Dual-side Sparse Tensor Core

Yang Wang*†‡, Chen Zhang**‡, Zhiqiang Xie*§, Cong Guo*¶, Yunxin Liu‡, Jingwen Leng¶
† *University of Electronic Science and Technology of China,* ‡ *Microsoft Research*
§ *ShanghaiTech University,* ¶ *Shanghai Jiao Tong University*
{*t-yangwa, zhac, yunxin.liu*}*@microsoft.com*
xiezhq@shanghaitech.edu.cn, {*guocong, leng-jw*}*@sjtu.edu.cn*

***Abstract*—Leveraging sparsity in deep neural network (DNN) models is promising for accelerating model inference. Yet existing GPUs can only leverage the sparsity from weights but not activations, which are dynamic, unpredictable, and hence challenging to exploit. In this work, we propose a novel architecture to efficiently harness the dual-side sparsity (i.e., weight and activation sparsity). We take a systematic approach to understand the (dis)advantages of previous sparsity-related architectures and propose a novel, unexplored paradigm that combines outer-product computation primitive and bitmap-based encoding format. We demonstrate the feasibility of our design with minimal changes to the existing production-scale inner-product-based Tensor Core. We propose a set of novel ISA extensions and co-design the matrix-matrix multiplication and convolution algorithms, which are the two dominant computation patterns in today's DNN models, to exploit our new dual-side sparse Tensor Core. Our evaluation shows that our design can fully unleash the dual-side DNN sparsity and improve the performance by up to one order of magnitude with small hardware overhead.**

***Index Terms*—Neural Networks, Graphics Processing Units, General Sparse Matrix-Matrix Multiplication, Convolution, Pruning**

I. Introduction

As deep learning is widely deployed, there is an emerging need to support billions of queries of inference per day, which is rapidly outpacing training in data centers [25]. Especially, many AI applications have a stringent constraint of service level agreement. Running models at high-scale, low-latency, and high energy efficiency has always been extremely desirable. Model compression and sparsification have become critical optimizations to reduce the number of parameters as well as arithmetic operations and to improve the computational and energy efficiency on various hardware platforms, including ASICs [3], [8], [17], [23], [28], [49], [69]–[71], GPUs [14], [15], [21], [65], [66], and FPGAs [2], [7], [22], [40], [68].

Realizing the acceleration potential of sparse neural networks, GPU vendors have introduced architectural support to exploit this opportunity. In particular, sparse Tensor Core [42], [45], [72] is newly invented to leverage the weight sparsity in DNN models. The latest NVIDIA Ampere architecture [45] [42] introduces a new sparse Tensor Core design with a fixed 50% weight pruning target and achieves a better accuracy and performance trade-off [7], [66].

Besides the weight sparsity, DNN models also exhibit another form of sparsity called *activation* sparsity, which is introduced by activation functions [1] and is widely embedded in activation feature maps, for both computer vision [56] and natural language processing [13] tasks. Many previous works have reported a high activation sparsity ranging from 50% to 98% [3], [49]. However, the current sparse Tensor Core is only able to take advantage of weight sparsity but not activation sparsity. How to effectively leverage the activation sparsity remains an open and challenging research problem, because the activation sparsity dynamically changes with the input and cannot be pre-determined and controlled by the pruning method such as the vector-based pruning [72].

TABLE I: Technical differences to related work.

	Inner-product	Outer-product	Misc
CSR	[29], [72]	[48], [49], [70]	[58]
Bitmap	[17]	**Our work**	-

Although there are prior efforts that tackle the dual-side sparsity in the context of ASIC designs, they cannot be directly adopted by GPUs. The wide applicability of GPUs requires the support of both sparse general matrix-matrix multiplication (SpGEMM) and sparse convolution (SpCONV). Those two are key computation kernels in today's DNN models ranging from convolutional neural networks (CNNs) [27], [57], recurrent neural networks (RNN) [18], [41] to attention-based neural networks [13], [19]. Unfortunately, current ASIC designs only consider SpGEMM kernel [48], [58], [70] or SpCONV kernel [17], [29], [49]. In this work, we aim to accelerate *both* dual-side SpCONV and SpGEMM on Tensor Core.

The greatest challenge of supporting the dual-side SpCONV and SpGEMM on Tensor Core is the unpredictable and randomly distributed non-zero elements inside the input tensors. The dot product unit, a basic computational unit in Tensor Core hardware, conducts a vector-vector inner product. Sparse Tensor Core [45] [42] resolves the irregularity of weight sparsity by applying a structural pruning scheme, which enforces a constant 50% weight sparsity to balance the workload and to exploit parallelism in the dot-product unit. However, this method cannot be applied to SpGEMM because activation sparsity is input-dependent and cannot be pre-determined by pruning methods. Prior ASIC designs take two approaches to leverage dual-side sparsity, as is summarized in Table I. SparTen and Extensor [17], [29] accelerate inner-product by designing dedicated hardware for the inner joint process, which figures out non-zero elements by matching positions in two sparse vectors and accessing those elements. But their

* Contribution during internship at Microsoft Research
** Corresponding Author

method introduces considerable overhead, including complex prefix sum hardware and explicit barrier as the number of non-zeros to be matched is unpredictable. The other type of work (e.g., OuterSPACE [48] and SpArch [70]) accelerate SpGEMM with outer-product, but they do not design for SpCONV, which incurs great performance overhead to transform SpCONV to SpGEMM straightforwardly. Moreover, they are targeted at matrix density 6×10^{-3} to 5×10^{-5}, which become inefficient for mainstream DNN models with the typical density range of 5×10^{-1} to 1×10^{-2}. Likewise, SCNN [49] only considers SpCONV but does not support SpGEMM.

It's also challenging to accelerate SpCONV. GPUs usually transform a CONV operator into a GEMM operator via the im2col method. Sparse Tensor Core [42], [45], [72] only leverages weight sparsity and the input remains dense, which requires only dense im2col. However, to leverage dual-side sparsity, the convolution computation now must consider the *sparse* im2col, which slides over the sparse input tensor and incurs irregular memory accesses. Moreover, because of the space and time overhead of performing an explicit im2col, vendor-supplied DNN acceleration library cuDNN [9] provides optimization of *implicit im2col*. The implicit method fuses the address generation process of im2col into matrix multiplication, and has the best performance in general cases.[1] However, performing the implicit im2col on the sparse input tensors is significantly more challenging than on the dense tensors because of the randomly distributed non-zero elements. In fact, we show that a naive implementation of implicit sparse im2col can be $10\times$ to $100\times$ slower than its dense version.

To tackle the problems above, we analyze the computation patterns of sparse im2col and SpGEMM and compare combinations of different approaches. We argue that bitmap-based encoding format is more friendly for efficient sparse im2col acceleration and outer-product is more efficient to leverage SpGEMM's opportunities on Tensor Core, as is summarized in Table I. We thereby propose a bitmap-based sparse im2col algorithm for SpCONV and an outer-product-based dual-side sparse Tensor Core architecture for SpGEMM. To further co-optimize sparse im2col and SpGEMM, we propose an outer-product-friendly sparse im2col method and a bitmap-based outer-product SpGEMM algorithm. Combining the above techniques, we achieve an efficient implicit sparse im2col design for SpCONV acceleration. Verified on Accel-Sim with V100 architecture, we demonstrate efficient acceleration of both SpCONV and SpGEMM on the proposed dual-side sparse Tensor Core architecture, achieving a speedup of up to one order of magnitude compared with state-of-the-art baselines and imposing negligible hardware overhead.

The key technical contributions of this work are as follows:

- We propose a novel method that combines outer product and bitmap encoding to accelerate SpGEMM (Section III) and SpCONV (Section IV). To the best of our knowledge, our work is the first to study the sparse and implicit im2col method that is critical for SpCONV acceleration on GPUs.
- We show the architectural friendliness of our method to existing GPUs by proposing a small set of modifications, which transform the existing Tensor Core to harness the dual-side sparsity. We also propose novel instruction set extensions that let us leverage the existing high-performance libraries to accelerate SpGEMM and SpCONV (Section V).
- Through extensive evaluations, our dual-side sparse Tensor Core achieves a speedup of up to $7.49\times$ for SpCONV and up to $8.45\times$ for SpGEMM over state-of-the-art methods, with a small 1.5% area overhead. (Section VI)

[1]Although there are other faster convolution methods [35] than implicit im2col, they only prevail on specific matrix shapes, like Winograd for 3×3 convolution kernel. In this work, we only focus on implicit im2col, which outperforms other methods on the majority cases.

II. Background and Related Work

A. Opportunities of sparsity in DNNs

Weight sparsity has been extensively explored in many prior arts, including computer vision and natural language processing tasks [20], [24], [31], [32], [37], [61]–[64]. They demonstrate high sparsity with various pruning methods. However, a significant reduction in weights can only save storage costs, but hardly speed up inference due to the fragmented irregular pattern of the pruning method [20]. Some researchers [20], [39], [60], [61], [67], [72] achieve practical speedup by proposing hardware friendly pruning methods. In NVIDIA's latest Ampere GPU, sparse Tensor Core is first introduced in GPU architecture by adopting the fine-grained structural pruning [7], [42], [45], [66].

Activation sparsity naturally occurs in CNN's feature maps and RNN's hidden layers, followed by ReLU [1] activation functions. Different from weight sparsity that can apply structural pruning by artificial efforts, activation sparsity dynamically changes with input images and is featured with a highly unstructured pattern. Previous works [6], [49], [56] demonstrate that activation sparsity can be as high as 45% to 98%. Some researchers [34], [53] accelerate activation sparsity on GPUs by adding blocked masks, but it requires external knowledge and is not generic. Besides performance acceleration, prior works have also exploited the DNN sparsity to improve their robustness [16], [51]. However, to the best of our knowledge, no previous work has demonstrated meaningful speedup by exploiting activation sparsity on GPU.

B. Computation kernels

Deep neural networks are composed of multiple structurally connected layers of linear and non-linear functions. Among them, matrix multiplication and convolution are the major computation kernels with dominating amount of parameters and computation workloads [12], [68].

Matrix multiplication (GEMM) is the major computation kernel in NLP models, e.g., RNNs [32] and attention-based models [13]. Dense GEMM is one of the fundamental computation primitives provided by GPU, which has been under continuous optimization. Especially, Tensor Core, as the specialized hardware, has recently been deployed in GPU to boost GEMM performance by an order of magnitude.

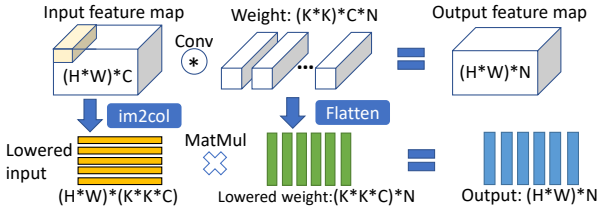


Fig. 1: The im2col-based transformation of CONV to GEMM.

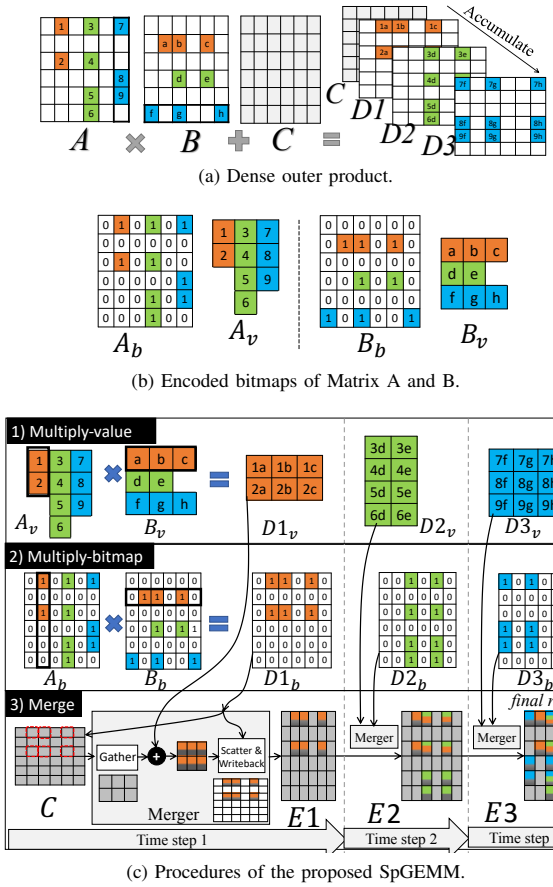
Convolution has played a key role in CNNs [27], [53], which often takes over more than 90% of the overall workload [12]. The convolution takes in a number of C input feature maps, each sized of $H \times W$. Every input feature map is convolved by a sliding kernel sized of $K \times K$ to calculate one pixel in the output feature map. A total of N feature maps will be generated as output to the next layer.

To leverage tensor core's matrix-multiply primitives, state-of-the-art DNN acceleration libraries, e.g., cuDNN, usually transforms convolution into matrix-multiplication by applying im2col function. Figure 1 shows an overview of the im2col function. It re-arranges and expands convolution's input feature maps into a matrix, called lowered feature map, whose each row corresponds to a location of 2-dim sliding window in convolution's input feature maps. Im2col on weight parameters is simply flattening each $K \times K \times C$ kernel. Since im2col's data expansion is mainly applied on input feature maps, the weight-sparse architecture treats im2col as *dense im2col* while the dual-side sparse architecture has to deal with *sparse im2col*, which has not been fully discussed in previous work.

The naïve approach, called *explicit im2col*, conducts im2col and GEMM separately. One concern about *explicit im2col* is that the lowered feature map usually takes $K \times K$ times more global memory than the original feature map because the overlapped sliding windows generate duplicated data. To improve the input data reuse, the state-of-the-art DNN library (e.g., cuDNN) uses *implicit im2col*, which keeps original feature map layout in global memory and uses an address conversion scheme to do im2col transformation in on-chip caches, instead of physically duplicating data in global memory. Implicit im2col has been widely used as the state-of-the-art method in accelerating convolution with GEMM operators.

C. Design philosophy and challenges

In this paper, our design target is to transform the existing Tensor Core with minimal modification to exploit both weight and activation sparsity for better performance. The alternative design is to directly port an existing ASIC as the GPU's co-processor. However, this design philosophy leads to a significant area cost. E.g., SpArch [70] uses an array with 16 floating point multipliers to compute the outer-product for SpGEMM, and requires specialized Merge Tree and matrix read/write hardware, which occupies the 98.4% die area ($\approx 28mm^2$ @ 40nm). If we were to scale the design to 40960 FP multipliers in V100 [47], the estimated area cost would be prohibitively expensive (i.e., 71680 mm^2). Instead, our design tries to reuse



(a) Dense outer product.

(b) Encoded bitmaps of Matrix A and B.

(c) Procedures of the proposed SpGEMM.

Fig. 2: Our proposed bitmap-based outer-product SpGEMM.

the GPU's hardware resources such as memory hierarchies and data path, which leads to a small overhead ($\approx 12.8mm^2$, or 1.5% as we show later).

The challenges for our design philosophy lie in the aspect that we need to support both the SpGEMM and SpCONV. The prior SpGEMM designs such as OuterSPACE [48] and SpArch [70] target at matrix density 6×10^{-3} to 5×10^{-5}, which become inefficient for mainstream DNN models with the typical density range of 5×10^{-1} to 1×10^{-2}. Moreover, those designs do not support SpCONV, where the aforementioned im2col approach imposes a unique challenge as it needs to handle the sparse and unpredictable non-zero elements in the input tensor. As such, none of the prior designs serve our purpose, and we need new architectural innovations.

III. Bitmap-based SpGEMM

We propose an outer-product-based algorithm to accelerate SpGEMM using the bitmap-based sparse encoding format.

A. Overview

To exploit the dual-side sparsity, we propose an efficient SpGEMM algorithm based on outer-product matrix multiplication. A basic step in outer-product-based matrix multiplication

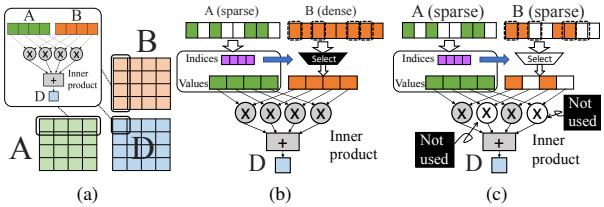


Fig. 3: (a) 4×4×4 matrix multiplication primitive used in inner-product-based Tensor Core [47]; (b) A sparse inner-product unit in A100 [45] [42]; (c) Dual-side sparsity unit based on inner-product.

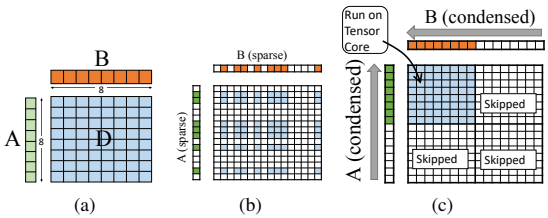


Fig. 4: (a) 8×8×1 matrix multiplication primitive in outer-product-based Tensor Core ; (b) Outer-product with two sparse inputs; (c) Leveraging dual-side sparsity with outer-product.

is to compute a cross product between a column of $\mathbb{A}$ (sized of $M \times 1$) and a row of $\mathbb{B}$ (sized of $1 \times N$), which leads to the output $M \times N$ partial matrix (e.g., $\mathbb{D}1$, $\mathbb{D}2$, $\mathbb{D}3$ in Figure 2a). To generate the final output, we need to accumulate all those partial results and bias matrix $\mathbb{C}$ with multiple rounds.

Our approach achieves an efficient outer product by using bitmap representation, as shown in Figure 2b. Each input matrix is represented by a two-tuple encoding of a bitmap (e.g., $\mathbb{A}_b$ and $\mathbb{B}_b$), and a collection of non-zero values (e.g., $\mathbb{A}_v$ and $\mathbb{B}_v$). The bitmap uses 1's for positions of non-zero values and 0's for zeros. To support outer-product, matrix $\mathbb{A}_v$ is encoded in column-major and $\mathbb{B}_v$ is encoded in row-major.

The proposed SpGEMM algorithm has three major operations on the bitmap encoded matrices, which are *multiply-value*, *multiply-bitmap* and *merge* respectively. In Figure 2c, the *multiply-value* operation computes the cross-product on each vector-vector pair of $\mathbb{A}_v$ and $\mathbb{B}_v$ to generate values of the partial matrices (e.g., $\mathbb{D}1_v$, $\mathbb{D}2_v$, and $\mathbb{D}3_v$). Since the outer-product avoids the explicit inner-join process, its multiplication is regular and easy to accelerate. The *multiply-bitmap* operation computes 1-bit cross product on the bitmaps $\mathbb{A}_b$ and $\mathbb{B}_b$. The output bitmap contains the sparsity information of the corresponding partial matrix, such as $\mathbb{D}1_b$ to $\mathbb{D}3_b$. At last, the *merge* operation uses values (e.g., $\mathbb{D}1_v$) and bitmaps (e.g., $\mathbb{D}1_b$) of the partial matrices from the previous two operations to do the accumulation with multiple rounds, such as from $\mathbb{E}1$ to $\mathbb{E}3$. Despite the benefits from the regular multiplication provided by the outer product, the partial matrix is sparse and irregular. However, it is worthwhile to make this trade-off because dealing with a single-side irregular accumulation is much cheaper than dual-side sparse multiplication. We propose a gather-scatter method to merge the non-zero values from different partial matrices with multiple rounds.

In the following section, we present a detailed analysis on the problems of inner-product-based Tensor Core with accelerating SpGEMM and the advantages of outer-product-based Tensor Core. Then we propose a SpGEMM algorithm for the outer-product Tensor Cores in a warp. At last, we extend to the whole device.

B. SpGEMM in a Warp

1) Problems of Inner-product Tensor Core: In V100 architecture [47], a warp controls two tensor cores simultaneously. Each tensor core can complete a $4 \times 4 \times 4$ dense matrix multiplication per cycle in a 4-stage pipeline [52]. The basic computation unit in a dense tensor core is a parallel 4-element vector-vector dot product unit that multiplies and accumulates A matrix row and B matrix column, as Figure 3a shows. For single-side sparse matrix multiplication, the dot product requires selecting and accessing non-zero elements in matching positions in the dense vector. To solve the irregular addressing introduced in sparse models, sparse tensor core [45] [42] uses a structural pruning scheme that conducts a 2-out-of-4 pruning in each partitioned sub-vector, which enforces a constant 50% weight sparsity to balance the workload and to exploit parallelism in the dot-product unit, as shown in Figure 3b. However, this method is *inefficient* for dual-side sparse matrix multiplication because activation sparsity is input-dependent and the number of non-zeros to be jointly matched is unpredictable, which results in difficulties to fulfill the dot product parallelism, as shown in Figure 3c. Although some prior ASIC designs [17], [29], [49] propose dedicated hardware to solve this problem, their method either uses complex prefix sum hardware, costly shuffling register, or explicit barrier, which introduces considerable overheads and significant modifications to the Tensor Cores.

2) Outer-product Tensor Core (OTC): Our design adopts an outer-product-based tensor core shown in Figure 4a. We use an $8 \times 8 \times 1$ tile size as it has the same number of multipliers and adders (i.e., 64 in FP16) as the inner-product tensor core.

OTC naturally avoids the inner-join process and can eliminate the irregular addressing by condensing two sparse inputs into two dense ones. As shown in Figure 4c, outer-product-based solutions can push all non-zeros in each column of matrix $\mathbb{A}$ to the upper-side and all non-zeros in each row of matrix $\mathbb{B}$ to the left, forming two dense vectors. Outer-product multiplication on the condensed inputs yields a condensed matrix multiplication. After condensing, non-zero elements are concentrated so that tensor cores take fewer instructions to complete a matrix multiplication and thus can achieve speedup over the original dense one.

3) Warp-level Outer-product SpGEMM: We propose an efficient warp-level SpGEMM algorithm with OTCs. Recall that there are two Tensor Cores working concurrently in a warp, each performing a $4 \times 4 \times 4$ matrix multiplication. In this section, we assume that the OTCs maintain an equivalent

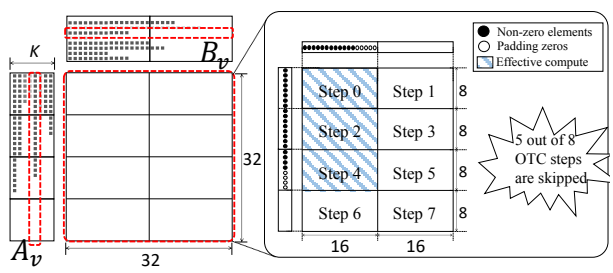


Fig. 5: SpGEMM in a warp.

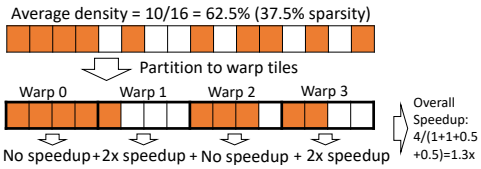


Fig. 6: Speedup on global matrix.

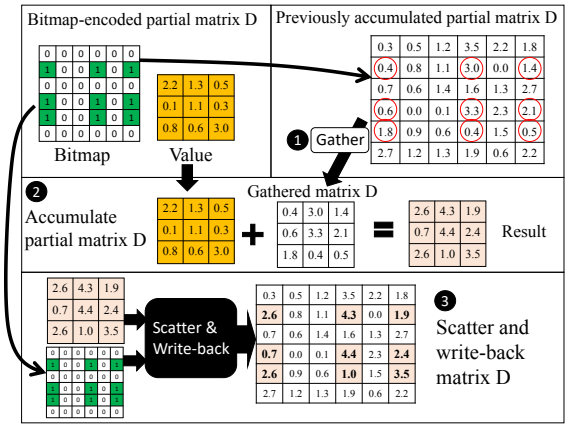


Fig. 7: Gather-Scatter accumulation.

computing power, with two tensor cores together performing $8 \times 16 \times 1$ outer product. Section V will provide more architectural details.

Figure 5 shows an example where our warp-level SpGEMM achieves speedup on OTC with sparse inputs. It shows a $32 \times 32 \times K$ warp tile computed in the outer-product manner. Since OTC computes an $8 \times 16 \times 1$ tile in a cycle, it takes 8 steps to complete the $32 \times 32 \times 1$ outer-product. For sparse inputs, $\mathbb{A}_v$ and $\mathbb{B}_v$ have fewer non-zeros elements in each column and row. Thus we can achieve speedup by skipping OTC steps. Figure 5's right-hand shows an example, where the column vector from $\mathbb{A}_v$ has 20 non-zeros in 32 elements, and row vector from $\mathbb{B}_v$ has 11 non-zeros in 32 elements. As such, 5 out of 8 OTC steps have pure zero elements and can be skipped, leading to a $\frac{8}{3} = 2.67\times$ speedup in theory. The number of skipped OTC steps depends on the sparsity of the input vectors, which are $\langle 0\%, 25\%, 50\%, 75\% \rangle$ on the $\mathbb{A}_v$ side and $\langle 0\%, 50\% \rangle$ on the $\mathbb{B}_v$ side. Zeros are padded to the inputs to fulfill OTC's 8×16 tile dimension.

Discussion: Although the acceleration opportunity within a warp relies on an enumerable number of fixed sparsity ratios (e.g., $\langle 0\%, 50\% \rangle$ for $\mathbb{B}_v$), our method at the global matrix level can go beyond this limitation. Figure 6 shows an example where a row of the global matrix has a 37.5% sparsity, which should have no speedup on the assumption that we can only benefit from $\langle 0\%, 50\% \rangle$ sparsity. However, we can still achieve an approximately $1.3\times$ speedup after considering warp tiling at the global matrix level. Because the non-zeros are usually not evenly distributed across the global matrix, some warps such as warp 1 and 3 in Figure 6 can still enjoy the speedup provided by our SpGEMM.

One design challenge in our warp-level SpGEMM algorithm is that the outer product requires all $M \times N$ elements of $\mathbb{D}$ to be stored in Tensor Core's local buffer so that it can be accumulated immediately. Therefore, the warp-tile size is majorly constrained by the Tensor Core's local buffer size. Hardware design and evaluation are described in Section V and Section VI, respectively.

4) Merge: The *merge* operation accumulates the partial matrices in multiple steps, as in Figure 2c. In each step, we merge the newly generated partial matrix in bitmap encoding (e.g., $\mathbb{D}2_v$ and $\mathbb{D}2_b$) with the previously accumulated results (e.g., $\mathbb{E}1$). The pre-computed bitmap matrix (e.g., $\mathbb{D}1_b$) lets us easily derive the positions of non-zeros to be accumulated.

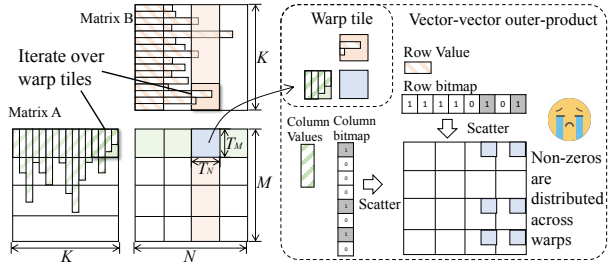
Fig. 7 shows three steps in our *merge* operation. First, we use the bitmap to ❶*gather* corresponding elements from the previous accumulated matrix. Then, the gathered elements are ❷*accumulated* to the values from *multiply-value* output. We finally use ❸*scatter* function to restore non-zeros' positions by matching the 1's in bitmap and write to the result matrix.

To map *merge* on the OTC, we integrate the gather-accumulate-scatter procedure into Tensor Core's output matrix buffer with two optimizations. On one side, parallel accumulations are required to match OTC's multiply throughput. We design an efficient multiply-accumulate pipeline with 128-way parallel accumulators. On the other side, we design a light-weight operand collector to deal with irregular memory access in the gather-scatter function. Hardware design and evaluation are discussed in Section V and Section VI.

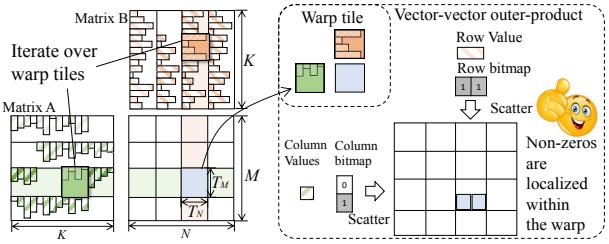
C. SpGEMM on the device

The biggest challenge to map the SpGEMM on the entire device is that the outer product has poor output data reuse. When running a large matrix multiplication, outer products will produce a large amount of data in partial matrices. For SpGEMM, randomly distributed non-zero elements (e.g., $\mathbb{D}1$) in the partial matrix will yield a large addressing space and may often exceed a warp's local buffer size. This long-range addressing will result in frequent and fragmented global memory access, as the example shows in Figure 8a.

To address the problem, we introduce a hierarchical bitmap encoding format that is aware of GPU's tiling scheme on SpGEMM. As Figure 8b shows, we first partition SpGEMM



(a) Tiled SpGEMM with one-level bitmap encoding.



(b) Tiled SpGEMM with two level bitmap encoding.

Fig. 8: Data-locality aware sparse representation.

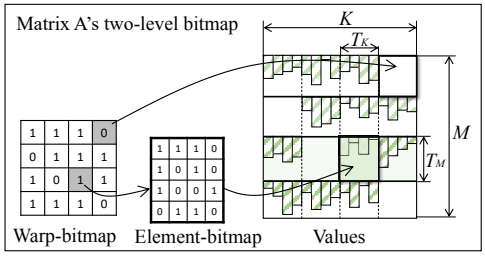


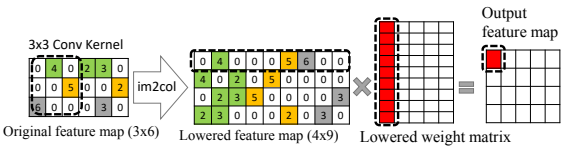
Fig. 9: Two-level bitmap encoding format.

into thread blocks, each of which computes an output block by iteratively loading blocks of A and B from input matrices. Each thread block computes a $32 \times 32 \times 16$ matrix multiplication by running a warp-level SpGEMM aforementioned.

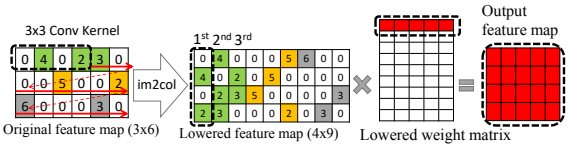
To achieve this optimization, we propose a two-level bitmap encoding format. It contains three tuples, as shown in Figure 9. The first level bitmap encodes each partitioned matrix tile. Each '1' or '0' in this bitmap represents elemental zeros or non-zeros in the warp tile, and thus it is called element-bitmap. Since the sparse inputs' non-zero elements are located within the tile, the positions of output partial matrix non-zeros are also located within this tile, and thus can be fitted in Tensor Core's fastest local buffer and avoid external memory access. The second level bitmap is called warp-bitmap, which uses a '1' or '0' to represent the entire tile, where '0's means the tile is empty and '1's is not. The warp with a '0' warp-bit can be skipped because either of the two input tiles is pure zeros.

IV. Dual-side Sparse Convolution

GPU usually accelerates dense convolution by transforming it into a GEMM operator through im2col function. To leverage the proposed SpGEMM algorithm, we propose a novel sparse im2col method for dual-side sparse convolution.



(a) Inner product friendly im2col.



(b) Outer product friendly im2col.

Fig. 10: Outer product friendly im2col on dense matrix.

Im2col mainly rearranges the data organization of input feature maps as an input of GEMM. Thus, improperly designed im2col may harm input data re-use for matrix multiplication. We propose an outer-product friendly im2col method for generic outer-product. To avoid the space and time overhead by doing im2col explicitly, we present our *implicit* sparse im2col algorithm that supports efficient data rearranging in registers.

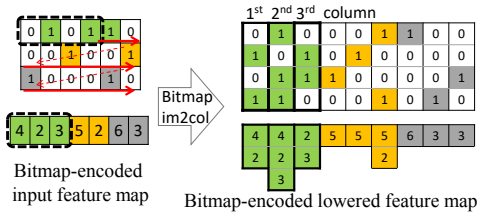
A. Outer-product friendly im2col

Figure 10a shows an example of im2col on a 3×6 feature map with a 3×3 convolution kernel. It rearranges all elements in the 3×3 sliding window into a row in the lowered feature map. And the output matrix of im2col is generated by shifting the sliding window by one element per step with multiple rounds. This process is friendly for inner-products because one row in each step matches inner-product's multiply-and-accumulated computation. On the contrary, outer-product requires a column of data in each step, which im2col cannot utilize the fully lowered feature map.

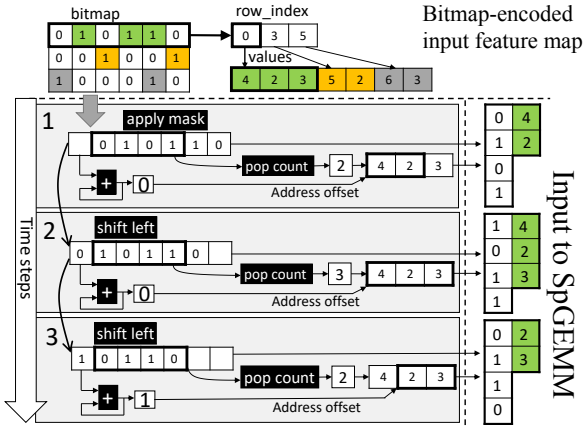
In contrast, the proposed outer-product-friendly im2col generates a column of data in the lowered feature map at a time in Figure 10b. For example, the first three columns come from the first row in the feature map. And these three columns share four data from each other. If sliding a 1×4 window scanning over the feature map in a zig-zag way, we will get a column-major lowered feature map as the input of GEMM. In generic cases, the number of values in a column is decided by feature map and convolution kernel size, which is calculated by $B = (R - K + S)/S$, where $\langle R, K, S \rangle$ stands for row size of feature map, convolution kernel size, and stride size, respectively. This transformation is equivalent to permuting the loop nest of accessing the lowered feature map by structuring the inner-most loop as the outer-most.

B. Bitmap-based sparse im2col

Similar to dense *implicit im2col*, our *sparse implicit im2col* keeps the bitmap-encoded sparse feature maps in global memory and re-arranges data layout in registers. Matrix B is simply bitmap-encoding of the flattened sparse weight matrix.



(a) Bitmap im2col's input and output.



(b) The proposed sparse im2col on bitmap. This bitmap format comprises three fields. The bitmap field uses 0/1 to represent zero and non-zero positions. The value field contains non-zeros. The row offset is specifically introduced for the convenience of sparse im2col.

Fig. 11: The im2col on a bitmap-encoded feature map.

Bitmap is efficient for sparse im2col because it inherits the structural information from a dense matrix. So bitmap-based encoding format can first conduct im2col on bitmap using a method similar to dense im2col and generate a lowered bitmap. Then, we use the bitmap as a mask to fetch the corresponding non-zero values. Figure 11 shows a detailed flow of our approach with an example. We use a 3×6 feature map convoluted by a 3×3 kernel, and get a 4×9 lowered feature map in Figure 11a. Due to the outer-product-friendly im2col method, the first three columns of data, which we highlight, are generated sequentially in this example.

S0 We first encode the original feature map in bitmap format.
S1 We take the first bitmap row from bitmap encoding and its corresponding non-zero values.
S2 For the first column, we *apply a mask* on the bitmap row and output the bits falling in the mask as the first column bitmap for the lowered feature map. For the subsequent columns, we *shift left* the bitmap row, which leads to shifted-out bit.
S3 We *accumulate* the shifted-out bit and use the accumulated result as the address offset to access the non-zero values.
S4 We use *population count* to count the number of non-zeros in the mask. We find corresponding non-zero data in the value vector and output its value, address offset and length.

Our approach is efficient for two reasons. First, all required operations are low cost and can be conducted in register files. Second, the result is already in the condensed format and can be directly fed to outer-product SpGEMM via register reads.

V. Outer Product Sparse Tensor Core

In this section, we introduce the micro-architecture extensions to support our bitmap-based SpGEMM and SpCONV.

A. Outer-product Tensor Core (OTC)

We modify Tensor Core hardware from inner-product to outer-product for *dense matrix multiplication* because it is a pre-requisite for our SpGEMM algorithm.

1) Hardware modification: Tensor Core, a specialized matrix multiplication hardware, is integrated into NVIDIA's GPGPU since Volta architecture [47] to accelerate machine learning tasks. Figure 12a shows an overview of a Sub-Core [11] in a Volta's streaming processor (SM). Each Sub-Core contains a bunch of math function units and two Tensor Cores. Each Tensor Core completes a $4 \times 4 \times 4$ dense matrix multiplication in a cycle in a 4-stage pipeline [52]. In the V100 GPU, a total number of 640 Tensor Cores are distributed across 80 SMs, with 2 Tensor Cores per Sub-Core and 4 Sub-Cores per SM, providing a peak performance of 125 TFLOPS at 1530 MHz clock frequency.

Figure 12b shows a detailed architecture of the two Tensor Cores in a Sub-Core. Each Tensor Core contains 16 inner-product units. Each inner product unit performs a four-element dot-product (FEDP), yielding a total computing power of 64 multiply-accumulate per cycle in a single Tensor Core. Figure 12c details a FEDP structure, which multiplies and accumulates two four-element vectors from A and B in parallel. The 16 FEDPs are grouped into two 'Octets,' eight to each Octet. One Octet is further split into two thread groups. Each thread group contains four FEDPs and computes four elements in the 4×4 output matrix, as is shown in Figure 12e.

We modify the above-mentioned inner-product Tensor Core to fit for *dense outer-product*'s computation. Figure 12d depicts our changes to the FEDP hardware. Our four-element outer product (FEOP) multiplies one element from A with four elements from B in parallel and accumulates partial results with the adders. As such, four FEOPs in a thread group collectively perform a 4×4 outer-product, as in Figure 12f.

2) ISA extensions for dense outer product: Recall that each tensor core performs a $4 \times 4 \times 4$ dense matrix multiplication. Two tensor cores work cooperatively and form a machine-level HMMA.884 instruction to compute an $8 \times 8 \times 4$ output block by taking an 8×4 tile of A and a 4×8 tile of B, as shown in Figure 13a. Four sets of HMMA instructions are used to compute an $8 \times 8 \times 16$ tile. At the warp level, CUDA exposes a WMMA API that computes a larger $16 \times 16 \times 16$ matrix operation with these HMMA instructions in 32 cycles.

Fig. 13b depicts our outer-product tensor core (OTC) interface. Each OTC conducts an $8 \times 8 \times 1$ dense vector-vector outer product, which has the same 64 FP16 multipliers as the original $4 \times 4 \times 4$ Tensor Core. Two OTCs form a $8 \times 16 \times 1$ Outer-product HMMA (OHMMA.8161) instruction that takes a 8×1 tile of $\mathbb{A}_v$ and a 1×16 tile of $\mathbb{B}_v$ as input. We define

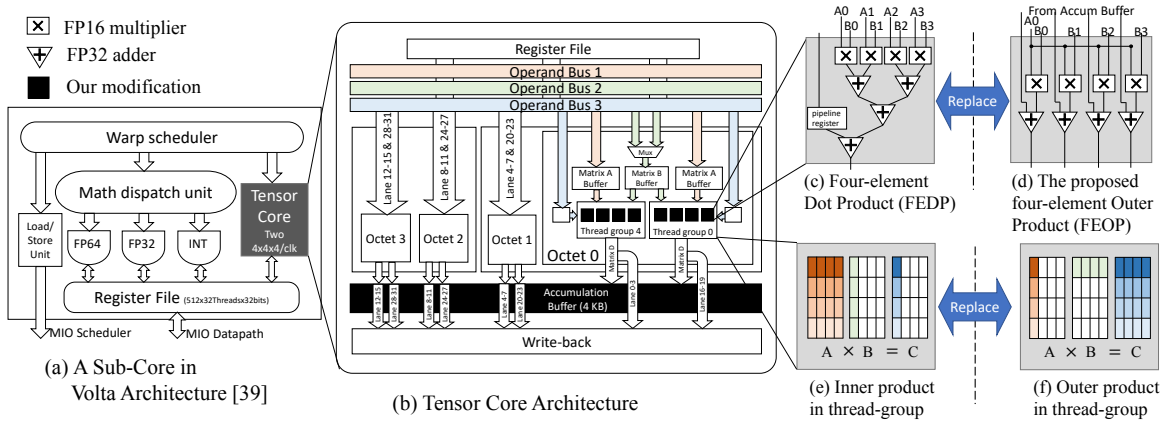


Fig. 12: Our modification to the Tensor Core includes the accumulation buffer in (b) and the replacement of Four-Element Dot Product Unit (c) with Four-Element Outer Product (d). Since there are four FEDPs in each thread group, the thread-group level inner-product (e) is replaced with outer-product (f).

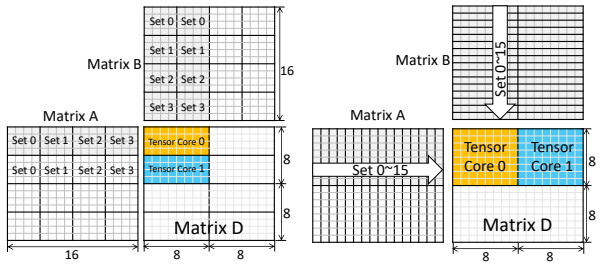


Fig. 13: The original WMMA operation computes a 16×16×16 matrix multiplication in a warp tile. We define OWMMA (outer-product WMMA) operation that computes the dense warp-tile with 16 sets of OHMMA.8161 instructions.

16 sets of OHMMA instructions to complete the full-warp $16 \times 16 \times 16$ matrix multiplication (OWMMA). In total, an OTC also takes 32 cycles to finish an OWMMA operation.

The bitmap outer-product is also an essential step in our bitmap-based SpGEMM. To accelerate bitmap operation, we execute *multiply-bitmap* on the OTC in Tensor Core. We define Binary OHMMA instruction (BOHMMA), which conducts outer product on 1-bit inputs. Since Volta architecture [11], Tensor Core has already started to support binary operations that process 16× larger matrix tile than the FP16 operations [10]. We inherit the binary operator design from the native Tensor Core and extend the binary OHMMA (BOHMMA) instruction to support $32 \times 32 \times 1$ binary outer product. The *multiply-bitmap* achieves low cost because BOHMMA is 16× faster than HMMA on FP16 outer product. OHMMA and BOHMMA instructions are defined in Figure 14.

```
HMMA.OHMMA.8161.F32.F32 {R9, R10, R11, R12},
                        {R1, R2}, {R3, R4}, {R5, R6, R7, R8};
HMMA.BOHMMA.32321.B32.B32 R3, R1, R2;
```

Fig. 14: Extended machine-level OHMMA/BOHMMA instructions.

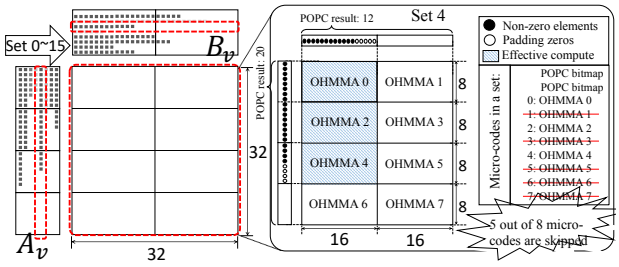


Fig. 15: The proposed SpWMMA includes 8 OHMMA instructions in dense mode, which can be skipped during the sparse mode.

B. Dual-side Sparse Tensor Core

We propose two adaptations to achieve speedup on dual-side sparsity with the above hardware and instruction extensions support. On the software side, we define SpWMMA, warp-level dual-side sparse matrix multiplication API that exploits sparsity in matrix A and B by dynamically skipping OHMMA instructions. On the hardware side, we propose the accumulation buffer to gather partial results from outer-product units.

1) Warp-level interface: We define a SpWMMA API that works on a warp-level matrix tile in Figure 16. A SpWMMA breaks down to 16 sets, and each set includes a $32 \times 32 \times 1$ outer product in Figure 13. Since the machine-level OHMMA instruction computes an $8 \times 16 \times 1$ outer product within a warp. And each SpWMMA API call is complied to 8 OHMMA instructions, as shown in Figure 17.

For sparse inputs, $\mathbb{A}_v$ and $\mathbb{B}_v$ have fewer non-zeros elements and thus achieve speedup by skipping OHMMA instructions with predication operations. Predication operation is widely used in GPGPU to skip instruction executions. We utilize population count instructions (POPC, commonly supported in GPGPU to count the number of "1" in binary numbers) to set predication bits of OHMMA instructions. The number of "1" in $\mathbb{A}_v$'s and $\mathbb{B}_v$'s bitmaps identify the number of element-wise multiplication in each row/column of the condensed sparse

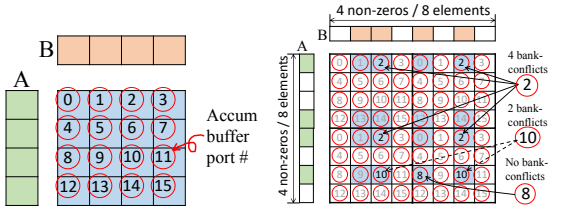
matrix multiplication. By counting "1" bits in the bitmap with POPC, we can determine which OHMMA instructions should be enabled. The right side of Figure 15 shows an example of Set 4's computation with POPC and OHMMA instructions. We count $\mathbb{A}_v$'s and $\mathbb{B}_v$'s bitmaps, indicating that the sparse multiplication takes 12/20 multiplications in each row/column. In our design, each OHHMA instruction covers 8×16 condensed sparse outer product multiplication. We should enable OHHMA0/2/4 by setting predication bits and skip OHHMA1/3/4/5/6/7 for Set 4.

```
SPWMMA.MMA.SYNC.A_LAYOUT.B_LAYOUT.M32N32K1.set.f32.f32
{%RD0~%RD7}, {%RB0~%RB7}, {%RA0~%RA7}, {%RC0~%RC7};
```

Fig. 16: Our SpWMMA API.

```
HMMA.BOHMMA.32321.B32.B32 R3, R1, R2;
// ...
@p0 HMMA.OHMMA.8161.F32.F32 {R8, R9, R10, R11},
    {R4, R5}, {R6, R7}, {R8, R9, R10, R11};
@p1 HMMA.OHMMA.8161.F32.F32 {R16, R17, R18, R19},
    {R12, R13}, {R14, R15}, {R16, R17, R18, R19};
// ...
@p7 HMMA.OHMMA.8161.F32.F32 {R119, R120, R121, R122},
    {R115, R116}, {R117, R118},{R119, R120, R121, R122};
```

Fig. 17: SpWMMA API complied to machine-level instructions.



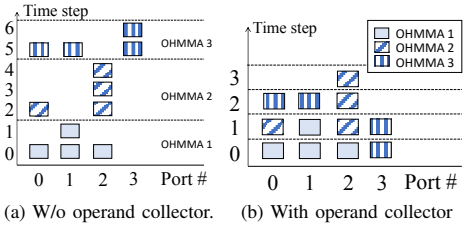
(a) Dense mode. (b) Sparse mode.

Fig. 18: Memory access pattern in the accumulation buffer.

2) Accumulation buffer: The accumulation buffer has two modes, a *dense* mode, and a *sparse* mode. In dense mode, the accumulation buffer configures each read/write port directly connected to each output from FEOP units. In sparse mode, a large amount of partial matrix is generated (e.g., 32×32 FP32 for the warp-tile in SpWMMA). We extend the accumulation buffer to a multi-bank memory of 4 KByte ($32\times32\times4$ Bytes). Furthermore, the gather-accumulate-scatter method, discussed in Section III-B4, requires random access to multiple banks. We design an operand collector to schedule bank reads and writes to optimize the effective bandwidth.

a) Dense mode: Figure 18a shows an example of the FEOPs' outputs memory access to accumulation buffer port. For simplicity, we use a 4×4 example. Since one OHMMA instruction is issued per cycle, the accumulation buffer uses 16 ports (e.g., the numbers in circles) for each FEOP output (e.g., elements in the blue matrix).

b) Sparse mode: Figure 18b shows an example of accumulation buffer port memory access pattern when FEOP is running the *sparse* matrix multiplication (e.g., SpWMMA in Figure 15). We assume an $8\times8\times1$ warp tile with both input vectors being 50% sparse. Under this setting, OHMMA still



(a) W/o operand collector. (b) With operand collector

Fig. 19: Memory access schedule optimization via operand collector.

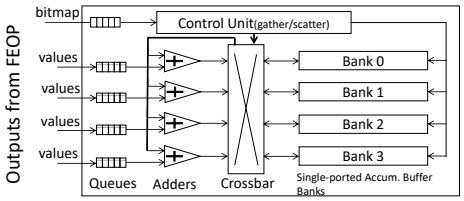


Fig. 20: Accumulation buffer design.

generates 16 outputs per cycle. But the accumulation of these outputs may cause many bank conflicts as they are randomly distributed across the partial matrix, as shown in Figure 18b.

To solve this issue, we propose to integrate a small operand collector in our accumulation buffer for better memory bandwidth utilization. Operand collector [38] is a technique used in NVIDIA's GPU micro-architecture to overlap the reading on the source operands from register file banks among multiple instructions. Figure 19 shows an example of our accumulation buffer's memory access schedules with four operands per cycle on four ports with(out) operand collectors. The operand collector can significantly increase memory throughput by combining non-conflict memory accesses from different instructions.

Figure 20 shows an overall design of our accumulation buffer integrated with the aforementioned operand collector. It can support both dense and sparse outer product. The sparse mode is automatically turned on with the SpWMMA API.

VI. Evaluation

We conduct comprehensive experiments with various micro benchmarks and DNN models to evaluate our software and hardware design, focusing on 1) how effective the bitmap-based im2col can reduce the decoding overhead; 2) how much performance our SpGEMM can improve upon a variety of sparsity ratios; 3) how much we can speed up the inference of diverse neural-network layers; and 4) how much hardware overhead is introduced in terms of hardware area. Evaluation results show that our design achieves significant performance improvements of up to one order of magnitude compared to the baselines and imposes small hardware overhead.

A. Experimental Setup and Methodology

Simulation Platform We use Accel-Sim [33], a cycle-accurate simulator based on GPGPU-Sim [36], to evaluate our design. The simulator provides flexible front-end architecture, optimizes cache and shared memory models, and improves

TABLE II: Details of our evaluated sparse DNN model.

Models	Pruning Scheme	Dataset	Accuracy
VGG-16	AGP [73]	ImageNet	88.86% (top 5)
ResNet-18		ImageNet	86.46% (top 5)
Mask R-CNN		COCO	35.2 (AP)
BERT-base encoder	MP [30] [54]	SQuAD	83.3 (F1 score)
RNN	AGP	WikiText-2	85.7 (ppl)

simulation accuracy significantly compared with previous generations. We model a Tesla V100 GPU [47] on the simulator. To support SpWMMA instructions, we implement a cycle-accurate tensor core model based on our hardware design in Section V. In addition, we extend the simulator front-end to support our instruction extensions, as shown in Figure 15.

Baselines We choose CUTLASS [46] and cuDNN [9] as dense GEMM and convolution baselines, respectively. CUTLASS is an open-sourced GEMM library that achieves high performance comparable with cuBLAS [44]. cuDNN [9] is a vendor-optimized, widely used library for DNN acceleration. For our SpGEMM and SpCONV algorithms, we compare with two baselines: the vendor-optimized sparse matrix library cuSparse [43], and the state-of-the-art research work of Sparse Tensor Core [72]. For fair comparisons, our SpGEMM and SpCONV implementations build on the same loop tiling and software computation pipeline as CUTLASS [46].

DNN Models and Pruning We evaluate our algorithms using various types of DNN models, including 1) three widely-used CNN models: VGG-16 [57], ResNet-18 [27], and Mask R-CNN [26]; 2) one RNN model for word-level language modeling with a 2-layer LSTM encoder and a 4-layer LSTM decoder that was also used in Sparse Tensor Core [72] and we use the same configuration; and 3) BERT-base [13] encoder, a representative and well-known attention-based model.

We fine-tune and prune the CNN models with Automated Gradual Pruner (AGP) [73] on Distiller [74]. We use the fine-pruned BERT-base encoder model [54] [30] on the SQuAD task. We also fine-tune and prune the RNN model with AGP on Wikitext-2 [5] dataset. Unlike CNN models, BERT encoder and RNN models usually have high sparsity on only weights but not feature maps. Note that our work does not affect the model accuracy because we do not propose any new pruning algorithm. Table II summarizes the sparse model accuracy, which is consistent with previous pruning works. The detailed layer-wise activation and weight sparsity ratios are listed in Figure 22.

B. Performance of Bitmap-based Im2col

We first evaluate the performance of our bitmap-based im2col, compared with dense im2col and CSR-encoded im2col. We compare against the CSR [55] as it is one of the most widely-used sparse matrix encoding methods. We implement these three im2col algorithms based on PyTorch ATEN library [50] and use a typical convolution layer from ResNet-18 to make the comparison. We measure the execution time of these algorithms and normalize the results over the dense im2col case. We tune different feature-map sparsity of 0% - 99.9% and show the results in Table III.

TABLE III: Normalized im2col time comparison using a typical convolution layer from ResNet-18 (feature map H/W=56, filter H/W=3, in/out channel=128) under different sparsity ratios.

Sparsity (%)	0	25	50	75	99	99.9
Dense Im2col	1	1	1	1	1	1
CSR Im2col	101.3	67.1	45.2	14.5	4.7	1.2
Bitmap Im2col	8.31	6.87	4.73	2.5	1.5	1.1

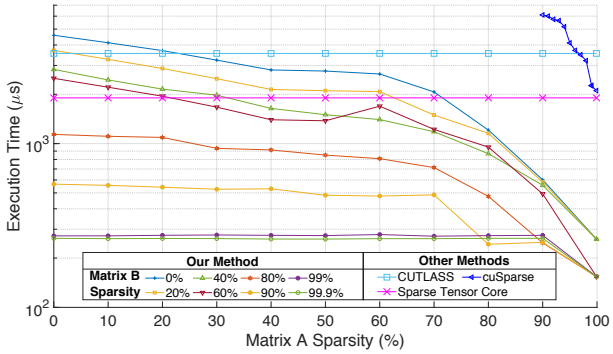


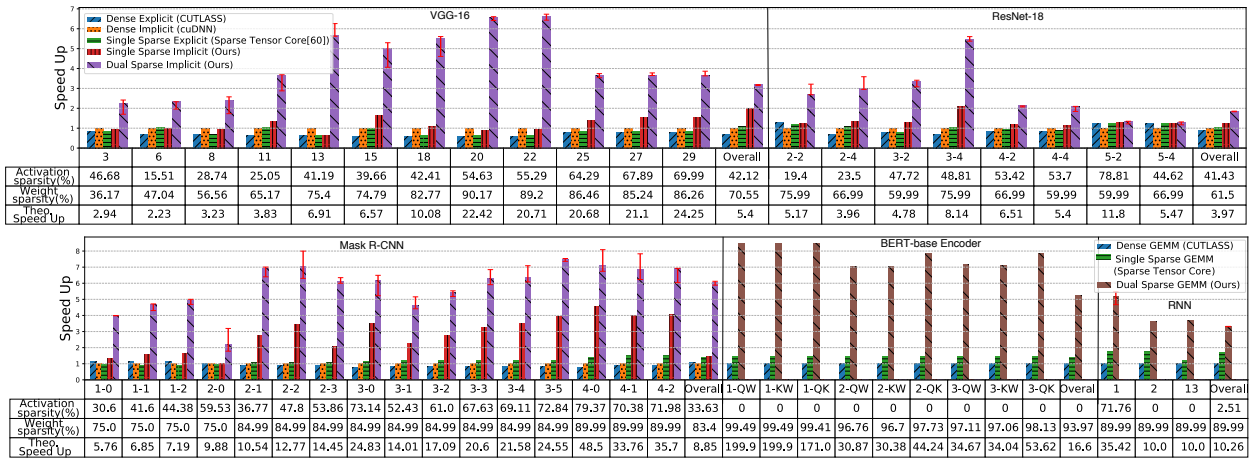
Fig. 21: Performance comparison of SpGEMM on the CUTLASS baseline. cuSparse only outperforms the baseline when the sparsity is large(>95%). CSR-based Sparse Tensor Core cannot fully exploit dual-side sparsity. Our SpGEMM achieves a much higher speedup and also supports a very wide range of sparsity of matrices A and B.

The results reveal that our bitmap-based im2col significantly outperforms CSR-encoded im2col across different sparsity ratios and is one order of magnitude faster when the sparsity ratio is less than 50%. Only when the sparsity ratio is extremely high, e.g., 99.9%, CSR-encoded im2col achieves a comparable (but still lower) performance with our bitmap-based im2col. This big discrepancy is attributed to the fact that CSR encoding introduces two additional data-dependent memory reads for each non-zero data access, while bitmap encoding compresses non-zero data offsets into bits that significantly reduce the operational intensity in im2col.

C. Performance of SpGEMM

We then evaluate the performance of our SpGEMM, compared with CUTLASS, cuSparse, and Sparse Tensor Core [72]. Among them, CUTLASS is the baseline of dense matrix multiplication. We measure the execution time of the multiplication of matrix A and matrix B (both are 4096×4096) with various sparsity ratios. For cuSparse, we fix the sparsity of matrix B to 99% and vary the sparsity of matrix A from only 90% to 99.9% because it is too slow when the sparsity of matrix A is less than 90%. Figure 21 shows the results, and we make the following observations.

First, cuSparse is not applicable for accelerating sparse neural networks. Although matrix B already has a high sparsity of 99%, cuSparse becomes faster than CUTLASS (i.e., the dense case) only when the sparsity of matrix A is higher than 95%. Even when the sparsity of matrix A is as high as 99.9%, it achieves a speedup of only 1.67×. When the sparsity of matrix A is 90%, it is 1.75 times slower than CUTLASS.



	3	6	8	11	13	15	18	20	22	25	27	29	Overall	2-2	2-4	3-2	3-4	4-2	4-4	5-2	5-4	Overall
Activation sparsity(%)	46.68	15.51	28.74	25.05	41.19	39.66	42.41	54.63	55.29	64.29	67.89	69.99	42.12	19.4	23.5	47.72	48.81	53.42	53.7	78.81	44.62	41.43
Weight sparsity(%)	36.17	47.04	56.56	65.17	75.4	74.79	82.77	90.17	89.2	86.46	85.24	86.26	70.55	75.99	66.99	59.99	75.99	66.99	59.99	59.99	66.99	61.5
Theo. Speed Up	2.94	2.23	3.23	3.83	6.91	6.57	10.08	22.42	20.71	20.68	21.1	24.25	5.4	5.17	3.96	4.78	8.14	6.51	5.4	11.8	5.47	3.97

	1-0	1-1	1-2	2-0	2-1	2-2	2-3	3-0	3-1	3-2	3-3	3-4	3-5	4-0	4-1	4-2	Overall	1-QW	1-KW	1-QK	2-QW	2-KW	2-QK	3-QW	3-KW	3-QK	Overal	1	2	13	Overal
Activation sparsity(%)	30.6	41.6	44.38	59.53	36.77	47.8	53.86	73.14	52.43	61.0	67.63	69.11	72.84	79.37	70.38	71.98	33.63	0	0	0	0	0	0	0	0	0	0	71.76	0	0	2.51
Weight sparsity(%)	75.0	75.0	75.0	75.0	84.99	84.99	84.99	84.99	84.99	84.99	84.99	84.99	84.99	89.99	89.99	89.99	83.4	99.49	99.49	99.41	96.76	96.7	97.73	97.11	97.06	98.13	93.97	89.99	89.99	89.99	89.99
Theo. Speed Up	5.76	6.85	7.19	9.88	10.54	12.77	14.45	24.83	14.01	17.09	20.6	21.58	24.55	48.5	33.76	35.7	8.85	199.9	199.9	171.0	30.87	30.38	44.24	34.67	34.04	53.62	16.6	35.42	10.0	10.0	10.26

Fig. 22: Model-inference performance comparison for different layers in the five DNN models. Note that the theoretical speedup is a loose upper bound, e.g., $100\times$ speedup on 99% sparsity.

Thus, cuSparse is only useful for matrix multiplication with extremely high sparsity, which rarely happens in practice.

Second, Sparse Tensor Core [72] has a fixed speedup of $1.86\times$ over CUTLASS because it is designed to apply a fixed pruning ratio of 75% (and thus the same sparsity ratio of 75%) on only one sparse input matrix (matrix B in this experiment). As a result, it cannot take advantage of the other input matrix's sparsity, which significantly limits the acceleration ratio.

In contrast, our bitmap-based dual-side SpGEMM exploits the sparsity of both input and weight matrices and thus performs much better than cuSparse and Sparse Tensor Core. E.g., when the sparsity of matrix B is 99%, we achieve a speedup of $13.4\times$ over CUTLASS even when the sparsity of matrix A is 0; and if the sparsity of matrix A increases to 99.9%, the speedup is as high as $23\times$, which is $13.7\times$ better than cuSparse under the same sparsity of matrices A and B (i.e., 99.9% and 99%, respectively). Furthermore, even when the sparsity of matrix B is 0, our SpGEMM becomes faster than CUTLASS when the sparsity of matrix A is higher than $\sim$25%. Consequently, our SpGEMM achieves a significant speedup over the dense case for a wide range of sparsity (i.e., unless the sparsity ratios of *both* matrix A and matrix B are lower than $\sim$25%), making it highly useful for various SpMM.

D. Performance of Real Neural-Network Inference

Putting it together, we next evaluate the performance of real neural-network inference using the aforementioned five DNN models. For CNN models, we compare the performance in five cases: 1) *Dense Explicit* is dense GEMM based on CUTLASS with explicit im2col; 2) *Dense Implicit* is dense GEMM with implicit im2col provided by cuDNN; 3) *Single Sparse Explicit* is Sparse Tensor Core [72] with explicit im2col; 4) *Single Sparse Implicit* only exploits weight matrix sparsity with our SpCONV; and 5) *Dual Sparse Implicit* is our dual-side sparsity method with both feature map and weight sparsity. For BERT-base encoder and RNN models without im2col, we compare the performance in three cases: 1) *Dense GEMM* based on CUTLASS; 2) *Single Sparse GEMM* based on Sparse Tensor Core [72]; and 3) *Dual Sparse GEMM* which is our method.

Figure 22 shows the layer-wise and full-model speedup of the five DNNs. We select a set of representative layers for brevity because the rest layers have the same shape. For CNN models, the speedup is normalized to *Dense Implicit* which outperforms *Dense Explicit* due to optimized im2col operation in convolution. *Single Sparse Explicit* [72] is also faster than *Dense Explicit* by leveraging the sparsity of weight matrix, but is not always faster than *Dense Implicit* with a speedup ranging from $0.78\times$ to $1.74\times$ ($1.36\times$ on average). Benefiting from our bitmap-based implicit im2col and sparse weight matrix, *Single Sparse Implicit* is faster than *Dense Implicit* in most cases, even it only takes advantage of weight matrix sparsity. It achieves an average speedup of $1.92\times$ ranging from $0.63\times$ to $4.5\times$. By exploiting dual side sparsity and bitmap-based implicit im2col, our *Dual Sparse Implicit* method significantly outperforms all the other methods and achieves a speedup of $1.25\times$-$7.49\times$ over *Dense Implicit*. The average speedup is $4.38\times$ which is $2.22\times$ higher than *Single Sparse Explicit* [72].

Figure 22 also shows that our method achieves a speedup close to the theoretical upper bound in some CONV layers. A tight estimation of the upper bound is difficult because it depends on the non-zeros' distributions. The small speedups for some layers (e.g., ResNet-18 layer 5-4) are due to their small sizes, where the performance is bound by data movement.

For BERT-base encoder and RNN models, the speedup is normalized to *Dense GEMM*. *Single Sparse GEMM* [72] is always faster than *Dense GEMM* but has a small speedup of only $1.20\times$-$1.77\times$ ($1.51\times$ on average). Our method significantly outperforms *Single Sparse GEMM* with a speed of $3.62\times$-$8.45\times$. Our average speedup is $6.74\times$, which is $3.46\times$ higher than *Single Sparse GEMM* because Sparse Tensor Core [72] only accelerates SpMM with a hard limit of 75%, while the pruned BERT-base encoder model [54] and RNN [73]

has more than 90% weight sparsity. Recall the example in Figure 6, our work can go beyond the fixed-ratio limit due to our sparse tiling approach. For very sparse matrices, the proposed two-level bitmap encoding also helps because some empty warps are skipped as a whole, as shown in Figure 9.

E. Hardware Overhead

Finally, we evaluate the hardware overhead and power consumption of shared buffers and queues using CACTI 7 [4] with 22 nm process technology and scale them to 12 nm [59]. We estimate Accumulation Operand Collector and Float Point Adders overheads and energy consumption in RTL implementation. As shown in Table IV, our design introduces a total hardware overhead of 12.846 mm^2, which is 1.5% of the whole V100 die area of 815 mm^2, and it consumes an additional 3.89 W that is 1.6% of V100's 250 W TDP.

TABLE IV: Area and power overhead estimation.

Module Name	Area Overhead (mm^2, 12 nm)	Power Consumption (W, 12 nm)
Float Point Adders	0.121	2.35
Accumulation Operand Collector	1.51	0.46
Shared Accumulation Buffer	11.215	1.08
Total overhead on V100	12.846 (1.5%)	3.89 (1.60%)

VII. Conclusion

In this paper, for the first time, we demonstrate the feasibility of achieving a meaningful speedup for both SpGEMM and SpCONV on GPU Tensor Core with minimal hardware extension. The key insight is combing outer product of matrix multiplication and bitmap-base sparse encoding to fully leverage dual-side sparsity for highly-efficient GEMM and implicit im2col. Our design supports a wide range of sparsity ratios and outperforms state-of-the-art baselines by up to one order of magnitude with negligible hardware overhead, shedding light for the next performance breakthrough of future GPUs.

Acknowledgements We thank the anonymous reviews for their thoughtful comments and suggestions. The contribution of Jingwen Leng to this work was supported by the National Natural Science Foundation of China (NSFC) grant 62072297.

References

[1] A. F. Agarap, "Deep learning using rectified linear units (relu)," *CoRR*, vol. abs/1803.08375, 2018.

[2] A. Aimar, H. Mostafa, E. Calabrese, A. Rios-Navarro, R. Tapiador-Morales, I.-A. Lungu, M. B. Milde, F. Corradi, A. Linares-Barranco, S.-C. Liu *et al.*, "Nullhop: A flexible convolutional neural network accelerator based on sparse representations of feature maps," *IEEE transactions on neural networks and learning systems*, 2018.

[3] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: Ineffectual-neuron-free deep neural network computing," *ACM SIGARCH Computer Architecture News*, 2016.

[4] R. Balasubramonian, A. B. Kahng, N. Muralimanohar, A. Shafiee, and V. Srinivas, "Cacti 7: New tools for interconnect exploration in innovative off-chip memories," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 14, no. 2, pp. 1–25, 2017.

[5] J. Bradbury, S. Merity, C. Xiong, and R. Socher, "Quasi-recurrent neural networks," in *International Conference on Learning Representations (ICLR)*, 2017.

[6] S. Cao, L. Ma, W. Xiao, C. Zhang, Y. Liu, L. Zhang, L. Nie, and Z. Yang, "Seernet: Predicting convolutional neural network feature-map sparsity through low-bit quantization," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2019, pp. 11 216–11 225.

[7] S. Cao, C. Zhang, Z. Yao, W. Xiao, L. Nie, D. Zhan, Y. Liu, M. Wu, and L. Zhang, "Efficient and effective sparse lstm on fpga with bank-balanced sparsity," in *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2019, pp. 63–72.

[8] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE journal of solid-state circuits*, 2016.

[9] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cudnn: Efficient primitives for deep learning," *arXiv preprint arXiv:1410.0759*, 2014.

[10] J. Choquette and W. Gandhi, "Nvidia a100 gpu: Performance & innovation for gpu computing," in *2020 IEEE Hot Chips 32 Symposium (HCS)*. IEEE Computer Society, 2020, pp. 1–43.

[11] J. Choquette, O. Giroux, and D. Foley, "Volta: Performance and programmability," *Ieee Micro*, vol. 38, no. 2, pp. 42–52, 2018.

[12] J. Cong and B. Xiao, "Minimizing computation in convolutional neural networks," in *International conference on artificial neural networks*. Springer, 2014, pp. 281–290.

[13] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: pre-training of deep bidirectional transformers for language understanding," *CoRR*, vol. abs/1810.04805, 2018.

[14] X. Dong, J. Huang, Y. Yang, and S. Yan, "More is less: A more complicated network with less inference complexity," in *IEEE Conference on Computer Vision and Pattern Recognition*, 2017.

[15] M. Figurnov, M. D. Collins, Y. Zhu, L. Zhang, J. Huang, D. Vetrov, and R. Salakhutdinov, "Spatially adaptive computation time for residual networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2017, pp. 1039–1048.

[16] Y. Gan, Y. Qiu, J. Leng, M. Guo, and Y. Zhu, "Ptolemy: Architecture Support for Robust Deep Learning," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020.

[17] A. Gondimalla, N. Chesnut, M. Thottethodi, and T. Vijaykumar, "Sparten: A sparse tensor accelerator for convolutional neural networks," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 151–165.

[18] K. Greff, R. K. Srivastava, J. Koutník, B. R. Steunebrink, and J. Schmidhuber, "Lstm: A search space odyssey," *IEEE transactions on neural networks and learning systems*, vol. 28, no. 10, pp. 2222–2232, 2016.

[19] Y. Guan, J. Leng, C. Li, Q. Chen, and M. Guo, "How Far Does BERT Look At:Distance-based Clustering and Analysis of BERT′s Attention," *arXiv preprint arXiv:2011.00943*, 2020.

[20] C. Guo, B. Y. Hsueh, J. Leng, Y. Qiu, Y. Guan, Z. Wang, X. Jia, X. Li, M. Guo, and Y. Zhu, "Accelerating sparse dnn models without hardware-support via tile-wise sparsity," in *International Conference for High Performance Computing, Networking, Storage and Analysis*, 2020.

[21] C. Guo, Y. Zhou, J. Leng, Y. Zhu, Z. Du, Q. Chen, C. Li, B. Yao, and M. Guo, "Balancing Efficiency and Flexibility for DNN Acceleration via Temporal GPU-Systolic Array Integration," in *Proceedings of the Design Automation Conference*, 2020.

[22] 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.

[23] 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*, 2016.

[24] 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.

[25] K. Hazelwood, S. Bird, D. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, A. Kalro *et al.*, "Applied machine learning at facebook: A datacenter infrastructure perspective," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018, pp. 620–629.

[26] K. He, G. Gkioxari, P. Dollár, and R. Girshick, "Mask r-cnn," in *IEEE international conference on computer vision*, 2017.

[27] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[28] K. Hegde, H. Asghari-Moghaddam, M. Pellauer, N. Crago, A. Jaleel, E. Solomonik, J. Emer, and C. W. Fletcher, "Extensor: An accelerator for sparse tensor algebra," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 319–333.

[29] ——, "Extensor: An accelerator for sparse tensor algebra," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 319–333.

[30] Huggingface. [Online]. Available: https://github.com/huggingface/block_movement_pruning#fine-pruned-models

[31] Y. Ioannou, D. Robertson, R. Cipolla, and A. Criminisi, "Deep roots: Improving cnn efficiency with hierarchical filter groups," in *IEEE Conference on Computer Vision and Pattern Recognition*, 2017.

[32] N. Kalchbrenner, E. Elsen, K. Simonyan, S. Noury, N. Casagrande, E. Lockhart, F. Stimberg, A. Oord, S. Dieleman, and K. Kavukcuoglu, "Efficient neural audio synthesis," in *International Conference on Machine Learning*. PMLR, 2018, pp. 2410–2419.

[33] M. Khairy, Z. Shen, T. M. Aamodt, and T. G. Rogers, "Accel-sim: an extensible simulation framework for validated gpu modeling," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 473–486.

[34] T. Kong, F. Sun, A. Yao, H. Liu, M. Lu, and Y. Chen, "Ron: Reverse connection with objectness prior networks for object detection," in *IEEE Conference on Computer Vision and Pattern Recognition*, 2017.

[35] A. Lavin and S. Gray, "Fast algorithms for convolutional neural networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 4013–4021.

[36] J. Lew *et al.*, "Analyzing machine learning workloads using a detailed GPU simulator," in *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS*. IEEE, 2019, pp. 151–152.

[37] B. Liu, M. Wang, H. Foroosh, M. Tappen, and M. Pensky, "Sparse convolutional neural networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2015, pp. 806–814.

[38] S. Liu, J. E. Lindholm, M. Y. Siu, B. W. Coon, and S. F. Oberman, "Operand collector architecture," Nov. 16 2010, uS Patent 7,834,881.

[39] X. Liu, J. Pool, S. Han, and W. J. Dally, "Efficient sparse-winograd convolutional neural networks," in *International Conference on Learning Representations*, 2018.

[40] L. Lu and Y. Liang, "Spwa: an efficient sparse winograd convolutional neural networks accelerator on fpgas," in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*. IEEE, 2018, pp. 1–6.

[41] T. Luong, H. Pham, and C. D. Manning, "Effective approaches to attention-based neural machine translation," in *The Conference on Empirical Methods in Natural Language Processing*, L. Màrquez, C. Callison-Burch, J. Su, D. Pighin, and Y. Marton, Eds., 2015.

[42] A. Mishra, J. A. Latorre, J. Pool, D. Stosic, D. Stosic, G. Venkatesh, C. Yu, and P. Micikevicius, "Accelerating sparse deep neural networks," 2021.

[43] M. Naumov, L. Chien, P. Vandermersch, and U. Kapasi, "Cusparse library," in *GPU Technology Conference*, 2010.

[44] Nvidia, "Cublas library," *NVIDIA Corporation, Santa Clara*, 2008.

[45] ——, "Nvidia a100 tensor core architecture," in *Technical report*. NVIDIA, 2020.

[46] C. Nvidia, "Cutlass library," *NVIDIA Corporation, Santa Clara, California*, vol. 15, no. 27, p. 31, 2008.

[47] T. NVIDIA, "V100 gpu architecture. the world's most advanced data center gpu. version wp-08608-001_v1. 1," *NVIDIA. Aug*, p. 108, 2017.

[48] S. Pal, J. Beaumont, D.-H. Park, A. Amarnath, S. Feng, C. Chakrabarti, H.-S. Kim, D. Blaauw, T. Mudge, and R. Dreslinski, "Outerspace: An outer product based sparse matrix multiplication accelerator," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018, pp. 724–736.

[49] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "Scnn: An accelerator for compressed-sparse convolutional neural networks," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 2, pp. 27–40, 2017.

[50] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in neural information processing systems*, 2019, pp. 8026–8037.

[51] Y. Qiu, J. Leng, C. Guo, Q. Chen, C. Li, M. Guo, and Y. Zhu, "Adversarial Defense Through Network Profiling Based Path Extraction," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019.

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

[53] M. Ren, A. Pokrovsky, B. Yang, and R. Urtasun, "Sbnet: Sparse blocks network for fast inference," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 8711–8720.

[54] V. Sanh, T. Wolf, and A. M. Rush, "Movement pruning: Adaptive sparsity by fine-tuning," in *Advances in Neural Information Processing Systems*, 2020. [Online]. Available: https://papers.nips.cc/paper/2020/file/eae15aabaa768ae4a5993a8a4f4fa6e4-Paper.pdf

[55] N. Sato and W. Tinney, "Techniques for exploiting the sparsity or the network admittance matrix," *IEEE Transactions on Power Apparatus and Systems*, vol. 82, no. 69, pp. 944–950, 1963.

[56] S. Shi and X. Chu, "Speeding up convolutional neural networks by exploiting the sparsity of rectifier units," *arXiv:1704.07724*, 2017.

[57] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *International Conference on Learning Representations (ICLR)*, Y. Bengio and Y. LeCun, Eds., 2015.

[58] N. Srivastava, H. Jin, J. Liu, D. Albonesi, and Z. Zhang, "Matraptor: A sparse-sparse matrix multiplication accelerator based on row-wise product," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 766–780.

[59] A. Stillmaker and B. Baas, "Scaling equations for the accurate prediction of cmos device performance from 180 nm to 7 nm," *Integration*, vol. 58, pp. 74–81, 2017.

[60] G. Varma, K. Kothapalli *et al.*, "Dynamic block sparse reparameterization of convolutional neural networks," in *Proceedings of the IEEE International Conference on Computer Vision Workshops*, 2019, pp. 0–0.

[61] 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.

[62] H. Yang, S. Gui, Y. Zhu, and J. Liu, "Automatic neural network compression by sparsity-quantization joint learning: A constrained optimization-based approach," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020.

[63] H. Yang, Y. Zhu, and J. Liu, "Energy-constrained compression for deep neural networks via weighted sparse projection and layer input masking," in *International Conference on Learning Representations*, 2018.

[64] ——, "Ecc: Platform-independent energy-constrained deep neural network compression via a bilinear regression model," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019, pp. 11 206–11 215.

[65] Z. Yao, V. Gripon, and M. Rabbat, "A gpu-based associative memory using sparse neural networks," in *International Conference on High Performance Computing & Simulation (HPCS)*, 2014, pp. 688–692.

[66] Z. Yao, S. Cao, W. Xiao, C. Zhang, and L. Nie, "Balanced sparsity for efficient dnn inference on gpu," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, 2019, pp. 5676–5683.

[67] J. Yu, A. Lukefahr, D. Palframan, G. Dasika, R. Das, and S. Mahlke, "Scalpel: Customizing dnn pruning to the underlying hardware parallelism," *ACM SIGARCH Computer Architecture News*, 2017.

[68] C. Zhang, G. Sun, Z. Fang, P. Zhou, P. Pan, and J. Cong, "Caffeine: Toward uniformed representation and acceleration for deep convolutional neural networks," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 11, pp. 2072–2085, 2018.

[69] 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 *International Symposium on Microarchitecture (MICRO)*, 2016.

[70] Z. Zhang, H. Wang, S. Han, and W. J. Dally, "Sparch: Efficient architecture for sparse matrix multiplication," in *International Symposium on High Performance Computer Architecture (HPCA)*, 2020.

[71] X. Zhou, Z. Du, Q. Guo, S. Liu, C. Liu, C. Wang, X. Zhou, L. Li, T. Chen, and Y. Chen, "Cambricon-s: Addressing irregularity in sparse neural networks through a cooperative software/hardware approach," in *International Symposium on Microarchitecture (MICRO)*, 2018.

[72] 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.

[73] M. Zhu and S. Gupta, "To prune, or not to prune: Exploring the efficacy of pruning for model compression," in *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Workshop Track Proceedings*.

[74] N. Zmora, G. Jacob, L. Zlotnik, B. Elharar, and G. Novik, "Neural network distiller: A python package for DNN compression research," *CoRR*, vol. abs/1910.12232, 2019.