Adaptive Sparse Deep Neural Network Inference on Resource-Constrained Cost-Efficient GPUs

Ming Dun[1], Xu Zhang[1,2], Huawei Cao[1,3], Yuan Zhang[1,2], Junying Huang[1], Xiaochun Ye[1]
[1]State Key Lab of Processors, Institute of Computing Technology, CAS, Beijing, China,
[2]School of Computer and Control Engineering, UCAS, Beijing, China,
[3]Nanjing Institute of InforSuperbahn, Nanjing, China.
{dunming, zhangxu22s1, caohuawei, zhangyuan-ams, huangjunying, yexiaochun}@ict.ac.cn

***Abstract*—Sparse Deep Neural Networks (SpDNNs) has gained great popularity and been widely applied in various machine learning area. Compared to traditional dense DNNs, the unpredictable irregularity and sparsity in the sparse weight matrices of SpDNNs make them difficult to be efficiently parallelled. Moreover, most of the recent advanced efforts to optimize SpDNNs are based on high-end GPUs like NVIDIA V100, which may not be affordable to individuals and smaller research groups. However, migrating the SpDNNs to those cost-efficient but resource-constrained GPUs confronts enormous challenges, including limitations in both memory and computing resources, as well as the tiresome hyper-parameter tuning in batch parallelism. In this paper, we accelerate SpDNNs on GPUs with more restricted resources through exploiting the memory and computing resources. On one hand, we design the adaptive memory-aware data partition scheme to reduce memory consumption automatically. On the other hand, we propose the Tensor core/CUDA core fusion mechanism to efficiently utilize the hetergeneous computing resources on modern GPU architecture. To the best of our knowledge, we are the first to improve the performance on SpDNNs through adaptive memory tuning and utilizing hetergeneous computing core concurrency. We compare our implementation with the state-of-art previous champions, and the results demonstrate that our work achieves the highest speedup of 1.35 $\times$ and 1.39$\times$ compared to 2022 champion S&Z on single and multiple NVIDIA Tesla T4 GPUs, respectively. What's more, our work can reach similar or even better throughput compared to 2020 champion H&P on 6 V100 GPUs with only 4 T4 GPUs.**

***Index Terms*—Sparse Neural Networks, GPU, Performance Optimization, Resource-Constrained**

I. Introduction

Deep Neural Networks (DNNs) have become the cornerstone in the field of modern artificial intelligence, providing pivotal contributions to massive machine learning applications ranging from image enhancement [1] [2], natural language processing [3], [4] and autonomous driving [5], [6]. In the mean time, DNN is rapidly evolving with growing network size, and the resultant extreme large models like GPT-3 (175B parameters) [7] and BERT (340M parameters) [8] are consuming overwhelming computation and memory resources during training and inference. Therefore, recent advanced researches focus on pruning the large DNNs to reduce the amount of weight parameters, leading to the prosperity of Sparse Deep Neural Networks (SpDNNs). However, due to the unpredictable irregularity within the sparse weight matrices, it is not trivial to optimize SpDNNs through efficient parallelism on high-performance architecture.

Aiming to accelerate the inference process of SpDNN, the 2019 MIT/IEEE/Amazon proposed the SpDNN Challenge track [9], which harnesses networks generated by Radix-Net [10]. The datasets provided by SpDNN Challenge consist of sparse neural network with diverse neuron size ranging from 1024 to 65536, layer number spanning from 120 to 1920, and sparsity degrees varying from 3% to merely 0.05%. Moreover, the amount of parameters within the network varies from 4M to 4B.

The inference procedure of SpDNN mostly depends on an iterative process involving two fundamental kernels: sparse matrix-dense matrix multiplication (SpMM) and ReLU activation. Prior works [11], [12] have demonstrated SpMM has the highest memory demand and is the most time-consuming operation in SpDNN, and thus the acceleration of SpMM is the key to accelerate the inference of SpDNN.

In the past few years, many researchers have been paying great attention to accelerate the inference of SpDNN, focusing on the main bottleneck SpMM. The 2020 champion H&P [12] proposed a novel SpMM algorithm based on the sliced-ELL format on GPU architecture, which can improve sparse data reuse. Meanwhile, another 2020 champion SNIG [13] developed an efficient engine for large sparse DNN inference using task graph parallelism, in order to effectively avoid unnecessary computations caused by zero entries during the inference iterations. In addition, the 2021 champion X&Y [11] defined SpMM optimization space, and then applied GPU-friendly space pruning along with designing the cost model to find the optimal SpMM execution scheme. Moreover, 2022 champion S&Z [14] implemented a similarity-based matrix transformation scheme to enable block-based SpMM algorithm on the high-throughput Tensor cores [15] in GPUs.

Nonetheless, most of the recent champions implement their optimization mechanisms on high-end GPUs like V100 [11], [14], [16], lacking the support for more resource-constrained but more cost-efficient GPUs like T4. We present the comparison between the features of the representative NVIDIA Tesla V100 and Turing T4 in Table I, where the prices are from Amazon. It is obvious that the V100 GPU doubles computing resources (CUDA core and Tensor core) and memory resources, and its memory bandwidth is 2.78$\times$ higher than

that of T4, but T4 GPU is 3.97× cheaper and only needs 28% power, thus it is more afforable for individuals or smaller research institutions to explore the field of deep learning.

However, the resource limitation poses new challenges when accelerating SpMM. Specifically, restricted computing and memory resources hinder the algorithm to catch up its performance on high-end GPUs. What's more, most of the previous efforts to alleviate memory consumption for neural networks like Zero-offload [17] and Superneurons [18] address on optimization for dense DNNs, which cannot be directly applied to SpDNNs because of their sparsity and irregularity. Besides, the batch parallelism [19], adopted by previous champions H&P and SNIG, use empirical parameters that need strenuous expert tuning when transferring to novel architecture. In terms of computing resources, a majority of advanced researches only use either the CUDA core [12] or the Tensor core [14], leaving the other cores idle, which is a waste of the precious resources.

TABLE I: GPU Feature Comparison

Feature	Tesla V100 GPU	Turing T4 GPU
Price	$4686	$1179
Power	250W	70W
Memory	32GB	16GB
Tensor/CUDA Core Number	640/5120	320/2560
Global Memory Bandwidth	900 GB/s	320 GB/s

In this paper, we improve the performance of SpDNN inference on cost and energy efficient GPUs with more restricted memory and computing resources. To alleviate the memory consumption of SpMM, we design the adaptive memory-aware data partition scheme that can automatically set the batch size of input and weight matrices at one time, in order to eliminate additional expert hyper-parameter tuning. In addition, we demonstrate the feasibility of concurrency between the heterogeneous CUDA core and Tensor core in NVIDIA Turing architecture, and recapitulate the concurrency pattern. To reach maximum concurrency, we propose the Tensor core/CUDA core fusion mechanism that splits the computation task of SpMM to two kind of cores, according to profiled specified load ratio [20]. What's more, we explore the data partition size in the two dimensions of matrices for tensor core routine to improve the efficency of data reuse and memory access. To guarantee the precision of SpMM in Tensor core, we adopt the lightweight emulation algorithm in S&Z [14]. We compare our implementation with the state-of-art champions in T4 GPUs, and the results depict that our work exceeds the previous champions in terms of throughput, with the highest acceleration rate at 1.35 × and 1.39× compared to 2022 champion S&Z on single and multiple T4 GPUs, respectively.

In summary, this paper makes the following contributions:

- We design the adaptive memory-aware partition scheme to automatically determine the batch size and layer transfer number through formulating a memory model, which can avoid memory overflow on large datasets at resource restriced GPUs.
- We propose the Tensor core/CUDA core fusion mechanism to fully leverage the heterogenous computing resources on GPU architecture following the concurrency pattern formulated by systematic analysis.
- We conduct comprehensive evaluation for our implementation with the state-of-art preivous champions on the representative resource-constrained NVIDIA T4 GPUs on official datasets. The results prove that our work obtains the highest throughput compared to previous champions, and can be executed on every dataset without encoutering memory overflow issues.

II. Background and Motivation

In this section, we present a refined overview of the Sparse DNN Challenge and GPU architecture. In addition, we design an experiment to demonstrate the potential concurrency between CUDA core and Tensor core to further exploit computing resources. Besides, we summarize the challenges to optimize SpMM on resource constrained GPUs.

A. Overview of Sparse DNN Challenge

The Sparse DNN Challenge [9] targets at improving the performance of inference tasks on sparse deep neural networks generated by RadiX-Net [10], which has drawn great attention worldwide. The input of the neural network contains 60,000 stacked images interpolated from MNIST dataset [21], where each row is a linearized resized image. The iterative inference process at the l^{th} layer of sparse DNN with N neurons can be formulated as (1), where $\mathbf{Y}_l \in \mathbb{R}^{M\times N}$, $\mathbf{W}_l \in \mathbb{R}^{N\times N}$ and $\mathbf{B} \in \mathbb{R}^{M\times N}$ denotes the input, weight and bias matrix, respectively. It is noteworthy that every weight matrix $\mathbf{W}_l$ is sparse in the Sparse DNN Challenge. As a consequence, the SpMM algorithm becomes the hotspot of performance optimization.

$$\mathbf{Y}_{l+1} = \mathrm{ReLU}(\mathbf{Y}_l \times \mathbf{W}_l + \mathbf{B}) \tag{1}$$

B. GPU Architecture

GPU is one of the most widely applied Single Instruction Multiple Data (SIMD) architecture, well known for its massive parallelism and high computation throughput. Take advanced NVIDIA Turing architecture [22] as an instance, each GPU is consist of multiple Streaming Multiprocessors (SMs). Within one of the four blocks in one SM, there are 16 INT32 CUDA cores, 16 FP32 CUDA cores, 2 Tensor cores [23] and a 64KB register file, as illustrated in Fig 1. In a word, a GPU is equipped with heterogeneous computation resources, where CUDA cores are typically responsible for scalar computing, and Tensor Cores support direct matrix multiplication ($\mathbf{D} = \mathbf{A} \times \mathbf{B} + \mathbf{C}$). Among them, Tensor cores can reach higher computation throughput than CUDA cores and enable better data reuse in GEMM [14]. Nonetheless, in Turing and Volta architecture, Tensor cores only support half-precision in matrices $\mathbf{A}$ and $\mathbf{B}$, while CUDA cores support computation in half-, single- or double-precision.

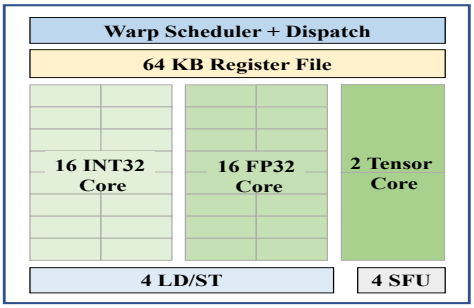

Fig. 1: Illustration of a Streaming Multiprocessor block in NVIDIA Turing architecture.

C. Potential Heterogeneous Core Concurrency Opportunity

To further investigate the feasibility of utilizing Tensor core and CUDA core concurrently, we construct a sample SpMM kernel following the reference inference calculation code [9], where each block contains Tensor core warps and CUDA core warps. Then we test the execution time under different task load ratio on a T4 GPU, whose results are shown in Fig 2. The metric *load ratio* [20] denotes the ratio of the original time of Tensor core kernel and CUDA core kernel, which is proportional to their computation size. We fix the computation size of tasks at Tensor cores and vary that in CUDA cores, and we abbreviate the two types of computing resources as TC and CC respectively in Fig 2. From the results, we can draw the conclusion that the growth of execution time follows a two-stage linear model: the co-running stage before the turning point and the solo-running stage after the turning point. Thus there exists the opportunity of concurrency between the heterogeneous cores, which can be utilized to further exploit the computing resources and accelerate SpMM process.

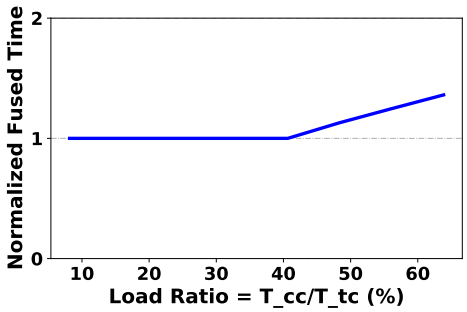

Fig. 2: Normalized execution time of the fused kernel under different load ratios with fixed Tensor Core load, TC and CC are the abbreviation for Tensor Core and CUDA core respectively.

D. Challenges on Resource Constrained Devices

The cost and energy efficiency, as well as the aspiration to bring large model inference to everyone, motivate us to optimize Sparse DNN on cheaper but resource constrained devices like T4 GPU. However, it is nontrivial to accelerate SpMM when resources are limited. There are two main challenges: GPU memory boundary and computing resource exploitation.

GPU Memory Boundary: Since cheaper GPUs often contains more confined global memory as depicted in Tabel I, how to fit the large input and sparse network under more restricted GPU memory boundary become one of the main obstacle. We have tested the open source code of previous 2020 champion SNIG [13], H&P [12] and 2021 champion X&Y [11] on a single T4 GPU, and they all fail for memory overflow at the network size 65536. Though SNIG and H&P does address the memory limitation through batch parallelism [24], they empirically assign the batch size, resulting in potential deficient in adaptability when switching to new architecture.

Efficient Computing Resource Exploitation: Most of advanced researchs only focus on accelerating the SpMM either on CUDA core [25] [26] or Tensor core [14] [27], leaving the other kind of cores idled. However, the validation of fused Tensor-CUDA core concurrency encourages us to fully exploit the GPU computing resource to optimize SpMM. What's more, although previous research Tacker [20] improve the performance between different dense routines based on fused cores on GPU, to the best of our knowledge, how to optimize sparse matrix multiplication through splitting one kernel between heterogeneous computing resources on GPUs is a problem still remain unsolved. Moreover, how to dividing the kernel properly is also a challenge: in one hand, improper computation load assignment on CUDA core may cause the fused core enter into solo-running stage in Fig 2; in the other hand, improper data granularity on Tensor and CUDA core can reduce the data reuse, causing performance degradation.

III. Our Approach

In this section, we present the details of our optimization approaches to accelerate the main hotspot SpMM, including adaptive memory-aware data partition scheme, heterogenous core fusion mechanism in GPU architecture, and parallel granularity tuning. We transform the weight matrices like S&Z [14], where the sparse parts of weight matrices are stored in CSR format while the dense parts are in BCSR format.

A. Adaptive Memory-aware Data Partition Scheme

As illustrated in Figure 3, there are two stages in the adaptive memory-aware data partition scheme: the initialization stage and online tuning stage. Before the inference process begins, the partition scheme automatically initializes the batch size of the input matrix and the number of layer transferred to GPU global memory at a time. It generates the batch size by solving the optimization problem with constrains formulated in (2). The object of the optimization problem is to maximize the memory for input, denoted as M_B, in order to reduce time-consuming input transmission between CPU and GPUs. M_B is proportional to batch size of input, thus from which the input partition length N_B can be derived.

Meanwhile, we formulate the constrains according to the following two criteria:

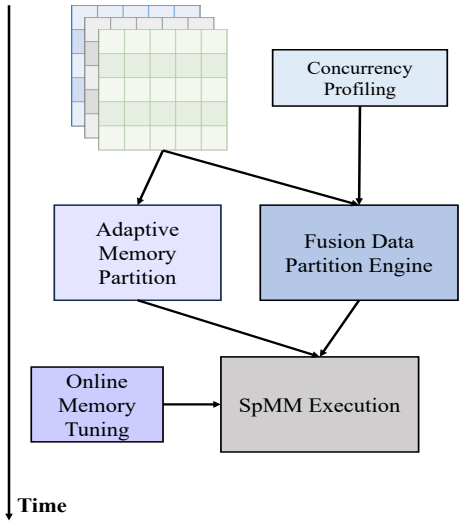

Fig. 3: Overview of optimizations on resource-constrained GPUs.

- **The overall memory footprint of SpMM process should not exceed the GPU global memory capacity $\mathbf{M_G}$.** The total memory consumption contains memory for partitioned input matrix and output matrix (M_B), for batched $2N_L$ weight matrices (M_L) and other auxiliary arrays (M_G). Typically, the auxilliary arrays mainly include the *category* array and *active* [12], which are responsible for permuting the rows of nonzero input and intermediate matrices respectively.
- **The batched layer transmission time should be overlapped with the corresponding computation time.** This criteria is to ensure the overlap between communication and computation to improve execution efficiency. The total batched layer transmission time is $N_L T_L$ with weight batch size N_L. T_0 is the computation time at input batch size N_0 with the same neuron, thus $T_0 * \frac{N_B}{N_0}$ is the approximation for computation time at batch size N_B.

Moreover, we add the constrain that the batch size of weight matrices needs to be larger than 1 to ensure the results are nonzero.

$$\text{maximize } M_B \tag{2}$$

$$\text{subject to } 2M_B + 2N_L M_L + M_O \leq M_G, \tag{2a}$$

$$N_L T_L \leq T_0 * \frac{N_B}{N_0}, \tag{2b}$$

$$N_L \geq 1. \tag{2c}$$

As shown in (2), the optimization problem is equivalent to a linear programming problem [28]. The solution of the problem depends on whether the whole input matrix can fit into the global memory of GPU. On one hand, if the whole input can be stored in the global memory, the optimal value M_B equals the original input memory consumption, and we only need to solve (2b) to derive the optimal value for weight matrices batch size. We use the largest solution in (2b) to improve global memory utilization. On the other hand, when the original input is larger than the GPU global memory capacity, we set the N_L to the minimal value 1 and then solve (2c) to generate optimal M_B. As the automatic hyper-parameter tuning in the initialization stage is through solving concise linear programming problem once, it will not deteriorate the performance, but can help SpMM to fully leveraging the memory resource. Once the batch size of weight matrices is decided in initialization stage, the corresponding memory buffer will be alloced. The batched input is executed sequentially.

Meanwhile, during the SpDNN iteration, we design the online memory tuning scheme to adjust the weight batch size dynamically to maintain the communication-computation overlap. The tuning scheme is invoked only when the SpMM execution fetch the last cached layer in one buffer for weight matrices and it needs update, in order to reduce additional overhead. If called at the start of layer l, it collects the T_l and the ratio between nonzero input rows at two iterations, and then generates the new weight buffer length following the same approach in (2b) by estimating T_{l-1}. Since the estimation process is only through simple equations, the online tuning will only incur negligible overhead and not cause performance degradation. Besides, as the rank of result matrices Y_{l+1} cannot be larger than rank of Y_l and W, the batch of input will not increase, thereby preventing memory overflow during online tuning stage. Once the batch size of weight matrices is decided in the online stage, the buffer update process starts and the corresponding layers will be transferred asynchronously.

B. Tensor Core/CUDA Core Fusion Mechanism

To fully exploit the heterogeneous computing resources in the GPUs, we implement the Tensor core/CUDA core fusion mechanism utilizing concurrency profiling results and data parallelism. As the profiling for hetergeneous core concurrency can be conducted during the training process, it will not lead to additional overhead in inference procedure. We mainly focus on splitting the dense blocks in the transformed weight matrices, which takes most of the place in weights and will be originally accelerated on Tensor cores, since on the resource-constrained GPUs the capability and quantity of Tensor cores are lower than that of high-end GPUs. Furthermore, the CUDA cores natively support single precision, while computing data in single precision on Turing architecutre Tensor cores requires time-consuming type convertion.

We provide the illustration of the mechanism for one layer in SpMM in Fig 4. After the profiling process, we obtain the two-stage concurrency model as shown in Fig 2. In order to execute the fusion core in concurrent stage, it is of great importance to ensure the load ratio is below the turning point. As the load ratio is proportional to the computing task load ratio on the two types of cores, we record the maximum computing task load ratio P_M at the turning point. P_M is utilized as the guideline for weight matrix partition. Before the SpMM execution, we split the BCSR blocks in the dense

parts of weight matrices with the maximum split ratio P_M. The BCSR blocks are divided along column dimension to avoid writing conflict. We implement the Tensor core warps and CUDA core warps in the same thread block to enable concurrency, and they share the same input row blocksize for alignment. To improve parallelism, the input data is divided in the block size of Y_{TB} and Y_{CB} for Tensor core warps and CUDA core warps, while the weight matrices are also partitioned in the block size of W_{TB} and W_{CB}. What's more, to improve data reuse, the input Y_{TB} and weight W_{CB} are cached in shared memory. The number of Tensor core warps and CUDA core warps inside a thread block is fixed. Each CUDA core thread is responsible for updating at least one element in the result matrix to avoid writing conflict. In cases where the computation load in Tensor core is too small, and the number of BCSR blocks assigned to CUDA core cannot meet the requirement, we switch the computing pattern to Tensor core solo-running to avoid load imbalance.

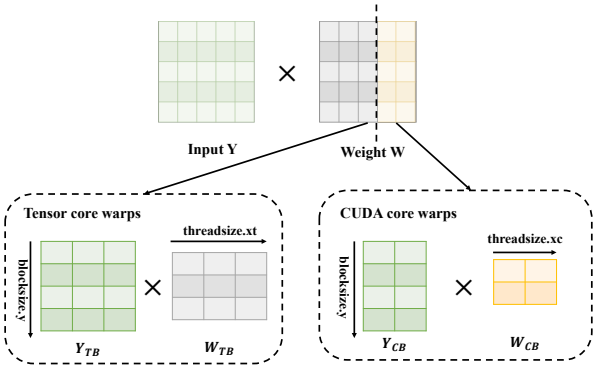

Fig. 4: Illustration of Tensor core/CUDA core fusion mechanism.

C. Other Optimizations

In addition to the adaptive memory aware data partition scheme and Tensor core/CUDA core fusion mechanism, we also implement several relatively general optimization techniques to further improve the performance of SpMM on resource constrained GPU architecture.

Parallel Granularity Tuning. We tune the data granularity for both of the input and weight matrices based on two kinds of memory access patterns: access patterns in NVIDIA WMMA data loading APIs for Tensor cores, and the global memory access pattern of GPU threads. In terms of partitioning the input matrices for fused GPU blocks in size of $blocksize.y$, the larger $blocksize.y$ results in larger stride when the WMMA APIs load input row datum from shared memory, impacting the efficiency; whereas the smaller $blocksize.y$ reduces data reuse and causes warp divergence in thread blocks, since the threads within the same warp may need to fetch inconsistent data of input stored in column major. Meanwhile, the larger warp size of Tensor core can cause resource oversubscription, while the smaller warp size of Tensor core cannot fully leverage the streaming processors. We make a compromise and empirically tune the $blocksize.y$ to be 64 and warp size of Tensor core to be 4.

Hash Table Fusion. After the similarity-based matrix transformation [14], two transformation hash tables p_l and q_l are generated, which indicate the map of rows and columns between original weight matrices and transformed matrices, respectively. The matrix transformation equals two elementary transformations which can be denoted as $W'_l = P_l W_l Q_l$, where P_l and Q_l are the transformation matrices derived from p_l and q_l. Therefore, we define a fusion matrix as $F_l = Q_{l-1}^T P_l^T$ and the SpMM iteration is converted to $Y'_{l+1} = Y'_l F_l P_l W_l Q_l = Y_{l+1} Q_l$, which only has column permutation and will not affect final accuracy. The fusion matrices are generated during training process and it can reduce memory consumption efficiently.

IV. Evaluation

A. Experimental Setup

We evaluate the performance of our implementation on all datasets officially provided by SpDNN Challenge track. The experiment platform is a server with four 20-core 2.4 GHz Intel Xeon Gold 6148 CPUs, four NVIDIA Turing T4 GPUs and 256 GB host memory. Each T4 GPU has 16 GB global memory, 320 Tensor Cores and 2560 CUDA Cores. The host compiler for the SpDNN programs is GNU GCC-8.1.0, the device compiler is CUDA NVCC-11.6, and the device driver version is 530.30.02. We compare our approach with previous champions, including 2022 champion S&Z [14], 2021 champion X&Y [11], 2020 champions H&P [12] and SNIG [13] according to throughput and execution time. Since 2022 champion S&Z is not open-sourced, we discreetly implement the source code following the instruction in [14]. Moreover, another 2022 champion X&W [16] is not included, as S&Z has better average performance reported in their paper. The throughput is calculated through dividing input edges by inference time. If the comparison libraries confront memory overflow and failed at a certain dataset, the corresponding results are dentoed as '-'.

B. Performance Comparison

1) Single GPU: From Table II, we can conclude that our implementation obtain the best performance among all official datasets. We select the 2022 champion S&Z as the baseline. The highest acceleration rate is 1.35× at the dataset with neuron size 1024 and layer length 480, while the average acceleration rate is 1.26×. What's more, our implementation on a single T4 GPU can closely match the performance of one of the cutting-edge champion H&P on a single V100 GPU according to their results reported in [12]. There are two reasons for performance improvement: on one hand, we leverage the concurrency between Tensor core and CUDA core through hetergeneous core fusion; on the other hand, we tune the data block size within the Tensor core warps to balance between different memory access pattern. Meanwhile, our implementation is the only one that can be directly executed on every dataset without manual tuning, while the others confront

TABLE II: Throughput (TeraEdges/Second) and Speedup Comparison on A Single T4 GPU

		Our Work		S&Z [14]		X&Y [11]		H&P [12]		SNIG [13]	
Neurons	**Layers**	**Throughput**	**Time**	**Throughput**	**Speedup**	**Throughput**	**Speedup**	**Throughput**	**Speedup**	**Throughput**	**Speedup**
1024	120	9.85	0.02s	7.62	1.29×	4.377	2.25×	2.27	4.33×	0.33	30.17×
	480	17.34	0.05s	12.82	1.35×	5.78	3.0×	3.33	5.2×	0.41	42.02×
	1920	17.59	0.20s	16.67	1.06×	7.04	2.5×	4.74	3.72×	0.41	42.87×
4096	120	9.81	0.10s	7.35	1.33×	7.35	1.33×	2.71	3.61×	0.43	22.96×
	480	21.0	0.18s	16.65	1.26×	11.28	1.86×	3.61	5.82×	0.5	41.93×
	1920	26.59	0.56s	19.74	1.34×	13.12	2.03×	4.02	6.62×	0.51	51.99×
16384	120	10.76	0.35s	9.88	1.09×	6.64	1.62×	1.65	6.52×	0.54	19.88×
	480	18.12	0.83s	15.24	1.19×	10.18	1.78×	2.03	8.91×	0.62	29.31×
	1920	21.33	2.83s	17.47	1.22×	11.81	1.81×	2.12	10.04×	0.64	33.38×
65536	120	9.07	1.66s	-	-	-	-	-	-	-	-
	480	18.38	3.29s	-	-	-	-	-	-	-	-
	1920	23.42	10.31s	-	-	-	-	-	-	-	-

TABLE III: Scalability and Speedup Comparison on Multiple T4 GPUs in Throughput (TeraEdges/Second)

		Our Work			S&Z [14]			X&Y [11]			H&P [12]		
Neurons	**Layers**	**1**	**2**	**4**	**GPUs**	**Throughput**	**Speedup**	**GPUs**	**Throughput**	**Speedup**	**GPUs**	**Throughput**	**Speedup**
1024	120	9.85	15.76	19.7	4	15.76	1.25×	4	6.39	3.08×	4	5.252	3.75×
	480	17.3	19.92	26.75	4	20.81	1.28×	4	7.87	3.4×	4	6.41	4.17×
	1920	17.59	24.95	28.84	4	23.93	1.20×	4	7.68	3.75×	4	7.31	3.94×
4096	120	9.81	20.92	30.37	4	18.82	1.39×	4	10.12	3.0×	4	4.53	6.71×
	480	21.0	32.31	42.0	4	31.5	1.33×	4	14.0	3.0×	4	6.86	6.12×
	1920	26.59	45.36	68.04	4	54.23	1.25×	4	18.46	3.69×	4	7.22	9.42×
16384	120	10.76	17.9	32.01	4	24.85	1.29×	4	13.59	2.36×	4	5.46	5.86×
	480	18.12	27.24	48.69	4	42.28	1.15×	4	19.06	2.55×	4	6.4	7.61×
	1920	21.33	32.24	59.15	4	53.21	1.11×	4	19.31	3.06×	4	6.69	8.84×
65536	120	9.07	16.19	31.97	4	30.36	1.05×	4	17.3	1.85×	4	4.06	7.88×
	480	18.38	24.52	49.74	4	43.85	1.13×	4	16.89	2.94×	4	4.6	10.81×
	1920	23.42	28.53	53.69	4	50.33	1.07×	4	-	-	4	-	-

memory overflow at weight size 65536. Our implementation automatically avoid global memory oversubscription through adopting adaptive data partitioning scheme, which can decide the batch size of both input and weight matrices based on architecture feature.

2) Multiple GPUs: The results of scalability and performance comparison between our implementation and previous champions on multiple GPUs are depicted in Table III. We still select the S&Z as the baseline. It is obvious that our implementation has the best performance compared to previous champions at 4 T4 GPUs, with the highest speedup of 1.39× and average acceleration rate of 1.20×. Besides, our work can reach the similar or even better throughput with H&P on up to 6 V100 GPUs (compared to the throughput reported in their paper [12]). Moreover, the throughput of our implementation grows as the number of GPU increases from 1 GPU, 2GPUs to 4 GPUs, which demonstrates its scalability. Furthermore, only our implementation and S&Z can be executed on all datasets with 4 GPUs, while X&Y and H&P still fail for memory overflow with the largest dataset, which proves the effectiveness of the memory aware data partition scheme.

V. Conclusion

In this paper, we exploit the memory and computing resources to accelerate the SpDNNs on cost efficient but resource constrained low-end GPUs. We develop the adaptive memory aware data partition scheme to automatically tune the batch size for both input and weight matrices, by solving concise linear programming problem, which can fully utilize the global memory and avoid memory overflow. Moreover, we adopt the Tensor core/CUDA core fusion mechanism leaveraging the concurrency between heterogeneous computing resources on GPUs. We compare our implementation with the state-of-art previous Graph Challenge champions on single and multiple Tesla T4 GPUs, and the results demonstrate that our implementation achieves the best performance with the highest acceleration rate at 1.35 × and 1.39× compared to 2022 champion S&Z on single and multiple GPUs respectively, and it can catch up the performance of H&P at up to 6 V100 GPUs with only 4 T4 GPUs, which enhances the availability and efficiency of large-scale SpDNN to individuals and small research groups.

VI. Acknowledgement

This work was supported by National Key Research and Development Program (Grant No. 2022YFB4501404), the Beijing Natural Science Foundation (4232036), CAS Project for Youth Innovation Promotion Association. The corresponding author is Huawei Cao.

REFERENCES

[1] M. Gharbi, J. Chen, J. T. Barron, S. W. Hasinoff, and F. Durand, "Deep bilateral learning for real-time image enhancement," *ACM Transactions on Graphics (TOG)*, vol. 36, no. 4, pp. 1–12, 2017.

[2] G. Li, Y. Yang, X. Qu, D. Cao, and K. Li, "A deep learning based image enhancement approach for autonomous driving at night," *Knowledge-Based Systems*, vol. 213, p. 106617, 2021.

[3] X. Liu, P. He, W. Chen, and J. Gao, "Multi-task deep neural networks for natural language understanding," in *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, 2019, pp. 4487–4496.

[4] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[5] B. Wu, F. Iandola, P. H. Jin, and K. Keutzer, "Squeezedet: Unified, small, low power fully convolutional neural networks for real-time object detection for autonomous driving," in *Proceedings of the IEEE conference on computer vision and pattern recognition workshops*, 2017, pp. 129–137.

[6] Y. Tian, K. Pei, S. Jana, and B. Ray, "Deeptest: Automated testing of deep-neural-network-driven autonomous cars," in *Proceedings of the 40th international conference on software engineering*, 2018, pp. 303–314.

[7] R. Dale, "Gpt-3: What's it good for?" *Natural Language Engineering*, vol. 27, no. 1, pp. 113–118, 2021.

[8] I. Tenney, D. Das, and E. Pavlick, "Bert rediscovers the classical nlp pipeline," in *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, 2019, pp. 4593–4601.

[9] J. Kepner, S. Alford, V. Gadepally, M. Jones, L. Milechin, R. Robinett, and S. Samsi, "Sparse deep neural network graph challenge," in *2019 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2019, pp. 1–7.

[10] J. Kepner and R. Robinett, "Radix-net: Structured sparse matrices for deep neural networks," in *2019 IEEE international parallel and distributed processing symposium workshops (IPDPSW)*. IEEE, 2019, pp. 268–274.

[11] J. Xin, X. Ye, L. Zheng, Q. Wang, Y. Huang, P. Yao, L. Yu, X. Liao, and H. Jin, "Fast sparse deep neural network inference with flexible spmm optimization space exploration," in *2021 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2021, pp. 1–7.

[12] M. Hidayetoğlu, C. Pearson, V. S. Mailthody, E. Ebrahimi, J. Xiong, R. Nagi, and W.-m. Hwu, "At-scale sparse deep neural network inference with efficient gpu implementation," in *2020 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2020, pp. 1–7.

[13] D.-L. Lin and T.-W. Huang, "A novel inference algorithm for large sparse neural network using task graph parallelism," in *2020 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2020, pp. 1–7.

[14] Y. Sun, L. Zheng, Q. Wang, X. Ye, Y. Huang, P. Yao, X. Liao, and H. Jin, "Accelerating sparse deep neural network inference using gpu tensor cores," in *2022 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2022, pp. 1–7.

[15] C. A. Navarro, R. Carrasco, R. J. Barrientos, J. A. Riquelme, and R. Vega, "Gpu tensor cores for fast arithmetic reductions," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 1, pp. 72–84, 2020.

[16] S. Xu, M. Wu, L. Zheng, Z. Shao, X. Ye, X. Liao, and H. Jin, "Towards fast gpu-based sparse dnn inference: A hybrid compute model," in *2022 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2022, pp. 1–7.

[17] J. Ren, S. Rajbhandari, R. Y. Aminabadi, O. Ruwase, S. Yang, M. Zhang, D. Li, and Y. He, "Zero-offload: Democratizing billion-scale model training." in *USENIX Annual Technical Conference*, 2021, pp. 551–564.

[18] L. Wang, J. Ye, Y. Zhao, W. Wu, A. Li, S. L. Song, Z. Xu, and T. Kraska, "Superneurons: Dynamic gpu memory management for training deep neural networks," in *Proceedings of the 23rd ACM SIGPLAN symposium on principles and practice of parallel programming*, 2018, pp. 41–53.

[19] C. J. Shallue, J. Lee, J. Antognini, J. Sohl-Dickstein, R. Frostig, and G. E. Dahl, "Measuring the effects of data parallelism on neural network training," *Journal of Machine Learning Research*, vol. 20, pp. 1–49, 2019.

[20] H. Zhao, W. Cui, Q. Chen, Y. Zhang, Y. Lu, C. Li, J. Leng, and M. Guo, "Tacker: Tensor-cuda core kernel fusion for improving the gpu utilization while ensuring qos," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2022, pp. 800–813.

[21] L. Deng, "The mnist database of handwritten digit images for machine learning research [best of the web]," *IEEE signal processing magazine*, vol. 29, no. 6, pp. 141–142, 2012.

[22] NVIDIA, "Nvidia turing architecture whitepaper," 2019. [Online]. Available: https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turingarchitecture/NVIDIA-Turing-Architecture-Whitepaper.pdf

[23] A. Dakkak, C. Li, J. Xiong, I. Gelado, and W.-m. Hwu, "Accelerating reduction and scan using tensor core units," in *Proceedings of the ACM International Conference on Supercomputing*, 2019, pp. 46–57.

[24] A. Gholami, A. Azad, P. Jin, K. Keutzer, and A. Buluc, "Integrated model, batch, and domain parallelism in training neural networks," in *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, 2018, pp. 77–86.

[25] G. Dai, G. Huang, S. Yang, Z. Yu, H. Zhang, Y. Ding, Y. Xie, H. Yang, and Y. Wang, "Heuristic adaptability to input dynamics for spmm on gpus," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, 2022, pp. 595–600.

[26] C. Hong, A. Sukumaran-Rajam, I. Nisa, K. Singh, and P. Sadayappan, "Adaptive sparse tiling for sparse matrix multiplication," in *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, 2019, pp. 300–314.

[27] Z. Chen, Z. Qu, L. Liu, Y. Ding, and Y. Xie, "Efficient tensor core-based gpu kernels for structured sparsity under reduced precision," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021, pp. 1–14.

[28] E. D. Andersen and K. D. Andersen, "Presolving in linear programming," *Mathematical Programming*, vol. 71, pp. 221–245, 1995.