Accelerating Sparse Deep Neural Networks on FPGAs

Sitao Huang*, Carl Pearson*, Rakesh Nagi*, Jinjun Xiong†, Deming Chen*, Wen-mei Hwu*
*University of Illinois at Urbana-Champaign, †IBM Research
{shuang91, pearson, nagi, dchen, w-hwu}@illinois.edu, jinjun@us.ibm.com

***Abstract*—Deep neural networks (DNNs) have been widely adopted in many domains, including computer vision, natural language processing, and medical care. Recent research reveals that sparsity in DNN parameters can be exploited to reduce inference computational complexity and improve network quality. However, sparsity also introduces irregularity and extra complexity in data processing, which make the accelerator design challenging. This work presents the design and implementation of a highly flexible sparse DNN inference accelerator on FPGA. Our proposed inference engine can be easily configured to be used in both mobile computing and high-performance computing scenarios. Evaluation shows our proposed inference engine effectively accelerates sparse DNNs and outperforms CPU solution by up to $4.7\times$ in terms of energy efficiency.**

***Index Terms*—Deep learning, Sparse DNN, Graphs, FPGA**

I. Introduction

Recent years have witnessed the success of deep learning in many domains, including computer vision, natural language processing, medical care, autonomous driving and so on [1] [2]. The extraordinary high accuracy of deep learning based approaches is made possible by performing inference using pre-trained deep neural network (DNN) models, which have very high computation and memory space demands. This complexity presents a significant challenge to adopting DNNs for many real-world applications, especially edge-computing scenarios, which have stringent power and latency requirements for computation. Researchers have invested significant effort in making efficient low-computational-cost DNN based systems possible. The research efforts mainly fall into three aspects: designing light-weight DNNs, reducing the amount of computation in DNN without sacrificing accuracy, and accelerating DNNs with customized hardware.

Recent research reveals that many parameters in deep neural networks are redundant and can be pruned away. Parameter pruning reduces the number of parameters and the amount of computation; after parameter pruning, the parameters in DNN layers become sparse. However, sparsity in DNN layers also introduces irregularity and extra complexity from sparse data formats and scheduling computation workload. It is challenging for the processors and accelerators to handle the irregularity and extra complexity which may lead to non-negligible overhead in execution time. The overhead diminishes the benefits from sparsity and may even result in worse performance compared to non-sparse approaches if not handled properly.

As deep learning conquers more and more complicated cognitive computing tasks, the size and complexity of DNN architectures explodes. Practitioners realize that it is getting harder and harder to achieve performance and power efficiency targets for deep learning systems with CPU and GPU, and specialized deep learning accelerators are needed. Many deep learning accelerators have been proposed and they have all kinds of optimization objectives [3] [4]. Adoption of specialized deep learning accelerators has made many challenging application scenarios possible, including intelligent wearable devices, real-time high-definition video processing systems, etc. FPGAs have been one of the ideal platforms for DNN acceleration as FPGAs provide the combination of low latency, high energy efficiency, and high reconfigurability, which make FPGAs adaptable to many application scenarios.

In this work, we design and build a configurable sparse DNN inference engine on an FPGA that accelerates the inference of sparse DNNs. We target very deep sparse DNNs that can be used as the backbone network for future complex cognitive tasks. These DNNs have many layers and there are many neurons inside a layer. Our proposed inference engine can be reconfigured and adopted in different FPGA platforms, depending on the available hardware resources and application requirements.

The contribution of this work can be summarized as follows:

- We design and build a configurable sparse DNN inference engine that is highly flexible and capable of processing different sizes of sparse DNNs.
- We propose several design optimization techniques for sparse DNN inference that are general enough to potentially benefit future works on sparse DNN acceleration.
- We model and analyze the computation of sparse DNNs, and we show how the accelerator design can be parameterized and how the accelerator design space looks like.

The rest of this paper is organized as follows. Section II provides background information on sparse DNNs and FPGA accelerators. Section III discusses our proposed accelerator design optimization techniques. Section IV presents the details of our proposed sparse DNN inference engine. Section V shows our experiment setups and results. Section VI reviews the recent works on similar areas. Finally, Section VII concludes the whole paper.

II. Background

A. Sparse Deep Neural Networks

In this work, we focus on feedforward deep neural networks that consist of fully connected layers. Note that our formulation below for fully connected layers can also be extended for sparse convolution neural networks (CNNs), since convolution

978-1-7281-5020-8/19/$31.00 ©2019 IEEE

can be reduced to matrix multiplication operations. Motivated by the Sparse DNN Graph Challenge [5], we consider a DNN with L layers. Assume each layer in the DNN has M neurons, i.e. the dimension of the input and output feature vectors of each layer is M.

Let $y_{l-1} = (y_{l-1,1}, y_{l-1,2}, \dots, y_{l-1,M})$ be a single input sample to the l-th layer, and y_l be the corresponding output from the l-th layer. y_0 is the input feature vector to the neural network, e.g. y_0 can be one input image. There can be N input samples, and these N input samples can be stacked as input matrix $\mathbf{Y}_0 = (y_0^{(1)}, y_0^{(2)}, \dots, y_0^{(N)})^\top$, where $y_0^{(i)}$ is a row vector and is the i-th input sample to the network. Similarly, the input and output of the l-th layer with multiple samples can be represented as matrices $\mathbf{Y}_{l-1}$ and $\mathbf{Y}_l$ respectively. Feature matrices $\mathbf{Y}$'s are $N \times M$ matrices.

The computation of the l-th layer can be formulated as

$$\mathbf{Y}_l = h(\mathbf{Y}_{l-1}\mathbf{W}_l + b_l), 1 \le l \le L \tag{1}$$

where $h(\cdot)$ is the ReLU function $h(x) = \max(0, x)$. $\mathbf{W}_l$ is a $M \times M$ matrix whose element $\mathbf{W}_l(i, j)$ at i-th row and j-th column represents the weight of the connection from the i-th input neuron to the j-th output neuron. b_l is the bias vector of dimension M.

$\mathbf{W}_l$ and b_l $(1 \le l \le L)$ are the parameters of the DNN which are determined by DNN training. y_0 is the input feature vector to the neural network. After DNN weight pruning, $\mathbf{W}$'s become sparse matrices. In some application, y_0 is also sparse. For example, in the hand-written digit recognition task (MNIST dataset), only a small subset of image pixels are black ("1") and the rest are white ("0"). In the problem setting of this work, all $\mathbf{W}$'s and y_0 are sparse.

B. FPGA Accelerators

FPGAs have been used in many application scenarios such as Internet-of-Things (IoT), wearable devices, autonomous driving, cloud computing, and scientific computing. Many different applications benefit from FPGA's low processing latency and high energy efficiency. Programming FPGAs has been a big challenge for decades, which prevents FPGA from being rapidly deployed and adopted in more domains. High-level synthesis (HLS) tools have greatly improved the productivity of FPGA designers. The C/C++/OpenCL to hardware description language (HDL) design flow enabled by HLS tools makes FPGA more accessible for designers.

III. Design Optimizations

This section presents the optimizations used in our sparse DNN accelerator. In this work, we use the test sparse networks provided by Graph Challenge [5]. The sparse DNN has 120 layers, each of which contains 1024 neurons. The optimization techniques presented here can be easily generalized to accelerate any sparse DNN.

A. Dense Feature Vectors and Sparse Parameters

As mentioned in Section II-A, both input image and DNN parameters are sparse. The input files of images and parameters are also in sparse format. The input files contains all the edges between neurons that have weights larger than 0. Therefore, the multiplication in Equation (1) is the multiplication of two sparse matrices, and one straightforward design would be implementing multiplication of two sparse matrices directly. In this way, the dot product of a sparse row vector and sparse column vector requires computing the set intersection of their indices. Since these sets are typically small (less than 1024 elements) , and the resource complexity of parallelizing the intersection is not trivial, the sequential comparison algorithm will be the most straightforward algorithm. Sequential comparison requires $\mathcal{O}(m + n)$ comparisons, where m and n are the number of non-zero elements in two vectors respectively.

However, we observe that treating both $\mathbf{Y}$ and $\mathbf{W}$ matrices sparse may not be optimal. In practice, the number of non-zero elements in the column vectors in the weight matrices are typically constrained. In the test data used in this work, the number of non-zero elements in each column of weight matrices $\mathbf{W}$'s is less than or equal to 32 (in 1024 neurons case). However, the sparsity of feature maps varies a lot. The number of non-zero elements in the row vectors in feature maps can vary from 0 to 1024 (in 1024 neurons case).

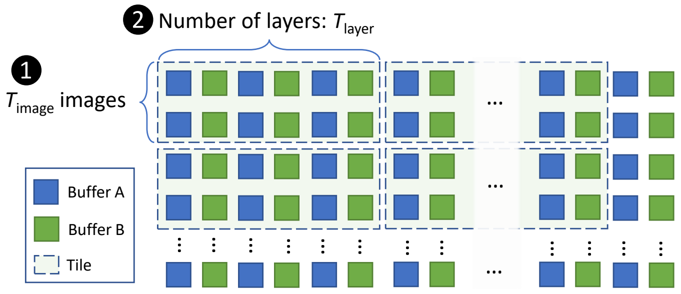
In this work, we treat the feature maps as dense matrices and the DNN parameters as sparse ones. In this way, the storage of DNN parameters is compact while access to parameter and feature maps is more efficient. With this dense-feature / sparse-parameter scheme, the multiplication of input feature vector and a column (weights of incoming edges to a neuron) in the parameter matrix can be done in a way shown in Listing 1. In Listing 1, `param` is the sparse representation of a column in parameter matrix, an array of pairs of indices and weights. The code iterates through `param` array, uses index to retrieve the feature vector `fvec` element and multiplies it with the corresponding weight. With this approach, there are `param.size()` random accesses into the feature vector array `fvec`. Note that the difference from regular dense matrix multiplication is that here the feature vector is randomly accessed as parameter arrays are sparse. This random access pattern is not friendly to memory access efficiency.

```
float sum = 0.0;
for(int i = 0; i < param.size(); i++) {
  sum += fvec[param[i].idx] * param[i].weight;
}
```

Listing 1. Feature vector `fvec` multiplied with a column `param` of the parameter matrix

B. Grid Representation and Data Dependencies

In order to better illustrate the data dependencies in the sparse DNN computation, we represent the computation of sparse DNN as a 2D grid, as shown in Fig. 1. Each row in the grid represents the computation on one input sample (one image). The i-th row represents the processing of the i-th input sample. The number of rows equals the number



❷ Number of layers: T_{layer}
❶
T_{image} images
Buffer A
Buffer B
Tile

Fig. 1. Tiling scheme

of input test images. Each column of the 2D grid represents the computation of a specific layer in the sparse DNN on all input samples' feature maps. The j-th column in the 2D grid represents the computation of all input samples' feature maps going through the j-th layer. The grid point at the i-th row and the j-th column represents the computation of the i-th input samples' feature maps goes through the j-th layer in the DNN. Each grid point includes sparse vector-matrix multiplication (Fig. 2), bias, and non-linear operations. Using the previous notation of M neurons, inside each grid point, the dimensions of vector and matrix in the sparse vector-matrix multiplication are M and $M \times M$ respectively.

The computation of rows of the 2D grid is independent from each other, since there is no data dependency between input samples. The computation of columns has dependency on the computation of the previous column. This is because the input to the next layer is the output from the previous layer. Within each grid point, the computation of inner products of vector and M columns of matrix is independent from each other.

C. Ping-Pong Buffering

In this work, we use very deep DNNs to test the performance of our system. The number of layers in the test DNNs varies from 120 to 1920. In order to keep track of the activations in this many layers, we use ping-pong buffers to store the input and the output feature maps. For example, when computing layer $2i$, buffer `buf_a` is used to store the input feature maps, while buffer `buf_b` is used to store the output feature maps. When processing layer $2i+1$, the roles of buffers `buf_a` and `buf_b` switch, the output feature map from layer $2i$ in buffer `buf_b` is read and processed and the output is stored back into buffer `buf_a`.

Fig. 1 uses blue and green color to mark the usage of ping-pong buffers. The feature maps in neighbor layers are stored in two buffers. With this design, we only need two buffers to store the intermediate feature maps no matter how many layers there are. Note that we are not sacrificing performance here as there are intrinsic data dependencies between layers and layers need to be processed sequentially.

D. Multi-Level Tiling

The on-chip memory resource in FPGA is limited. The size of input samples and DNN parameters are much larger than the capacity of FPGA on-chip memory. Tiling is necessary to reuse on-chip memory space and improve the processing efficiency. In this work, we use tiling along multiple dimensions at multiple levels. The combination of tiling along multiple dimensions at multiple levels enables high flexibility of the design. Given input sizes and the amount of resources on the target FPGA platform, this tiled design can be easily configured for best performance by changing tile sizes.

In our design, tiling happens along three different dimensions and levels: ❶ across input samples (input batch); ❷ across layers (inter-layer); ❸ within a layer, across neurons (intra-layer).

The first type of tiling happens across input samples. Fig. 1 and Fig. 2 illustrate this type of tiling (see ❶ in the figures). Multiple input samples (T_{image} samples) are grouped into a tile and processed together. The input samples within a tile share the same copy of DNN parameters. Each load of DNN parameters is reused for T_{image} times. Therefore, the larger tile size T_{image} is, the more times DNN parameters are reused. At the same time, larger T_{image} requires more on-chip memory to store images and parameters.

The second type of tiling is done across DNN layers. ❷ in Fig. 1 shows this type of tiling. T_{layer} layers form a tile. The parameters in a tile are loaded into on-chip BRAM all at once, and the input samples go through each layer in the tile. The processing within a tile is fully pipelined. That means larger T_{layer} requires more intermediate buffers. The output of the tile is the output feature vector from the last layer in the tile. Depending on the tile execution order, the output of the tile may need to be written back to the global DRAM on the FPGA board if image tiles are iterated first. Let L be the number of layers in DNN, then with tile size of T_{layer}, there will be $\lfloor L/T_{\text{layer}} \rfloor$ feature vectors being written back to the on-board DRAM in that case. The larger T_{layer} is, the fewer write-backs there are, while more on-chip memory are required to store parameters.

The third type of tiling happens at a different level than the previous two types of tiling. This type of tiling happens across neurons within a layer. ❸ in Fig. 2 illustrates this type of tiling. Multiple columns (neurons) in a DNN layer are grouped into a tile and multiply with input sample to get the partial sums of corresponding columns. As discussed before, these partial sums are independent from each other and can be done in parallel. The feature vector is duplicated and stored in T_{neuron} different BRAMs so that they can be accessed in parallel. In our design, each of these partial sums is calculated by a separate sparse vector dot product unit.

E. Dynamic Workload Balancing

In our sparse DNN inference engine, there are multiple accelerator instances. Each accelerator can be controlled independently. The host program assigns packs of images to these accelerators. We use a dynamic workload assignment algorithm (Algorithm 1) in the host CPU program to balance the workloads of the accelerators [6]. The input samples are partitioned into small packs and used as the minimal

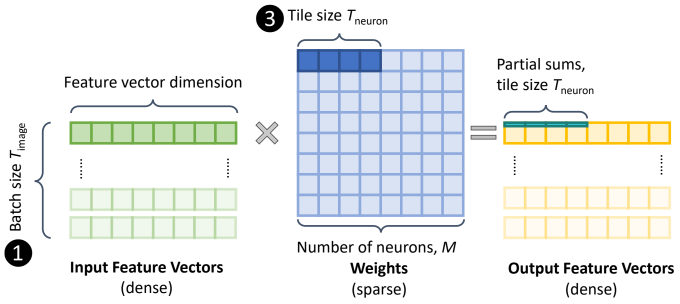

3 Tile size T_{neuron}
Feature vector dimension
Partial sums, tile size T_{neuron}
Batch size T_{image}
1
Number of neurons, M
Input Feature Vectors (dense)
Weights (sparse)
Output Feature Vectors (dense)

Fig. 2. Sparse vector-matrix multiplication

Algorithm 1 Dynamic Workload Assignment

```
Input: Number of input samples $N$, pack size $S$, accelerator
    pool P = {P[0], ..., P[m-1]}.
 1: curr_img ← 0, acc_ptr ← 0
 2: size ← MIN($S$, $N$ − curr_img)
 3: while curr_img < $N$ do
 4:     if P[acc_ptr].ISIDLE() then
 5:         ASSIGN(curr_img, size, P[acc_ptr])
 6:         curr_img ← curr_img + size
 7:     else if P[acc_ptr].ISDONE() then
 8:         COLLECTRESULTS(P[acc_ptr])
 9:         ASSIGN(curr_img, size, P[acc_ptr])
10:         curr_img ← curr_img+size
11:     end if
12:     acc_ptr ← (acc_ptr+1)%m
13: end while
```

assignment unit. In Algorithm 1, one pack contains S input samples, e.g. $S = 32$. The high-level idea of dynamic workload balancing is that the host program checks the status of each accelerator and assigns a pack of input samples to the idle accelerator. There are two cases where the accelerator can accept new workload assignment. The first case is that the accelerator has finished the previous assignments, has results ready, and is ready to accept new ones. Line 7 in Algorithm 1 deals with this case. In this case, the host program collects the results returned from the accelerator, and assigns the current pack to this accelerator. The second case is that the accelerator is idle and doesn't have results to report (Line 4). In this case, the host program simply assigns a pack of input samples to this accelerator. In practice, we choose $S = 32$, and achieve nearly perfect workload balancing of accelerators.

IV. Sparse DNN Accelerator Architecture

In this section, we present the hardware architecture of our sparse DNN inference engine, which incorporates all the optimizations discussed in Section III.

Fig. 3 depicts the high-level view of sparse DNN inference engine in an FPGA chip and the structure of a sparse DNN accelerator. Our sparse DNN inference engine consists of a pool of accelerators. Each of these accelerators can be controlled independently by the host CPU and they don't require synchronization during the processing. Each accelerator can process any number of input samples and any number of DNN layers.

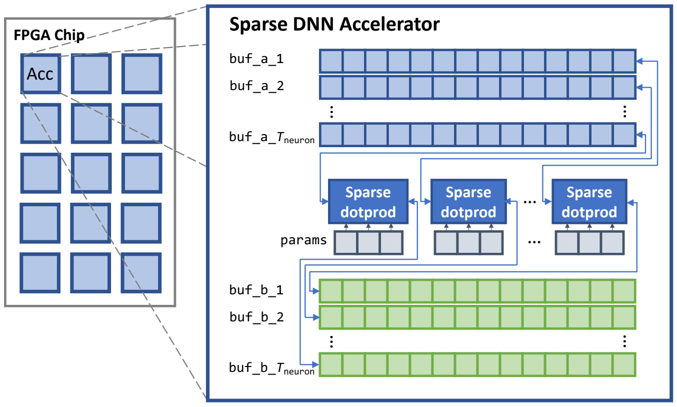


Fig. 3. Sparse DNN accelerator architecture

Our inference engine design can be adopted in both power-constrained edge computing scenarios as well as high-performance cloud computing scenarios. The number of accelerators in the engine is determined by the hardware resource in the FPGA chip. Each accelerator is light-weight but fully capable of running sparse DNN inference. In low-power FPGAs, we can instantiate one single accelerator and achieve low-power high-efficiency processing, while on high-performance FPGAs, many accelerators can be instantiated to achieve high processing throughput.

As illustrated in Fig. 3, inside each accelerator, multiple pairs of ping-pong buffers (group A buf_a_i's and group B buf_b_i's in Fig. 3) and sparse vector dot product processing elements (PEs) are instantiated.

Each PE processes the vector dot product of the same input feature vector with one different column in the parameter matrix. These PEs calculate the partial sums synchronously in a single instruction multiple data (SIMD) manner. The buffers in the accelerator are instantiated with block RAMs (BRAMs) in FPGA. Note that each of these BRAMs has two read ports and can provide two data point per clock cycle. In order to fully utilize the parallelism between columns in the parameter matrix, one input feature vector is actually replicated into T_{neuron} separate buffers, together comprising buffer group A. In this way, T_{neuron} buffers can all be accessed and processed at the same time. The outputs from sparse vector dot product engines are all stored into the same buffer, buf_b_1. After processing of one layer, the values in buf_b_1 are copied into all the other buffers in group B. Then, the weights of the next layer are loaded into parameter buffers (params in Fig. 3) and group B buffers are used as inputs to sparse vector dot product engines. The outputs are stored into the same buffer buf_a_1. Again, before processing the next layer, the values in buf_a_1 are copied to the other buffers in group A.

V. Experiments

A. Test Platform and Dataset

In this work, we use the Xilinx VC709 board [7] as the target FPGA platform. The basic information about our test

TABLE I
TEST PLATFORM INFORMATION

FPGA Board	Xilinx Virtex-7 FPGA VC709 Board
FPGA Chip	Xilinx XC7VX690TFFG1761-2 FPGA
On-Board Memory	2×4GB DDR3 (up to 933MHz)
On-Chip Memory (Kb)	52,920
On-Chip DSP Slices	3,600
On-Chip Logic Cells	693,120
Host-FPGA Interconnect	PCIe Gen3 up to 8 lanes
Host CPU	Intel(R) Core(TM) i7 CPU 950 @ 3.07GHz
Host Memory	24 GB DDR3 (800MHz)
Operating System	Ubuntu 14.04 LTS
Host Compiler	g++ 4.8.4

system is listed in Table I. We use the synthetic sparse DNN dataset from Graph Challenge [8] to evaluate our solution. The information on the synthetic dataset we used in this work is the sparse DNN with 120 layers. Each layer has 1024 neurons. The input sample dimension is 1024 as well. The total parameter size is 176MB.

The input data are stored in text files. Each line in the text file follows the graph edge representation of `(node_a, node_b, weight)`. Our host code reads the input text files and stored the DNN parameters in Compressed Column Storage (CCS) format. The input images are stored in the dense format.

B. Design Parameters

In our final design targeting Xilinx VC709 board, we choose the following design parameters to maximize the performance of the system. Please note that the choice of these design parameters highly depends on the target FPGA platform and the design goals.

- Number of accelerators in FPGA $P = 15$. We put as many accelerators as possible onto the FPGA chip and $P = 15$ is the maximum possible number of accelerators to be integrated into the target Virtex-7 FPGA. Placing more accelerators will lead to severe place and route congestion and timing problem.
- Tiling across input samples $T_{\text{image}} = 1$. We choose $T_{\text{image}} = 1$ and optimize the design for $T_{\text{image}} = 1$ case so that our design can have optimal latency processing one single image. Although choosing $T_{\text{image}} > 1$ can increase the parameter reuse, $T_{\text{image}} > 1$ also requires larger buffer sizes to store intermediate results. Choosing $T_{\text{image}} = 1$ minimizes the pressure on the local on-chip memory.
- Tiling across layers $T_{\text{layer}} = 2$. We load the parameters for two layers at a time to accommodate the processing with ping-pong buffer. We choose to iterate through the layers and not reusing the parameters for images. This way, only the final classification result (one single integer per input sample) needs to be written back, which minimizes the amount of intermediate results being written back. This reduces the pressure on DRAM bandwidth and improves the efficiency.
- Tiling across neurons $T_{\text{neuron}} = 16$. We create a script to automatically generate synthesizable C code with various T_{neuron} values and test the latency of the design. It turns out that $T_{\text{neuron}} = 16$ is the optimal setting under the timing constraint of 4ns per clock cycle (250 MHz). Smaller T_{neuron} doesn't fully exploits the parallelism within a DNN layer, while larger T_{neuron} introduces larger overhead in extra buffering space.
- Workload assignment pack size $S = 32$. We evaluate the accelerator performance with different S values, such as 32, 64, and 256. The differences in performance with these sizes are not significant for the current setting.

TABLE II
FPGA RESOURCE UTILIZATION

Look-Up Tables	209,814 / 433,200 (48.43%)
Flip-Flop	232,720 / 866,400 (26.86%)
BRAM	815 / 1,470 (55.44%)
DSP	150 / 3,600 (4.17%)

C. Evaluation

With the design parameters listed in Section V-B, the resource utilization of the inference engine on VC709 board is listed in Table II. Note that different design parameters will lead to different FPGA resource utilization ratios. As we explained in Section V-B, we conservatively uses around 50% of FPGA resources so that the frequency and timing quality of the synthesized circuit can be guaranteed. The power consumption of this design is around 12W, which is estimated by the synthesis flow in Xilinx Vivado.

To fully evaluate the benefits of our proposed techniques, we measure the performance of two FPGA designs, one ("Optimized") is with all optimizations described in Section III, the other ("Basic") is a basic FPGA design without tiling. Figure 4 and Table III show the performance of two designs with various number of accelerators, as well as the efficiency improvement from the optimized FPGA solution compared to the CPU solution. As shown in the figure, the optimized design can achieve more than five times speedup compared to the basic design. In our evaluation, the best number of accelerators for the optimized design is around 7. For smaller number of accelerators, adding more accelerators exploits parallelism in processing images and therefore improves performance. However, when there are enough accelerators, FPGA on-board memory bandwidth becomes the bottleneck of the whole system. Even though adding more accelerators increases computational capabilities, it also increases memory access pressure. At some point, memory bandwidth saturates and adding more accelerators no longer improves system performance.

We evaluated the baseline MATLAB code provided by Graph Challenge on a high-performance server with four AMD Opteron 6272 Processors (4×16 cores). The execution time of the MATLAB code is 124.07 seconds, and its power consumption is estimated to be 114W [9]. Although the performance of multi-core CPU is around two times faster than the FPGA solution, the power efficiency of our design is up to $4.7\times$ higher than the multi-core solution. Here are a few notes to help understand the difference in performance between our

TABLE III
SPARSE DNN ACCELERATOR PERFORMANCE

P	Basic (s)	Optimized (s)	Speedup	Efficiency over CPU
1	4618.45	906.81	5.09	1.30
2	2313.04	474.29	4.88	2.49
4	1159.95	310.14	3.74	3.80
6	773.26	254.76	3.03	4.63
7	662.16	251.31	2.63	4.69
8	579.26	269.89	2.14	4.37
12	386.85	489.37	0.79	2.41
15	310.53	484.23	0.64	2.43

solution and the MATLAB based multiple CPU solution. First, the MATLAB implementation uses the sparse BLAS libraries in MATLAB, which is highly optimized for sparse matrix operations. Therefore the MATLAB version is not a trivial reference solution. Second, the target platform has much more hardware resources and higher computation capability. We are comparing our single FPGA solution against a solution based on 64-core server grade high-performance CPUs here. Third, the memory access path from host memory to FPGA device memory has lower peak data transfer bandwidth compared the CPU. In our design, in order to reduce the complexity of FPGA place and route and create designs with good timing properties, we only uses one single PCIe lane (up to 8 lanes are allowed in hardware) and only one host-device channel (up to 4 channels are allowed in hardware). Based on our experiment results, using more PCIe lanes or host-device channels will result in bad circuit timing and the design frequency will be low. This narrow CPU-FPGA interface becomes a bottleneck. The performance of our accelerator solution can be further improved with better optimized CPU-FPGA inference. This will be done as a future work.

Given enough DRAM bandwidth, our solution can be easily scaled to larger FPGAs with more accelerators or even multiple FPGAs, as our accelerators can operate independently on different set of input samples. This way the parallelism in input samples and neurons in the layers can be further exploited, as we discussed in Section III. With more accelerator instances and multiple FPGAs, our solution should be able to outperform

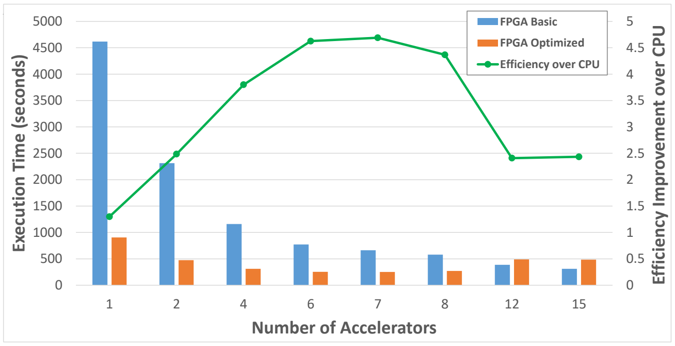


Fig. 4. Sparse DNN accelerator performance

VI. RELATED WORKS

In this work, we focus on hardware acceleration of sparse deep neural networks (DNNs). Converting dense deep neural networks into sparse ones is out of the scope of this work. The computation in sparse DNNs we focus on is essentially sparse vector-matrix multiplication with non-linear activation functions. There are several works on acceleration of sparse matrix vector multiplication. Fowers et al. [10] proposed an FPGA design for sparse matrix-vector multiplication. The accelerator is designed in RTL code. The design consumes 25W and achieve 2.6× and 2.3× higher energy efficiencies than CPU and GPU. The performance of the design is around two thirds of CPU performance and one third of GPU performance. This work uses Compressed Interleaved Sparse Row (CISR) matrix encoding which enables simultaneous multiply-accumulate operations on multiple rows of the matrix. The problem solved here is similar to our work, the major difference is that our work focuses on sparse DNNs specifically instead of sparse matrix vector multiplication. Also, our sparse DNN inference engine is parameterized and is capable of exploiting various types of parallelism and data reuse opportunities.

Giefers et al. [11] did a thorough comparison of the energy efficiency of sparse matrix multiplication on CPU, Xeon Phi and FPGAs, in the context of heterogeneous systems. The FPGA platform in this work is Nallatech 385N FPGA board, which contains an Altera Stratix V FPGA. The design is done with OpenCL SDK for FPGA. The evaluation results show that FPGA is remarkably efficient. This work focuses on energy efficiency comparison across platforms, and the FPGA design uses the general OpenCL code which may not be best optimized for FPGA and the design flow.

Besides, there are several recent works focus on FPGA acceleration of sparse convolutional neural networks (CNN) [12] [13] and sparse long short-term memory (LSTM) [14]. These works accelerate the CNNs and LSTMs while this work focus on very deep fully connected networks. The computation inside sparse CNN and LSTM have similar memory random access patterns as sparse matrix vector multiplication. However, sparse CNNs and LSTMs have unique data dependency patterns, therefore the high-level data access and computation patterns in these works are different from this work. Besides, this work targets very deep and wide networks which are generally larger than the networks used in current applications.

VII. CONCLUSION

In this work, we proposed and built a configurable sparse DNN inference engine. The proposed inference engine is parameterized and it can be configured to have different sizes and different processing capabilities. The inference engine can be adopted in both edge computing and high-performance computing scenarios. We also modeled and analyzed the computation of sparse DNN inference, parameterized sparse DNN hardware design, and presented the design space of the sparse DNN accelerators. The proposed design was evaluated on Xilinx VC709 FPGA board. Evaluation results show that the proposed design achieve up to 4.7× better energy efficiency compared to CPU.

ACKNOWLEDGMENT

This work is supported by IBM-ILLINOIS Center for Cognitive Computing Systems Research (C3SR) - a research collaboration as part of the IBM AI Horizons Network.

REFERENCES

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2012, pp. 1097–1105. [Online]. Available: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

[2] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *CoRR*, vol. abs/1512.03385, 2015. [Online]. Available: http://arxiv.org/abs/1512.03385

[3] X. Zhang, X. Liu, A. Ramachandran, C. Zhuge, S. Tang, P. Ouyang, Z. Cheng, K. Rupnow, and D. Chen, "High-performance video content recognition with long-term recurrent convolutional network for FPGA," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, Sep. 2017, pp. 1–4.

[4] C. Hao, X. Zhang, Y. Li, S. Huang, J. Xiong, K. Rupnow, W.-m. Hwu, and D. Chen, "FPGA/DNN Co-Design: An efficient design methodology for IoT intelligence on the edge," in *Proceedings of the 56th Annual Design Automation Conference 2019*, ser. DAC '19. New York, NY, USA: ACM, 2019, pp. 206:1–206:6. [Online]. Available: http://doi.acm.org/10.1145/3316781.3317829

[5] J. Kepner, S. Alford, V. Gadepally, M. Jones, L. Milechin, R. Robinett, and S. Samsi, "Sparse deep neural network graph challenge," 2019. [Online]. Available: https://graphchallenge.mit.edu/challenges

[6] S. Huang, L.-W. Chang, I. El Hajj, S. Garcia de Gonzalo, J. Gómez-Luna, S. R. Chalamalasetti, M. El-Hadedy, D. Milojicic, O. Mutlu, D. Chen, and W.-m. Hwu, "Analysis and Modeling of Collaborative Execution Strategies for Heterogeneous CPU-FPGA Architectures," in *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '19. New York, NY, USA: ACM, 2019, pp. 79–90.

[7] "Xilinx virtex-7 fpga vc709 connectivity kit," https://www.xilinx.com/products/boards-and-kits/dk-v7-vc709-g.html.

[8] J. Kepner, S. Alford, V. Gadepally, M. Jones, L. Milechin, R. Robinett, and S. Samsi, "Sparse deep neural network graph challenge," 2019.

[9] "Bulldozer for servers: Testing amd's "interlagos" opteron 6200 series," https://www.anandtech.com/show/5058/amds-opteron-interlagos-6200/8.

[10] J. Fowers, K. Ovtcharov, K. Strauss, E. S. Chung, and G. Stitt, "A high memory bandwidth fpga accelerator for sparse matrix-vector multiplication," in *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, May 2014, pp. 36–43.

[11] H. Giefers, P. Staar, C. Bekas, and C. Hagleitner, "Analyzing the energy-efficiency of sparse matrix multiplication on heterogeneous systems: A comparative study of gpu, xeon phi and fpga," in *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, April 2016, pp. 46–56.

[12] L. Lu, J. Xie, R. Huang, J. Zhang, W. Lin, and Y. Liang, "An efficient hardware accelerator for sparse convolutional neural networks on fpgas," in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, April 2019, pp. 17–25.

[13] J. Chang, K. Kang, and S. Kang, "SDCNN: An efficient sparse deconvolutional neural network accelerator on FPGA," in *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2019, pp. 968–971.

[14] S. Cao, C. Zhang, Z. Yao, W. Xiao, L. Nie, D. Zhan, Y. Liu, M. Wu, and L. Zhang, "Efficient and effective sparse LSTM on fpga with bank-balanced sparsity," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '19. New York, NY, USA: ACM, 2019, pp. 63–72. [Online]. Available: http://doi.acm.org/10.1145/3289602.3293898