SparseNN: An Energy-Efficient Neural Network Accelerator Exploiting Input and Output Sparsity

Jingyang Zhu[1], Jingbo Jiang[1], Xizi Chen[1] and Chi-Ying Tsui[2]
Department of Electronic and Computer Engineering, Hong Kong University of Science and Technology, Hong Kong
Email: [1]{jzhuak, jingbo.jiang, xchenbn}@connect.ust.hk, [2]eetsui@ust.hk

***Abstract*—Contemporary Deep Neural Network (DNN) contains millions of synaptic connections with tens to hundreds of layers. The large computational complexity poses a challenge to the hardware design. In this work, we leverage the intrinsic activation sparsity of DNN to substantially reduce the execution cycles and the energy consumption. An end-to-end training algorithm is proposed to develop a lightweight (less than 5% overhead) run-time predictor for the output activation sparsity on the fly. Furthermore, an energy-efficient hardware architecture, SparseNN, is proposed to exploit both the input and output sparsity. SparseNN is a scalable architecture with distributed memories and processing elements connected through a dedicated on-chip network. Compared with the state-of-the-art accelerators which only exploit the input sparsity, SparseNN can achieve a 10%-70% improvement in throughput and a power reduction of around 50%.**

I. Introduction

Deep Neural Networks (DNNs) are one of the fundamental machine learning models. In the past decade, DNNs have attracted great research interest due to their promising results in various domains, including visual recognition [1], natural language processing [2], and artificial intelligence [3]. Although DNNs can outperform many traditional machine learning models, the large computation and storage requirements pose an obstacle to the extensive deployment in embedded applications. Therefore, considering the limited resources in the embedded platform, both algorithmic and architectural optimizations are required to deliver an energy-efficient solution for DNNs.

In order to avoid overfitting and to be biologically plausible, Rectified Linear Unit (ReLU) is extensively used in DNNs, which leads to a large amount of zeros in the activations of hidden layers. It is reported that there is around 50% sparsity in the contemporary DNN models [4]. The zero activations can be exploited for the design of an energy-efficient implementation as the multiplications and memory access associated with these zero activations can be safely bypassed without affecting the performance. The activation sparsity can be classified into two categories: the input activation sparsity and the output activation sparsity. The input activation sparsity refers to the zero activations within the input feature map, and it is already known when the computation starts. On the other hand, the output activation sparsity, indicating the zero activations in the output feature map, is unknown until the computation of the current layer is finished. In this work, we propose an efficient end-to-end training algorithm to form a run-time predictor that can predict the output activation sparsity before the actual computation of the current layer. The computation overhead of making the prediction is less than 5% of the original feedforward calculation. To efficiently exploit these sparsity to achieve high energy-efficiency, a specialized hardware architecture, SparseNN is proposed. SparseNN is a scalable Network-on-Chip (NoC) based

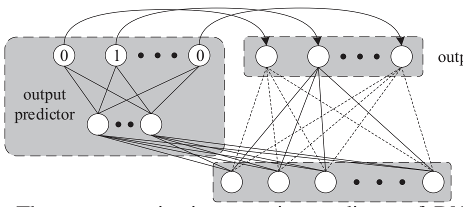

Fig. 1: The output activation sparsity predictor of DNNs. Only the nonzero output activations are scheduled for the feedforward computation (solid line). The predicted zero output activations are bypassed (dotted line).

architecture with distributed processing elements and memories. It can effectively take the advantage of both input and output activation sparsity. From the experimental results, it is shown that the throughput can be improved by 10%$\sim$70% with a power reduction of 50% when these two types of sparsity are jointly utilized.

Traditional deep learning accelerators either take advantage of input activation sparsity [4], [5] or output activation sparsity [6]. The input activation sparsity can be easily exploited using the leading nonzero detector [7] because the input activation vector $a^{(l)}$ is already known in the feedforward pass. On the other hand, the output activation sparsity can be predicted using a lightweight prediction phase as shown in Fig. 1, where the prediction phase is obtained from the truncated Singular Value Decomposition (SVD) of weight matrix $W^{(l)}$ [8]. Specifically, the feedforward pass of DNNs using truncated SVD output predictor can then be summarized as follows:

$$p^{(l+1)} = \mathbf{sign}(U^{(l)}V^{(l)}a^{(l)}) \tag{1}$$

$$a^{(l+1)} = p^{(l+1)} \circ f(W^{(l)}a^{(l)}) \tag{2}$$

where $p^{(l+1)}$ is the sparsity predictor of the output activations and $U^{(l)}$ and $V^{(l)}$ are the first r left-singular vectors and right-singular vectors of $W^{(l)}$, respectively. However, the truncated SVD scheme always looks for a solution with the minimum difference of Frobenius norm [8] and may not be an optimal sparsity predictor. Moreover, $U^{(l)}$ and $V^{(l)}$ are only updated once-per-epoch in the training [8]. The static updating rule limits the flexibility of the backpropagation.

In summary, this work brings the following contributions: (a) A novel end-to-end training algorithm is proposed to generate the output sparsity predictor of the neural network. (b) A scalable NoC based architecture is developed to take advantage of both input and output activation sparsity.

II. Sparsity Predictor: End-to-End Training

In order to address the limitation of the truncated SVD approach, we propose a more powerful end-to-end training algorithm to search for a better solution for the output sparsity predictor. The internal structure of the predictor keeps the same as [8], containing $U^{(l)}$ and $V^{(l)}$. However, they are derived from an end-to-end training phase rather than SVD.

During training we need to backpropagate the gradient of loss ℓ into not only the original feedforward pass but also the sparsity predictor pass. Most of the derivative calculation is straightforward except the passing of the derivative from the predictor to U and V. In Eq. (1), the derivative of the **sign** function will block the output gradient propagate back to U and V during the backpropagation since the value of the derivative is zero for all input except when it is 0. Inspired by [9], we adopt a similar approach using the "straight-through estimator". The basic idea is to approximate the $\mathbf{sign}(x)$ with the piece-wise linear function $\max(-1, \min(1, x))$, whose derivative is 1 when the input is in $[-1, 1]$. The modified gradient calculation in the proposed end-to-end training algorithm is summarized in Alg. 1. To regularize the sparsity of the output activations,

Algorithm 1: Modified backpropagation algorithm.

```
   Input  : Target output $a^{(L)}$
   Output: Gradients w.r.t. $W^{(l)}$, $U^{(l)}$, $V^{(l)}$
1  Compute $\delta^{(L)} = \frac{\partial \ell}{\partial a^{(L)}}$ knowing $a^{(L)}$ and $a^*$;
2  for $l \leftarrow L-1$ to $1$ do
3      $\frac{\partial \ell}{\partial p^{(l+1)}} = \delta^{(l+1)} \circ a_{\text{ori}}^{(l+1)} + \lambda\,\mathbf{sign}(p^{(l+1)})$;
4      $\frac{\partial \ell}{\partial a_{\text{ori}}^{(l+1)}} = \delta^{(l+1)} \circ p^{(l+1)}$;
5      $\theta^{(l)} = \frac{\partial \ell}{\partial U^{(l)} V^{(l)} a^{(l)}} = \frac{\partial \ell}{\partial p^{(l+1)}} \mathbf{1}_{|U^{(l)} V^{(l)} a^{(l)}|<1}$;
6      $\gamma^{(l)} = \frac{\partial \ell}{\partial W^{(l)} a^{(l)}} = \frac{\partial \ell}{\partial a_{\text{ori}}^{(l+1)}} \mathbf{1}_{W^{(l)} a^{(l)}>0}$;
7      $\delta^{(l)} = \frac{\partial \ell}{\partial a^{(l)}} = (W^{(l)})^T \gamma^{(l)}$;
8      $\frac{\partial \ell}{\partial U^{(l)}} = \theta^{(l)} (V^{(l)} a^{(l)})^T$; $\frac{\partial \ell}{\partial V^{(l)}} = (U^{(l)})^T \theta^{(l)} (a^{(l)})^T$;
9      $\frac{\partial \ell}{\partial W^{(l)}} = \gamma^{(l)} (a^{(l)})^T$;
10 end
```

we add the ℓ_1 norm of the sparsity predictor $p^{(l)}$ to the original loss function to optimize both error rate and sparsity level during training as shown in Alg. 1 Line 3.

III. SparseNN: A Scalable Hardware Architecture

After the output sparsity predictor $U^{(l)}$ and $V^{(l)}$ are obtained using the proposed end-to-end training algorithm, a specialized hardware architecture is required to accelerate the inference phase of the DNNs with both input and output sparsity. Traditional Single Instruction Multiple Data (SIMD) microarchitecture like [6][10] is not a scalable solution because the memory bandwidth increases linearly with the SIMD width. To exploit the input sparsity, a distributed hardware accelerator, called EIE, targeted for accelerating DNNs with compressed weights was proposed in [7]. In this work, we enhance the microarchitecture of EIE to exploit both the input and output sparsity.

A. Hierarchical Architecture of SparseNN

SparseNN is a scalable distributed hardware architecture consisting of 64 processing elements (PEs). As shown in Fig. 2, 64 PEs are connected through a dedicated 3-level on-chip H-tree network, which has routers at the leaf-level, the internal-level, and the root-level. The computation of the matrix-vector multiplication is distributed to each PE. More specifically, all rows of the weight matrix $W_{(j,:)}$, and the input activations i_j

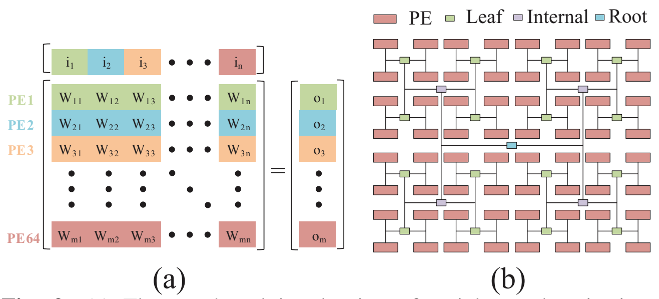

Fig. 2: (a) The row-based interleaving of weights and activations in SparseNN. (b) The hierarchical structure of SparseNN with 64 processing elements and 3-level routing fabrics.

are stored in the k^{th} PE, and output activations o_j are computed by the k^{th} PE, where $j \mod 64 = k$. Since each PE only stores a subset of the input activations, the output activation can not be computed locally until all the input activations are received. As a result, an additional broadcasting stage is required to distribute the local input activations stored in each PE along with the input indices to all PEs through the on-chip network. In order to exploit the input sparsity, only the nonzero activations in the PE will be broadcasted. Each PE starts the local multiplication and accumulation of the input activations as soon as it receives the nonzero input activations from the on-chip network. During the inference computation, SparseNN is first used to calculate the sparsity predictor (i.e. Eq. (1)) and then the original matrix-vector multiplication in Eq. (2) is computed. Since the dimensions of the matrices U, V and W are very different, different schedulings for computing these matrices are needed and will be discussed in Section III.C.

B. On-chip Network Design

In EIE, the timing overhead of broadcasting the input activations to the PEs is hidden by the computations of the input activation received at each PE. However, in a general DNN accelerator, the weight matrix may not necessarily be a square one. For example, the weight matrix of the output layer has smaller number of rows. Very few output activations are mapped to each PE if the weight matrix has a smaller number of rows, and hence for each PE, it only takes a few cycles to consume the received input activation. So if the next input activation does not arrive on time, there will be idling cycles and it affects the overall performance. As a result, the on-chip network of SparseNN is deliberately designed to make sure the activation can arrive every clock cycle to keep the datapath in the PE always busy. Here we adopt a general buffered flow control for the on-chip network. Four nonzero input activations are arbitrated at each level of the routing node. The activation with the smallest index will be granted to the next level while the others will be stored in the buffer at the current node, waiting for the arbitration in the next cycle. The transmission of activations is fully pipelined, and hence each PE can receive the data every cycle. The buffered flow control needs additional temporary storage in the routing node but as shown in Section IV, such overhead is negligible.

C. Computation Schedule for Sparsity Predictor

In the original computation scheduling of EIE, each row of the weight matrix W is distributed to one of the 64 PEs, and the

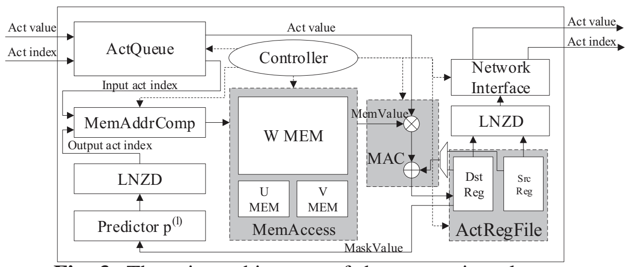

Fig. 3: The microachitecture of the processing element.

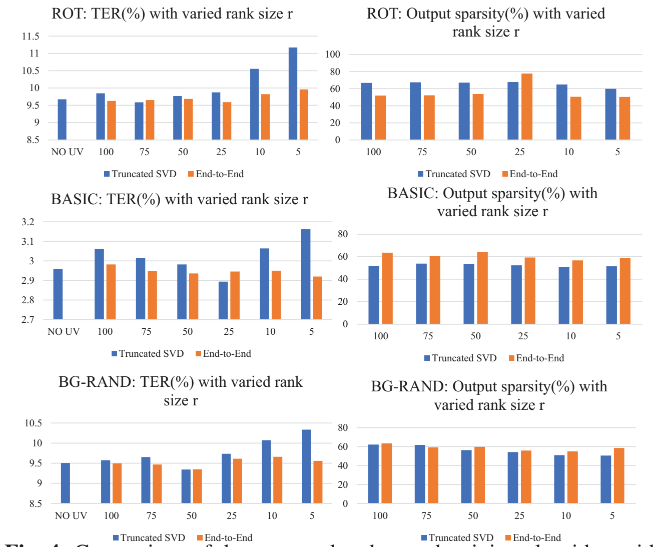

Fig. 4: Comparison of the proposed end-to-end training algorithm with the truncated SVD scheme on the neural network with one hidden layer.

corresponding output activations are calculated. We call this the row-based scheduling. However, if the row number of the weight matrix is smaller than the number of PEs, some of PEs become idle under the row-based scheduling. This situation happens for the V matrix in the sparsity predictor because the rank size r is typically smaller than 64. In order to address this limitation of row-based scheduling of the matrix-vector multiplication, we propose another column-based scheduling. In column-based scheduling, the columns (instead of rows) of V are interleavedly mapped to the 64 PEs. Each PE calculates the partial sum of the output activations o on the right hand side. The accumulation of the partial sums is conducted through the 3-level H-Tree, and the final results of the output activations are stored in the root node. The accumulation operation is embedded in the 4-stage pipelined router. The utilization rate of the V computation is closed to 100% even when the rank size r is as low as 16. The following U computation stage in the sparsity predictor uses the original row-based scheduling as the row number is the same as the number of output activations of W, which is usually much greater than 64.

D. Architecture of PE with the Output Sparsity Bypass

The architecture of the PE in SparseNN is shown in Fig. 3. The datapath of the PE consists of 5 pipeline stages: memory address computation, memory access, multiplication, addition, and write back. The two physical register files are organized as a pair of ping-pong buffers, which alternatively act as the source and destination register files from layer to layer. A complete computation flow of the PE undergoes three matrix-vector computation phases for V, U, and W, respectively. *V computation phase:* The local nonzero input activation a_j and its associated index j are scanned from the source register file which stores all local input activations, and pushed into the datapath. The column-based scheduling is then proceeded to calculate the partial sum in each row. When the partial sum of one row is finished, the result will be sent to the on-chip network for the accumulation. The root node receives the final accumulated result of V computation and broadcasts it back to all 64 PEs. *U computation phase:* With the received V results, the row-based scheduling of U computation is conducted in each PE. In each clock cycle, the PE only processes the head of activation queue, and pushes the locally-stored rows of the U matrix and the results of V computation phase to the datapath. At the end of the U computation phase, the output sparsity predictor $p^{(l)}$ is stored in a dedicated 1-bit register bank. *W computation phase:* The local nonzero input activation a_j and its associated index j are scanned from the source register file, and broadcasted to all the PEs through the H-Tree. After receiving the nonzero input activation and the index, each PE then multiplies the received input activation with the weights of all output activations mapped to the PE that are predicted by the sparsity predictor to be nonzero. In each cycle, the leading nonzero detector of the predictor register bank searches the next nonzero output activation for computation and the intermediate results are stored in the destination registers.

IV. Experimental Results

A. Experimental Setup

We first compare the performance of the proposed end-to-end training algorithm with that of the conventional truncated SVD scheme on MNIST-BASIC dataset (BASIC) along with two challenging variants [11], ROT and BG-RAND. Two different neural network architectures are explored in this work: the 3-layer (with 1 hidden layer) and the 5-layer (with 3 hidden layers). Each hidden layer has 1000 neurons. The nonlinear function in each layer is ReLU.

To evaluate the hardware performance, we implement S-parseNN using Verilog HDL. SparseNN is synthesized using the Synopsys Design Compiler with TSMC 65nm LP library. CACTI 6.5 [12] is used to characterize the memory model. The power consumption of SparseNN is estimated from back-annotating the toggling rate to the synthesized netlist.

B. Performance of the End-to-End Training Algorithm

The test error rate (TER) and the predicted output sparsity $\rho^{(l)}$ of the 3-layer neural network are shown in Fig. 4. From Fig. 4, we can observe that the proposed end-to-end training algorithm of the sparsity predictor scales well with the rank size of the UV predictor. For instance, the TER of the truncated SVD scheme is around 1% larger than the end-to-end training algorithm in ROT dataset when a small rank size is used. The performance difference is mainly because the UV update is static in the conventional truncated SVD scheme and cannot be tuned.

In Table. I, we compare the TER and the output sparsity at each hidden layer of the 5-layer neural network. The network trained by the proposed end-to-end training algorithm preserves

TABLE I: Test error rate and predicted output sparsity ρ of 5-layer neural network with rank size 15

Dataset	Algorithm	TER(%)	$\rho^{(1)}$	$\rho^{(2)}$	$\rho^{(3)}$
ROT	NO UV	**8.54**	N.A.	N.A.	N.A.
	SVD	10.69	**90.74**	28.12	34.27
	End-to-End	8.8	69.41	**64.13**	**71.07**
BASIC	NO UV	2.738	N.A.	N.A.	N.A.
	SVD	2.728	**62.5**	38.15	39.38
	End-to-End	**2.718**	56.34	**65.89**	**66.7**
BG-RAND	NO UV	10.08	N.A.	N.A.	N.A.
	SVD	10.036	51.61	**51.49**	24.01
	End-to-End	**10.03**	**52.79**	48.23	**41.44**

a similar (or even better) accuracy to the SVD approach, but with a higher average sparsity ratio of the hidden layers. It is due to the output sparsity is considered in the end-to-end training algorithm as we use the ℓ_1 regularization in the cost function.

C. Performance of the SparseNN

The design parameters of the mircroarchitecture of the proposed architecture, SparseNN, are listed in Table. II. The microarchitecture of SparseNN is inspired by EIE. For instance, the total on-chip weight memory is 8MB, and the maximum number of activations in each layer is 4K. The critical path delay of SparseNN is 2ns and hence it can run at 500MHz.

TABLE II: The mircroarchitecture parameters of 64-PE SparseNN

Micro-architectural parameters	Value
Quantization scheme	16-bit fixed point
On-chip $W/U/V$ memory per PE	128KB/8KB/8KB
Activation register no. per PE	64

The area breakdown of SparsNN is listed in Table. III. The routing nodes occupy only a small fraction (less than 1%) of the total area, and the major area contributors are the PEs. The main reason is the large on-chip SRAMs for W, U, and V in each PE, which take around 95% of the overall area.

TABLE III: The area breakdown of SparseNN by component

	Area (μm^2)	(%)
Combinational	1,716,373	(2.4%)
Buf/Inv	199,038	(0.2%)
Non-combinational	2,068,996	(2.6%)
Macro (Memory)	74,426,310	(94.8%)

The results on the execution cycle and the power consumption of SparseNN on the three benchmarks are shown in Fig. 5. When UV predictor is not used, SparseNN is the same as the conventional EIE architecture which only exploits the input activation sparsity. From Fig. 5, it can be seen that the improvement in the number of execution cycles with the output sparsity varies from layer to layer. For the 1st hidden layer, the reduction of cycles ranges from $10\%\sim31\%$. The inputs to the 1st hidden layer are the same for the UV enabled and the UV disabled networks, and hence the improvement of throughput only comes from the output sparsity. The difference of the throughput improvement at different layers and benchmarks is due to the difference in predicted output sparsity. In addition, the number of nonzero output activations predicted by the sparsity predictor also varies from PE to PE. For the remaining hidden layers, the reduction of cycles can be as high as 70%. The predicted output sparsity of the previous layer will increase the input sparsity of the current layer. Therefore, the throughput

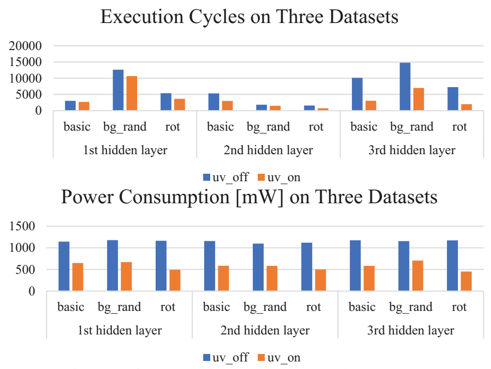

Fig. 5: Comparison of execution cycles and power consumption on three datasets using the 5-layer DNN. The results are organized in layer-wise, where *uv_on* and *uv_off* represent the output sparsity predictor of SparseNN is enabled and disabled, respectively.

is jointly improved by the input sparsity as well as the output sparsity. The improvement in power consumption with output sparsity is almost uniform among all datasets and all hidden layers: around 50%. The reasons for the power reduction are twofold: the number of access to the large W memory decreases with the output sparsity, and the access energy to the U, V memory during sparsity prediction phase is small.

V. Conclusion

In this work, we first propose an end-to-end training algorithm to obtain the U and V matrices for the sparsity predictor from the backpropagation. The scalability with rank and the predicted sparsity are better than the traditional truncated SVD scheme. Then, a specialized architecture, SparseNN, is developed to exploit both the input and output sparsity. Our evaluations demonstrate that with the output sparsity, the throughput of SparseNN can be improved by 10% to 70% while the power consumption is approximately reduced by half.

References

[1] Alex Krizhevsky et al. Imagenet classification with deep convolutional neural networks. In *NIPS*, pages 1097–1105, 2012.
[2] Dario Amodei et al. Deep speech 2: End-to-end speech recognition in english and mandarin. In *ICML*, pages 173–182, 2016.
[3] David Silver et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.
[4] Jorge Albericio et al. Cnvlutin: ineffectual-neuron-free deep neural network computing. In *ISCA*, pages 1–13. IEEE, 2016.
[5] Song Han et al. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.
[6] Jingyang Zhu et al. Lradnn: High-throughput and energy-efficient deep neural network accelerator using low rank approximation. In *ASP-DAC*, pages 581–586. IEEE, 2016.
[7] Song Han et al. Eie: efficient inference engine on compressed deep neural network. In *ISCA*, pages 243–254. IEEE Press, 2016.
[8] Andrew Davis et al. Low-rank approximations for conditional feedforward computation in deep neural networks. *arXiv preprint arXiv:1312.4461*, 2013.
[9] Matthieu Courbariaux et al. Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. *arXiv preprint arXiv:1602.02830*, 2016.
[10] Paul N Whatmough et al. 14.3 a 28nm soc with a 1.2 ghz 568nj/prediction sparse deep-neural-network engine with $>$ 0.1 timing error rate tolerance for iot applications. In *ISSCC*, pages 242–243. IEEE, 2017.
[11] Hugo Larochelle et al. An empirical evaluation of deep architectures on problems with many factors of variation. In *ICML*, pages 473–480. ACM, 2007.
[12] Naveen Muralimanohar et al. Cacti 6.0: A tool to model large caches. *HP Laboratories*, pages 22–31, 2009.