Profiler-Gated Fusion of V:N:M Sparse Tensor-Core Kernels for By-Construction Non-Regressive CNN Speedups

Jayden Vap
University of Notre Dame
South Bend, IN, USA
jvap2@nd.edu

Peter Kogge
University of Notre Dame
South Bend, IN, USA
kogge@nd.edu

Reem Shehayib
University of Notre Dame
South Bend, IN, USA
rshehayi@nd.edu

Abstract

The high sparsity ratios reported by modern pruning methods for neural nets routinely fail to translate into wall-clock speedups on commodity hardware, because unstructured zeros still stream through a dense compute pipeline. NVIDIA GPUs provide sparse tensor cores that execute a fixed 2:4 structured pattern at up to 2× the dense rate for the same size tile. The vector-structured V:N:M format extends this to arbitrary *N*:*M* ratios (e.g., 2:16). However, published V:N:M kernel libraries expose only a dense-input GEMM for linear layers. For computer vision networks, naively lowering every convolution in a network to a sparse GEMM makes tangible performance gains uncertain. This paper first presents a fused im2col-sparse-GEMM kernel that folds patch gathering directly into the kernel's global-to-shared-memory load so no im2col matrix is ever materialized, extending V:N:M sparse execution to stride-1 convolutions. Secondly, we provide a simple per-layer profiler that measures each convolution against its dense cuDNN baseline at calibration time and routes a layer through the sparse kernel only when it wins, falling back to dense otherwise. On an RTX 4080 with 2:16 (87.5%) V:N:M-sparse CNNs, we measure end-to-end speedups ranging from 1.00× (narrow-channel networks, fully falling back to dense) to 1.13× (wide-channel VGG-19), with only slight accuracy loss.

Keywords

GPU, sparse CUDA, structured sparsity, 2:4 sparsity, computer vision

1 Introduction

Unstructured magnitude pruning can remove a large portions of a CNN's weights while preserving accuracy, but on its own that reported sparsity does not make the network faster. A general-purpose matrix multiply on standard inference hardware platforms, e.g. CPU, GPU, or through an inference runtime such as ONNX still streams and multiplies the zeros. Consequently, this causes unchanged floating-point operation count, latency, and energy, and therefore no significant benefits from the induced sparsity. To reap an actual wall-clock benefit from sparsity requires either restructuring networks for dense hardware to skip over sparsity, or specialized hardware designed to skip zeros.

Modern GPUs are starting to supply sparse support. NVIDIA's Ampere- and Ada-generation GPUs provide *sparse tensor cores* that execute a fixed 2:4 weight pattern (two non-zeros per group of four) at up to twice that of the dense throughput. AMD's Matrix Core Technology [1] supports similar technology.

The vector-structured V:N:M format [3] abstracts this hardware-based 2:4 ratio to arbitrary *N*:*M* block sparsity by letting a block of output rows share one column-selection mask, thereby widening the space of viable pruned networks to accelerate. Libraries such as VENOM/Spatha provide published V:N:M kernel libraries, but these methods ship a dense-input sparse GEMM for *linear* layers only. To generalize to computer vision networks, convolutions require an "*im2col*" [5, 9] lowering[1] that is not natively provided. Additionally, given current constraints, not every convolution can reap the benefits from sparse execution due to size constraints imposed by available hardware. Notably, the sparse matrix-multiply-accumulate (MMA) instruction has a minimum occupancy tile (32 rows), resulting in under-utilization with narrow-channel layers, meaning these layers can then run *slower* through the sparse path than through a well-tuned dense cuDNN kernel[2]. A network built by uniformly substituting sparse GEMMs for every convolution therefore has no guarantee of a net speedup, and our measurements show several common architectures (ResNet, DenseNet, small LeNet variants) would regress if forced through the sparse path unconditionally.

We address these two issues by combining a carefully designed sparse convolution kernel with a per-layer fallback to optimized dense execution. The specific contributions of this work include:

- A **fused im2col-sparse-GEMM kernel** for stride-1 convolutions. The im2col patch gather is folded into the kernel's global-to-shared-memory load stage, so no im2col matrix is ever materialized in global memory.
- A **per-layer profiler with dense fallback**, meaning at calibration time, each convolution is timed through both the sparse-fused kernel and the dense cuDNN baseline on representative input. The layer is subsequently routed through whichever is faster at deployment. This makes the whole-network speedup $\geq 1\times$ by construction, since no layer is ever assigned to a slower path than it would have had without sparsity.
- End-to-end results measured on real hardware (RTX 4080) against dense cuDNN across nine network/dataset combinations at 2:16 V:N:M sparsity. The resulting speedups are up to 1.13× on wide-channel networks, and a clean 1.00× (correct full fallback, not a regression) on narrow-channel ones, when compared to cuDNN.

Note that these end-to-end numbers are not directly comparable to VENOM's own reported [3]. They provide speedup metrics of a single sparse-GEMM speedup over `cuSPARSELt` (not dense cuDNN) at fixed 2:4 sparsity on BERT-sized weight matrices deployed on an RTX 3090. Their result is a different-baseline, different-hardware, and different-workload quantity from our whole-network, dense-cuDNN-relative, convolutional result. To compare on equal footing, Table 2 and Table 3 hold the baseline (`cuSPARSELt`) and hardware (RTX 4080) fixed. Our fused V:N:M kernel proves to be

[1] im2col lowering is a technique for converting a multidimensional convolution to a standard GEMM operation.
[2] cuDNN is an optimized NVIDIA kernel for specific deep neural net matrix-related operations.

4.4–11.6× faster than cuSPARSELt at the convolution level and at the raw GEMM level, Spatha's kernels are 1.8–8.6× faster, with 4 times fewer nonzeros.

2 Background

A *dense tensor core* implements in a single cycle a matrix multiply of the form A*B+C where A is an nxk matrix (weights), B is a kxm matrix (activations), and C is an nxm matrix (biases and prior partial accumulations). They issue a fixed number of FMAs per clock cycle, namely the product of its fixed tile dimensions, thereby dictating the number of clock cycles needed for a n×k×m GEMM. A *sparse tensor core* is one where zeros in the A (weight) matrix can be skipped, and the hardware FMAs reallocated to perform additional non-zero multiplies. The same number of actual FMAs per clock are performed as in the dense version, but the result is equivalent to performing a GEMM with a bigger k. The measured flop rate is the same, but the "effective" flop rate is larger.

2.1 Structured Sparsity on Tensor Cores

NVIDIA's Ampere and Ada architectures include sparse tensor cores [7] that accept weights structurally pruned to a fixed 2:4 pattern (2 nonzeros per contiguous group of 4) and execute the corresponding GEMM at up to 2× the dense tensor-core rate [2, 6]. To do so, two 2-bit indices for each group of four weights are generated during a software compression step and consumed by the hardware to record which 2 of the 4 positions were kept. Given the rigidity of this format, it poses a larger challenge to map non-zero weights from unstructured pruning, a setting in which the 2-out-of-4 pattern of zeros is not guaranteed.

2.2 The V:N:M Generalization

V:N:M sparsity [3] relaxes the fixed 2:4 ratio to arbitrary N:M block patterns by having a block of V output rows share a single column-selection mask, so the same sparse-MMA hardware can execute the forward pass of a sparse weight and dense activation GEMM with higher sparsity. This is significant for pruning research because most works report much higher sparsity ratios than 2:4, allowing these resulting networks to be re-expressed at coarser N:M ratios like 2:16 without the accuracy collapse . The reference V:N:M kernel library (VENOM/Spatha) targets linear layers with a dense-input activation sparse-weight GEMM, but it does not provide a convolution path.

2.3 Why Uniform Substitution Is Unsafe

Two effects limit the blanket use of sparse convolution kernels. First, an im2col lowering for convolutions implemented naively as a separate materialization step adds a memory-bound pass that can dominate the savings from sparse compute [11]. Second, the sparse-MMA instruction *mma.sp.m16n8k32* on Ampere/Ada tasks a warp with the tiled matrix multiplication $D_{16\times8} = A_{16\times32}B_{32\times8}\ C_{16\times8}$, where A is sparsified along is 32-wide contraction dimension, and this operation emits 16 rows. Note that a single 16-row MMA will not saturate the warp; thus, the kernel issues several per warp, V/16 of them stacked along the output-channel axis, sharing one activation fetch, so the warp computes a tile V channels tall. V=16 (one MMA) is not efficable, V=32 is occupancy-bound, and V=64 is the smallest tile that wins. Note, however, that if C_{out} is not a multiple of V, it is padded up (a 16-channel layer fills a V=64 tile at 25%). Additionally, a small spatial size leaves the GEMM free dimension too small to saturate the cores. Channel width and spatial size vary widely within a network (LeNet's tens of channels vs. VGG-19's hundreds), so no global policy guarantees a speedup, motivating the per-layer profiler (Sec. 3.2).

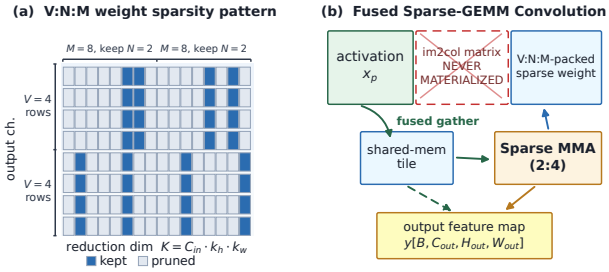


Figure 1: Convolutional V:N:M Sparsification & Fused im2col-sparse GEMM convolution

3 Method

Our method involves three components which include a custom fused kernel, a profiler, and a re-pruning procedure.

3.1 Fused im2col-Sparse-GEMM Kernel

To fold the im2col patch gather into the kernel's global-to-shared-memory load (Figure 1), each thread block loads the input patches it needs directly from the activation tensor in global memory into shared memory in the layout the sparse-MMA instruction expects (Fig. 1b). More precisely, each thread gathers its activation operands on the fly from a pre-padded input buffer, $\texttt{dst[j]} = B\big[\texttt{col_off}[n_g^0\texttt{+j}] + k_{\text{part}}\big]$, where n_g indexes the output pixel (the GEMM's free dimension) and k_{part} is the per-thread offset into the reduction dimension $K = C_{\text{in}}{\cdot}R{\cdot}S$, where R and S are the filter height and width respectively. This removes the extra memory round-trip that a two-stage (materialize-then-GEMM) im2col approach would incur, and is particularly advantageous because the sparse-MMA compute savings would otherwise be easily erased by memory-bound overhead at these sparsity levels.

The fused kernel is layout-sensitive due to manner in which activations are loaded. Its innermost loop reads eight adjacent output columns for a fixed reduction index, and the warp coalesces along this output-pixel axis. Under the NCHW format ([Batch, Channel, Height, Width]), those columns are neighbors, allowing a warp to fetch them with a single coalesced transaction. In contrast, under the NHWC format, they are C elements apart, splitting the fetch into eight separate transactions (Fig. 2). Therefore, the sparse kernel runs in NCHW order to maximize performance gains.

cuDNN's fastest fp16 convolutions use the channels-last (NHWC) tensor-core path, whose MMA fragments coalesce along the channel axis. Therefore, both cuDNN and our fused kernel coalesce along orthogonal axes, meaning that there is no optimal single absolute network-wide layout. To ensure performance guarentees, the network

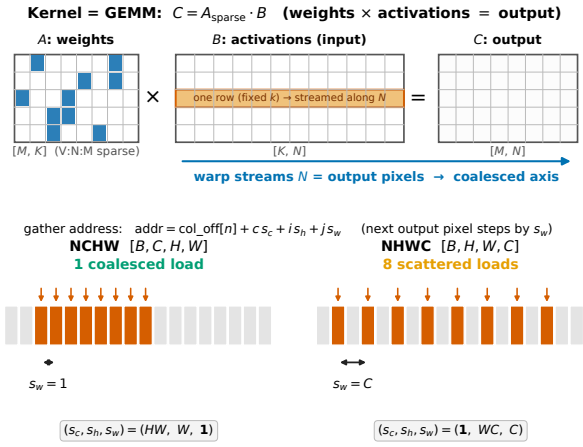


Figure 2: Why VENOMs fused sparse conv prefers NCHW

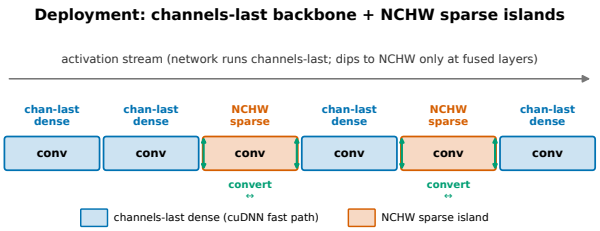


Figure 3: NCHW sparse islands visual

is run in channels-last format, to leverage native cuDNN acceleration, each sparse layer is executed as an NCHW *island*. To implement this, the activation is converted channels-last→NCHW on entry to the fused kernel and NCHW→channels-last on exit (Fig. 3). Each kernel then operates in the layout it coalesces in, at the cost of two layout conversions per fused layer, which the profiler charges against the sparse path.

3.2 Per-Layer Profiling and Dense Fallback

For each convolutional layer ℓ in the pruned network, the profiler:
(1) Re-prunes ℓ's weights to the 2:16 V:N:M pattern.
(2) Measures wall-clock latency $t_{\text{sparse}}\ell$ of the fused im2col-sparse-GEMM kernel on representative input dimension sizes.
(3) Measures wall-clock latency $t_{\text{dense}}\ell$ of the vendor cuDNN channel last convolution on the same input.
(4) Assigns ℓ to the sparse kernel iff $t_{\text{sparse}}\ell\, 1\ \varepsilon < t_{\text{dense}}\ell$ for a fixed margin ε; otherwise assign ℓ to the dense channel last kernel.

The margin ensures a layer is switched to the sparse path only when its measured win exceeds profiling noise. Because each layer independently uses whichever kernel measured faster, the per-layer latency after profiling is $\min t_{\text{sparse}}\ell, t_{\text{dense}}\ell \le t_{\text{dense}}\ell$ for every ℓ. Therefore, aggregate layer speedups result in a whole-network latency no worse than the all-dense baseline. In general terms, the profiler guarantees that the whole-network speedup is $\ge 1\times$ *by construction*.

3.3 Sparsification

The 2:16-sparse checkpoints used in this study are produced by re-pruning networks obtained from a prior layerwise magnitude-pruning method [10] to the V:N:M pattern rather than to unstructured sparsity. Source sparsity before this re-pruning vary, with 90% sparsity for VGG-19/CIFAR-10, 95% for ResNet-32/CIFAR-10, and 86% for ResNet-32/CIFAR-100. This paper is agnostic to the specific pruning algorithm used to reach a given sparsity level, and instead focuses on what happens *after* a V:N:M-sparse checkpoint exists. However, after imposing the V:N:M pattern, we use fine-tuning because vector-structured column selection may not align with the surviving weights. In our experiments, we find not all networks are lossless, e.g. VGG-19/CIFAR-10 goes from 92.9%→91.9%.

4 Experimental Setup

All experiments were run on a single NVIDIA RTX 4080, at fp16 precision, with a batch size of 128. Wall-clock measurements were utilized for comparing against the vendor cuDNN dense-convolution baseline, or the cuSPARSELt 2:4 baseline, using im2col in order to leverage their GEMM algorithms. Nine network/dataset combinations were evaluated at 2:16 (87.5%) V:N:M sparsity including:
(1) VGG-19 on CIFAR-10, CIFAR-100, and Tiny-ImageNet
(2) ResNet-32 on CIFAR-10, CIFAR-100, and Tiny-ImageNet
(3) ResNet-56 on CIFAR-10
(4) DenseNet-40 on CIFAR-10
(5) LeNet-5 on MNIST

Each network is profiled once per deployment target using the procedure in Section 3.2. The fused/total column in Table 1 reports how many of a network's stride-1 convolutions used the sparse kernel.

5 Results

Table 1 reports the measured end-to-end speedups over dense channels-last cuDNN baselines. VGG-19 has the highest fraction of layers routed to the sparse kernel (6 to 9 of its 16 stride-1 convolutions, depending on input resolution) due to its wide, purely feed-forward nature. Consequently, VGG-19 also had the largest measured speedups, up to 1.13× on CIFAR-10. In contrast, ResNet32, DenseNet40, ResNet56 and LeNet-5 all route *zero* layers to the sparse format, landing at a measured 1.00× speedup. In every case, these networks' channel widths force the kernel to use the smallest sparse-MMA tile size, providing negligible gains. The profiler correctly identifies this, verifying the lower bound of speedup the profiler provides.

Table 2 compares the fused V:N:M kernel against the cuSPARSELt hardware-native 2:4 path at the convolution level, on an RTX 4080 with the same specifications as Table 1(batch 128, fp16, VGG-scale layers). Column 2 is the dense cuDNN time, columns 3 and 4 are speedups over that dense baseline, and Column 5 is our kernel's speedup over cuSPARSELt. Because cuSPARSELt provides no convolution kernel, its conv path must materialize the patch matrix with im2col (`unfold`) and then run a 2:4 structured SpMM, a memory-bound two-stage lowering that leaves it *below* dense (columns 3). Our kernel instead gathers the im2col inline, avoiding that round-trip, as is reflected in the results of column 5. For an *optimized* vendor sparse-conv baseline rather than this naive one. Table 3 has the same layout as Table 2, but instead isolates the same comparison at the

Table 1: Measured whole-network V:N:M (2:16, 87.5%) acceleration on an RTX 4080 (batch 128, fp16), against the *channels-last* cuDNN baseline (cuDNN's fastest fp16 conv path).

Model	Input	Fused / total conv layers	GPU × (measured)
VGG-19 (CIFAR-10)	32^2	8/16	1.13
VGG-19 (Tiny-ImageNet)	64^2	9/16	1.09
VGG-19 (CIFAR-100)	32^2	6/16	1.08
ResNet-32 (CIFAR-10)	32^2	0/33	∼1.00
ResNet-32 (CIFAR-100)	32^2	0/33	∼1.00
ResNet-32 (Tiny-ImageNet)	64^2	0/33	∼1.00
ResNet-56 (CIFAR-10)	32^2	0/57	∼1.00
DenseNet-40 (CIFAR-10)	32^2	0/39	∼1.00
LeNet-5 (MNIST)	28^2	0/2	∼1.00

Table 2: Convolution-level comparison against the hardware-native 2:4 path on an RTX 4080 (batch 128, fp16, VGG-scale layers).

Conv layer ($C_{\text{in}} \to C_{\text{out}}$, r^2)	dense (ms)	cuSPARSELt (×dense)	fused V:N:M (×dense)	fused V:N:M vs cuSPARSELt (×)
$64 \to 64$, 32^2	0.112	0.09	1.03	11.6
$64 \to 128$, 32^2	0.214	0.17	1.16	7.0
$128 \to 128$, 16^2	0.105	0.14	1.25	9.1
$128 \to 256$, 16^2	0.194	0.24	1.43	5.96
$256 \to 256$, 8^2	0.105	0.20	1.26	6.4
$256 \to 512$, 8^2	0.213	0.33	1.44	4.4
$512 \to 512$, 4^2	0.123	0.23	1.40	6.0
geomean	—	0.19	**1.27**	**6.9**

Table 3: GEMM-level comparison at the shapes VGG-scale convolutions lower to ($M = C_{\text{out}}$, $K = C_{\text{in}}RS$, $N = BO_HO_W$) on an RTX 4080 (batch 128, fp16)

GEMM ($M \times K \times N$)	dense (ms)	cuSPARSELt 2:4 (× dense)	Spatha 2:16 (× dense)	Spatha / cuSPARSELt
64×576×131072	0.279	1.00	8.65	8.6
128×576×131072	0.335	1.13	2.03	1.8
128×1152×32768	0.154	0.97	5.18	5.4
256×1152×32768	0.251	1.53	3.88	2.5
256×2304×8192	0.147	1.76	5.32	3.0
512×2304×8192	0.269	2.28	5.45	2.4
512×4608×2048	0.134	1.62	4.94	3.0
geomean	—	1.41	**4.70**	**3.3**

raw GEMM level, using the resulting GEMM dimensions that VGG-scale convolutions lower to ($M = C_{\text{out}}$, $K = C_{\text{in}}RS$, $N = BO_HO_W$). Both operands are already in matrix form, so no im2col is involved and cuSPARSELt instead runs *above* dense, and the pure Spatha 2:16 GEMM is from [3]. Both tables show that this method achieves higher sparsity *and* a faster kernel simultaneously.

6 Related Work

N:M structured sparsity was popularized for training-aware pruning by NVIDIA's Automatic SParsity (ASP) toolkit and the associated 2:4 fine-grained sparse training recipe [6], and has support at inference time from TensorRT's native 2:4 sparse kernels [8]. Both target the fixed 2:4 ratio the hardware was originally designed around. V:N:M [3] is, to our knowledge, the main published generalization to coarser N:M ratios such as 2:16, and is the format this paper builds on. None of these prior systems address convolutional im2col fusion at the V:N:M ratio. TensorRT does provide the capability to fall back to dense formats when the sparse counterpart provides no performance gain, but it is available only in the fixed 2:4 format.

TVM provides a per-operator schedule search [4], which profiles candidate implementations of an operator and keeps the fastest. Our profiler applies the same measure-and-select principle, but to a narrower, structurally motivated choice rather than a general schedule search.

7 Discussion and Limitations

This work provides a complementary route to acquire speedup from sparse computer vision networks. It works on *commodity* GPUs available today, applies safely to residual and dense topologies alike, and comes with a profiler to ensure the resulting network is no slower than its dense counterpart. The performance floor is a direct consequence of the sparse-MMA's minimum tile size (32 output rows). Networks whose convolutions are consistently narrower than this floor, e.g. small models like LeNet, or architectures such as ResNet-56 have channel counts which do not clear the floor at the resolutions we test. Hence, they have to fall back to fully dense execution, and the profiler adds only its one-time offline calibration cost. This is by design, and the contribution of this work is the safety property, not a claim that sparse tensor cores help every network.

Two limitations follow directly from this work. First, the current fused kernel supports stride-1 convolutions only, which we treat in future work. Second, the profiling decision is made once, offline, per deployment shapeand does not adapt to shape changes at runtime.

8 Conclusion

Reported pruning sparsity and measured hardware speedup are not the same quantity, and closing the gap between them safely requires more than lowering every layer to whatever sparse kernel is available. We show that combining a fused im2col-sparse-GEMM kernel with a simple per-layer profiler against the dense cuDNN baseline turns V:N:M sparse execution into a strictly non-regressive transformation. Every layer is only ever routed to the kernel that measured faster for it, so the whole network's speedup is $\geq 1\times$ by construction. On real RTX 4080 hardware at 2:16 V:N:M sparsity, this yields up to $1.13\times$ measured end-to-end speedup on wide-channel CNNs while correctly, and safely, falling back to dense execution on narrow-channel networks rather than regressing. Extending the fused kernel to strided/dilated convolutions and replacing offline profiling with a runtime-adaptive cost model are natural next steps.

Acknowledgments

The work discussed here was supported by funding from the Univ. of Notre Dame.

References

[1] AMD Corp. [n. d.]. AMD CDNA 3 Architecture.

://www.amd.com/content/dam/amd/en/documents/instinct-tech-docs/white-papers/amd-cdna-3-white-paper.pdf.

[2] Hongxiao Bai and Yun Li. 2026. Structured Sparsity in the NVIDIA Ampere Architecture and Applications in Search Engines. https://developer.nvidia.com/blog/structured-sparsity-in-the-nvidia-ampere-architecture-and-applications-in-search-engines/.

[3] R. L. Castro, A. Ivanov, D. Andrade, T. Ben-Nun, B. B. Fraguela, and T. Hoefler. 2023. VENOM: A Vectorized N:M Format for Unleashing the Power of Sparse Tensor Cores. In *Proc. Int'l Conf. for High Performance Computing, Networking, Storage and Analysis (SC)*.

[4] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *Proc. USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

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

[6] A. Mishra, J. A. Latorre, J. Pool, D. Stosic, D. Stosic, G. Venkatesh, C. Yu, and P. Micikevicius. 2021. Accelerating Sparse Deep Neural Networks. *arXiv preprint arXiv:2104.08378* (2021).

[7] NVIDIA Corp. 2026. NVIDIA A100 Tensor Core GPU Architecture. https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf/.

[8] NVIDIA Corporation. 2023. *TensorRT Developer Guide: Structured Sparsity*. NVIDIA Documentation.

[9] Philippe Tillet. [n. d.]. TMA im2col mode and Convolution via Implicit GEMM. https://triton-lang.org/main/getting-started/tutorials/gluon/conv-im2col.html.

[10] Jay Vap, Reem Shehayib, and Peter Kogge. 2025. JORTs: Jenk's Optimization for Restrictive Training. In *2025 IEEE International Conference on Data Mining Workshops (ICDMW)*. IEEE, 510–519.

[11] Yangjie Zhou, Mengtian Yang, Cong Guo, Jingwen Leng, Yun Liang, Quan Chen, Minyi Guo, and Yuhao Zhu. 2021. Characterizing and demystifying the implicit convolution algorithm on commercial matrix-multiplication accelerators. In *2021 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 214–225.