Efficient Tensor Core-Based GPU Kernels for Structured Sparsity under Reduced Precision

Zhaodong Chen[*]
chenzd15thu@ucsb.edu
University of California, Santa Barbara

Zheng Qu[*]
zhengqu@ucsb.edu
University of California, Santa Barbara

Liu Liu
liu_liu@ucsb.edu
University of California, Santa Barbara

Yufei Ding
yufeiding@ucsb.edu
University of California, Santa Barbara

Yuan Xie
yuanxie@ucsb.edu
University of California, Santa Barbara

ABSTRACT

The success of DNN comes at the expense of excessive memory/-computation cost, which can be addressed by exploiting reduced precision and sparsity jointly. Existing sparse GPU kernels, however, fail to achieve practical speedup over cuBLASHgemm under half-precision. Those for fine-grained sparsity suffer from low data reuse, and others for coarse-grained sparsity are limited by the wrestling between kernel performance and model quality under different grain sizes. We propose **column-vector-sparse-encoding** that has a smaller grain size under the same reuse rate compared with block sparsity. Column-vector-sparse-encoding can be applied to both SpMM & SDDMM, two major sparse DNN operations. We also introduce the **Tensor-Core-based 1D Octet Tiling** that has efficient memory access and computation patterns under small grain size. Based on these, we design SpMM and SDDMM kernels and achieve 1.71-7.19x speedup over cuSPARSE. Practical speedup is achieved over cuBLASHgemm under >70% and >90% sparsity with 4x1 grain size and half-precision.

KEYWORDS

Neural Networks, Sparse Matrices, GPGPU, Tensor Core

1 INTRODUCTION

Areas like Computer Vision and Natural Language processing have witnessed rapid development in recent years driven by the powerful deep neural networks. However, the achievements come at the expense of enormous memory footprint and computation consumption. To mitigate this issue, reduced precision and sparsity are usually exploited jointly [9, 13, 19, 27]. The former uses fewer bits for each operands, and specialized units like Tensor Core are introduced to improve the computation throughput. The latter explores the redundancy in neural network models to reduce the total number of operands and necessary computations. There are two most commonly used sparse operations in deep neural networks, sparse matrix-matrix multiplication (SpMM) and sampled dense-dense matrix multiplication (SDDMM) [6]. High-performance GPU kernels have been developed for these two operations. For instance, Gale et al. [6] proposed GPU kernels for fine-grained sparsity and achieve considerable speedup over dense GEMM kernel under single precision. cuSPARSE [17] also presents a set of APIs like *cusparseSpMM* and *cusparseSDDMM* that target on > 95% sparsity.

However, to the best of our knowledge, existing implementations are insufficient for achieving practical speedup over their dense counterparts when reduced precision (e.g., FP16) is used. With detailed profiling (Section 3.1), we find that the reasons are of three folds: (1) After using reduced precision, fast memory can cache more operands to improve data reuse, which is fully exploited by dense GEMM kernels. Whereas, fine-grained sparse kernels with much lower data reuse fail to exploit this benefit. (2) Although structured sparsity can be introduced to improve data reuse, existing libraries like cuSPARSE cannot deliver practical speedup with small sparsity granularity: e.g., the blocked-ELL format based SpMM kernel [17] requires block size to be larger than 8 to achieve speedup (Section 3.2). (3) As larger block size is desired, more challenges are brought to the algorithm side to maintain the model accuracy [15].

To address these challenges, we first propose the column vector sparse encoding in Section 4. It is inspired by the commonly used compressed sparse row (CSR) encoding, except that each index now corresponds to a nonzero column vector. We prove that under grain size $V\times 1$, our column encoding has the same data reuse rate as the $V\times V$ block sparsity in both SpMM and SDDMM operations. Moreover, the $V\times 1$ column vector has much smaller grain size compared with the block sparsity pattern, which can better maintain the accuracy under the same sparsity level [15]. Notably, it can also be used for block sparse matrices with arbitrary number of columns in the block by encoding each column vector separately.

To improve the performance of structured sparse kernel under small grain size, we profile the blocked-ELL based SpMM kernel in cuSPARSE using small block size in Section 3.2. We observe that its performance is limited by the instruction cache capacity, dependency between instructions, and shared memory bandwidth. With these observations along with the best practices guide for kernel design, we propose **five key guidelines** for designing an

[*]Both authors contributed equally to this research.

SC '21, November 14–19, 2021, America's Center, St. Louis, MO
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8442-1/21/11...$15.00
https://doi.org/10.1145/3458817.3476182

structured sparse kernel with efficient memory access and computation pattern. As existing implementations using Floating-point Unit (FPU) or Tensor Core Unit (TCU) fail to achieve these guidelines at the same time, we propose a novel mapping between the warp tile and the TCU, namely TCU-based 1-D Octet Tiling (Section 5 and 6), that satisfies all the five guidelines simultaneously.

We extensively evaluate our kernels on the sparse matrices from DLMC [22] dataset in Section 7 using different grain size, problem size, and sparsity ratio. In short, for SpMM, we achieve **1.71-7.19x** geometric mean speedup over the Blocked-ELL SpMM kernel from cuSPARSE and **1.34-4.51x** geometric mean speedup over a FPU-based kernel that we directly extended from Sputnik [6]. For SDDMM, we achieve **1.27-3.03x** speedup over the FPU-based kernel extended from Sputnik [6] and **0.93-1.44x** speedup over the TCU-based kernel that uses the classic mapping between GEMM-like warp tile and TCU. Compared with the cuBLASHgemm, our SpMM and SDDMM kernel achieve practical speedup under > **70%** and > **90%** sparsity with the tiny 4 × 1 grain size. Benefited from our design, we achieve 1.41x end-to-end speedup and 13.37x peak memory reduction on the sparse transformer inference task.

2 BACKGROUND

We first introduce some background knowledge and related studies.

2.1 Graphic Processing Units Background

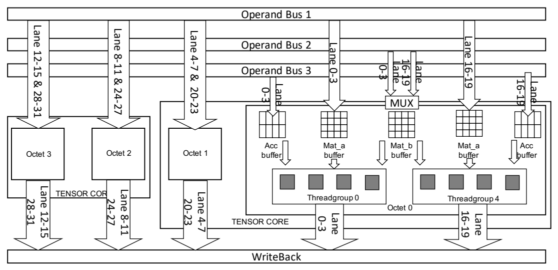
We first provide a basic description of the NVIDIA GPU architecture.

Thread Hierarchy. The threads in a GPU kernel are organized into *grid* of *CTAs* (thread blocks). We use the term *grid size* to refer the total number of CTAs. Each CTA may contain up to 1024 threads. Consecutive 32 threads in a CTA are grouped into *warps*, and consecutive 4 threads in a warp is called a *thread group*, the thread group id of a thread is $\lfloor \frac{threadIdx\%32}{4} \rfloor$. Furthermore, thread group i∈ {0,1,2,3} and thread group i+4 together form the *Octet* i. We call thread group i and i+4 low group and high group, respectively.

Memory Hierarchy[11]. NVIDIA GPU consists of an array of streaming multiprocessors (SMs). In Volta, all the SMs share a 6 MiB L2 Cache and a 16 GiB DRAM. Each SM has a set of sub-cores where each sub-core has its own register file, scheduler, and execution units. All the sub-cores within the same SM share a private 128 KiB L1 cache, part of which can be configured as the shared memory. To maximize the memory bandwidth utilization, vector memory operations using *LDG.128* can be used to increase the Sectors per Request to L1 Cache. The 128 means each thread reads 128 bits from the global memory (e.g. 8 operands under the half precision). Besides, the memory access pattern is expected to be 128B coalesced to exploit the 128B transaction between L1 and L2 caches.

TCU. Volta architecture first introduces the Tensor Core Unit [18] that provides 8× peak FLOPs than FPU. While the Volta TCU focuses on accelerating dense GEMM, its high computation throughput is also exploited for non-GEMM applications like reduction[5].

Figure 1 shows the Volta TCU architecture. A warp uses two TCUs at the same time. Each TCU is controlled by two octets. Each thread group in the octet has its own buffer for storing the LHS matrix (Mat_a buffer) and accumulation result (Acc buffer), and each thread can access a four-by-four inner product unit (gray blocks). The RHS matrix buffer (Mat_b buffer) is shared by the two thread groups within each Octet. Its source is selected by a multiplexer.



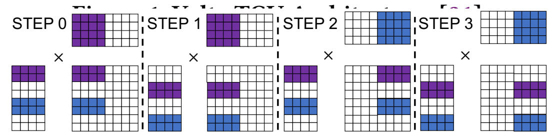


Figure 2: Visualization of the 4 steps in *mma.m8n8k4*. Purple blocks: low group; Blue blocks: high group.

CUDA provides two levels of APIs for TCU. Firstly, warp-level matrix multiply and accumulate (WMMA) in C++ performs a dense matrix multiplication with a warp. For instance, *wmma.m8n32k16* computes a (8×16)·(16×32) GEMM with a warp. Secondly, matrix multiply and accumulate (MMA) in PTX performs 4 dense matrix multiplications with a warp, one for each Octet. For instance, *mma.m8n8k4* completes four (8×4)·(4×8) matrix multiplications. During the compilation, the *mma.m8n8k4* is further decomposed into four *HMMA* instructions: *HMMA.884.F32.F32.STEP{0,1,2,3}* in Figure 2. In each step, every thread group loads its LHS tile to its Mat_a buffer and the input accumulation values to its Acc_buffer. The multiplexer for RHS tile selects the low group in step 0&1, and high group in step 2&3.

2.2 Compression in Deep Learning

To reduce the computational and memory intensity of large neural networks, two types of methods are usually exploited [9]. The first one, Quantization, reduces the number of of bit that represents each operand from 32 to 16 or even lower. The second one exploits the sparsity in the neural networks. For example, we can sparsify the dense matrices used in matrix multiplication and convolution while maintaining comparable model quality [4, 9, 10, 16]. We can also directly formulate the computation with sparse operation when the sparsity already exists. For instance, the forward propagation of Graph Convolutional Neural Networks (GCNs) naturally adopts sparsity in the graph adjacent matrix [3, 7].

In general, we can characterize the most commonly used sparse operations in deep learning models into two categories. The first one is sparse matrix-matrix multiplication (SpMM). For example, in weight pruning methods, we apply specific sparsity constraints to produce an NN model where the weight matrices contain relatively high portion of zero values [9, 16]. The second type of sparse operation is sampled dense-dense matrix multiplication (SDDMM), where the sparsity locates in the output matrix of the equation to help reduce the required computations. For example, prior work [2, 14] have proposed different mechanisms to efficiently approximate and predict the zero values in the output feature map of CNNs and RNNs so that to skip these computations during execution. More recently, in the study of transformers, output sparsity helps to reduce the computation overhead of self-attention layers, which is particularly beneficial for applications like long sequence modeling [4, 23].

2.3 Existing Sparse Kernels on GPU

To further exploit the benefit of sparsity on parallel architectures like GPUs, efforts have been made on both algorithm and hardware level. As shown in Figure 3, apart from fine-grained sparsity, structures like 1D-vector [31] and 2D-block [29] have been enforced to the topology of the sparse matrix to benefit architecture and kernel design. Corresponding GPU kernels for SpMM and SDDMM are also proposed to deliver practical speedup. NVIDIA introduces the cuSPARSE library that targets on 95% or higher sparsity and provides the *cusparseSpMM* and *cusparseSDDMM* APIs. The former one supports half, single, or higher precision, and the sparse matrix can be either fine-grained sparsity or Blocked-ELL format. The latter one only supports fine-grained sparsity with single or higher precision. Gale et al. [6] introduce a library called Sputnik that targets on fine-grained sparsity and outperforms cuSPARSE under relatively low sparsity. E.g., Sputnik achieves speedup over the dense baseline under > 71% sparsity with single precision.

3 OPPORTUNITIES AND CHALLENGES

In this section, we identify whether prior sparse kernels are sufficient for exploiting the sparsity plus quantization combo. Specifically, we evaluate the speedup achieved over cuBLAS by Sputnik and cuSPARSE under single and half precision. Similar to Sputnik [6], we use the sparse matrices from ResNet-50 with magnitude pruning in the DLMC dataset [22].

3.1 Half Precision Fine-grained Sparse Kernel

As shown in Figure 4, under fine-grained sparsity and single precision, both Sputnik and cuSPARSE achieve good speedup when sparsity is greater than 80% [1]. However, when it comes to half precision, the SpMM kernel in Sputnik only outperforms cublasHgemm under extremely high sparsity, and cuSAPRSE has a even lower performance. Moreover, as both Sputnik and cuSPARSE do not support SDDMM under half precision, we modified the source code of Sputink and find that it is also inferior to cublasHgemm.

To identify the reason behind this, we further profile the Sputnik's SpMM kernel on $\boldsymbol{A}_{2048\times1024} \times \boldsymbol{B}_{1024\times256}$, the sparsity of $\boldsymbol{A}$ is 90%. Under half precision, as shown in Figure 5, the number of missed sectors in L1 cache is reduced by 77.04% in GEMM kernel, whereas that of SpMM is only 48.77%. Thus, the latter one needs to load more sectors from L2 cache than the former one. This is because in GEMM kernel, the cached operands show good data reuse pattern. This benefit is further enhanced when the precision is reduced to 16 bits, as two times as much as operands can be stored in the shared memory. From another perspective, Kwasniewski et al. [12] give the I/O lower bound of GEMM to be $Q = b\frac{2mnk}{p\sqrt{S/b}}$, where b is the number of bits of each operand, m, n, k are the matrix dimensions, p is the number of processors, and S is the fast memory size. Therefore, we have $Q_H \approx 0.35Q_S$ for single and half precision

[1]The SDDMM in cuSPARSE is faster than Sputnik [6] as we use the cuSPARSE v11.2.2.

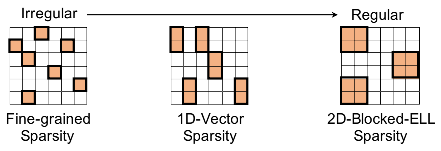


Figure 3: Different sparse structures.

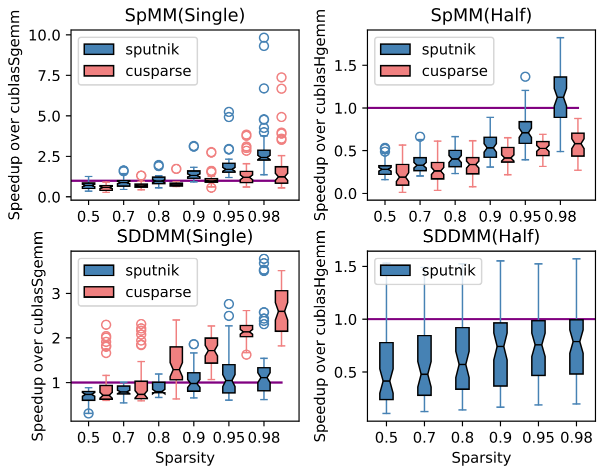


Figure 4: Speedup over cuBLAS with fine-grained sparsity.

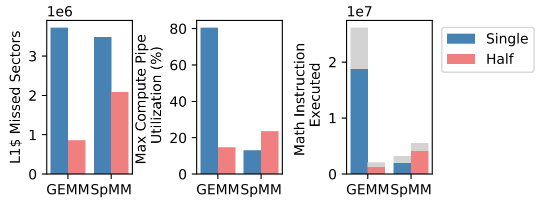


Figure 5: GEMM and SpMM under different precision.

GEMM kernel. On the contrary, the SpMM kernel has lower data reuse. As a result, even though the fast memory is "enlarged" for operands storage in half precision, the actual benefit delivered on cache miss rate is limited, causing the gap shown in Figure 5.

Another portion of performance speedup of HGEMM comes from the use of TCU. Figure 5 shows that the maximum compute pipeline utilization of GEMM is reduced from 88.44% (FMA) to 14.6% (Tensor), which suggests that the compute bound is addressed by TCU. Besides, multiple FMA instructions are fused into a single HMMA instruction, which removes 92.3% instructions. On the contrary, Sputnik still uses the FPU and additional instructions to convert result to single precision to reduce accumulation error.

3.2 Half Precision Structured Sparse Kernel

To further improve the performance of sparse computation, cuSPARSE v11.2.1 introduces the Blocked-ELL format to speedup its SpMM kernel. With block sparsity, the kernel not only improves the data reuse rate but also utilizes the TCUs.

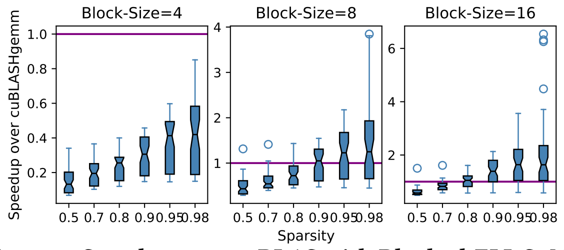


Figure 6: Speedup over cuBLAS with Blocked-ELL SpMM.

However, as shown in Figure 6, the Blocked-ELL based SpMM has low performance when block size is smaller than 8. Also, previous studies [15] show that the model suffers from accuracy degradation under the same sparsity with larger block size. As a consequence, the design space is significantly limited due to the wrestling between kernel performance and model quality. To dissect the inefficiency of the Blocked-ELL SpMM kernel under small block size (i.e., 4), we profile it on $\boldsymbol{A}_{2048\times 1024}\times \boldsymbol{B}_{1024\times 256}$ under 90% sparsity and list the top three reasons that cause pipeline stall in Table 1.

The "No Instruction" is usually caused by instruction cache miss. Volta uses one 128-bit word to encode each instruction, and each sub-core has a 12 KiB L0 instruction cache [11]. So the L0 instruction cache can only store 768 instructions. However, we find that when block size is 4, the SASS code has 4600 lines. So the "No Instruction" is majorly caused by L0 instruction cache capacity miss.

The "Wait" happens when warp is stalled waiting on a fixed latency execution dependency. From the instruction statistics we observe that the IMAD (Integer Multiply & Add) and IADD3 (3-input Integer Add) account for 27.4% of total executed instructions. From the SASS code, we interpret that these integer instructions compute the addresses of tiles in the global and shared memory.

The "Short Scoreboard" usually happens when the warp is stalled waiting for loading data from the shared memory. When block size is 4, the $\frac{\#shared\ memory\ load\ requests}{\#global\ load\ requests}$ ratio is 0.87, whereas this ratio for HGEMM is 4.17. This implies that the data loaded into the shared memory by the SpMM kernel do not get many opportunities to be reused compared with dense GEMM kernel. Therefore, it makes less sense to put them into the shared memory. Moreover, to ensure the correctness when shared memory is involved, the synchronization barriers needs to be used, which not only introduces additional barrier stalls, but also makes it difficult to hide latency through interleaving the load and compute instructions. Last but not least, as part of the L1 cache is configured into the shared memory, this actually reduces the implicit data reuse through L1 cache.

With the analysis above and best practices guide for CUDA kernel design, here we propose five key guidelines to be considered during the kernel implementation. Guideline I and II improve the overall kernel performance, III focuses on the computation efficiency, IV and V influence the memory access efficiency.

- **I**. Reduce program size to avoid overflow the instruction cache.
- **II**. Increase the grid size to hide the latency through thread-level-parallelism (TLP).
- **III**. Reduce fixed latency operations through looping unrolling, computing offset and constants at compile time, as well as merging floating point operations to *HMMA* with TCU.
- **IV**. Directly load data with few reuse opportunities to the register file without using the shared memory.
- **V**. Improve bandwidth utilization with 128B coalesced transactions and long vector memory operations (*LDG.128*).

In conclusion, it is more challenging for the sparse kernel to achieve speedup over GEMM under half precision than single precision. Generally, increasing data reuse and exploiting the benefit of TCU through the structured sparsity is a promising way. However, there are still two challenges. First, how to maintain model accuracy under structured sparsity. Second, how to design a kernel that satisfies the five guidelines above simultaneously. In this paper, we address the first challenge by proposing the column vector sparse encoding in Section 4, which has finer granularity under the same data reuse rate as block sparsity. The second challenge is addressed by the novel TCU-based 1-D Octet Tiling in Section 5 and 6.

Table 1: Stall Reasons in Blocked-ELL based SpMM kernel

Block Size	No Instruction	Wait	Short Scoreboard
4	42.6%	21.0%	11.9%

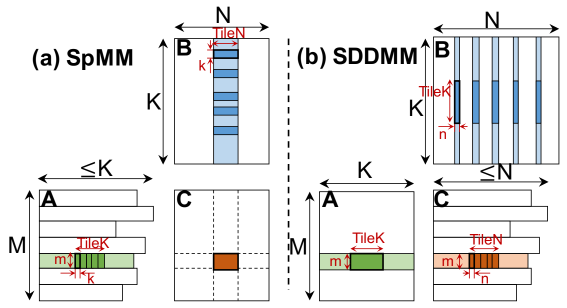


Figure 7: Generalized Block Sparse representation.

4 COLUMN VECTOR SPARSE ENCODING

In this section, we first show that for both SpMM and SDDMM, a block sparse matrix has the same data reuse regardless of the number of columns within the block. Then, based on this observation, we propose the *column vector sparse encoding* that achieves good balance between sparsity granularity and computation efficiency.

4.1 Problem Description

Figure 7 shows the SpMM and SDDMM under block sparsity. The nonzero blocks are aligned in the vertical dimension. The problem size is $(M\times K)\cdot(K\times N)$, $TileK$ and $TileN$ are the tiling sizes constrained by the shared memory or register file capacity. Tensors in mainstream frameworks like PyTorch [20] and TensorFlow [1] are stored in row-major format. Therefore, for SpMM, we store both $\boldsymbol{B}$ and $\boldsymbol{C}$ in row major, and $\boldsymbol{A}$ in compressed sparse row (CSR). For SDDMM, we store $\boldsymbol{C}$ in CSR and $\boldsymbol{A}$ in row major. However, as $\boldsymbol{B}$ is usually a transposed row-major matrix, e.g. in the self-attention layer [26], we store $\boldsymbol{B}$ in column major to replace the transpose.

For SpMM, each block is an $m\times k$ matrix where m and k are user defined sizes for block sparsity. We follow the workflow in Sputnik [6]: each tile takes $TileK/k$ consecutive nonzero blocks in matrix A, and samples a vector with width $TileN$ from the corresponding rows in matrix B. At last, the partial sums in matrix C is computed. For SDDMM, each block in matrix C is an $m\times n$ matrix where m and n are user defined grain sizes. Each tile computes the partial sum of $TileN/n$ consecutive nonzero blocks in matrix C by taking $TileK$ columns of the corresponding rows in matrix A and $TileK$ rows from the corresponding columns in matrix B.

With the above settings, it is obvious that for both SpMM and SDDMM, each operand from the LHS matrix is reused for $TileN$ times, while each RHS operand is reused for m times. Therefore,

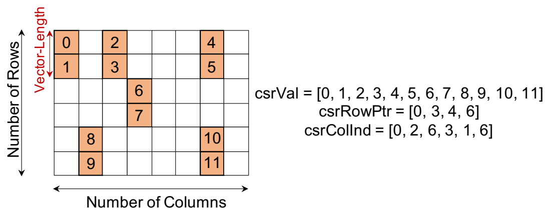


Figure 8: Column Vector Sparse Encoding.

the number of data reuse is determined by m and $TileN$, but not the number of columns (k in SpMM and n in SDDMM) in each block.

4.2 Column Vector Sparse Encoding

As the data reuse is independent of the column number, and smaller granularity is preferred, we propose to reduce the number of columns to 1, which yields the column vector sparse encoding in Figure 8. Our encoding is equivalent with replacing each nonzero scalar in the CSR sparse matrix with a nonzero column vector, i.e. *half2* for $V = 2$, *half4* for $V = 4$, and *float4* for $V = 8$. The elements within each nonzero column vector are stored in consecutive addresses, and the consecutive vectors in the same row are also consecutive in the memory space. Notably, this encoding can also cover the cases of general block sparse matrix by encoding each column separately.

5 SPMM KERNEL DESIGN

In this section, we detail the design of our SpMM kernel with column vector sparse encoding. We first describe two baseline implementations. The first one is an FPU-based design that we directly extended from the Sputnik [6]. This implementation is tailored for high memory access efficiency. The second one uses TCU and adopts the classic GEMM-like tiling, which maximizes the compute and kernel efficiency. At last, we present a more efficient design that covers the above advantages simultaneously.

5.1 FPU-based 1-D Subwarp Tiling

Gale et al. [6] propose the FPU-based 1-D subwarp tiling for fine-grained SpMM kernel that maximizes the memory access efficiency. It is called "1-D subwarp tiling" because under the fine-grained setup (V=1), as illustrated in Figure 9 (a), the LHS operand is a $1{\times}TileK$ 1-D vector handled by a subwarp of threads. Under the column vector sparse encoding, we still call the $(V{\times}TileK){\cdot}(TileK{\times}TileN)$ tile a "1-D tile", as its LHS operand can be regarded as a $1{\times}TileK$ 1-D set of column vectors. As shown in Figure 9 (a), a CTA tile contains multiple independent 1-D tiles that are assigned to subwarps ($Subwarp\ Size \leq 32$). Each 1-D tile is further decomposed to $Subwarp\ Size$ independent $(V{\times}TileK){\cdot}(\frac{TileK{\times}TileN}{Subwarp\ Size})$ thread tiles. The threads in the same subwarp first load the LHS fragment into the shared memory corporately. Then, each thread loads the RHS fragment corresponded to its tile and computes the MMA.

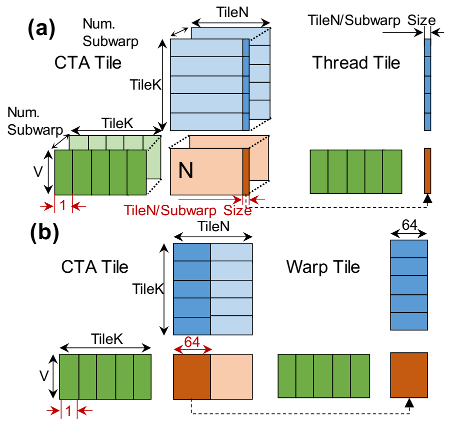


Figure 9: Decomposition of SpMM with 1-D tiling. (a) The FPU tiling extended from Sputnik[6]; (b) Our TCU tiling.

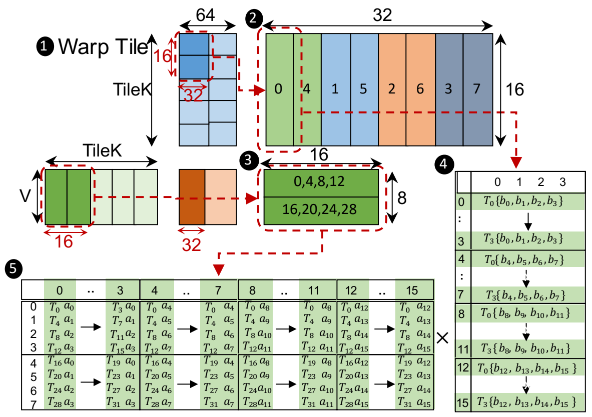


Figure 10: Classic mapping of the warp tile to TCU.

This tiling design opts for maximizing memory access efficiency. First, it satisfies the aforementioned guideline IV as the RHS operands are directly loaded to register file. For guideline V, by choosing $TileN = 64$ and $Subwarp\ Size = 8$, each subwarp can load a row of consecutive 64 half operands from the RHS fragment with the vector memory operation *LDG.128* in a single 128B transaction.

However, the design choice causes low kernel and compute efficiency. For kernel efficiency, it violates guideline I because we need to fully unroll the loops along V, $TileK$, and $TileN$. This generates huge amount of instructions. But without this unrolling, the compiler may not know the index to RHS operands at compile time, and the operands would be put into the local memory (in DRAM) for indexing. For guideline II, we have $\frac{TileN}{Subwarp\ Size} = 8$ with guideline V and $\#Subwarp \geq \frac{32}{Subwarp\ Size}$. Therefore, the grid size is bounded by $\frac{M{\times}N}{V{\times}\#Subwarp{\times}TileN} = \frac{M{\times}N}{256V}$. Oppositely, we can have $\frac{TileN}{Subwarp\ Size} = 2$ and the upper bound improved to $\frac{M{\times}N}{64V}$ only if we give up guideline V and use *LDG.32*. For compute efficiency, it violates guideline III because each thread tile is computed by a sequence of *HMUL* (FP16 Multiply) and *FADD* (FP32 Add).

5.2 TCU-based 1-D Warp Tiling

We present a tiling design that opts for maximizing kernel and compute efficiency. With guideline II, we assign 1-D tiles to CTAs instead of subwarps, such that the upper bound of grid size is $\frac{M{\times}N}{64V}$. Besides, motivated by I&III, we leverage the TCU as it merges multiple *HMUL* and *FADD* into a single *HMMA* instruction. Besides, its fixed computation pattern helps us avoid many index computation. With all these features, we present the TCU-based 1-D Warp Tiling in Figure 9 (b). The 1-D CTA tile is further decomposed into warp-level tiles with size $(V{\times}TileK){\cdot}(TileK{\times}64)$. The 64 is chosen as it is the smallest number that perfectly fills the 128B transaction.

Although this tiling design has high kernel and compute efficiency, its memory access pattern is sub-optimal. Figure 10 illustrates how the warp tile is further decomposed to each thread in the classic way. As our kernel targets on $V \in \{2,4,8\}$, the *wmma.m8n32k16* in CUDA C++ is used to reduce the waste of computation. As we can see in ❷, each small rectangle represents a 16×4 block and the number on it indicates the thread group that holds it. An example for the thread group 0 is illustrated in ❹, where T_i represents thread i and b_j indicates the register j of the

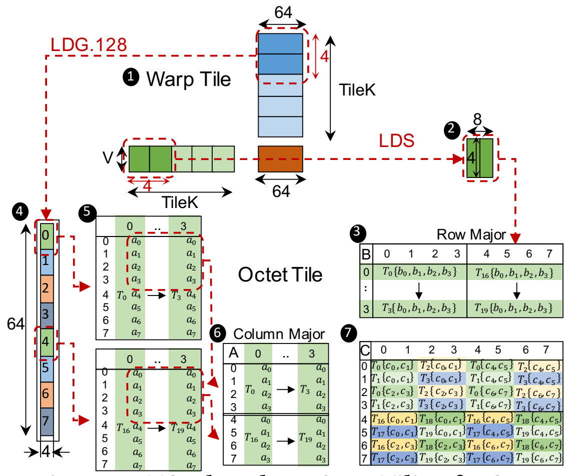


Figure 11: TCU-based 1-D Octet Tiling for SpMM.

thread. In ❸, each block represents a 4×16 block and the number on it indicates the thread groups that hold a copy, the detailed layout is shown in ❺. If we have IV and directly load the RHS fragment (❷) from global memory to register file, as shown in ❹, each thread has 4 registers in each row. So the longest vector memory operation we can use is *LDG.64*, and the global memory access pattern is at most 64B coalesced because each row in ❷ is mapped to 8 threads in a warp. Therefore, guideline V is violated. On the other hand, to achieve guideline V, we need to coalesce the global memory access with the shared memory, which violates guideline IV.

Besides, $TileK$ has to be the multiple of 16, which introduces additional overhead during residue handling when the number of nonzeros in the current row is not divisible by $TileK$. At last, when V is smaller than 8, each computation step with *wmma.m8n32k16* actually computes a $(V\times16)\cdot(16\times32)$ tile, which indicates a portion of wasted computation.

5.3 TCU-based 1-D Octet Tiling

While the performance of the previous two designs are limited by the wrestling between kernel/compute and memory access efficiency, we propose a new tiling technique that is efficient in all three aspects. To achieve good kernel and compute efficiency (guideline I, II, and III), we leverage the CTA and warp tiling in the TCU-based 1-D Warp tiling in Figure 9 (b). The memory access efficiency (guideline IV and V) is achieved by redesigning the mapping between the warp tile and the TCU on Octet level. Therefore, we name it as the TCU-based 1-D Octet Tiling.

The new mapping is visualized in Figure 11. There are two major differences from the classic mapping in Figure 10. First, as shown in Figure 2, STEP 0&1 generate the left four columns in the output while STEP 2&3 produce the rest. This motivates us to put V to the horizontal direction by switching the LHS and RHS fragments, which creates opportunity to skip STEP2&3 when $V \le 4$. Specifically, while ❹ is from the RHS fragment of the warp tile, it is regarded as the LHS fragment for the computation in TCU. Each block in ❹ represents an 8×4 block and the number on it indicates the thread group that loads it. Similarly, ❷ from the LHS fragment of the warp tile is regarded as the RHS fragment in TCU, and each block in ❷ is a 4×4 block. Therefore, as the original warp tile has column-major LHS fragment and row-major RHS fragment, ❷ is in row major and ❹ is in column major. When $V \le 4$, one could remove the STEP 2&3 from the SASS code if an assembler were available. Second, the warp tile is partitioned to octets in a different way to guarantee both efficient computation and memory access. In detail, the warp tile is decomposed to $TileK/4$ steps to be processed in serial by each warp, and each step processes a $(64\times4)\cdot(4\times V)$ subtile (after the switching). With guideline IV, we directly load ❹ to the register file as it has few reuse opportunities. For guideline V, as each column of consecutive 64 half operands in ❹ are mapped to 8 different threads and each thread holds consecutive 8 half operands, ❹ can be loaded with a single *LDG.128* instruction which generates four 128B coalesced global memory transactions. For ❷, as the LHS fragment in the warp tile is reused for many times, we directly load it into the shared memory at the beginning of the tile, so it does not influence the memory access efficiency. Besides, the new mapping only requires $TileK$ to be the multiple of 4, which is more friendly to residual handling.

5.4 Implementation Details

For an SpMM with size $A_{M\times K}\times B_{K\times N}=C_{M\times N}$, we take $TileN=64$ and $CTA\ size=32$ to have as much CTAs as possible while maintaining the best memory access pattern. Therefore, $\lceil M/V\rceil\times\lceil N/64\rceil$ CTAs are launched, each processes an $V\times64$ output tile.

To generate the output tile, each CTA traverses all the nonzero vectors in its row with stride $TileK$, and accumulates the partial sums in the register file. For each stride, all the threads first work jointly to load the LHS fragment in Figure 11 ❶ to shared memory. Then, each thread group loads its share in the 64×4 RHS fragment in Figure 11 ❹. Next, an *mma.m8n8k4* is launched to compute a $(64\times4)\cdot(4\times V)$ matrix multiplication (❻×❸=❼). This is repeated for $TileK/4$ times until the warp tile is done. We observe that the compiler tends to reuse the registers that store the source operand of each *mma.m8n8k4* to reduce register consumption. Whereas, this actually hurts the Instruction Level Parallelism (ILP) as the load and computation of different $(64\times4)\cdot(4\times V)$ tiles will depend on each other if they use the same set of registers. To improve ILP, we first call all the $TileK/4$ load instructions, then insert a *__threadfence_block()*, at last call the $TileK/4$ *mma.m8n8k4* instructions. This prevents the compiler from reusing the registers.

When the last few nonzero vectors cannot fill the $TileK$ width, the load and computation of each $(64\times4)\cdot(4\times V)$ tile will be interleaved until all the nonzero vectors are processed. This helps reduce the residual handling overhead. After all the nonzero vectors are processed, we use the warp shuffle primitives to reorganize the data and write them to DRAM with vector memory operations.

6 SDDMM KERNEL DESIGN

Similar to Section 5, we first describe two baseline SDDMM designs. One is extended from Sputnik [6] that opts for memory access efficiency, the other is based on classic mapping between GEMM and TCU for high compute and kernel efficiency. Then, we present our design that achieves high efficiency in all three aspects.

6.1 FPU-based 1-D Subwarp Tiling

The FPU-based 1-D subwarp tiling is illustrated in Figure 12 (a). Similarly, each CTA tile contains multiple independent 1-D tiles assigned to subwarps. Each 1-D tile is decomposed to $Subwarp\ Size$ independent thread tiles with size $(V\times\frac{TileK}{Subwarp\ Size})\cdot(\frac{TileK}{Subwarp\ Size}\times$

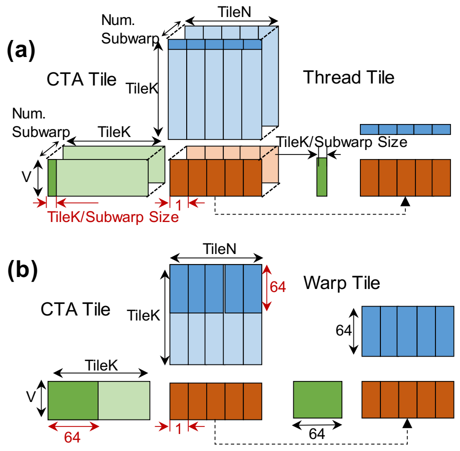


Figure 12: Decomposition of SDDMM with 1-D tiling. (a) The FPU tiling extended from Sputnik [6]; (b) Our TCU tiling.

$TileN$). Each thread loads its LHS and RHS tile into registers and computes partial sums. At last, partial sums stored by different threads in the same subwarp are reduced with warp shuffle.

This tiling design also has high memory access efficiency. Specifically, with $TileK = 64$ and $Subwarp\ Size = 8$, the rows in LHS fragment and columns in RHS fragment of the 1-D tile can be loaded with a single *LDG.128* instruction in the 128B coalesced pattern. Therefore it satisfies guideline IV and V. However, kernel and compute efficiency are sub-optimal due to the same reasons as in the FPU-based SpMM. Moreover, each thread holds a $V \times TileN$ array in the register file to store the partial sums. For instance, when $V = 8$ and $TileN = 32$, the partial sum consumes 256 registers of each thread, which exceeds the register file capacity and causes register spilling. Even without the register spilling, the large amount of registers actually reduces the occupancy.

6.2 TCU-based 1-D Warp Tiling

The TCU-based 1-D Warp Tiling for SDDMM is illustrated in Figure 12 (b). Each CTA tile has only one 1-D tile, which is further decomposed to warp tiles with size $(V \times 64) \cdot (64 \times TileN)$. Similarly, 64 is chosen because it is the smallest number that perfectly fills the 128B transaction. As shown in Figure 13 ❶, the warp tile is further processed with $\frac{TileK \times 64}{32 \times 16}$ steps in serial, and each step is computed with a $wmma.m8n32k16$.

On the positive side, Similar to SpMM kernel, it has high kernel and compute efficiency (guideline I, II, and III) for the same reasons. Besides, it uses fewer registers to store the partial sum. E.g., with

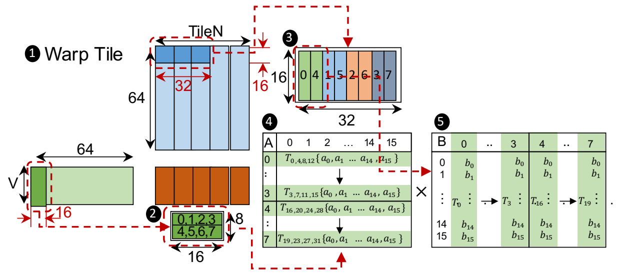


Figure 13: Classic mapping of the warp tile to TCU.

$TileK = 64$, while the FPU-based implementation has $Subwarp\ Size$ copies of the partial sums, this one only has one copy.

On the negative side, it has sub-optimal memory access pattern. Figure 13 ❷ and ❸ visualizes the operand layout for each $wmma.m8n32k16$. In ❷, each block represents a 4×16 block and the numbers on it indicate the thread groups that hold a copy. ❹ gives a detailed data layout for ❷. In ❸, each block represents a 16×4 block and the number on it represents the thread group that holds it. We illustrates the data layout for thread group 0 and 4 in ❺ as an example. If we have guideline IV and directly load the LHS fragment (❷) and RHS fragment (❸) into the register file, we can only have 16B coalesced access. In detail, the 16 operands in each row of the row-major ❷ and column of the column-major ❸ are consecutive and they are mapped to the 16 registers of a thread. However, each $LDG.128$ can only load 8 of them. So it is 16B coalesced. On the other hand, If we have V and coalesce the global memory access with the shared memory, it violates IV.

Furthermore, the LHS fragment ❷ is copied for 4 times, which consumes addition registers that reduces occupancy. Also, $TileN$ has to be a multiple of 32, which introduces additional overhead in residual handling. At last, redundant computations occurred when V is smaller than 8.

6.3 TCU-based 1-D Octet Tiling

Based on the analysis above, we propose our SDDMM kernel that tackles the limitations of the two designs. First, it adopts the same warp tiling as the TCU-based 1-D Warp Tiling in Figure 12 (b), which indicates good kernel and compute efficiency (guideline I, II, and III). To achieve optimal global memory access pattern, we redesign the mapping between warp tile and the TCU in the Octet granularity which we named the TCU-based 1-D Octet Tiling.

The new mapping is shown in Figure 14. There are two major differences from the classic $m8n32k16$ mapping in Figure 13. First, under the same motivation in Section 5.3, we switch the LHS and RHS fragment to expose the opportunity to remove redundant *HMMA* in the SASS code when $V \leq 4$. Second, a novel warp tile partition is applied to guarantee efficient computation and memory access pattern. Specifically, our warp tile is decomposed to $TileN/8$ sub-tiles to be processed sequentially, and the size of each sub-tile is $(8 \times 64) \cdot (64 \times V)$ (after the switch). With guideline IV, the switched LHS and RHS fragments are partitioned and stored in the register file of different thread groups, as shown in ❷ and ❸. Each block in ❷ and ❸ represents a 4×8 or 8×4 matrix held by a single thread group. We take the thread group 0 and 4 as examples and illustrate the detailed data layout in ❹ and ❻. Under this setup, both ❷ and ❸ can be loaded with a *LDG.128* instruction and together generate eight 128B coalesced transactions. In detail, each row vector with length 64 of the row-major ❷ is partitioned to 8 sub-vectors with length 8, and different sub-vectors are loaded by different threads in the warp. This is the same in ❸. Besides, all the operands are only stored by a single thread group, whereas the LHS fragment in the TCU-based 1-D Warp tiling is copied for 4 times.

However, the loaded operands in ❹ and ❻ cannot be directly used for $mma.m8n8k4$, as the register index of each row in ❹ and column in ❻ in thread group i and i+4 do not match. If we compute the index based on the thread group index at runtime, the operands will be moved into the local memory to enable dynamic indexing.

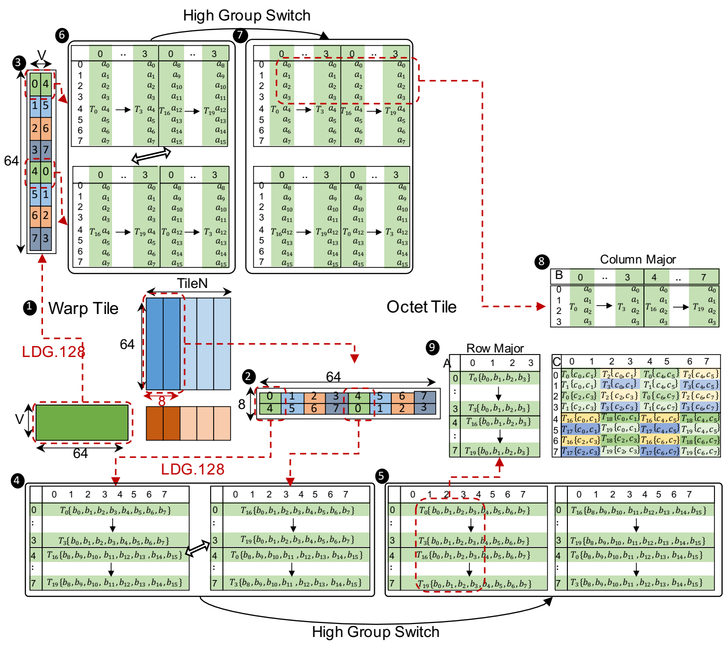


Figure 14: TCU based 1-D Octet Tiling for SDDMM.

To avoid this, we further apply the "High Group Switch", which switch the content in register j and $(j+8)Mod16$ in the thread group 4, 5, 6, and 7 (high group). The data layout after the High Group Switch is illustrated in ❺ and ❼, respectively.

After the High Group Switching, each Octet has an (8×16)·(16×8) tile to compute. While each *mma.m8n8k4* can compute an (8×4)·(4×8) tile, it takes 4 steps to finish the computation. Notably, the upper 4 rows in ❾ and the left 4 columns in ❽ are held by thread group 0 in step 1&2, but they are in thread group 4 in step 3&4. This inverted pattern of source operands also inverts the pattern of the output.

On the algorithm side, the above problem can be solved by either shuffling the operands in thread group i and i+4 with the warp shuffle primitives before calling *mma.m8n8k4*, or using an additional set of registers to accumulate the partial sums from the last two steps. While the shuffle introduces additional overhead, the second solution reduces the occupancy as additional registers are used.

On the hardware side, we propose to extend the original *HMMA* instruction by adding an additional switch Flag that directly switches the source operand in low and high groups within the TCU, i.e. *HMMA.884.F32.F32.STEP{0,1,2,3}.SWITCH*. To support this switch, as shown in Figure 15, we add a pair of multiplexers between the operand bus 1 and the Mat_a buffer of the two thread groups. This pair switches the source of the two Mat_a buffers if switch is set. The source of the Mat_b buffer is switched by XORing the original control signal with the *SWITCH* bit.

6.4 Implementation Details

For an SDDMM with size $\boldsymbol{A}_{M\times K}\times \boldsymbol{B}_{K\times N}\odot \boldsymbol{D}_{M\times N}=\boldsymbol{C}_{M\times N}$, where $\boldsymbol{D}$ is a binary mask stored under the column vector sparse encoding, we take $TileK=64$ and $CTA\ size=32$ to reduce the residual processing overhead while maintaining the best memory access pattern. We heuristically pick $TileN=32$ as it achieves good balance between the data reuse ratio and the number of CTA, but any multiple of 8 is acceptable. Therefore, $\lceil M/V\rceil\times\lceil N/32\rceil$ CTAs will be launched, each processes an $V\times 32$ output tile.

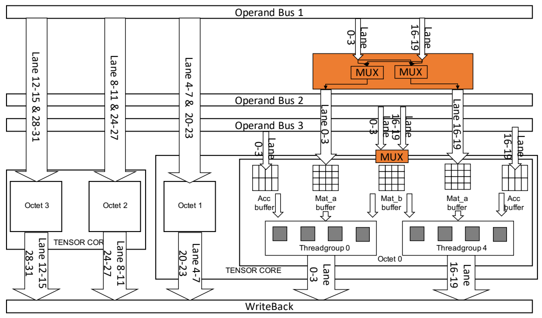


Figure 15: Proposed TCU Architecture.

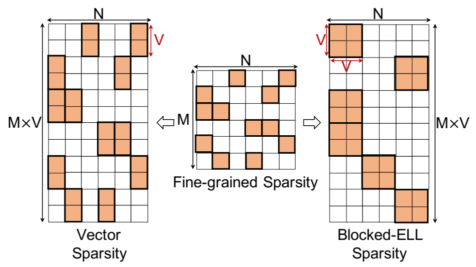


Figure 16: Benchmark Construction

To generate the output tile, each CTA traverses the dimension K with stride 64. Each octet holds the partial sums in its local registers. For each step, The LHS fragment in Figure 14 ❶ is loaded. Then, the warp takes four sub-steps, each sub-step load ❷ in Figure 14 and computes a $(8\times 64)\cdot(64\times V)$ tile (after the switching) as we mentioned before.

This kernel has a tighter budget for the registers, as each octet hold at least one set of the partial sums, and we rely on the compiler to determine the best strategy for register reusing. When K is traversed, the partial sums in different Octets are accumulated with warp shuffle primitives. The final result is reordered and written to DRAM with vector memory operations.

7 EXPERIMENTS

In this section, we compare the performance of our proposed kernels with cuSPARSE and the FPU baseline extended from Sputnik [6]. We justify the speedup and our motivations with detailed profiling results on a representative benchmark.

7.1 Experiment Setup

7.1.1 Benchmark Construction. Figure 16 illustrates how we construct the benchmarks from ResNet 50 under magnitude pruning in the DLMC dataset [22]. Given a $M\times N$ sparse matrix from DLMC with sparsity S, as our column vector sparse encoding is equivalent with replacing the nonzero scalars with vector types, we use the *csrRowPtr* and *csrColInd* of the sparse matrices, and randomly generate a nonzero vector with length V for each indexed position. To construct the blocked-ELL format sparse matrix, we first set the block size to the vector length V, and then compute the number of blocks in each row with $\lceil N/V\times S\rceil$. At last we generate the column indices of the nonzero blocks with uniform distribution. Therefore, the Blocked-ELL format has the same sparsity and problem size with the column vector sparse encoding.

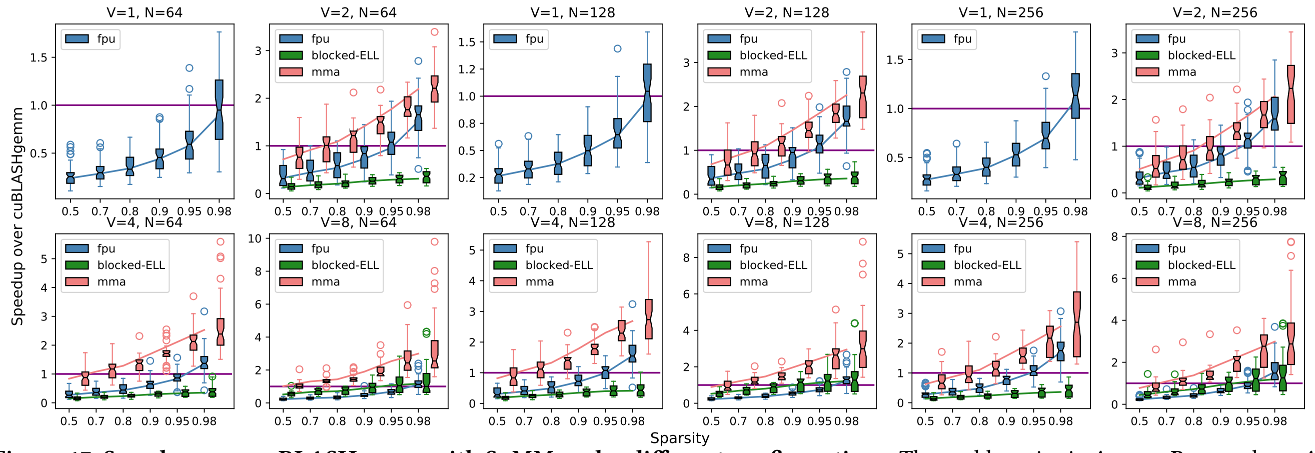


Figure 17: Speedup over cuBLASHgemm with SpMM under different configurations. The problem size is $A_{M \times K} \cdot B_{K \times N}$ where A is the sparse matrix under sparsity in {0.5,0.7,0.8,0.9,0.95,0.98}. The size M and K are given in the benchmarks, N is picked from $\{64, 128, 256\}$.

7.1.2 Baseline Kernels: For both SpMM and SDDMM, we compare our TCU-based 1-D Octet Tiling with an FPU baseline and a TCU baseline. The FPU baseline is obtained by extending the kernels in Sputnik [6] following Section 5.1 and 6.1 to support the column vector sparse encoding. The tiling sizes are tuned on a subset of benchmarks to find a configuration that brings the highest geometric mean speedup. We directly use the Blocked-ELL based SpMM kernel in cuSAPRSE as the TCU baseline for SpMM. As the SDDMM under structured sparsity is not supported by off-the-shelf libraries, we use the kernel in Section 6.2 as the TCU baseline. Finally, we use cuBLASHgemm kernel as the dense baseline.

7.1.3 Removing HMMA instructions: While our tiling design exposes opportunities for removing additional *HMMA*s, as existing SASS assemblers like [28] do not support this modification, we leave it for future work.

7.2 SpMM

The speedup achieved by our SpMM kernel is summarized in Figure 17. We use the box plot to illustrate the distribution of the speedup achieved on different benchmarks. Furthermore, following Gale et al. [6], we compute the geometric mean speedup and plot it with the solid line in Figure 17. "fpu", "blocked-ELL", and "mma" correspond to the FPU baseline extended from Sputnik[6], blocked-ELL based SpMM kernel in cuSPARSE, and our implementation with TCU-based 1-D Octet Tiling, respectively.

7.2.1 Overall Performance Description: Across all benchmarks, our TCU based 1-D Octet Tiling (mma) achieves **1.34-4.51x** and **1.71-7.19x** geometric mean speedup over the FPU and TCU baselines. Moreover, it outperforms cuBLASHgemm under **> 80%**, **> 70%**, and **> 50%** sparsity under the tiny 2×1, 4×1, and 8×1 grain size. This illustrates that higher speedup can be achieved with larger vector length V, which justifies our motivation of achieving practical speedup under moderate sparsity with column vector sparse encoding.

7.2.2 Performance Analysis: We further justify the above speedup with detailed profiling. Following Section 3, we profile the kernels on $\boldsymbol{A}_{2048 \times 1024} \times \boldsymbol{B}_{1024 \times 256}$ under 90% sparsity. First, we compare the three implementations of SpMM in terms of the five guidelines we propose in Section 3. We use the percentage of pipeline stall caused by "No Instruction", "Wait", and "Short Scoreboard" for guideline I, III, and IV, number of thread blocks for II, and Sector per Request to L1 cache for V. The results are listed in Table 2, we mark the results that are significantly worse than others with red.

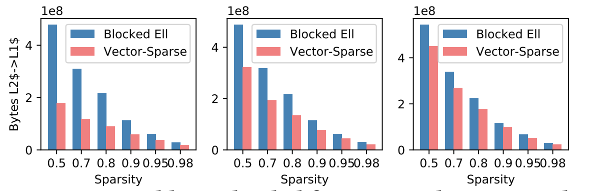


Figure 18: Total bytes loaded from L2 cache to L1 cache

Compared with our TCU-based 1-D Octet Tiling, the FPU baseline suffers more from the "No Instruction" and "Wait" stalls. We observe that the FPU baseline has 3776 and 6968 lines in their SASS code. Moreover, 3,402,752 and 3,407,872 *HUML+FADD* instructions are executed under $V = 4$ and $V = 8$, respectively. On the other hand, our kernel has only 384 and 416 lines in the SASS code, with 429,504 and 215,104 executed HMMA instructions. This greatly reduces instruction cache miss and stalls for dependency on fixed latency instructions. Besides, the "Sectors/Req" of the FPU baseline is only around 4. This is because when tuning its tiling size, we find that having $\#Subwarp = 1$ to improve the grid size generally improves the overall performance, while this comes at the cost of using shorter vector memory operation. The Blocked-ELL kernel suffers from "No Instruction", "Wait", and "Short Scoreboard" stalls, which accords with our analysis in Section 3.

To justify the argument in Section 4 that the data reuse is independent of the column number in the block sparse matrix, we profile the total number of bytes loaded from L2 cache to L1 cache in our column vector sparse encoding and blocked-ELL format under the same problem size and sparsity. As shown in Figure 18, our column vector sparse encoding loads even fewer data from L2 to L1 cache than the Blocked-ELL format, across all the sparsity levels.

Table 2: The 5 guidelines in different implementations.

Kernel	No Instruction	# Thread Block	Wait	Short Scoreboard	Sectors/Req
SpMM, V=4					
MMA	1.1%	2048	4.7%	4.5%	12.56
CUDA	11.0%	2048	11.6%	2.6%	4.04
Blocked-ELL	42.6%	1024	21.0%	11.9%	14.92
SpMM, V=8					
MMA	1.1%	1024	6.2%	2.6%	13.22
CUDA	52.2%	1024	8.3%	2.0%	4.27
Blocked-ELL	35.1%	512	16.2%	12.1%	13.85

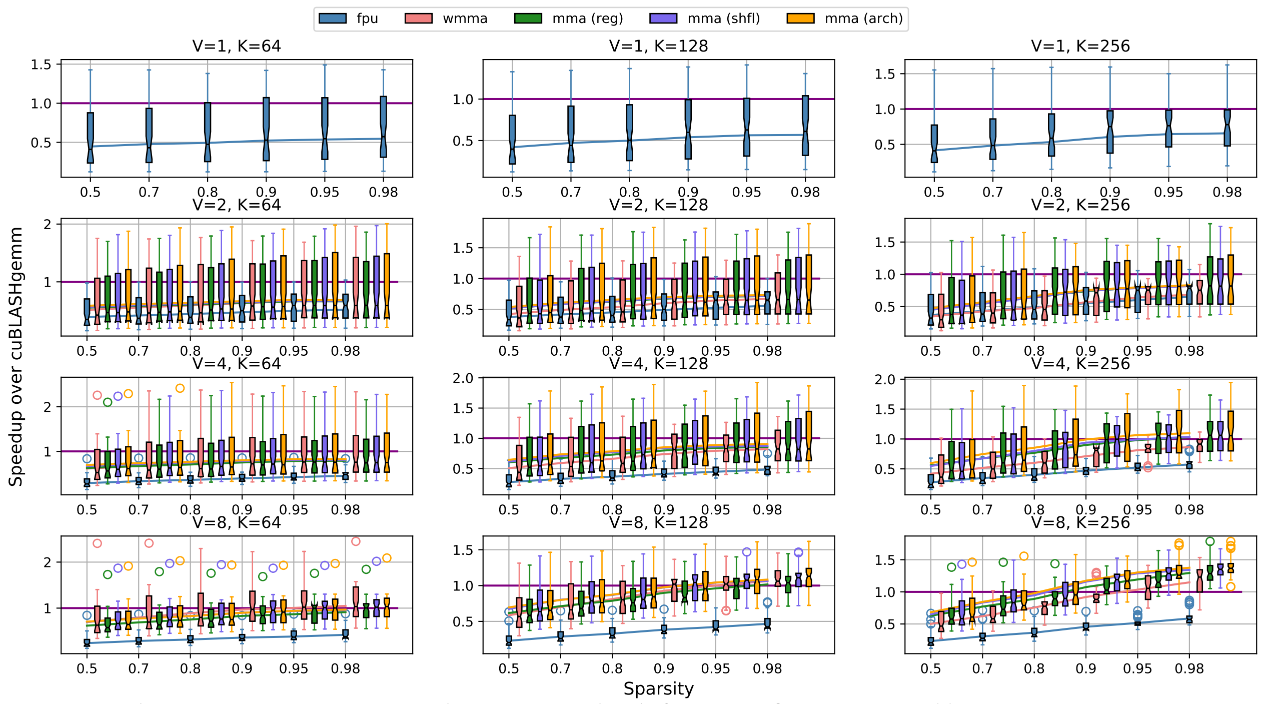


Figure 19: Speedup over cuBLASHgemm with SDDMM under different configurations. Problem size is $A_{M\times K} \cdot B_{K\times N} = C_{M\times N}$, C is the sparse matrix under sparsity in {0.5,0.7,0.8,0.9,0.95,0.98}. The size M and N are given in the benchmarks, K is picked from $\{64, 128, 256\}$.

7.3 SDDMM

The speedup achieved by our SDDMM kernels are summarized in Figure 19. The "fpu" denotes the FPU baseline in Section 6.1, "wmma" corresponds to the TCU baseline in Section 6.2. As we propose three different methods to handle the inverted pattern, we mark the one that adds additional registers with "mma (reg)", the one that shuffles the source operands before computation with "mma (shfl)", and the one based on the new TCU architecture with "mma (arch)". For the last one, we develop a fake kernel to simulate the performance by assuming that operands are switched in TCU.

7.3.1 Overall Performance Description: Our TCU-based 1-D Octet Tiling achieves considerable speedup over the baselines across all the setups except for $K = 64, V = 8$. Specifically, it achieves **1.27-3.03x** and **0.93-1.44x** geometric mean speedup over the FPU and TCU baselines. Besides, speedup at **> 90%** sparsity is achieved under $V = 8$ and $K = 256$. Moreover, with the modified TCU architecture, the mma (arch) consistently outperforms the mma (reg) and mma (shlf). This demonstrates that our simple architecture optimization can effectively improve the performance.

7.3.2 Performance Analysis: We observe that the *SHFL* (Warp Wide Register Shuffle) + *FADD* accounts for 29.5% of the total instructions executed under $V = 8$ and $K = 64$ settings with our TCU based 1-D Octet tiling. The percentage is reduced to 17.2% under $V = 8$ and $K = 256$. As these instructions are primarily used for accumulating the partial sums of each Octet at the end of the kernel. This suggests that When K is small and V is large, the overhead of this reduction is not negligible and will offset the benefit of the dedicated tiling design. When K gets larger, it is obvious that our Octet tiling (mma) achieves significant better performance.

We present more detailed profiling results to justify the speedup achieved by our kernels. With the same setup for Table 2 expect that the benchmark size is $A_{2048\times256} \times B_{256\times1024} = C_{2048\times1024}$ and C has 90% sparsity, the results are summarized in Table 3. As all the three implementations of our MMA are more or less similar in term of these five guidelines, we just list the result of mma (reg).

Similarly, the FPU baseline suffers more from "No Instruction", "Wait" stalls, and it has smaller "Sector/req" than the other two implementations. The TCU baseline is limited by the shared memory bandwidth. These observations accord with our arguments in Section 6. We also compare the mma (reg), mma (shfl), and mma (arch) to justify the architecture level optimization. We observe that given credit to our architecture modification, our mma (arch) uses 33% fewer registers and has 21.3% more active warps per scheduler than mma (reg), as it removes the need of additional registers for partial sums from the inverted pattern. Besides, it has 10.4% fewer instructions that is majorly contributed by the removed SHFL instructions for operands switching.

Table 3: The 5 guidelines in different implementations.

Kernel	No Instruction	# Thread Block	Wait	Short Scoreboard	Sectors/Req
		SDDMM, V=4			
MMA	0.8%	16384	10.7%	2.1%	8.83
CUDA	6.1%	16384	28.1%	2.5%	3.53
WMMA	0.3%	16384	10.6%	14.4%	8.82
		SDDMM, V=8			
MMA	1.0%	8192	11.0%	1.9%	9.25
CUDA	7.3%	16384	24.6%	3.1%	3.33
WMMA	0.4%	8192	9.5%	17.9%	9.26

7.4 Application: Sparse Transformer

Transformer models based on the self-attention mechanism have achieved unrivaled performance in natural language processing and vision[8, 25, 30]. Under sequence length l and feature dimension m, the self-attention layer takes query, key, and value $\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V} \in \mathbb{R}^{l\times k}$

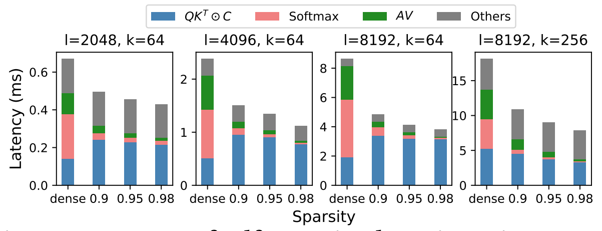


Figure 20: Latency of self-attention layer in various setups.

and computes the output with

$$A = Softmax\left((QK^T \odot C)/\sqrt{k}\right), Attention(Q, K, V) = AV, \quad (1)$$

where C is an optional sparse mask that prunes most of entries in the matrix A. In the dense setting, C is just a dense matrix filled with ones, thus Equation 1 has quadratic computation and memory complexity in terms of sequence length l, which limits it from modeling long sequences like documents or images. To address this issue, many studies have been proposed to apply a fixed or learned sparse mask C [25]. Under this setup, $QK^T \odot C$ and AV can be formulated as SDDMM and SpMM, respectively. However, without efficient GPU kernels, they could be much slower than their dense counterpart, thus previous studies [30] apply block sparsity with large block size like 32 or 64. In this section, we will show that with our SDDMM and SpMM kernels, speedup over the dense implementation can be achieved under much smaller granularity, thus offers larger design space when constructing the sparse masks.

Experimental Setup: We trained a transformer model with a fixed sparse attention mask on the byte-level text classification task in Long-Range Arena (LRA) [24], a benchmark for transformers under long-sequence scenarios. In this task, the sequence length is 4000. We set the model configuration and training parameters to be the same as the original dense baseline. The 4-layer transformer model has 4 attention heads for each attention layer, and the feature dimension of each head is 64. For the sparse attention pattern, we follow Gale et al. [6] but add our 8×1 vector sparsity constraints. Specifically, we generate fixed attention masks with a dense band of size 256 along the diagonal and off-diagonal random attention. The overall sparsity is 90% and the attention mask can be expressed by our column-vector sparse encoding. We also implement a custom softmax kernel that works on column vector sparse encoding. For the half-precision models, we directly quantize the weights and activations to half without finetuning. We evaluate the inference throughput and peak memory usage under batch size 8 and average the results over 10 runs.

Table 4: Sparse Transformer Results

Model	Dense(float)	Dense(half)	Sparse(half)
Accuracy	65.12%	65.09%	65.01%
Throughput (seq / s)	74.7	182.6	258
Peak Memory	4.44 GB	2.22 GB	170.03 MB

Results & Analysis: As shown in Table 4, the sparse model under half precision achieves 3.45x and 1.41x end-to-end speedup over the dense model under single and half precision, respectively. The peak memory usage is reduced by 26.74x and 13.37x. With vector-wise sparsity, we are able to achieve only 0.11% of accuracy degradation compared with the dense transformer baseline.

Ablation Study: Under the long-sequence scenario, the multi-head attention layer could contribute more than 70% of total execution time. We further illustrate the speedup achieved in different parts in the attention layer in Figure 20. In terms of the whole layer, we achieve $1.35 - 1.78x$, $1.48 - 2.09x$, and $1.57 - 2.30x$ speedup under 90%, 95%, and 98% sparsity, respectively. Our SpMM and Softmax kernels effectively reduce the latency contributed by $Softmax$ and AV. The SDDMM kernel is slower than its dense counterparts when $k = 64$. It is because 64 is too small, which accords with the observation in Figure 19. Notably, the sparsity cannot be utilized in $Softmax$ and AV without the SDDMM kernel, and our SDDMM achieves better performance than other baselines. Besides, as shown in the last figure in Figure 20, the SDDMM outperforms its dense counterpart when $k = 256$.

8 DISCUSSION & CONCLUSION

While the SpMM and SDDMM operations in our paper are defined in terms of row-major matrices, they can also be applied to column-major matrices by mathematically transposing both LHS and RHS of the equation. Specifically, we can have $D^T = B^T C^T$ and $D^T = (B^T)^T A^T \odot C^T$ for SpMM and SDDMM, where D^T, A^T, and B^T are column-major dense matrices. C^T is a transposed sparse matrix under column-vector sparse encoding, which can be viewed as "row vector sparse encoding" that is composed of short row vectors aligned along the horizontal dimension. The position of these short row vectors are encoded in compressed sparse column (CSC).

Although our column vector sparse encoding only requires the sparse matrix to be composed of short column vectors aligned along the vertical dimension, additional constraints can be added along the horizontal dimension in need. While additional adjustment can be applied to the way that operands are indexed, the CTA tile remains identical under different constraints, so our TCU-based 1-D Octet tiling is still applicable. Besides the case with default setting provided in Section 7.4, we further discuss two cases below.

Case 1: When applying our method to neural network training, one can store the activations $X \in \mathbb{R}^{n \times N}$ in row-major, where n is the input feature dimension and N is the batch size. We have

$$❶\, Y = WX,\; ❷\, \frac{\partial \mathcal{L}}{\partial X} = W^T \frac{\partial \mathcal{L}}{\partial Y},\; ❸\, \frac{\partial \mathcal{L}}{\partial W} = \frac{\partial \mathcal{L}}{\partial Y} X^T. \quad (2)$$

❶ and ❷ in the above equation can be computed with our SpMM kernel, and SDDMM kernel is applicable in ❸. As both W and W^T are used, we need to have square nonzero blocks aligned in both vertical and horizontal dimensions, then we can encode both W and W^T with our column-vector sparse encoding. As square nonzero blocks are used, we can use one column index per block, and access the columns in the block with an unrolled loop in the kernel.

Case 2: Another extreme case is all the column vectors in the same row should be zero or nonzero at the same time (a short and wide matrix), which is used in the global attention in sparse transformer [30]. Because all the entries are nonzero in a nonzero row, we can directly access the entries in a for loop.

All in all, in this paper, we propose efficient sparse GPU kernels for half precision SpMM and SDDMM operation under structured sparsity. Our kernels are based on two essential contributions. The first is column vector sparse encoding, which achieves same data reuse as block sparsity while delivering smaller granularity to help maintain neural network model quality. The second contribution is a novel mapping and tiling strategy namely TCU-based 1-D Octet Tiling. With the proposed tiling method, our kernels are able to achieve both efficient memory access and efficient computation even with tiny sparse granularity. Experiments on the DLMC sparse matrix benchmark illustrates that the proposed kernels achieve **1.71-7.19x** geometric mean speedup over the Blocked-ELL based SpMM kernel. Moreover, our SpMM and SDDMM kernel achieve practical speedup over their dense counterparts with > **70%** and > **90%** sparsity under the 4×1 grain size and half precision. Benefited from our design, we achieve 1.41x end-to-end speedup and 13.37x peak memory reduction on the sparse transformer inference task.

9 ACKNOWLEDGEMENTS

This work was supported in part by NSF 1925717. Use was made of computational facilities purchased with funds from the National Science Foundation (OAC-1925717) and administered by the Center for Scientific Computing (CSC). The CSC is supported by the California NanoSystems Institute and the Materials Research Science and Engineering Center (MRSEC; NSF DMR 1720256) at UC Santa Barbara.

REFERENCES

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://www.tensorflow.org/ Software available from tensorflow.org.

[2] V. Akhlaghi, A. Yazdanbakhsh, K. Samadi, R. K. Gupta, and H. Esmaeilzadeh. 2018. SnaPEA: Predictive Early Activation for Reducing Computation in Deep Convolutional Neural Networks. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. 662–673. https://doi.org/10.1109/ISCA.2018.00061

[3] Zhaodong Chen, Mingyu Yan, Maohua Zhu, Lei Deng, Guoqi Li, Shuangchen Li, and Yuan Xie. 2020. fuseGNN: accelerating graph convolutional neural network training on GPGPU. In *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE, 1–9.

[4] Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. 2019. Generating Long Sequences with Sparse Transformers. *CoRR* abs/1904.10509 (2019). arXiv:1904.10509 http://arxiv.org/abs/1904.10509

[5] Abdul Dakkak, Cheng Li, Jinjun Xiong, Isaac Gelado, and Wen-mei Hwu. 2019. Accelerating reduction and scan using tensor core units. In *Proceedings of the ACM International Conference on Supercomputing*. 46–57.

[6] Trevor Gale, Matei Zaharia, Cliff Young, and Erich Elsen. 2020. Sparse GPU Kernels for Deep Learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020*.

[7] Tong Geng, Ang Li, Tianqi Wang, Chunshu Wu, Yanfei Li, Antonino Tumeo, and Martin C. Herbordt. 2019. UWB-GCN: Hardware Acceleration of Graph-Convolution-Network through Runtime Workload Rebalancing. *CoRR* abs/1908.10834 (2019). arXiv:1908.10834 http://arxiv.org/abs/1908.10834

[8] Kai Han, Yunhe Wang, Hanting Chen, Xinghao Chen, Jianyuan Guo, Zhenhua Liu, Yehui Tang, An Xiao, Chunjing Xu, Yixing Xu, et al. 2020. A Survey on Visual Transformer. *arXiv preprint arXiv:2012.12556* (2020).

[9] Song Han, Huizi Mao, and William Dally. 2016. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding.

[10] Song Han, Jeff Pool, John Tran, and William J. Dally. 2015. Learning both Weights and Connections for Efficient Neural Networks. *CoRR* abs/1506.02626 (2015). arXiv:1506.02626 http://arxiv.org/abs/1506.02626

[11] Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele P Scarpazza. 2018. Dissecting the NVIDIA volta GPU architecture via microbenchmarking. *arXiv preprint arXiv:1804.06826* (2018).

[12] Grzegorz Kwasniewski, Marko Kabić, Maciej Besta, Joost VandeVondele, Raffaele Solcà, and Torsten Hoefler. 2019. Red-blue pebbling revisited: near optimal parallel matrix-matrix multiplication. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–22.

[13] Bing Li, Wei Wen, Jiachen Mao, Sicheng Li, Yiran Chen, and Hai Li. 2018. Running sparse and low-precision neural network: When algorithm meets hardware. In *2018 23rd Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 534–539.

[14] Liu Liu, Lei Deng, Zhaodong Chen, Yuke Wang, Shuangchen Li, Jingwei Zhang, Yihua Yang, Zhenyu Gu, Yufei Ding, and Yuan Xie. 2020. Boosting Deep Neural Network Efficiency with Dual-Module Inference. In *Proceedings of the 37th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 119)*, Hal Daumé III and Aarti Singh (Eds.). PMLR, 6205–6215. http://proceedings.mlr.press/v119/liu20c.html

[15] Huizi Mao, Song Han, Jeff Pool, Wenshuo Li, Xingyu Liu, Yu Wang, and William J Dally. 2017. Exploring the regularity of sparse structure in convolutional neural networks. *arXiv preprint arXiv:1705.08922* (2017).

[16] Sharan Narang, Gregory F. Diamos, Shubho Sengupta, and Erich Elsen. 2017. Exploring Sparsity in Recurrent Neural Networks. *CoRR* abs/1704.05119 (2017). arXiv:1704.05119 http://arxiv.org/abs/1704.05119

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

[18] Tesla NVIDIA. 2017. V100 GPU Architecture: The world's most advanced datacenter GPU. *NVIDIA Corporation* (2017).

[19] Mi Sun Park, Xiaofan Xu, and Cormac Brick. 2018. Squantizer: Simultaneous learning for both sparse and low-precision neural networks. *arXiv preprint arXiv:1812.08301* (2018).

[20] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.). Curran Associates, Inc., 8024–8035. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[21] Md Aamir Raihan, Negar Goli, and Tor M Aamodt. 2019. Modeling deep learning accelerator enabled gpus. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 79–92.

[22] Google Research. [n.d.]. Deep Learning Matrix Collection. https://github.com/google-research/google-research/tree/master/sgk.

[23] Sainbayar Sukhbaatar, Edouard Grave, Piotr Bojanowski, and Armand Joulin. 2019. Adaptive Attention Span in Transformers. *CoRR* abs/1905.07799 (2019). arXiv:1905.07799 http://arxiv.org/abs/1905.07799

[24] Yi Tay, Mostafa Dehghani, Samira Abnar, Yikang Shen, Dara Bahri, Philip Pham, Jinfeng Rao, Liu Yang, Sebastian Ruder, and Donald Metzler. 2020. Long Range Arena: A Benchmark for Efficient Transformers. *arXiv preprint arXiv:2011.04006* (2020).

[25] Yi Tay, Mostafa Dehghani, Dara Bahri, and Donald Metzler. 2020. Efficient transformers: A survey. *arXiv preprint arXiv:2009.06732* (2020).

[26] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *NIPS*. 6000–6010. http://papers.nips.cc/paper/7181-attention-is-all-you-need

[27] Ganesh Venkatesh, Eriko Nurvitadhi, and Debbie Marr. 2017. Accelerating deep convolutional networks using low-precision and sparsity. In *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2861–2865.

[28] Da Yan, Wei Wang, and Xiaowen Chu. 2020. Optimizing Batched Winograd Convolution on GPUs. In *25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '20)*. ACM, San Diego, CA, USA.

[29] Manzil Zaheer, Guru Guruganesh, Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontanon, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, and Amr Ahmed. 2021. Big Bird: Transformers for Longer Sequences. arXiv:2007.14062 [cs.LG]

[30] Manzil Zaheer, Guru Guruganesh, Kumar Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontanon, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, and Amr Ahmed. 2020. Big Bird: Transformers for Longer Sequences. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., 17283–17297. https://proceedings.neurips.cc/paper/2020/file/c8512d142a2d849725f31a9a7a361ab9-Paper.pdf

[31] Maohua Zhu, Tao Zhang, Zhenyu Gu, and Yuan Xie. 2019. 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* (Columbus, OH, USA) *(MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 359–371. https://doi.org/10.1145/3352460.3358269