Efficient and Effective Sparse LSTM on FPGA with Bank-Balanced Sparsity

Shijie Cao[*]
Harbin Institute of Technology
caoshijie0501@gmail.com

Chen Zhang
Microsoft Research
zhac@microsoft.com

Zhuliang Yao[*]
Tsinghua University
v-zhuyao@microsoft.com

Wencong Xiao[*]
Beihang University
v-wencxi@microsoft.com

Lanshun Nie
Harbin Institute of Technology
nls@hit.edu.cn

Dechen Zhan
Harbin Institute of Technology
dechen@hit.edu.cn

Yunxin Liu
Microsoft Research
yunxin.liu@microsoft.com

Ming Wu
Microsoft Research
miw@microsoft.com

Lintao Zhang
Microsoft Research
lintaoz@microsoft.com

ABSTRACT

Neural networks based on Long Short-Term Memory (LSTM) are widely deployed in latency-sensitive language and speech applications. To speed up LSTM inference, previous research proposes weight pruning techniques to reduce computational cost. Unfortunately, irregular computation and memory accesses in unrestricted sparse LSTM limit the realizable parallelism, especially when implemented on FPGA. To address this issue, some researchers propose block-based sparsity patterns to increase the regularity of sparse weight matrices, but these approaches suffer from deteriorated prediction accuracy.

This work presents *Bank-Balanced Sparsity* (BBS), a novel sparsity pattern that can maintain model accuracy at a high sparsity level while still enable an efficient FPGA implementation. BBS partitions each weight matrix row into banks for parallel computing, while adopts fine-grained pruning inside each bank to maintain model accuracy. We develop a 3-step software-hardware co-optimization approach to apply BBS in real FPGA hardware. First, we propose a bank-balanced pruning method to induce the BBS pattern on weight matrices. Then we introduce a decoding-free sparse matrix format, *Compressed Sparse Banks* (CSB), that transparently exposes inter-bank parallelism in BBS to hardware. Finally, we design an FPGA accelerator that takes advantage of BBS to eliminate irregular computation and memory accesses. Implemented on Intel Arria-10 FPGA, the BBS accelerator can achieve 750.9 GOPs on sparse LSTM networks with a batch size of 1. Compared to state-of-the-art FPGA accelerators for LSTM with different compression techniques, the BBS accelerator achieves 2.3 ~3.7x improvement on energy efficiency and 7.0 ~34.4x reduction on latency with negligible loss of model accuracy.

[*]Contribution during internship at Microsoft Research.

KEYWORDS

FPGA; Deep Neural Networks; LSTM; Weight Pruning; Inference; Bank-Balanced Sparsity

ACM Reference Format:
Shijie Cao, Chen Zhang, Zhuliang Yao, Wencong Xiao, Lanshun Nie, Dechen Zhan, Yunxin Liu, Ming Wu, and Lintao Zhang. 2019. Efficient and Effective Sparse LSTM on FPGA with Bank-Balanced Sparsity. In *The 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '19), February 24–26, 2019, Seaside, CA, USA*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3289602.3293898

1 INTRODUCTION

Neural networks based on Long Short-Term Memory (LSTM) have been widely used in interactive and latency-sensitive applications such as machine translation, speech recognition and speech synthesis [13, 20, 24]. The size and computational cost of these LSTM models continue to grow in order to achieve better model accuracy. However, the stringent requirement on computational resources makes it challenging to achieve low inference latency for large networks. The most time-consuming part of LSTM inference is *matrix-vector multiplication* (MxV). As the size of the LSTM network grows, MxV cost grows quadratically, thus significantly increasing the inference cost.

Weight pruning is a model compression technique to reduce overall memory and computational costs. Early works [8, 10] discover that removing LSTM weights below a small threshold has negligible impact on model accuracy. By clamping a significant portion of the weights to 0, weight pruning approach converts dense weight matrices to unstructured sparse matrices, thus reducing the computation and memory required to carry out inference.

After pruning, the most significant part of LSTM inference changes from dense MxV to *sparse matrix-vector multiplication* (SpMxV). Though requiring less computation, the irregularity of SpMxV limits the maximum performance and energy efficiency achievable on hardware accelerators [17, 19, 27]. Unstructured sparse matrices cannot efficiently utilize underlying hardware resources due to three reasons: 1) the unbalanced non-zero weights distribution might cause workload skew among processing elements (PEs); 2) concurrent irregular memory accesses to a dense vector lead to memory access conflicts, which could stall the parallel execution; and 3) sparse matrix representations such as *compressed sparse row* (CSR) use indexes to track non-zero values, which require decoding before computation.

To address these issues, further works [17, 19] suggest using coarser-grained weight pruning methods to induce more structured sparsity patterns for better hardware acceleration. Coarse-grained pruning methods prune weights in the granularity of blocks. From the hardware perspective, blocks of non-zero weights can enable contiguous memory accesses and better utilize parallel computation resources. Unfortunately, it becomes challenging to maintain the same model accuracy when block sparsity is applied. Block sparsity constrains the locality of the non-zero weights, and important weights could be mistakenly pruned, resulting in model accuracy loss. Furthermore, the block size (i.e., pruning granularity) is application-sensitive, making it another hyper-parameter to tune. Existing work often needs to search a range of block sizes to find a trade-off between model accuracy and hardware efficiency [13, 17].

This work presents *Bank-Balanced Sparsity* (BBS), a novel sparsity pattern for pruning LSTM. Bank-balanced pruning splits each weight matrix row into multiple equal-sized banks, and adopts fine-grained pruning to each bank independently to obtain identical sparsity among banks. BBS preserves the unstructured distribution of non-zero weights inside each bank, thus maintaining higher model accuracy than that of block sparsity. Experimental results in Section 6 demonstrate that BBS achieves almost the same model accuracy as unstructured sparsity and significantly outperforms block sparsity when pruning weights at the same sparsity level.

Importantly, BBS is also amenable to FPGA acceleration because it inherently provides a balanced matrix partitioning for parallel computing. We design an FPGA accelerator to take advantage of the benefits of BBS to eliminate the computational overheads existed in unstructured sparsity. Specifically: 1) our accelerator utilizes the intrinsic bank-balanced property in BBS to achieve high parallelism in SpMxV with guaranteed load balance; 2) our accelerator supports concurrent random access requests to vector elements without conflicts in SpMxV by adopting banked scratchpad memory to buffer vectors; 3) to avoid decoding overheads of sparse matrix formats, we introduce a novel format for BBS matrices that is decoding-free in our FPGA accelerator. Notably, the BBS accelerator is highly efficient even for inference with a batch size of 1, by exploiting fine-grained parallelism from a single sample which is challenging for unstructured sparsity.

Overall, this paper makes the following contributions:

(1) We propose Bank-Balanced Sparsity, a novel sparsity pattern that can both maintain model accuracy and enable an efficient FPGA accelerator implementation.
(2) We design an FPGA-based accelerator for BBS that eliminates load-imbalance, irregular memory accesses and decoding overheads, and achieves good efficiency for LSTM inference even at a batch size of 1.
(3) Implemented on Intel Arria-10 FPGA, the BBS accelerator achieves 750.9 GOPs on large LSTMs without batching. Compared to state-of-the-art LSTM FPGA accelerators, we achieve 2.3 ~3.7x improvement on energy efficiency and 7.0 ~34.4x reduction on latency with negligible loss of model accuracy.

2 BACKGROUND

2.1 Long Short-Term Memory.

LSTM is one of the most successful cells used in Recurrent Neural Networks (RNNs) [11]. An LSTM network computes a mapping from an input sequence $X = (x_1, ..., x_T)$ to an output sequence $Y = (y_1, ..., y_T)$ by using the following equations iteratively from t = 1 to T:

$$i_t = \sigma(W_{ix}x_t + W_{ir}y_{t-1} + W_{ic}c_{t-1} + b_i) \tag{1}$$
$$f_t = \sigma(W_{fx}x_t + W_{fr}y_{t-1} + W_{fc}c_{t-1} + b_f) \tag{2}$$
$$g_t = \sigma(W_{cx}x_t + W_{cr}y_{t-1} + b_c) \tag{3}$$
$$c_t = f_t \odot c_{t-1} + g_t \odot i_t \tag{4}$$
$$o_t = \sigma(W_{ox}x_t + W_{or}y_{t-1} + W_{oc}c_{t-1} + b_o) \tag{5}$$
$$m_t = o_t \ \odot \ h(c_t) \tag{6}$$
$$y_t = W_{ym}m_t \tag{7}$$

where the W terms denote weight matrices, the b terms denote bias vectors. The symbols i, f, o and c are respectively the input gate, forget gate, output gate and cell activation (long-term memory). The $\odot$ operator denotes element-wise multiplication, and the + operator denotes element-wise addition. σ is the logistic activation function and h is the hyperbolic tangent (Tanh) activation function.

Among all operators in LSTM, matrix-vector multiplication (MxV) is the most memory-intensive and computation-intensive operator. The dimensions of x_t, y_t and c_t are often the same, say D. Therefore, the number of weights is $12 \times D^2$. In each step of the inference calculation, the number of operations in MxV is $24 \times D^2$, and the number of operations in element-wise operators (EWOP) is $9 \times D$. As a consequence, accelerating MxV is the key to low latency LSTM inference.

2.2 Weight Pruning

It is widely observed that Deep Neural Networks (DNNs) have a lot of redundancy in weights. Pruning away (forcing to zero) a proper number of unimportant weights won't affect model accuracy. Moreover, weight pruning can reduce the model size and computational complexity for energy efficient hardware acceleration. Deep Compression [9, 10] provides a threshold-based weight pruning technique. This method prunes away small weights whose absolute values are less than a predefined threshold and retrains the remaining weights. Pruning and retraining are iteratively applied to generate the sparse DNN model.

As mentioned in the introduction, unrestricted pruning of weight matrices is unfriendly to hardware acceleration. Further work [17, 19] proposes coarse-grained pruning methods to prune blocks of weights. They pick the maximum magnitude or the average magnitude of the weights within a block as the representative of the entire block. If the representative magnitude is less than a predefined threshold, the entire block will be pruned. However, the pruning granularity affects hardware efficiency as well as model accuracy. Deep neural network designers struggle to balance model accuracy and hardware efficiency.

3 BANK-BALANCED SPARSITY

Our proposed sparsity pattern, *Bank-Balanced Sparsity* (BBS), achieves both high model accuracy and high hardware efficiency. In this section, we first describe the pattern of BBS and the motivation for designing it. Then, we present the detailed bank-balanced pruning algorithm to induce BBS on LSTM weight matrices. Finally, we analyze the pruning effectiveness of BBS in terms of achievable accuracy and sparsity. The efficient hardware acceleration design for BBS will be introduced in the next section.

3.1 Bank-Balanced Sparsity Pattern

For matrices represented in BBS, each matrix row is split into multiple equal-sized banks (i.e., sub-rows), and each bank has the same number of non-zero values. Figure 1 illustrates BBS with an example and compares it with unstructured sparsity and block sparsity. In this example, three sparse matrices with different sparsity patterns are all pruned from the dense example weight matrix in Figure 1(a)) with a sparsity ratio of 50%. Fine-grained pruning globally sorts the weights and prunes the smallest 50% of weights, leading to an unstructured sparse matrix (Figure 1(b)); Coarse-grained pruning induces a block sparse matrix (Figure 1(c)) by setting the block size to 2x2 and the block representative with the block average; Our bank-balanced pruning induces a bank-balanced sparse matrix (Figure 1(d)) by splitting each matrix row into 2 equal-sized banks and applying fine-grained pruning inside each bank independently.

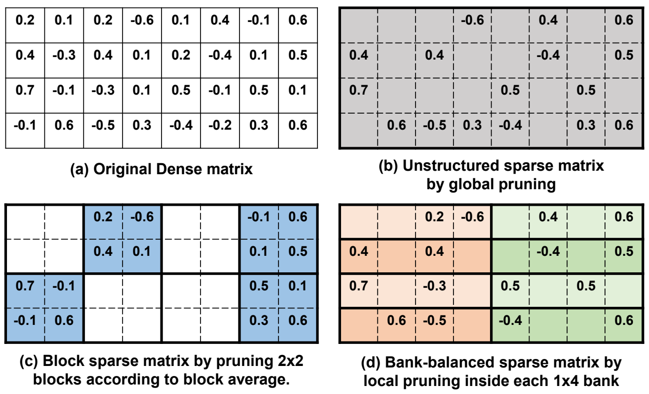


Figure 1: Comparing BBS with unstructured sparsity and block sparsity by pruning a dense matrix with a sparsity ratio of 50%.

We design this BBS sparsity pattern with consideration of both hardware efficiency and model accuracy. In general, partitioning weight matrix into multiple sub-matrices is mandatory for parallel computing. In BBS, each matrix row is split into multiple banks with the same size and same sparsity. This bank-balanced partitioning enables an efficient SpMxV design to exploit both inter-row parallelism and intra-row parallelism (i.e., inter-bank parallelism) with guaranteed load balance and no vector access conflicts. The detailed SpMxV design for BBS will be described in Section 4.1. In addition, since BBS applies fine-grained pruning within each bank independently, the relatively large weights which contribute more to model accuracy in each bank can be preserved.

Another potential design for a sparsity pattern would be to split weight matrices into 2-D blocks like block sparsity and apply fine-grained pruning within each 2-D block. Larger weights within each block can be preserved as well in this scheme. However, after pruning, each 2-D block is still an unstructured sparse matrix. It is still challenging to design an efficient hardware accelerator architecture due to the irregularity of sparse sub-matrices. For example, parallelizing SpMxV across 2-D blocks leads to concurrent irregular vector accesses.

3.2 Bank-Balanced Pruning Algorithm

To induce BBS on LSTM weight matrices, we adopt a bank-balanced pruning method that prunes each bank independently with the same threshold percentage to obtain the same sparsity ratio among banks.

Algorithm 1 Bank-Balanced Pruning Algorithm

```
Input:
    The matrix to be pruned, $M$;
    The number of banks per row, $BankNum$;
    The expected sparsity, $Sparsity$;
Output:
    The pruned matrix, $M_p$;
1:  for each $M_i \in M.rows$ do
2:      Divide the row $M_i$ into $BankNum$ blocks;
3:      for each $bank \in M_i$ do
4:          Sort the elements in $bank$;
5:          Calculate the bank internal threshold $T$ in line with
            $Sparsity$;
6:          for each $element \in bank$ do
7:              prune $element$ if $element < T$;
8:          end for
9:      end for
10: end for
11: return the pruned matrix, $M_p$;
```

Like previous pruning methods, we apply the bank-balanced pruning method iteratively to a pre-trained network, and fine-tune the network after each pruning iteration to restore the model accuracy. Algorithm 1 illustrates the detailed bank-balanced pruning method to induce BBS on LSTM weight matrices. In each pruning iteration, bank-balanced pruning first partitions each matrix row to multiple equal-sized banks and sorts the weights within each bank by their absolute values. The importance of weights is represented as their bank internal ranking of absolute values. Iteratively, a percentage of weights with the smallest absolute values are pruned. We slowly increase the pruning percentage from 0% to the target sparsity, while the rate of increase decreases with each pruning iteration. During pruning, if the model accuracy drops significantly and cannot be recovered via fine-tuning, we withdraw this pruning iteration and stop the pruning procedure.

3.3 Analysis of Our Pruning Method

Intuitively, a pruning method should remove only smaller weights and preserve larger weights that contribute more to model accuracy. Fine-grained pruning clamps weights of small magnitudes to

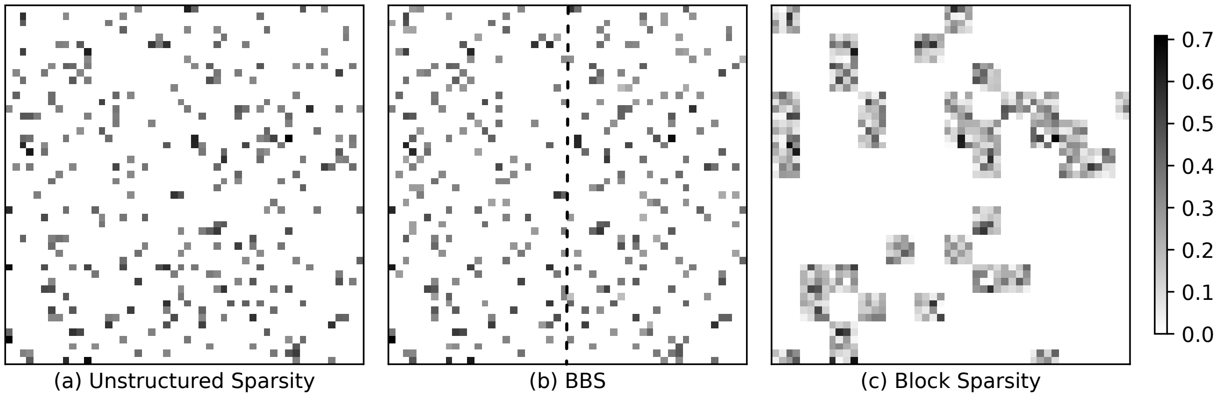


Figure 2: Weight map visualization after pruning with (a) unstructured sparsity, (b) BBS, and (c) block sparsity (sparsity ratio = 90%). These weight maps are 64 × 64 sub-matrices of the whole 1500 × 1500 matrix.

zero and preserves large weights to maintain model accuracy. For bank-balanced pruning, we adopt fine-grained pruning inside banks independently, so large weights inside each bank can be preserved. In contrast, coarse-grained pruning prunes blocks of weights, which constrains the locality of preserved non-zero weights, and therefore some important weights could be mistakenly pruned while some unimportant weights are instead preserved. For the example in Figure 1, the bank-balanced sparse matrix in (d) preserves similar larger weights as the unstructured sparse matrix in (b), but the block sparse matrix in (c) removes some large weights (e.g., 0.4 and 0.5) but preserves some small weights (e.g. 0.1 and -0.1).

Table 1: Percentages of the largest weights that are preserved in various sparsity patterns (sparsity ratio = 90%).

Weight Matrices	Unstructured Sparsity	BBS	Block Sparsity
W_{ix}	100.00%	91.30%	42.76%
W_{fx}	100.00%	81.39%	24.26%
W_{cx}	100.00%	84.45%	24.24%
W_{ox}	100.00%	85.62%	22.97%

To verify the pruning effectiveness of BBS and compare it with unstructured sparsity and block sparsity, we analyze and visualize the weight matrices after corresponding pruning methods in a real LSTM model [28]. The hidden size of this LSTM model is 1500. Table 1 shows the percentage of the largest weights that are preserved in various sparsity patterns. Here we show the results of W_{ix}, W_{fx}, W_{cx} and W_{ox}, other weight matrices have similar results. In this analysis, the sparsity ratios are all 90%, the bank size of BBS is 32 and the block size of block sparsity is 4 × 4. Unstructured sparsity by fine-grained pruning naturally preserves 100% largest weights because it globally prunes weights with smallest magnitudes. BBS preserves more than 80% of the largest weights by fine-grained pruning inside each bank, while block sparsity only preserves less than half of (or even quarter of) the largest weights. Figure 2 visualizes these three kinds of sparse weight matrices of a 64 × 64 sub-matrix which is randomly selected from the whole 1500 × 1500 W_{ix}. Grey grids indicate non-zero parameters and the grey level indicates the magnitude of the absolute value. For the second matrix represented in BBS, each row has two banks (left and right sides of the dashed line). Each bank has 3 non-zero weights. We can see that the weight map of BBS is very similar to the weight map of unstructured sparsity, but the weight map of block sparsity is quite different because of the locality constraint.

In terms of achievable sparsity and accuracy, experimental results on two typical data sets [7, 18] demonstrate BBS has almost the same effectiveness as unstructured sparsity and outperforms block sparsity, described in Section 6.2.

4 SPARSE MATRIX COMPUTATION AND FORMAT FOR BBS

As mentioned, the irregularity of unstructured sparsity is not hardware friendly due to unbalanced computation, irregular memory accesses and decoding overheads. In contrast, the intrinsic bank-balanced property of BBS enables effective hardware designs to address these issues. For BBS, we introduce a highly parallel SpMxV design with guaranteed load balance and no vector access conflicts, and an associated decoding-free sparse matrix format for the SpMxV design.

4.1 Highly Parallel SpMxV Design

SpMxV consists of multiple dot product operations, one for each sparse matrix row and the dense vector. The standard practice of using multiple PEs to parallelize dot products across matrix rows can reduce computation time. However, irregular memory access patterns of unstructured sparse matrices restrict further parallelism within a dot product.

In addition to inter-row parallelism, BBS enables an efficient SpMxV design to exploit intra-row parallelism (i.e. inter-bank parallelism) through the bank-balance partitioning. Figure 3 illustrates how to exploit inter-bank parallelism in computing a dot product of two vectors (i.e., a BBS matrix row and the dense vector). The multiplications for the non-zero elements inside each bank are performed serially, while the multiplications in different banks are performed in parallel. In this example, the sparse matrix row is divided into 4 banks, as is shown in different colors. The size of each bank is 3 and the sparsity is 1/3. The multiplied dense vector is divided into 4 banks accordingly. Our design computes the dot product of two vectors by accumulating dot products of sub-vectors whose sizes are all the number of banks (N). Each bank of the sparse matrix row provides one non-zero element to form one sub-vector (e.g., (A, C, E, G)), while dense vector elements are fetched based on the indices of non-zero values to form another sub-vector (e.g., (v_0, v_3, v_7, v_9)). For computing a dot product of sub-vectors, N pair-wise multiplications are executed in parallel. Multiple dot products of sub-vectors are calculated in sequential and accumulated to obtain the dot product of complete vectors.

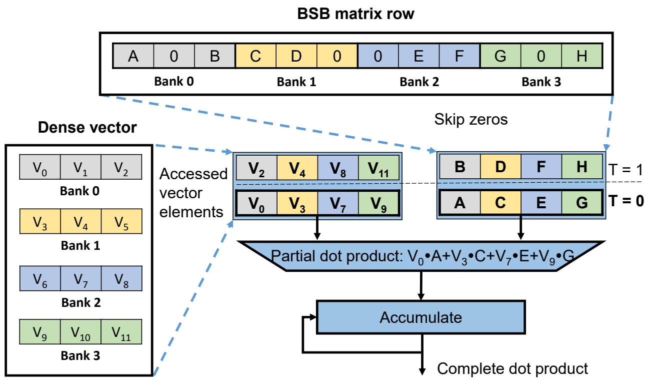


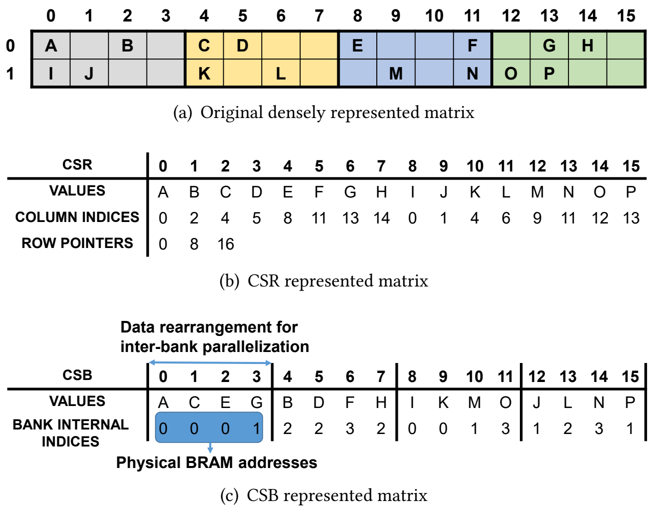
Figure 3: Exploiting inter-bank parallelism in dot product computation of one BBS matrix row and the dense vector.

The bank-balanced property in BBS eliminates load imbalance and irregular memory accesses. In BBS matrices, every row (and every bank) has the same number of elements which automatically guarantees the load balance across rows and banks in SpMxV. When calculating a partial dot product, BBS ensures one and only one element is accessed in each bank. Therefore, storing each vector bank in an independently accessible block RAM can supply vector elements simultaneously with high bandwidth and without memory access conflicts. The detailed FPGA implementation is shown in Section 5.

4.2 Decoding-Free Sparse Matrix Format

Various sparse matrix formats have been proposed to reduce the memory footprint of sparse matrices. However, existing formats introduce decoding overheads when performing sparse matrix multiplications. For FPGA implementation, decoding sparse formats consumes hardware resources and incurs latency. In order to eliminate decoding overheads, we introduce a sparse matrix format called *Compressed Sparse Banks* (CSB) that is specifically designed for BBS.

Compressed Sparse Row (CSR) is a commonly used sparse matrix format [1]. We use CSR as a representative encoding of existing formats for explanation and comparison. Figure 4(a) shows a bank-balanced sparse matrix represented in dense format. Figure 4(b) shows its corresponding CSR encoding. CSR incurs two types of overheads for SpMxV operation. First, CSR format encodes all non-zero elements in a row-major order. Thus, rearranging the non-zero elements are inevitable in order to exploit inter-bank parallelism in SpMxV. Second, CSR format stores column indices and row pointers



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	A		B		C	D			E			F		G	H	
1	I	J			K		L			M		N	O	P		

(a) Original densely represented matrix

CSR	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
VALUES	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
COLUMN INDICES	0	2	4	5	8	11	13	14	0	1	4	6	9	11	12	13
ROW POINTERS	0	8	16													

(b) CSR represented matrix

Data rearrangement for inter-bank parallelization

CSB	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
VALUES	A	C	E	G	B	D	F	H	I	K	M	O	J	L	N	P
BANK INTERNAL INDICES	0	0	0	1	2	2	3	2	0	0	1	3	1	2	3	1

Physical BRAM addresses

(c) CSB represented matrix

Figure 4: The comparison between CSR and CSB.

to track the location of each non-zero value. Thus, calculating memory addresses is required to fetch vector elements. Other encoding formats, such as CSC and COO have similar limitations [1].

The proposed CSB format takes advantage of the balanced property of BBS and eliminates the need for decoding. Figure 4(c) shows the CSB representation of the corresponding matrix. The CSB encoding uses two arrays to represent a bank-balanced sparse matrix. In the first array (i.e., values), all non-zero values are first arranged in row-major order. Inside each row, the first non-zero elements in each banks (e.g., (A, C, E, G)) are listed first, then the second elements, and so on. The purpose of this data rearrangement is to explicitly expose inter-bank parallelism, thus every successive N elements in CSB can be directly fetched and computed upon in parallel. The second array (i.e., indices) lists the bank internal indexes of non-zero values which are column indices modulo bank size K. When each of the N vector banks is stored in a separate BRAM block on FPGA, the bank internal indices can be directly regarded as physical addresses to fetch the N corresponding vector elements in the BRAM blocks.

5 LSTM ACCELERATOR

In this section, we introduce the BBS accelerator, an FPGA-based accelerator for LSTM networks with bank-balanced pruning. The BBS accelerator is implemented as an accelerator on the PCIe I/O bus to serve LSTM inference requests from the host server. Our design specially accelerates LSTM networks at a batch size of one to reduce inference latency by devoting the on-chip resources to exploiting as much parallelism as possible from one single sample.

5.1 Overall Architecture

Figure 5 shows the overall architecture of the BBS accelerator, which consists of a sparse matrix-vector multiplication unit (SpMxV Unit), an element-wise vector operation unit (EWOP Unit), a direct memory access module (DMA) for load/store operations, on-chip memories for matrices and vectors (Matrix Memory and Vector Memory), and a central controller. Before hardware acceleration, the host

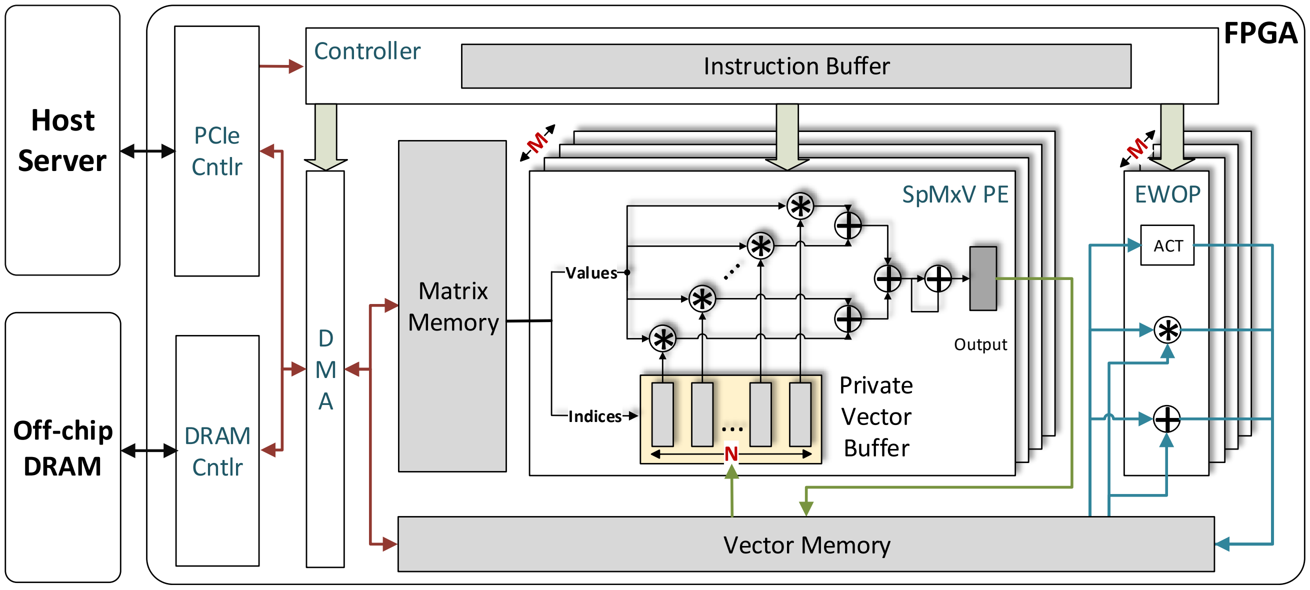


Figure 5: Overall architecture.

server uses the bank-balanced pruning method to prune weight matrices and represents sparse matrices in our proposed Compressed Sparse Banks (CSB) format, then a lightweight compiler generates instructions for the hardware accelerator to accomplish the computation of LSTM. The controller receives and stores instructions from the host server in the instruction buffer and dispatches them to their corresponding modules to execute.

The two important types of instructions are load/store instructions and computational instructions:

Load/Store Instructions. Load/Store instructions are executed in the DMA module to transfer weight matrices and input/output vectors. A load instruction reads data (model weights and inputs) from host memory/off-chip DRAM to on-chip memories. A store instruction writes data (outputs) from on-chip memories to host memory/off-chip DRAM. In practice, in many cases weight pruning can reduce model size enough to fit in on-chip memories. For serving real-time LSTM with low latency, the default mode is to completely rely on on-chip memories. For large models that can not fully fit into on-chip memories even with compression, the BBS accelerator uses load/store instructions to read/write weight matrices from/to off-chip DRAM.

Computational Instructions. As introduced in Section 2, all operations in sparse LSTM can be put into 2 categories: SpMxV and EWOP (including addition, multiplication and three kinds of activations). Therefore, we design two kinds of computational instruction: SpMxV instruction and EWOP instruction to fulfill LSTM computation. The SpMxV instruction is executed in the SpMxV unit to read the required matrix and vector from on-chip memories, then compute dot products for matrix rows, and finally write the result vector back to the vector memory. The EWOP instruction is executed in the EWOP unit to read required vector(s) from the vector memory and write the resulting vector of element-wise addition/multiplication/activations back to the vector memory.

5.2 SpMxV Unit

The SpMxV unit implements the highly parallel design described in Section 4.1. The SpMxV unit consists of M parallel processing elements (PEs) that compute dot products of distinct matrix rows and the dense vector concurrently to exploit inter-row parallelism, while each PE is designed to exploit intra-row (i.e., inter-bank) parallelism in a single dot product operation.

In the center of Figure 5, we show the detailed architecture of a PE. Each PE contains a private vector buffer (PVB) to buffer the dense vector being multiplied, because vector elements are randomly accessed multiple times for all matrix rows in SpMxV. The PE computes the dot product of two vectors by accumulating dot products of sub vectors. This computation includes 5 steps: (1) The PE reads N matrix row elements from the matrix memory and N vector elements based on the sparse indices from the private vector buffer. (2) N multipliers operate simultaneously to obtain N scalar products. (3) An N-input adder tree sums N scalar products to calculate the partial dot product. (4) One more accumulator is used to obtain the complete dot product. (5) The dot product result is written back to global vector memory. The PE is fully pipelined so that one operation can be processed per clock cycle.

With M PEs and N multipliers per PE, this PE array achieves $M \times N$ parallelism for a single SpMxV operation.

5.3 Private Vector Buffer

In each SpMxV PE, N weight elements can be simultaneously accessed in one clock cycle because non-zero values have already been rearranged by CSB encoding format and contiguously stored in matrix memory. However, to access dense vector elements, the PVB needs to support N random memory accesses concurrently. Each BRAM in FPGA provides only two read and/or write ports. Using a single BRAM to buffer dense vectors can not supply N elements from random addresses concurrently. Multi-pumping [25] and vector replication [14] are two alternative solutions. Multi-pumping supplies N elements by running the PEs with N times lower frequency than the BRAM. This approach decreases clock rate significantly. Vector replication provides more ports by creating replicas of the entire vector. Although this approach is simple to implement, it is difficult to scale due to limited on-chip storage resources in FPGA and generally large input/output/state vectors in LSTM. Since each private vector buffer has stored a replicate of the multiplied vector for parallel computing across PEs, further replicating vectors inside each PE is unacceptable.

In order to support random vector accesses at a high bandwidth without replicas inside a PE, we adopt the banking approach to buffer vectors [5]. In this approach, the multiplied vector is also split into banks according to the bank partitioning of matrix rows in BBS. As shown in Figure 6, N banks of vector elements are stored in N independently accessible BRAMs. Therefore, the PVB can provide N elements simultaneously with N bank internal indices (i.e., physical addresses for each BRAM). Weight matrices in LSTMs usually have the same size, so we use a unified N in pruning and configure N as the number of BRAMs in PVB. However, for some LSTMs that have weight matrices of different sizes, different Ns are selected in pruning to find an optimal sparsity, and the largest N is configured as the number of BRAMs in PVB.

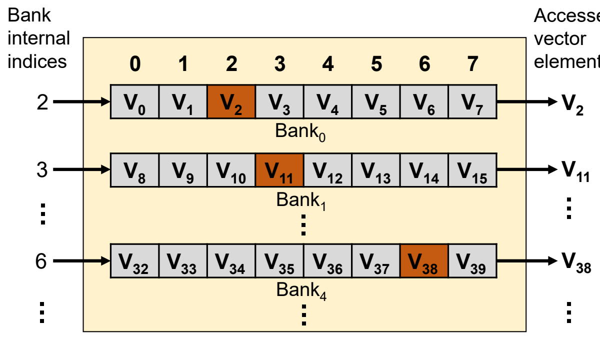

Bank internal indices
Accessed vector elements
0 1 2 3 4 5 6 7
2
V_0 V_1 V_2 V_3 V_4 V_5 V_6 V_7
V_2
Bank_0
3
V_8 V_9 V_{10} V_{11} V_{12} V_{13} V_{14} V_{15}
V_{11}
Bank_1
6
V_{32} V_{33} V_{34} V_{35} V_{36} V_{37} V_{38} V_{39}
V_{38}
Bank_4

Figure 6: Banked private vector buffer.

In some studies, banking is adopted to support random memory accesses to achieve high memory bandwidth [5, 31]. However, due to the irregularity of data accesses, banked memory cannot handle imbalance workloads across banks and concurrent access requests to the same BRAM. Addressing these issues requires additional logic and clock cycles [5, 31]. The biggest difference of our banked private vector buffer is that balanced memory access requests and no memory access conflicts are automatically guaranteed because of the intrinsic bank-balanced property of BBS. The SpMxV PE accesses one and only one element in each BRAM per cycle.

Before a SpMxV operation, the vector to be multiplied requires to be duplicated in each PE's private vector memory to exploit inter-row parallelism. This brings new challenges. First, broadcasting vector elements to various PEs leads to high fan-out and thus results in a low achievable clock frequency. We use a systolic array structure to achieve high clock frequency, similar to [22]. The second is the additional access latency. We double-buffer the private vector buffer for pipelined data transfer and computation.

5.4 EWOP Unit

The EWOP unit performs various element-wise operations on vectors based on the instruction opcode. Vector addition and multiplication generate one result vector by reading two source vectors. Activation functions only read one source vector and apply non-linear functions to it to generate one result vector. The EWOP unit contains M operators operating in parallel for each kind of operations to reduce latency.

5.5 Controller

In the computation flow of LSTM, some SpMxV operations and EWOP operations among different gates can be performed simultaneously. The software compiler analyzes the dependencies and indicates the dependencies to instructions. The controller parallelizes instructions according to their dependent instructions indicated by the software compiler. When the SpMxV unit or the EWOP unit is idle (which means an instruction is finished), the controller checks whether the next instruction has a dependency on the instruction being executed on the other unit. If not, the controller dispatches the next instruction to the idle unit, so that the SpMxV unit and EWOP unit can work simultaneously.

6 EVALUATION

Our evaluation centers around two aspects: the model accuracy of BBS and the hardware efficiency of BBS accelerator.

6.1 Experimental Setup

We implemented the BBS accelerator in System Verilog, synthesized with Quartus Prime 17.1, and evaluated on a custom FPGA PCIe card with an Intel-Arria 10 FPGA [3]. The FPGA has 4 GB DDR3-1600 DRAM external memory. The host CPU is an Intel Xeon E5 2650 processor which is only responsible for data pre-processing and result collecting. The FPGA communicates with the host CPU through a PCIe Gen 3x8 bus, which supports up to 16 GB/s of bidirectional bandwidth.

We evaluate the system with an LSTM language model of the PTB dataset [18] and an LSTM speech recognition model of the TIMIT dataset [7]. PTB dataset is widely used in Natural Language Processing (NLP) research. It consists of 929k training words, 73k validation words, and 82k test words and it has 10k words in its vocabulary. We adopt the LSTM model in [28], which achieves very good quality on the PTB dataset. The small model has 200 hidden units per layer, while the medium one has 650 and the large one has 1,500. The TIMIT corpus is designed to provide speech data for acoustic-phonetic studies. It contains broadband recordings of 630 speakers of eight major dialects of American English, each reading ten phonetically rich sentences. For the LSTM speech recognition model, we set the input size to 153, the hidden size to 1024, and projection size to 512 which are consistent with previous studies [8, 21].

6.2 BBS Model Accuracy

6.2.1 ***Comparison with Unstructured and Block Sparsity.*** We first evaluate the model accuracy of BBS and compare it with unstructured sparsity and block sparsity. Figure 7 and Figure 8 show the sparsity-accuracy trade-off results of various sparsity patterns on PTB and TIMIT data sets, respectively. We use 64 banks in BBS and 4 × 4 blocks in block sparsity. For experiments of the LSTM language model, we use the large model with the hidden size of 1,500. Perplexity is a metric to quantify language model quality. As shown in Figure 7, the perplexity curve of our BBS is very close to the perplexity curve of unstructured sparsity. Both unstructured sparsity and BBS can preserve the perplexity until 80% of weights are pruned away. These two patterns even achieve slightly better model accuracy at around 60% sparsity compared to the dense baseline one. The perplexity of block sparsity starts to increase significantly at 40% sparsity. Experiments on the LSTM speech recognition model show similar results (shown in Figure 8). BBS and unstructured sparsity can achieve 90% sparsity without accuracy loss, while block sparsity can only achieve 70% sparsity. These experimental results demonstrate that BBS has almost the same effectiveness as random sparsity and outperforms block sparsity in terms of achievable accuracy or sparsity during pruning.

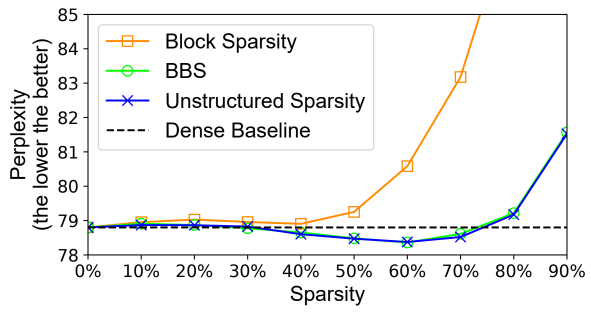


Figure 7: Sparsity-Perplexity trade-off of various sparsity patterns on PTB dataset.

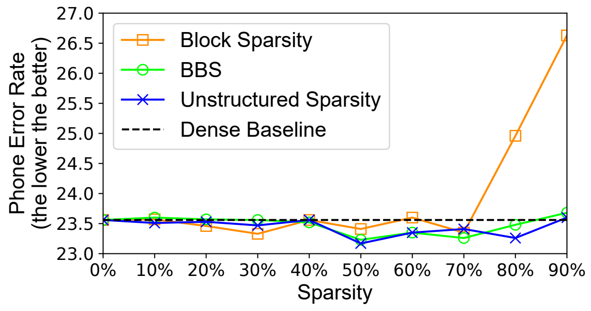


Figure 8: Sparsity - Phone Error Rate trade-off of various sparsity patterns on TIMIT dataset.

6.2.2 ***Sensitivity to Bank Size.*** We further explore the accuracy sensitivity of BBS to the bank size. As a comparison, we also explore the accuracy sensitivity of block sparsity to the block size. Table 2 shows the model accuracy at varying block/bank sizes for the large LSTM language model. As shown, BBS achieves almost the same model accuracy regardless of the change of bank size. For block sparsity, however, increasing the block size adversely affects model accuracy.

6.2.3 ***Quantization on Pruned Model.*** Quantization can achieve more compression rate and hardware efficiency for deep learning models by reducing the number of bits that represents a weight [9, 15]. In this work, we study the accuracy sensitivity of BBS to

Table 2: Perplexity sensitivity to the block size in block sparsity and the bank size in BBS.

Model		Perplexity on Sparsity		
		60%	70%	80%
Block Sparsity	block size: 4×4	80.6	83.2	88.1
	block size: 8×8	82.4	86.4	95.2
	block size: 16×16	83.7	88.3	99.5
BBS	bank size: 25	78.3	78.6	79.4
	bank size: 50	78.4	78.7	79.2
	bank size: 100	78.4	78.6	79.2

quantization bits. We apply the linear quantization method to LSTM models after bank-balanced pruning with 16-bit, 8-bit, and 4-bit fixed points. Both weights and activations are quantized. Table 3 shows the effects of quantization under different bits on the large LSTM language model after bank-balanced pruning. The perplexity is 78.8 for the original dense model and slightly increases to 79.2 after pruning away 80% weights with BBS. 16-bit quantization on the pruned model maintains the same perplexity, while more aggressive quantization deteriorates perplexity.

Table 3: Language model perplexity after quantization under different bits.

Quantization Scheme	Perplexity (%)
float-32 dense model	78.8
float-32 BBS model	79.2
fixed-16 BBS model	79.2
fixed-8 BBS model	79.8
fixed-4 BBS model	143.1

6.3 BBS Accelerator Efficiency

6.3.1 ***Resource Utilization, Clock Rate and Power Consumption.*** Table 4 shows the resource utilization, clock rate and power consumption of our BBS accelerators. The reported results are based on post-fit results from Quartus Prime 17.1. The operator bits (i.e., data precision) is 16-bit since 16-bit is accurate enough to maintain model accuracy. The BBS accelerator sets to M = 64, N = 64, and thus the accelerator contains 64 PEs in the SpMxV unit, and each PE has 64 multipliers executing in parallel. The Intel Arria 10 FPGA contains 1518 DSPs which can be implemented as 3036 multipliers. The LSTM accelerator fully utilizes DSPs for multipliers, and use additional ALMs for extra multipliers. We use M20Ks for the matrix memory, and use MLABs for the private vector buffer because it consists of relatively small memories that require independently accessible ports.

Table 4: Resource utilization, clock rate and power consumption.

ALMs (%)	M20Ks (%)	DSPs (%)
289k (68%)	2509 (92%)	1518 (100%)
Clock Rate (MHz)		Power (Watt)
200		19.1

6.3.2 ***Latency and Throughput.*** Our accelerator is highly efficient even with a batch size of 1, so we measure the latency of our BBS accelerator without batching and calculate the corresponding throughput. For small, medium and large LSTM language models on the PTB data set, we also use three different numbers of banks (16,32,64) to prune models. Pruning away 80% weights incurs no effect on model accuracy. Table 5 shows the latency of one LSTM and its corresponding throughput. The achievable performance increases as the model scale or the number of bank increases because of higher hardware utilization of the underlying PEs. In the case of the large model with 1,500 hidden units and using 64 banks in matrix partitioning, our accelerator takes 4.8us to finish a whole LSTM layer, corresponding to 750.9 GOPS at a batch size of one.

Table 5: Latency and throughput results of running LSTM language networks of various scales and various numbers of banks.

LSTM hidden size	Num of banks	Latency (us)	Throughput (GOPS)
200 (small)	16	1.7	37.3
	32	1.4	43.4
	64	1.3	47.4
650 (medium)	16	4.3	158.8
	32	2.8	238.0
	64	2.1	318.5
1500 (large)	16	13.9	257.7
	32	7.8	458.5
	64	4.8	**750.9**

6.3.3 ***Comparison with state-of-the-art LSTM Accelerators.*** We compare the performance of our BBS accelerator with three state-of-the-art LSTM accelerators on FPGA: ESE [8], C-LSTM [21] and DeltaRNN [6]. These three studies adopt different optimization techniques to reduce computation requirements. ESE [8] uses the weight pruning based compression technique and improve inference efficiency through batching multiple samples, but lacks optimization of irregular memory accesses to reduce latency for a single batch request. C-LSTM [21] represents weight matrices with block-circulant matrices and proposes an accelerator with an FFT-based computing kernel. DeltaRNN [6] uses the delta network algorithm to reduce MxV operations and corresponding weight fetches by skipping dispensable neuron activation changes below a threshold.

Table 6 shows the comparison results. We apply BBS to the same LSTM model on the TIMIT dataset as ESE and C-LSTM adopt. We use the accuracy and performance numbers of ESE, C-LSTM and DeltaRNN reported in their papers. The performance numbers of DeltaRNN are based on GRU which is an optimistic estimation because GRU is simpler than LSTM. With the same model on the same data set, BBS achieves comparable compression rate and model accuracy as ESE and C-LSTM. While our BBS accelerator achieves 2.3x and 3.7x improvement on energy efficiency, and 34.4x and 7.0x speedup on latency (or throughput at a batch size of one) compared to ESE and C-LSTM. The reason why BBS accelerator can achieve better single batch performance than ESE is that it enables the extra dimension of parallelism and addresses the low memory bandwidth issue of irregular memory access in SpMxV.

7 RELATED WORK

Network Compression. Network compression can reduce the memory and computation requirements of a neural network, increase its inference speed and save energy [9]. Compression algorithms mainly include pruning [10], sparsity-inducing regularization [23], quantization [15]. Based on the original sparsity method, further studies propose structured sparsity methods by adding constraints on the locality of non-zero weights [17, 19, 26]. Structured sparsity is more amenable to hardware acceleration compared to unstructured sparsity.

DNN accelerators. Hardware acceleration of DNNs has received significant attention from both industry and academia [4, 12, 16, 29]. Due to the widely adopted pruning-based compression techniques, many accelerators for sparse neural networks are proposed [8, 21, 30]. These works explored specialized sparse matrix multiplication module that directly operates on sparse neural networks. Although these accelerators achieve higher performance than general processors, the irregular computation and memory accesses in sparse neural networks still restrict the maximum parallelism achievable on customized accelerators.

SpMxV accelerators. SpMxV is most computation-intensive and memory-intensive part in LSTM inference. Many FPGA and GPU accelerators for SpMxV have been proposed [2, 5]. However, SpMxV is hard to optimize due to its irregular memory access characteristics. By contrast, neural network pruning methods bring a restricted freedom to define the sparsity structure (e.g. hardware friendly sparsity) in weight matrices. BBS is a kind of structured sparsity pattern that increases hardware efficiency, while incurs negligible loss on model accuracy.

8 CONCLUSION

This paper proposes a novel sparsity pattern, BBS (bank-balanced sparsity), that achieves both high model accuracy for pruning LSTM and high hardware efficiency on FPGA. Our insight into designing BBS is partitioning weight matrix rows into banks for parallel computing and adopting fine-grained pruning inside each bank to maintain model accuracy. Evaluated on speech recognition and language model tasks, BBS achieves almost the same model accuracy as purely unstructured sparsity at various sparsity levels. Our BBS accelerator on FPGA takes advantage of the intrinsic bank-balanced property of BBS, achieving high efficiency even for a batch size of 1. Compared to state-of-the-art FPGA accelerators for LSTM with different compression techniques, BBS accelerator achieves 2.3 ~3.7x improvement on energy efficiency and 7.0 ~34.4x reduction on latency with negligible loss of model accuracy.

9 ACKNOWLEDGEMENTS

We would like to thank Ningyi Xu, Wenqiang Wang, Bojie Li and Yun Wang for all technical discussions and valuable suggestions on improving this paper. We thank the anonymous reviewers for their insightful feedbacks and comments. Shijie Cao was partly supported by National Nature Science Foundation of China (No.61772159).

Table 6: Speedup comparison with state-of-the-art LSTM accelerators

	ESE[8]	C-LSTM[21]	DeltaRNN[6]	Ours
Platform	XCKU060	Virtex-7	XC7Z100	Arria 10 GX1150
Frequency (MHz)	200	200	125	200
Sparsity (%)	88.7	87.5	-	87.5
Quantization	fixed-12	fixed-16	fixed-16	fixed-16
Accuracy Degradation	0.30%	0.32%	-	0.25%
Throughput (GOPS)	282.2	131.1	192.0	304.1
Power (W)	41.0	22.0	7.3	19.1
Energy Efficiency (GOPS/W)	6.9	6.0	26.3	15.9
Latency(us)	82.7	16.7	-	**2.4**
Throughput at batch 1 (GOPS)	8.8	43.7	192.0	**304.1**
Effective Throughput at batch 1 (GOPS)	79.2	349.6	1198.0	**2432.8**

REFERENCES

[1] 2018. Sparse Matrix Formats. https://docs.scipy.org/doc/scipy/reference/sparse.html/. (2018).

[2] Nathan Bell and Michael Garland. 2008. *Efficient sparse matrix-vector multiplication on CUDA*. Technical Report. Nvidia Technical Report NVR-2008-004, Nvidia Corporation.

[3] Adrian M Caulfield, Eric S Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, and others. 2016. A cloud-scale acceleration architecture. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Press, 7.

[4] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, and others. 2018. A Configurable Cloud-Scale DNN Processor for Real-Time AI. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE.

[5] Jeremy Fowers, Kalin Ovtcharov, Karin Strauss, Eric S Chung, and Greg Stitt. 2014. A high memory bandwidth fpga accelerator for sparse matrix-vector multiplication. In *Field-Programmable Custom Computing Machines (FCCM), 2014 IEEE 22nd Annual International Symposium on*. IEEE, 36–43.

[6] Chang Gao, Daniel Neil, Enea Ceolini, Shih-Chii Liu, and Tobi Delbruck. 2018. DeltaRNN: A Power-efficient Recurrent Neural Network Accelerator. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 21–30.

[7] John S Garofolo, Lori F Lamel, William M Fisher, Jonathan G Fiscus, and David S Pallett. 1993. DARPA TIMIT acoustic-phonetic continous speech corpus CD-ROM. NIST speech disc 1-1.1. *NASA STI/Recon technical report n* 93 (1993).

[8] Song Han, Junlong Kang, Huizi Mao, Yiming Hu, Xin Li, Yubin Li, Dongliang Xie, Hong Luo, Song Yao, Yu Wang, and others. 2017. Ese: Efficient speech recognition engine with sparse lstm on fpga. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 75–84.

[9] Song Han, Huizi Mao, and William J Dally. 2015. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149* (2015).

[10] Song Han, Jeff Pool, John Tran, and William Dally. 2015. Learning both weights and connections for efficient neural network. In *Advances in neural information processing systems*. 1135–1143.

[11] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[12] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, and others. 2017. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ACM, 1–12.

[13] Nal Kalchbrenner, Erich Elsen, Karen Simonyan, Seb Noury, Norman Casagrande, Edward Lockhart, Florian Stimberg, Aaron van den Oord, Sander Dieleman, and Koray Kavukcuoglu. 2018. Efficient Neural Audio Synthesis. *arXiv preprint arXiv:1802.08435* (2018).

[14] Charles Eric LaForest, Ming G Liu, Emma Rae Rapati, and J Gregory Steffan. 2012. Multi-ported memories for FPGAs via XOR. In *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*. ACM, 209–218.

[15] Darryl Lin, Sachin Talathi, and Sreekanth Annapureddy. 2016. Fixed point quantization of deep convolutional networks. In *International Conference on Machine Learning*. 2849–2858.

[16] Shaoli Liu, Zidong Du, Jinhua Tao, Dong Han, Tao Luo, Yuan Xie, Yunji Chen, and Tianshi Chen. 2016. Cambricon: An instruction set architecture for neural networks. In *ACM SIGARCH Computer Architecture News*, Vol. 44. IEEE Press, 393–405.

[17] Huizi Mao, Song Han, Jeff Pool, Wenshuo Li, Xingyu Liu, Yu Wang, and William J Dally. 2017. Exploring the regularity of sparse structure in convolutional neural networks. *arXiv preprint arXiv:1705.08922* (2017).

[18] Mitchell Marcus, Beatrice Santorini, Mary Ann Marcinkiewicz, and Ann Taylor. 1999. Treebank-3 LDC99T42. *CD-ROM. Philadelphia, Penn.: Linguistic Data Consortium* (1999).

[19] Sharan Narang, Eric Undersander, and Gregory Diamos. 2017. Block-Sparse Recurrent Neural Networks. *arXiv preprint arXiv:1711.02782* (2017).

[20] Haşim Sak, Andrew Senior, and Françoise Beaufays. 2014. Long short-term memory recurrent neural network architectures for large scale acoustic modeling. In *Fifteenth annual conference of the international speech communication association*.

[21] Shuo Wang, Zhe Li, Caiwen Ding, Bo Yuan, Qinru Qiu, Yanzhi Wang, and Yun Liang. 2018. C-LSTM: Enabling Efficient LSTM using Structured Compression Techniques on FPGAs. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 11–20.

[22] Xuechao Wei, Cody Hao Yu, Peng Zhang, Youxiang Chen, Yuxin Wang, Han Hu, Yun Liang, and Jason Cong. 2017. Automated systolic array architecture synthesis for high throughput CNN inference on FPGAs. In *Design Automation Conference (DAC), 2017 54th ACM/EDAC/IEEE*. IEEE, 1–6.

[23] Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. 2016. Learning structured sparsity in deep neural networks. In *Advances in Neural Information Processing Systems*. 2074–2082.

[24] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, and others. 2016. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144* (2016).

[25] Hasan Erdem Yantir, Salih Bayar, and Arda Yurdakul. 2013. Efficient implementations of multi-pumped multi-port register files in FPGAs. In *Digital System Design (DSD), 2013 Euromicro Conference on*. IEEE, 185–192.

[26] Zhuliang Yao, Shijie Cao, and Wencong Xiao. 2018. Balanced Sparsity for Efficient DNN Inference on GPU. *arXiv preprint arXiv:1811.00206* (2018).

[27] Jiecao Yu, Andrew Lukefahr, David Palframan, Ganesh Dasika, Reetuparna Das, and Scott Mahlke. 2017. Scalpel: Customizing dnn pruning to the underlying hardware parallelism. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ACM, 548–560.

[28] Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. 2014. Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329* (2014).

[29] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. 2015. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 161–170.

[30] Shijin Zhang, Zidong Du, Lei Zhang, Huiying Lan, Shaoli Liu, Ling Li, Qi Guo, Tianshi Chen, and Yunji Chen. 2016. Cambricon-X: An accelerator for sparse neural networks. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*. IEEE, 1–12.

[31] Shijie Zhou, Rajgopal Kannan, Yu Min, and Viktor K Prasanna. 2018. FASTCF: FPGA-based Accelerator for STochastic-Gradient-Descent-based Collaborative Filtering. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 259–268.