

Accelerating Binarized Neural Networks: Comparison of FPGA, CPU, GPU, and ASIC

Eriko Nurvitadhi, David Sheffield, Jaewoong Sim, Asit Mishra, Ganesh Venkatesh and Debbie Marr
Accelerator Architecture Lab, Intel Corporation

Abstract— Deep neural networks (DNNs) are widely used in data analytics, since they deliver state-of-the-art accuracies. Binarized neural networks (BNNs) are recently proposed optimized variant of DNNs. BNNs constraint network weight and/or neuron value to either +1 or -1, which is representable in 1 bit. This leads to dramatic algorithm efficiency improvement, due to reduction in the memory and computational demands. This paper evaluates the opportunity to further improve the execution efficiency of BNNs through hardware acceleration. We first proposed a BNN hardware accelerator design. Then, we implemented the proposed accelerator on Aria 10 FPGA as well as 14-nm ASIC, and compared them against optimized software on Xeon server CPU, Nvidia Titan X server GPU, and Nvidia TX1 mobile GPU. Our evaluation shows that FPGA provides superior efficiency over CPU and GPU. Even though CPU and GPU offer high peak theoretical performance, they are not as efficiently utilized since BNNs rely on binarized bit-level operations that are better suited for custom hardware. Finally, even though ASIC is still more efficient, FPGA can provide orders of magnitudes in efficiency improvements over software, without having to lock into a fixed ASIC solution.

Keywords— Deep learning, binarized neural networks, FPGA, CPU, GPU, ASIC, data analytics, hardware accelerator.

I. INTRODUCTION

The proliferation of Internet technologies led to the abundance and rapidly growing digital data, from sources such as social media, blogs, Internet-of-things (IoT) applications, etc. Data analytics extract knowledge from such data, often by using machine learning (ML) algorithms. In particular, deep neural networks (DNNs) have been widely adopted, as they show state-of-the-art accuracies for various analytics classification tasks (e.g., computer vision, speech, etc).

With advances in DNNs, there is a trend towards deeper networks that consequently carry more network parameters with increased model size. For example, AlexNet [8] contains 60M parameters, which demands storage size of 240MB when stored as 32-bit numbers.

Larger DNN models are challenging to execute efficiently. Especially, in fully connected layers where there is no data reuse, processing a larger model that does not fit in on-chip RAMs would lead to off-chip DRAM accesses. Such accesses are very energy inefficient compared to on-chip operations (e.g., for 45nm CMOS [9], a 32-bit DRAM access requires 172x more energy than a floating point multiply). Moreover, performance becomes limited by bandwidth available to access the model from DRAM. Batching multiple inputs together can help improve data re-use, but in practice only small batch size is tolerable due to real-time latency requirements in analytics

servers [3]. For IoT platforms, real-time requirements will be even more stringent, and batching may not be feasible at all.

Binarized Neural Networks (BNNs) [1][2] have very recently been proposed to address the aforementioned challenge. A BNN offers an extremely more compact representation of network weights and neuron values than a normal DNN by constraining each value to either +1 or -1. As such, storage need is dramatically reduced since the weights can be stored in a single bit (i.e., +1 stored as 1, and -1 as 0). Furthermore, multiply operations can be replaced by bit-wise operations instead, thereby reducing computational demand as well. So far, BNNs have been shown to offer comparable accuracies to full-precision DNNs for some known datasets (e.g., CIFAR10), and they are actively being studied to improve accuracies for more datasets (e.g., ImageNet). However, while prior works [1][2] have offered in-depth algorithm studies and analyses of BNNs, we are not aware of any that has proposed a hardware accelerator for BNNs.

Neural network analytics workloads are deployed in a wide range of settings, from high-end servers in data centers for cloud-scale analytics to mobile platforms for Internet-of-Things (IoT) applications. In all cases, there is a strong need for extreme energy efficiency in addition to high performance. To this end, both cloud servers as well as IoT platforms have become heterogeneous in recent years, where they integrate hardware accelerators alongside general purpose CPUs to deliver significant execution efficiency for computations offloaded to these accelerators, while maintaining generality to execute the rest of the workloads. FPGAs, GPUs, and ASICs are the well-known accelerators available in the market today. In particular, FPGAs have become more widely adopted in cloud servers as well IoT platforms. Leading technology companies are pushing towards integrating FPGAs into data centers (e.g., Intel Xeon+FPGA, Microsoft Catapult). There are also IoT platforms (e.g., Altera SoC FPGA family) integrating embedded processor(s) and FPGA in a single package.

This paper investigates the opportunities for accelerating BNNs. We made the following contributions. First, we propose hardware accelerator architecture for BNNs. Second, we explore software enhancements for BNNs (e.g., replace full-precision with binary operations) for CPU and GPU. Third, we evaluate our accelerators on state-of-the-art Altera Aria 10 FPGA and 14nm ASIC, and compare them against optimized software on a cloud-server with Intel Xeon CPU and Nvidia Titan X GPU and IoT platform with mobile Nvidia TX1 GPU. We show that software optimized for BNNs deliver significant performance improvements over standard DNNs. Moreover, we show that hardware accelerators offer further order of magnitude efficiency improvements over optimized BNN

software CPU and GPU implementations, since they take better advantage of BNN bitwise data formats and operations.

The rest of the paper is organized as follows. Section II gives background on ML analytics and BNNs. Section III presents the proposed BNN accelerator. Section IV details the BNN software optimizations on CPU and GPU. Section V presents our evaluation results. Finally, section VI and VII offer related work and concluding remarks, respectively.

II. BACKGROUND

A. Machine Learning for Data Analytics

Classification vs. Training. Many data analytics workloads rely on machine learning (ML) algorithms. A typical ML setup for data analytics consists of two phases. First, during *training phase*, a known set of data samples is fed into an ML algorithm, which then creates a model with predictive power. Then, in the *classification phase*, this model is used by the ML algorithm to make predictions for any new given data samples. This paper focuses on binarized neural networks (BNNs) for classification phase.

Batched Classification. In the classification phase, a popular optimization is to process a batch of multiple input samples together to improve data reuse and throughput. However, batching increases processing latency since a batch of outputs is produced at a time, instead of a single output at a time. Moreover, batching can increase implementation complexity, due to the need to group incoming requests into batches and schedule them properly for processing. In practice, it can be impractical to use large batch sizes. E.g., in a commercial analytics based on neural networks in [3], ~90% of the time there are only up to 4 inputs that can be grouped together (batch size of 4), with a maximum of 10 inputs (batch size of 10). This is due to the need to meet the stringent processing latency constraints. This paper considers normal as well as batch-mode in our evaluations.

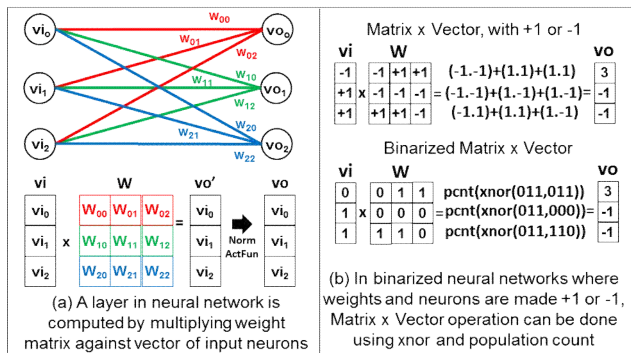


Fig. 1. In binarized neural networks, the matrix x vector operation to compute each network layer can be replaced by xnor and bit counting because weights and neurons are constrained to either +1 or -1, each representable in 1-bit.

B. Binarized Neural Networks (BNNs)

In a deep neural network, a fully connected layer performs the following computation

$$v_o = f(W.v_i + b) \quad (1)$$

Where v_i is a vector of input neurons, W is a matrix of the network weights, b is the bias, and v_o is the vector of output neurons for the layer. f is the activation function, such as Rectified Linear Unit (ReLU). Optionally, f may also include normalization (e.g., batch normalization [10]) prior to applying the activation function. Often times, b can be merged into v_i . Figure 1(a) shows an example DNN layer computation as a W matrix x v_i vector operation.

There has been recent trend towards deeper networks with more parameters, since such networks can provide better accuracies. As such, the size of W , v_i , and v_o have become noticeable large. For example, one of the fully connected layers in AlexNet [8] and VGG [11] use a 4K x 4K weight matrix (W). When each weight is represented as a 32-bit number, storing the W matrix would require 64MB of storage. In practice, processing such a model efficiently is very challenging, since it does not fit in on-chip RAMs of a typical system. Hence, some or most of the model will have to reside in DRAM memory, which is power consuming and has much lower bandwidth than on-chip RAMs, thereby imposing performance constraints. As stated earlier, DRAM accesses are significantly more energy consuming than on-chip operations.

Binarized neural networks (BNNs) have the potential to address this issue. BNNs [1][2] have been proposed recently to improve the efficiency of the standard neural networks. In a BNN, each network weight and neuron value is constrained to be of only two possible values, +1 or -1. As such, it can be represented using a single bit. Therefore, BNNs require significantly less storage than standard DNNs. In our previous example of 4K x 4K weight matrix, instead of needing 64MB storage when using 32-bit number representation, a binarized weight matrix would require only 2MB storage (i.e., 32x less). BNNs also improve computation efficiency, as discussed next.

There are three types of computations in a BNN.

Binarized Matrix x Full-Precision Vector. In the first layer, the input neurons represent the input sample data. Thus, they cannot be binarized. So, in this case, v_i is still represented using full 32-bit floating or fixed point. Each weight in a binarized weight matrix (W_b), however, is a 1-bit value. Thus, the computation for the first layer is a multiplication of 32-bit v_i against binarized W_b . This operation can efficiently be done by adjusting the sign bit of v_i against the 1-bit weight of W_b . I.e., if they are of the same sign, the output should maintain the sign bit. Otherwise, the output should have the opposite sign.

Binarized Matrix x Binarized Vector. Since activation function in BNN [1][2] produces a +1 or -1 value, neurons (v_i and v_o) after the first BNN layer would be representable as 1-bit values. As such, the computation multiplies a binarized vector of input neurons (v_i) against a binarized weight matrix W_b . Such operation can be done using xnor and a variant of a population count (pcnt), thereby eliminating the need for full-precision operations. Figure 1(b) illustrates how a matrix x vector operation of +1 and -1 values can be binarized and computed using xnor and pcnt.

Normalization and Activation Function. Lastly, normalization and activation function are applied to finalize the

output neurons. It has been recommended [2] to use batch normalization [10] with BNN, which involves applying several constant parameters obtained from training phase (i.e., γ , β). For the activation function, ReLU is very commonly used, and is also used in BNN [2]. As such, this paper uses ReLU.

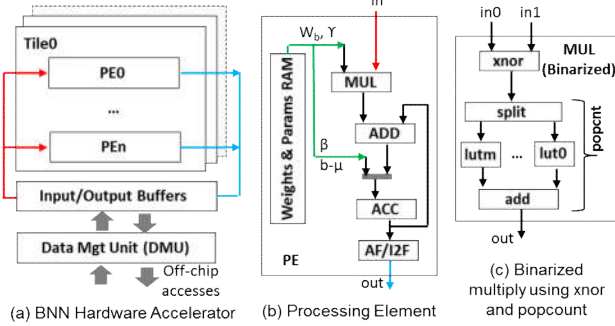


Fig. 2. The proposed accelerator for binarized neural networks (BNNs).

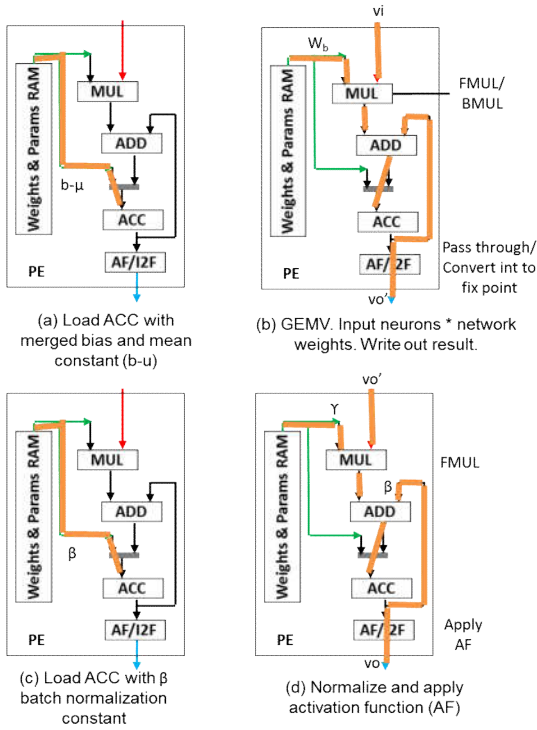


Fig. 3. Sequences of operations that a processing element takes to process a BNN layer. (a) Load initial ACC constant. (b) Multiplication of input neurons against weights. (c)(d) Normalization and activation function.

III. HARDWARE ACCELERATOR

We propose hardware accelerator architecture for BNNs. It supports all the operations needed to process arbitrary BNNs. It is especially designed to realize the efficiency benefits of BNNs. It contains a scalable number of processing elements, along many distributed on-chip RAMs. The network parameters (e.g., binarized weights, normalization constants) are kept in these on-chip RAMs and supplied to the many PEs

performing the computation in parallel. In result, the many on-chip RAMs deliver sufficient bandwidth to the PEs to achieve high throughput at extreme efficiency. This section first describes the architecture of the proposed accelerator. Then, it details implementations of such architecture onto an Altera Aria 10 FPGA as well as 14nm ASICs.

A. Accelerator Architecture

Architecture Details. The high-level architecture of the proposed BNN accelerator is shown in Figure 2(a). The architecture consists of a number of processing elements (PEs). It can be scaled up (or down) by adding more (or less) PEs. Each PE works on computing either a single full-precision neuron value or multiple binarized values in a packed format. The PEs are connected to on-chip RAM buffers, which are used to keep the input and output neuron values, as well as temporary values, for the BNN layers being processed. The data management unit (DMU) handles the movement of data in and out of the accelerator. It brings in the input neuron values and writes out the final output neuron values. It also loads network parameters to internal PE RAMs.

The PE internal design is shown in Figure 2(b). It consists of a local RAM that keeps network weights. Each weight is 1-bit. In our PE design, we pack 32 weights into a 32-bit value for efficient processing. The RAM also keeps initialization (e.g., 0, b, $b-\mu$) and batch normalization (i.e., β , Y) parameters.

A PE also contains a multiplier unit (MUL), an adder unit (ADD), an accumulator register (ACC), and an AF/I2F unit. To cover all BNN operations discussed earlier, the PE supports both full-precision and binarized operations. However, since binarized operations are more performance critical, and there is only few BNN operations that rely on full-precision, we chose to evaluate the more efficient fixed point for full precision support in this paper (i.e., instead of floating point).

The MUL unit supports both full-precision fixed point (FMUL) and binarized multiplication (BMUL) operations. The datapath to support BMUL is shown in Figure 2(c). It consists of an xnor unit, as well as a set of look up tables and adders to perform the specialized population count needed for BMUL.

The PE ADD unit is a full-precision adder, used either to accumulate the integer BMUL output or full-precision results from the first-layer computation or batch normalization.

The AF/I2F unit applies transformations to the accumulated value prior to writing it to the output RAM buffer. These transformations include: applying activation function (we use ReLU in this study) and converting integer to fixed point.

Accelerator Operations. The proposed accelerator supports all the operations needed to process BNNs. Figure 3 illustrates the sequence of PE operations when processing a BNN layer. They work as follows.

First, an initialization parameter is loaded to ACC register. An initialization parameter is the constant offsets to be applied to output neuron values. In a typical setup where a BNN layer includes a bias node and utilizes batch normalization at its output [10], the offset would be $b-\mu$. This parameter can be adjusted for other BNN variants. For example, if batch

normalization is not used, then this could be set to the bias parameter b . Further, if bias is not used, this could be set to 0.

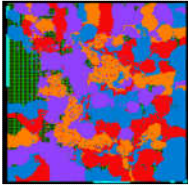
Second, the input neuron values to the layer are multiplied against network weights. For first BNN layer, input neurons are fixed points. Hence, a PE will multiply-accumulate a single neuron value with a single weight at a time (i.e., FMUL and FADD). For other layers (hidden and output layers), the input neuron values and the weights are binarized (single bit each). Therefore, the PE can multiply-accumulate a set of packed weights and neuron values at a time (i.e., BMUL and integer ADD). In our study, we pack 32 weights and neuron values together into 32-bit chunks. So, a PE can perform 32 binarized multiply-accumulate at a time. This improves efficiency and speeds up computation. E.g., relative to a 32-bit representation of weights and neurons, this means 32x speedup in multiply-accumulate computation. The accumulated results are then written to PE temporary buffers. If this is the first BNN layer, no data transformation is needed, and AF/I2F unit is set to simply pass through the result to write out. For other layers, the accumulated result is integer, and AF/I2F unit is set to convert it into fixed point (I2F operation).

To produce the final output neurons for the layer, the ACC is loaded with batch normalization parameter β (figure 3(c)). Then the accumulated result that was written out to temporary buffer is read back into the PE. It is then multiplied against the other batched normalization parameter γ and accumulated with β that was loaded into the ACC earlier. The updated ACC value is then fed into AF/I2F unit, where activation function (AF) is applied to produce the final neuron output. The final output is written back to the PE buffers. (Figure 3(d)).

B. Implementations on FPGA and ASIC

For evaluation, we developed a Verilog RTL implementation of the BNN accelerator detailed in the previous sub-section. We used the BNN software from [2] as functional reference. The RTL is parameterizable to facilitate design space exploration. For example, a parameter can be set to output an RTL instance with arbitrary number of PEs, which we can use to scale up/down various design instances for us to study. From this parameterized Verilog RTL, we map our accelerator architecture onto FPGA and ASIC, which we describe in further detail below.

Design name	FPGA64	FPGA1024	ASIC64	ASIC256
Frequency	220MHz	150MHz	1GHz	1GHz
Implementation	Aria 10	Aria 10	14nm	14nm
Number of PEs	64	1024	64	256
On-chip RAMs	~4MB	~4MB	~4 MB	~4 MB
Peak throughput (TOP/sec)	0.9	9.8	4	16



(a) Specifications of accelerators under study

(b) ASIC64 place & routed

Fig. 4. FPGA and ASIC accelerators under study. (b) shows ASIC64 design place and routed on 14nm technology. Each color is a 16-PE tile.

FPGA. FPGA technologies have advanced rapidly. There are increasing numbers of on-chip RAMs, hard DSPs for arithmetic operations, and reconfigurable fabric resources in newer FPGAs. As such, FPGAs have the potential to offer very efficient BNN accelerator implementations. The compact

binarized weights for interesting problem sizes can fit in many distributed on-chip FPGA RAMs that deliver abundance of on-chip bandwidth to the reconfigurable fabric and DSPs to perform high-throughput computation on packed binarized neuron and weight values.

This paper targets a high-end Altera Aria 10 FPGA, which contains ~6MB of on-chip RAMs (i.e., 2713 M20Ks resources) and 1518 hard DSP units. Note that while Aria 10 is the latest Altera family available today, the next-generation Stratix 10 family is slated for release soon. Stratix 10 will offer up to ~28 on-chip RAMs, ~5K DSPs, and higher frequency. Thus we expect dramatic increase in FPGA performance in the near future when Stratix 10 becomes available.

In our evaluation, we first start by using our parameterized Verilog RTL to produce a small design instance (e.g., few PEs). Then, we increase the design parameters to scale up, until we can no longer fit the design onto the FPGA. This largest design will be used to represent a high-performance design for server applications. Additionally, we also study a smaller scale design for IoT application.

We use Altera Quartus Prime to do our synthesis and mapping to FPGA. To calculate power estimate for FPGA, we use Altera’s PowerPlay Early Power Estimator tool [13]. We check to ensure that we are properly writing the RTL such that the tool infers the appropriate FPGA resources. E.g., on-chip PE RAMs are mapped to M20Ks, and the full-precision multiplier units are mapped to DSPs.

The largest design we can fit our target Aria 10 FPGA contains 1024 PEs and ~4MB of on-chip RAMs. We also chose another smaller scale design to study, which contains 64 PEs. The specifications for these designs (FPGA64, FPGA1024) are shown in Figure 4(a). In FPGA1024, while we are able to utilize all the DSPs in the Aria 10, we are not able to use all the on-chip RAMs (M20Ks) due to routing constraints.

In Figure 4(a), we also report peak throughput as terra operations per second (TOP/sec). This represents 1-bit multiply and accumulation operations on network weights and neurons. It is calculated as follows. As an example, the FPGA1024 design contains 1024 PEs and each PE does 32-bit packed weights calculation in parallel in a pipelined fashion to retire 32 new results each cycle. So, at 150MHz frequency, the peak throughput is 1024 PEs x (32 bits packed x (1 multiply + 1 accumulate)/PE) x 150M operations per second. This results in 9.8 TOP/sec. Such a high peak throughput is feasible due to the significant efficiency benefit of binarization.

ASIC. For ASIC evaluation, we study design instances with 64 and 256 PEs. These designs are synthesized using Intel 14nm ASIC flow, for which the area and power estimates are obtained. Both designs meet the target frequency of 1 GHz. Memory elements are modeled using CACTI. The summary of both implementations are provided in Figure 4(a). Figure 4(b) shows a place-and-routed 64-PE design (i.e., ASIC64). In the figure, each of the four tiles in the design is highlighted with a different color, where each tile contains 16 PEs.

In ASIC64, since the design runs at 1GHz, the 64-PE design can deliver a peak throughput of 64 PEs x (32 bits packed x (1 multiply + 1 accumulate)/PE) x 1G operations per

second, which results in 4 TOP/sec. Scaled accordingly, the 256-PE design can deliver a peak throughput of 16 TOP/s. In both designs, the on-chip RAMs account for a non-trivial portion of the total chip power and area.

IV. SOFTWARE ON CPU AND GPUS

To evaluate the effectiveness of the proposed hardware accelerator architecture, we compare the FPGA and ASIC implementations against a variety of optimized software implementations on CPU and GPU platforms. For all the platforms, we evaluate optimized software implementations of baseline SGEMV for standard neural networks as well as binarized GEMV for BNNs.

```
#pragma omp parallel for
for(int i=0; i<n; i+=fBlkI)
for(int j=0; j<m; j+=fBlkJ)
for(int k=0; k<_k; k+=fBlkK){
for(int jj=0; jj<fBlkJ; jj++)
for(int kk=0; kk<fBlkK; kk++){
bt[jj][kk] = B[(k + kk)*m + j + jj];
for(int ii=0; ii<fBlkI; ii+=fBlkII)
for(int jj=0; jj<fBlkJ; jj+=fBlkJJ)
for(int kk=0; kk<fBlkK; kk+=fBlkKK){
ct_00 = C[(i+ii+0)*m+jj+0]; ct_01 = C[(i+ii+0)*m+jj+1];
ct_10 = C[(i+ii+1)*m+jj+0]; ct_11 = C[(i+ii+1)*m+jj+1];
ct_20 = C[(i+ii+2)*m+jj+0]; ct_21 = C[(i+ii+2)*m+jj+1];
ct_30 = C[(i+ii+3)*m+jj+0]; ct_31 = C[(i+ii+3)*m+jj+1];
for(int kkk=0; kkk<fBlkKK; kkk++){
b0 = bt[jj+0][kk+kkk]; b1 = bt[jj+1][kk+kkk];
ct_00 += popcnt(A[(i+ii+0)*_k + k+kk+kkk]^b0);
ct_01 += popcnt(A[(i+ii+0)*_k + k+kk+kkk]^b1);
ct_10 += popcnt(A[(i+ii+1)*_k + k+kk+kkk]^b0);
ct_11 += popcnt(A[(i+ii+1)*_k + k+kk+kkk]^b1);
ct_20 += popcnt(A[(i+ii+2)*_k + k+kk+kkk]^b0);
ct_21 += popcnt(A[(i+ii+2)*_k + k+kk+kkk]^b1);
ct_30 += popcnt(A[(i+ii+3)*_k + k+kk+kkk]^b0);
ct_31 += popcnt(A[(i+ii+3)*_k + k+kk+kkk]^b1);
}
C[(i+ii+0)*m+jj+0] = ct_00; C[(i+ii+0)*m+jj+1] = ct_01;
C[(i+ii+1)*m+jj+0] = ct_10; C[(i+ii+1)*m+jj+1] = ct_11;
C[(i+ii+2)*m+jj+0] = ct_20; C[(i+ii+2)*m+jj+1] = ct_21;
C[(i+ii+3)*m+jj+0] = ct_30; C[(i+ii+3)*m+jj+1] = ct_31;
}
}
```

Fig. 5. CPU implementation of binarized matrix multiply ($C = A \times B$).

A. Baseline SGEMV/SGEMM on CPU/GPUs

For CPU evaluation, we use a high-performance 2.3 GHz Intel® Xeon E5-2699v3 server (i.e., Haswell-EP). It has 90 MB of aggregate LLC and 36 physical cores. For baseline SGEMV, we enabled MKL and OpenMP, ensuring that the software is taking advantage of multi-threaded execution across the 36 physical cores. Runtime and power measurements are done using performance counters.

For GPU evaluation, we use a high-performance Nvidia Titan X GPU, as well as Nvidia mobile GPU (mGPU) on TX1 embedded development platform. For baseline SGEMV on GPU, we use cuBLAS libraries. We measure power using nvidia-smi utility on Nvidia Titan X. Since TX1 did not provide such facility, we measured power using Kill-A-Watt power meter. We ran the software in a loop until wall power measurement stabilized. To get best performance in TX1, we forced all clocks to run at maximum speed (i.e., ~1 GHz), as

the default clock management scheme provided sub-optimal performance (i.e., ran at ~70MHz).

B. Binarized GEMV/GEMM on CPU

For binarized GEMV, our Haswell-EP platform has built-in instructions for population count exposed through the SSE4a extension to the x86 ISA. These instructions are `popcnt` for 32-bit operands and `popcntl` for 64-bit operands. While included in the SSE4a set of instruction extensions, they are not SIMD instructions and only execute on scalar register values. On the Haswell microarchitecture, a population count instruction can be initiated every cycle –yielding 64 “binary ops” per cycle. In contrast, a well-tuned single precision implementation of matrix multiply using AVX2 FMA instructions can retire at most 32 flops per cycle, or $\frac{1}{2}$ the throughput of the population count based binary operation. Therefore, a tuned binary matrix multiply implementation has a performance roofline of 2x over a tuned single precision implementation of matrix multiply.

As binary matrix multiply is not included in standard BLAS packages, we wrote our own implementation (shown in Figure 5). Our implementation uses an outer level of cache blocking and an inner-level of register blocking in order to achieve compute-bound performance. The outer block is sized to fit in the 256 kB L2 cache of our Haswell CPU. In code listing shown in Figure 5, we explicitly copy and transpose the outer cache block into the 2d array “bt” in order to achieve better memory locality and increase the cache hit rate.

The inner block is sized to fit in CPU registers (“ct_xx” in the code listing). We experimentally determined that a 4x2 register block yields the highest performance on our platform as larger register block sizes incur spilling while smaller block sizes do not have enough register reuse. Finally, we use OpenMP to parallelize across CPU cores.

C. Binarized GEMV/GEMM on GPUs

We evaluate a binary matrix multiply kernel (`xnor_gemm`) from BinaryNet [2]. The CUDA implementation uses shared memory blocking to reduce the number of access to global memory. For matrix multiplication of $C = A \times B$, each thread block loads sub-matrices of A and B from global memory into shared memory. Then, each thread in thread blocks computes one element of the sub-matrix C using `xnor` and `__popc()` operations. The evaluated `xnor_gemm` kernel is similar to the blocked version of matrix multiply in the CUDA Programming Guide except for the code for computing the product C.

The population count operation is natively supported in Nvidia GPU devices via `__popc()` (for 32-bit operands) and `__popcll()` (for 64-bit operands) intrinsic functions. These are directly used in the CUDA kernel, and the CUDA compiler maps `__popc()` to a single instruction and `__popcll()` to a few instructions.

On our evaluated GTX Titan X platform, 32 32-bit population count operations can be issued every cycle per Streaming Multiprocessor (SM) – yielding 1024 “binary ops” per cycle. As GTX Titan X can issue up to 128 32-bit floating-point operations every cycle per SM, the performance roofline of “binary ops” over FP32 operations is 4x.

V. EVALUATION

We studied a set of neural network layer configurations that are used by popular networks, such as AlexNet [8], VGG [11], and Neural Talk (NT) [12]. See Table I. We focus on the fully connected layer, which contains most of the weights in the network and are the most challenging due to the large model size. As stated in the introduction, larger models in fully-connected layers are challenging to execute efficiently since they do not fit on-chip, necessitating off-chip DRAM accesses that are very energy inefficient and imposes performance limit on the DRAM bandwidth available to access these models.

TABLE I. NEURAL NETWORK LAYER CONFIGURATIONS UNDER STUDY.

Name	Outputs	Inputs	Binarized model size (MB)
Alex/VGG 7	4096	4096	2.00
Alex/VGG 8	1000	4096	0.49
NT-We	600	4096	0.29
NT-Wd	8791	600	0.63
NTLSTM	2400	1201	0.34

For FPGA, we only evaluate layers that can fit on the ~4MB RAMs that our FPGA design could use. We evaluate

performance and performance/watt. For the high-performance platforms (Xeon CPU, Titan X GPU), we also evaluate batched execution with batch size of 10, as suggested in [3]. For non-binarized software evaluation, the batched experiments called CPU or GPU SGEMM kernels, while the non-batched experiments called SGEMV kernels.

The evaluation results are shown in Figure 6, 7, and 8. Figure 6 and 7 show performance and performance/watt, relative to non-batched baseline CPU software. Figure 8 depicts the fraction of peak performance that is achievable, indicating platform utilization. E.g., 50% means only half of the peak performance available in the platform was achievable during our experiments.

A. CPU versus GPU

In a normal (no batching) mode, CPU performs comparably well to GPU, as show in Figure 6. On average, non-batched CPU has ~90% better performance than non-batched GPU. Among the five network layer configurations, GPU performs almost comparable to CPU only for Alex/VGG 7 where the number of outputs is equal to number of inputs (i.e., the weight matrix is square). In other cases, GPU is always noticeably inferior to CPU.

Even though GPU has much higher peak performance, it is

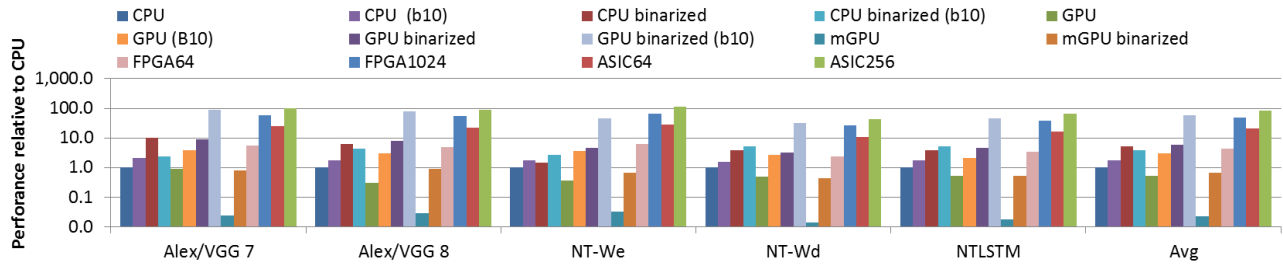


Fig. 6. Performance relative to baseline software on CPU. I.e., above 1 means speedup, while less than 1 means slowdown.

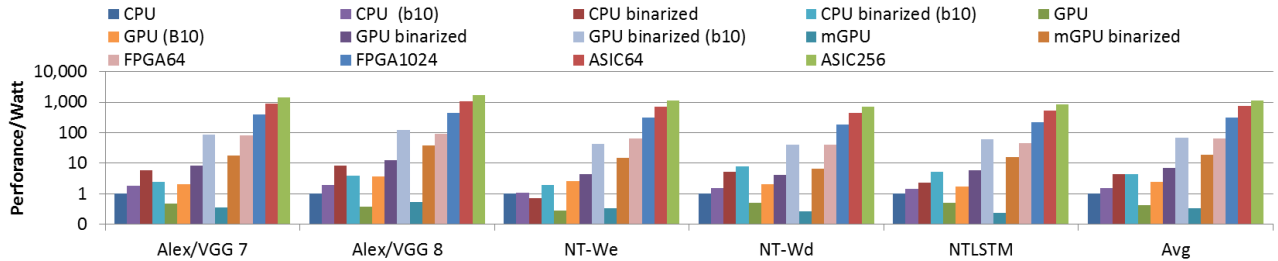


Fig. 7. Performance/Watt relative to baseline software on CPU.

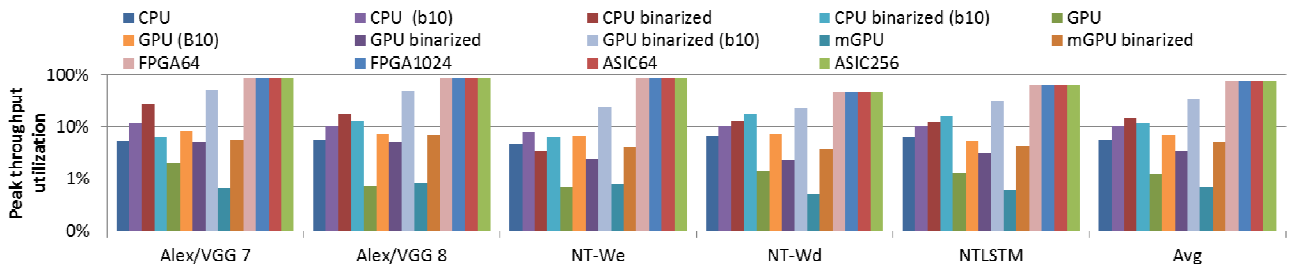


Fig. 8. Achieved performance relative to peak. E.g., 50% means only half of peak performance is realized.

extremely underutilized (i.e., $\sim 1\%$ utilization on average, as shown in Figure 8). The CPU is also underutilized ($\sim 6\%$), but not as much as the case with GPU. The low utilization is due to the challenge in being able to extract fine-grained parallelism out of the weight matrices. Without batching, there is only a single set of inputs (i.e., a vector) that is being multiplied against the weight matrix. Thus, there is limited data re-use. Unless the platform can extract sufficient fine-grained parallelism from this single matrix \times vector operation to utilize the available platform resources, it is inevitable that the platform would suffer from underutilization.

For CPU and GPU, when scaling up to multiple software threads, if there is only a small amount of data to process, the overhead of threading can end up being the dominant one.

For the mobile GPU, as Figure 6 shows, its performance is much worse than a server CPU (i.e., $\sim 40\times$ worse on average). The mobile GPU also suffer from extreme underutilization ($\sim 1\%$ on average, as shown in Figure 8), as in the case with high-performance GPU. However, the mobile GPU has much lower peak performance.

Consequently, CPU achieved a better overall performance/watt than both the high-performance GPU as well as mobile GPU, as depicted in Figure 7. As such, for non-batched neural networks, CPUs can be a better overall solution than GPU, delivering comparable performance while achieving better energy efficiency.

B. Impact of Batching Multiple Inputs/Outputs

Batching improves performance as well as utilization for both CPU and GPUs. This is because batching enables more data reuse, since there are multiple input vectors (forming an input matrix) to be multiplied against the weight matrix.

As shown in Figure 6, batching improves performance by $\sim 80\%$ for CPU and $\sim 5.8\times$ for GPU. Accordingly, performance/watt improves by similar degree, as shown in Figure 7.

Batching improves CPU utilization by almost $2\times$ (from 6% to 10%) and GPU by $7\times$ (from 1% to 7%). Even though batching leads to noticeable improvements in utilization, at 10% utilization for CPU and 7% for GPU, in overall these platforms are still underutilized.

Furthermore, as explained in Section 2, batching increases latency. So, if possible, a solution that improves performance without necessitating batched operations would be preferable.

C. Impacts of Binarization

Binarization provides the potential to deliver significant performance improvements, since it reduces the storage requirements as well as computational demands. For CPU and GPU, smaller datasets means that they are more cacheable and can be kept on-chip. Further, binarized GEMV operation requires less computation than SGEMV, as discussed earlier.

Indeed, our results in Figure 6 show that binarized CPU software has $5\times$ better performance than baseline CPU. For GPU, binarization improves performance by $\sim 11\times$. Binarization leads to larger speedups than batching. For

example, while batching delivers 80% performance boost for CPU, binarization offers $5\times$ improvements, which is $6\times$ better than batching. GPU has similar trend as well. Moreover, binarized operations can be batched as well. Further speedups can be achieved by combining both batching and binarization.

Hence, one can choose to do binarization only, which delivers improvements better than batching, while meeting low latency requirements. Or, one can combine binarization with batching to achieve better throughput, if latency constraints are not as stringent.

Accordingly, as shown in Figure 7 and 8, binarization leads to improvements in performance/watt as well as utilizations.

D. Hardware Acceleration

Beyond software optimizations, both FPGA and ASIC accelerators can deliver even further improvements in performance and performance/watt. As shown in Figure 6, our FPGA and ASIC accelerators deliver one to two orders of magnitude speedups over the baseline CPU. The high-performance FPGA1024 design delivers almost $50\times$ performance improvement over the baseline CPU.

These large performance speedups from accelerators are due to the custom hardware design for BNN, which consists of PEs that are well integrated with distributed on-chip RAMs to deliver neural network parameters to the PEs at a sufficiently high bandwidth to keep the PEs well utilized. The PE is also equipped with native support for binarized operations.

Indeed, as shown in Figure 8, our accelerators achieve significantly higher utilizations (i.e., $\sim 75\%$) than the software implementations on CPU and GPU. As such, even though our accelerators have lower peak performance than the high-performance GPU, they are able to utilize most of it, resulting in significant performance improvements over GPU.

As depicted in Figure 7, energy efficiency improvements achieved by the accelerators are even better. The ASIC implementations offer four orders of magnitudes in improvements over CPU baseline, while the FPGA offers three orders of magnitude.

E. FPGA versus ASIC

The general rule of thumb is that FPGA will be about an order of magnitude less efficient than ASIC. However, modern FPGAs contain “hardened” resources, such as DSPs for arithmetic operations and M20Ks (in Altera FPGA) for on-chip RAMs. When an FPGA design is implemented such that it uses these hard blocks, the efficiency gap between FPGA and ASIC can be reduced. This is the case for our BNN accelerators, which heavily use M20Ks on-chip RAMs and DSPs for arithmetic operations.

Both FPGA64 and ASIC64 designs adopts the same microarchitecture (i.e., number of PEs and RAMs), hence they provide a direct comparison between FPGA and ASIC. Between these two designs, ASIC64 has $\sim 4.5\times$ higher performance than FPGA64 since it has higher frequency.

In terms of energy efficiency (i.e., performance/watt), ASIC64 is $\sim 11\times$ better than FPGA64. However, Aria 10 FPGA

is fabricated on a 20nm TSMC process technology, while the ASIC is on 14 nm Intel technology. Normalizing for such process technology difference, the FPGA/ASIC efficiency gap in this case is estimated to be less than $\sim 8x$, which is lower than the abovementioned rule of thumb. However, we think the less than $\sim 8x$ ASIC/FPGA gap is due to the fact that our BNN accelerator heavily take advantage of the hard FPGA blocks (M20K for on-chip RAMs, hard DSPs for multiply/add).

For the larger scale high-performance designs (FPGA1024, ASIC256), the large Aria 10 FPGA allowed us to implement 1024-PE design, but at a lower frequency than ASIC (150MHz vs. 1GHz). Thus, while FPGA has more PEs, it runs slower, resulting in worse performance than the ASIC256 design.

F. Opportunities for FPGAs

The upcoming Altera Stratix 10 FPGA will offer even more M20Ks and DSP hard blocks. Therefore, we can expect to deploy designs with even more number of PEs in the Stratix 10 when it becomes available.

Furthermore, Stratix 10 has the new HyperFlex technology to deliver higher operating frequency through retiming. Since our BNN accelerator does not have tight data dependencies and is amenable to re-timing, we expect that our accelerator can take advantage of Stratix 10 support for higher frequency.

The aforesaid trends highlight the tremendous opportunities for FPGAs. Unlike with fixed ASIC design, FPGAs can be reconfigured for other uses as well as newer improved versions of an accelerator. Thus, if the FPGA-to-ASIC efficiency gap narrows, there is a stronger case to adopt FPGA solutions.

VI. RELATED WORK

To the best of our knowledge, we are the first to propose hardware accelerator for BNNs. The original BNN paper [1] focused on the BNN algorithm. It describes the benefits of BNNs through algorithmic complexity analysis. A more recent BNN work (BinaryNet [2]) shows an evaluation of binarized GEMM on GPU using xnor and population count. In contrast, this paper proposes hardware accelerator architecture for BNNs, and offers comprehensive comparative evaluation across various interesting problem sizes, on FPGA, ASIC, server CPU, server GPU, and mobile GPU.

Aside from BNNs, there are myriad of existing accelerators for Deep Learning (DL), targeting both FPGAs (e.g., [6]) as well as ASICs (e.g., [7]). However, none of them target BNNs. BNNs are unique, since they represent each network weight using a single bit, which requires a proper acceleration strategy to take full advantage of such bit-level representations. There are also existing studies on machine learning accelerators (e.g., [14][15]), which unlike this work, target non-DL algorithms.

Multiplication of a dense matrix against a dense vector (GEMV) is a well-known construct that is part of the standard BLAS library. There are existing studies (e.g., [4]) that evaluate BLAS on CPUs, GPUs, and FPGAs. Unlike prior work, this paper focuses on a binarized GEMV. Moreover, this work offers comparison with an ASIC, while others only consider CPU, GPU, and/or FPGA. And, this paper targets more modern platforms. Finally, a recent study [16] evaluates

neural network (NN) implementations on CPU, GPU, FPGA, and ASIC. However, it focuses on recurrent NNs, not BNNs.

VII. CONCLUSION

Binarized neural networks offer significant algorithmic efficiency improvements over standard full-precision networks. This paper proposed hardware accelerator architecture for BNNs, which delivers superior performance while consuming energy efficiently. We evaluated our accelerator to target Aria 10 FPGA and 14nm ASIC. We compared these accelerator instances against optimized software on a high-performance multi-core CPU and GPU for cloud server, as well as a mobile GPU suitable for IoT. Our evaluation results show that the proposed accelerator can deliver orders of magnitude improvements in performance and performance/watt over well-optimized software on CPU and GPU. Lastly, while FPGA is less efficient than ASIC, the FPGA-ASIC gap may be reduced for designs that heavily utilize hard blocks (DSP, M20K), such as our BNN accelerator. Hence, FPGA offers an attractive solution, which deliver superior efficiency improvements over software, without having to lock into a fixed ASIC solution.

REFERENCES

- [1] M. Courbariaux, Y. Bengio, J-P. David, "BinaryConnect: Training Deep Neural Networks with binary weights during propagations," Neural Information Processing Systems (NIPS), 2015.
- [2] M. Courbariaux, I. Hubara, D. Soudry, et al., "Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1," arXiv:1602.02830 [cs.LG].
- [3] D. Amodei, R. Anubhai, E. Battenberg, "Deep Speech 2: End-to-End Speech Recognition in English and Mandarin," arXiv:1512.02595 [cs.CL].
- [4] S. Kestur, J. D. Davis, O. Williams, "BLAS Comparison on FPGA, CPU and GPU," ISVLSI, 2010.
- [5] D. Mukonoki, T. Imamura, D. Takahashi, "Fast implementation of General Matrix-Vector Multiplication (GEMV) on Kepler GPUs," Euromicro International Conference on Parallel, Distributed, and Network-based Processing, 2015.
- [6] C. Farabet, C. Poulet, J. Y. Han, and Y. LeCun. "Cnp: An fpga-based processor for convolutional networks," In Field Programmable Logic and Applications (FPL), 2009.
- [7] T. Chen, Z. Du, N. Sun, et al., "Dianna: A small-footprint high-throughput accelerator for ubiquitous machine-learning," Architectural Support for Programming Languages and Operating Systems, 2014.
- [8] A. Krizhevsky, et al., "Imagenet classification with deep convolutional neural networks," NIPS, 2012.
- [9] M. Horowitz. Energy table for 45nm process, Stanford VLSI wiki.
- [10] S. Ioffe, C. Szegedy "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," arXiv:1502.03167 [cs.LG].
- [11] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," arXiv preprint arXiv:1409.1556, 2014.
- [12] A. Karpathy and L. Fei-Fei, "Deep visual-semantic alignments for generating image descriptions," arXiv preprint arXiv:1412.2306, 2014.
- [13] Altera's PowerPlay Early Power Estimators (EPE) and Power Analyzer. URL: <https://www.altera.com/support/support-resources/operation-and-testing/power/pow-powerplay.tablet.html>
- [14] E. Nurvitadhi, A. Mishra, D. Marr "A sparse matrix vector multiply accelerator for support vector machine," CASES, 2015.
- [15] E. Nurvitadhi, A. Mishra, Y. Wang, G. Venkatesh, D. Marr, "Hardware accelerator for analytics of sparse data," DATE, 2016.
- [16] E. Nurvitadhi, et al., "Accelerating recurrent neural networks in analytics servers: Comparison of FPGA, CPU, GPU, and ASIC," FPL, 2016.