Probing the Efficacy of Hardware-Aware Weight Pruning to Optimize the SpMM routine on Ampere GPUs

Roberto L. Castro*
roberto.lopez.castro@udc.es
Universidade da Coruña, CITIC
A Coruña, Spain

Diego Andrade
diego.andrade@udc.es
Universidade da Coruña, CITIC
A Coruña, Spain

Basilio B. Fraguela
basilio.fraguela@udc.es
Universidade da Coruña, CITIC
A Coruña, Spain

ABSTRACT

The Deep Learning (DL) community found in pruning techniques a good way to reduce the models' resource and energy consumption. These techniques lead to smaller sparse models, but sparse computations in GPUs only outperform their dense counterparts for extremely high levels of sparsity. However, pruning up to such sparsity levels can seriously harm the accuracy of the Neural Networks (NNs). To alleviate this, novel performance-aware pruning techniques favor the generation of more regular sparse matrices that can improve the exploitation of the underlying hardware. Nevertheless, an important drawback is that these techniques heavily condition the location of the non-pruned values, which can strongly degrade the accuracy of the models.

This paper focuses on improving the performance of the SpMM routine on DL workloads by combining performance-aware pruning with pruning-independent SpMM kernels to relax input-format constraints. We start with a microarchitecture-level performance study of SOTA SpMM implementations to identify their main bottlenecks and flaws. Then, the paper centers on maximizing the performance of the routine by adjusting the parameters of performance-aware pruning techniques to the hardware properties. This second study explains the intrinsic causes of the observed performance results. We show that, following this approach, a generic SpMM routine can perform up to 49% and 77% better for half and single precision, respectively, than using non-performance-aware pruning, providing speedups over cuBlas of up to 1.87× and 4.20×, respectively. Additionally, the performance achieved on half precision is boosted with a new Ampere-ready specialized implementation for the column-vector sparse format, CLASP, which achieves a 2.42× speedup over cuBlas. Finally, we also introduce ad-colPrune, a novel pruning technique that widens the design space of possible trade-offs between performance and accuracy.

CCS CONCEPTS

• **Computing methodologies → Parallel programming languages**; *Machine learning*; • **Computer systems organization** → *Single instruction, multiple data*; • **General and reference** → Performance.

KEYWORDS

deep learning, network pruning, sparsity, SpMM, CUDA, GPU

ACM Reference Format:
Roberto L. Castro, Diego Andrade, and Basilio B. Fraguela. 2022. Probing the Efficacy of Hardware-Aware Weight Pruning to Optimize the SpMM routine on Ampere GPUs. In *International Conference on Parallel Architectures and Compilation Techniques (PACT '22), October 8–12, 2022, Chicago, IL, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3559009.3569691

1 INTRODUCTION

Sparse computation can be used in Machine Learning (ML) to reduce the models' size without harming the accuracy. However, existing research on sparse computing for scientific applications is not fully portable to ML because the characteristics of both types of workloads are different. For instance, while in scientific applications sparsity levels can reach values close to 100%, in ML they are usually significantly below that ratio, as otherwise accuracy could be seriously affected [15]. Furthermore, in ML there are additional relevant factors to consider such as the pruning technique that generates the sparsity, and the architecture of the pruned models.

In Deep Learning (DL), the usage of GPUs to speed up linear algebra kernels is the norm [38]. In addition, new generations of GPUs are equipped with specialized hardware that targets the core computations of this type of workloads, such as Tensor Core Units (TCUs) [37]. Thus, the design of kernels that exploit such hardware is critical to achieve good performance. Although there has been progress on sparsity for ML, it is mainly focused on the design of new pruning techniques. The contributions focused on optimizing sparse computation to address the peculiarities of ML workloads are scarce and very recent.

The usage of sparse computation in the context of ML seeks to reduce the operation count, the memory accesses and the memory consumption. Previous works have shown that the sparsity level, the size of the original input matrices, the mean row length and the load imbalance between rows are key factors in the performance of these kernels [8]. One way to optimize sparse computation in ML is to design hardware-aware pruning techniques, i.e., techniques created to generate more regular hardware-optimized sparsity patterns, such as pre-defined shapes of groups of non-zero elements (e.g., vertical/column vectors) or removing whole components to keep the computation dense. However, forcing this regularity can harm the accuracy of the resulting network as this conditions, to a certain extent, the location of the non-zeros [18, 21]. Thus, balancing performance and accuracy is key to properly exploit sparsity in DL.

This paper tests a solution based on combining hardware-aware pruning techniques with pruning-independent SpMM kernels, i.e.,

*Corresponding author: Roberto L. Castro (roberto.lopez.castro@udc.es), Universidade da Coruña, CITIC, Computer Architecture Group, 15071 A Coruña, Spain

PACT '22, October 8–12, 2022, Chicago, IL, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9868-8/22/10…$15.00
https://doi.org/10.1145/3559009.3569691

kernels with no input sparsity-pattern constraints. The hypothesis is that tuning some pruning techniques' configuration parameters to favor hardware utilization can boost SOTA SpMM implementations' performance. This would not only enable more flexible weight selection heuristics and the reuse of CUDA kernels, but it would also allow modulating the balance between performance and accuracy at pruning time. Our main contributions are:

(1) A microarchitecture-level study of the performance of SOTA pruning-independent SpMM implementations on genuine ML workloads. It focuses on the efficiency of the routines on sparse matrices generated by non-hardware-aware pruning techniques and unveils the main flaws of that implementations.
(2) An in-depth study on hardware-aware pruning techniques' configuration parameters, their linkage with microarchitecture aspects, and their effect on pruning-independent SpMM kernels. We show that hardware-optimized sparse matrices can provide up to 49% (half) and 77% (single) larger speedups than non-optimized ones without modifying the SpMM code.
(3) A new column-vector pruning-aware implementation of the SpMM routine that supports the characteristics of the Ampere platform, achieving up to 2.42× speedup w.r.t. cuBlas [1].
(4) A novel hardware-aware pruning technique built on top of the conclusions of the two initial studies, which increases the design space of possible performance-accuracy trade-offs.

2 BACKGROUND

This section introduces the technical background of the paper, consisting of an overview of the weight network pruning techniques and the most relevant aspects of the Nvidia Ampere architecture. We assume the reader is generally familiar with GPGPU programming concepts and terminology (see [23] for an overview).

2.1 Network pruning

Network pruning techniques focus on removing some connections of neural networks to speed up network inference, shorten training time and reduce memory usage, while avoiding accuracy degradation. This is relevant for mobile devices or to shrink huge network architectures such as GPT-3 [3] or Megatron-Turing NLG [1].

Weight pruning algorithms can be classified according to the size of the parts of the network that are preserved into fine-grained and coarse-grained ones. At one end of the scale, fine-grained algorithms eliminate individual connections (weights) of a network in a non-structured way. While this is more flexible, the resulting sparse matrices are more irregular and they prevent sparse kernels from efficiently exploiting the hardware resources [12, 33].

In the middle of the scale, group-level pruning algorithms select medium-sized groups of elements rather than individual ones. According to the group's dimension, there are vector-level (1D) and kernel-level (2D) methods. Furthermore, vector-level pruning can be separated into column-vector and row-vector pruning depending on whether the elements are aligned in the vertical or horizontal dimension, respectively. This approach sacrifices flexibility, and thereby accuracy, for performance, as the resulting sparse matrices are more regular, which enhances data reuse, and as a result sparse kernels perform better on them [2, 39].

[1]Code available at: https://github.com/UDC-GAC/CLASP

Method	Scheme	Drop
random pruning	none	$w_{i,j} \in_R W$
magnitude pruning	data-free	$\lvert w_{i,j}\rvert \leq thresh$
variational dropout	training-aware	$log\ \alpha \geq thresh$; $w_{i,j} \sim \mathcal{N}(w_{i,j}\lvert\theta,\ \alpha\theta^2)$
L_0 regularization	training-aware	$\beta\lvert\lvert w_{i,j}\rvert\rvert_0$

Table 1: Comparison of different pruning methods, being W a NN, w each individual weight, (θ, α) NN parameters, and β a penalty term

At the other end of the scale, coarse-grained algorithms modify the structure of the network by removing whole filters, neurons or heads [17]. For this reason, these algorithms are also called structural pruning, in opposition to the two previous types, which are referred to as non-structural approaches. After applying structural pruning computation remains dense, making it possible to use existing optimized libraries such as cuDNN [5]. However, this approach can seriously harm the accuracy of the pruned network [21]. This paper focuses on non-structured pruning techniques, both fine-grained and group-level ones.

2.1.1 Candidate selection for removal. The policy to choose the elements to prune is the core of network pruning methods. Random selection can be quite effective in some settings [20, 25]. Another very general, simple and effective method is magnitude pruning (MP), which picks weights whose absolute value is below a threshold. This is a data-free scheme, meaning that it bases its decisions only on the network structure. MP has shown SOTA performance and high compression rates with minimal accuracy loss [7]. There are also training-aware schemes, which perform a full training for candidate selection. Some examples are L_0 regularization and variational dropout, which rely on the L_0 norm and a Bayesian approach of the weights, respectively, to select candidates [15]. Table 1 summarizes these concepts.

2.2 Ampere architecture

This section introduces relevant details of the Ampere architecture focusing on the Nvidia RTX 3090 GPU [27] used in this paper.

The Nvidia Ampere GPU architecture has an array of 82 Streaming Multiprocessor (SMs) elements that share a 6MB L2 cache and 24GB of DRAM. Each SM is divided in partitions of four processing blocks, each one with its corresponding warp scheduler, Register File (RF), dispatch unit and L0 instruction cache. The four blocks share a 128KB L1 cache that can be partially used as Shared MEMory (SMEM). Each of those processing blocks is equipped with different sub-core units. In total, a SM contains 128 Floating-Point Units (FPUs) and 4 Tensor-Core Units (TCUs), which in this architecture also incorporate Sparse Tensor Cores. Let us now introduce the different types of the aforementioned sub-core units:

- **Floating-Point Units (FPU)**. Traditional CUDA Cores that carry out FP32 operations. FPUs can perform up to one single precision multiply-and-accumulate operation (FMA) per cycle. The Ampere SMs have been redesigned to support double-speed processing, delivering a $2x$ speedup w.r.t. the Turing generation.
- **Tensor-Core Units (TCU)**. First introduced in the Volta architecture, Tensor-cores are able to carry out one matrix multiply-and-accumulate operation (MMA) per cycle, peaking $8x$ more performance than FPUs. Ampere delivers third-generation tensor cores that double the performance of the Turing generation.
  - **Sparse Tensor Cores (SPTCU)**. The aforementioned third-generation tensor cores introduced in Ampere support sparse

computation and take advantage of these workloads to accelerate math operations by up to $2x$.

3 RELATED WORK

3.1 Software resources

The Sparse matrix-dense Matrix Multiplication (SpMM) and Sample Dense-Dense Matrix Multiplication (SDDMM) are the main operations used in deep neural networks' forward and backpropagation steps. The main available implementations of these routines are:

- **Nvidia cuSparse** [28] is a library that implements several basic linear algebra subroutines for sparse matrices. It was originally designed for scientific workloads and it offers three formats to represent sparse matrices in the SpMM routine: COO, CSR and Blocked-Ellpack.
- **Sputnik** [8] is a library designed from scratch for sparsity in Deep Learning. It focuses on gaining flexibility on workload scheduling by defining a one-dimensional tiling scheme. It stores sparse matrices in CSR format.
- **Vector-Sparse** [4] is an evolution of Sputnik focused on exploiting Tensor-Core Units. It takes advantage of 1D groups of elements to improve locality. Matrices are stored in the *Column-Vector Sparse Encoding* format.
- **Nvidia cuSparseLt** [29] is a new library from Nvidia that supports Sparse-Tensor Cores. It focuses on general matrix-matrix operations where at least one of the operands is sparse. It uses a new format to represent sparse matrices called N:M.

3.2 Performance-aware pruning on GPUs

Fine-grained pruning can preserve the original network accuracy and achieve high compression ratios [11, 13] but usually at the cost of performance [14, 36]. Hence the emergence of group-level pruning techniques [4, 19, 35] that seek to generate more regular sparse matrices has lead to the proliferation of new custom sparse representation formats [32]. As an example, the Blocked-Ell format [28], introduced to encode 2D blocks of non-zero elements, enables more regular memory accesses and improves data reuse. However, it has low performance with block sizes smaller than 8. Additionally, larger block sizes have been proved to seriously degrade network accuracy [21]. Alternatively, the N:M format enables the usage of sparse tensor cores, which can provide up to $2x$ speedup over the dense counterpart without harming accuracy [16, 24, 40]. However we have observed that the performance gain using SPTCU only starts to arise with big models or when large batch sizes are considered.

A new way to improve sparse computation performance is to transfer part of the hardware knowledge to the pruning phase, in order to design performance-aware pruning techniques [4, 18, 22, 32]. However, our general view is that the constraints established in these works are still too rigid. This paper seeks to demystify the most relevant microarchitecture aspects on the most recent Nvidia GPU architecture in order to define more flexible weight selection heuristics without sacrificing performance.

4 PERFORMANCE OPPORTUNITIES WITH PRUNING-AGNOSTIC SPMM KERNELS

This section describes the microarchitecture-level benchmarking of the SpMM routine on existing pruning-independent SOTA implementations using sparse matrices obtained after applying fine-grained pruning techniques, which are not hardware-aware. The contenders are cuBLAS, taken as a dense reference implementation, and Sputnik and cuSparse as the sparse references. In the case of cuSparse, we consider the performance separately using the COO and CSR formats. In the following, we will call pruning-independent or pruning-agnostic SpMM implementation to one whose code does not make any assumption about the input distribution of non-zero elements. Thus, it has not been optimized for a particular pattern (e.g., column-vector). The sparse input matrices used in this study and through the paper belong to the DLMC dataset [10] and come from different ResNet-50 sparse models trained on the ImageNet dataset [6], sparsified with: (1) variational dropout, (2) random pruning, (3) magnitude pruning and (4) extended magnitude pruning methods [7]. The testing hardware platform is a RTX 3090 Nvidia GPU belonging to the latest Nvidia Ampere architecture with CUDA 11.5.

Firstly, we analyzed the overall performance of the SpMM routine for single and half precision, for different sparsity levels and using different compressed storage formats in the case of cuSparse. The analysis showed that Sputnik outperforms cuBLAS for sparsity levels above 80% and 95% for single precision and half precision, respectively. However, cuSparse does it in single precision for sparsity levels above 99% and never for half precision. In cuSparse, COO performs better than CSR, especially in single precision. This preliminary study confirms the cuSparse limitations on ML workloads since it is optimized for scientific ones. However, although Sputnik was designed from scratch for ML problems, it also does not surpass cuBLas until relatively high sparsity levels, especially for half precision.

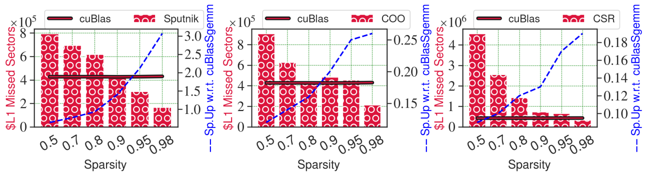
Now, we perform a study focused on identifying opportunities in these libraries for hardware usage improvement at the microarchitecture level.

4.1 Microarchitecture study definition

The study is based on metrics related to *memory usage* and others related to *computation* that have been obtained using the Nvidia Nsight Compute kernel profiler [30].

Memory aspects. An analysis of open-source SpMM implementations shows that sparse matrices are usually stored in SMEM due to frequent memory accesses while the dense matrix values are directly moved from GMEM to registers. Thus, data locality promotion is key to reduce costly Global MEMory (GMEM) accesses. Let us recall that the minimum memory transaction unit is the sector (32 *bytes*). Furthermore, GMEM accesses cached both in L1 and L2 are serviced with 128-byte transactions (4 sectors). Therefore, if the word size is larger than 4 bytes, each GMEM request is split into independent 128-byte memory requests: 2 memory requests, one per half-warp, for float2 or 4 memory requests, one per quarter-warp, for float4. The metrics selected to assess the memory behavior are:

- The number of **L1 missed sectors**, which is used to measure the unified L1/TEXture (TEX) cache performance.



(a) Sputnik (b) cuSparse COO (c) cuSparse CSR

Figure 1: L1 missed sectors of Sputnik, cuSparse and cuBlas (left y-axis) and speedup of Sputnik and cuSparse over cuBlas (right y-axis)

- The ratio of **Sectors/Requests**, to evaluate the efficiency of memory transactions. High ratios could indicate possible uncoalesced memory accesses. For instance, a value of 32 on float loads means that each thread of a warp is accessing a different 4-byte word in a different sector.

Compute aspects. Arithmetic intensity and efficiency are critical in order to hide memory latency with computation. To assess this, the following metrics have been selected.

- The number of **arithmetic instructions**, which directly affects the computation time and computational load. It can also affect the usage and performance of the L0 instruction cache.
- The **Branch efficiency**, that indicates the ratio of branches where all the active threads select the same branch target. It is used to detect thread divergence.

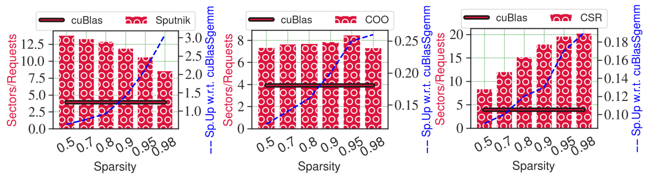
4.2 Performance analysis and evaluation on non-hardware-optimized sparse matrices

In our performance study, sparse methods outperformed their dense counterparts at lower sparsity levels for single precision data than for half precision. Still, this happened at relatively high sparsity levels, considering that pruning a model by more than 80% can start to impact its accuracy [31]. Let us analyze the causes for the observed performance for single and half precision separately.

4.2.1 Single precision. Figure 1 shows the evolution of the number of L1 missed sectors for Sputnik (CSR) and cuSparse (COO and CSR formats) for input matrices with increasing sparsity levels. The figure also shows the overall performance speedup they achieve w.r.t. cuBLAS. We can see that the number of missed sectors in Sputnik and cuSparse+COO is similar for low sparsity levels (50%), while it is much larger for cuSparse+CSR.

At 50% of sparsity, Sputnik's missed sectors are around twice those of cuBLAS, but they evolve favorably with sparsity. In fact, Sputnik outperforms cuBLAS when the number of missed sectors in both libraries is similar, at a sparsity level of 90%. In cuSparse+COO, while the number of missed sectors is substantially better than in Sputnik at 70% and 80% sparsity levels, the transition to higher levels of sparsity is not so favorable. The improvement of this metric is only evident when the sparsity reaches 98%. In cuSparse+CSR, the metric evolves promisingly with the sparsity level, but it starts from a much higher initial value, and thus cuSparse does not generate fewer missed sectors than cuBLAS until a sparsity level of 98%.

Figure 2 shows the ratio of Sectors to Requests. The Sputnik routine starts with 13 sectors per request since its implementation can use 128-bit LoaDs from Global memory instructions (LDG.128), float4 datatypes, which is quite close to the ideal value of 16 (see



(a) Sputnik (b) cuSparse COO (c) cuSparse CSR

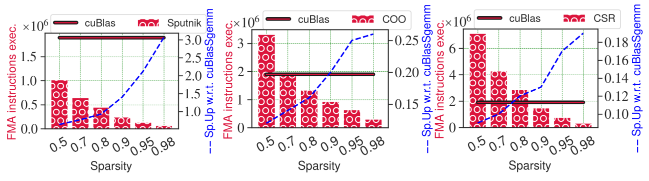
Figure 2: Sectors to Request ratio of Sputnik, cuSparse and cuBlas (left y-axis) and speedups of the first two over cuBlas (right y-axis)

Section 5.2). The theoretical peak is not reached in this case because residual values are loaded using LDG.32 instructions. As sparsity increases, residue values tend to have a bigger impact since it is more difficult to hide their processing with full tiles of non-zero values, leading to intra-warp load imbalance and to worsen the sector to requests ratio. Relatedly, an analysis of the Shader ASSembly (SASS) code of cuSparse shows that this routine always uses 32-bit loads, which implies that this metric should have a value close to 4. However, in cuSparse+COO the value is around 8, and it slightly increases with the sparsity level, which denotes a problem of poor memory coalescence among the threads of the same warp. The cuSparse+CSR version shows a much higher ratio, as it reaches 20 sectors per request for the highest sparsity level. This implies that, in this case, the coalescence problem is even worse.

library/Sparsity	0.5	0.7	0.8	0.9	0.95	0.98
cuSparse+COO	91.47	91.35	91.01	88.90	86.14	84.11
cuSparse+CSR	50.91	50.20	49.20	39.19	35.45	35.85
Sputnik	92.95	90.87	90.58	90.56	91.07	90.74

Table 2: Brach Efficiency of Sputnik and cuSparse

Table 2 shows the branch efficiency metric for the three sparse routines and different sparsity levels: the lower this metric, the more thread divergence. Thread efficiency is expected to worsen as the sparsity level increases. This effect is especially noticeable in cuSparse+CSR, which could explain the observations made in Figure 2. However, we observed that Sputnik yields relatively high ratios in this metric with some fluctuations for high sparsity levels. This is achieved with an internal workload balance strategy.



(a) Sputnik (b) cuSparse COO (c) cuSparse CSR

Figure 3: FMA instructions executed by Spunik, cuSparse and cuBlas (left y-axis) and speedups of the first two over cuBlas (right y-axis)

Figure 3 shows the evolution of the number of FMA instructions executed by each routine. Sputnik executes almost half as many FMAs as the dense version with a sparsity of 50%, near the theoretical instruction count reduction. Furthermore, this reduction evolves adequately with the sparsity level. Nevertheless, cuSparse+COO uses about twice the number of FMAs as the dense code at 50% of sparsity, the difference being even larger using CSR. The reason for the different behavior between both libraries is that in Sputnik, techniques like subwarp-tiling and Reverse Offset Memory Alignment (ROMA) [8] increase the opportunities to use vector memory instructions, which reduces the number of instructions required.

4.2.2 *Half precision.*

The same tests were conducted for the half precision routines, most of the conclusions being analogous to those in single precision. Due to space limitations, we briefly discuss the most important differences observed for half precision:

(1) L1 cache misses: The observations for Sputnik and cuSparse are equivalent but with larger differences w.r.t. cuBLAS. In Sputnik, the number of missed sectors is halved since we pass from single to half precision. In cuBlas, the value of this metric is reduced by eight on average because it benefits from data reuse, and it loads the information to SMEM directly using the new Ampere asynchronous copy instruction LDGSTS [26]. This instruction bypasses L1 and RF and copies data directly from L2 to SMEM.

(2) Math instructions executed: This metric drops dramatically in the case of cuBlas, achieving a lower number of math instructions executed than the sparse routines for all the considered sparsity levels. While in cuBlas multiple FMA instructions are fused into a single HMMA, executed in Tensor-Cores, both Sputnik and cuSparse continue to use the FPU. In Sputnik, this happens because internally it uses mixed-precision despite the input/output data being 16-bit floating-point values. Hence, Sputnik converts FP16 data to FP32, in order to reduce the accumulation error. Since cuSparse is not open-source, we cannot be sure of the reason behind its behavior, so we assume it to be the same as for Sputnik.

4.3 Challenges and opportunities

Sputnik showed the best hardware utilization among all the contenders of the SpMM routine, proving to scale relatively well as sparsity increases. Therefore, we will consider Sputnik as the pruning-agnostic SpMM reference for ML workloads. Based on the previous analysis, the next step consists in using hardware-aware pruning techniques to boost the performance of Sputnik without modifying the routine code, addressing, from the pruning technique perspective, different problems of this implementation:

- The L1 cache usage can be enhanced for lower sparsity levels. Although it evolves well with the sparsity, L1 misses are much higher than cuBLAS despite having a substantial theoretical operation count reduction w.r.t dense computation.
- The memory transactions efficiency can be improved in two different ways: (1) selecting the widest memory instructions whenever it is possible, and (2) alleviating the MIO (Memory Input/Output) instructions queue pressure by reducing the number of global memory requests.
- The thread load balance can be enhanced, as the irregularity of fine-grained-based sparse matrices prevents Sputnik's branch efficiency from getting higher ratios. Hence, despite implementing workload balance, non-hardware-optimized sparse matrices structure can severely limit its effectiveness in this metric.

Finally, the Sputnik's implementation for half precision presents a high math instruction count and this must be corrected at CUDA level. This issue will be addressed separately in Section 6.

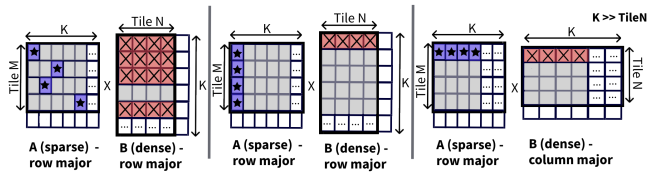


Figure 4: Different possibilities of non zeros distribution across a sparse matrix considering different storage formats on dense one

5 EXPLORING HARDWARE-OPTIMIZED SPARSE MATRICES FOR PRUNING-AGNOSTIC SPMM KERNELS

The shape and distribution of sparse matrices affects the performance of the SpMM operation. This section explores the use of existing hardware-aware pruning methods to improve the performance of Sputnik, a SOTA pruning-agnostic SpMM implementation.

5.1 Problem description

Let us first decide the matrices layout to be used. Figure 4 represents three situations that differ in the location, but not in the number, of non-zero values in the sparse matrix (A) and the corresponding positions affected in the dense input matrix (B). The situation represented on the left is cache-unaware, while the other two ones implement some kind of cache-awareness: the one in the middle uses a typical 1-D vertical vector tiling scheme called "column-vector" in the literature, while the rightmost one uses the equivalent "row-vector" scheme. The vector-based techniques differ in the order in which the dense matrix B is stored, row-major (middle) or column-major (right). Storing B in column-major order can maximize locality in tile traversals and mitigate the impact of non-coalesced or unaligned memory accesses when output matrix (C) is computed in column-major order. That is why a few previous works prefer the column-major order for the SpMM routine [18, 22]. However, as ML datasets and SOTA pruning-agnostic SpMM implementations rely on the row-major order, we selected this layout because it represents the most realistic and practical situation.

The microarchitecture study of the previous section identified data locality, memory efficiency, thread load balance, and math instruction count as the four most crucial factors for the performance of the SpMM routine. In this section we will cover the first three ones, making an in-depth study of how these hardware aspects can be improved by tuning the pruning configuration parameters, and following with the measurement of their performance impact.

5.2 Analysis of the pruning impact on data locality and memory transactions efficiency

This section covers the *data locality* and *memory transactions efficiency* aspects. We will explore the use of the column-vector pattern as a way to (1) improve the L1 cache usage and (2) reduce the number of GMEM requests per instruction. Concerning memory efficiency, we will cover the MIO instructions queue pressure reduction.

Terminology and scenarios. Figure 5 represents the behavior of the L1 and L2 caches within the scope of a warp when we

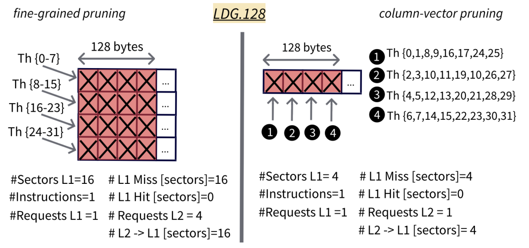


Figure 5: L1 cache behavior with different memory access patterns

pruned following a fine-grained (left figure) or a column-vector-based (right) policy. The two scenarios of this figure correspond to the first two situations shown in Figure 4, respectively. We assume that the widest GMEM instructions are used, LDG.128. Hence, in Figure 5, each 128-byte cache line is divided into four sectors (32 bytes). Remember that if the word size is larger than 4 bytes, each memory request is split into several 128-byte memory requests, that is, four memory requests issued *independently*, one per quarter-warp, for 128-bit instructions (float4 datatypes). In the left situation of the figure (fine-grained), each quarter-warp accesses a different row of a matrix, meaning that 8-thread groups access consecutive and independent 128-bytes in a fully coalesced manner. In the second situation (column-vector), the four quarter-warps access the same 128-byte line, distributing lanes across it equally balanced and coalesced. For each one of the two scenarios, the bottom part of the figure summarizes the final state of the microarchitecture: L1/TEX cache hits and misses, number of sectors, number of executed instructions, memory requests per instruction to L1 and L2, and number of sectors transferred from L2 to L1 cache, assuming that caches are initially empty.

Microarchitecture impact of fine-grained pruning. In the fine-grained sparse matrix, each warp retrieves 16 (= 4x4) sectors with a single LDG.128 instruction. This generates one request to L1 that results in 16 missed sectors since no data is cached. Next, there are four 128-byte memory requests to the L2 cache that result in a new cache miss each. Therefore, there are no hits in either L1 or L2, which means that the 16 sectors must be fetched from GMEM. The loaded data is cached in L1 and L2 since data must return to RF.

Microarchitecture impact of column-vector pruning. In the column-vector-based matrix, each quarter-warp accesses different parts of the same 4 sectors. As a result, after missing in L1, *a single request* is generated to L2 *instead of four*, thus reducing the memory pipeline pressure. A new cache miss is obtained, but only 4 sectors must be fetched from GMEM and returned to RF. Notice that we are not exploiting the L1 benefits yet (no cache hits). Instead, we are taking advantage of the GPU capability to detect four identical quarter-warp memory requests and perform just one of them (Sectors/Requests = 4). Thus, there is room for further improvement if we also exploit L1 data locality. This can be done, for instance, by: (1) using lower-width memory operations (e.g., residual elements) and processing rows iteratively, (2) using a smaller column-vector length than the number of rows processed by a warp (e.g., different groups of threads access the same GMEM elements but in a different order), (3) using a column-vector length larger than the number of rows processed by a warp (e.g., different warps sharing the same SM access the same elements). Thus, as we can see, there exists a set of tunable parameters that must be carefully and simultaneously selected on our pruning algorithms (e.g., vector length) and SpMM implementations (e.g., tiling scheme, memory instructions width, GPU thread-block scheduling policy). Next, let us go deeper into the symbiosis between the pruning technique and the SpMM kernel's configuration, taking Sputnik's code as a reference.

Dissecting Sputnik's tunability. Work distribution of rows among threads dramatically influences the final performance and must be carefully chosen. Furthermore, it indirectly affects the memory instructions width, one of the aspects to improve according to our preliminary study. To analyze this, let us consider as an example the three predefined thread-blocks that are configured in Sputnik: 32x1, 16x2 and 8x4 threads. The Subwarp Tiling technique breaks down each warp (32 threads) into different dimensions. The dense matrix tile size processed by each one of the previous configurations is, per K-dim iteration, following a *rows x columns* notation: 1x32, 2x32 and 4x32, respectively. This enables the usage of 4, 8 and 16-byte words to load those different tile sizes in a single instruction (LDG.32, LDG.64 and LDG.128, respectively). Hence, according to how local/global memory transactions work in GPUs (Figure 5), we should try to promote the usage of the 8x4 thread-block configuration since, it not only uses wider instructions but it also enables the usage of *longer column-vectors* inside a warp, which reduces the number of GMEM requests per instruction.

General guidelines on vector-length selection. There are two aspects of a CUDA kernel that can affect the performance of SpMM when using column-vector pruned matrices: (1) the number of rows of the sparse matrix that each thread-block will process (thread-block dimension) and (2) the dense matrix tile size; both in conjunction with the memory instructions width to be used. Note that (1) will determine the maximum length v of column-vectors within the *scope of a warp*. Thus, processing a single row of A with a 32x1 thread-block size and LDG.32 instructions will not imply an improvement of data locality despite considering column-vector pruning with $v > 1$. If two rows of A were processed using a block size of 16x2 and a single LDG.64, or if two rows were processed by a thread-block of size 32x1 with two sequential LDG.32 instructions, v could be extended up to 2. With a 8x4 thread-block size and a single LDG.128, v could be expanded up to 4. All these considerations are done to optimize memory transactions and L1 cache locality within a warp's scope. However, let us recall that in an Nvidia GPU the global L2 cache is shared among all the SMs, while the L1 is private to each one of them, meaning that it is only accessible from the threads belonging to the same SM. That means that v could be further extended beyond the warp's scope if we knew beforehand which thread-blocks will share the same SM, but that depends on the GPU thread-block scheduler policy.

Pushing vector-length to the limit. The policy of the Ampere thread-block scheduler is not public, so we have reverse-engineered it following the same approach as [34]. In general terms for our study case the expression that describes how thread blocks are assigned to SMs in the first wave, using CUDA-like notation, is:

$$SM(blockIdx) = \left(2\left\lfloor\frac{blockIdx}{2}\right\rfloor\right) \; mod\; 82 + \left\lfloor\frac{blockIdx}{82}\right\rfloor \; mod\; 2$$

where:

$$blockIdx = blockIdx.x + blockIdx.y * gridDim.x$$

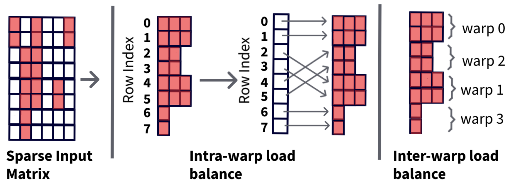


Figure 6: Load balance assuming a vector length of 2 elements

and where the 82 is the number of SMs in the RTX 3090 GPU. Overall, it corresponds to a block-cyclic distribution of the thread blocks on the SMs in blocks of two. Also, while the scheduler can behave slightly different depending on the SM's local resource availability when assigning thread-blocks to SMs [9], we have verified that this does not happen during the execution of sparse workloads using the Sputnik SpMM implementation.

As a result of this scheduler's knowledge, we can double the vector length defined in the previous section (to 8 in Sputnik), since two consecutive blocks are assigned to the same SM, sharing the same L1 cache space and opening up opportunities for inter-warp data reuse. Further increasing v could improve the locality of the L2 cache, but it should be considered whether the performance gain is worth the accuracy impact. The convenience of this L2 optimization will be discussed later.

5.3 Load Balancing

This section covers the *load balancing* problem detected in Section 4.3. We analyze the intra and inter-warp load imbalance as crucial factors for the performance of the SpMM routine since:

(1) **Intra-warp load imbalance** can damage the Instruction-Level-Parallelism since it generates divergence between threads, which causes hardware underutilization of resources and bandwidth both of the memory and compute pipelines.

(2) **Inter-warp load imbalance** can cause that some SMs are busy while others are idle, damaging the Thread-Level-Parallelism.

To tackle this, we defined a variant of the Row Swizzle technique [8]. The idea is to see the groups of rows that form the column-vectors as a block. Then, we assign a weight to each block based on the sum of its row lengths. This preserves the formed blocks and prevents destructuring them. Finally, we sort the row indices by block weight in decreasing order.

Figure 6 illustrates the aforementioned heuristic. It shows, from left to right, (1) an input sparse matrix in column-vector format using a vector length $v = 2$, (2) the application of our reorder technique variant to that input matrix and (3) the per-warp distribution of the row bundles, considering that each warp will process 2 rows.

This strategy enhances the inter-warp load balance since the first wave of blocks will be distributed cyclically across the SMs. Then, the remaining ones will be assigned to the idle SMs in decreasing order of weight. Regarding intra-warp load balance, the column-vector format ensures that all the threads within a warp will have the same number of elements to process, excluding residual values.

5.4 Hardware-optimized sparse matrices design

Our previous microarchitecture-level study helped us to:

(1) Define the *optimal range of vector lengths* v to maximize the hardware utilization according to our SpMM kernel configuration. For instance, in Sputnik, v should take the values 2 and 4. A v value of 4, 128-bit instructions and an $8x4$ thread-block size is expected to provide the best performance in a warp's scope.

(2) Reverse-engineer the Ampere scheduler and suggest doubling v (to 8) in order to push the L1 usage to the limit.

(3) Advise that further increasing the v value may enhance the L2 cache usage, although we must check whether the performance gain is worth it depending on the accuracy impact it may imply.

(4) Propose a load balancing technique to improve the intra and inter-warp load balance.

Section 5.5 will rely on the contributions above to configure hardware-aware pruners that raise the performance of pruning-agnostic SpMM kernels, taking Sputnik as a reference.

5.5 Performance evaluation on hardware-optimized sparse input matrices

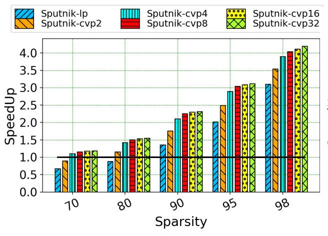
We first evaluate the performance of using matrices generated with the column-vector pruning technique on pruning-agnostic SpMM kernels, taking Sputnik as the reference SpMM implementation. The evaluation is based on the same ResNet-50 architecture trained on ImageNet used in Section 4.

Column-vector pruning is applied with five vectors lengths v: 2, 4, 8, 16 and 32. Figure 7 shows the speedups of Sputnik w.r.t. cuBLAS using this kind of matrices as inputs for different sparsity levels and for single (left) and half (right) precision. "Sputnik-lp" is the result of using hardware-unaware sparse matrices generated by a variant of magnitude-pruning called level-pruning [41]. Instead of receiving a threshold magnitude α, it gets a target sparsity level to accomplish and automatically selects the α value accordingly. This case is a baseline to put into perspective the performance gain obtained using each vector length.

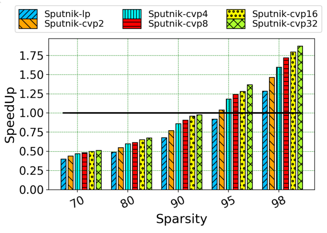
For single precision (Figure 7a), the vectors lengths 2, 4 and 8 are the most interesting choices, as they keep the accuracy of the network at reasonable levels, while being up to 32%, 62%, and 71% faster, respectively, than level pruning. For vector lengths 16 and 32, the loss in accuracy is not compensated by a significant performance gain. The stabilization of this performance gain when $v > 8$ confirms our hypothesis that increasing v beyond that value will not improve the L1/TEX cache usage.

Notice also that using a sparse matrix pruned following a column-vector approach with a vector length $v > 2$ and a 80% of sparsity performs better than a non-hardware-optimized matrix pruned with a 90% sparsity. Furthermore, in that specific situation, for $v = 4$ the accuracy achieved is 0.2% worse than using a fine-grained approach. This could imply that pruning less but following a column-vector strategy can yield a better performance-accuracy trade-off than pruning more using a fine-grained strategy. However, a systematic validation of this hypothesis is out of the scope of this paper.

Figure 7b shows the results for half precision. The main difference w.r.t. to single precision is that the performance gain is more limited, since Sputnik uses FPU instead of TCU for half precision, with the already mentioned limitations that it imposes. Furthermore, as will be discussed later, hardware-optimized sparse matrices cannot avoid this effect since this can only be done by changing the underlying kernel implementation (Section 8.1). However, the other conclusions of the analysis of Figure 7a apply to this one.



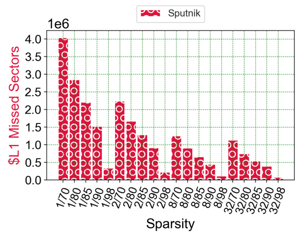
(a) Single precision



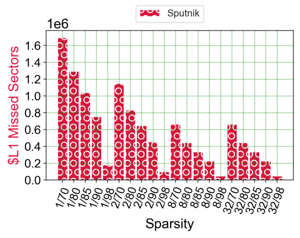
(b) Half precision

Figure 7: Speedup of Sputnik over cuBlas for different fine-grained based input matrices ("Sputnik-lp") and column-vector based ones of different v lengths ("Sputnik-cvp$v$").

Now, we proceed to verify that the observed behavior matches some of the microarchitecture effects hypothesized through this section. Figure 8 shows the evolution of the number of L1/TEX missed sectors. The less relevant v values are omitted to simplify the figure. In single precision, shown in Figure 8a, the number is almost halved when moving from fine-grained pruning to column-vector pruning with $v = 2$. For example, for 70% of sparsity, it is reduced from $4e6$ missed sectors to almost $2e6$. For $v = 8$, the reduction is almost $4x$; around $1e6$ for 70% of sparsity. Confirming previous observations, we can see that the reduction is negligible between $v = 8$ and $v = 32$. The same observations apply to the half precision case (Figure 8b).



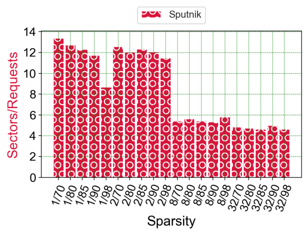
(a) Single precision



(b) Half precision

Figure 8: Number of L1/TEX missed sectors for different vector lengths and sparsity levels. X-axis labels follow the notation v/sparsity[%], v being the vector length and sparsity[%] the sparsity level. The case $v = 1$ corresponds to fine-grained pruning.

Figure 9 shows the evolution of the "Sectors-to-Request ratio" metric for different vector sizes and sparsity levels. The initial value is around 16 (residue values preclude reaching the best theoretical value) and it begins to converge to 4 as $v \geq 4$, where it stabilizes. The reasons for this behavior have already been discussed in Figure 5. Table in Figure 9 shows the reduction of L1 and L2 missed sectors w.r.t the usage of $v = 1$. The L1 missed sectors metric clearly drops until when $v = 8$, where it stabilizes, which matches previous observations. The evolution of the L2 missed sectors metric is the opposite. Let us recall that the L2 cache is shared by all the SMs. Thus, for a vector length above 8, if two blocks access the same global memory position despite being assigned to different SMs, they can still find that information in L2, and that is why L2 cache gets more relevant for $v > 8$. This is one of the reasons behind the small performance gain for vector lengths of 16 and 32 in Figure 7.



	L1	Misses	L2	Misses
V/Precision	Single	Half	Single	Half
2	-41.56%	-35.57%	-0.03%	-0%
4	-66.04%	-60.68%	-0.20%	-0.56%
8	-74.99%	-65.51%	-2.26%	-5.02%
16	-74.65%	-65.64%	-6.87%	-16.76%
32	-74.34%	-65.56%	-18.27%	-31.44%

Figure 9 & Table 3: On the left, the Sectors/Requests ratio following the same notation as Figure 8. On the right, the reduction percentage of L1 and L2 missed sectors w.r.t. the baseline $v = 1$.

Table 4 shows the evolution of the branch efficiency metric for different vector lengths. For single and half precision, the larger the vector the higher the branch efficiency. The biggest increase happens when going from $v = 1$ to $v = 4$, that is, within the scope of a warp (**intra-warp load balance**). Remember that with column-vector-based sparse matrices, if $v = 4$ and 128-bit memory instructions are used, each quarter-warp processes exactly the same number of elements. These considerations are equivalent for single and half precision. However, residual values can slightly degrade the theoretical efficiency ratio and half precision kernels seem to be more affected by this. When $v > 4$ we can appreciate a more modest increase in branch efficiency. This is because once we are out of the scope of a warp (**inter-warp load balance**), i.e. $v \geq 8$, blocks of threads with the same workload will be generated; however it demonstrates to have a smaller impact.

v	1	2	4	8	16	32
Single	94.86	96.55	99.19	99.80	99.86	100
Half	93.88	94.62	95.48	95.88	96.10	95.40

Table 4: Brach Efficiency percentage for single and half precision

Conclusions. This study and its associated experiments prove our hypothesis that using hardware-optimized matrices on pruning-agnostic SpMM kernels can raise performance. However, this requires the parameters of the pruning technique and the SpMM implementation to be carefully tuned to maximize hardware utilization. We have proved that our guidelines can help to appropriately tune these parameters, and that our microarchitecture analysis can explain the performance variations observed in the experiments.

6 CLASP: COLUMN-VECTOR PRUNING-AWARE SPMM KERNEL

One way to *fully prioritize* the performance of sparse routines is to design specialized kernels that take advantage of the knowledge of the pruning technique used to generated the input sparse matrix (e.g. column-vector). An implementation of this kind of the SpMM routine has been proposed targeting the Volta Nvidia architecture [4], but it does not support the newest Ampere generation. The reason is that the Volta implementation extensively relies on the *mma.m8n8k4* instruction forming a mapping between the warp tile and the TCU at Octet thread level, while the Ampere architecture does not support the $8x8x4$ matrix instruction size on 16-bit precision. As a result, when this Volta-ready implementation is used on Ampere, each *mma.m8n8k4* is decomposed into multiple FMA instructions, yielding an important performance drop.

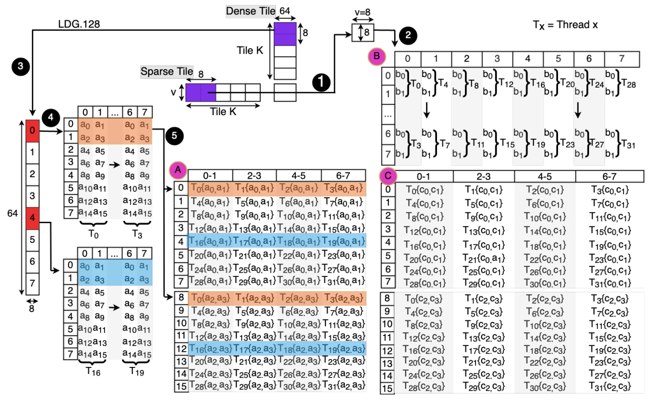


Figure 10: SpMM kernel design for column-vector input matrices

We propose an Ampere-ready kernel design based on the new mapping shown in Figure 10, which can be implemented using the $16x8x8$ matrix instructions size, available on Ampere. It also removes Sputnik's limitation related to FPU usage on FP16 precision. This implies a reduction of the generated SASS code from 4328 lines using the Volta-ready implementation to just 512. Each block of the dense tile (middle top of the figure) is composed by $64x8$ elements, while for the sparse matrix tile the block size is $vx8$ (left and below the dense tile), v being the vector length. The first step 1 loads the sparse matrix blocks from GMEM to SMEM. Each thread loads 8 continuous elements of the sparse matrix, which enables the usage of LDG.128 instructions. The storage to SMEM 2 is done using STS.128 instructions. At this point, all the threads in a warp retrieve their part of the data from shared memory to registers following the thread mapping represented in B. Note that the loaded sparse block will change from being the LHS (Left-Hand-Side) fragment on SpMM to be the RHS (Right-Hand-Side) one in the TCU scope. For this reason, it is stored in shared memory as a column-major matrix, to match the initial data layout order.

Step 3 consists in moving the dense block directly to registers using two LDG.128 instructions. Each warp processes a complete tile in Tile_K/8 steps. Next, step 4 shows how the different lanes are mapped to the $64x8$ dense block in order to collaborate in the load, maximizing memory bandwidth. As can be seen, 16 elements are assigned to each thread, which are processed in 4 rounds of $16x8x8$ each. For each one of those steps 5, (A) represents the LHS fragment in TCU with the corresponding thread mapping.

Finally each thread stores its partial results using two STG.128 instructions following C's mapping, so that memory bandwidth is maximized in every single access to global memory.

7 AD-COLPRUNER: ADAPTIVE COLUMN-VECTOR PRUNER

While column-vector pruning can generate hardware-optimized sparse matrices that improve the performance of the SpMM routine, it can also have a negative effect on accuracy. Some of the constraints that the original column-vector pruning imposes are: the vector length is fixed and it is picked among a limited set of "good" values (e.g. 2,4,8), all the elements within a vector must be non-zeros and the vectors must be aligned w.r.t. the vector length.

Algorithm 1 Sparse Mask selection.

```
1:  Inputs: layer $\alpha$ to be pruned, list $\tau$ of triplets $(k,v,p)$ to guide
    the pruning, and number $nnz$ of $\alpha$'s weights to preserve.
2:                    ▷ $\tau$ is sorted in decreasing order of $k$'s length
3:  for each $(k_i, v_i, p_i) \in \tau$ do
4:      $nnz\prime \leftarrow p_i * nnz$          ▷ elements belonging to $\tau_i$ config.
5:      $\delta^{\theta} \leftarrow$ Set of column-vectors from $\alpha$ of size $k_i$, aligned, and
        non-overlapped
6:      for each $vector \in \delta^{\theta}$ do
7:          Calculate the sum of $v_i$ largest elements (absolute value)
8:      end for
9:      Pick the best $nnz\prime / v_i$ vectors and flag their $v_i$ selected
            weights to avoid considering them again
10:     Update binary mask values
11: end for
```

This paper proposes an intermediate approach between fine-grained and column-vector pruning that allows a better balance between performance and accuracy. This approach is based on (1) grouping the sparse matrix in vectors but with a non-constant length, where (2) not necessarily all the elements within a vector must be non-zeros and thus, enabling the appearance of (3) unaligned sub-column-vectors.

One of the advantages of pruning-agnostic kernels is that they do not assume a specific distribution of the non-zero elements, thus supporting more flexible column-vector variants that balance performance and accuracy such as the one we propose.

The aforementioned relaxed constraints of our approach allow us to initially pick virtual windows of k consecutive elements and then to keep just the v more significant ones, even if they are not consecutive. Notice that $v \leq k$. This process can be applied iteratively for different pairs of k and v, which can give place to vectors of different sizes v within the same pruned sparse matrix.

The implementation of such a technique starts with a pruning configuration consisting on several triplets of the form (k_i, v_i, p_i), where for the $i-th$ iteration of the process, k_i is the length of each initial virtual window of consecutive elements, v_i is the number of possibly non-consecutive elements selected among those k_i, and p_i is the percentage of the dense elements of the sparse matrix picked with a k_i and v_i combination. This process is applied iteratively for all the triplets. Algorithm 1 presents the pseudocode for this idea.

8 EVALUATION

This section evaluates the performance of CLASP on different DL workflows and assesses the ad-colPruner method on real-world computer vision models.

8.1 Ampere-ready pruning-aware CLASP kernel

Figure 11 shows the speedup achieved by the original Sputnik implementation using our hardware-aware pruning guidelines, the original non-Ampere-ready *VectorSparse* proposal [4], our CLASP approach, and cuSparseLt, the baseline being cuBLAS. For this evaluation, problems extracted from two real-world applications are considered, one from computer vision (ResNet-50 on the ImageNet dataset) and the other from natural language processing (Transformer on the WMT English-German 2014 dataset).

Excepting cuSparseLt, which follows a 2 : 4 pattern, the input data are column-vector-based sparse matrices. Two different numbers of columns N for the dense matrix are considered for ResNet-50. For Transformers only the $N = 256$ case was tested, as it is the most common one and this makes the figure more readable. Let us remember that custom SpMM implementations supporting column-vector sparse encoding must set a *private vector length* v. Our implementation supports three different values for v: 2, 4 and 8. Hence, the input matrices must be generated using the same vector lengths t, that is $t = 2, 4$ and 8. However, since v is private to warp-level, we can further extend t to 16 in order to improve the L1/TEX locality when $v = 8$, as two consecutive thread-blocks will be assigned to the same SM, according to the Ampere scheduler policy. We have also considered $t = 32$ to evaluate the impact on the L2 and load balance. In order to support matrices with $t = 16, 32$ despite not having a 1:1 relation in our custom implementation, they have been encoded as matrices with $t = 8$.

Note that the number of matrices with different shapes is larger in ResNet-50 than in Transformers. That is why the results show a large variability in ResNet-50 but not in Transformer.

The results show that the original vector-sparse implementation [4] only performs better than the dense routine in Ampere when a 98% of sparsity is reached on ResNet-50, and 95% on Transformer. The reason is that, since *mma.m8n8k4* instructions do not exist in Ampere for half precision, they are decomposed into multiple FMAs, increasing the total number of instructions and promoting L0 cache instruction overflow.

Despite using the new SPTCU units, cuSparseLt achieves a poor performance in our experiments. We found that this library is memory bound and it requires larger matrix sizes [24] to achieve performance gains. The reason is that the memory pipeline is much more heavily used than the compute one because of inefficient accesses to L1 and L2, representing a bottleneck in the computation. Increasing the GEMM size relaxes this situation, as memory accesses are better overlapped with computation. While batch computation could be used to increase the problem size, this is out of the scope of this paper. Furthermore, this workaround is not always possible; for instance, in online inference applications (e.g., self-driving cars), whose batch size is usually 1.

We can see that our implementation outperforms the original Sputnik implementation for column-vector-generated input matrices, and that the performance difference increases with the value of v. The performance of our implementation surpasses the dense one from sparsity levels of 70%.

While our implementation does not support $v = 16$ and $v = 32$, we demonstrate that our guideline to generate hardware-aware sparse matrices can also be beneficial in ad-hoc solutions introducing a new abstraction layer: $t = 16$ significantly increases the performance obtained since the L1 cache usage is improved by two consecutive blocks of threads that share the same SM. Furthermore, the use of $t = 32$ only benefits the L2 cache, which has a significantly lower impact on performance.

8.1.1 Portability of CLASP. Although the main target of this work is the latest Nvidia Ampere architecture, CLASP can also be used in other GPU architectures supporting the $16x8x8$ matrix instruction size available from Turing. Hence, we have replicated our

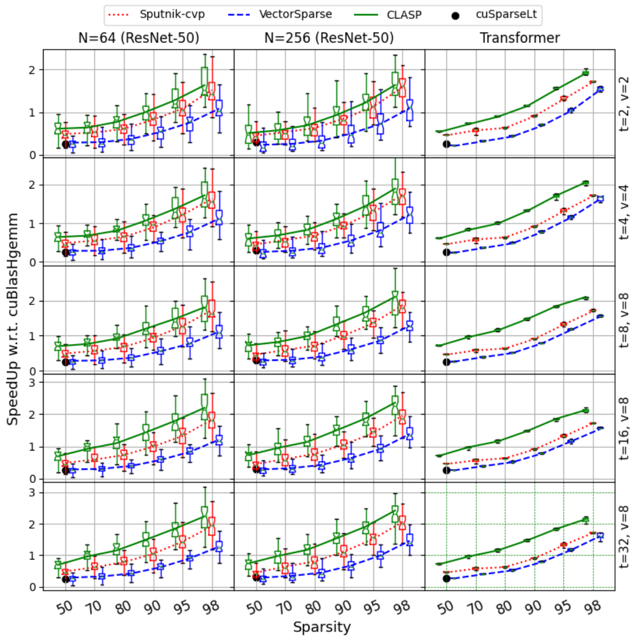


Figure 11: Sputnik using hardware-aware pruning vs. VectorSparse vs. CLASP vs. cuSparseLt

previous experiments on a Nvidia RTX 2080Ti Turing GPU. The results reported a speedup of up to 2.67× on ResNet-50, slightly larger than the one obtained on Ampere. Also, CLASP obtained on Turing a speedup of up to 3.50× for Transformers, while it was around 2.20× on Ampere. Furthermore, on Turing we get this performance from a vector length greater than 4, yielding an incredible performance boost from lower vector sizes. That translates into a performance improvement on Turing of around 60% w.r.t. Sputnik using hardware-optimized sparse matrices. The reason is that Turing's architecture presents a lower ratio of computation capability to memory bandwidth, which relaxes the data reuse demand.

8.2 ad-colPruner application: Computer Vision

A pruning-aware kernel is the best option to prioritize performance. However, if we want to balance accuracy and performance, our proposal of relaxing the constraints of the column-vector pruning technique (Section 7) can be a good choice. We evaluated ad-colPruner using two datasets and two Pytorch pre-trained dense models widely used in computer vision problems: (1) CIFAR-10 dataset with a ResNet-20 model (Top-1 accuracy 92.54%) and (2) ImageNet with ResNet-50 (Top-1 accuracy 76.2%).

Following the triplet notation presented in Section 7, Figures 12 and 13 show the results in terms of Top-1 error and execution time latency for single (left) and half precision (right) using different combinations of virtual window lengths k, number of non-zeros v to pick from those vectors and the percentage p of the total *nnz* elements assigned to a combination of k and v. For example, $k8v4p30 + k4v2p70$ uses a sparse matrix where 30% of the *nnz* elements were selected from the virtual column-vectors of size 8 that contain the largest 4-elements sum (absolute value) while the other 70% were selected from those virtual column-vectors of size 4 with the largest 2-element sum. Recall that the v elements are

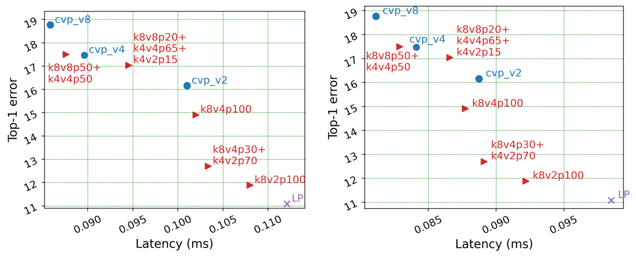


Figure 12: Accuracy and Latency on ResNet-20 at 90% of sparsity

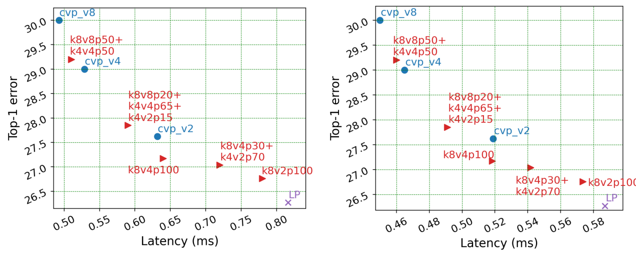


Figure 13: Accuracy and Latency on ResNet-50 at 90% of sparsity

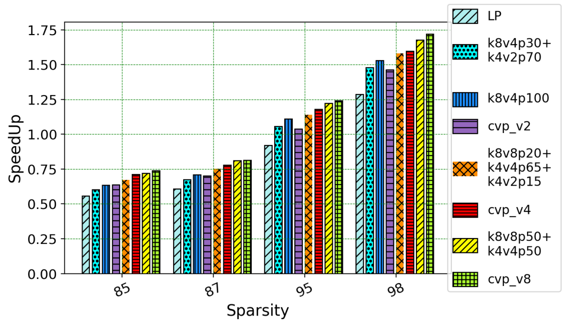


Figure 14: ad-colPruner sparsity sensitivity on performance

chosen with no location constraints within a virtual window of k consecutive elements. Vectors of size k are aligned and not overlapped. Original column-vector pruning (cvp) with vector lengths $v = 2, 4$ and 8 and fine-grained pruning using the level-pruning (lp) algorithm (magnitude pruning variant) are also represented in these figures. Models have been pruned to a 90% of sparsity.

The ad-colPruner configurations follows the [7] heuristic that achieves SOTA performance using magnitude pruning. Thus, ad-colPruner also leaves the first convolutional layer fully dense and restricts the final fully connected layer sparsity. We have verified that [7] approach offers an effective solution for group-level pruning and can preserve an almost homogeneous sparsity level across the remaining layers.

The results show that our balanced pruning proposal can increase the design space of existing pruning techniques: both for ResNet-20 and ResNet-50, the newly proposed combinations (in red) are able to fill the intermediate points in the accuracy-to-performance map, providing intermediate solutions to achieve more balanced alternatives. Moreover, what is even more relevant, some hybrid configurations are able to not only reduce the Top-1 error with respect to cvp, but also to reduce the execution time. This happens, for example, in the $k8v4p100$ configuration compared to cvp_v2: while for single precision $k8v4p100$ achieves better accuracy in the two considered models with a similar execution latency, for half precision the execution time performance is even better than using cvp_v2, specially in ResNet-20. **The general conclusions are**:

- Using a pruning-unaware SpMM implementation and transferring our hardware knowledge to the pruning phase can meaningfully improve performance and give more flexibility on weight selection than using a custom implementation.
- When a warp processes a subset of 4 consecutive rows (a quarter-warp per row), and $v < 4$ is considered, non-zero elements do not need to be in contiguous positions, since this will not enhance data locality within that scope. The important point is having the maximum possible number of non-zero elements per column within that subset of 4 consecutive rows, independently of their row position (GMEM instructions count reduction).
- Not selecting weights as described in the previous point would lead to extra intra-warp load imbalance, since central rows will tend to accumulate more non-zero elements.
- The same idea can be applied to $k = 8$ with $v < k$ since two consecutive thread-blocks will share the same SM and hence, the same L1 cache.
- A large k with a small v gives more flexibility on weight selection, sacrificing performance. A v close to k yields a better performance since data locality and load balance will improve.

8.2.1 Discussion. We discuss here the sparsity and pruning sensitivity of our approach. The sparsity sensitivity of ad-colPruner was analyzed by modifying the sparsity level and the previously described pruning configurations. Observing the general trend as sparsity increases, we found that the trade-off between accuracy and performance obtained with our approach can be even better for other sparsity levels, including its ability to outperform some column-vector configurations (such as cvp_v2) even in absolute terms. As an example of this, Figure 14 shows the speedups obtained w.r.t. cuBLAS for half-precision on ResNet-50. The accuracy aspect is not either affected by the sparsity variation. The points representing the performance-accuracy relationships can be more closer or distanced depending on the sparsity. The lower the sparsity, the closer are the points corresponding to column-vector and level-pruning configurations, and viceversa. This means that high sparsity levels widen the design space, which can be an interesting property. Thus, we conclude that our method is not only robust to sparsity variability, but it can also increase further the design space and improve the column-vector pruning performance on more flexible weight selection policies.

Regarding the impact of varying the pruning method, we found that while magnitude-pruning and its variants can generate almost constant sparsity levels for each layer with excellent performance-accuracy tradeoffs, other techniques like variational dropout lead to layers with a big sparsity variability (values between 0 to 100%) that, on average, fulfill the target sparsity, which can degrade the final performance. Thus, considering these results and the already demonstrated ability of magnitude pruning to achieve the same accuracy versus sparsity trade-off on ResNet-50 and Transformer architectures [7], we finally chose this method.

9 CONCLUSIONS

This work tested the idea of how tuning a hardware-aware pruning technique can raise generic SOTA SpMM implementations' performance for DL workloads. We experimentally proved this hypothesis through exhaustive benchmarking, also identifying the hardware causes that explain the performance fluctuations. This combination of experimentation and hardware behavior analysis certifies the robustness of our conclusions. The results show how hardware-optimized sparse matrices can lead existing pruning-unaware implementations to be up to 49% (for half precision) and 77% (single) faster without modifying their implementation. We have also presented a new Ampere-ready SpMM kernel based on the column-vector format that achieves up to 2.42x speedup over cuBlas in half precision. Finally, we propose a novel hardware-aware pruning technique, ad-colPruner, that balances performance and accuracy. It enables hybrid configurations that are more accurate and faster than the original column-vector-based sparse matrices.

ACKNOWLEDGMENTS

This research was supported by the Ministry of Science and Innovation of Spain (PID2019-104184RB-I00, AEI, 10.13039/501100011033), the Ministry of Education (predoctoral grant of Roberto L. Castro, FPU19/03974), and by Xunta de Galicia under the Consolidation Program of Competitive Reference Groups (ED431C 2021/30). We also acknowledge the support from CITIC, funded by Xunta de Galicia and FEDER funds of the EU (Centro de Investigación de Galicia accreditation 2019-2022, ED431G 2019/01). Finally, we acknowledge the Centro de Supercomputación de Galicia (CESGA) for the use of their computers.

REFERENCES

[1] Paresh Kharya Ali Alvi. 2021. Using DeepSpeed and Megatron to Train Megatron-Turing NLG 530B, the World's Largest and Most Powerful Generative Language Model. Retrieved December 3, 2021 from https://www.microsoft.com/en-us/research/blog/using-deepspeed-and-megatron-to-train-megatron-turing-nlg-530b-the-worlds-largest-and-most-powerful-generative-language-model/

[2] Sajid Anwar, Kyuyeon Hwang, and Wonyong Sung. 2017. Structured pruning of deep convolutional neural networks. *ACM Journal on Emerging Technologies in Computing Systems (JETC)* 13, 3 (2017), 1–18.

[3] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. arXiv:2005.14165 [cs.CL]

[4] Zhaodong Chen, Zheng Qu, Liu Liu, Yufei Ding, and Yuan Xie. 2021. 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 (SC '21)*. Association for Computing Machinery, New York, NY, USA, Article 78, 14 pages. https://doi.org/10.1145/3458817.3476182

[5] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cudnn: Efficient primitives for deep learning. arXiv:1410.0759 [cs.NE]

[6] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. ImageNet: A large-scale hierarchical image database. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. IEE, Miami, FL, USA, 248–255.

[7] Trevor Gale, Erich Elsen, and Sara Hooker. 2019. The State of Sparsity in Deep Neural Networks. arXiv:1902.09574 [cs.LG]

[8] 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)*. IEEE Press, Atlanta, Georgia, 1–14.

[9] Guin Gilman, Samuel S. Ogden, Tian Guo, and Robert J. Walls. 2021. Demystifying the Placement Policies of the NVIDIA GPU Thread Block Scheduler for Concurrent Kernels. *SIGMETRICS Perform. Eval. Rev.* 48, 3 (mar 2021), 81–88. https://doi.org/10.1145/3453953.3453972

[10] Google Research. 2020. Deep Learning Matrix Collection. Retrieved December 3, 2021 from https://github.com/google-research/google-research/tree/master/sgk

[11] Yiwen Guo, Anbang Yao, and Yurong Chen. 2016. Dynamic Network Surgery for Efficient DNNs. In *Proceedings of the 30th International Conference on Neural Information Processing Systems* (Barcelona, Spain) *(NIPS'16)*. Curran Associates Inc., Red Hook, NY, USA, 1387–1395.

[12] Song Han, Junlong Kang, Huizi Mao, Yiming Hu, Xin Li, Yubin Li, Dongliang Xie, Hong Luo, Song Yao, Yu Wang, et al. 2017. Ese: Efficient speech recognition engine with sparse lstm on fpga. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. Association for Computing Machinery, New York, NY, USA, 75–84.

[13] Song Han, Jeff Pool, John Tran, and William J. Dally. 2015. Learning Both Weights and Connections for Efficient Neural Networks. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1* (Montreal, Canada) *(NIPS'15)*. MIT Press, Cambridge, MA, USA, 1135–1143.

[14] Yihui He, Xiangyu Zhang, and Jian Sun. 2017. Channel pruning for accelerating very deep neural networks. In *Proceedings of the IEEE international conference on computer vision*. IEEE, Venice, Italy, 1389–1397.

[15] Torsten Hoefler, Dan Alistarh, Tal Ben-Nun, Nikoli Dryden, and Alexandra Peste. 2021. Sparsity in Deep Learning: Pruning and growth for efficient inference and training in neural networks. *Journal of Machine Learning Research* 22, 241 (2021), 1–124.

[16] Itay Hubara, Brian Chmiel, Moshe Island, Ron Banner, Joseph Naor, and Daniel Soudry. 2021. Accelerated sparse neural training: A provable and efficient method to find n: m transposable masks. *Advances in Neural Information Processing Systems* 34 (2021), 21099–21111.

[17] François Lagunas, Ella Charlaix, Victor Sanh, and Alexander M Rush. 2021. Block pruning for faster transformers. arXiv:2109.04838 [cs.LG]

[18] Kwangbae Lee, Hoseung Kim, Hayun Lee, and Dongkun Shin. 2020. Flexible group-level pruning of deep neural networks for on-device machine learning. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE Computer Society, Grenoble, France, 79–84.

[19] Mingbao Lin, Yuchao Li, Yuxin Zhang, Bohong Chen, Fei Chao, Mengdi Wang, Shen Li, Jun Yang, and Rongrong Ji. 2021. 1×N Block Pattern for Network Sparsity. arXiv:2105.14713 [cs.CV]

[20] Shiwei Liu, Tianlong Chen, Xiaohan Chen, Li Shen, Decebal Constantin Mocanu, Zhangyang Wang, and Mykola Pechenizkiy. 2022. The unreasonable effectiveness of random pruning: Return of the most naive baseline for sparse training. arXiv:2202.02643 [cs.LG]

[21] 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:1705.08922 [cs.LG]

[22] Atefeh Mehrabi, Donghyuk Lee, Niladrish Chatterjee, Daniel J Sorin, Benjamin C Lee, and Mike O'Connor. 2021. Learning Sparse Matrix Row Permutations for Efficient SpMM on GPU Architectures. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, Stony Brook, NY, USA, 48–58.

[23] Paulius Micikevicius. 2012. GPU Performance Analysis and Optimization. Retrieved April 20, 2022 from https://on-demand.gputechconf.com/gtc/2012/presentations/S0514-GTC2012-GPU-Performance-Analysis.pdf

[24] Asit Mishra, Jorge Albericio Latorre, Jeff Pool, Darko Stosic, Dusan Stosic, Ganesh Venkatesh, Chong Yu, and Paulius Micikevicius. 2021. Accelerating sparse deep neural networks. arXiv:2104.08378 [cs.LG]

[25] Deepak Mittal, Shweta Bhardwaj, Mitesh M Khapra, and Balaraman Ravindran. 2018. Recovering from random pruning: On the plasticity of deep convolutional neural networks. In *Proceedings of the IEEE Winter Conference on Applications of Computer Vision (WACV)*. IEEE, Lake Tahoe, NV, USA, 848–857.

[26] Nvidia. 2021. NVIDIA A100 Tensor Core GPU Architecture. Retrieved April 23, 2022 from https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf

[27] Nvidia. 2021. Nvidia Ampere GA102 GPU architecture. Retrieved February 1, 2022 from https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.pdf

[28] Nvidia. 2022. cuSparse Library. Retrieved February 2 2022 from https://docs.nvidia.com/pdf/CUSPARSE_Library.pdf

[29] Nvidia. 2022. cuSPARSELt: A High-Performance CUDA Library for Sparse Matrix-Matrix Multiplication. Retrieved February 2, 2022 from https://docs.nvidia.com/cuda/cusparselt/index.html

[30] Nvidia. 2022. Nsight Compute. Retrieved February 2, 2022 from https://docs.nvidia.com/nsight-compute/NsightCompute/index.html

[31] Carl Edward Rasmussen and Zoubin Ghahramani. 2000. Occam's Razor. *Advances in neural information processing systems* 13 (2000), 294–300.

[32] Masuma Akter Rumi, Xiaolong Ma, Yanzhi Wang, and Peng Jiang. 2020. Accelerating sparse cnn inference on gpus with performance-aware weight pruning.

In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*. Association for Computing Machinery, New York, NY, USA, 267–278.

[33] Victor Sanh, Thomas Wolf, and Alexander Rush. 2020. Movement pruning: Adaptive sparsity by fine-tuning. *Advances in Neural Information Processing Systems* 33 (2020), 20378–20389.

[34] Sreepathi Pai. 2014. How the Fermi Thread Block Scheduler Works. Retrieved February 3, 2021 from https://www.cs.rochester.edu/~sree/fermi-tbs/fermi-tbs.html

[35] Wei Sun, Aojun Zhou, Sander Stuijk, Rob Wijnhoven, Andrew O Nelson, Henk Corporaal, et al. 2021. DominoSearch: Find layer-wise fine-grained N: M sparse schemes from dense neural networks. *Advances in Neural Information Processing Systems* 34 (2021), 20721–20732.

[36] Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. 2016. Learning structured sparsity in deep neural networks. *Advances in neural information processing systems* 29 (2016), 2074–2082.

[37] Da Yan, Wei Wang, and Xiaowen Chu. 2020. Demystifying Tensor Cores to Optimize Half-Precision Matrix Multiply. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, Portland, Oregon USA, 634–643. https://doi.org/10.1109/IPDPS47924.2020.00071

[38] Da Yan, Wei Wang, and Xiaowen Chu. 2020. *Optimizing Batched Winograd Convolution on GPUs*. Association for Computing Machinery, New York, NY, USA, 32–44. https://doi.org/10.1145/3332466.3374520

[39] Xuda Zhou, Zidong Du, Qi Guo, Shaoli Liu, Chengsi Liu, Chao Wang, Xuehai Zhou, Ling Li, Tianshi Chen, and Yunji Chen. 2018. Cambricon-S: Addressing irregularity in sparse neural networks through a cooperative software/hardware approach. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE Press, Fukuoka, Japan, 15–28.

[40] 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*. Association for Computing Machinery, New York, NY, USA, 359–371.

[41] Neta Zmora, Guy Jacob, Lev Zlotnik, Bar Elharar, and Gal Novik. 2019. Neural Network Distiller: A Python Package For DNN Compression Research. arXiv:1910.12232 [cs.LG]