HSIM-DNN: Hardware Simulator for Computation-, Storage- and Power-Efficient Deep Neural Networks

Mengshu Sun
Northeastern University
Boston, Massachusetts
sun.meng@husky.neu.edu

Pu Zhao
Northeastern University
Boston, Massachusetts
zhao.pu@husky.neu.edu

Yanzhi Wang
Northeastern University
Boston, Massachusetts
yanz.wang@northeastern.edu

Naehyuck Chang
Korea Advanced Institute of Science and Technology
Daejeon, South Korea
naehyuck@cad4x.kaist.ac.kr

Xue Lin
Northeastern University
Boston, Massachusetts
xue.lin@northeastern.edu

ABSTRACT

Deep learning that utilizes large-scale deep neural networks (DNNs) is effective in automatic high-level feature extraction but also computation and memory intensive. Constructing DNNs using block-circulant matrices can simultaneously achieve hardware acceleration and model compression while maintaining high accuracy. This paper proposes HSIM-DNN, an accurate hardware simulator on the C++ platform, to simulate the exact behavior of DNN hardware implementations and thereby facilitate the block-circulant matrix-based design of DNN training and inference procedures in hardware. Real FPGA implementations validate the simulator with various circulant block sizes and data bit lengths taking into account accuracy, compression ratio and power consumption, which provides excellent insights for hardware design.

CCS CONCEPTS

• **Computing methodologies** → **Neural networks**; • **Hardware** → **Emerging simulation**; • **Software and its engineering** → **Simulator / interpreter**;

KEYWORDS

Deep neural network; hardware simulator; block-circulant matrix; FFT; hardware acceleration

ACM Reference format:
Mengshu Sun, Pu Zhao, Yanzhi Wang, Naehyuck Chang, and Xue Lin. 2019. HSIM-DNN: Hardware Simulator for Computation-, Storage- and Power-Efficient Deep Neural Networks. In *Proceedings of Great Lakes Symposium on VLSI 2019, Tysons Corner, VA, USA, May 9–11, 2019 (GLSVLSI '19),* 6 pages.
https://doi.org/10.1145/3299874.3317996

GLSVLSI '19, May 9–11, 2019, Tysons Corner, VA, USA

ACM ISBN 978-1-4503-6252-8/19/05…$15.00
https://doi.org/10.1145/3299874.3317996

1 INTRODUCTION

Deep learning based on deep neural networks (DNNs) has experienced breakthroughs in recent years in broad areas, such as computer vision [11, 18] and speech recognition [14]. Deep learning applications have been migrating to smart objects like mobile and Internet-of-Things (IoT) devices [1, 20], which can compose the smart world and ubiquitous intelligence. Modern DNNs tend to consist of multiple cascaded layers with at least millions of parameters [18] that contribute to high computation accuracy through high-level feature extraction. Nonetheless, the growing model size also increases the computation and memory complexity, which becomes the key challenge of DNN applications especially in smart world. Aiming for higher scalability, performance and energy efficiency in deep learning systems, two contradictory trends in research and development for DNNs have emerged, namely *hardware acceleration* and *model compression*.

Hardware acceleration based on FPGAs or ASICs yields higher performance and better energy efficiency than that based on general-purpose computing systems like CPUs and GPUs. Representative works include Google's Tensor Processing Units (TPUs) [16], IBM's TrueNorth chips [9, 22], and optimization related to memory transfers [3, 4], dynamic-precision data quantization [23], processing dataflow [5], weight sharing [10], bit-serial computation [17], etc. Most designs of this type suffer from frequent accesses to off-chip DRAM, which may consume over 100× energy than capacity-limited on-chip SRAM[10, 12, 13], and easily dominate power consumption of the entire system.

Model compression has been realized by several algorithm level techniques, including weight quantization [7, 21, 26, 27], connection pruning [12, 13], and low rank approximation [15, 25]. A special case of quantization is the low precision fixed-point representation [21], which is prevalent in hardware like FPGAs and ASICs, only using one or two bits for representation in extreme cases [7, 27]. Connection pruning removes all connections with weights below a threshold [12, 13], and low rank approximation represents CNN filter banks using a low rank basis of filters that are separable in the spatial domain [15, 25]. Although these approaches offer reasonable parameter reductions with less data storage and transfer and minor accuracy loss, they have respective limitations. Quantization achieves the compression ratio in a fixed-point form but

requires the gradients to be denoted in floating-point numbers in the training process. Pruning typically causes the network structure to be irregular, thereby limiting the compression ratio and performance, and both pruning and low rank approximation are likely to increase the training complexity. Moreover, the compression ratios achieved by all these approaches are heuristic and cannot be precisely controlled.

A distinct method in [6] explores parameter redundancy of DNNs by imposing a circulant structure for weight representation of fully-connected layers. This work proposes corresponding accelerated inference and training algorithms for fully-connected layers. Such a method is generalized for both fully-connected layers and convolution layers in [8], based on block-circulant weight matrices considering the trade-off between the accuracy degradation and compression ratio or computation acceleration. It provides a cross-platform hardware design and optimization solution for deep learning systems.

DNN hardware implementations are constrained by limited hardware resources, i.e., the computing and storage units in FPGAs or the tapeout area in ASICs, and it is time- and labor-consuming work to fit the DNN architectures into available resources while meeting the accuracy and power requirements. On the other hand, software simulators for deep learning, such as Caffe and Tensorflow, could not produce exact hardware results that are restricted by precision and computing resources in hardware implementations. Therefore, this paper proposes HSIM-DNN, an accurate hardware simulator that simulates the behavior of DNN hardware implementations for both training and inference procedures on the C++ platform. This simulator accelerates the design and optimization cycles of DNN hardware implementations by testing and checking the feasibility of the hyperparameters (variables) in the hardware design. It thus serves as a transition from pure software descriptions to hardware description languages like VHDL and Verilog.

The main features of HSIM-DNN are as follows:

1) HSIM-DNN produces fast and exact results with high accuracy for DNN hardware implementations, and detects the existence of data overflow of computations in hardware, allowing to avoid overflow and improve the design at an early stage.

2) HSIM-DNN provides the energy/power performance under different parameter configurations, and serves as the starting point for future hardware platform-specific functional modules, such as power and timing estimators.

3) HSIM-DNN supports online training of DNNs in hardware by analyzing and verifying the feasibility and performance, which is currently a technical difficulty for DNN hardware implementations.

2 BACKGROUND

2.1 Basics of Deep Neural Networks

DNNs may take various architectures, whereas they are all constructed by cascading multiple functional layers for feature extraction at multiple abstraction levels. The most fundamental types of functional layers in DNNs are *fully-connected layers*, *convolution layers* and *pooling layers*.

Fully-connected (FC) layers are the most computation- and storage-intensive layers in DNNs [10, 23], due to the fully-connectedness between adjacent layers. The computation for an FC layer consists of matrix-vector arithmetic (including multiplication and addition) and activation transformation, described as

$$\mathbf{y} = \psi(\mathbf{W}\mathbf{x} + \theta), \tag{1}$$

where $\mathbf{x}$ is the input of the FC layer, $\mathbf{y}$ is the output, and $\mathbf{W} \in \mathbb{R}^{m \times n}$ is the *weight matrix* of the synapses (i.e., connections) between the m neurons in the FC layer and the n neurons in the previous layer, $\theta \in \mathbb{R}^m$ is the *bias vector*, and $\psi(\cdot)$ is the *activation function*. The multiplication calculation $\mathbf{W}\mathbf{x}$ determines the total computational complexity, and the remaining calculation in this equation contributes to a lower complexity of $O(n)$. The most widely utilized activation function in DNN applications is the Rectified Linear Unit (ReLU), given by $\psi(x) = \max(0, x)$.

Convolution (CONV) layers extract features from the input data with a set of *kernels* (i.e., *filters*) [19] mainly by multi-dimensional convolutions, and the output *feature maps* are fed into the subsequent layers for higher-level feature extraction. The input data of a CONV layer can be images or feature maps from the previous layer. Specifically, the convolution of an input map and a kernel is generated by moving the kernel through the input map. With multiple kernels, a CONV layer is often associated with multiple input and multiple output feature maps. The CONV computation can be expressed in the form of *tensor computation*, as

$$\mathcal{Y}(a, b, d) = \sum_{i=1}^{r} \sum_{j=1}^{r} \sum_{c=1}^{C} \mathcal{F}(i, j, c, d)\mathcal{X}(a + i - 1, b + j - 1, c), \tag{2}$$

where $\mathcal{X} \in \mathbb{R}^{W \times H \times C}$, $\mathcal{Y} \in \mathbb{R}^{(W-r+1) \times (H-r+1) \times D}$, $\mathcal{F} \in \mathbb{R}^{r \times r \times C \times D}$ represent the input, output, and *weight tensors* of the CONV layer, respectively. There are C input maps and D output maps, and each kernel of size $r \times r$ is applied to each input map with the spatial dimensions W and H to generate an output map.

Pooling (POOL) layers subsample the extracted features to reduce data dimensions and mitigate overfitting problems. The dominant pooling strategy in state-of-the-art DNN applications is *max pooling* because of the high overall accuracy and high convergence speed [4, 5].

Among these three types of layers, CONV and FC layers are responsible for the majority of computation, and a POOL layer has a lower computational complexity of $O(n)$. The storage requirement of DNNs originates from the weight matrices $\mathbf{W}$'s for FC layers and the convolution kernels represented by the tensors $\mathcal{F}$'s for CONV layers. Therefore, FC and CONV layers become the major research focus for efficient DNN implementations.

2.2 Block-Circulant Matrix-Based Weight Representation

A block-circulant matrix is constructed by arraying equal-sized square circulant sub-matrices in two dimensions, each circulant matrix being considered as a block. A circulant matrix with $k \times k$ elements has only k free elements, so the size of the block is defined as k, which indicates the compression degree. Particularly, there is no compression for blocks with $k = 1$. Fig. 1 gives an example of a block-circulant matrix $\mathbf{W}$ with the calculation process with regard to a certain circulant sub-matrix $\mathbf{W}_{ij}$. The calculation of $\mathbf{W}\mathbf{x}$ can then be reduced to the calculation of $\mathbf{W}_{ij}\mathbf{x}_j$, which can be performed through the "FFT → element-wise multiplication →

IFFT" procedure according to the *circulant convolution theorem* [2], which is elaborated in Section 3.

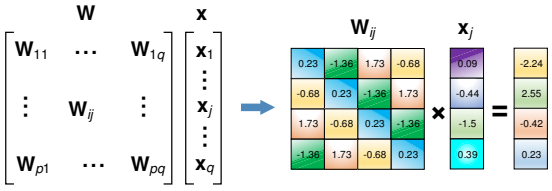

Figure 1: Illustration of a block-circulant matrix.

3 SIMULATOR IMPLEMENTATION

In the proposed simulator HSIM-DNN, the block size (size of the circulant matrix) and data precision (bit length) are considered as two fundamental hardware optimization variables that are chosen to balance the accuracy, the model compression ratio and the hardware utility rate. A higher block size results in a greater compression ratio with more computation acceleration but degrades the accuracy. Increase in the bit length improves the accuracy but requires more storage and computation resources.

The computation and validation process of HSIM-DNN is illustrated in Fig. 2. The image data and weights are first preprocessed, i.e., they are quantized according to the bit length and segmented based on the block size. After that, the FFT-based circulant multiplication is done for all circulant blocks. The outputs from the blocks are then reformulated to generate the overall result, which is checked on the correctness and overflow to give a feedback to adjust the block size and bit length. The software validation provides insights into the hardware design with much less effort and time compared with hardware validation.

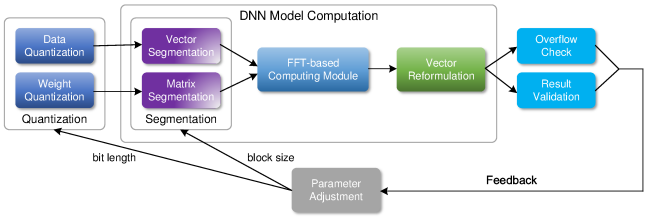

Figure 2: Computation and validation of HSIM-DNN.

The data in hardware is represented in the form of fixed-point binary numbers, i.e., in 2's complement, in order to fit the limited hardware resources. Therefore, the simulator contains a data conversion module to convert decimals to fixed-point binary numbers, and then it manages all operations based on fixed-point binary numbers to simulate the actual calculations in DNN hardware implementations. A fixed-point binary number consists of an integral part and a fractional part. More integral bits can represent data in a wider range and avoid overflow, and more fractional bits can provide higher precision and better accuracy. An overflow detection mechanism is added to report if the data value exceeds the acceptable data range and leads to computation errors.

The simulation procedure of HSIM-DNN has two main phases, simulating DNN training and DNN inference in hardware, respectively. The FFT-based computing module serves as the core calculation kernel for both phases.

3.1 FFT-Based Computing Module

As mentioned in Section 2.2, the calculation of $\mathbf{W}_{ij}\mathbf{x}_j$ can be performed through the "FFT → element-wise multiplication → IFFT" procedure, which is the basic computation in FC and CONV layers in both training and inference. This procedure can be expressed as

$$\mathbf{W}_{ij}\mathbf{x}_j = \text{IFFT}(\text{FFT}(\mathbf{w}_{ij}) \circ \text{FFT}(\mathbf{x}_j)), \tag{3}$$

where $\mathbf{w}_{ij}$ denotes the first row vector of $\mathbf{W}_{ij}$, and $\circ$ denotes element-wise multiplication. An FFT-based computing module is designed for HSIM-DNN to execute this calculation as shown in Fig. 3. This module can be split into an FFT submodule and an IFFT submodule, the latter based on the former, as an IFFT can be realized by an FFT with conjugate operations on the input and output.

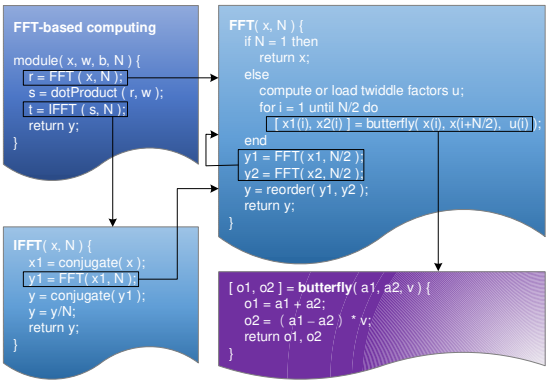

Figure 3: Algorithm of FFT-based computing module.

A recursive FFT is implemented by the radix-2 decimation in frequency (DIF) algorithm as illustrated in Fig. 4. More specifically, an n-size FFT with n inputs and n outputs can be separated into two FFTs each with size $n/2$ and one additional level of butterfly calculation, and an $n/2$-size FFT can be further divided into two $n/4$-size FFTs with one butterfly calculation level. A large-scale FFT can be calculated by recursively performing FFTs on the same computing module, and multiple small-scale FFTs can be calculated on several computing modules in parallel. This recursive property of FFT calculation is the key for hardware to accommodate DNN models of different types and sizes on relatively small hardware footprints with appropriate control logic and memory storage organization. Compared with the FFT computation, the rest of computations including component-wise multiplication, ReLU activation and pooling have little time complexity and memory overhead, which can be ignored.

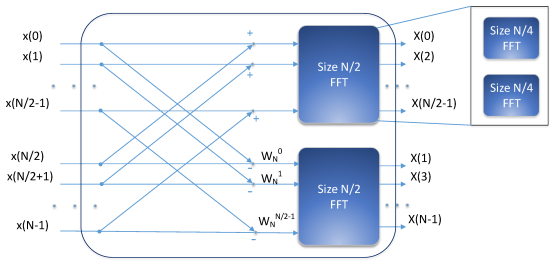

Figure 4: Recursive FFT computation.

3.2 Inference of HSIM-DNN

HSIM-DNN applies block-circulant matrix-based calculation in FC and CONV layers in the training and inference phases to mitigate the computation and storage burdens owing to the large amount of matrix-vector multiplication. The simulator first trains the DNN model through forward-propagation and backward-propagation to optimize parameters, and then it performs inference to make predictions for new input data. The inference phase is first introduced since it is similar to the forward-propagation step in training.

3.2.1 Inference of FC layers. The original weight matrix $\mathbf{W} \in \mathbb{R}^{m\times n}$ of an FC layer can have an arbitrary size. To obtain a block-circulant weight matrix, $\mathbf{W}$ is partitioned into equal-sized square sub-matrices, i.e., blocks. Each block $\mathbf{W}_{ij}$ is a circulant matrix with $k \times k$ elements (the block size is k), and $\mathbf{W}$ contains $p \times q$ blocks where $p = \lceil m/k \rceil$ and $q = \lceil n/k \rceil$. If the division of m or n has a remainder, zero padding should be applied for $\mathbf{W}$ to make every block square. The weight matrix with partition is represented as

$$\mathbf{W} = [\mathbf{W}_{ij}], \quad i \in \{1, \ldots, p\}, j \in \{1, \ldots, q\}. \tag{4}$$

The input vector $\mathbf{x}$ is also partitioned into q vectors each with size k, represented as $\mathbf{x} = [\mathbf{x}_1^{\mathrm{T}}, \mathbf{x}_2^{\mathrm{T}}, \ldots, \mathbf{x}_q]^{\mathrm{T}}$, correspondingly. Thus the original matrix-vector multiplication $\mathbf{Wx}$ can be replaced with multiple sub-matrix-vector multiplication calculations. The inference process in an FC layer, omitting the bias and ReLU components in (1), is described as

$$\mathbf{a} = \mathbf{Wx} = \begin{bmatrix} \sum_{j=1}^{q} \mathbf{W}_{1j}\mathbf{x}_j \\ \sum_{j=1}^{q} \mathbf{W}_{2j}\mathbf{x}_j \\ \cdots \\ \sum_{j=1}^{q} \mathbf{W}_{pj}\mathbf{x}_j \end{bmatrix} = \begin{bmatrix} \mathbf{a}_1 \\ \mathbf{a}_2 \\ \cdots \\ \mathbf{a}_p, \end{bmatrix}, \tag{5}$$

where $\mathbf{a}_i \in \mathbb{R}^k$ is a column vector. Furthermore, each block $\mathbf{W}_{ij}$ can be denoted only by its first row $\mathbf{w}_{ij}$, and the calculation of $\mathbf{W}_{ij}\mathbf{x}_j$ can be implemented by (3).

3.2.2 Inference of CONV layers. For a CONV layer that is associated with multiple input and output feature maps, the convolution computation might be executed in multi-dimensions, namely in tensor form, as mentioned in (2). The concept of block-circulant structure is here generalized to a rank-4 tensor $\mathcal{F}$, with all slices of the form $\mathcal{F}(i, j, \cdot, \cdot)$ being circulant matrices. The tensor needs to be reshaped into a 2D matrix to fit the block-circulant matrix-based method. Specifically, the tensor computation in (2) is reformulated as the matrix multiplication $\mathbf{Y} = \mathbf{XF}$, where $\mathbf{X} \in \mathbb{R}^{(W-r+1)(H-r+1)\times Cr^2}$, $\mathbf{Y} \in \mathbb{R}^{(W-r+1)(H-r+1)\times D}$ and $\mathbf{F} \in \mathbb{R}^{Cr^2\times D}$. The element $\mathcal{F}(i, j, c, d)$ is reformulated as $f_{C(i-1)+Cr(j-1)+c,d}$ in $\mathbf{F}$, making $\mathbf{F}$ a block-circulant matrix. Hence the fast multiplication approach for block-circulant matrices can be applied to implement $\mathbf{Y} = \mathbf{XF}$, leading to computation acceleration.

3.2.3 Complexity Analysis. An FFT of size k has computational complexity of $O(k \log k)$. For a block $\mathbf{W}_{ij}$ of block size k, the replacement of the original matrix-vector multiplication by the FFT-based computation can reduce the computational complexity from $O(k^2)$ to $O(k \log k)$. In addition, only the first row vector $\mathbf{w}_{ij}$ or FFT($\mathbf{w}_{ij}$) requires to be stored for each circulant matrix $\mathbf{W}_{ij}$, reducing the memory complexity from $O(k^2)$ to $O(k)$.

Provided an $m \times n$ block-circulant matrix with the block size of k, the computational complexity of an FC layer is $O(pqk \log k)$, which is equivalent to $O(n \log n)$ (assuming $n > m$) when the values of p and q are small. Similarly, the memory complexity is $O(pqk)$, equivalent to $O(n)$ for small p and q values. The computational complexity of a CONV layer is also reduced by the block-circulant matrix-based method, from $O(WHr^2CD)$ to $O(WHQ \log Q)$, where $Q = \max(Cr^2, P)$. It can be seen that both computation acceleration and model parameter compression are obtained.

3.3 Training of HSIM-DNN

The purpose of training is to optimize the DNN parameters including weights and biases to minimize the loss, namely the gap between the prediction results from the DNN and the original correct results given by the training dataset. In image classification problems, the original results are labels indicating the category to which each image belongs. The training process consists of two steps, i.e., forward propagation and backward propagation. Forward propagation is similar to the inference process, where the DNN receives input image data and makes a prediction, and the only difference from inference is that a loss function is evaluated by comparing the results from forward propagation and the original labels. Backward propagation, also called backpropagation, calculates the derivatives of the loss with respect to weights and biases, which are then updated sequentially from the last layer to the first layer. Forward propagation and backward propagation are performed alternately until the loss decreases to an enough small value. The forward and backward processes are shown in Fig. 5.

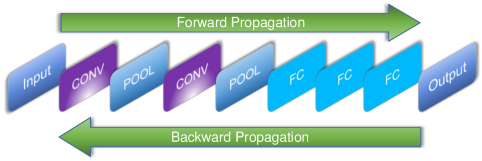

Figure 5: Forward and backward propagation in a multi-layer DNN.

Letting the loss function be denoted by L, the weight matrix $\mathbf{W}$ is updated in the backward propagation as

$$\mathbf{W} = \mathbf{W} - \epsilon \frac{\partial L}{\partial \mathbf{W}}, \tag{6}$$

where ϵ denotes the learning rate. Subtracting the derivative of L with respect to $\mathbf{W}$ moves $\mathbf{W}$ to decrease the loss and thus improve the accuracy. Usually the derivative $\frac{\partial L}{\partial \mathbf{W}}$ cannot be computed directly, so the chain rule would be applied to obtain the derivatives sequentially from the last layer to the first layer, expressed as

$$\frac{\partial L}{\partial \mathbf{w}_{ij}} = \frac{\partial L}{\partial \mathbf{a}_i}\frac{\partial \mathbf{a}_i}{\partial \mathbf{w}_{ij}} = \sum_{l=1}^{k} \frac{\partial L}{\partial a_{il}}\frac{\partial a_{il}}{\partial \mathbf{w}_{ij}}, \tag{7}$$

$$\frac{\partial L}{\partial \mathbf{x}_j} = \sum_{i=1}^{p} \frac{\partial L}{\partial \mathbf{a}_i}\frac{\partial \mathbf{a}_i}{\partial \mathbf{x}_j} = \sum_{i=1}^{p}\sum_{l=1}^{k} \frac{\partial L}{\partial a_{il}}\frac{\partial a_{il}}{\partial \mathbf{x}_j}, \tag{8}$$

where a_{il} represents the l-th output element in $\mathbf{a}_i$. Since $\mathbf{W}_{ij}$ is a circulant matrix, $\frac{\partial \mathbf{a}_i}{\partial \mathbf{w}_{ij}}$ and $\frac{\partial \mathbf{a}_i}{\partial \mathbf{x}_j}$ are also circulant matrices, defined respectively by the base vector $\mathbf{x}'_j = [x_{j,1}, x_{j,k}, x_{j,k-1}, \ldots, x_{j,2}]'$,

and the base vector of $\mathbf{W}_{ij}$. And then $\frac{\partial L}{\partial \mathbf{w}_{ij}}$ and $\frac{\partial L}{\partial \mathbf{x}_j}$ can be calculated using the "FFT → element-wise multiplication → IFFT" procedure to reduce the computational complexity of each layer. The vectors $\mathbf{w}_{ij}$'s, composing the circulant blocks $\mathbf{W}_{ij}$'s of each layer, are trained directly so that the trained network naturally follows the block-circulant structure. The key advantage is that no additional steps are required in training and inference.

3.3.1 Training Acceleration. The FFT-based computing module is improved to enhance the simulator performance in the following three steps: (a) *Encapsulating the FFTW++ library* [24], which provides a simple interface for FFTs of 1D, 2D, and 3D complex-to-complex, real-to-complex, and complex-to-real Fast Fourier Transforms with automation of memory allocation, alignment, wisdom, and communication on both serial and parallel architectures and with support in in-place and out-of-place multithreaded transforms of arbitrary size. (b) *Changing the structure of FFT implementation* to perform FFTs iteratively instead of recursively to save overhead of implicit calls on the stack and to enable fine-grained memory management. (c) *Implementing multi-threads or multi-cores* to compute the inference and training individually in each batch.

3.4 Energy/Power Performance

The energy/power performance of DNN hardware implementations is another important concern in addition to the computation speed and accuracy, and it is affected by two optimization variables, the block size and bit length, as well as the hardware device type. Overall, large-scale devices dissipate more power, and they are needed to accommodate large block sizes and long bit lengths, which add more to the power dissipation. The power dissipation of FPGA devices is composed of dynamic power dissipation, static power and I/O power, with the first two terms as the main components. To assist the DNN implementations, the power dissipation is measured with several typical combinations of block sizes and bit lengths.

4 EXPERIMENTAL RESULTS

The HSIM-DNN design is tested for each module in DNN implementations in the Altera Cyclone V GT FPGA development kit, to validate the accuracy, compression ratio and power performance of the simulator. The training and inference of HSIM-DNN adopt the representative benchmarks MNIST and CIFAR-10. The effects of varying the block size and bit length are compared between the proposed simulator and the original uncompressed DNN models based on the software framework Caffe.

4.1 Results of MNIST and CIFAR-10 Datasets

Table 1 displays the accuracy and compression ratio of designed with different block sizes, with the integral bit length fixed at 8 and the fractional bit length fixed at 12. The compression ratios are obtained by comparing with the uncompressed DNN models in software with 64-bit floating-point data. The degradation in accuracy and increase of compression ratio can be observed as the block size increases. Specifically, compression for the block size of 32 is around 100× with accuracy degradation of 1.6%.

Table 2 describes the accuracy and compression ratio when optimizing the bit length and fixing the block size at 64. Generally, increasing the bit length will improve the accuracy. Overflow can occur when the integral bits are not enough, e.g., when the bit length is (6, 8), accompanied by a low accuracy. The low accuracy with bit length (7, 4) originates from the low fractional precision with 4 bits.

Table 1: Comparison on various block sizes for MNIST

Block Size	Accuracy	Compression Ratio
1	97.9%	3.2
32	96.3%	98.9
64	95.2%	191.0
128	93.3%	357.5
256	92.1%	633.8

Table 2: Comparison on various bit lengths for MNIST

Integral Bits + Fractional Bits	Accuracy	Compression Ratio
(6, 8) (overflow)	91.5%	272.8
(7, 4)	77.8%	347.3
(7, 5)	94.4%	318.4
(7, 8)	95.0%	254.7
(7, 10)	95.1%	224.7
(8, 10)	95.1%	212.2
(8, 12)	95.2%	191.0

Furthermore, the storage space for weights can be reduced by using less bits to represent weights in storage. Table 3 shows that 15 bits in computation can be reduced to 8 bits in storage with slight accuracy degradation.

Table 3: Accuracy for storage with less bits

Bit Length for Computation	Bit Length for Storage	Accuracy	Compression Ratio
(6, 6) (overflow)	(5,3)	92.0%	381.0
(6, 8) (overflow)	(5,3)	91.2%	444.5
(7, 8)	(5,3)	94.4%	476.3

The results from CIFAR-10 dataset with different bit lengths and fixed block size of 128 are given in Table 4. Longer bit length also results in better accuracy.

Table 4: Comparison on various bit lengths for CIFAR-10

Integral Bits + Fractional Bits	Accuracy	Compression Ratio
(4, 8)	77.8%	453.3
(5, 8)	78.0%	378.5
(5, 10)	80.2%	317.7
(5, 11)	80.5%	287.5

4.2 Results of Power Measuring

The power data of DNN hardware implementations are obtained from tests on the same Cyclone V FPGA device, with the dataset CIFAR-10 as the benchmark. This process can be generalized to different datasets on various platforms. The Cyclone V devices

dissipate much less static power than other Altera devices like Stratix V, which helps to implement DNN applications with low power consumption. The power measuring results with varying block sizes and bit lengths are described in Table 5. Without loss of generality, the integral bit length is fixed at 4, and the fractional bit length changes from 6 to 9, as the accuracy and power performance are more sensitive to the fractional part.

Table 5: Power Comparison on various block sizes and bit lengths for CIFAR-10 (Unit: mW)

Bit Length \ Block Size	32	64	128
(4, 6)	1103.07	1322.59	1899.50
(4, 7)	1180.63	1494.39	2158.22
(4, 8)	1244.73	1640.22	2347.46
(4, 9)	1289.36	1739.61	n/a
(4, 10)	1328.70	1845.50	n/a

The total power dissipation of the Cyclone V device can be approximated as

$$P_{tot} = 0.1647 + 0.06064 \cdot t + 9.294 \times 10^{-4} \cdot s \cdot t \quad (9)$$

where s is the block size and t the total bit length of integral bits plus fractional bits. The last term of this equation implies the correlation of the block size and the bit length. If one of the two variables is fixed at some value, the total power will grow almost linearly with the other variable.

5 CONCLUSION

This work implements a hardware simulator HSIM-DNN that visualizes the exact behavior of DNN hardware implementations on a software platform to facilitate the corresponding block-circulant matrix-based approach in terms of acceleration and compression for both training and inference phases. Simultaneously, the proposed simulator provides the hardware power data to help predict the power consumption under different parameter settings, thus creating excellent insights for more energy-efficient DNN implementations on embedded platforms. Real FPGA hardware implementations validate the whole design.

ACKNOWLEDGMENTS

This work is supported by the National Science Foundation CCF-1733701.

REFERENCES

[1] Sourav Bhattacharya and Nicholas D Lane. 2016. From smart to deep: Robust activity recognition on smartwatches using deep learning. In *Proceedings of the IEEE International Conference on Pervasive Computing and Communication Workshops (PerCom Workshops)*. 1–6.

[2] Dario Bini and Victor Y Pan. 2012. *Polynomial and Matrix Computations: Fundamental Algorithms*. Springer Science & Business Media.

[3] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. 2014. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. *ACM Sigplan Notices* 49, 4 (2014), 269–284.

[4] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, et al. 2014. DaDianNao: A Machine-Learning Supercomputer. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. 609–622.

[5] Yu-Hsin Chen, Tushar Krishna, Joel S Emer, and Vivienne Sze. 2017. Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks. *IEEE Journal of Solid-State Circuits* 52, 1 (2017), 127–138.

[6] Yu Cheng, Felix X Yu, Rogerio S Feris, Sanjiv Kumar, Alok Choudhary, and Shi-Fu Chang. 2015. An Exploration of Parameter Redundancy in Deep Networks with Circulant Projections. In *Proceedings of the IEEE International Conference on Computer Vision*. 2857–2865.

[7] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2016. BinaryNet: Training Neural Networks with Weights and Activations Constrained to +1 or âĹŠ1. *CoRR* abs/1602.02830 (2016).

[8] Caiwen Ding, Siyu Liao, Yanzhi Wang, Zhe Li, Ning Liu, Youwei Zhuo, Chao Wang, Xuehai Qian, Yu Bai, Geng Yuan, et al. 2017. CirCNN: Accelerating and Compressing Deep Neural Networks Using Block-circulant Weight Matrices. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. 395–408.

[9] Steve K Esser, Rathinakumar Appuswamy, Paul Merolla, John V Arthur, and Dharmendra S Modha. 2015. Backpropagation for Energy-Efficient Neuromorphic Computing. In *Proceedings of Neural Information Processing Systems*. 1117–1125.

[10] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. 2016. EIE: Efficient Inference Engine on Compressed Deep Neural Network. In *Proceedings of the 43rd Annual International Symposium on Computer Architecture*. 243–254.

[11] Pu Zhao, Sijia Liu, Yanzhi Wang, and Xue Lin. 2018. An ADMM-Based Universal Framework for Adversarial Attacks on Deep Neural Networks. In *Proceedings of the 26th ACM International Conference on Multimedia*. 1065–1073.

[12] Song Han, Huizi Mao, and William J Dally. 2015. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. *CoRR* abs/1510.00149 (2015).

[13] Song Han, Jeff Pool, John Tran, and William Dally. 2015. Learning both Weights and Connections for Efficient Neural Networks. In *Proceedings of the 28th International Conference on Neural Information Processing Systems*. 1135–1143.

[14] Geoffrey Hinton, Li Deng, Dong Yu, George E Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N Sainath, et al. 2012. Deep Neural Networks for Acoustic Modeling in Speech Recognition: The Shared Views of Four Research Groups. *IEEE Signal Processing Magazine* 29, 6 (2012), 82–97.

[15] Max Jaderberg, Andrea Vedaldi, and Andrew Zisserman. 2014. Speeding up Convolutional Neural Networks with Low Rank Expansions. *CoRR* abs/1405.3866 (2014).

[16] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*. 1–12.

[17] Patrick Judd, Jorge Albericio, Tayler Hetherington, Tor M Aamodt, and Andreas Moshovos. 2016. Stripes: Bit-serial deep neural network computing. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture*. 1–12.

[18] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems*. 1097–1105.

[19] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324.

[20] He Li, Kaoru Ota, and Mianxiong Dong. 2018. Learning IoT in Edge: Deep Learning for the Internet of Things with Edge Computing. *IEEE Network* 32, 1 (2018), 96–101.

[21] Darryl Lin, Sachin Talathi, and Sreekanth Annapureddy. 2016. Fixed Point Quantization of Deep Convolutional Networks. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning*. 2849–2858.

[22] Paul A Merolla, John V Arthur, Rodrigo Alvarez-Icaza, Andrew S Cassidy, Jun Sawada, Filipp Akopyan, Bryan L Jackson, Nabil Imam, Chen Guo, Yutaka Nakamura, et al. 2014. A million spiking-neuron integrated circuit with a scalable communication network and interface. *Science* 345, 6197 (2014), 668–673.

[23] Jiantao Qiu, Jie Wang, Song Yao, Kaiyuan Guo, Boxun Li, Erjin Zhou, Jincheng Yu, Tianqi Tang, Ningyi Xu, Sen Song, et al. 2016. Going Deeper with Embedded FPGA Platform for Convolutional Neural Network. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 26–35.

[24] SourceForge. 2010. FFTW++: Fast Fourier Transform C++ Header/MPI Transpose for FFTW3. http://fftwpp.sourceforge.net/

[25] Cheng Tai, Tong Xiao, Yi Zhang, Xiaogang Wang, and Weinan E. 2015. Convolutional neural networks with low-rank regularization. *CoRR* abs/1511.06067 (2015).

[26] Jiaxiang Wu, Cong Leng, Yuhang Wang, Qinghao Hu, and Jian Cheng. 2016. Quantized Convolutional Neural Networks for Mobile Devices. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 4820–4828.

[27] Chenzhuo Zhu, Song Han, Huizi Mao, and William J Dally. 2016. Trained Ternary Quantization. *CoRR* abs/1612.01064 (2016).