Taming Unstructured Sparsity on GPUs via Latency-Aware Optimization

Maohua Zhu
Department of Electrical and Computer Engineering
University of California, Santa Barbara
maohuazhu@ece.ucsb.edu

Yuan Xie
Department of Electrical and Computer Engineering
University of California, Santa Barbara
yuanxie@ece.ucsb.edu

***Abstract*—Neural Networks (NNs) exhibit high redundancy in their parameters so that pruning methods can achieve high compression ratio without accuracy loss. However, the very high sparsity produced by unstructured pruning methods is difficult to be efficiently mapped onto Graphics Processing Units (GPUs) because of its decoding overhead and workload imbalance. With the introduction of Tensor Core, the latest GPUs achieve even higher throughput for the dense neural networks. This makes unstructured neural networks fail to outperform their dense counterparts because they are not currently supported by Tensor Core. To tackle this problem, prior work suggests structured pruning to improve the performance of sparse NNs on GPUs. However, such structured pruning methods have to sacrifice a significant part of sparsity to retain the model accuracy, which limits the speedup on the hardware. In this paper, we observe that the Tensor Core is also able to compute unstructured sparse NNs efficiently. To achieve this goal, we first propose ExTensor, a set of sparse Tensor Core instructions with a variable input matrix tile size. The variable tile size allows a matrix multiplication to be implemented by mixing different types of ExTensor instructions. We build a performance model to estimate the latency of an ExTensor instruction given an operand sparse weight matrix. Based on this model, we propose a heuristic algorithm to find the optimal sequence of the instructions for an ExTensor based kernel to achieve the best performance on the GPU. Experimental results demonstrate that our approach achieves 36% better performance than the state-of-the-art sparse Tensor Core design.**

I. Introduction

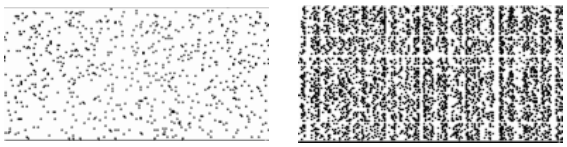
Deep Neural Networks (DNNs) [1] have achieved impressive progress in many different tasks, such as image recognition, speech recognition, and natural language processing [2]–[6]. The high representational and computational cost motivates both the industry and the academia to explore approaches on increasing the efficiency of the execution, including tensor decomposition [7], [8], data quantization [9]–[11], and network pruning [12]–[16]. The matrix or tensor decomposition generates matrices or tensors in lower rank, which are naturally faster on the commodity hardware platforms such as CPUs, GPUs, and ASICs. Data quantization has been supported by general purpose processors with low-precision pipelines. However, the network pruning is not supported efficiently by these hardware platforms, especially the latest GPUs that are highly optimized for dense neural networks.

Although the network pruning shrinks the model size, the irregular sparsity in the pruned networks requires dedicated sparse BLAS libraries. The randomness in the irregular sparsity exhibits imbalanced workload and introduce decoding overhead, which affects the performance on the commodity hardware. On the latest GPUs, the performance of the pruned sparse model may be even worse than the original dense counterpart if the sparsity is not sufficiently high [13], [17], [18]. This is because the latest GPUs are equipped with machine learning accelerator, Tensor Core [19], that is specifically designed for dense neural networks.

As such fine-grained granularity pruning is not hardware-friendly, coarse-grained pruning methods [13], [15], [20] are proposed to enable the pruned networks to utilize the optimized dense library and hardware. However, the coarse-grained pruned networks usually have lower sparsity than the fine-grained sparse networks and incur accuracy drop for large datasets. To tackle this problem, sparse NN accelerators [21]–[23] are proposed to boost the performance with specialized hardware design for sparse workload. Due to hardware resource limitations, these accelerators are designed for computing some specific sparse patterns that can be efficiently computed on dot-product engines. If a tile of the sparse weight matrix does not match the supported pattern, it will be computed as a dense tile. Consequently, structural pruning methods with spatial constraints must be applied to a network to ensure the supported pattern, which usually produces much less sparsity to retain the accuracy.

In this work, we enable the Tensor Cores [19] to efficiently compute unstructured sparse NNs with support both from hardware architecture and compiler. To achieve this goal, we first propose ExTensor, an extension to the Tensor Core PTX instruction set. The ExTensor architecture enables a sparse Tensor Core instruction to compute a variable-sized matrix multiplication instead of a fixed size in prior sparse Tensor Core design [24]. This allows programmers or compilers to mix multiple types of the ExTensor instructions to implement a sparse matrix multiplication for better performance. To assist the performance optimization, we further present a performance model to estimate the latency of all the proposed types of sparse Tensor Core instructions given an input sparse matrix. Finally, we propose a heuristic algorithm to find the optimal sequence of ExTensor instructions to achieve the best performance on the GPU. It is worth noting that, our method is orthogonal to coarse-grained or structured pruning methods. By utilizing our model and heuristic, the structured pruning

978-1-7281-1085-1/20/$31.00 ©2020 IEEE



(a) (b)

Fig. 1. The spatial distribution of non-zero elements in (a) an unstructured sparse neural network and (b) the corresponding part of a vector-wise pruned sparse neural network.

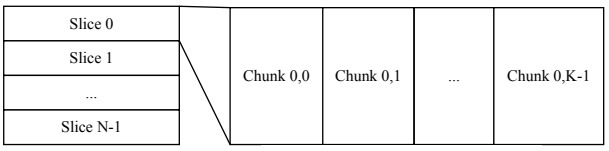


Fig. 2. The division of a weight matrix to exploit the hardware parallelism. Each chunk is 16×16 when the matrix is mapped to Tensor Cores. A chunk is computed with one Tensor Core instruction.

method will also have performance gains.

II. Background and Motivation

Sparse neural networks have achieved better performance and energy efficiency than dense networks on customized accelerators. However, much less effort has been put on GPUs, which are the most important commodity machine learning hardware platforms. Prior work either aims on improving the performance of sparse neural networks on GPUs with either unstructured sparsity or structured sparsity.

A. Directly Mapping Unstructured Sparsity on GPUs

Unstructured pruning has been popular since Deep Compression [12] achieved very high compression ratio on the commercial neural networks without accuracy loss. The high sparsity enables many neural networks to fit into the on-chip memories of hardware accelerators [21], [25], [26] so that off-chip memory traffic is significantly reduced. However, GPUs cannot exploit the sparsity in the same way because the Streaming Multiprocessors (SMs) only communicate through the off-chip memory. Therefore, the benefit of sparsity for GPUs only lies in the reduction of computation and memory footprint [13], [15], [22]. Unfortunately, the decoding overhead of the unstructured sparse matrix formats, e.g. Compressed Sparse Row (CSR), and the incontinuous memory accesses make the performance gain marginal. As dedicated hardware primitives (such as Tensor Core [19]) are introduced, the peak GPU throughput for the dense neural networks becomes $12\times$ higher than the CUDA cores [19]. This makes unstructured sparsity even more difficult to outperform the dense counterparts.

B. Structured Sparsity on GPUs

To efficiently improve the performance of the neural networks on GPUs, structured pruning has been studied from both the hardware side and the software side. A straightforward structured pruning method, often referred to as unified pruning [13], removes entire row and columns to form a smaller dense neural network. The neural networks generated by the unified pruning can be computed with the highly optimized software libraries and hardware primitives. However, the unified pruning puts too strict constraints on the positions of the non-zero elements, which incurs significant impact on the model's accuracy [27]. To alleviate the accuracy drop, fine-grained structured pruning methods [22], [24], [28] are proposed. These methods put less spatial constraints on the non-zero elements so that they could achieve moderate speedup with acceptable accuracy loss, i.e. less than 1%. Although the spatial constraints are more flexible, the sparsity achieved by fine-grained structured pruning methods is generally much lower than that by the unstructured pruning if the same model accuracy is enforced. For example, Figure 1 shows the spatial distribution of the non-zero elements in a part of a weight matrix in an unstructured sparse ResNet-50 and that of the corresponding part in a fine-grained structured sparse ResNet-50 pruned by the vector-wise pruning method [24].

Both the networks have the same accuracy, but the unstructured network exhibits 96% sparsity while the structured one has only 75% sparsity. This is because the vector-wise structured pruning method only aims to reduce the number of dot product operations in the neural network so that it is more efficient on the dot-product (DP) unit based Tensor Core [29]. However, the relatively low sparsity puts a tight theoretical upper bound to the speedup on the GPUs. To achieve higher speedup than the structured pruning, we explore if the high unstructured sparsity could be efficiently mapped to the Tensor Core with a software and hardware co-design to minimize the number of dot product operations.

C. Baseline Tensor Core Architecture

To increase the performance of the neural networks on GPUs, Tensor Core was introduced since the Volta architecture [19]. The Tensor Core is designed to compute half-precision floating-point matrix multiplication, which is the core operation in neural networks. In a Tensor Core based matrix multiplication $D = A \times B + C$, the weight matrix A is divided into multiple slices and chunks to be mapped to Tensor Core instructions, as shown in Figure 2. Every $16 \times 16 \times 16$ tile in the weight matrix A is computed with a warp-level Tensor Core instruction called `wmma.mma`. A `wmma.mma` instruction is mapped to two Tensor Cores in one SM.

Figure 3 shows how a `wmma.mma` operation is mapped to the Tensor Core architecture [29]. There are two *octets* in a Tensor Core. An octet consists of eight Dot Product (DP) units, each of which is able to compute a 4-dim vector dot product in a cycle. The octet has operand buffers to feed the DP units with the tiled data when executing the `wmma.mma` instruction. The eight DP units are further divided into two groups, and each group has a dedicated Matrix `A` buffer and an accumulator buffer. The operand Matrix `B` buffer is shared by the two groups for data reuse.

Since the Tensor Core architecture is based on 4-dim DP units, the sparsity does not help to reduce the computational latency unless all elements in a 4-dim vector are all zero. Therefore, prior work [24] prunes each chunk to $1/4$ of its original size to cut the latency of the instruction `wmma.mma`

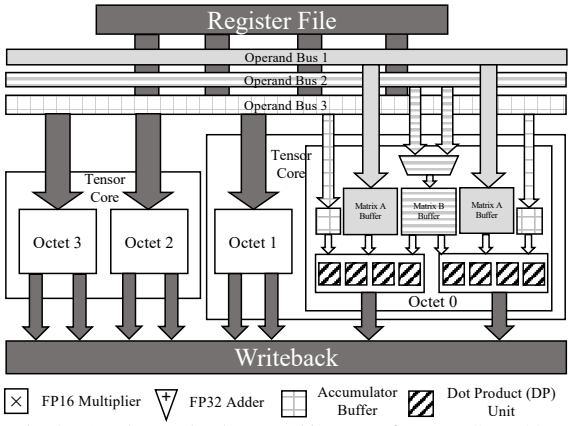


Fig. 3. An abstract hardware architecture of Tensor Core [29]

to half of the original instruction with an algorithm-hardware co-design. However, this vector-wise pruning enforces every vector to have the same sparsity, in which the sparsity is bound by the densest vector in the matrix. To further remove the redundancy and achieve higher speedup, we aim to efficiently map the unstructured sparsity on the Tensor Core in this work, and *use the improved Sparse Tensor Core architecture [24] as the baseline for comparison* in our experiments.

III. Enabling Arbitrary Sparsity on Tensor Core

Prior work in utilizing the Tensor Core to compute sparse NNs requires a specialized vector-wise pruning method [24]. Figure 4(a) illustrates how the vector-wise sparse network is computed with the Tensor Core. In this baseline architecture, each 16-dim vector in a chunk is pruned to have at most 4 non-zero elements so that the number of 4-dim operations is reduced to 25% of the dense chunk. The pruned chunk has associated coordinates of each non-zero element for fetching the corresponding rows from the dense input matrix. This baseline sparse Tensor Core design adds an offset-based register fetcher to enable the 4 non-zero elements to be computed in one 4-dim DP unit. Therefore, the hardware utilization rate of the DP units is high. However, the vector-wise pruning could not achieve very high sparsity since it enforces every chunk to have the same number of non-zero elements.

A. Extension for Sparse Tensor Core Instructions

With the observation that the baseline architecture could also be used to compute unstructured sparsity without further hardware change, we propose *ExTensor*, an extension for the Tensor Core instruction set to enable the GPU accelerator to efficiently compute unstructured sparse neural networks. While still being capable to compute $16 \times 16 \times 16$ matrix multiplications as the dense Tensor Core, ExTensor also supports a variable chunk width `L`, which can be any multiple of 16. The ExTensor instruction with chunk width `L` is denoted as `wmma.smma.L`, where `smma` stands for sparse matrix multiply-accumulate, and `L` is the chunk width computed by the instruction. By this definition, a `wmma.smma.L` instruction computes a $16 \times 16 \times L$ sparse matrix multiplication. Since the $16 \times L$ sparse weight matrix is unstructured, ExTensor encodes the 16 L-width chunks to a new sparse format to

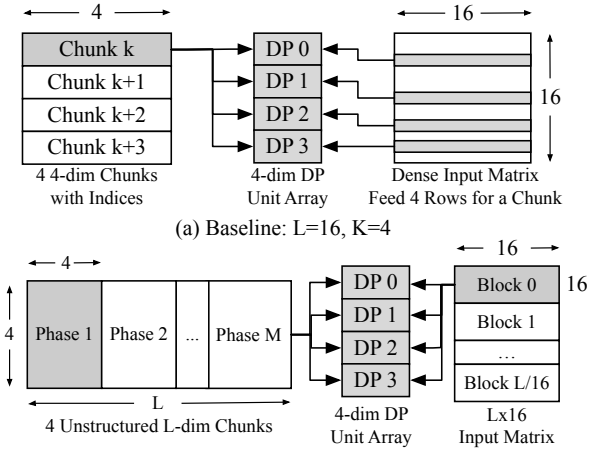


Fig. 4. The execution flow in a group of DP units of the sparse Tensor Core.

minimize the number of *phases* in the execution of a warp-level instruction. Each L-dim row in a chunk is encoded to its non-zero elements along with their coordinates. Since the number of non-zero elements may vary across the rows, the tail of the shorter rows are padded with zeros to form a chunk.

In the structured baseline scheme, there are maximally 4 non-zero elements in each row of a chunk so that all the elements can be computed in a 4-dim DP unit concurrently. Different from the baseline, ExTensor does not presume the number of non-zero elements. Therefore, the execution of an ExTensor instruction is divided into multiple phases, in which 4 non-zero elements are consumed by the DP units. The number of phases `M` is determined by the maximum number of the non-zero elements `K` in any of the 16 L-width chunks, where $M = \lceil K/4 \rceil$. For example, if a row in a 32-width chunk has 9 non-zero elements and any other row does not have more than 9 non-zero elements, this ExTensor instruction will be executed in $\lceil 9/4 \rceil = 3$ phases.

In each phase, the 4 non-zero elements $NZ\,[4 \times M]$ through $NZ\,[4 \times M + 3]$ from each row are fetched from the register file to the DP units. The Tensor Core input register block size is 16×16 [19]. If `L` is greater than 16, the $L \times 16$ input matrix has to be stored in $L/16$ register blocks. During one phase of the ExTensor execution, all necessary blocks are fetched sequentially from Block 0 to Block $L/16$. If there is no coordinate between $16 \times i$ and $16 \times i + 15$ in the weight chunk of this phase, Block i will not be fetched from register because the data in that block is not referred. Ideally, if all the coordinates of the non-zero elements in a phase fall in the range $[16 \times i, 16 \times i + 15]$, only Block i will be fetched.

B. Performance Model of the Sparse Tensor Core Instructions

The latency of the ExTensor instructions depends on their weight and input operands. Furthermore, since ExTensor supports variable `L`, a weight matrix can be computed with any type of `wmma.smma.L` instruction. Therefore, there is an optimal combination of the `wmma.smma.L` instructions of different `L` that minimizes the overall latency for the computation of a weight matrix. To find the optimal combination of

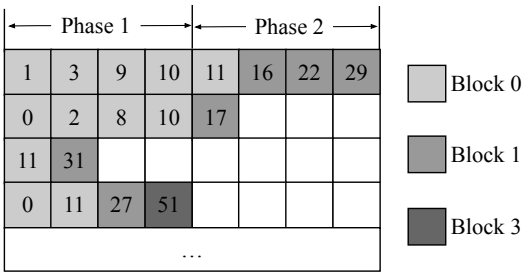


Fig. 5. An example of the coordinate matrix of a sparse weight chunk being computed by an ExTensor instruction `wmma.smma.64`. The chunk width $L = 64$ so that the chunk size is 16×64. The corresponding $L \times 16$ input matrix is stored in four register blocks, each of which contains a 16×16 tile.

the instructions, an estimation of the latency of the instructions is required during the compile time. We thus present a performance model of the `wmma.smma.L` instruction to solve this problem.

Based on the execution flow of the ExTensor instructions, the latency of an `wmma.smma.L` instruction depends on the chunk width `L` and the coordinates of the non-zero elements in the $16 \times L$ weight chunk. Figure 5 shows an example to show how a `wmma.smma.64` instruction is executed on the Tensor Core. In this example, the weight chunk size is $16 \times L$ and the maximum number of non-zero elements K equals 8. The number of phases is thus determined by $M = \lceil K/4 \rceil = 2$. In Phase 1, the non-zero elements require input rows from Block 0, 1 and 3 according to their coordinates.

The Tensor Core first fetches the coordinates and elements of the weight chunk to the buffer. The operand register bus allows 32 operand elements to be fetched from the register to the buffer in 2 cycles. Bound by the register access latency, it takes at least 2 cycles to access a new operand block. Then the coordinates are decoded in hardware to setup the data path to fetch the indexed rows from the input operand block. When the input rows are ready in the buffer, it takes 2 cycles to finish the 16 4-dim DP operations associated with a row in the chunk on the Tensor Core. Therefore, the latency of one phase can be modeled as

$$PhaseLat = \Sigma_{i=1}^{N_{row}} \Sigma_{j=0}^{N_{block}-1} max(Hist_{Row_i, Block_j}, 2) + 2$$

where `Hist` is the number of non-zero elements of Row i in the chunk that fall in the range of the indices of Block j. This model is based on the pipeline design that overlaps the register fetch with the DP operations. The constant 2 is the latency of the last DP operation.

With the latency of each phase of an instruction, the total latency of the ExTensor instruction can be calculated by adding them together. Given an input weight chunk, this performance model can be used to find the optimal set of instructions to compute this chunk. For example, a 16×48 chunk can be computed with (1) one `wmma.mma.48` instruction, or (2) one `wmma.mma.32` followed by one `wmma.mma.16`, or (3) one `wmma.mma.16` followed by one `wmma.mma.32`, or (4) three `wmma.mma.16` instructions. The latency varies across the four approaches and we can use our performance model to estimate the latency for each approach and pick the one with the shortest modeled latency.

Algorithm 1: Heuristic optimization for ExTensor code generation.

input : Coordinates of the sparse weight matrix, $Coord$;
Number of Tensor Cores, T;
Maximum allowed chunk width, $maxL$.
output: A list of chunk width, L.

```
slices = reshape(Coord, Coord.shape[0]/16, 16, -1)
N_tc = T/slices.shape[0]
for each s in slices do
    chunks = list(s[:, i : i + 16], i=0,16,32,...,s.shape[1] )
    converged = False
    while not converged do
        converged = True
        gain = 0
        idx = 0 for i in range(chunks.length − 1) do
            c = merge(chunks[i], chunks[i + 1])
            g = latency(chunks[i]) + latency(chunks[i + 1]) -
              latency(c)
            if g > gain and c.width ≤ maxL then
                idx=i
                gain=g
                converged=False
            end
        end
        if chunks.length ≤ N_tc then
            converged=True
        end
        if not converged then
            c = merge(chunks[idx], chunks[idx + 1])
            chunks[idx]=c
            remove(chunks[idx + 1])
        end
    end
end
return chunks
```

IV. Optimizing Kernel Execution Latency

A dense Tensor Core instruction `wmma.mma` computes a 16×16 matrix multiplication so that a weight matrix is divided to 16×16 chunks as shown in Figure 2. On the contrary, the ExTensor instruction `wmma.smma.L` computes a variable-size $16 \times L$ matrix multiplication. As a result, a 16-row slice in a weight matrix can be computed by any sequence of the ExTensor instructions `wmma.smma.`$\mathtt{L}_i$, if only $\Sigma_i L_i$ equals to the matrix width. As the latency of a `wmma.mma.L` instruction depends on the coordinates and the `L`, a compiler or programmer has to optimize the sequence for shorter total latency. Ideally, the sequence with the shortest latency is the optimal solution to the optimization problem. However, enumerating all the sequences is NP-complete and thus prohibited by the time complexity during the compilation time.

To optimize the instruction sequence in an acceptable time, we propose a heuristic algorithm to search the "optimal" sequence that minimizes the GPU kernel latency. This heuristic algorithm is hardware-dependent since it aims to improve the performance of a kernel on a target GPU. Similar to the `wmma.mma`, a `wmma.smma.L` instruction is executed by two Tensor Cores regardless of `L`. Assuming a GPU has T Tensor Cores, each 16-row slice of a weight matrix will be mapped to $N_{tc} = \frac{T}{N_{slice}}$ in parallel. Therefore, the goal of the heuristic is

to minimize the longest accumulated latency of the instructions mapped to a Tensor Core.

Algorithm 1 shows how the optimization method generates the instructions. The initial state of the algorithm divides each 16-row slices to 16×16 chunks (Line 4). Then the optimizer evaluates the latency of each chunk with the proposed performance model. It is worth noting that each chunk is computed with only one ExTensor instruction. The optimizer tries to find a merge of two adjacent chunks so that it returns the maximum latency reduction, which is defined by Line 11. If the maximum latency reduction is positive, the optimizer will merge the corresponding two chunks to one larger chunk. The optimization will stop when no further performance gain could be achieved or the number of the chunks is no more than the number of Tensor Cores. At last, the optimizer returns a list of chunks (Line 28) and every chunk in it is consumed by a `wmma.smma.L` instruction, where `L` is the width of that chunk.

V. Evaluation

To evaluate the performance of the ExTensor architecture and the code optimizer, we picked three popular neural networks, VGG-19 [2], ResNet-50 [3], and NMT [30] and simulated our design with a Tensor Core enabled NVIDIA Titan V GPU configuration in GPGPU-Sim [29].

A. Experimental Methodology

In the experiments, we extended the `wmma` PTX code model in the GPGPU-Sim simulator [29], [31]–[33]. We implemented the `wmma.smma.L` instructions with $L = 16, 32, 48, 64$. The `wmma.smma.16` is the baseline architecture [24] shown in Figure 4(a). The simulated Titan V GPU has 80 SMs with 640 Tensor Cores. On the software side, we implemented the sparse matrix multiplication kernels based on CUTLASS [34], which is an open-source high-performance template library for matrix multiplication. The ExTensor based kernels are similar to the `wmma.mma` based kernels, but the `wmma.smma` instructions are called.

The benchmarks are pruned with the unstructured pruning method in Deep Compression [12] to 90% sparsity and 96% sparsity, respectively, to show the performance of different sparsity levels. The 90% sparse benchmarks have better accuracy than their dense counterparts, while the 96% sparse benchmarks exhibit more than 1% accuracy drop. We first evaluated the baseline with the three NN benchmarks. The baseline is reported to be 49% faster than the dense Tensor Core on average [24]. In our ExTensor design, the baseline is a special case when the sparse matrix multiplication is only computed with `wmma.smma.16` instructions. To compare the extended instructions with the baseline, we evaluated the benchmarks with `wmma.smma.32` and `wmma.smma.64`, respectively. Then we further optimized the kernels by enumerating `L` in the range $[16, 64]$ for each slice. That is, we compute a slice with different `L`s and *statically* select the `L` with the best performance. And finally we *dynamically optimized* the instructions in the compiling time with Algorithm 1.

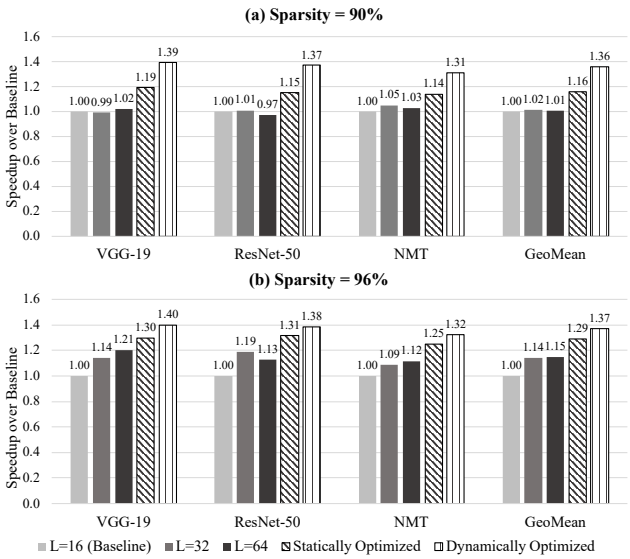


Fig. 6. The overall speedup over the baseline sparse Tensor Core architecture.

B. Evaluation Results

Figure 6 shows the normalized performance over the baseline, in which every Tensor Core instruction computes a $16 \times 16 \times 16$ matrix multiplication. When $L = 32$, all the chunks are 16×32 and computed with `wmma.mma.32` instructions. Since the positions of the non-zero elements vary across different layers and workloads, the speedup numbers are also different for the benchmarks. Similarly, the $L = 64$ results are the performance with only `wmma.smma.64` instructions. The *statically optimized* results are always better than the single-instruction results because this approach picks the best performing L for each slice in each layer. With our heuristic algorithm, the performance is further improved to $1.36\times$ on average for the 90% sparse benchmarks with no accuracy drop compared with the dense counterparts. If the sparsity is increased to 96%, the speedup becomes $1.37\times$, which is still better than any other schemes.

From Figure 6(a), we observe that the performance of the single-`L` schemes are similar on the 90% sparse benchmarks. The observation indicates that the unstructured sparsity does not favor any specific chunk size so that we have to optimize the kernel by mixing the instructions. For very sparse benchmarks in Figure 6(b), there is no unique single-`L` solution to all benchmarks, either. However, sparser networks generally favor larger `L`. Our heuristic optimizer outperforms the static optimizer because it allows multiple `L`'s within a slice. Especially, the benchmarks favor the dynamic optimization when they are less sparse because they have more diverse structures.

VI. Conclusion

In this work, we observe that the Tensor Core on the latest GPUs can also be used to compute sparse neural networks. To efficiently map sparse neural networks to the Tensor Core hardware, we propose ExTensor, an instruction set extension to the `wmma` PTX APIs. The ExTensor sparse Tensor Core instructions support $16 \times L \times 16$ sparse matrix multiplications

where L is a multiple of 16. The variable L allows a programmer or compiler to have multiple options to implement a matrix multiplication kernel by mixing different types of the ExTensor instructions. To find the optimal sequence of instructions to implement a matrix multiplication, we then propose a performance model to estimate the latency of an ExTensor instruction given its operands. However, searching the optimal sequence based on the latency model is an NP-complete problem. We thus design a heuristic optimization problem to improve the performance of a ExTensor based kernel in the compiling time. This algorithm can optimize the code in a reasonable time span and produce a faster kernel for popular NN workloads. Experimental results demonstrate that it is 1.36× faster than the state-of-the-art sparse Tensor Core design.

VII. Acknowledgement

This work was supported in part by NSF 1817037, 1725447, 1730309, and NSF grant CCF 1740352 and SRC nCORE NC-2766-A.

References

[1] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, p. 436, 2015.

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

[3] 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 (CVPR)*, 2016, pp. 770–778.

[4] H. Sak, A. W. Senior, and F. Beaufays, "Long short-term memory recurrent neural network architectures for large scale acoustic modeling." in *Interspeech*, 2014, pp. 338–342.

[5] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using rnn encoder-decoder for statistical machine translation," *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2014.

[6] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," *Advances in neural information processing systems (NeurIPS)*, pp. 3104–3112, 2014.

[7] J. Xue, J. Li, D. Yu, M. Seltzer, and Y. Gong, "Singular value decomposition based low-footprint speaker adaptation and personalization for deep neural network," in *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2014, pp. 6359–6363.

[8] A. Novikov, D. Podoprikhin, A. Osokin, and D. P. Vetrov, "Tensorizing neural networks," in *Advances in neural information processing systems*, 2015, pp. 442–450.

[9] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1," *arXiv preprint arXiv:1602.02830*, 2016.

[10] S. Zhou, Y. Wu, Z. Ni, X. Zhou, H. Wen, and Y. Zou, "Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients," *arXiv preprint arXiv:1606.06160*, 2016.

[11] L. Deng, P. Jiao, J. Pei, Z. Wu, and G. Li, "Gxnor-net: Training deep neural networks with ternary weights and activations without full-precision memory under a unified discretization framework," *Neural Networks*, vol. 100, pp. 49–58, 2018.

[12] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *International Conference on Learning Representations (ICLR)*, 2016.

[13] 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 (NeurIPS)*, 2016, pp. 2074–2082.

[14] P. Molchanov, S. Tyree, T. Karras, T. Aila, and J. Kautz, "Pruning convolutional neural networks for resource efficient inference," *arXiv preprint arXiv:1611.06440*, 2016.

[15] X. Sun, X. Ren, S. Ma, and H. Wang, "meprop: Sparsified back propagation for accelerated deep learning with reduced overfitting," in *International Conference on Machine Learning*, 2017, pp. 3299–3308.

[16] J. Ye, X. Lu, Z. Lin, and J. Z. Wang, "Rethinking the smaller-norm-less-informative assumption in channel pruning of convolution layers," *arXiv preprint arXiv:1802.00124*, 2018.

[17] 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*, vol. 45, no. 2, pp. 548–560, 2017.

[18] Y. Ji, L. Liang, L. Deng, Y. Zhang, Y. Zhang, and Y. Xie, "Tetris: tile-matching the tremendous irregular sparsity," in *Advances in Neural Information Processing Systems*, 2018, pp. 4115–4125.

[19] NVIDIA, "V100 gpu architecture. the world's most advanced data center gpu. version wp-08608-001_v1. 1," *NVIDIA. Aug*, p. 108, 2017.

[20] T. Zhang, K. Zhang, S. Ye, J. Li, J. Tang, W. Wen, X. Lin, M. Fardad, and Y. Wang, "Adam-admm: A unified, systematic framework of structured weight pruning for dnns," *arXiv preprint arXiv:1807.11091*, 2018.

[21] 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 *The 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE Press, 2016, p. 20.

[22] 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 *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 15–28.

[23] 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," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2017, pp. 27–40.

[24] 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*. ACM, 2019, pp. 359–371.

[25] 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," in *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA)*. IEEE Press, 2016, pp. 243–254.

[26] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: Ineffectual-neuron-free deep neural network computing," in *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*. IEEE, 2016, pp. 1–13.

[27] H. Mao, S. Han, J. Pool, W. Li, X. Liu, Y. Wang, and W. J. Dally, "Exploring the regularity of sparse structure in convolutional neural networks," *arXiv preprint arXiv:1705.08922*, 2017.

[28] Z. Yao, S. Cao, and W. Xiao, "Balanced sparsity for efficient dnn inference on gpu," *arXiv preprint arXiv:1811.00206*, 2018.

[29] M. A. Raihan, N. Goli, and T. Aamodt, "Modeling deep learning accelerator enabled gpus," *arXiv preprint arXiv:1811.08309*, 2018.

[30] T. Luong, H. Pham, and C. D. Manning, "Effective approaches to attention-based neural machine translation," in *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, 2015, pp. 1412–1421.

[31] A. Bakhoda, G. L. Yuan, W. W. 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*. IEEE, 2009, pp. 163–174.

[32] M. Khairy, J. Akshay, T. M. Aamodt, and T. G. Rogers, "Exploring modern gpu memory system design challenges through accurate modeling," *arXiv preprint arXiv:1810.07269*, 2018.

[33] J. Lew, D. Shah, S. Pati, C. Shaylin, M. Zhang, A. Sandhupatla, C. Ng, N. Goli, M. D. Sinclair, T. G. Rogers, and T. M. Aamodt, "Analyzing machine learning workloads using a detailed gpu simulator," *arXiv preprint arXiv:1811.08933*, 2018.

[34] NVIDIA, "Cutlass: Fast linear algebra in cuda c++." [Online]. Available: https://devblogs.nvidia.com/cutlass-linear-algebra-cuda/