Accelerating Binarized Convolutional Neural Networks with Software-Programmable FPGAs

Ritchie Zhao[1,*], Weinan Song[2], Wentao Zhang[2], Tianwei Xing[3], Jeng-Hau Lin[4],
Mani Srivastava[3], Rajesh Gupta[4], Zhiru Zhang[1,*]

[1]School of Electrical and Computer Engineering, Cornell University, USA
[2]School of Electronics Engineering and Computer Science, Peking University, China
[3]Department of Electrical Engineering, University of California Los Angeles, USA
[4]Department of Computer Science and Engineering, University of California San Diego, USA

*{rz252, zhiruz}@cornell.edu

Abstract

Convolutional neural networks (CNN) are the current state-of-the-art for many computer vision tasks. CNNs outperform older methods in accuracy, but require vast amounts of computation and memory. As a result, existing CNN applications are typically run on clusters of CPUs or GPUs. Research on FPGA acceleration of CNN workloads has achieved reductions in power and energy consumption. However, large GPUs outperform modern FPGAs in throughput, and the existence of compatible deep learning frameworks give GPUs a significant advantage in programmability.

Recent work in machine learning demonstrates the potential of very low precision CNNs — i.e., CNNs with binarized weights and activations. Such binarized neural networks (BNNs) appear well suited for FPGA implementation, as their dominant computations are bitwise logic operations and their memory requirements are greatly reduced. A combination of low-precision networks and high-level design methodology may help address the performance and productivity gap between FPGAs and GPUs. In this paper, we present the design of a BNN accelerator that is synthesized from C++ to FPGA-targeted Verilog. The accelerator outperforms existing FPGA-based CNN accelerators in GOPS as well as energy and resource efficiency.

1. Introduction

Deep convolutional neural networks (CNNs) have become an important class of machine learning algorithms widely used in computer vision and artificial intelligence. While CNNs have been known to researchers for decades, they were popularized after demonstrating high accuracy at the 2012 ImageNet recognition challenge [15]. Subsequently, CNNs have become the state-of-the-art for image classification, detection, and localization tasks. Research in CNNs and other areas of deep learning continues at a rapid pace, with hundreds of new papers published each year introducing new models and techniques. One indicator of this rate of progress is the improvement in the top-5 accuracy of the ImageNet competition winner over the years: 84.7% in 2012 [15] to 96.4% in 2015 [10].

One challenge to the widespread deployment of CNNs is their significant demands for computation and storage capacity. The VGG-19 network, for instance, contains over 140 million floating-point (FP) parameters and performs over 15 billion FP operations to classify one image [22]. Consequently, the training and inference of modern CNNs is almost exclusively done on large clusters of CPUs and GPUs [4]. One additional benefit of such platforms is the availability of compatible deep learning frameworks such as Caffe [12], Theano [24], or TensorFlow [9], which allow users to make use of the latest models or to train a custom network with little engineering effort.

While CPU and GPU clusters are currently the go-to platforms for CNN and many other machine learning applications, a customized hardware solution on FPGA can offer significant improvements in energy efficiency and power dissipation. These factors may be critical in enabling the increased use of CNNs in low-power settings such as unmanned drones or embedded computing. Recent work by Microsoft has even explored cost-effective acceleration of deep learning on FPGAs at datacenter scale [18]. There are also efforts in the academic community on FPGA-based CNN accelerators [27, 19] as well as tools for generating them automatically [23, 26]. Yet, there remains a sizable gap between GPU and FPGA platforms in both CNN performance *and* design effort. The latter is especially distressing given the rate of algorithmic innovation in deep learning — an FPGA-based CNN accelerator (or CNN design compiler) is unlikely to support the most up-to-date models, putting them at a severe competitive disadvantage.

We observe two trends which may help overcome these obstacles. The first is a series of recent papers in the machine learning community regarding very-low-precision CNNs. Networks with binary weights [6], or binary weights *and* activations [7, 21] have in certain cases demonstrated accuracy comparable to full precision nets. Such *binarized neural networks* (BNNs) may be the key to efficient deep learning on FPGA. Binarization reduces storage and memory bandwidth requirements, and replace FP operations with binary operations which can be very efficiently performed on the LUT-based FPGA fabric.

Concerning the cost and effort of FPGA implementation, we see a steady improvement in FPGA design automation tools over the past decade. High-level synthesis (HLS) tools such as Xilinx Vivado HLS [5] and LegUp [1] enable a user to write code in a high-level programming language, then algorithmically compile that code down to a register-transfer level (RTL) design specification. More recent tools such as Intel FPGA SDK for OpenCL [8] and Xilinx SDSoC [13] offer further automation features for generating the hardware-software interface and on-chip memory network. In the context of deep learning, these tools have the potential to critically reduce time-to-market on new accelerator designs and thus reduce the aforementioned innovation gap.

In this paper we present the design of a BNN accelerator for FPGAs. In order to take full advantage of the binarized values and operations, our design differs in multiple aspects from CNN accelerators in literature. Our specific contributions are as follows:

- To our best knowledge, we are the first to study FPGA acceleration for very low precision CNNs. Compared to their full-precision counterparts, such networks are potentially a better fit for the LUT-based fabric and limited on-chip storage in modern FPGAs.

- We employ an HLS design methodology for productive development of our FPGA-based BNN accelerator. Existing HLS work has examined loop ordering, unrolling, and local buffering for CNNs [27]. Our HLS implementation leverages these optimizations, and further propose novel BNN-specific hardware constructs to ensure full throughput and hardware utilization across the different input feature sizes.

- We implement our BNN classifier on a low-cost FPGA development board (ZedBoard) and show promising improvements over CPU and embedded GPU baselines as well as existing FPGA accelerators. Our source code is publicly available on the authors' websites.

The rest of this paper is organized as follows: Section 2 gives a primer on CNNs and BNNs; Section 3 describes our BNN accelerator design; Section 4 provides some details on our HLS code; Section 5 reports our experimental findings, Section 6 reviews previous work on FPGA-based CNN accelerators; and we conclude the paper in Section 7.

2. Preliminaries

In this section we briefly review the basic principles and terminology of CNNs, the differences between a CNN and BNN, and the specific CIFAR-10 BNN model that our accelerator will target.

2.1 Convolutional Neural Network Primer

A CNN is a machine learning classifier that typically takes in a multi-channel image and produces the probabilities of that image belonging to each output class. A typical CNN consists of a pipeline of connected *layers*. Each layer takes as input a set of *feature maps* (*fmaps*), performs some computation on them, and produces a new set of fmaps to be fed into the next layer. The input fmaps of the first layer are the channels of the input image. Layers may require configuration values known as *parameters*, which must first be determined by *training* the CNN offline on pre-classified data. Once the parameters are finalized, the CNN can be deployed for *inference* — the classification of new data points. For most practical machine learning applications, the first-class concerns are the accuracy and execution time of online classification. This paper will thus focus on accelerating the inference task without compromising accuracy.

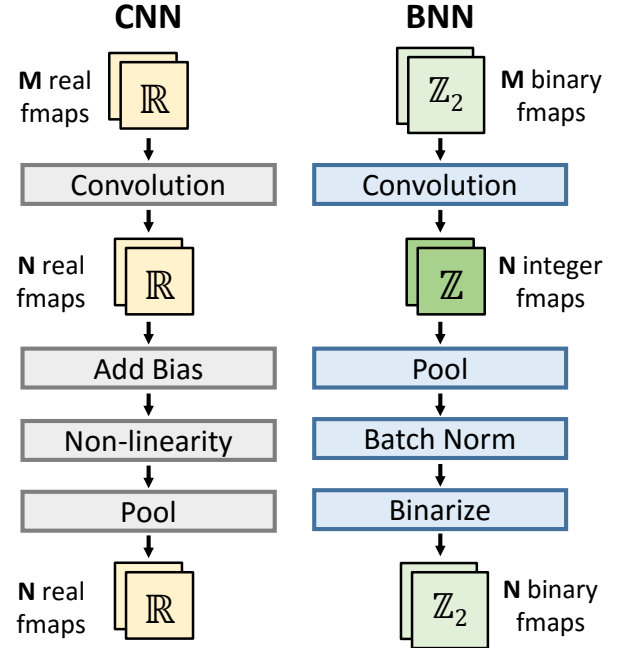Figure 1: **Comparison of CNNs and BNNs —** Left: the order of operations in a CNN for a conv and pool layer. Right: the (modified) order of operations in the BinaryNet BNN [7]. Pooling is performed early and a batch normalization precedes the binarization to minimize information loss. Biases have been removed from the BNN.

Below we describe three layer types which are found in most CNNs, including our CIFAR-10 BNN model.

Convolutional (conv) layers convolve each input fmap with a $K \times K$ weight filter. The conv results are summed, added with a bias, and passed through a non-linearity function (such as ReLU or sigmoid) to produce a single output fmap. In this paper, we assume conv layers *pad* the input fmaps at the borders to produce output fmaps of the same size. Equation (1) below shows the operation of a conv layer with M input fmaps $\mathbf{x}_1, ..., \mathbf{x}_M$, N output fmaps $\mathbf{y}_1, ..., \mathbf{y}_N$, and non-linearity f.

$$\mathbf{y}_n = f(\sum_{m=1}^{M} \mathbf{x}_m * \mathbf{w}_{n,m} + b_n) \qquad (1)$$

The parameters of this conv layer are $M \times N \times K \times K$ weights and N biases.

Pooling layers maps each input fmap to an output fmap whose every pixel is the max/mean of a $K \times K$ window of input pixels. Unlike conv layers the windows do not overlap, and the output fmaps are K times smaller in each dimension. Pooling layers are inserted throughout a CNN to gradually reduce the size of the intermediate feature maps.

Dense or fully-connected (FC) layers take an input vector of 1×1 feature maps (pixels) and perform a dot product with a weight vector. The result is added to a bias and passed through a non-linearity to produce a single 1×1 output. Equation (2) below shows the operation of a FC layer with M input pixels, N output pixels, and non-linearity f.

$$y_n = f(\sum_{m=1}^{M} x_m w_{n,m} + b_n) \qquad (2)$$

The parameters are $M \times N$ weights and N biases.

2.2 Binarized Neural Networks

A BNN is essentially a CNN whose weights and fmap pixels are binarized to -1 or +1; they can be seen as an extreme example of the quantized, reduced-precision CNN models commonly used for hardware acceleration. In this paper we focus on an architecture developed by Courbariaux et al. in [6] and later refined in [7]. The first paper binarizes only the weights while the follow-up binarizes both weights and fmaps. We focus on the latter version and refer to it as the BinaryNet architecture/model. This architecture achieves near state-of-the-art results on both CIFAR-10 and SVHN datasets at time of publication. Other more recent work on low precision networks promise accuracy close to state of the arts on ImageNet [16, 28].

In the BinaryNet model, the weights and outputs of both conv and FC layers are binarized using the Sign function (i.e., positive weights are set to +1 and negatives to -1). Figure 1 illustrates the flow of data through a conv and pooling layer in both a CNN and a BNN. For the CNN, the order of operations matches Equation (1) and the fmaps are real-valued at all times. In the BNN, the feature maps go from binary to integer (after convolution) until it is binarized

again. Biases have been removed (see Section 3.1). Pooling in the BNN is always performed on the integer data.

The BNN also introduces a new layer type — **Batch normalization** [11] layers reduce the information lost during binarization by linearly shifting and scaling the input distribution to have zero mean and unit variance. This reduces quantization error compared to an arbitrary input distribution. [1] The transformation is given in Equation (3) below,

$$y = \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}} \gamma + \beta \qquad (3)$$

where x and y are input and output, respectively, μ and σ are statistics collected over the training set, γ and β are trained parameters, and ϵ is to avoid round-off problems. During inference, all parameters are fixed, so we need only be concerned with efficiently applying Equation (3) to each input fmap pixel. Each output fmap require its own set of batch norm parameters.

The primary advantages of BNNs over their higher precision counterparts are twofold:

1. The convolution operation in Equation (1) (which nominally requires a $K \times K$ element multiply-accumulate) can now be implemented as a bitwise XNOR between two $K \times K$ bit vectors and a popcount. This is highly relevant to FPGA design, as these operations can be implemented very efficiently in the logic fabric.

2. Assuming comparable numbers of feature maps and FC layer units, binarizing weights and fmaps greatly reduces their memory size. This is again compelling for FPGAs as existing FPGA accelerators are typically constrained in performance by a combination of on-chip storage space and off-chip memory bandwidth.

2.3 CIFAR-10 BNN Model

The CIFAR-10 dataset [14] contains sixty thousand 32×32 3-channel images consisting of photos taken of real world vehicles and animals. The images come from 10 classes (airplane, truck, cat, etc.) and are divided into a training set of 50000 and a test set of 10000.

The BinaryNet architecture consists of six conv layers followed by three FC layers. All conv layers use 3×3 filters and edge padding, and all conv/FC layers apply batch norm before binarization. There is a 2×2 max pooling layer after the 2nd, 4th, and 6th conv layers. The first conv layer is different from the rest: its input is the image, which is floating-point, not binary; its weights are still binary. The architecture is summarized in Table 1; the size of the fmaps gets smaller deeper into the network, and that the first two dense layers contain most of the weights.

---

[1] Batch normalization can also speed up training and regularize the activations in full-precision CNNs, but this is beyond the scope of this paper.

Layer	Input Fmaps	Output Fmaps	Output Dim	Output Bits	Weight Bits
Conv1	3	128	32	128K	3456
Conv2	128	128	32	128K	144K
Pool	128	128	16	32K	
Conv3	128	256	16	64K	288K
Conv4	256	256	16	64K	576K
Pool	256	256	8	16K	
Conv5	256	512	8	32K	1.1M
Conv6	512	512	8	32K	2.3M
Pool	512	512	4	8192	
FC1	8192	1024	1	1024	8.0M
FC2	1024	1024	1	1024	1.0M
FC3	1024	10	1	10	10K
Total					13.4M
Conv					4.36M
FC					9.01M

Table 1: **Architecture of the BinaryNet CIFAR-10 BNN** — The weight bits exclude batch norm parameters, whose total size after optimization (see Section 3.1) is 0.12M bits, less than 1% of the size of the weights.

Training of the CIFAR-10 BNN model was done using open-source Python code provided by Courbariaux et al. [2], which uses the Theano and Lasagne deep learning frameworks. We reached 11.58% test error out-of-the-box, in line with their results. Their paper also presents more advanced training techniques such as stochastic binarization, which further reduce error rate. We did not use them in this work. Different training schemes do not affect the inference pass or the compatibility of our accelerator.

3. FPGA Accelerator Design

In this section, we first outline how we optimize the BinaryNet model for hardware, then describe the design of our system and the specific compute units.

3.1 Hardware Optimized BNN Model

As with the design of conventional CNN accelerators, a key optimization we made to the BNN model is parameter quantization. While the weights are already binarized, the biases and batch norm parameters are real numbers. During bias quantization, we noticed that nearly every bias was much smaller than 1. Given that the inputs have magnitude 1, we tried setting the biases to zero and observed no effect on accuracy. We then retrained the network with biases removed from the model, and reached a test error of 11.32%. For the rest of the paper we use this as the baseline error rate.

A second optimization involved noting that the batch norm calculation (Equation (3)) is a linear transformation, and can thus be formulated as $y = kx + h$, where:

$$k = \frac{\gamma}{\sqrt{\sigma^2 + \epsilon}} \qquad \text{and} \qquad h = \beta - \frac{\mu\gamma}{\sqrt{\sigma^2 + \epsilon}} \qquad (4)$$

[2] https://github.com/MatthieuCourbariaux/BinaryNet

This reduces the number of operations and cuts the number of stored parameters to two. Furthermore, the BNN always binarizes immediately after batch norm. Thus we do not need the magnitude of y, only the sign, allowing us scale k and h by any multiplicative constant. We exploit this property during quantization by scaling each k and h to be within the representable range of our fixed-point implementation. Empirical testing showed that k and h can be quantized to 16 bits with negligible accuracy loss while being a good fit for power-of-2 word sizes. We also quantized the floating point BNN inputs to 20-bit fixed point. Table 2 summarizes the impact of each algorithmic modification on test error. The HLS accelerator has the same accuracy as the C++ code.

3.2 Retraining for +1 Edge-Padding

One complication in the BinaryNet model is the interaction between binarization and edge padding. The model binarizes each activation to -1 or +1, but each input fmap is edge padded with zeros, meaning that a convolution can see up to 3 values: -1, 0, or +1. Thus the BinaryNet model actually requires some 2-bit operators (though the fmap data can still be stored in binary form). We managed to modify and retrain the BinaryNet model to pad with +1, eliminating the zeros and creating a truly binarized CNN. This +1 padded BNN achieves a test error of 11.82% in Python and 12.27% in C++/FPGA, only slightly worse than the original.

For our FPGA implementation we used the 0 padded BNN as the resource savings of the +1 padded version was not particularly relevant for the target device.

Source	Model	Padding	Test Error
From [7]	-	0	11.40%
Python	Default	0	11.58%
Python	no-bias	0	11.32%
Python	no-bias	+1	11.82%
C++	no-bias, fixed-point	0	11.46%
C++	no-bias, fixed-point	+1	12.27%

Table 2: **Accuracy of the BNN with various changes** — `no-bias` refers to retraining after removing biases from all layers and `fixed-point` refers to quantization of the inputs and batch norm parameters.

3.3 System Architecture

Our system architecture, shown in Figure 2(a), consists of three compute units, data and weight buffers, a direct memory access (DMA) system for off-chip memory transfer, and an FSM controller. The three compute units work on different types of layers: the *FP-Conv* unit for the (non-binary) first conv layer, the *Bin-Conv* unit for the five binary conv layers, and the *Bin-FC* unit for the three binary FC layers. Of the three, the Bin-Conv and Bin-FC units must handle different numbers of input and output fmaps, and (in the case of Bin-Conv) different fmap sizes.
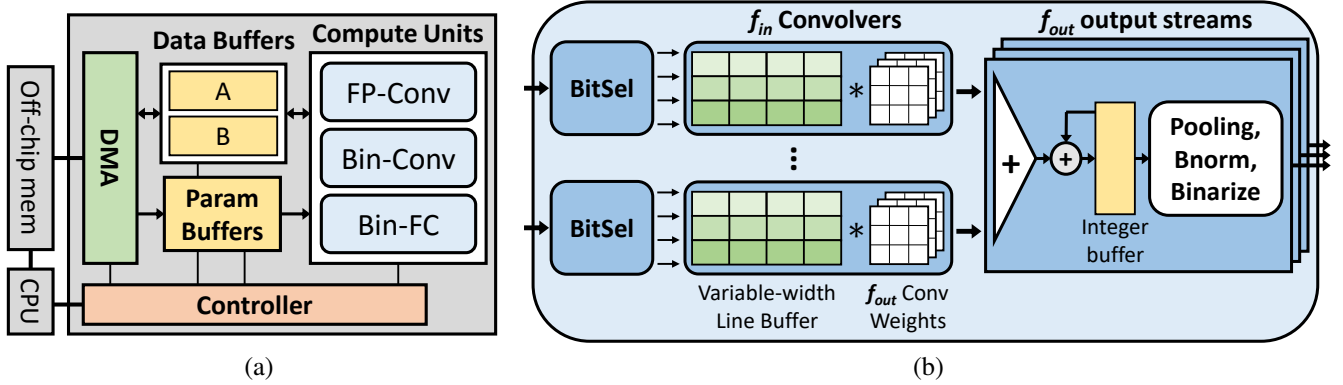
Figure 2: **Architectural diagrams of our BNN accelerator** — (a) system-level block diagram showing the three compute units, buffers, and how the accelerator is connected to the CPU and off-chip memory hierarchy; (b) architecture of the Bin-Conv unit with input and output parallelization factors $f_{in} = 2$ and $f_{out} = 3$. The unit can stream in two words per cycle and produce three output fmaps per invocation.

The storage of intermediate data in our accelerator differs from most existing designs. In full-precision CNN accelerators, the size of a single set of fmaps between two layers typically exceed the size of FPGA on-chip storage. This necessitates the continuous transfer of fmaps to and from off-chip RAM. However, as Table 1 shows, the size of the largest set of fmaps in our BNN is only 128K bits, which easily fits on-chip even in smaller FPGAs. Our design uses two in-out data buffers A and B of equal size. One layer reads from A and write its outputs to B; then (without any off-chip data transfers) the next layer can read from B and write to A. Thus, off-chip memory transfers are only needed for the input image, output prediction, and loading each layer's weights.

Unlike the fmaps, there is only enough memory on-chip to store a portion of a layer's weights. Multiple accelerator invocations may be needed for a layer; in each invocation we load in a new set of weights and produce a new set of fmaps. The next invocation produces the next set of fmaps, and etc, until all output fmaps have been generated and stored in the on-chip data buffer. Invoking the accelerator requires passing it arguments such as pointers to the weights, the layer type and size, the fmap size, and whether pooling should be applied. Inside the accelerator, the controller decodes these inputs and coordinates the other modules.

3.4 Compute Unit Architectures

In our accelerator, each compute unit must store *binarized* data to the on-chip RAMs at the end of its execution. As Figure 1 reveals, the first operation of a conv or FC layer transforms the binary inputs to integers; we make sure each unit will also perform the subsequent batch-norm, pooling, and binarization before writing data out to the buffers. One of our design goals is to limit the amount of integer-valued intermediate data buffered inside each compute unit,

FP-Conv — The fixed-point conv unit utilizes the well-known line buffer architecture for 2D convolutions. Because this unit only targets a single layer, we hardwire it to handle

a 3-channel 32×32 input. While the input pixels are 20-bit fixed-point, the weights are binarized, so we can replace the multiplies in the conv operation with sign inversions. We fully parallelize across the three input channels: each cycle we stream in three input pixels, add them to three line buffers, and compute a 3×3×3 convolution. The result is put through batch norm and binarization to produce one output bit per cycle. Greater parallelism in this unit is achievable, but the first conv layer takes up a very small portion of the overall runtime, and we focused our efforts elsewhere.

Bin-Conv — The binary conv unit is the most critical component of the accelerator, as it will be responsible for the five binary conv layers which take up the vast majority of the runtime. The unit must maintain high throughput and resource efficiency while handling different input widths *at runtime*; our design targets 8, 16, or 32, and can support larger power-of-two widths with minor changes. To efficiently compute a convolution, multiple rows of input pixels need to be buffered for simultaneous access. However, a standard line buffer (i.e., from video processing applications) is unsuitable for this task due to two reasons:

1. A line buffer must be sized for the largest input fmap; in this case it must have a width of 32. This not only causes buffer under-utilization when the fmap is 16 or 8 wide, it also leads to loss of throughput, as we can only perform as many convolutions per cycle as the input width.

2. A line buffer is designed to shift one pixel per cycle to always store the most recent rows. However, with binarized in puts we have access to not one but many lines of new pixels each cycle (for instance, a 32-bit word can hold 4 lines from an 8×8 fmap). This radically changes how we should update the line buffer.

In full precision CNN accelerators, the size problem can be addressed by tiling, where input fmaps are processed in tiles which are always the same size. This is unsuitable in a BNN since the fmaps are typically very small in terms of number
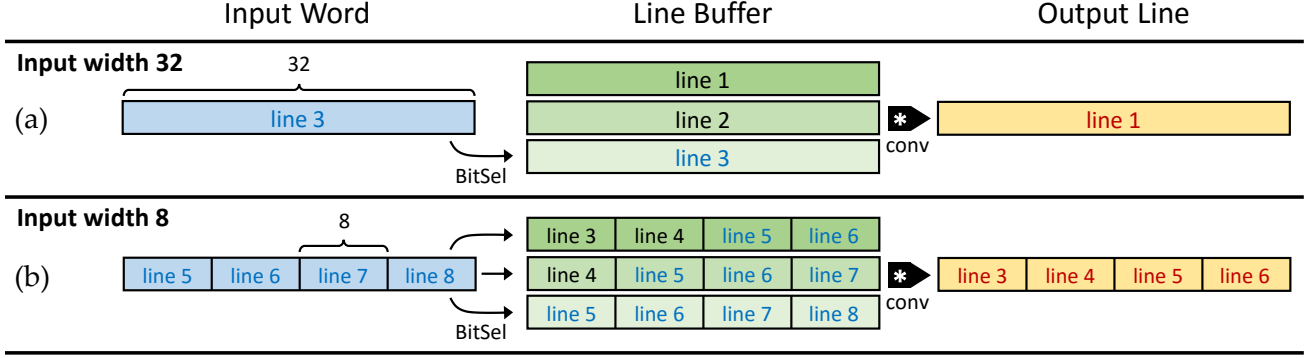
19

Figure 3: **Example usage of the variable-width line buffer** — we show how a 32-bit input word is divided up by BitSel and inserted into the VWLB. The line buffer has height 3 and width 32. Sliding a 3×3 conv filter across the VWLB produces 32 output pixels, ignoring edge effects. (a) for a 32-wide input fmap, each row of the VWLB stores one line and applying the conv filter produces one 32-wide output line; (b) for an 8-wide input fmap, each row of the VWLB stores four lines and applying the conv filter produces four consecutive 8-wide output lines given the mapping of input lines to banks shown.

of bits. One possible solution to the second problem is to reorganize the data so each bit in an input word comes from a different fmap, and assign each bit to a separate line buffer. However, this requires an exorbitant number of line buffers and is not area efficient.

To address the above issues, we introduce two new modules: the *BitSel* module and the *variable-width line buffer* (VWLB). Figure 2(b) shows the basic structure of the Bin-Conv unit, whose execution proceeds in two phases. In the first phase, input fmaps from the on-chip buffers are streamed in on the left side, through the BitSel modules, and into the VWLBs. The Convolver modules compute the partial conv sums and accumulates them in the integer buffers. The BitSel is responsible for reordering the input bits so that the Convolver logic can be agnostic of the fmap width. In Figure 2(b), f_{in} is the input parallelization factor — the Bin-Conv unit accepts f_{in} input words per cycle and the data buffers are partitioned to match this rate. f_{out} is the output parallelization factor — each Convolver applies f_{out} 3×3 conv filters per cycle to the data in the VWLB and generates partial sums for f_{out} different fmaps. The first phase ends when all input fmaps in the current layer have been processed. At this point each integer buffer contains a finished conv map. In the second phase we compute max pooling, batch norm, and binarization to produce f_{out} binary output fmaps. Note that max-pooling and binarization are nonlinear operations, so we cannot apply them to partially finished conv maps and accumulate afterwards.

Figure 3 explains the operation of the BitSel and VWLB in greater detail. The diagram assumes we have a word size of 32 bits and a 3×3 conv filter, which requires a VWLB with three rows and 32 elements per row. We demonstrate how the VWLB works for input fmap widths 32 and 8 and ignore edge padding for the sake of simplicity.

1. For a 32-wide input, each word contains exactly one line. Each cycle, the VWLB shifts up and the new 32-bit line

is written to the bottom row. We can then slide the 3×3 conv window across the VWLB to generate one 32-bit line of conv outputs.

2. For an 8-wide input, each word contains four lines. We split each VWLB row into four banks, and map each input line to *one or more* VWLB banks. The mapping is done in such a way that sliding the conv window across the VWLB produces four consecutive 8-bit output lines. Each cycle the VWLB shifts both up and to the left.

The BitSel is responsible for slicing the input word and mapping the slices to the row banks. Because the smallest input width is 8, each slice and VWLB bank is sized at 8 bits. For a 32-wide input, BitSel maps four contiguous 8-bit slices to the bottom row. For an 8-wide input, the mapping is more complex, but still highly regular and can be computed in hardware with just adds and shifts. Each pixel in the output lines in Figure 3 is an integer conv sum, and each sum is accumulated at a different location in the integer buffer.

The BitSel and VWLB provides three primary advantages: (1) the VWLB achieves full hardware utilization regardless of input width, (2) a new input word can be buffered every cycle, and (3) the BitSel deals with various input widths by itself, allowing the actual buffer and convolution logic to be fixed. Note that the VWLB used in our design differs from Figure 3 in a few details. First, we have neglected edge padding. The actual VWLB contains two additional elements per bank to hold horizontal pad bits; vertical padding is handled by inserting lines of zeros. Second, because the pad bits are 0 rather than +1 or -1, we must make each element in the VWLB two bits instead of one. The conv operation is performed between the 2-bit data and 1-bit weights, and can be implemented as sign inversion and accumulate.

Bin-FC — The binary FC unit is comparatively simple. Each cycle we read in f_{in} data words and an equal number of weight words. f_{in} here is the input parallelization factor

just as in Bin-Conv. Because there is no edge padding in an FC layer the computations can be truly binary. We perform a dot product between the data and weight words by applying a bitwise XOR operation and then summing the resulting bits with a popcount. Similar to the Bin-Conv unit, we accumulate the sum in an integer buffer and apply binarization after all inputs have been processed. Note that the FC layers are typically bound by memory bandwidth of the off-chip connection, rather than the throughput of the accelerator.

Data Buffers — To accommodate multiple reads per cycle, the data buffers are partitioned into f_{in} banks, and feature maps are interleaved across the different banks. Figure 4 shows an example with $f_{in} = 2$ and four words per fmap. The data words are read sequentially by address, so a compute unit always accesses f_{in} consecutive fmaps in parallel.
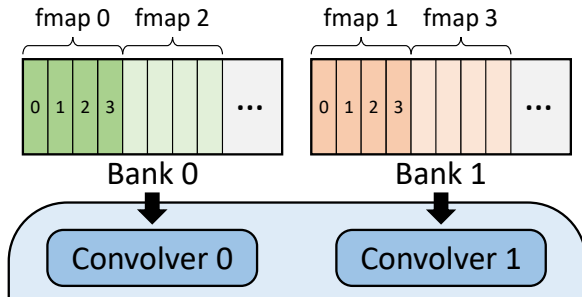

Figure 4: **Example of data buffer banking** — The compute unit and memory system have $f_{in} = 2$. Each fmap contains four words which are laid out sequentially. The fmaps are interleaved across banks, and both Bin-Conv and Bin-FC benefit from this banking.

4. HLS Accelerator Implementation

Figure 5 shows the HLS pseudocode for the front half of the Bin-Con unit, and demonstrates a key difference between BNN and CNN hardware design. For a CNN the code typically loops over an fmap processing one pixel at a time; key design decisions include loop ordering and unroll factors (see [27] for a good example). In our BNN accelerator, the basic atom of processing is not a pixel but a word. The example code is designed to sustain one word per cycle throughput over the entire input feature map set. Each fmap consists of `words_per_fmap` words, a number which differs between layers. As it processes the input set, the code updates the weights on each new fmap and accumulates the conv results in `outbuf`. We call `BitSel` and `conv` inside the loop to instantiate the BitSel units and conv logic as shown in Figure 2(b). To increase the number of input streams we can tile the loop and unroll the inner loop body.

A key design decision here is the input word size, which controls the level of parallelism across the pixels of an fmap. To guarantee correctness, `words_per_fmap` must be an integer greater than zero; this constrains the word size to at most the size of the smallest input fmap ($8 \times 8 = 64$ bits in our case). The word size restriction is not a significant limiting

```
1  VariableLineBuffer linebuf;
2  ConvWeights wts;
3  IntegerBuffer outbuf;
4
5  for (i = 0; i < n_input_words; i++) {
6    #pragma HLS pipeline
7
8    // read input word, update linebuffer
9    WordType word = input_data[i];
10   BitSel(linebuf, word, input_width);
11
12   // update the weights each time we
13   // begin to process a new fmap
14   if (i % words_per_fmap == 0)
15     wts = weights[i / words_per_fmap];
16
17   // perform conv across linebuffer
18   for (c = 0; c < LINE_BUF_COLS; c++) {
19   #pragma HLS unroll
20     outbuf[i % words_per_fmap][c] +=
21       conv(c, linebuf, wts);
22   }
23 }
```

Figure 5: **HLS pseudocode for part of the Bin-Conv unit** — the pseudocode implements a pipeline which reads and performs convolution on one input word each cycle. Many details are left out; the goal is to illustrate how our design can be expressed in high-level code.

factor in our design, as 64 is already a very large parallelization factor (it means we perform 64 convolutions per cycle), and there are other sources of parallelism to exploit in the BNN. We chose a word size of 64 bits for the data buffers and sized each data buffer A and B at 2048 words, which is just enough to store the largest set of fmaps in the BNN.

We also explored different values for f_{in} and f_{out} in Bin-Conv. It was observed that both have roughly similar effects on execution time, but increasing f_{out} has a more severe effect on total area. f_{in} controls the number of BitSels and VWLBs while f_{out} controls the number of pooling/batch norm units and integer buffers. In terms of logic a BitSel and a pooling/batch norm unit is similar, but each VWLB contains 32×3 2-bit registers while each integer buffer contains 32×32 12-bit registers. Thus all else being equal it is better to increase f_{in}. This result shows the importance of minimizing the storage of intermediate values and only committing binarized data to memory.

We use Xilinx SDSoC as the primary design tool for our BNN application. SDSoC takes as input a software program with certain functions marked as "hardware". It invokes Vivado HLS under the hood to synthesize the "hardware" portion into RTL. In addition, it automatically generates the data motion network and DMA necessary for memory transfer between CPU and FPGA based on the specified software-hardware partitioning. We selected a DMA engine built for contiguous memory since it has the highest throughput, and

a neural network's data and weights can be laid out contiguously. We used directives to ensure that data is only transferred on the first and last accelerator invocation; weights are transferred on every invocation.

5. Experimental Results

We evaluate our design on a ZedBoard, which uses a low-cost Xilinx Zynq-7000 SoC containing an XC7Z020 FPGA alongside an ARM Cortex-A9 embedded processor. We make use of Xilinx SDSoC 2016.1 as the primary design tool, which leverages Vivado HLS and Vivado to perform the actual HLS compilation and FPGA implementation. We compared our design against two server-class computing platforms: an Intel Xeon E5-2640 multicore processor (**CPU**) and an NVIDIA Tesla K40 GPU (**GPU**). We also compared against an NVIDIA Jetson TK1 embedded GPU board (**mGPU**). As BNNs are a recent development, our baseline applications will not be as well optimized compared to CNN baselines (where implementations can be found in frameworks such as Caffe). The CPU and GPU baselines are adapted from code provided in [7]. The code leverages Theano, and calls OpenBLAS for CPU and CUDA for GPU. However, it does not perform bitwise optimizations since they are not natively supported in Theano, and instead uses floating-point values binarized to -1 and +1. For the baselines we used the BNN model with no biases and k and h, and on the GPU we always used the largest batch size.

Power measurement is obtained via a power monitor. We measured 4.5W idle and 4.7W max power on the Zedboard power suppy line when running our BNN. This indicates the dynamic power consumption of the FPGA is very low.

Table 3: **Comparison of different configurations —** Last row shows the resources available on the device; **Runtime** is in milliseconds. * indicates our chosen configuration.

f_{in}	LUT	FF	BRAM	DSP	Runtime
1	25289	28197	86	3	17.5
2	35291	37125	87	3	10.8
4	38906	36771	87	3	7.98
8*	46900	46134	94	3	5.94
Dev.	53200	106400	140	220	-

Table 3 shows the performance and resource utilization of our accelerator using different values of f_{in} for the Bin-Conv and Bin-FC units. All numbers are post place and route. In our experiments f_{out} is set to 1 for reasons outlined in Section 4. Performance scaling is clear, though the scaling is less than unity due to memory transfer and other overheads. We use $f_{in} = 8$ for the rest of the experiments.

We compare the performance of our accelerator to the various baselines in Table 4. As raw throughput depends heavily on device size, we also show the power consumption and the throughput per Watt. The FPGA design ob-

tains 15.1x better performance and 11.6x better throughput per Watt over **mGPU**, which has a similar power envelope. Against the x86 processor, it achieves a 2.5x speedup. While the binary conv layers were faster, the FC layers were slower, which is unsurprising as the FC layers are bound by external memory bandwidth. Versus **GPU**, the FPGA is 8.1x worse in performance. But as expected, it has much lower power consumption and better throughput per Watt.

To show that the FC layers are indeed limited by memory bandwidth, we created a design where the FC computations are removed but the memory transfers are kept. The new execution time of the FC layers is within 5% that of the original, demonstrating that there is not much to gain by further parallelizing them beyond the current design.

Table 4: **Performance comparison — Conv1** is the first FP conv layer, **Conv2-5** are the binary conv layers, **FC1-3** are the FC layers. A – indicates a value we could not measure. Numbers with * are sourced from datasheets. The last row shows power efficiency in throughput per Watt.

	Execution time per image (ms)			
	mGPU	**CPU**	**GPU**	**FPGA**
Conv1	–	0.68	0.01	1.13
Conv2-5	–	13.2	0.68	2.68
FC1-3	–	0.92	0.04	2.13
Total	90	14.8	0.73	5.94
Speedup	1.0x	6.1x	123x	15.1x
Power (Watt)	3.6	95*	235*	4.7
imgs/sec/Watt	3.09	0.71	5.83	35.8

Table 5 compares our implementation against state-of-the-art FPGA accelerators found in literature — all numbers are retrieved from the respective papers. Note that two of the comparisons are against larger FPGAs while one is against the same device. Throughput is shown in giga-operations-per-second (GOPS), and we count adds and multiplies following [27]: each binary xor, negation, or addition counts as one operation. Our BNN accelerator beats the best known FPGA accelerators in pure throughput, and is also much more resource and power efficient. BNNs save especially on the number of DSPs since multiplication/division is only needed for batch norm and not for the compute-intensive conv or FC calculations. The metrics of throughput per kLUT and throughput per Watt are especially important — due to the relative novelty of the BNN, we were only able to obtain a suitable network for CIFAR-10 while previous work shows results for larger ImageNet networks. However, our data provides evidence that the BNN is algorithmically better suited for FPGA than CNN, enabling far more efficient usage of resource and power. With a more advanced network and larger device, our design should scale up and achieve similar gains. Very recent work on low-precision

CNNs have also made great strides into achieving near state-of-the-art accuracy on the ImageNet dataset [16, 28].

Table 5: **Comparison of our work against state-of-the-art FPGA accelerators** — GOPS counts multiplies and adds per second. * indicates values approximated from charts.

	[23]	[19]	[25]	Ours
Platform	Stratix-V GSD8	Zynq 7Z045	Zynq 7Z020	Zynq 7Z020
Capacity (kLUTs)	695	218.6	53.2	53.2
Clock(MHz)	120	150	100	143
Power(W)	19.1	9.6	-	4.7
Precision	8-16b	16b	-	1-2b
GOPS (conv)	136.5	187.8	-	318.9
GOPS (all)	117.8	137.0	12.73	207.8
kLUTs	120*	182.6	43.2	46.9
DSPs	760*	780	208	3
GOPS/ kLUT	0.98	0.75	0.29	4.43
GOPS/ Watt	6.17	14.3	7.27	44.2

While it may not be completely fair to compare GOPS between a binarized and conventional network, it is currently the (de facto) standard practice for hardware accelerator studies to compare reduced and full-precision implementations that use different data types.

6. Related Work

Our paper owes much to the groundbreaking work on BNNs in the machine learning community [6, 7]. These papers contain some discussion on the advantages of BNNs over CNN for hardware, but to our best knowledge we are the first to present a working FPGA implementation.

There have been many studies on the design of CNN accelerators for FPGA. Zhang et al. [27] describe how to optimize an HLS design by reordering and tiling loops, inserting the proper pragmas, and organizing external memory transfers. Ensuing publications have mostly eschewed HLS in favor of RTL designs. Qiu et al. [19] propose an architecture that computes conv and FC layers on the same hardware, as well as dynamic fixed-point quantization. Their paper demonstrates an area-efficient accelerator for AlexNet on the Xilinx ZC706 board.

A related line of research focuses on creating CNN design compilers which can generate optimized hardware for a family of models. These works typically use a set of RTL modules combined with a design space exploration tool to find the optimal architectural parameters. Rahman et al. [20] propose a scalable array-based CNN accelerator with heavy input reuse. Motamedi [17] uses a roofline model for performance to guide hardware generation. Wang [26] proposes DeepBurning, which targets a variety of CNN architectures and performs data layout optimization.

OpenCL frameworks for deep learning on FPGA have also been proposed. Suda et al. [23] use parameterized OpenCL alongside analytical models for performance and resource, enabling a genetic algorithm to search for the optimal configuration. Venieris and Bouganis [25] study the use of synchronous dataflow to capture CNN workloads and use graph partitioning to control resource consumption.

There is also a great deal of research work on ASIC CNN co-processors. Among the most well know is the DianNao line of architectures [2]. The Eyeriss paper [3] contains a comprehensive study of popular dataflows for spatial architectures and derive an optimal one.

Our approach differs from existing work in two major ways: (1) we are the first to study BNNs for FPGA acceleration; (2) we make use of a C-based HLS methodology and propose design constructs to maximize throughput on different layers. Existing CNN accelerators on FPGA are not well-equipped to handle BNNs due significant differences in compute and storage requirements, and layer organization. Our final design differs greatly from previous work.

7. Conclusions and Future Work

We are the first to implement an accelerator for binarized neural networks on FPGA. BNNs feature potentially reduced storage requirements and binary arithmetic operations, making them well suited to the FPGA fabric. However, these characteristics also render CNN design constructs such as input tiles and line buffers ineffective. We introduce new design constructs such as a variable-width line buffer to address these challenges, creating an accelerator radically different from existing work. We leverage modern HLS tools to write our design in productive, high-level code, and our accelerator outperforms existing work in raw throughput, throughput per area, and throughput per Watt.

Future BNN work should focus both on algorithmic and architectural improvements. On the algorithmic side we would like to explore techniques to reduce model size. From the architectural side one action item is to implement a low-precision network for ImageNet, which would involve a much larger and more complicated accelerator design.

Acknowledgements

References

[1] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, T. Czajkowski, S. D. Brown, and J. H. Anderson. LegUp: An Open-Source High-Level Synthesis Tool for FPGA-Based Processor/Accelerator Systems. *ACM Trans. on Embedded Computing Systems (TECS)*, 13(2):24, 2013.

[2] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam. Diannao: A Small-Footprint High-Throughput Accelerator for Ubiquitous Machine-earning. *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Mar 2014.

[3] Y.-H. Chen, T. Krishna, J. Emer, and V. Sze. Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2016.

[4] A. Coates, B. Huval, T. Wang, D. J. Wu, A. Y. Ng, and B. Catanzaro. Deep Learning with COTS HPC Systems. *Int'l Conf. on Machine Learning (ICML)*, pages 1337–1345, Jun 2013.

[5] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang. High-Level Synthesis for FPGAs: From Prototyping to Deployment. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, Apr 2011.

[6] M. Courbariaux, Y. Bengio, and J.-P. David. BinaryConnect: Training Deep Neural Networks with binary weights during propagations. *Advances in Neural Information Processing Systems (NIPS)*, pages 3123–3131, 2015.

[7] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio. Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1. *arXiv e-print*, arXiv:1602.02830, Feb 2016.

[8] T. S. Czajkowski, U. Aydonat, D. Denisenko, J. Freeman, M. Kinsner, D. Neto, J. Wong, P. Yiannacouras, and D. P. Singh. From OpenCL to High-Performance Hardware on FPGAs. *Int'l Conf. on Field Programmable Logic and Applications (FPL)*, pages 531–534, Aug 2012.

[9] M. A. et al. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems, 2015. Software available from tensorflow.org.

[10] K. He, X. Zhang, S. Ren, and J. Sun. Deep Residual Learning for Image Recognition. *arXiv e-print*, arXiv:1512.0338, Dec 2015.

[11] S. Ioffe and C. Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *arXiv e-print*, arXiv:1502.03167, Mar 2015.

[12] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint*, arXiv:1408.5093, 2014.

[13] V. Kathail, J. Hwang, W. Sun, Y. Chobe, T. Shui, and J. Carrillo. SDSoC: A Higher-level Programming Environment for Zynq SoC and Ultrascale+ MPSoC. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, pages 4–4, Feb 2016.

[14] A. Krizhevsky and G. Hinton. Learning Multiple Layers of Features from Tiny Images, 2009. Master's Thesis. Department of Coumputer Science, University of Toronto.

[15] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet Classification with Deep Convolutional Neural Networks. *Advances in Neural Information Processing Systems (NIPS)*, pages 1097–1105, 2012.

[16] F. Li and B. Liu. Ternary Weight Networks. *arXiv e-print*, arXiv:1605.04711, May 2016.

[17] M. Motamedi, P. Gysel, V. Akella, and S. Ghiasi. Design Space Exploration of FPGA-Based Deep Convolutional Neural Networks. *Asia and South Pacific Design Automation Conf. (ASP-DAC)*, pages 575–580, Jan 2016.

[18] K. Ovtcharov, O. Ruwase, J.-Y. Kim, J. Fowers, K. Strauss, and E. Chung. Accelerating Deep Convolutional Neural Networks Using Specialized Hardware. *Microsoft Research*, Feb 2015.

[19] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song, et al. Going Deeper with Embedded FPGA Platform for Convolutional Neural Network. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, pages 26–35, Feb 2016.

[20] A. Rahman, J. Lee, and K. Choi. Efficient FPGA Acceleration of Convolutional Neural Networks using Logical-3D Compute Array. *Design, Automation, and Test in Europe (DATE)*, pages 1393–1398, Apr 2016.

[21] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi. XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks. *European Conference on Computer Vision (ECCV)*, Oct 2016. arXiv:1603.05279.

[22] K. Simonyan and A. Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. *arXiv e-print*, arXiv:1409.15568, Apr 2015.

[23] N. Suda, V. Chandra, G. Dasika, A. Mohanty, Y. Ma, S. Vrudhula, J.-s. Seo, and Y. Cao. Throughput-Optimal OpenCL-based FPGA Accelerator for Large-Scale Convolutional Neural Networks. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, pages 16–25, Feb 2016.

[24] Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-print*, arXiv:1605.02688, May 2016.

[25] S. I. Venieris and C.-S. Bouganis. fpgaConvNet: A Framework for Mapping Convolutional Neural Networks on FPGAs. *IEEE Symp. on Field Programmable Custom Computing Machines (FCCM)*, May 2016.

[26] Y. Wang, J. Xu, Y. Han, H. Li, and X. Li. DeepBurning: Automatic Generation of FPGA-based Learning Accelerators for the Neural Network Family. *Design Automation Conf. (DAC)*, page 110, Jun 2016.

[27] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong. Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, pages 161–170, Feb 2015.

[28] S. Zhou, Y. Wu, Z. Ni, X. Zhou, H. Wen, and Y. Zou. DoReFar-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients. *arXiv e-print*, arXiv:1606.06160, Jul 2016.