Generic, Sparse Tensor Core for Neural Networks

Xiaolong Wu
Purdue University
West Lafayette, IN, USA
wu1565@purdue.edu

Yang Yi
Virginia Tech
Blacksburg, VA, USA
cindy_yangyi@vt.edu

Dave (Jing) Tian
Purdue University
West Lafayette, IN, USA
daveti@purdue.edu

Jiajia Li
Pacific Northwest National Laboratory
Richland, WA, USA
Jiajia.Li@pnnl.gov

***Abstract*—Sparse neural networks attract broad attention to model compression, fast execution, and power reduction. The state-of-the-art designed sparse tensor cores for NVIDIA GPUs target on structured and static sparsity, and do not support generic or dynamic sparsity well. We design a sparse tensor core architecture to support generic sparsity pruning with a novel hybrid and blocked sparse matrix storage format, HB-ELL, which saves computation and storage while keeping the most significant elements, as well as supporting dynamic sparsity for data flow in neural networks. We achieve better performance in our preliminary results than the state of the art on an NVIDIA GPU simulator.**

***Index Terms*—sparse matrix, pruning, sparse neural networks, NVIDIA GPU, tensor core**

I. Introduction

Deep neural networks have become popular for a wide range of applications, such as image recognition, natural language processing, and auto-piloting [1]. Sparsity shows the importance in model compression, fast execution, and storage space saving, and attracts more attention from the community [2]–[6]. Two types of sparsity exist in a neural network, static and dynamic. Static sparsity refers to the sparsity in weight parameters that do not depend on the input data, while dynamic sparsity includes the sparsity in input data and activations.

There are various approaches to weight pruning of neural networks [3]–[7]. From the study [6], magnitude pruning could achieve the upper bound of model accuracy. Thus, we focus on magnitude pruning and categorize it as generic, unified, and structural sparsifying. Generic sparsifying, such as top-K pruning [8]–[11], gets a very high compression ratio due to keeping the most significant elements. However, the generated sparse matrices bring challenges for the algorithm performance on GPUs because of the random positions of non-zeros. Unified sparsifying prunes a weight matrix by the units of rows or columns rather than the units of elements in generic sparsifying. This coarse-grained pruning could harm the model accuracy caused by removing whole rows/columns with significant values. Structural sparsifying is in the middle, consisting of block or other special structures. The sparse tensor core work [2] designed a structured pruning method, named Vectorsparse pruning, obtaining around 75% sparsity. OpenAI company developed an efficient block-sparse matrix computation on GPUs, with small block sizes of 8×8, 16×16, and 32×32 [3].

We investigate generic sparsity in this work because of the following reasons: 1) it obtains the best model accuracy from the work [2] by keeping the most significant values; 2) it generally achieves the best compression ratio among the above three sparsifying categories; 3) the small amount of pruned non-zero elements, which could be less than 1%, is especially important for performance-efficient sparse operation implementation to reduce both time complexity and memory footprints; 4) it is popular in input data from applications, thus an efficient generic sparse implementation is beneficial for both dynamic and static sparsity in a neural network.

The effectiveness of a generic sparsifying method largely depends on the performance-efficient implementation of sparse operations. NVIDIA Turing and Ampere tensor core architecture supports highly efficient dense matrix-matrix multiplication with low floating-point precision and short-bit integers. The NVIDIA Ampere A100 GPU supports no more than 50% sparsity with $2\times$ extra performance speedup, allowing 2:4 structured sparsity, two non-zero values in every four-entry vector. Sparse tensor core work [2] improves the tensor core architecture by supporting 4:16 structured sparsity (up to 75%), which achieves good performance speedup with less than 1% accuracy loss. Our work extends the work [2] to support generic sparisity and customizes the tensor core architecture accordingly.

Overall, our main contributions are listed below:

- We propose a new sparse matrix format – hybrid and blocked ELLPACK, named HB-ELL, for compressed storage and efficient computation. We support dense and sparse blocks by representing them differently. (Sections III-A and III-B)
- We extend the instruction set of tensor core for an algorithm-data structure-architecture co-design for sparse matrix-dense matrix multiplication. (Section III-C)
- Our approach achieves up to $3.3\times$ speedup compared to the state-of-the-art SpMM. (Section IV)

II. Background

A. Neural Network Sparsifying

Big data and large-scale tasks such as image classification have forced the construction of large artificial neural networks. Current deep learning networks can contain over a billion parameters [12], which incur large resources to train these parameters. Recent research shows that the large amount of parameters can be compressed by sparsifying/pruning methods. Prior studies [13] have shown that neural networks over-parameterized, thus resulting in significant redundancy. Pruning has been introduced as a solution to this problem with little accuracy loss.

In general, there are three kinds of sparsifying methods, which are generic sparsifying, unified sparsifying, and structural sparsifying. The generic sparsifying method is illustrated in Figure 1(a), which chooses the largest K elements (here $K = 4$) in absolute value of the matrix (highlighted in red). The key elements are retained while the remaining values are changed to zero. The unified sparsifying is shown in Figure 1(b). Different from generic sparsifying, the values of an entire column is reserved or removed as a whole while the rest are given the value of zeros. Structural sparsifying as shown in Figure 1(c) picks the largest value of every 4-dim vector and all the remaining elements are forced to zeros.

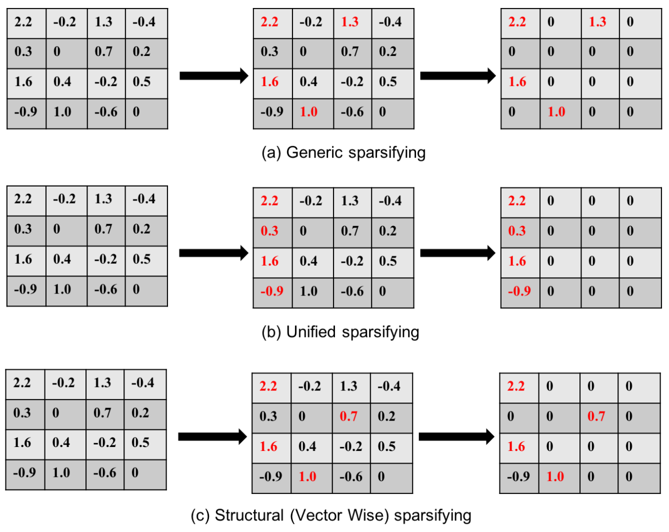

Fig. 1. Generic, unified, and vector-wise sparsifying for 4×4 matrix.

Our work emphasizes on generic sparsifying due to the reasons mentioned in Section I. We use the top-K pruning from the work [8]–[10] for generic sparsifying, where the largest K values are retained.

B. NVIDIA Tensor Core

NVIDIA Volta architecture [14], [15] introduced the tensor cores for dense matrix multiplication. The computation units of tensor cores are dot product (DP) units, and one tensor core has 32 DP units. Each DP unit computes a 4-dim vector dot product per cycle; thus one tensor core completes a 4×4×4 matrix multiplication per cycle. To program tensor cores, NVIDIA provides CUDA Warp Matrix Multiply and Accumulate (WMMA) API with load, store, and matrix multiplication instructions. In Volta GPU, each SM has four subcores, and each subcore contains two tensor cores. The 32 threads in a warp are divided into 8 threadgroups, four threads in a threadgroup together compute 4×4 tile multiplications. One worktuple consisting of two threadgroups is responsible for computing 8×8 tile of matrix D. A WMMA operation will be compiled into four sets of machine-level Hardware Matrix Multiply (HMMA) instructions. Tensor cores support two types of execution mode: single mode that matrix A, B, and C are all in FP16 type and mixed mode that A and B are in FP16, matrix C is in FP32 type.

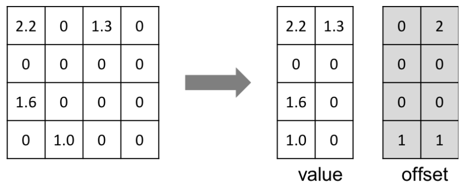

Fig. 2. The ELL sparse matrix format of the generic sparsified matrix in Figure 1.

C. Sparse Format

We describe the ELL format for sparse matrices, used in the work [2], as our basic format. ELL, short for the ELLPACK/ITPACK format [16], [17], packs all non-zeros towards left, then stores the packed dense matrix. As illustrated in Figure 2, "value" array stores this dense matrix with floating-point values and necessary zero-filling, and "offset" stores the corresponding column indices, colored as gray. ELL uses the maximum row length as the number of columns for "value" and "offset" to cover all non-zero entries, e.g., 2 in Figure 2. The offsets of the filled zeros are equal to the last column index of that row. We choose the ELL format to make the extension because: 1) the work [2], employing ELL with the fixed number of columns generated by Vectorsparse pruning, shows good performance with their instruction set extension and micro-architecture design on sparse tensor cores; 2) ELL is a suitable format for GPUs because of its good SIMD parallelization feature according to prior studies [18], [19].

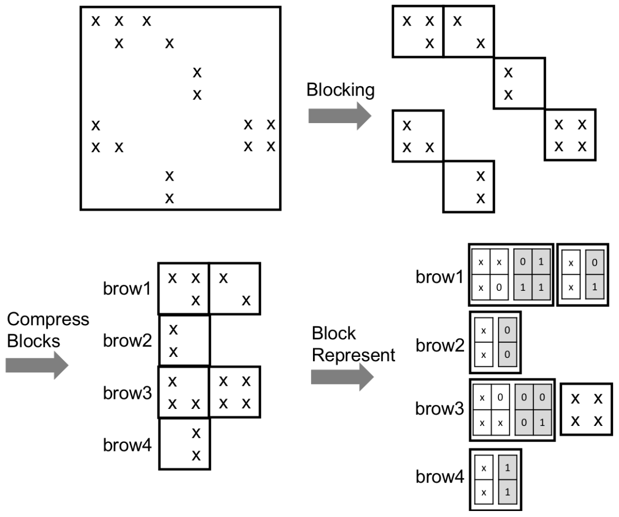

Fig. 3. The HB-ELL format for an example 8×8 sparse matrix using 2×2 blocks.

III. Generic, Sparse Tensor Core Design

To efficiently implement the generic sparsity, we describe our novel data structure and the algorithm for a sparse matrix-dense matrix multiplication first, and then introduce the corresponding extension of the NVIDIA WMMA instruction set.

A. Data Structure

We design a sparse format for a generic sparse matrix, inspired by the work [2], [19]. Our hybrid and blocked ELL format, named HB-ELL, is an extension of the work [19] by using the ELL format for sparse blocks while dense storage for dense blocks. Figure 3 depicts HB-ELL using a simple 16×16 matrix.

We first generate a sparse matrix using the top-K pruning where non-zero positions are unpredictable. HB-ELL first splits a sparse matrix into blocks, where only non-zero blocks (at least one non-zero entry shown in a block) are saved. In Figure 3, only six non-zero blocks are saved in a 16×16 sparse matrix. All non-zero blocks are organized by blocked rows (*brow1-4* in Figure 3, sized 16). ELL or dense storage is used depending on the sparsity of a non-zero block. Using ELL for small blocks not only saves storage but also limits the offset index range thus can be represented in less bits. For example, only 2 bits are needed in Figure 2, as indices are in the range [0,3). In Figure 3, one dense block is stored in dense storage, i.e., row-major storage in our work; the rest five blocks are all sparse, so we use ELL for each. Actually, in our work, we use dense storage for the blocks with sparsity smaller than 25%, because dense storage should yield better performance than a sparse format on relative dense data, according to studies [2], [19]. We observe different numbers of columns in the ELL representation of blocks. The ELL representations of three blocks only have one ELL column, while those of two blocks have two ELL columns. Thus, an extra array $bcols$ is stored to save this information. For the storage, we save all blocks in a row-major order, e.g., blocks in *brow1*, blocks in *brow2*, etc. In this work we set the block size as 16 for the ease of tensor core programming.

Our work distinguishes from the work [2] by non-uniform column lengths in the whole data. Only one column length, e.g., 4, is employed in [2], which means every vector-wise sparsity generates a 16×4 ELL matrix. However, due to the pruning and the randomness of generic sparsity, this uniformed number of columns has to be the maximum row length to cover all non-zeros (as mentioned in [2]). Thus, zero-filling is necessary for the ELL format to store and compute these zeros, which increases memory footprints and time complexity. HB-ELL supports non-uniform number of columns, and every matrix block has its affiliated column length saved in $bcols$, reducing the zero-filling ratio.

B. Sparse Algorithm Design

Sparse matrix-dense matrix multiplication (SpMM) is a common computational kernel in neural networks [2]. We design SpMM algorithm based on the HB-ELL format. Figure 4 plots the multiplication between two 8×8 matrices

Algorithm 1 Sparse matrix-dense matrix multiplication (SpMM) using HB-ELL sparse format and row-major dense storage.

```
Require: Sparse matrix $\mathbf{A} \in R^{I \times K}$ in HB-ELL format, dense matrix $\mathbf{B} \in R^{K \times J}$, constants $\alpha$ and $\beta$, block size $BS$;
Ensure: Sparse matrix $\mathbf{C} \in R^{I \times J}$;
     // Compute $\mathbf{C} \leftarrow \alpha\mathbf{AB} + \beta\mathbf{C}$
 1: Registers $A_{reg}, B_{reg}, C_{reg}, ACC_{reg} \leftarrow 0$
 2: $w_M = (bid.x * bdim.x + tid.x)/WARP\_SIZE$;
 3: $w_N = (bid.y * bdim.y + tid.y)$;
     // Parallelize ELL blocks
 4: $cRow = bridx[brow\_ptr[w_M]]$
 5: $cCol = w_N \times BS$
 6: for $i$ in $[brow\_ptr[w_M], brow\_ptr[w_M + 1])$ do
 7:     $bRow = bcidx[i]$
        // Customized WMMA load inst.
 8:     Load $A_{reg} \leftarrow \mathbf{A} + brow\_valptr[i]$
 9:     Load $B_{reg} \leftarrow \mathbf{B} + bRow * J + cCol$
        // Customized WMMA compute inst.
10:     Compute $ACC_{reg} \leftarrow A_{reg} \times B_{reg} + ACC_{reg}$
11: end for
12: Load $C_{reg} \leftarrow \mathbf{C} + cRow * J + cCol$
13: for $i$ in $[0, I \times J)$ do
14:     $C_{reg} = \alpha \times ACC_{reg} + \beta \times C_{reg}$
15: end for
16: Store $C_{reg} \rightarrow \mathbf{C} + cRow * J + cCol$
17: Return $\mathbf{C}$;
```

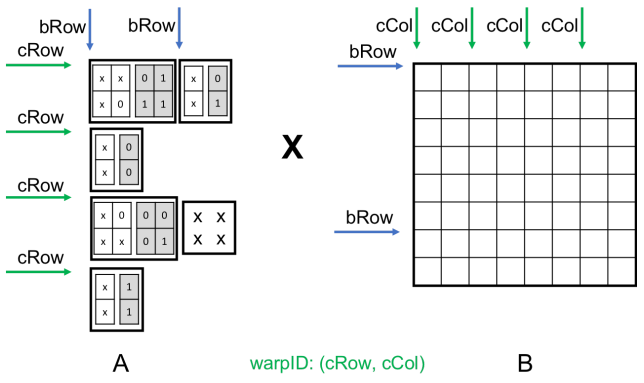

Fig. 4. Sparse matrix-dense matrix multiplication (SpMM) on 8×8 matrices with 2×2 blocks. The sparse matrix $\mathbf{A}$ is from Figure 3.

$\mathbf{A}$ and $\mathbf{B}$. Aligning with hardware tensor core that one warp is in charge of one dense block of matrix $\mathbf{C}$, our algorithm uses the same approach. $cRow$ and $cCol$ index the blocks of $\mathbf{C}$. The difference is that a warp multiplies a sequence of sparse blocks of $\mathbf{A}$ with corresponding dense blocks of $\mathbf{B}$. For example, the first blocked row multiplies two sparse blocks with the corresponding $\mathbf{B}$, indexed by $bRow$ in Figure 4; the second blocked row multiplies one sparse block with the corresponding $\mathbf{B}$. Due to the generic sparsity of our matrix $\mathbf{A}$, each blocked row could have different number of blocks, which could lead load imbalance issues to be considered for further improvement.

The details of SpMM based on the HB-ELL format is shown in Algorithm 1. Four auxiliary small arrays are saved: $brow\_ptr$, $brow\_valptr$, $bridx$, and $bcidx$, along with $bcols$. Assume nb stands for the number of blocks, 6 in Figure 3, nbr is the number of blocked rows, 4 in Figure 3. $bcols$, a size-nb

array, represents the number of columns of each block; $bridx$, a size-nb array, represents the starting row index of each block; $bcidx$, a size-nb array, represents the starting column index of each block; $brow\_valptr$, a size-$nb+1$ array, represents the starting data position of each block; $brow\_ptr$, a size-$nbr+1$ array, points the starting block index of each blocked row. We use $brow\_ptr$ to locate all blocks in each blocked row and loop over them at Line 6. Each block uses the information in $brow\_valptr$, $bridx$, and $bcidx$. $bridx$, and $bcidx$ are used to calculate $cRow$ and $bRow$ to index the correct location of rows and columns, because zero blocks or rows are not saved in HB-ELL. $brow\_valptr$ specifies the location of the "value" array of matrix $\mathbf{A}$.

Algorithm 1 computes the blocks in a blocked row with the corresponding blocks in B, and then accumulate the results to ACC_{reg}. Load of sparse matrix $\mathbf{A}$ is located by $brow\_valptr$. After the computation, matrix $\mathbf{C}$ is loaded and then applied with constants α, β. Finally, the matrix $\mathbf{C}$ is stored back to memory from C_{reg}. Because of our proposed HB-ELL format, the "load $\mathbf{A}$" and "computation" cannot be supported by CUDA. We introduce necessary instructions in the next section. Note that this SpMM algorithm is a basic implementation. Advanced optimization techniques could further improve the performance.

C. *Instruction Set Extension*

For the algorithm-data structure-architecture co-design, we extend the tensor core PTX instruction set [14] to generic, sparse WMMA instructions. For the hybrid sparse and dense feature of HB-ELL, we provide unified instructions support.

We first realized the sparse tensor core work [2] to support the SpMM algorithm based on the ELL format by providing the following sparse WMMA (SWMMA) instructions.

- Load A: swmma.load.a.PW ra, [pa];
- Load Offset: swmma.load.offset.PW ro, [po];
- Math: swmma.mma.f32.f32.PW rd, ra, rb, rc, ro;

The PW in these instructions represents the number of non-zero elements in rows of the sparse matrix $\mathbf{A}$, which is chosen as 4 in [2]. We extend the three instructions to support PW as 8, 12, 16 cases by providing a unified instruction interface as in the list. Due to the hybrid sparse and dense blocked storage in HB-ELL, for the blocks with $PW = 4, 8$ we call our sparse instructions underneath; while for the blocks with $PW = 12, 16$, we call original dense WMMA instructions. We use 8-bit to represent each element of the offset array for easy implementation, which could be further reduced to 4-bit in the future.

IV. Experiments

A. *Platform Configuration*

We design our generic sparse Tensor Core on GPGPU-Sim simulator v3.2.2 [14], [20], to evaluate the performance. [1] The simulator is configured to model a Tesla V100 GPU containing 80 SMs and 640 Tensor Cores. We use GCC 6.5.0 and CUDA

[1]We use the "dev" branch for the state-of-the-art development.

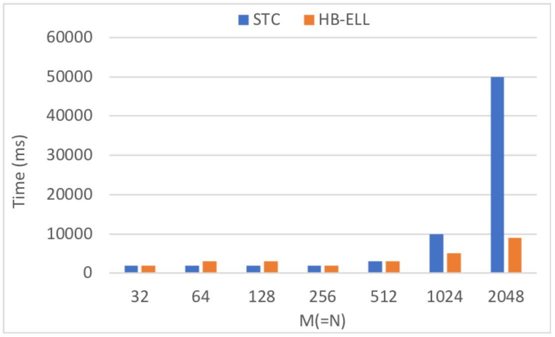

Fig. 5. The execution time of SpMM using our HB-ELL versus sparse tensor core approach [2].

9.1. In our experiments, we set 512 threads per thread block, row-major for the dense matrix storage, and sparse/dense block size as 16x16.

B. *Results*

Since SpMM is a critical bottleneck of a neural network, we test the execution cycles of SpMM of our HB-ELL-based Algorithm 1 with the extended WMMA instructions. We also implemented the SpMM algorithm described in the work [2] by supporting their SWMMA instructions as our baseline. For SpMM $\mathbf{C} \leftarrow \alpha\mathbf{AB}+\beta\mathbf{C}$, α, β are set to 1.0. Our experiments vary the size of M with $M = N$, and K is set to 16. We use FP16 for matrices $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ in evaluation. The sparsity of $\mathbf{A}$ is set to 10%, and its non-zero values and positions are randomly generated. We observe more than $3\times$ speedup over the state-of-the-art sparse tensor core work [2]. Notice that for $M = N \leq 512$, the number of cycles does not change much. This is caused by our setting of 512 threads per thread block. For matrices requiring less than 512 threads, the time does not vary a lot by simulating only one thread block. Besides, for these small matrices, our HB-ELL behaves a bit worse than sparse tensor core work. This shows that HB-ELL is more beneficial for large data. Due to the slow running of the simulator, we have not done testing on larger matrices which will be our future work. In the future, we will also introduce more accurate timing module to conduct better comparison.

V. Conclusion

We designed a generic, sparse tensor core for sparse matrix-matrix multiplication and neural networks by introducing a new hybrid and blocked ELL format (HB-ELL) and extending NVIDIA WMMA instruction set. Better performance is achieved by simulating our method on GPGPU-Sim compared to the state-of-the-art work. In the future, we will also compare to the hardware sparsity support with NVIDIA Ampere A100 architecture and verify the effectiveness of our HB-ELL format on general CUDA cores. Besides, we will also pursue a scheduling strategy for better load balance among warps. Our methods will be applied to real neural network models as well.

Acknowledgment

This research is partially funded by the US Department of Energy, Office for Advanced Scientific Computing (ASCR) under Award No. 66150: "CENATE: The Center for Advanced Technology Evaluation" and the Laboratory Directed Research and Development program at PNNL under contract No. ND8577. Pacific Northwest National Laboratory (PNNL) is a multiprogram national laboratory operated for DOE by Battelle Memorial Institute under Contract DE-AC05-76RL01830.

References

[1] I. Y. Noy, D. Shinar, and W. J. Horrey, "Automated driving: Safety blind spots," *Safety science*, vol. 102, pp. 68–78, 2018.

[2] M. Zhu, T. Zhang, Z. Gu, and Y. Xie, "Sparse tensor core: Algorithm and hardware co-design for vector-wise sparse neural networks on modern gpus," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 359–371.

[3] S. Gray, A. Radford, and D. P. Kingma, "Gpu kernels for block-sparse weights," *arXiv preprint arXiv:1711.09224*, vol. 3, 2017.

[4] X. Sun, X. Ren, S. Ma, and H. Wang, "meprop: Sparsified back propagation for accelerated deep learning with reduced overfitting," *arXiv preprint arXiv:1706.06197*, 2017.

[5] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li, "Learning structured sparsity in deep neural networks," in *Advances in neural information processing systems*, 2016, pp. 2074–2082.

[6] T. Gale, E. Elsen, and S. Hooker, "The state of sparsity in deep neural networks," *arXiv preprint arXiv:1902.09574*, 2019.

[7] T. Gale, M. Zaharia, C. Young, and E. Elsen, "Sparse gpu kernels for deep learning," *arXiv preprint arXiv:2006.10901*, 2020.

[8] S. Han, J. Kang, H. Mao, Y. Hu, X. Li, Y. Li, D. Xie, H. Luo, S. Yao, Y. Wang *et al.*, "Ese: Efficient speech recognition engine with sparse lstm on fpga," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2017, pp. 75–84.

[19] J. W. Choi, A. Singh, and R. W. Vuduc, "Model-driven autotuning of sparse matrix-vector multiply on gpus," *ACM sigplan notices*, vol. 45, no. 5, pp. 115–126, 2010.

[9] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "Eie: efficient inference engine on compressed deep neural network," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 243–254, 2016.

[10] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv preprint arXiv:1510.00149*, 2015.

[11] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen, "Cambricon-x: An accelerator for sparse neural networks," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–12.

[12] J. Dean, G. S. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Y. Ng, "Large scale distributed deep networks," in *NIPS*, 2012.

[13] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," in *Advances in neural information processing systems*, 2015, pp. 1135–1143.

[14] M. A. Raihan, N. Goli, and T. M. Aamodt, "Modeling deep learning accelerator enabled gpus," in *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2019, pp. 79–92.

[15] Z. Jia, M. Maggioni, B. Staiger, and D. P. Scarpazza, "Dissecting the nvidia volta gpu architecture via microbenchmarking," *arXiv preprint arXiv:1804.06826*, 2018.

[16] J. R. Rice and R. F. Boisvert, *Solving elliptic problems using ELLPACK*. Springer Science & Business Media, 2012, vol. 2.

[17] R. G. Grimes, D. R. Kincaid, and D. M. Young, *ITPACK 2.0 user's guide*. Center for Numerical Analysis, Univ., 1979.

[18] N. Bell and M. Garland, "Implementing sparse matrix-vector multiplication on throughput-oriented processors," in *Proceedings of the conference on high performance computing networking, storage and analysis*, 2009, pp. 1–11.

[20] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing cuda workloads using a detailed gpu simulator," in *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, 2009, pp. 163–174.