precision values; thus *PrecisionBatching* may be used in conjunction with these quantization techniques.

We developed *PrecisionBatching* on three key observations:

- **Insight 1: Small Batch Neural Network Inference is Memory Bound**
  User facing products perform network inference with a small batch size to reduce response time / latency (it is not uncommon to see a batch size of 1) [20]–[22]. However, on traditional hardware platforms like GPUs, memory transfer speeds are much slower than arithmetic compute capabilities (FLOPs), so performing neural network inference with low batch size is *memory bound*, meaning most of the time executing the network is spent on fetching data, rather than on arithmetic computations [20], [23]. Specifically, in a regime with batch size 1, the data bottleneck is transferring the weight layers of the neural network, which incurs communication cost on the order of $O(mn)$ where m and n are sizes of the network's hidden layers; conversely, the memory cost of transferring activations is significantly less at $O(m + n)$. Observing that small batch neural network inference is memory bound is significant for two reasons: 1) it indicates that during neural network inference, the compute cores of the hardware platform are idle, suggesting that one may attain free compute cycles during this duration and 2) significant speedups may be attained by reducing the time spent on transferring weights.
- **Insight 2: More Bits for Activation Precision Improves Model Quality**
  Quantization literature has shown that using more precision for activations improves model quality [15], [17]. Intuitively this makes sense, for example, between a network with 4 bit weights and 4 bit activations and a network with 4 bit weights and 8 bit activations one would expect the one with higher precision activations to attain higher accuracy. Additionally, from insight 1, a single bit of precision for weights does not hold the same value as a single bit of precision for activations. Specifically, as inference is weight memory bound, reducing the precision of weights by a single bit is much more valuable than reducing the precision of activations by a single bit. Unfortunately, on traditional hardware platforms like GPUs, kernels fail to capitalize on this insight by requiring both operands of a computation (weights+activations) be the same precision.
- **Insight 3: Matrix Multiplication may be Decomposed Bitserially**
  Full precision matrix vector multiplication may be decomposed into a sum of 1 bit matrix-matrix multiplications; this logical decomposition is known as bitserial computation in the hardware architecture space [24], [25]. In the context of traditional hardware platforms like GPUs, implementing such a routine incurs significantly more arithmetic as the terms of the sum are separated out, which may be a reason why, to the best of our knowledge, such a kernel is not used widely. However, leveraging insights 1 (inference is memory bound) and 2 (activations improve model quality) we can see that such a bitserial kernel may yield significant gains: the extra arithmetic incurred by the routine is absorbed for free by the idle compute units and now one may perform inference with higher activation precision and lower weight precision, reducing the weight memory-boundedness of network inference and achieving a speedup.

To demonstrate the value of *PrecisionBatching*, we develop optimized computational kernels to perform our algorithm on the GPU and evaluate our method against standard quantized inference implementations (NVIDIA's Cutlass linear algebra library [26]) on various applications including fully connected networks for MNIST and reinforcement learning, and LSTMs/RNNs for language modeling and natural language inference. Across this range of applications and models we demonstrate significant end-to-end speedups over using standard quantized inference methods. We also extensively developed a CPU implementation, however we found that the lack of vectorized 1-bit operations (specifically, popcount), limited the memory boundedness of the operation, and yielded little speedup. We believe that future CPU hardware capabilities (and especially hardware accelerators) would enable these gains on the CPU, which we leave for future research.

In summary, our contributions are as follows:

- We develop *PrecisionBatching* an algorithm for quantized neural network inference targeted to traditional hardware platforms. *PrecisionBatching* enables quantized inference at lower bitwidths and achieves better speedup per accuracy over standard quantized inference without retraining.
- We evaluate *PrecisionBatching* over a variety of applications (MNIST, language modeling, natural language inference, reinforcement learning) and neural network architectures (fully connected, LSTM, RNN) and show net speedups of $> 10\times$ over the full precision baseline ($> 1.5\times$-$2\times$ over standard 8-bit quantized inference) within the same error tolerance. Furthermore, we leverage the finer granularity of precisions supported by *PrecisionBatching* to boost speed vs model quality.
- We show how using higher precision activations for quantized models as enabled by *PrecisionBatching* allows faster retraining times and achieves higher quality.
- We release optimized GPU kernels for our algorithm (and corresponding baselines) in the form of PyTorch modules.

II. Precision Batching

We describe the mechanics behind *PrecisionBatching* including how to decompose matrix-multiplies of various weight/activation bitwidths to be amenable for computation on traditional hardware platforms. Additionally, we describe how to efficiently implement our algorithm on standard hardware platforms and furthermore describe how we implemented our baseline standard quantized inference kernels.

A. *Precision Batching Quantized Inference*

PrecisionBatching decomposes weights and activations into 1-bit tensors and replaces the main matrix-vector multiplication operation with a sum of 1-bit matrix-matrix operations.

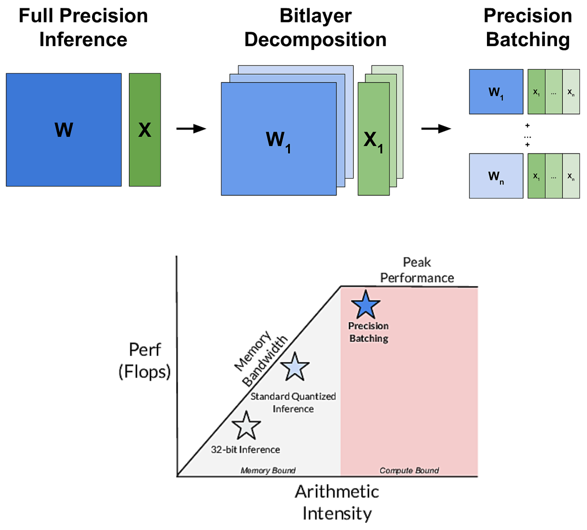


Fig. 1. *PrecisionBatching* quantized inference decomposes weights and activations into 1-bit tensors and re-frames full precision matrix-vector multiplication as a sum of binary matrix-matrix opeartions, increasing the arithmetic intensity of the operation, improving computational efficiency.

Figure 1 presents a diagram showing the core mechanism behind *PrecisionBatching*. The core operation of neural network inference with a batch size of 1 is matrix-vector multiplication.

$$L_i(x) = Wx$$

L_i represents the function that transforms activation input x at the specific layer of the neural network and W is the trained weights of the neural network at layer i. Assuming that $W > 0$ and $x > 0$, we can decompose W and x into a sum of bitlayers (binary tensors) as in fixed point format

$$W = \frac{1}{2^{16}}(2^{n-1}W_0^{(b)} + ... + 2^0 W_{n-1}^{(b)}) \text{ where } W_i^{(b)} \in [0, 1]$$

$$x = \frac{1}{2^{16}}(2^{k-1}x_1^{(b)} + ... + 2^0 x_k^{(b)}) \text{ where } x_i^{(b)} \in [0, 1]$$

In the decomposition above, n and k represent the precision at which weights and activations are quantized to, respectively. Making n and k larger provides more accurate approximations of W and x. n describes the precision at which W is estimated and represents the number of bitlayers to accumulate. The fraction $\frac{1}{2^{16}}$ represents the location of the fixed point and enables representation of values 16 binary digits < 1. The fixed point may be changed depending on the scale of values of the weights and activations. Substituting back into the first equation and rearranging we get

$$L_i(x) = Wx$$

$$= \frac{1}{2^{32}}(2^{n-1}W_0^{(b)} + ... + 2^0 W_{n-1}^{(b)})(2^k x_1^{(b)} + ... + 2^0 x_k^{(b)})$$

$$= \frac{1}{2^{32}} \sum_{i=0}^{n} 2^{n-i-1} W_i^{(b)} (2^k x_1^{(b)} + ... + 2^0 x_k^{(b)})$$

One key observation is that the terms of the sum above can be rewritten as a single matrix multiplication. The idea is to batch together the bitlayer decomposition of x into a single matrix and to frame the equation as a sum of matrix-matrix products.

$$\frac{1}{2^{32}} \sum_{i=0}^{n} 2^{n-i-1} (W_i^{(b)} [x_1^{(b)} ... x_k^{(b)}])[2^{k-1} ... 2^0]$$

The main workload $W_i^{(b)}[x_1^{(b)} ... x_k^{(b)}]$ exclusively consists of terms that are binary and facilitates efficient computation using 1-bit operations on CPU and GPU. Memory is reduced by a factor of approximately $\frac{32}{n}$, given that the matrix W dominates the majority of memory accesses. Note that the number of arithmetic ops is increased by a factor of k as separating out the sum induces more work. However, as the reformulation leverages batching, the cost of the extra compute is negated by the higher computational efficiency of the matrix-matrix multiplication, and the reduction in memory accesses yields a net speedup.

As indicated, by choosing n and k, any precision of weights and activations can be attained. In this paper k (activation precision) is set to either 8, 16 or 32. Note that higher activation precision does not linearly impact performance due to the increase in computational efficiency. However, for CPUs that are less efficient (more compute bound), setting k to be lower may significantly improve overall speed versus accuracy; hence k and n are parameters that determine the precision and speedup for quantized execution and may be tuned to the platform and requirement at hand. We analyze the impact of varying n and k on both speed and accuracy in the results.

Note that both the inputs and outputs of the *Precision-Batching* algorithm (as well as intermediate values such as partial sum accumulators) are full precision. The overhead of maintaining inputs and outputs as full precision is minimal as much of the computational and memory costs are attributed to large matrix multiply routines which are quantized (much of the memory costs are from loading the weights, rather than loading activations/inputs). Thus, keeping the intermediate inputs/activations in full precision is still aligned with the high level goal of speeding up inference.

B. Extending to Negative Values

We extend the formulation to any real valued W and x matrix (allowing negative values). Allowing any real valued input and matrix is important as it enables $PrecisionBatching$ to handle weights with negative values and cases where the input is not passed through a positive activation function (e.g: the first layer of the neural network whose inputs are real and may potentially contain negative values). The simple but effective idea is to leverage two's complement by adding an extra bitlayer with a negative scale to handle negative values.

$$W = \frac{1}{2^{16}}(-2^{n-1}W_0^{(b)} + ... + 2^0 W_n^{(b)}),\ W_i^{(b)} \in [0, 1]$$

$$x = \frac{1}{2^{16}}(-2^{k-1}x_0^{(b)} + ... + 2^0 x_k^{(b)}),\ x_i^{(b)} \in [0, 1]$$

Here, the first bitlayer for both x and W are negated, allowing for a complete representation of values between $[-2^n, 2^n-1]$. This formulation is logically equivalent to two's complement format. Note that this technique incurs an extra bitlayer of computational overhead (for weights) and thus increases the computational and memory costs; we found in practice that the extra bitlayer of computational overhead for activations is minimal.

C. Weight/Activation Quantization

In the *PrecisionBatching* formulation, W and x are converted into fixed point format and quantized to reduce computation and memory accesses. However, any standard post training quantization technique (e.g: KL divergence, MSE, etc) can be applied to W and x to improve accuracy, as long as the resulting set of quantization values are linearly spaced.

For applications, we use standard post training quantization before quantized execution.

$$Q(W) = d \times round\left(\frac{W}{d}\right), \, d = \frac{max(W) - min(W)}{2^n}$$

This rounds W to the corresponding closest n-bit representable fixed point values. We found that in practice, rounding produces significantly better results than truncation at very lower bitwidths (< 4 bits). Additionally, for quantizing to 1 bit, we found it extremely beneficial to exclude representing 0 and instead opt to represent a positive and negative value. After the n-bit rounding, $Q(W)$ is applied in the *PrecisionBatching* algorithm where the corresponding bitlayers and scales are deduced. Additionally, we also optimize over a clipping threshold to find a quantized matrix with the smallest mean error versus the full precision weight matrix. Note that quantizing W is a preprocessing step that is done offline and hence does not affect inference performance measurements.

The full *PrecisionBatching* algorithm is broken into two stages: a preprocessing step which converts full precision weights to bitlayers, listed in algorithm 1, and the inference stage which makes predictions given a full precision input, listed in algorithm 2.

D. Efficient Implementation

As indicated above, the core computation is an accumulation of products of binary tensors.

$$W_i^{(b)}[x_1^{(b)} ... x_k^{(b)}]$$

As all values are 0 or 1, memory is reduced by packing the 0s and 1s into the bits of an integer array, yielding $32\times$ reduction in memory for each product of bitlayers. Operating over these packed formats is inspired by standard binary quantized neural networks which uses logical operations and popcounts for implementing multiply accumulate. An important difference is that typical binary quantized neural network weights contain values that are -1 or 1 rather than 0 or 1. Hence, instead of the $xnor$ operation we use the and operation to simulate 1-bit multiplication. This is an important distinction for current and future hardware accelerators; current hardware accelerators (e.g: T4 binary MMA) perform the xnor operation rather than the and/or operation. Hence, in this work we are only capable of leveraging basic GPU ands/popcounts rather than the accelerator, though using an accelerator would yield much better performance improvements due to heightened compute speed vs memory speeds.

To leverage binary operations to compute over full precision values, the floating point input vector must be converted to fixed point and then packed in such a way to layout the bits to be conducive to the $and/popcount$ instruction. Conversion to fixed point is a simple multiply and cast. Rearranging the bits is done with a bitwise matrix transpose, for which there are efficient implementations on both CPUs and GPUs that leverage parallelism / SIMD. In practice, we found the bitwise matrix transpose to have negligible overhead. We furthermore note that multiple bitlayers may be stacked together so that the entire product across bitlayers can be performed with a single operation. However, in practice we found that there is negligible performance difference in accumulating multiple bitlayers separately, though a more optimized implementation may be the subject of analysis for future work.

E. Integer Quantized Inference

Standard quantized inference methods quantize both weight and activation to the same precision before execution (so that both operands are the same datatype); for example, 8-bit quantized execution quantizes both weights and activations to 8-bit ints before operation. Weights and activations are scaled down before quantization (so that the maximum value is representable in the quantized range), then dequantized after the operation. Like in *PrecisionBatching* we apply the same quantization preprocessing techniques (rounding, optimizing a clipping threshold) to weights before evaluation. In our experiments, we leverage NVIDIA's T4 tensorcore capability (via NVIDIA's Cutlass linear algebra library) in the implementation of the standard quantized inference baselines (1, 4, 8, 16, 32-bit inference methods).

III. Experimental Setup

This section describes the hardware we use to demonstrate *PrecisionBatching* in practice. We also describe the different neural network benchmark applications we use for evaluation.

A. Hardware Testbed

We perform all performance benchmarks and tests on NVIDIA's Tesla T4 GPU (as no previous GPU version supports 1/4/8 bit inference). For benchmarking kernels, we measure the wall-clock time of performing at least 1000 iterations of the target algorithm.

Note that the choice of using GPUs for *PrecisionBatching* is key: GPUs exhibit much higher compute vs memory capabilities than CPUs, which allows us to fully leverage *PrecisionBatching*'s higher operational intensity. Current CPUs exhibit a much lower compute vs memory ratio without vectorized popcount instructions and so on current generations of CPUs *PrecisionBatching* attains lower speedups, though with more

Algorithm 1 PrecisionBatching Preprocessing

```
Input
    $W$          Full precision weight matrix
    $n$          Number of bits to quantize
Output
    $W^{(b)}$    Bitlayers corresponding to quantized W
    $S$          Scales corresponding to quantized bitlayers
1: $W_q \longleftarrow Int(QuantizeRound(W, n) \times 2^{16})$
2: $max\_bit \longleftarrow max(log_2(|W_q|))$
3: $W^{(b)} \longleftarrow [W_q \wedge (1 \ll i)$ for $i$ in $max\_bit - n$ .. $max\_bit + 1]$
4: $S \longleftarrow [1 \ll i$ for $i$ in $max\_bit - n$ .. $max\_bit + 1]$
5: $S[0] \longleftarrow S[0] \times -1$
6: return $W^{(b)}$, $S$
```

Algorithm 2 PrecisionBatching Inference

```
Input
    $W^{(b)}$    Weight bitlayers
    $S$          Weight bitlayer scales
    $x$          Full precision input
Output
    $z$          Full precision prediction
1: $z \longleftarrow 0$
2: $x_q \longleftarrow Int(x \times 2^{16})$
3: for $W_b$, $scale$ in $W^{(b)}$, $S$ do
4:     $z \longleftarrow z + \frac{scale}{2^{32}} \times (W_b x_q)[-2^{31}\ 2^{30}\ ..\ 2^0]$
5: end for
6: return $z$
```

advanced CPU architectures supporting vectorized popcounts we expect to see the same improvements.

B. Software Implementation

Baseline 4, 8 and 16 bit standard quantized inference utilizes the NVIDIA Cutlass library which performs low-precision matrix multiply using WMMA (warp matrix multiply accumulate) hardware operations that leverage Tensorcores for compute. We implemented *PrecisionBatching* using standard CUDA (no tensorcore acceleration). In all experiments the batch dimension is one, as we are targetting application scenarios for inference, where examples are processed one at a time where latency is important (rather than throughput).

C. Neural Network Benchmarks

We evaluate our method on the following applications: MNIST, language modeling, natural language inference and reinforcement learning.

- **MNIST** we train a 3-layer fully connected neural network with a hidden size of 4096 for 20 epochs, reaching a baseline accuracy of 98%. We uniformly quantize the weights and activations of each layer to the target precisions.
- **Language Modeling** We train a model with a 1-layer 2048 unit LSTM [2] as the encoder, and a 1-layer 2048 unit fully connected as the decoder (a common architecture used in language modeling [27]). We apply dropout with a factor of .5 to the inputs of the encoder LSTM's recurrence, and to the encoder LSTM's output. We train the model on the Wikitext-2 dataset [28] for 40 epochs, reaching a baseline perplexity of 93. During evaluation of quantization on model accuracy, we quantize the LSTM's input and hidden layers to the same weight and activation levels; however, we keep the final fully connected decoder in full precision (as it is not the main runtime bottleneck).
- **Natural Language Inference** We train a model with a 1-layer 3072 unit LSTM encoder and a 3-layer 3072 unit fully connected decoder (a larger version of that seen in [29]). We train on the SNLI dataset [29] for 10 epochs and reach a baseline accuracy of 78%. During evaluation of quantization on model accuracy, we uniformly quantize both the weights and activations of the LSTM encoder and the fully connected decoder to the target precisions.
- **Reinforcement Learning** We train models on reacher hard (easy), cheetah run (medium) and humanoid stand (hard) [30] using D4PG [31] with a 3-layer 4096 unit neural network (same, but larger, architecture in [30]) until convergence (task difficulties from [31]). We train on features rather than pixels. During evaluation we quantize all layers of the policy. An episode is 1000 steps, (so maximum evaluation score is 1000, but this is not always attainable).

IV. Results

Our results section is organized as follows. Firstly, we evaluate the performance of the *PrecisionBatching* kernel in isolation to verify that our quantized inference method attains similar or better speedup than standard quantized inference even with higher activation precision. Secondly, we verify (across various tasks) that higher activation precision yields better model quality than when activation precision is the same as weight precision (the case when using standard quantized inference). Then, we evaluate the end-to-end speedup vs accuracy benefits of *PrecisionBatching* over standard quantized inference. Finally, we evaluate the benefits of higher precision activations motivating *PrecisionBatching* for quantization with retraining.

A. Precision Batching Kernel Performance

We implement optimized GPU kernels for the *PrecisionBatching* algorithm and measure the speedup of the kernel over the full precision (32-bit) operation (provided by NVIDIA's Cutlass linear algebra library) across multiple precisions and matrix sizes. Inference times include all activation processing steps necessary for the algorithm, for example, transposing the activation bitmatrix before 1-bit execution.

TABLE I

QUANTIZED INFERENCE SPEEDUPS OVER 32-BIT INFERENCE ACROSS DIFFERENT METHODS, MATRIX SIZES AND ACTIVATION QUANTIZATION LEVELS ON THE NVIDIA T4 GPU. PBATCH-N (A=K) MEANS N+1 BITLAYERS ARE ACCUMULATED WITH K-BIT ACTIVATIONS. (N-BIT WEIGHTS, K-BIT ACTIVATIONS). WE SEE THAT USING MORE ACTIVATIONS YIELDS ONLY MINOR SLOWDOWNS AND DEMONSTRATES THE MEMORY-BOUNDEDNESS OF LOW BATCHED INFERENCE.

Method	512x512	1024x1024	2048x2048	4096x4096
PBatch-1 (a=8)	10.8	13.8	12.0	13.6
PBatch-1 (a=16)	9.5	12.1	10.3	13.2
PBatch-1 (a=32)	8.0	10.7	8.0	10.7
PBatch-2 (a=8)	6.6	9.9	8.3	11.8
PBatch-2 (a=16)	6.8	8.8	7.1	10.9
PBatch-2 (a=32)	5.7	7.5	5.4	8.3
PBatch-4 (a=8)	4.9	6.5	5.1	7.3
PBatch-4 (a=16)	4.2	5.5	4.3	6.8
PBatch-4 (a=32)	3.6	4.8	3.4	5.3
PBatch-8 (a=8)	2.9	3.6	3.2	4.7
PBatch-8 (a=16)	2.5	3.2	2.5	4.0
PBatch-8 (a=32)	2.0	2.7	2.1	3.1
Int1	3.6	5.0	8.5	34.3
Int4	3.6	4.7	5.8	11.0
Int8	3.3	4.0	4.2	8.0
Float16	2.3	1.8	2.0	2.8
Float32	1	1	1	1

Table I shows the performance of the *PrecisionBatching* kernel with weight bits $\in (1, 2, 4, 8)$ and activation bits $\in (8, 16, 32)$, along with baseline quantizated inference kernels (Int1, Int4, Int8, Float16, Float32). We see that at fewer bits, the *PrecisionBatching* kernel achieves significant speedups over full precision inference: 10-14x speedup for 1-bit, 5-7x for 4-bit (note that the optimal speedup for PBatch-n is $\frac{32}{n+1}$ with the sign layer taken into account). Using fewer activation bits increases performance only slightly as compute is not the main bottleneck in these operations.

Generally, higher performance is seen at larger matrix sizes as the effect of the reduction in memory on performance is more pronounced. Baseline kernels (Int1, Int4, Int8 especially) perform much better at larger matrix sizes; we believe this is the case as their kernels are more optimized than ours and leverage Tensorcore capability for more efficient compute. This table is useful for roughly estimating the amount of speedup that may be attained on various applications. For example, if we believe for our application that inference with 4-bit weights, 16/32 bit activations would achieve the same accuracy as 8-bit weights and activations, then using *PrecisionBatching* on various matrix dimensions would yield a $1.5\times - 2\times$ speedup.

We additionally plot the operational intensity of *PrecisionBatching* versus standard inference in Figure 2. For each method, we compute its operational intensity: the number of operations that the method performs, divided by the number of bytes of memory required for the method. Note that for standard inference, an operation is dependent on its bitwidth, for example, for 32-bit inference, a single operation is a 32-bit multiply or add, while for 8-bit inference, a single operation is an 8-bit multiply or add.

For *PrecisionBatching*, each 1-bit operation counts towards the computational ops. As shown in Figure 2, *PrecisionBatching* achieves much higher operational intensity as the number of compute operations is increased by a factor of the specified activation precision. Standard inference on the other hand achieves low operational intensity. Thus, *PrecisionBatching* achieves much greater efficiency than standard inference, and combined with the better model accuracy obtained by operating over higher activations, achieves better performance per accuracy threshold.

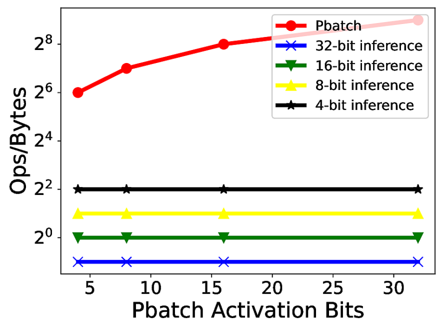


Fig. 2. Operational intensity of *PrecisionBatching* versus standard inference. *PrecisionBatching* achieves higher operational intensity with more activation bits, enabling it to operate more efficiently on GPUs. Standard inference is primarily memory bound and achieves low operational intensity, resulting in lower compute efficiency.

B. Accuracy Benefits of Higher Precision Activations

Next we show that using higher precision for activations leads to significantly better model accuracy at low bitwidths. We benchmark model accuracy across: MNIST, language modeling, natural language inference and reinforcement learning. For each we train one baseline full precision model and evaluate the effects of various levels of weight and activation quantization on the model's end performance. For each model/application we quantize weights and activations to 1, 4, 8, 16 and 32 bits. Note the $Q_{activ.} = Q_{weight}$ column uses standard quantized inference while the other columns use *PrecisionBatching*.

Table II shows model performance (accuracy for MNIST and natural language inference, perplexity for language modeling, reward for reinforcement learning) for different weight and activation precisions. For weight bitlevels < 8, keeping activations at higher precision (8, 16 or 32 bit) greatly increases model accuracy; generally, keeping activations at a higher precision allows quantizing twice as many bits, from 8-bits to 4-bits, without significant loss in model accuracy.

For MNIST, with 1-bit weights, using higher precision activations is the difference between 85% accuracy and random guessing (10% accuracy); with 4-bit weights, higher precision activations maintains within $<$ 1% of the full precision model's performance. Similarly, for language modeling, with 1-bit weights, higher precision activations reduces perplexity from 800 to 180; for 4-bit weights, higher precision activations reduce perplexity from 180 to within a few points of the full precision performance. For natural language inference,

TABLE II
Benefits of using more precision for activations on model quality, evaluated on MNIST, language modeling (Wikitext-2), natural language inference (SNLI) and reinforcement learning (reacher:hard, cheetah:run, humanoid:stand). Generally, using higher precision activations allows quantizing twice as many bits with little degradation of model accuracy. Best scores per weight precision is bolded.

Task		$Q_{activ.} = 32$	$Q_{activ.} = 16$	$Q_{activ.} = 8$	$Q_{activ.} = Q_{weight}$
MNIST (acc.)	$Q_{weight} = 1$	85.8	86.7	**87**	10.1
	$Q_{weight} = 4$	97.1	**97.3**	**97.3**	94.3
	$Q_{weight} = 8$	**98.0**	97.8	97.8	**98.0**
	$Q_{weight} = 16$	**98.0**	97.9	97.9	**98.0**
	$Q_{weight} = 32$	-	-	-	98.0
Language Modeling (ppl.)	$Q_{weight} = 1$	**188.0**	188.0	188.0	828.1
	$Q_{weight} = 4$	**94.3**	**94.3**	**94.3**	148.9
	$Q_{weight} = 8$	**94.0**	**94.0**	**94.0**	**94.0**
	$Q_{weight} = 16$	**91.7**	**91.7**	**91.7**	92.8
	$Q_{weight} = 32$	-	-	-	92.8
Natural Language Inference (acc.)	$Q_{weight} = 1$	**76.1**	**76.1**	74.0	32.8
	$Q_{weight} = 4$	**78.7**	**78.7**	76.8	77.4
	$Q_{weight} = 8$	78.9	78.9	76.9	**79.1**
	$Q_{weight} = 16$	**78.9**	**78.9**	76.9	78.8
	$Q_{weight} = 32$	-	-	-	78.8
Reacher Hard (rew.)	$Q_{weight} = 1$	9.8	6.6	9.5	**24.9**
	$Q_{weight} = 4$	676.4	765.1	**960.1**	826.2
	$Q_{weight} = 8$	969.1	973.8	962.5	**976.4**
	$Q_{weight} = 16$	960.0	**974.6**	966.5	957.3
	$Q_{weight} = 32$	-	-	-	968.0
Cheetah Run (rew.)	$Q_{weight} = 1$	.8	**0.9**	0.8	0.0
	$Q_{weight} = 4$	616.3	**685.6**	651.6	480.3
	$Q_{weight} = 8$	700.3	**701.0**	681.6	700.7
	$Q_{weight} = 16$	706.1	**707.9**	682.4	705.4
	$Q_{weight} = 32$	-	-	-	702.9
Humanoid Stand (rew.)	$Q_{weight} = 1$	5.0	4.9	**7.4**	4.7
	$Q_{weight} = 4$	443.0	410.6	**466.9**	25.7
	$Q_{weight} = 8$	753.3	692.7	**816.0**	789.4
	$Q_{weight} = 16$	824.1	781.2	**864.2**	798.9
	$Q_{weight} = 32$	-	-	-	808.1

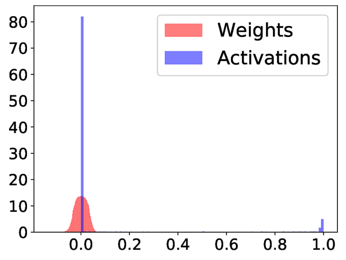
using full precision activations allows us to quantize down to 1-bit with only a couple percentages of accuracy degredation (78% to 76%), whereas quantizing activations to 1-bit degrades to random guessing (33%). Interestingly, for language inference, the 8-bit quantized model outperformed the full precision result, a known phenomenon seen in quantization literature [11], [14]. For reinforcement learning, trends are generally similar: better reward is attained with higher activation precision, though in some interesting cases (e.g: reacher hard), lower activation precision performed better. Harder tasks (e.g: humanoid stand) are generally more difficult to quantize and higher activations typically yield more consistent reward gains on such tasks. Additionally, on some tasks, the score achieved by weights=activations is dissimilar to that reported in the $Q_{activ.} = Q_{weight}$ column; this is due to slight differences in implementation between *PrecisionBatching* and standard quantized inference (e.g: we dynamically scale activations in standard quantized inference, whereas for *PrecisionBatching* a static scale is used).

Additionally, Figure 3 shows the histogram of values of both weights and activations on the MNIST task for each layer of the network. As shown, across all layers of the network, activation values have a much wider spread than the weights, with the exception of the first layer, for which the activations are the mnist input features. This indicates that quantizing activations would yield a significantly higher quantization error than for weights, and motivates keeping activations in higher precision.

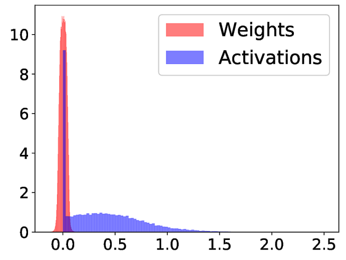
C. End to End Performance Gains

We demonstrate the end to end speedups achieved by *PrecisionBatching*. We combine the observations from our previous results: we leverage the high runtime performance of the *PrecisionBatching* kernel and the better model accuracy of keeping activations in higher precision to attain significant end-to-end speedups over the full precision model while maintaining model quality. We use the same applications (MNIST, language modeling (Wikitext-2), natural language inference (SNLI), reinforcement learning) with the same model architectures and training parameters described previously.

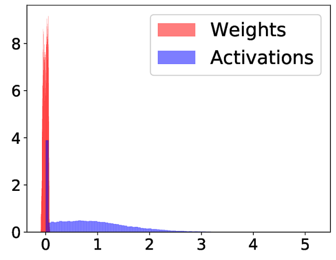
We apply each target quantized inference algorithm as follows. For the MNIST/reinforcement learning model, we replace each linear layer with the corresponding quantized inference algorithm; for the language modeling and natural language inference Seq2Seq model, we replace each linear layer of the encoder 1-layer LSTM with the target quantized inference algorithm, however we keep the final fully connected decoder in full precision as it is not the main runtime bottle-



(a) Layer 1



(b) Layer 2



(c) Layer 3

Fig. 3. Histogram of weights and activations of a 3 layer neural network on the MNIST task. Note that the plotted activations are the inputs to the corresponding matrix multiply operation with the weights. Besides the first layer (image input = activations), activations have a significantly wider distribution of values than weights, thus quantizing activations incurs more error and motivates using higher precision for activations.

neck. For reinforcement learning, we replace each layer of the policy with the target quantized inference algorithm.

Additionally, for both baseline quantized inference and *PrecisionBatching*, on MNIST, language modeling and natural language inference, we use variable-bit quantization on different layers (e.g: 1-bit quantization on layer 1, 4-bit quantization on layer 2, etc) to further boost performance per accuracy. Accordingly, we perform an exhaustive grid search over weight/activation precision assignments. On the 3-layer fully connected for MNIST, for baseline quantized inference we assign each layer a precision $\in (1, 4, 8, 16, 32)$ (note that for quantized inference activations are the same precision as weights); for *PrecisionBatching*, we assign each layer a precision $\in (1, 2, 3, 4, 8)$ and activations $\in (8, 16, 32)$. On the Seq2Seq LSTM for language modeling and natural language inference, for baseline quantized inference we assign each layer a precision $\in (1, 4, 8, 16)$; for *PrecisionBatching*, we assign each layer a precision $\in (1, 2, 4, 8)$ and activations $\in (8, 16, 32)$. For the reinforcement learning tasks, we opt instead to maintain each layer with the same weights/activation precision; however, we leverage *PrecisionBatching*'s finer precision granularity in the evaluation (weight precision $\in (1, 2, 3, 4, 5, 6, 7, 8)$). In benchmarking the runtime performance of each model/application, we measure the wall clock time of inference with a batch size of 1 for 10 iterations on a given input repeated over 10 runs and take the minimum. We measure speedups by comparing the model with the target quantized inference algorithm against the model with the baseline quantized inference method.

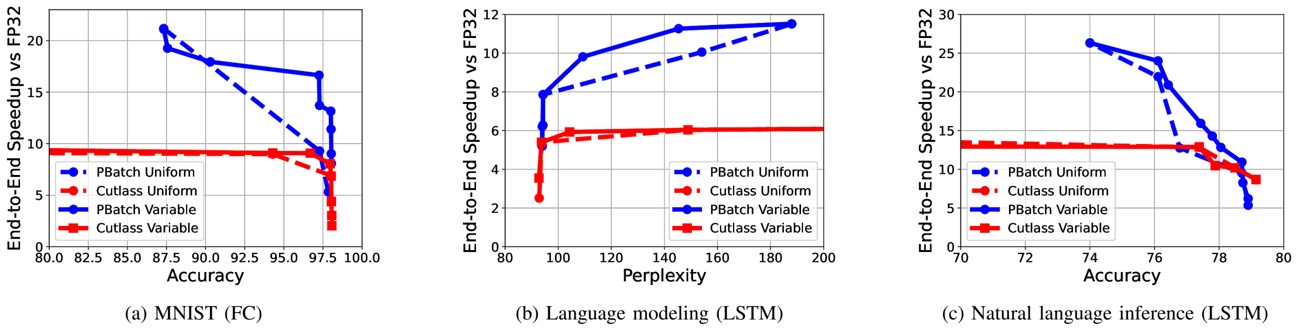
Figure 4 and Figure 5 shows the Pareto curves of the end-to-end speedups of *PrecisionBatching* over standard quantized inference. On average, *PrecisionBatching* yields speedups of $8\times$ - $10\times$ over full precision inference, and $1.5\times$ - $2\times$ over standard 8-bit quantized inference at the same error tolerance. Additionally, the finer granularity of precision supported by *PrecisionBatching* enables greater speedup per accuracy when using variable quantization across layers. The same data is reflected in Table III, which shows the corresponding best achieved speedup for each method for different error margins.

D. *Retraining Benefits of Higher Activation Precision*

We show that higher activation benefits the retraining process, leading to better convergence and accuracy. Retraining generally improves the quantized model's quality at lower bitwidths and works by training the model to account for quantization error. As standard quantized inference methods require weights and activations be the same precision, the model must be retrained with same precision weights and activations. However, this often makes retraining slow and quality frequently falls short of the corresponding full precision baseline at low bitwidths. We show that higher activation precision, as *PrecisionBatching* enables, facilitates retraining, thus leading to better final quality and faster convergence.

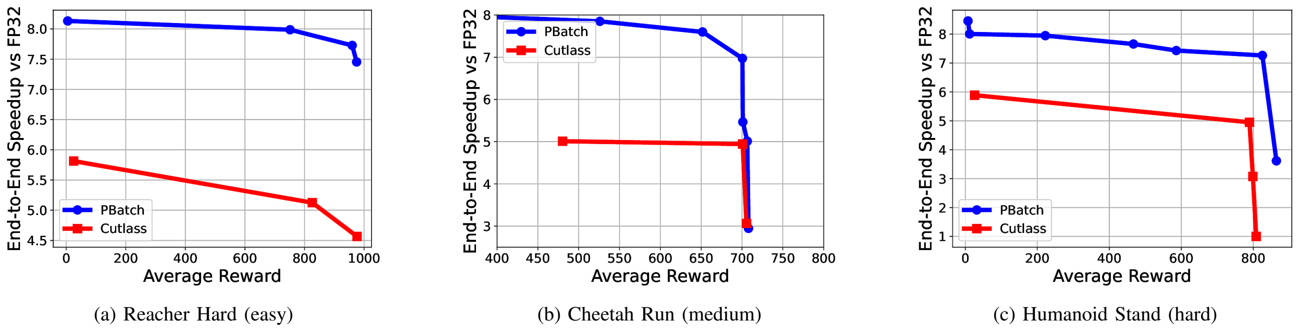
We perform retraining from scratch and from pretrained model on both MNIST and language modeling with quantization aware training, the standard method to retraining for a quantized model [32]. Note we do not perform any modifications to training (e.g: architectural changes to the network to assist better performance). We train MNIST for 100 epochs and the language model for 40 epochs using the same hyperparameters as described in previous sections. During retraining, quantization aware training is performed immediately from the very beginning (no quantization delay).

Figure 6 shows the results of retraining on 1 bit and 4 bit precision weights for MNIST and language modeling. Particularly for 1 bit weights, retraining with 32 bit (full precision) activations enables faster and better convergence. For MNIST, 1 bit weights and activations is stuck at 80% accuracy whereas 1 bit weights, 32 bit activations achieves near full precision performance. For language modeling, 1 bit weights and activations converges at a worse quality (300 ppl), while 1 bit weights and 32 bit activations achieves much better quality (190 ppl). Table IV, compares scores achieved with no retraining, retraining from scratch and retraining from pretrained model (finetuning) and demonstrates that higher precision activations consistently achieves better scores across the tasks in all regimes.



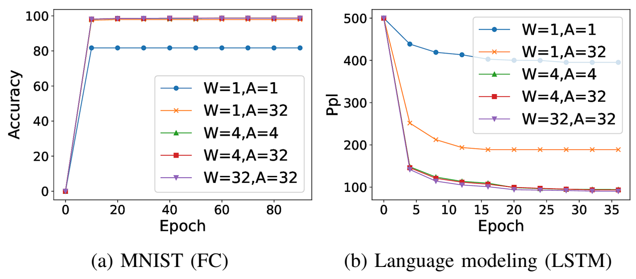
(a) MNIST (FC) (b) Language modeling (LSTM) (c) Natural language inference (LSTM)

Fig. 4. End-to-end speedup over full precision model vs model quality on MNIST, language modeling and natural language inference over a grid search of precisions for weights and activations. Standard quantized inference is limited to weights=activations $\in (1, 4, 8, 16, 32)$. *PrecisionBatching* leverages execution with finer granularity inference with weights $\in (1, 2, 4, 8)$, activations $\in (8, 16, 32)$. Dotted lines show Pareto boundary where layers have the same precision, while solid lines indicate layers may have different precision. *PrecisionBatching* attains significant speedups over standard quantized inference methods.



(a) Reacher Hard (easy) (b) Cheetah Run (medium) (c) Humanoid Stand (hard)

Fig. 5. End-to-end speedup over full precision model vs model quality on reinforcement learning tasks (Deepmind Control Suite; difficulty from [31]). All layers have the same precision/activations. Standard quantized inference is limited to weights=activations $\in (1, 4, 8, 16, 32)$. *PrecisionBatching* leverages execution with finer granularity inference with weights $\in (1, 2, 3, 4, 5, 6, 7, 8)$, activations $\in (8, 16, 32)$. *PrecisionBatching* attains significant speedups over standard quantized inference methods.



(a) MNIST (FC) (b) Language modeling (LSTM)

Fig. 6. Benefits of higher precision activations for retraining quantized models. Higher activations (A=32) achieves better convergence and quality after retraining. This is especially true for 1 bit weights. W=32,A=32 is the full precision baseline.

E. Larger Batch Sizes

PrecisionBatching favors low batch sizes (batch 1 is best, a matrix-vector multiplication) to leverage the weight boundedness of the problem. With larger batch sizes the technique sees significantly less gains, and may even incur a slowdown, as larger batch sizes are more compute and activation bound.

The significance of this limitation means that *PrecisionBatching* is primarily targeted for the linear components of a network (that have low batch dimension), which limits application of the algorithm primarily to fully connected neural networks, RNNs and LSTMs particularly for inference with low batch sizes where latency is important. For this reason, convolutions, which may be framed as a matrix-matrix multiply will see less speedup using *PrecisionBatching*.

However, despite these shortcomings, we argue that speeding up low batched fully connected layers of a network is a significant contribution as many real world applications deploy such networks in practice. For example, OpenAI Five [33] employs a 4096 layer LSTM and inference during game play processes frame by frame; Google's on-device keyword prediction model [34] similarly employs an LSTM and inference (not training) is performed sentence by sentence to minimize latency; likewise, Waymo's ChauffeurNet model [35] consists of large LSTM and RNN components which perform inference per environment step. We believe *PrecisionBatching* is a step towards fully utilizing the parallel compute capabilities of traditional hardware systems and will be useful in many high performance machine learning use cases.

TABLE III
BEST MODEL QUALITY, SPEEDUPS AND PRECISION ASSIGNMENTS PER ERROR MARGIN FOR *PrecisionBatching* AND BASELINES OVER A GRID SEARCH OF PRECISIONS FOR WEIGHTS AND ACTIVATIONS. *PrecisionBatching* PRECISION ASSIGNMENTS FORMAT: (L_i BITS, A_i BITS); QUANTIZED INFERENCE PRECISION ASSIGNMENTS FORMAT: (L_i=A_i BITS). STANDARD QUANTIZED INFERENCE IS LIMITED TO WEIGHTS=ACTIVATIONS $\in (1, 4, 8, 16, 32)$. *PrecisionBatching* LEVERAGES EXECUTION WITH FINER GRANULARITY INFERENCE WITH WEIGHTS $\in (1, 2, 4, 8)$, ACTIVATIONS $\in (8, 16, 32)$. FOR REINFORCEMENT LEARNING, ALL LAYERS HAVE THE SAME PRECISION (THOUGH FOR *PrecisionBatching* ACTIVATION PRECISION IS NOT NECESSARILY THE SAME AS WEIGHT PRECISION). *PrecisionBatching* YIELDS $1.5\times - 2\times$ SPEEDUP OVER STANDARD QUANTIZED INFERENCE.

Task	Error	Quality	Speedup vs FP32	Speedup vs Int8	Method	Precision Assign.
MNIST (acc.)	$< 1\%$	97.3%	16.6	2.4	PBatch	(4,8)(1,8)(1,8)
		97.9%	8.0	1.2	Baseline	(8,4,8)
	$< 5\%$	97.3%	16.6	2.4	PBatch	(4,8)(1,8)(1,8)
		94.3%	9.1	1.3	Baseline	(4,4,4)
	$< 15\%$	87.3%	21.0	3.1	PBatch	(1,8)(1,8)(1,8)
		94.3%	9.1	1.3	Baseline	(4,4,4)
Language Modeling (ppl.)	< 5	94.3	7.9	1.5	PBatch	(4,8)(4,8)
		93.7	5.4	1	Baseline	(8,8)
	< 25	109.3	9.8	1.8	PBatch	(1,8)(4,8)
		104.3	5.9	1.1	Baseline	(4)(8)
	< 50	145.3	11.3	2.1	PBatch	(1,8)(2,8)
		148.9	6.0	1.2	Baseline	(4,4)
Natural Language Inference (acc.)	$< 1\%$	77.8	14.3	1.6	PBatch	(4,16)(1,8)
		77.9	10.5	1.2	Baseline	(4,8)
	$< 5\%$	74.0	26.3	3.0	PBatch	(1,8)(1,8)
		77.4	12.9	1.5	Baseline	(4,4)
	$< 15\%$	74.0	26.3	3.0	PBatch	(1,8)(1,8)
		77.4	12.9	1.5	Baseline	(4,4)
Reacher Hard (rew.)	< 50	975.0	7.7	1.69	PBatch	(4,8)(4,8)(4,8)
		976.4	4.5	1	Baseline	(8,8,8)
Cheetah Run (rew.)	< 50	651.6	7.6	1.53	PBatch	(4,8)(4,8)(4,8)
		700.7	5.0	1	Baseline	(8,8,8)
Humanoid Stand (rew.)	< 50	825.3	7.2	1.47	PBatch	(6,8)(6,8)(6,8)
		789.5	4.95	1	Baseline	(8,8,8)

TABLE IV
FINAL SCORES ACHIEVED BY QUANTIZED MODELS WITH ≤ 4 BIT WEIGHTS, WITH AND WITHOUT RETRAINING. USING HIGHER PRECISION ACTIVATIONS, AS *PrecisionBatching* ENABLES, CONSISTENTLY ACHIEVE HIGHER QUALITY, EVEN WITH RETRAINING.

Task	Full Precision Score	Weight Bits	No Retrain		Retrain (scratch)		Retrain (finetune)	
			W=A	W=32	W=A	W=32	W=A	W=32
MNIST (acc.)	98.5	W=1	10.1	85.8	81.7	97.9	80.0	**98.2**
		W=4	94.3	97.1	**98.6**	**98.6**		
Language Modeling (ppl.)	90.1	W=1	828.1	188.0	396.0	190.7	316.5	**142.4**
		W=4	148.9	94.3	94.0	**93.2**		

V. Related Work

A. Post Training Quantization

Post training quantization is the standard method for quantizing neural networks without retraining and involves clipping the values of a pre-trained model based on statistics [17]. Various methods for post training quantization have been researched. Naively, post training quantization involves casting weight and activation values to the nearest n-bit representation. More sophisticated techniques involve clipping the weights and activations so as to minimize some form of error between the quantized and real values [16], [17]. Even more advanced techniques change the underlying floating point format to enhance speed/accuracy [36]–[38].

Pre-existing research in post training quantization methods often omit details as to how the resulting quantized weights/activations may be leveraged on existing CPU and GPU platforms to speed up inference. More unusual bitwidths (e.g: 2/3/4/5) lack a corresponding data type on traditional hardware platforms and hence it is unclear how these levels of quantization improve inference. The implied benefit of post training quantization methods on these bitwidths is either space/memory savings or deployment to specially developed hardware accelerators for which fixed point operations for various bitwidths may be developed. By framing n-bit fixed point inference operations as a sum of binary operations, *PrecisionBatching* is an effective solution to realize these quantization gains on traditional hardware platforms. Hence, *PrecisionBatching* extends the memory-savings benefits of various post training quantization methods to speed gains on traditional hardware architectures.

B. PACT

The importance of activations in quantization quality has been noted in research. Specifically, PACT (Parameterized Clipping Activation for Quantized Neural Networks) [15] demonstrated that neural network weights and activations may be quantized to very low bitwidths (< 4) if an activation scale is optimized during training. Although PACT requires changes to the training process (and hence does not work out of the box), their research demonstrates the importance and difficulty of quantizing activations in maintaining quantization quality. Motivated by their findings, *PrecisionBatching* opts to keep activations in higher precision (8,16,32 bit) to maintain accuracy at very low quantization level. This comes at minimal cost during inference as compute is dominated by memory access times. Thus, *PrecisionBatching* circumvents the need to maintain a quantization scale at training time by giving more bits of precision to activations at inference time.

C. Outlier Channel Splitting

Recently, research into quantization without retraining has emerged as a topic of interest. One notable work is Outlier Channel Splitting [17], which eliminates large magnitude weights/activations (which increase quantization error) by splitting them into separate channels, then applying standard post training quantization on the splitted weights, improving quantization performance. Outlier Channel Splitting demonstrates better performance-per-bit by using their technique in conjunction with standard post training quantization methods. Importantly, the authors note that outlier channel splitting may also be done to activations at runtime, though this is computationally difficult as it requires repeatedly finding the maximum of a matrix and adding rows to it. *PrecisionBatching* eliminates this need by using more bits to represent activations, improving accuracy. Like many standard post-training quantization methods, Outlier Channel Splitting may be applied along with *PrecisionBatching* to improve quantization quality and to extend their memory-saving gains to speed gains on traditional hardware platforms.

D. Bitserial Computation

Bitserial computation is a technique leveraged by *PrecisionBatching* for quantized inference and operates by decomposing fixed point operations into bitwise operations [24], [25], [39]. Bitserial computation frames n-bit fixed point operations as a sum of bitwise operations and accumulates the result layer by layer. This formulation is most popular in the hardware architecture space to develop specialized accelerators for machine learning and realizing the technique in this context requires dedicated hardware constructs. Various hardware accelerators that leverage the bitserial formulation to reduce energy costs include [24], [25], [39]. The bitserial formulation is less used in context of traditional hardware architectures (e.g: CPUs and GPUs) as it is less clear how the technique would perform as it greatly increases the total amount of arithmetic per operation, though early works such as [18], [40] explore CPU implementations. The key insight behind *PrecisionBatching* is that on traditional architectures, particularly the GPU, low batched inference is heavily memory bound and by batching the decomposed 1-bit vectors the extra overhead in compute is negated by the reduction in memory accesses, effectively turning a memory bound problem into a compute bound problem, and yielding a net speedup.

E. Streamlined Deployment for Quantized Neural Networks

Another related work to *PrecisionBatching* is Streamlined Deployment for Quantized Neural Networks [40], which leverages a bitserial formulation to speed up deployment on the CPU. Similar to *PrecisionBatching*, Streamlined Deployment for Quantized Neural Networks frames quantized operations in terms of 1-bit operations. However, the key difference is that Streamlined Deployment separates the the bitlayers of the activations into different product terms, rather than batching them into one large matrix multiplication. As shown in their paper, the impact is that both weights and activations must be kept in very low precision (e.g: 2-bit activations) due to the computational overhead of performing multiple matrix products, which naturally leads to significant degredation in accuracy. The key observation of *PrecisionBatching* is that activation bitlayers may be batched together into one single matrix and a single large matrix product may be performed over this batch at high efficiency. This allows quantized inference with activations at or near full precision with minimal computational overhead, enhancing quantization performance.

F. Automatic Generation of Quantized Machine Learning Kernels

Automatic Generation of High-Performance Quantized Machine Learning Kernels [41] leverages a similar bitserial decomposition of kernels as PrecisionBatching to automatically generate quantized kernels for machine learning applications. In their work, [41] build a compiler to generate quantized kernels and demonstrate a speedup on CPU hardware platforms. Our work on *PrecisionBatching* differentiates in several key respects. Firstly, we show that our quantized inference kernel can perform inference with higher precision activations and attain significant speedup-vs-accuracy benefits; the work in [41] do not explore the impacts of higher precision activation on model accuracy. Secondly, our method utilizes the GPU, while the work in [41] demonstrates their kernel on the CPU, so we attain better speedups as the GPU is more compute bound. Finally, our work leverages the observation that inference is memory bound to develop a more effective quantized inference kernel, while the work in [41] is primarily focused on building a compiler for kernel generation.

VI. Conclusion

We present *PrecisionBatching*, a quantized inference algorithm for speeding up neural network execution on traditional hardware platforms at low weight bitwidths. *PrecisionBatching* leverages the compute efficiency of traditional hardware platforms (e.g: GPUs) to perform inference with higher activation precisions, enabling execution with lower precision weight layers, achieving a net speedup. Across various models (fully connected, LSTMs, RNNs) and applications (MNIST, language modeling, natural language inference, reinforcement learning) we show that *PrecisionBatching* yields end-to-end speedups of over $8\times$ that of full precision inference ($1.5\times$ – $2\times$ that of standard 8-bit quantized inference) at the same error tolerance.

References

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2012, pp. 1097–1105.

[2] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997. [Online]. Available: https://doi.org/10.1162/neco.1997.9.8.1735

[3] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," 2014.

[4] A. Graves, A. rahman Mohamed, and G. Hinton, "Speech recognition with deep recurrent neural networks," 2013.

[5] Y. Bengio, R. Ducharme, P. Vincent, and C. Janvin, "A neural probabilistic language model," *J. Mach. Learn. Res.*, vol. 3, no. null, p. 1137–1155, Mar. 2003.

[6] P. Rajpurkar, J. Zhang, K. Lopyrev, and P. Liang, "SQuAD: 100,000+ questions for machine comprehension of text," in *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*. Austin, Texas: Association for Computational Linguistics, Nov. 2016, pp. 2383–2392. [Online]. Available: https://www.aclweb.org/anthology/D16-1264

[7] S. Wang and J. Jiang, "Learning natural language inference with lstm," 2015.

[8] T. Dao, A. Gu, M. Eichhorn, A. Rudra, and C. Ré, "Learning fast algorithms for linear transforms using butterfly factorizations," in *Proceedings of the 17th International Conference on Machine Learning (ICML 2019)*, 2019.

[9] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks," in *Advances in Neural Information Processing Systems 29*, D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, Eds. Curran Associates, Inc., 2016, pp. 4107–4115.

[10] ——, "Quantized neural networks: Training neural networks with low precision weights and activations," *J. Mach. Learn. Res.*, vol. 18, no. 1, p. 6869–6898, Jan. 2017.

[11] C. Xu, J. Yao, Z. Lin, W. Ou, Y. Cao, Z. Wang, and H. Zha, "Alternating multi-bit quantization for recurrent neural networks," *International Conference on Learning Representations (ICLR)*, 2018.

[12] V. Sze, Y.-H. Chen, T.-J. Yang, and J. Emer, "Efficient processing of deep neural networks: A tutorial and survey," 2017.

[13] Z. Lin, M. Courbariaux, R. Memisevic, and Y. Bengio, "Neural networks with few multiplications," *International Conference on Learning Representations (ICLR)*, 2015.

[14] S. Krishnan, S. Chitlangia, M. Lam, Z. Wan, A. Faust, and V. J. Reddi, "Quantized reinforcement learning (quarl)," 2019.

[15] J. Choi, Z. Wang, S. Venkataramani, P. I.-J. Chuang, V. Srinivasan, and K. Gopalakrishnan, "Pact: Parameterized clipping activation for quantized neural networks," 2018.

[16] H. Wu, "Nvidia quantization deck," 2018. [Online]. Available: https://developer.download.nvidia.com/video/gputechconf/gtc/2019/presentation/s9659-inference-at-reduced-precision-on-gpus.pdf

[17] R. Zhao, Y. Hu, J. Dotzel, C. D. Sa, and Z. Zhang, "Improving neural network quantization without retraining using outlier channel splitting," in *Proceedings of the 17th International Conference on Machine Learning (ICML 2019)*, 2019.

[18] M. Cowan, T. Moreau, T. Chen, and L. Ceze, "Automating generation of low precision deep learning operators," 2018.

[19] S. Zhou, Y. Wu, Z. Ni, X. Zhou, H. Wen, and Y. Zou, "Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients," 2016.

[20] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "Eie: Efficient inference engine on compressed deep neural network," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016.

[21] Baidu, "Baidu deepbench," 2017. [Online]. Available: https://github.com/baidu-research/DeepBench

[22] Groq, "Groq tsp leads in inference performance," 2020. [Online]. Available: https://groq.com/groq-tsp-leads-in-inference-performance/

[23] U. Gupta, S. Hsia, V. Saraph, X. Wang, B. Reagen, G.-Y. Wei, H.-H. S. Lee, D. Brooks, and C.-J. Wu, "Deeprecsys: A system for optimizing end-to-end at-scale neural recommendation inference," 2020.

[24] J. Albericio, A. Delmás, P. Judd, S. Sharify, G. O'Leary, R. Genov, and A. Moshovos, "Bit-pragmatic deep neural network computing," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-50 '17. New York, NY, USA: ACM, 2017, pp. 382–394. [Online]. Available: http://doi.acm.org/10.1145/3123939.3123982

[25] H. Sharma, J. Park, N. Suda, L. Lai, B. Chau, J. K. Kim, V. Chandra, and H. Esmaeilzadeh, "Bit fusion: Bit-level dynamically composable architecture for accelerating deep neural networks," 2017.

[26] NVIDIA, "Cutlass linear algebra library," 2018. [Online]. Available: https://github.com/NVIDIA/cutlass

[27] G. Melis, C. Dyer, and P. Blunsom, "On the state of the art of evaluation in neural language models," *International Conference on Learning Representations (ICLR)*, 2017.

[28] S. Merity, C. Xiong, J. Bradbury, and R. Socher, "Pointer sentinel mixture models," *International Conference on Learning Representations (ICLR)*, 2016.

[29] S. R. Bowman, G. Angeli, C. Potts, and C. D. Manning, "A large annotated corpus for learning natural language inference," in *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, 2015.

[30] Y. Tassa, Y. Doron, A. Muldal, T. Erez, Y. Li, D. d. L. Casas, D. Budden, A. Abdolmaleki, J. Merel, A. Lefrancq *et al.*, "Deepmind control suite," *arXiv preprint arXiv:1801.00690*, 2018.

[31] M. Hoffman, B. Shahriari, J. Aslanides, G. Barth-Maron, F. Behbahani, T. Norman, A. Abdolmaleki, A. Cassirer, F. Yang, K. Baumli *et al.*, "Acme: A research framework for distributed reinforcement learning," *arXiv preprint arXiv:2006.00979*, 2020.

[32] Google, "Quantization aware training with tensorflow," 2020. [Online]. Available: https://www.tensorflow.org/model_optimization/guide/quantization/training

[33] C. Berner, G. Brockman, B. Chan, and V. C. et al., "Dota 2 with large scale deep reinforcement learning," 2019.

[34] A. Hard, C. M. Kiddon, D. Ramage, F. Beaufays, H. Eichner, K. Rao, R. Mathews, and S. Augenstein, "Federated learning for mobile keyboard prediction," 2018. [Online]. Available: https://arxiv.org/abs/1811.03604

[35] M. Bansal, A. Krizhevsky, and A. Ogale, "Chauffeurnet: Learning to drive by imitating the best and synthesizing the worst," 2018.

[36] T. Tambe, E.-Y. Yang, Z. Wan, Y. Deng, V. J. Reddi, A. Rush, D. Brooks, and G.-Y. Wei, "Adaptivfloat: A floating-point based data type for resilient deep learning inference," *arXiv preprint arXiv:1909.13271*, 2019.

[37] E. H. Lee, D. Miyashita, E. Chai, B. Murmann, and S. S. Wong, "Lognet: Energy-efficient neural networks using logarithmic computation," in *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, mar 2017.

[38] J. Johnson, "Rethinking floating point for deep learning," 2018.

[39] P. Judd, J. Albericio, T. Hetherington, T. M. Aamodt, and A. Moshovos, "Stripes: Bit-serial deep neural network computing," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct 2016, pp. 1–12.

[40] Y. Umuroglu and M. Jahre, "Streamlined deployment for quantized neural networks," 2017.

[41] M. Cowan, T. Moreau, T. Chen, J. Bornholt, and L. Ceze, "Automatic generation of high-performance quantized machine learning kernels," in *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, ser. CGO 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 305–316. [Online]. Available: https://doi.org/10.1145/3368826.3377912

Appendix

A. Abstract

This is the artifact evaluation instructions for *PrecisionBatching: Bitserial Decomposition for Efficient Neural Network Inference on GPUs*. Our artifact includes code to install the PrecisionBatching and Cutlass kernels used in the paper, as well as to reproduce the following core results of the paper: PrecisionBatching vs standard quantized inference speedups, end to end speedups for MNIST, language modeling, and natural language inference.

B. Artifact check-list (meta-information)

- **Algorithm:** PrecisionBatching: Bitserial Decomposition for Efficient Neural Network Inference on GPUs.
- **Compilation:** GCC, G++, NVIDIA compiler.
- **Model:** Models are included in repository; no action needed here.
- **Data set:** Dataset download is coded in; no action needed here.
- **Run-time environment:** Linux, CUDA 10.2.
- **Hardware:** NVIDIA T4 GPU.
- **Run-time state:** We provide install instructions using conda + pip (requirement files in the repo).
- **Execution:** Execution will approximately take at most 3 hours.
- **Metrics:** We measure evaluation time (e.g: speedup) and model quality.
- **Output:** Output includes pdfs showing end to end speedups, as well as raw txt files containing the speedup/accuracy values. Expected results are included.
- **Experiments:** See README which should provide exact step by step instructions for installation + running scripts.
- **How much disk space required (approximately)?:** 50 GB max.
- **How much time is needed to prepare workflow (approximately)?:** 1.5 hour max.
- **How much time is needed to complete experiments (approximately)?:** 5 hour max.
- **Publicly available?:** Yes: https://github.com/precisionbatching/pbatch.
- **Code licenses (if publicly available)?:** Creative Commons Attribution 4.0 International
- **Data licenses (if publicly available)?:** Creative Commons Attribution 4.0 International
- **Archived (provide DOI)?:** 10.5281/zenodo.5130610

C. Description

1) How to access: Pull code from: https://github.com/precisionbatching/pbatch. See README for installation and run instructions.

2) Hardware dependencies: Linux OS with NVIDIA T4 GPU (cc7.5+). Cuda 10.2.

3) Software dependencies: Libraries are installed through Conda.

4) Data sets: None required – the code will automatically pull it.

5) Models: Included in the repo.

D. Installation

The README contains step by step instructions for installation.

E. Experiment workflow

The README contains step by step instructions for running the experiments.

F. Evaluation and expected results

See sample_script_out which shows the expected output.

G. Experiment customization

N/A.

H. Notes

Several things to note: we skip the reinforcement learning evaluations, as running these tasks require a paid license (Mujoco); furthermore, required models are included in the repository but are not the exact same ones we used in the paper, hence will not yield the exact same accuracies, though the broad trends (acc/speedup/etc) are the same; to reduce eval time, we do not evaluate different precisions per layer (variable precision) and hence reproduce the "uniform precision" results.

I. Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-badging
- http://cTuning.org/ae/submission-20201122.html
- http://cTuning.org/ae/reviewing-20201122.html