WHAT SETS A BLOCK-SPARSE TRAINING SPEEDUP?
SM COUNT, THE DENSE BASELINE AND THE KERNEL, ON SIX GPUS

Anonymous authors
Paper under double-blind review

ABSTRACT

Dynamic sparse training (DST) rarely delivers wall-clock speedups, because standard implementations multiply dense weights by masks and still execute dense products. We ask what sets the speedup once the sparsity is actually executed. Our instrument is BlockDST, which keeps a constant number of live blocks in every block-row so that our own load-balanced blocked-ELL kernels run all three per-layer products sparse, including a block sampled dense–dense matrix multiplication (block-SDDMM) for the weight gradient. On six NVIDIA graphics processing units (GPUs), the same kernel and shape run 1.08–$5.13\times$ faster than a dense General Matrix Multiply (GEMM) at 95% sparsity, and most of this spread comes from how fast the dense baseline runs on each card. Our kernel's throughput scales with streaming multiprocessor (SM) count $\times$ clock, so a one-parameter model predicts it on a held-out card to within 0.91–$1.19\times$ at the reference shape (0.78–$1.66\times$ across shapes). That scaling is not a property of block sparsity in general: cuSPARSE's Blocked-ELL kernel, timed on five of the same cards, does not follow it. It runs $1.5\times$ faster on an A100 than on an A40, while ours runs $0.95\times$ as fast, yet its speedup over dense still varies about $2\times$ from card to card. End to end the picture is narrower: on an L40, BlockDST reaches a fixed accuracy $1.44\times$ faster than dense (not significant with three seeds) with $2.1\times$ less memory, but it is slower than dense on an A40, slower than a dense model with the same number of live parameters, and, on a laptop GPU, 1.2–$1.4\times$ slower than the same block layout without rewiring. A sparse speedup should therefore be reported together with the kernel, the card's SM count and clock, and the dense throughput it was measured against.

1 INTRODUCTION

Sparsity promises cheaper training but rarely delivers it. Most dynamic sparse training (DST) implementations keep a dense parameter tensor and multiply it by a binary mask, so every sparse step still runs the full dense General Matrix Multiply (GEMM). Across 90–98% sparsity we find three published DST patterns running within 0.35 steps/s of the dense baseline they are meant to accelerate, and in the standard implementation the sparsity saves no memory either (Appendix K).

We therefore started by measuring which sparse formats NVIDIA libraries can accelerate. On identical matrices, cuSPARSE's unstructured Compressed Sparse Row (CSR) kernel is $4\times$ *slower* than a dense GEMM at 90% sparsity and only breaks even at 98%; its legacy Block Sparse Row (BSR) path runs in 32-bit floating point (fp32) without tensor cores and breaks even around 90%; its blocked ELLPACK (Blocked-ELL) path runs on tensor cores and beats a dense half-precision GEMM from 75% sparsity, reaching $2.68\times$ at 90% and $7.93\times$ at 98% (Appendix C). We built on Blocked-ELL and wrote our own kernels for it; every training result in this paper uses them, not cuSPARSE. Its one requirement is that every block-row holds the *same number* of live blocks, and we make that count, k, an invariant of training. Pruning and growth exchange blocks inside a block-row and never change k, so updating the mask becomes a permutation problem, and the forward kernel needs no row pointers, only a fixed-width array of block-column ids. The weight gradient inherits the pattern of W, and the input gradient reads the same blocks through $W^\top$, so all three per-layer products run sparse; for the weight gradient we wrote a block-granular sampled dense–dense product (block-SDDMM), which block-sparse libraries provide for their own layouts but not for blocked-ELL.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

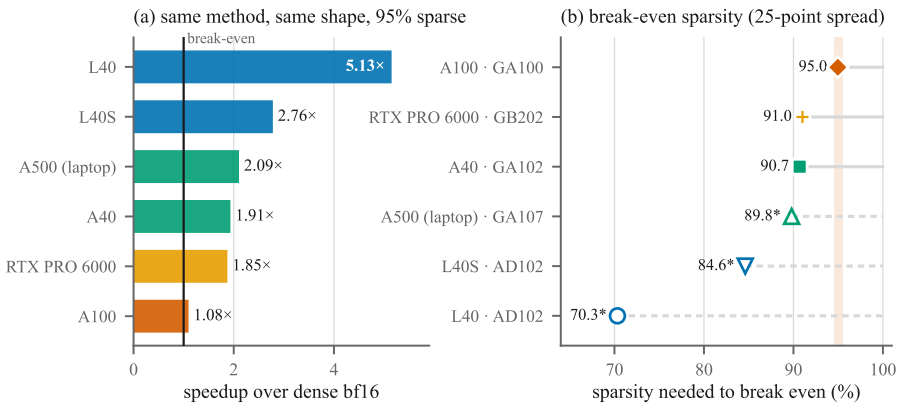

Figure 1: The same kernel on six accelerators. **(a)** Speedup of our kernel over a dense bfloat16 (`bf16`) GEMM at 8192^3 and 95% sparsity ($k = 6$ of 128 block-columns, 95.3%). It is above break-even on every card, from $5.13\times$ on the L40, whose dense `bf16` rate is half the L40S's by datasheet, to $1.08\times$ on the A100; at 90% it is at or below break-even on four of the six (Figure 7). A kernel speedup is not a training speedup: the A40's $1.91\times$ here ($2.41\times$ at the training shape) becomes a $0.70\times$ end-to-end result (Section 5.4). **(b)** Sparsity needed to break even on each card, interpolated between measured densities; starred values (hollow markers) lie beyond the densest point measured and are extrapolated with the card's fitted exponent (Table 1). Forward product only.

The more useful question turned out to be *where* the method is faster. Activation tiles are re-read once per live block, so memory traffic and floating-point operations (FLOPs) both grow as $\Theta(k)$ and arithmetic intensity hardly moves with density. The speedup over dense then scales as d^{-p} in density d with a card-dependent constant; if the kernel were bandwidth-bound, that constant would follow from bandwidth, arithmetic intensity and dense GEMM throughput, but on six accelerators the prediction is off by up to $3.3\times$, in both directions. Kernel throughput instead tracks streaming multiprocessor (SM) count $\times$ boost clock (an A100 with $1.96\times$ the bandwidth of an A40 runs the kernel 10% slower), and with that in place of bandwidth a model fitted on five cards predicts the sixth to within 0.91–$1.19\times$ at the reference shape (Figure 3). Most of the variation across cards, however, comes from the dense baseline: per SM-GHz, cuBLAS varies $4.53\times$ across our cards and our kernel $1.24\times$. We identify a scaling, not the per-SM resource behind it. Our kernel runs at 3–16% of the tensor-core peak, so we repeated the test on cuSPARSE's own Blocked-ELL kernel, timed on five of the same cards: it does not follow the scaling (Section 4.2). The constants belong to the kernel; the dependence of the speedup on the card holds for both.

A speedup measured on one unnamed accelerator therefore describes a kernel on a card, not the method (Figure 1). Our contributions form a measurement study with BlockDST as its instrument:

1. A speedup law for one block-sparse training kernel on six GPUs (Section 4): speedup $\propto d^{-p}$ with a similar exponent on every card ($p = 0.84 \pm 0.09$), while the constant varies $5.4\times$, mostly through the dense baseline.
2. A cost model (Section 4.1): our kernel's throughput scales with SM count $\times$ clock, with bandwidth's role bounded rather than excluded, and one fitted constant predicts a held-out card to within 0.91–$1.19\times$ at the reference shape (0.78–$1.66\times$ across shapes), against 0.51–$3.1\times$ for a bandwidth model fitted the same way. The same test on cuSPARSE's Blocked-ELL kernel shows that this scaling does not carry over to another kernel (Section 4.2).
3. The instrument (Section 3): row-balanced block-sparse training that keeps a constant number of live blocks per block-row, with optional rewiring, and runs all three per-layer products sparse, weight gradient included.
4. A time-to-target evaluation (Section 5) that separates kernel, step-rate and end-to-end speedup. On an L40 at 95% sparsity BlockDST reaches a fixed accuracy $1.44\times$ faster than dense (the three-seed interval includes 1) with $2.1\times$ less memory, but a second card, a parameter-matched dense model, a never-rewired block layout and per-epoch accuracy all favor the alternatives.

2 RELATED WORK

Dynamic sparse training. Sparse Evolutionary Training (SET) (Mocanu et al., 2018) and Rigged Lottery (RigL) (Evci et al., 2020) maintain unstructured masks with prune–grow cycles, MEST (Yuan et al., 2021) and SNIP (Lee et al., 2019) use other criteria, and Liu et al. (2022) show that random pruning is a strong baseline for all of them (see also Sparse Momentum, Dettmers & Zettlemoyer, 2019). They are usually evaluated in FLOPs, although FLOP savings rarely become wall-clock savings (Hoefler et al., 2021) and elaborate sparsification does not reliably beat magnitude pruning at scale (Gale et al., 2019). An unstructured mask gives a tensor core no dense sub-block to work on, an instance of the *hardware lottery* (Hooker, 2021), which we find varies even within one vendor's product line.

Structured and hardware-aware sparse training. Recent methods target patterns a kernel can exploit: structured RigL (SRigL) (Lasby et al., 2024) enforces constant fan-in, DynaDiag (Tyagi et al., 2025a) diagonal structure, ShuffleSparse (Tyagi et al., 2025b) learned permutations over block, N:M and diagonal structures, HASTE (Ullah et al., 2026) group-shared fan-in for extreme multi-label output layers, and SMET (Xiao et al., 2026) stabilizes DST for large language models (see also Chen et al., 2022; Jiang et al., 2022; Yin et al., 2023; Hubara et al., 2021). Most report forward or inference speedups, although the weight gradient is about a third of training. The 2:4 pattern that NVIDIA hardware accelerates (two nonzeros in every four consecutive weights) supports pre-training at a fixed 50% sparsity (Hu et al., 2024a;b), whereas post-training pruning such as SparseGPT (Frantar & Alistarh, 2023) and Wanda (Sun et al., 2024) does not reduce training cost.

Load-balanced formats and block-sparse kernels. A constant nonzero count per row is the known requirement of ELL (Bell & Garland, 2009), and Yao et al. (2019) made it a pruning constraint at element granularity. Our invariant applies the same idea at block granularity and differs in three ways: the count is kept exact during rewiring *within* a block-row, the weight gradient shares W's pattern and so also runs sparse, which Top-KAST (Jayakumar et al., 2020) pursues at a different granularity, and we provide a block-SDDMM for this layout. Among kernels, Sputnik (Gale et al., 2020) targets unstructured sparse–dense matrix multiplication (SpMM), MegaBlocks (Gale et al., 2023) casts mixture-of-experts routing as block-sparse products, BLaST (Okanovic et al., 2025) uses blocked compressed sparse columns with a variable number of blocks per column, and VENOM (Castro et al., 2023) and Samoyeds (Wu et al., 2025) extend sparse tensor cores beyond 2:4 (Mishra et al., 2021); OpenAI's kernels (Gray et al., 2017), Triton (Tillet et al., 2019) and TileLang (Wang et al., 2025) serve row-pointer (BSR) layouts, weight gradient included.

Performance models and evaluation. Our analysis follows the roofline approach (Williams et al., 2009) but measures the attained fraction instead of assuming it; per-SM limits are modeled by Ding & Williams (2019) and Volkov (2016). That kernel performance does not carry over between GPU architectures is established in high-performance computing (Davis et al., 2025); predicting training time on an unseen GPU from SM-count, clock and bandwidth ratios is the aim of Habitat (Yu et al., 2021) (see also Qi et al., 2017; Lym et al., 2019), and our cost model is a one-parameter case of it. We add that the SM $\times$ clock term suffices for our kernel, that the dense baseline carries most of the variation, and that a second kernel on the same cards does not follow it, so the term has to be found per kernel. A hierarchical roofline (Yang et al., 2020) would charge activation re-reads to L2. Blalock et al. (2020) and Dehghani et al. (2022) argued that efficiency results are hard to compare and depend on implementation and hardware; we add a measured magnitude and a model that says which numbers to report. Our time-to-target protocol follows MLPerf (Mattson et al., 2020). Appendix O covers further work.

3 METHOD

We partition $W \in \mathbb{R}^{M\times K}$ into $b\times b$ blocks ($b = 64$) and keep exactly k live blocks in every block-row, which gives density $d = kb/K$ (Figure 2). The invariant holds at initialization and after every update. Every ΔT steps (1000 for the MLP, 100 for the language models), each block-row drops its n live blocks with the smallest Frobenius norm and grows n dead blocks chosen uniformly at random *in the same block-row*, as SET does for single weights; grown blocks start at zero with reset momentum. By default the first rewiring happens at step 0, before any training. Because drops and grows are paired,

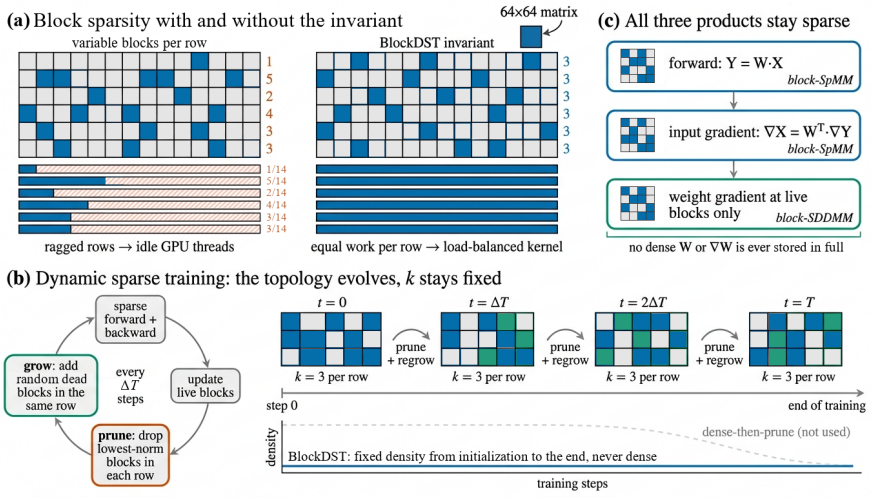

Figure 2: BlockDST: **(a)** Two block-sparse patterns at the same density (18 live blocks; every square is one $b \times b$ block). On the left the number of blocks per block-row varies, on the right the invariant fixes it at k. Blocked-ELL stores every block-row at the length of the longest, so an uneven pattern is padded or needs a row-pointer format with unequal work per row, while a constant k gives every row the same work and no padding. **(b)** The pattern changes during training: the network is sparse from initialization, and every ΔT steps each block-row drops its lowest-norm blocks and grows the same number of dead blocks, chosen at random in that row (green), so k never changes. **(c)** The forward pass uses W, with k blocks per row; the weight gradient is a block-SDDMM on the same pattern; the input gradient uses $W^\top$, whose rows are W's columns and so hold unequal counts. No dense W or ∇W is ever formed. Sizes are illustrative.

k stays constant by construction, with no penalty or projection. Random growth needs no gradient at dead blocks, so nothing dense is ever formed. Every timed result uses this rule; a gradient criterion (largest dead-block gradient norm) exists only in our masked reference implementation, used for the accuracy ablations (Appendix M). As k approaches 1 the update has too few blocks to work with, which is why 98% sparsity ($k = 1$ and 5) performs poorly at our widths. A group-lasso penalty on the live blocks does not help (Appendix D).

The forward product ($Y = W_{\text{blk}} X$) and the weight gradient address blocks by block-row, where every row holds exactly k blocks. The input gradient ($\nabla X = W_{\text{blk}}^\top \nabla Y$) reads the same blocks through a block-column index, rebuilt only when the mask changes, because the rows of $W^\top$ are not balanced. The weight gradient is a block-SDDMM, the product $\nabla Y X^\top$ evaluated at live blocks only, which we implemented with NVIDIA's warp matrix multiply-accumulate (WMMA) instructions.

4 The Speedup Law

Each block-row selects its own block-columns, so activation tiles cannot be shared between block-rows and are read k times. Traffic and FLOPs both scale as $\Theta(k)$, which keeps arithmetic intensity within 15.9–29.5 FLOP/byte: it varies $1.85\times$ across the density range and only 0.7% with layer geometry at a fixed density (Appendix B). Since dense time does not depend on sparsity while ours scales with d,

$$\text{speedup over dense } \texttt{bf16} \;=\; \frac{c}{d_0}\left(\frac{d_0}{d}\right)^p, \qquad c \;=\; \frac{\text{kernel TFLOP/s}}{D} \;\text{ at } d_0, \tag{1}$$

where $d_0 = 0.102$ is the reference density, D the dense throughput, and c/d_0 the measured speedup at d_0. Break-even is interpolated between measured densities where they bracket it and otherwise extrapolated as $d^* = d_0\,(c/d_0)^{1/p}$. A bandwidth-bound kernel with density-independent intensity would give $p = 1$. Fitted per (card, geometry) over three densities, p ranges from 0.677 to 0.970 with a mean of 0.84 ± 0.09 (median per-fit standard error 0.04), and all 18 fits lie below 1 (Figure 6); per-card intervals include 1, and the pooled density coefficient ($[0.12, 0.25]$, Equation (4)) is what

Table 1: The speedup law on six accelerators. D is the dense `bf16` throughput measured on the same shapes in the same process, in trillions of floating-point operations per second (TFLOP/s); $\mathrm{SM}\cdot f$ is SM count times clock in SM-GHz (datasheet boost clock for the RTX A500, A100 and L40, driver maximum for the others). c and break-even are at the reference shape, with the range of c across shapes in brackets. GDDR: graphics double-data-rate memory; HBM: high-bandwidth memory.

GPU	silicon / mem	$\mathrm{SM}\cdot f$	D	c (range)	break-even
RTX A500 (laptop)	GA107 / GDDR6	28	21 T	0.102 [0.070–0.107]	89.8%*
A100-PCIE-40GB	GA100 / HBM2e	152	213 T	**0.049** [0.039–0.054]	**95.0%**
A40	GA102 / GDDR6	146	124 T	0.093 [0.058–0.096]	90.7%
RTX PRO 6000	GB202 / GDDR7	457	428 T	0.090 [0.062–0.096]	91.0%
L40S	AD102 / GDDR6	358	209 T	0.141 [0.073–0.141]	84.6%*
L40	AD102 / GDDR6	354	109 T	**0.261** [0.154–0.266]	**70.3%***

*At the densest point measured ($d = 0.102$, 89.8% sparsity) the kernel is still at or above dense speed on these cards (1.00, 1.38 and 2.57×), so break-even is extrapolated with the card's fitted exponent (0.91, 0.78, 0.88).

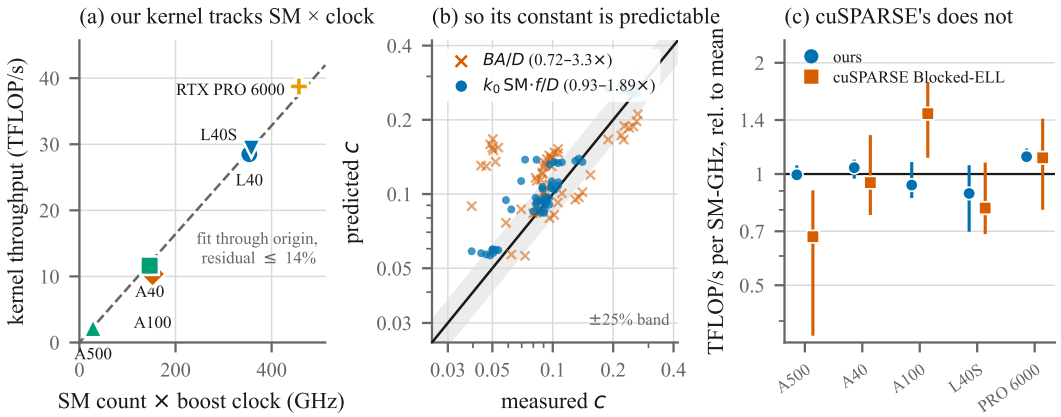

Figure 3: Our kernel's throughput scales with SM count × clock, which makes its constant predictable; cuSPARSE's does not. **(a)** Our kernel at the reference shape; a fit through the origin leaves at most 14% residual over a 16× range of SM count × clock. **(b)** Predicted against measured c for all 54 measurements, k_0 fitted on the six reference-shape rows: bandwidth model $c = BA/D$ (crosses) and SM-count model $c = k_0(\mathrm{SM}\cdot f)/D$ (dots, 0.93–1.16× at the reference shape; out-of-sample errors in Table 6), with a ±25% band. **(c)** Throughput per SM-GHz on the five cards both kernels were timed on, relative to each kernel's five-card mean; medians and ranges over configurations (9 for ours, 18 for cuSPARSE). A kernel that tracked SM count × clock would sit at 1 on every card.

places p below 1. The exponent is similar across cards (per-card means 0.79–0.90, with overlapping 95% intervals; Appendix B) while c varies 5.4×, so the constant is what needs to be predicted.

4.1 Predicting the constant

If the kernel were bandwidth-bound at a known arithmetic intensity A, its throughput would be $B\cdot A$ for memory bandwidth B. The dense baseline achieves D and our kernel performs a fraction d of its FLOPs, so

$$c = \frac{B \cdot A}{D}. \tag{2}$$

Testing this requires cards that differ in bandwidth, so we used six accelerators spanning 107–1482 GB/s (13.8×), with A derived per shape and B measured. Over the 54 measurements the ratio of predicted to measured c ranges from 0.72 to 3.3×, with errors in both directions (Figure 3(b), crosses). The attained fraction of this roofline, $\eta = c_{\mathrm{meas}} D/(BA)$, reaches 139% on the large-L2 AD102 cards, which shows that the traffic model for dynamic random-access memory (DRAM) is wrong (it ignores cache reuse) but not by itself that bandwidth is irrelevant. Three model-free observations point the same way.

An A40 (584 GB/s) runs the kernel at 11.6 TFLOP/s, 11% faster than an A100 (1146 GB/s) at 10.4; throughput per unit $\mathrm{SM}\cdot f$ varies by 6.4% across the six cards (coefficient of variation), against 47.1% per unit bandwidth (Figure 3(a)); and halving the weight traffic with `bf16` storage gains only 1.12–1.25×, not 2× (Appendix G.1). A regression of log kernel TFLOP/s on $\log \mathrm{SM}\cdot f$, $\log B$ and $\log d$ over the 54 rows gives coefficients of $+0.97$ and $+0.04$ for $\mathrm{SM}\cdot f$ and bandwidth ($R^2 = 0.986$; Equation (4)). With six cards this bounds bandwidth's role rather than excluding it: a wild cluster bootstrap (Cameron et al., 2008) gives $[-0.23, 0.30]$ for its coefficient, and without the A100 it rises to $+0.28$ (Appendix B.3). SM count × clock is also proportional to other per-SM resources, so we identify a scaling, not the limiting resource.

With SM count × clock in place of bandwidth, $c = k_0\,(\mathrm{SM}\cdot f)/D$ has one fitted constant, $k_0 = 0.078$ TFLOP/s per SM-GHz. Fitted on five cards, it predicts the sixth to within 0.91–1.19× at the reference shape, against 0.51–3.1× for the bandwidth model fitted the same way, and to within 0.78–1.66× against 0.53–3.15× over all 54 rows (Table 6). Because $c = \text{kernel TFLOP/s}/D$, the 5.4× variation splits per SM-GHz into 1.24× from our kernel and 4.53× from the dense `bf16` baseline; the L40, for instance, runs `bf16` with fp32 accumulation at half the L40S's datasheet rate, which halves D but not our kernel, far below peak (Appendix B.3). The model still needs D, so it replaces a full sweep with a single dense GEMM.

4.2 A SECOND KERNEL

Our kernel runs far below the tensor-core peak, so scaling with SM count × clock might simply be what any such kernel does. To check, we reused the cuSPARSE Blocked-ELL timings of our format study (Appendix C), taken on five of the six cards (all but the L40) before the cost model existed. The layout is the same, but the kernel is the vendor's, its throughput is 2.5–6.7× ours, and its matrices are smaller (fp16, weight 4096^2). It does not scale with SM count × clock (Figure 3(c)): across the five cards its throughput per SM-GHz has a median coefficient of variation of 33% over 18 configurations, against 8.7% for ours, and the one-parameter model misses a held-out card by 0.56–1.66×. The A100 shows the difference most clearly. It has 4% more SM-GHz than the A40, but twice the tensor-core width per SM, twice the bandwidth and an L2 that holds these working sets; cuSPARSE runs 1.46× faster on it (median; 1.19–2.11×), and ours 0.95× as fast (0.88–1.13×). The constants of Section 4.1 therefore do not transfer to another kernel, although we cannot say which resource cuSPARSE follows. The dependence on the card does: at 90.6% sparsity cuSPARSE's speedup over dense runs from 2.68× on the RTX A500 to 5.02× on the RTX PRO 6000 (Appendix B.9).

4.3 IMPLICATIONS FOR REPORTED SPEEDUPS

Headline speedups in the closest work range from 1.59× to 37× (Appendix O) on different accelerators, shapes, phases and baselines, while ours varies 5.4× with the card alone, so a single-card number cannot separate a better method from a better-suited GPU. We suggest reporting three inexpensive quantities: SM count and boost clock, the dense baseline's measured throughput D on the shapes used, and the fitted exponent with the density range it was fitted on. With these, a reader can estimate the constant for our kernel on another of our cards to within about 19% at the reference shape (Appendix F). The constants have to be fitted for each kernel, since ours do not carry over to cuSPARSE's.

5 EXPERIMENTS

Unless stated otherwise, we train a multi-layer perceptron (MLP) with two hidden layers of width 16,384 ($3072 \to 16{,}384 \to 16{,}384 \to 10$, 319M parameters when dense) on CIFAR-10 (Krizhevsky, 2009) with random crop and flip; both hidden layers are sparse with block size 64. Every arm uses stochastic gradient descent (SGD) with Nesterov momentum (Sutskever et al., 2013) 0.9, a constant learning rate of 0.05, batch size 8192 and bf16 mixed precision (Appendix M), and all arms of a comparison run one after another in one job on one L40 or A40, with three seeds. The baselines, SET, SRigL and DynaDiag, reimplement the published patterns at matched density with the same schedule and learning rate, tuned for no arm; they multiply a dense weight by a mask, so they measure what each pattern costs in accuracy, not what a kernel-based competitor achieves in time (Appendix O.1).

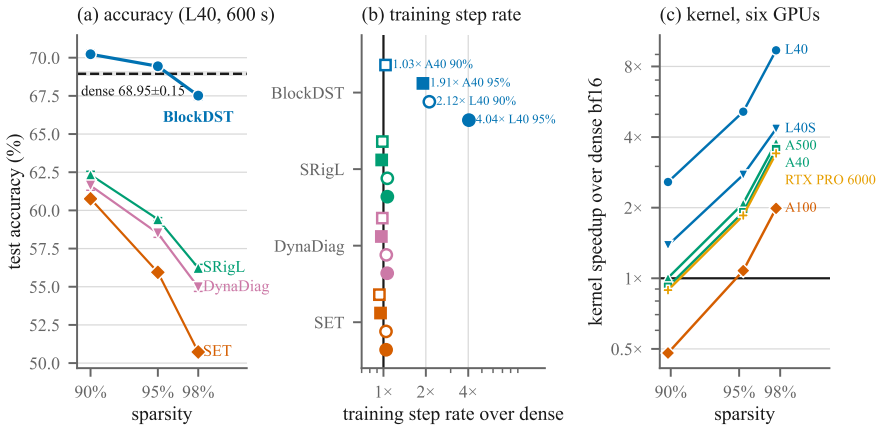

Figure 4: Accuracy and speed of all methods: **(a)** Test accuracy after 600 s on one L40 (mean ± standard deviation (sd), 3 seeds); the band is dense pooled over its nine runs (68.95 ± 0.15), which do not depend on sparsity; one batch failing our single-card check is excluded (Appendix L). **(b)** Training step rate relative to the dense arm of the same job; circles L40 (n=3), squares A40 (n=1), filled 95%, open 90%. **(c)** Kernel speedup of BlockDST over dense `bf16` on six accelerators.

With equal wall-clock budgets, BlockDST is the only sparse method that matches dense accuracy at 90% and 95% sparsity (Figure 4): it is 1.22 percentage points (pp) above dense at 90% and 0.53 pp at 95%, neither significant after Holm correction (Appendix J), and 7.9–13.5 pp above every masked baseline; at 98% dense leads by 1.40 pp. These margins are specific to this large-batch configuration and to the shared learning rate, which penalizes SET most (Appendix E): epoch-matched at batch 256, RigL is 0.7–2.8 pp more accurate than BlockDST at widths 1024–8192, so block structure does cost accuracy per step.

5.1 Time to a target accuracy

A fixed clock can show a throughput advantage only as extra accuracy, so we fix the accuracy and measure the time. The learning rate is constant, evaluations lie on a geometric time grid so that every arm has the same relative resolution, and evaluation time and 25 warm-up and calibration steps are untimed for every arm; BlockDST's rewiring at step 0 is timed. We report all three targets we set in advance, 0.45, 0.50 and 0.55 (Table 2 and Figure 5). BlockDST reaches them within its first 1000 steps, so these runs test the block layout after one rewiring at initialization. That rewiring does not help: on an RTX A500 at width 8192 the same layout never rewired reaches 0.55 and 0.60 1.19× and 1.36× sooner, over three seeds with non-overlapping ranges (Appendix E); delaying the first rewiring to step 1000 does not rescue it (one seed: 1.29× the static layout's steps to 0.60), so the gains below come from the kernel, not from dynamic sparsity.

BlockDST reaches 0.55 in 29.1 s against 41.8 s for dense (Table 2). The margin over dense is not significant with three seeds (1.44×, 95% confidence interval $[0.99, 2.07]$, $p = 0.050$). Two post hoc analyses, which we report as exploratory, agree: the dense arm ignores the density setting, so pooling its six runs gives 1.48× ($[1.21, 1.81]$), and interpolating the crossing gives 1.45× ($[1.08, 1.95]$) (Appendix J). It also needs the fewest training FLOPs of any method to reach 0.55 (7.1× fewer than dense; Appendix I). A dense MLP with as many live parameters is faster still: on an RTX A500 it reaches 0.55 10–14× sooner than BlockDST at widths 8192 and 16,384 (Appendix E), so BlockDST speeds up the wide model it sparsifies but is not a better use of a fixed parameter budget here.

The masked baselines take 2.7–7.0× longer than dense to reach 0.55, since they pay an accuracy cost for sparsity with no throughput gain. The ordering also depends on the target: at 95% sparsity BlockDST and dense are tied at 0.45, and at 90% dense is fastest to 0.45 and 0.50 and falls behind only at 0.55, the crossover of Figure 5(a) (see also Appendix N).

The two protocols favor different sparsities over dense, 90% with equal wall-clock budgets and 95% with time-to-target (Figure 12), but no margin is significant after correction.

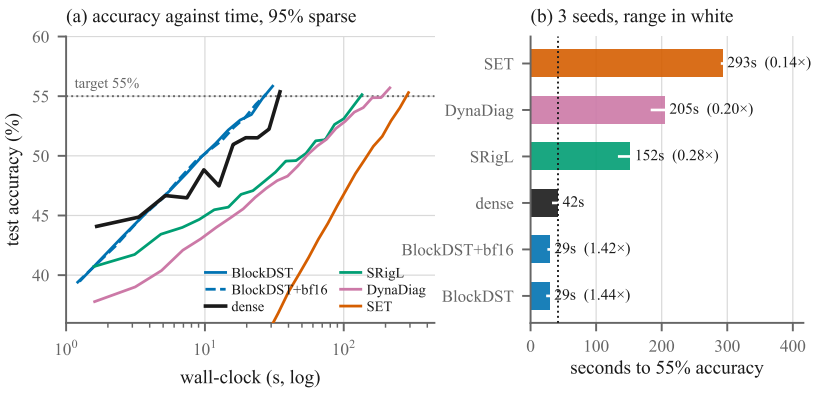

Figure 5: Accuracy against time: **(a)** Dense is ahead early, having more capacity per step, and is overtaken as BlockDST's step-rate advantage accumulates; a fixed time budget does not show this crossover. **(b)** Time to 55% accuracy over three seeds, range in white. L40, width 16,384, batch 8192, 95% sparsity, constant learning rate.

Table 2: Time (seconds) and GPU energy to first reach each test-accuracy target. Width 16,384, batch 8192, L40, n=3 seeds (mean ± sample sd); every arm reached every target on every seed. Energy, measured to the 0.55 target, tracks time. Near 0.45 the evaluation grid (steps of 15%) sets the resolution, so ties there are not evidence of a difference.

arm	steps/s	$t\to0.45$	$t\to0.50$	$t\to0.55$	energy
95% sparsity					
BlockDST	**21.7**	3.9±0.0	**11.1±1.4**	**29.1±2.9**	**8.7 kJ**
BlockDST+bf16	21.4	4.0±0.8	10.7±0.0	29.4±2.9	8.8 kJ
dense	5.4	4.0±1.1	16.0±0.0	41.8±7.3	12.5 kJ
SRigL	5.7	11.9±0.2	51.6±4.3	152.0±14.5	45.4 kJ
DynaDiag	5.7	17.6±2.1	57.9±5.4	204.8±16.9	61.1 kJ
SET	5.6	89.6±0.2	163.6±0.5	293.0±0.9	86.5 kJ
90% sparsity					
BlockDST	11.4	6.5±0.2	15.9±0.6	43.9±1.6	13.1 kJ
BlockDST+bf16	**13.0**	5.3±0.1	16.3±0.1	**37.7±3.8**	**11.1 kJ**
dense	5.4	**4.0±1.1**	**16.1±0.6**	44.6±5.0	13.3 kJ
SRigL	5.7	10.1±1.5	41.2±4.0	118.9±18.5	35.4 kJ
DynaDiag	5.6	15.4±0.4	47.3±1.1	127.3±8.5	37.9 kJ
SET	5.6	48.3±0.6	91.1±1.1	183.5±13.9	54.0 kJ

5.2 From kernel speedup to training speedup

At 95% sparsity on the L40 our kernel is 5.13× faster than a dense `bf16` GEMM, but training is 1.44× faster. Two measured factors account for the difference: an Amdahl (Amdahl, 1967) factor (÷1.27), because not all of a training step is sparsifiable GEMM, and the extra steps the sparse model needs to reach the target (×2.81). At 90% they cancel the kernel's advantage. On an A40 the step-efficiency factor transfers to within 3%, whereas the kernel factor must be measured at the training shape rather than at 8192^3 (Appendix H).

5.3 Peak memory

Measured as `max_memory_allocated` above the GPU-resident dataset, BlockDST uses 2459 MB against 5156 MB for dense (2.10× less; 1.85× with the dataset counted), while SRigL, SET and DynaDiag use 1.4–2.2× *more* than dense, because the standard implementation stores a dense weight, a mask, a gradient and the masked product. Of the implementations we compared, only BlockDST is both faster and smaller than dense (Figure 11).

5.4 Dependence on the accelerator

On two AD102 cards whose clocks are within 1.2% of each other but whose dense GEMM throughput differs $1.9\times$, BlockDST trains at nearly the same rate (13.1 and 13.4 steps/s at 90% with `bf16` storage, equal wall-clock runs not in the released files), as Equation (4) predicts, while dense training speeds up $1.7\times$ (5.6 and 9.5 steps/s). Our margin at 90% therefore falls from 1.22 pp on the L40 to a tie on the L40S: the advantage shrinks as the dense baseline gets faster, not as bandwidth changes.

We also repeated the time-to-target protocol on an A40 with the same code and seed, all six arms in one job. There BlockDST takes 1.4–$2.1\times$ *longer* than dense to reach 55% accuracy (56 against 39 s at 95%, 79 against 38 s at 90%; Table 16), a loss that the decomposition of Appendix H, fixed before this run, predicted from the A40's lower kernel speedup. With one seed this gives the direction, not a precise ratio: the training gains of Section 5.1 are specific to the L40.

5.5 Transformers and vision models

On a six-block Vision Transformer (ViT) (Dosovitskiy et al., 2021) of width 2048 (302M parameters, 99.9% in sparsifiable linear layers; CIFAR-10, 40 epochs, 90% sparsity, each arm at its best learning rate, three seeds), BlockDST reaches $80.41 \pm 0.10\%$ against $79.80 \pm 0.46\%$ for dense, parity at a tenth of the weights in matched epochs (dense's optimum may lie below the rates tested). On LLaMA-style (Touvron et al., 2023) decoders of about 411M and 604M dense-equivalent parameters with 95%-sparse feed-forward layers and dense attention (WikiText-103 (Merity et al., 2017), one hour, three seeds), BlockDST trains at $1.10\times$ and $1.16\times$ the dense step rate. Neither is a quality gain over the strongest baselines: SET and RigL are 1.1–1.6 pp more accurate on the ViT, and dense and 2:4 reach 5.4–13.6 lower perplexity on the language models (Appendix R).

6 Limitations

- **Hardware:** On an A40 the method is slower than dense; the A100 breaks even only at 95.0%, and our model predicts about 97% on an H100 (unmeasured), where the method would not pay. All cards are NVIDIA.
- **Statistics:** The $1.44\times$ gain over dense is significant only in post hoc analyses (pooled dense runs, interpolated crossings), and the A40 result and kernel timings are single runs (Appendix J).
- **Scope:** The method pays only when sparsifiable GEMMs dominate the step (93% here). A parameter-matched dense model and the same layout without rewiring train faster, tuned unstructured DST is more accurate per epoch (Appendix E), on transformers the method does not beat the strongest baselines, and the DST baselines are masked and untuned (Appendix O.1).
- **Mechanism:** We identify scaling with SM count $\times$ clock, not the limiting per-SM resource, for a kernel at 3–16% of tensor-core peak (counters and locked clocks need administrator rights we lack), and we do not identify the resource cuSPARSE follows either. The forward-only sweep is validated on five of six cards, and the cuSPARSE timings use smaller fp16 matrices (Appendix B.9).

Appendix Q discusses these and further limitations in detail.

7 Conclusion

A constant number of live blocks per block-row lets all three per-layer products run sparse, with $2.1\times$ less peak memory than dense. The training gains are narrow: $1.44\times$ to a fixed accuracy on an L40 (not significant at three seeds), a loss on an A40, and, on a laptop GPU, slower than a parameter-matched dense model and than the same layout without rewiring. Speedup scales as d^{-p} with $p = 0.84$ while the constant varies $5.4\times$ across cards, mostly through the dense baseline. For our kernel it can be predicted on an unseen card from its SM count, clock and one dense GEMM. cuSPARSE's kernel does not follow this scaling on the same cards, but its speedup also changes about twofold between them. A sparse speedup belongs to a method, a kernel, a card and a shape together, and should be reported with the SM count, clock, measured dense throughput and fitted exponent.

AI USE STATEMENT

In this work, we used generative AI tools to assist with grammar correction, improving writing clarity, debugging code, and identifying potentially relevant related works. We have not used generative AI tools to generate research hypotheses, design experiments or produce results of this work. Generation of experimental results, evaluation, interpretation of findings, and all scientific claims were conducted by the authors. Other disclosure categories are not applicable to this work. Additionally, we used generative AI tools to improve code readability and identify potential implementation issues during code development. All AI-assisted text and code suggestions were reviewed by the authors. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

All numbers come from released result files. The cost-model measurements (Sections 4 and 4.1) come from one benchmark run per card, and Table 1 and Figures 1 and 3 are generated from its outputs by script. The second-kernel comparison (Section 4.2) reuses the cuSPARSE timings of the format study, one run per configuration on five cards. The training results come from 60 time-to-target runs (48 on the L40, 12 on the A40), 45 equal wall-clock runs, 27 language-model runs, 26 ViT runs, 15 block-regularization runs and the small-scale ablations of Appendix E, each recording seed, hyperparameters and the full accuracy-versus-time trace. The regression intervals, break-even values and model comparisons are computed by script from the released sweep files. Appendix A lists hardware, software and architectures. The supplementary material contains the kernels, training code, protocol runners, analysis scripts and figure code, Appendix P maps every result to its files, and we will release the code publicly with the final version.

REFERENCES

Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS Spring Joint Computer Conference*, pp. 483–485, 1967.

Nathan Bell and Michael Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2009.

Davis Blalock, Jose Javier Gonzalez Ortiz, Jonathan Frankle, and John Guttag. What is the state of neural network pruning? In *Proceedings of Machine Learning and Systems (MLSys)*, 2020. arXiv:2003.03033.

A. Colin Cameron and Douglas L. Miller. A practitioner's guide to cluster-robust inference. *Journal of Human Resources*, 50(2):317–372, 2015.

A. Colin Cameron, Jonah B. Gelbach, and Douglas L. Miller. Bootstrap-based improvements for inference with clustered errors. *The Review of Economics and Statistics*, 90(3):414–427, 2008.

Roberto L. Castro, Andrei Ivanov, Diego Andrade, Tal Ben-Nun, Basilio B. Fraguela, and Torsten Hoefler. VENOM: A vectorized N:M format for unleashing the power of sparse tensor cores. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2023. arXiv:2310.02065.

Beidi Chen, Tri Dao, Kaizhao Liang, Jiaming Yang, Zhao Song, Atri Rudra, and Christopher Ré. Pixelated butterfly: Simple and efficient sparse training for neural network models. In *International Conference on Learning Representations (ICLR)*, 2022.

Selima Curci, Decebal Constantin Mocanu, and Mykola Pechenizkiy. Truly sparse neural networks at scale. *arXiv preprint arXiv:2102.01732*, 2021.

Tri Dao, Beidi Chen, Nimit Sohoni, Arjun Desai, Michael Poli, Jessica Grogan, Alexander Liu, Aniruddh Rao, Atri Rudra, and Christopher Ré. Monarch: Expressive structured matrices for efficient and accurate training. In *International Conference on Machine Learning (ICML)*, 2022.

Joshua Hoke Davis, Pranav Sivaraman, Joy Kitson, Konstantinos Parasyris, Harshitha Menon, Isaac Minn, Giorgis Georgakoudis, and Abhinav Bhatele. Taking GPU programming models to task for performance portability. In *Proceedings of the 39th ACM International Conference on Supercomputing*, ICS '25, pp. 776–791, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400715372. doi: 10.1145/3721145.3730423.

Mostafa Dehghani, Anurag Arnab, Lucas Beyer, Ashish Vaswani, and Yi Tay. The efficiency misnomer. In *International Conference on Learning Representations (ICLR)*, 2022. arXiv:2110.12894.

Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. ImageNet: A large-scale hierarchical image database. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2009.

Tim Dettmers and Luke Zettlemoyer. Sparse networks from scratch: Faster training without losing performance. *arXiv preprint arXiv:1907.04840*, 2019.

Nan Ding and Samuel Williams. An instruction roofline model for GPUs. In *IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, 2019.

Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. In *International Conference on Learning Representations (ICLR)*, 2021.

Utku Evci, Trevor Gale, Jacob Menick, Pablo Samuel Castro, and Erich Elsen. Rigging the lottery: Making all tickets winners. In Hal Daumé, III and Aarti Singh (eds.), *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pp. 2943–2952. PMLR, 13–18 Jul 2020.

Elias Frantar and Dan Alistarh. SparseGPT: Massive language models can be accurately pruned in one-shot. In *International Conference on Machine Learning (ICML)*, 2023.

Trevor Gale, Erich Elsen, and Sara Hooker. The state of sparsity in deep neural networks. *arXiv preprint arXiv:1902.09574*, 2019.

Trevor Gale, Matei Zaharia, Cliff Young, and Erich Elsen. Sparse GPU kernels for deep learning. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2020.

Trevor Gale, Deepak Narayanan, Cliff Young, and Matei Zaharia. MegaBlocks: Efficient sparse training with mixture-of-experts. In *Proceedings of Machine Learning and Systems (MLSys)*, volume 5, 2023. arXiv:2211.15841.

Scott Gray, Alec Radford, and Diederik P. Kingma. GPU kernels for block-sparse weights. Technical report, OpenAI, 2017.

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

Sture Holm. A simple sequentially rejective multiple test procedure. *Scandinavian Journal of Statistics*, 6(2):65–70, 1979.

Sara Hooker. The hardware lottery. *Communications of the ACM*, 64(12):58–65, 2021. arXiv:2009.06489.

Yuezhou Hu, Kang Zhao, Weiyu Huang, Jianfei Chen, and Jun Zhu. Accelerating transformer pre-training with 2:4 sparsity. In *International Conference on Machine Learning (ICML)*, 2024a.

Yuezhou Hu, Jun Zhu, and Jianfei Chen. S-STE: Continuous pruning function for efficient 2:4 sparse pre-training. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024b.

Itay Hubara, Brian Chmiel, Moshe Island, Ron Banner, Joseph Naor, and Daniel Soudry. Accelerated sparse neural training: A provable and efficient method to find N:M transposable masks. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2021.

Siddhant M. Jayakumar, Razvan Pascanu, Jack W. Rae, Simon Osindero, and Erich Elsen. Top-KAST: Top-K always sparse training. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2020. arXiv:2106.03517.

Peng Jiang, Lihan Hu, and Shihui Song. Exposing and exploiting fine-grained block structures for fast and accurate sparse training. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

Dhiraj Kalamkar, Dheevatsa Mudigere, Naveen Mellempudi, Dipankar Das, Kunal Banerjee, Sasikanth Avancha, Dharma Teja Vooturi, Nataraj Jammalamadaka, Jianyu Huang, Hector Yuen, Jiyan Yang, Jongsoo Park, Alexander Heinecke, Evangelos Georganas, Sudarshan Srinivasan, Abhisek Kundu, Misha Smelyanskiy, Bharat Kaul, and Pradeep Dubey. A study of BFLOAT16 for deep learning training. *arXiv preprint arXiv:1905.12322*, 2019.

Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, University of Toronto, 2009.

Mike Lasby, Anna Golubeva, Utku Evci, Mihai Nica, and Yani Ioannou. Dynamic sparse training with structured sparsity. In *International Conference on Learning Representations (ICLR)*, 2024. arXiv:2305.02299.

Namhoon Lee, Thalaiyasingam Ajanthan, and Philip H. S. Torr. SNIP: Single-shot network pruning based on connection sensitivity. In *International Conference on Learning Representations (ICLR)*, 2019.

Shiwei Liu, Tianlong Chen, Xiaohan Chen, Li Shen, Decebal Constantin Mocanu, Zhangyang Wang, and Mykola Pechenizkiy. The unreasonable effectiveness of random pruning: Return of the most naive baseline for sparse training. In *International Conference on Learning Representations (ICLR)*, 2022. arXiv:2202.02643.

Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In *International Conference on Learning Representations (ICLR)*, 2019.

Sangkug Lym, Donghyuk Lee, Mike O'Connor, Niladrish Chatterjee, and Mattan Erez. DeLTA: GPU performance model for deep learning applications with in-depth memory system traffic analysis. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2019.

Peter Mattson, Christine Cheng, Cody Coleman, Greg Diamos, Paulius Micikevicius, David Patterson, Hanlin Tang, Gu-Yeon Wei, Peter Bailis, Victor Bittorf, David Brooks, Dehao Chen, Debojyoti Dutta, Udit Gupta, Kim Hazelwood, Andrew Hock, Xinyuan Huang, Atsushi Ike, Bill Jia, Daniel Kang, David Kanter, Naveen Kumar, Jeffery Liao, Guokai Ma, Deepak Narayanan, Tayo Oguntebi, Gennady Pekhimenko, Lillian Pentecost, Vijay Janapa Reddi, Taylor Robie, Tom St. John, Tsuguchika Tabaru, Carole-Jean Wu, Lingjie Xu, Masafumi Yamazaki, Cliff Young, and Matei Zaharia. MLPerf training benchmark. In *Proceedings of Machine Learning and Systems (MLSys)*, 2020. arXiv:1910.01500.

Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models. In *International Conference on Learning Representations (ICLR)*, 2017.

Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. Mixed precision training. In *International Conference on Learning Representations (ICLR)*, 2018.

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

Decebal Constantin Mocanu, Elena Mocanu, Peter Stone, Phuong H. Nguyen, Madeleine Gibescu, and Antonio Liotta. Scalable training of artificial neural networks with adaptive sparse connectivity inspired by network science. *Nature Communications*, 9(1):2383, 2018.

Mahdi Nikdan, Tommaso Pegolotti, Eugenia Iofinova, Eldar Kurtic, and Dan Alistarh. SparseProp: Efficient sparse backpropagation for faster training of neural networks at the edge. In *International Conference on Machine Learning (ICML)*, 2023.

Patrik Okanovic, Sameer Deshmukh, Grzegorz Kwasniewski, Yi Zhu, Haruto Fujii, Sakina Fatima, Maciej Besta, Kentaro Katayama, Takumi Honda, Yusuke Nagasaka, and Torsten Hoefler. BLaST: High performance inference and pretraining using block sparse transformers. *arXiv preprint arXiv:2507.03117*, 2025.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2019.

Alexandra Peste, Eugenia Iofinova, Adrian Vladu, and Dan Alistarh. AC/DC: Alternating compressed/decompressed training of deep neural networks. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2021.

Hang Qi, Evan R. Sparks, and Ameet Talwalkar. Paleo: A performance model for deep neural networks. In *International Conference on Learning Representations (ICLR)*, 2017.

Franklin E. Satterthwaite. An approximate distribution of estimates of variance components. *Biometrics Bulletin*, 2(6):110–114, 1946.

Noam Shazeer. GLU variants improve transformer. *arXiv preprint arXiv:2002.05202*, 2020.

Jianlin Su, Murtadha Ahmed, Yu Lu, Shengfeng Pan, Wen Bo, and Yunfeng Liu. RoFormer: Enhanced transformer with rotary position embedding. *Neurocomputing*, 568:127063, 2024.

Mingjie Sun, Zhuang Liu, Anna Bair, and J. Zico Kolter. A simple and effective pruning approach for large language models. In *International Conference on Learning Representations (ICLR)*, 2024.

Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In *International Conference on Machine Learning (ICML)*, 2013.

Hidenori Tanaka, Daniel Kunin, Daniel L. K. Yamins, and Surya Ganguli. Pruning neural networks without any data by iteratively conserving synaptic flow. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2020.

Philippe Tillet, H. T. Kung, and David Cox. Triton: An intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL)*, 2019.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurélien Rodriguez, Armand Joulin, Édouard Grave, and Guillaume Lample. LLaMA: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

Abhishek Tyagi, Arjun Iyer, William H. Renninger, Christopher Kanan, and Yuhao Zhu. Dynamic sparse training of diagonally sparse networks. In *International Conference on Machine Learning (ICML)*, pp. 60469–60492, 2025a. arXiv:2506.11449.

Abhishek Tyagi, Arjun Iyer, Liam Young, William H. Renninger, Christopher Kanan, and Yuhao Zhu. ShuffleSparse: Learned shuffles for structured sparse networks. *arXiv preprint arXiv:2510.14812*, 2025b.

Nasib Ullah, Jinbin Zhang, Jean Lucien Randrianantenaina, Erik Schultheis, and Rohit Babbar. HASTE: Hardware-aware dynamic sparse training for large output spaces. In *International Conference on Machine Learning (ICML)*, 2026. arXiv:2606.01117.

Vasily Volkov. *Understanding Latency Hiding on GPUs*. PhD thesis, University of California, Berkeley, 2016.

Chaoqi Wang, Guodong Zhang, and Roger Grosse. Picking winning tickets before training by preserving gradient flow. In *International Conference on Learning Representations (ICLR)*, 2020.

Lei Wang, Yu Cheng, Yining Shi, Zhengju Tang, Zhiwen Mo, Wenhao Xie, Lingxiao Ma, Yuqing Xia, Jilong Xue, Fan Yang, and Zhi Yang. TileLang: A composable tiled programming model for AI systems. *arXiv preprint arXiv:2504.17577*, 2025.

Matthew D. Webb. Reworking wild bootstrap-based inference for clustered errors. *Canadian Journal of Economics*, 56(3):839–858, 2023.

Bernard L. Welch. The generalization of 'Student's' problem when several different population variances are involved. *Biometrika*, 34(1–2):28–35, 1947.

Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Learning structured sparsity in deep neural networks. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2016.

Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

Chenpeng Wu, Qiqi Gu, Heng Shi, Jianguo Yao, and Haibing Guan. Samoyeds: Accelerating MoE models with structured sparsity leveraging sparse tensor cores. In *Proceedings of the Twentieth European Conference on Computer Systems*, EuroSys '25, pp. 293–310, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400711961. doi: 10.1145/3689031.3717455.

Qiao Xiao, Boqian Wu, Patrik Okanovic, Tomasz Sternal, Maurice van Keulen, Elena Mocanu, Mykola Pechenizkiy, Decebal Constantin Mocanu, and Torsten Hoefler. Memory-efficient LLM training with dynamic sparsity: From stability to practical scaling. In *International Conference on Machine Learning (ICML)*, 2026. arXiv:2606.00888.

Charlene Yang, Thorsten Kurth, and Samuel Williams. Hierarchical roofline analysis for GPUs: Accelerating performance optimization for the NERSC-9 Perlmutter system. *Concurrency and Computation: Practice and Experience*, 32(20), 2020.

Zhuliang Yao, Shijie Cao, Wencong Xiao, Chen Zhang, and Lanshun Nie. Balanced sparsity for efficient DNN inference on GPU. In *AAAI Conference on Artificial Intelligence*, 2019. arXiv:1811.00206.

Lu Yin, Gen Li, Meng Fang, Li Shen, Tianjin Huang, Zhangyang Wang, Vlado Menkovski, Xiaolong Ma, Mykola Pechenizkiy, and Shiwei Liu. Dynamic sparsity is channel-level sparsity learner. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

Andy B. Yoo, Morris A. Jette, and Mark Grondona. SLURM: Simple Linux utility for resource management. In *Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, pp. 44–60, 2003.

Geoffrey X. Yu, Yubo Gao, Pavel Golikov, and Gennady Pekhimenko. Habitat: A runtime-based computational performance predictor for deep neural network training. In *USENIX Annual Technical Conference (USENIX ATC)*, 2021.

Geng Yuan, Xiaolong Ma, Wei Niu, Zhengang Li, Zhenglun Kong, Ning Liu, Yifan Gong, Zheng Zhan, Chaoyang He, Qing Jin, Siyue Wang, Minghai Qin, Bin Ren, Yanzhi Wang, Sijia Liu, and Xue Lin. MEST: Accurate and fast memory-economic sparse training framework on the edge. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2021.

Ming Yuan and Yi Lin. Model selection and estimation in regression with grouped variables. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 68(1):49–67, 2006.

Biao Zhang and Rico Sennrich. Root mean square layer normalization. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2019.

APPENDIX

Table of Contents

A Experimental setup 15
B The cost model in detail 16
B.1 Arithmetic intensity . . . 16
B.2 Per-card results for both models . . . 16
B.3 What six cards can and cannot identify . . . 17
B.4 Sustained clock and repeatability . . . 19
B.5 Is the forward-only sweep a fair proxy for a training step? . . . 19
B.6 Testing the bandwidth hypothesis . . . 19
B.7 Figure 1 at 90% sparsity . . . 20
B.8 Break-even depends on shape as well as card . . . 20
B.9 A second kernel: cuSPARSE Blocked-ELL on five cards . . . 20
C Choice of sparse format 21
D Block regularization 22
E Small-scale ablations 23
F Reporting recommendation 24
G Kernel benchmarks 25
G.1 Numerical precision . . . 25
G.2 Is the kernel itself competitive? . . . 26
H From kernel speedup to training speedup 26
H.1 Transfer to a second accelerator . . . 27
I Accuracy against training FLOPs 28
J Statistical treatment 28
K Peak memory 29
L Equal wall-clock results 30
M Hyperparameters 31
N Measurement protocol 32
O Comparison with related methods 32
O.1 Kernel-accelerated baselines . . . 32
O.2 Other related work . . . 33
O.3 Reported speedups . . . 33
P Provenance 33
Q Further limitations 34
R Transformer and vision-model results 35
R.1 Vision transformer with matched epochs . . . 35
R.2 Language models . . . 36
R.3 Vision transformer with equal wall-clock budgets . . . 36

A Experimental setup

Models and datasets. Table 3 lists every model and dataset. CIFAR-10 has 50,000 training and 10,000 test images (32×32 color, 10 classes); we normalize each channel and, during training, apply a random crop with 4-pixel padding and a random horizontal flip on the GPU. WikiText-103 is tokenized once and stored as a memory-mapped array. Training hyperparameters for the MLP runs are in Table 22; the Vision Transformer (ViT) runs use AdamW (Loshchilov & Hutter, 2019) (weight decay 0.05, 5% warm-up, batch 256, 40 epochs) and the language models use AdamW (learning rate 3×10^{-4}, batch 8, sequence length 1024).

The language models use RMSNorm (Zhang & Sennrich, 2019), rotary position embeddings (Su et al., 2024) and SwiGLU (Shazeer, 2020) feed-forward networks (FFNs); the parameter counts are for the dense-equivalent model (the stored sparse models have 214M and 302M parameters).

Hardware. Table 4 lists the six graphics processing units (GPUs). The cluster nodes have 32- or 64-core central processing units (CPUs) and 256 GB of memory; the RTX A500 is a laptop GPU.

Table 3: Models and datasets. Sparsified layers use BlockDST blocks of the size shown; all other layers are dense.

name	architecture	dataset	sparsified layers	results
MLP	$3072 \to 16{,}384 \to 16{,}384 \to 10$, ReLU, 319M parameters	CIFAR-10	both hidden layers ($b = 64$)	Section 5
ViT-2048	6 blocks, width 2048, 32 heads, MLP 8192, patch 4	CIFAR-10	attention and MLP projections ($b = 32$)	Table 25
ViT-3072	6 blocks, width 3072, 48 heads, MLP 12,288, patch 4	CIFAR-10	attention and MLP projections ($b = 32$)	Table 27
LM-411M	LLaMA-style, 6 layers, d_{model} 2048, 16 heads, SwiGLU 5632	WikiText-103	FFN projections ($b = 32$)	Table 26
LM-604M	LLaMA-style, 6 layers, d_{model} 2560, 20 heads, SwiGLU 6912	WikiText-103	FFN projections ($b = 32$)	Table 26
Kernel sweep	single layer, 3 shapes $\times$ 3 densities per card	synthetic	the layer ($b = 64$)	Table 1
Ablations	MLP, width 1024–8192 (and 16,384)	CIFAR-10	both hidden layers ($b = 16$–128)	Appendix E

Table 4: GPUs used. Bandwidth is measured with a read-only probe on each card.

GPU	architecture (chip)	memory	bandwidth	used for
RTX A500 Laptop	Ampere (GA107)	4 GB GDDR6	107 GB/s	kernel sweep; sparse-format study; ablations (Appendices D and E)
A40	Ampere (GA102)	48 GB GDDR6	584 GB/s	kernel sweep; time-to-target on a second card
A100-PCIe-40GB	Ampere (GA100)	40 GB HBM2e	1146 GB/s	kernel sweep
L40	Ada Lovelace (AD102)	48 GB GDDR6	729 GB/s	kernel sweep; all main training results; peak memory
L40S	Ada Lovelace (AD102)	48 GB GDDR6	720 GB/s	kernel sweep; L40S comparison
RTX PRO 6000 Blackwell	Blackwell (GB202)	96 GB GDDR7	1482 GB/s	kernel sweep; language models

Software. Cluster jobs run on Linux under the Slurm scheduler (Yoo et al., 2003) with the CUDA 12.8 toolkit, PyTorch (Paszke et al., 2019) built for CUDA 12.8, Python 3.11 and GCC 11.4; our kernels are compiled once for compute capabilities 8.0, 8.6, 8.9, 9.0 and 12.0. The RTX A500 runs use Windows 11 with NVIDIA driver 595.71 and PyTorch 2.13 (CUDA 12.6). Timings use CUDA events or a host timer after device synchronization, exclude evaluation, and each comparison runs all arms sequentially on one GPU (Appendix N); energy is read from the NVIDIA Management Library (NVML).

B The cost model in detail

B.1 Arithmetic intensity

Following the benchmark code, and unlike Section 3, here M is the batch: for an activation matrix with M rows and K columns, a block-sparse weight with K inputs and N outputs, block size b and k live blocks per block-row (the weight holds Nkb values), the arithmetic intensity of the forward product is

$$A \;=\; \frac{2MNkb}{s_x\,(MNk) \;+\; s_w\,(Nkb) \;+\; s_y\,(MN)}, \tag{3}$$

where s_x, s_w and s_y are the element sizes of activation, weight and output (4 bytes each in the sweep). The model assumes every activation tile is read from DRAM once per live block, with no cache reuse. Dividing by MN shows that A depends only on M, k and b, not on K or N. The shapes $(8192, 8192, 8192)$ and $(8192, 4096, 11008)$ at k=6 give identical values ($A = 27.25$ FLOP/byte) in our data. Over the density range we measure, A runs from 15.94 (k=1) to 29.50 (k=13), a $1.85\times$ change driven by density, while the effect of geometry at fixed k is 0.7%. This mild density dependence is why the fitted exponent is 0.84 rather than 1.

B.2 Per-card results for both models

Table 5 lists both models card by card. The leave-one-card-out column is the only out-of-sample number and is the one to judge the model by. The attained fraction of the bandwidth roofline, η, is shown for completeness; it equals $1/(BA/D$ error$)$ by construction.

The A100 is the only card on which the copy probe (1308 GB/s) is faster than the read probe (1146 GB/s); on the other five, copy is 5–12% slower, as expected, and the A100 read probe reaches 74% of datasheet peak against 83–96% elsewhere. The A100's leave-one-card-out residual ($+17\%$) is close to this probe anomaly ($+14\%$), so the model's largest miss may partly reflect the measurement of that card. Separating the two would need clock and power telemetry recorded during the sweep.

Table 5: Per-card results: Nine measurements per card. B is measured with a read-only probe, D is the dense `bf16` throughput on the same shapes, and $\mathrm{SM}\cdot f$ is SM count times clock. The last three columns are prediction errors of the bandwidth model, of the SM-count model fitted on all six cards, and of the SM-count model fitted on the other five (leave-one-card-out, LOCO). Brackets give per-measurement ranges.

card	silicon / mem	B (GB/s)	$\mathrm{SM}\cdot f$	D	η	BA/D	$\mathrm{SM}\cdot f$	LOCO
RTX A500	GA107 / GDDR6	107	28	21 T	73% [69–80]	1.45×	**1.01×**	1.02×
A100-PCIE	GA100 / HBM2e	1146	152	213 T	35% [30–44]	**3.26×**	1.16×	1.19×
A40	GA102 / GDDR6	584	146	124 T	71% [67–76]	1.48×	**0.98×**	0.97×
RTX PRO 6000	GB202 / GDDR7	1482	457	428 T	95% [88–109]	1.13×	**0.96×**	0.91×
L40S	AD102 / GDDR6	720	358	209 T	123% [99–139]	0.72×	**0.95×**	0.94×
L40	AD102 / GDDR6	729	354	109 T	128% [114–136]	0.75×	**0.96×**	0.96×

Table 6: Model comparison, predicted over measured c. In-sample, k_0 is the mean over the six reference-shape rows and is applied to all 54 rows, which is why its all-rows error exceeds the leave-one-card-out error; leave-one-card-out rows fit one scale (geometric mean) on five cards, on the same kind of rows they are evaluated on, and predict the sixth. "Constant kernel rate" predicts from the dense throughput D alone. Generated by `figures/model_compare.py` and `figures/review_stats.py`.

	error range	
model	reference shape	all 54 rows
in-sample		
$c = BA/D$, no free parameter	—	0.72–3.33×
$c = BA/D$, one fitted scale	—	0.58–2.66×
$c = k_0(\mathrm{SM}\cdot f)/D$	0.93–1.16×	0.93–1.89×
$c = k(\mathrm{SM}\cdot f)\,d^{0.19}/D$	—	0.81–1.45×
leave-one-card-out		
constant kernel rate, $c = k/D$	0.30–9.50×	0.26–13.0×
$c = k\,BA/D$	0.51–3.08×	0.53–3.15×
$c = k_0(\mathrm{SM}\cdot f)/D$	**0.91–1.19×**	0.78–1.66×
$c = k(\mathrm{SM}\cdot f)\,d^{0.19}/D$	0.91–1.19×	**0.79–1.47×**
$c = k\,\mathrm{SM}/D$ (SM count alone)	0.72–1.82×	0.62–1.98×
$c = kf/D$ (clock alone)	0.38–8.15×	0.33–11.2×

B.3 What six cards can and cannot identify

Across our six cards $\mathrm{corr}(\log \mathrm{SM}\cdot f, \log B) = 0.88$, and 0.96 without the A100. SM count × clock is also proportional, on every one of these chips, to warp-scheduler issue slots, load-store throughput, L1/shared-memory bandwidth and register-file bandwidth. The data therefore identify that throughput scales with SM count and not with bandwidth, but not which per-SM resource binds. A locked-clock sweep on one card, pinning the SM clock while varying the memory clock and then the reverse, would separate them without new hardware, but locking clocks and reading performance counters both need administrator rights: the cluster restricts profiling to administrators and Nsight Compute returns `ERR_NVGPUCTRPERM` on our laptop as well.

Leaving out the A100 changes $\hat{\beta}_B$ from $+0.04$ to $+0.28$, while leaving out any other card changes it by at most 0.07. The A100 is the card that breaks the trend, the card the model misses most, and the card with the unusual bandwidth probe, so a second HBM-based card would be the most useful addition to this study.

The regression. Regressing $\log(\text{kernel TFLOP/s})$ on $\log(\mathrm{SM}\cdot f)$, $\log B$ and $\log d$ over all 54 rows gives, with standard errors (SE) clustered by card (Cameron & Miller, 2015),

$$\hat{\beta}_{\mathrm{SM}\cdot f} = +0.97\ (\mathrm{SE}\ 0.07), \quad \hat{\beta}_B = +0.04\ (\mathrm{SE}\ 0.08), \quad \hat{\beta}_d = +0.19\ (\mathrm{SE}\ 0.02), \quad R^2 = 0.986. \tag{4}$$

$\hat{\beta}_d \approx 1 - p$ is the density dependence the exponent captures; its wild-bootstrap interval is $[0.12, 0.25]$.

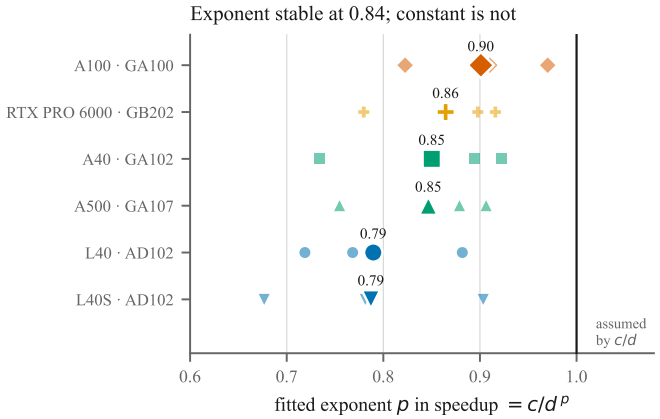

Figure 6: Exponent of the speedup law, fitted per (card, geometry). Small markers are individual fits and large markers per-card means. All 18 fits lie below 1; the per-card means are similar (0.79–0.90, 95% intervals across geometries: A100 $[0.72, 1.09]$, A40 $[0.60, 1.10]$, L40 $[0.58, 1.00]$, L40S $[0.51, 1.07]$, A500 $[0.65, 1.05]$, RTX PRO 6000 $[0.68, 1.05]$, all overlapping) while c varies $5.4\times$. With three geometries per card these intervals are wide and each includes or touches 1, so the data show a similar exponent, not an identical one; that $p < 1$ rests on the pooled density coefficient of Equation (4).

Inference with six clusters. The 54 rows are nine measurements of each of six cards, and the card-level regressors take only six values, so the effective sample for the hardware question is six. Cluster-robust standard errors are biased downward with so few clusters (Cameron & Miller, 2015). A wild cluster bootstrap-t with Webb six-point weights (Cameron et al., 2008; Webb, 2023) (9999 draws) gives 95% intervals of $[0.75, 1.19]$ for $\hat{\beta}_{\mathrm{SM}\cdot f}$ and $[-0.23, 0.30]$ for $\hat{\beta}_B$; a card-level regression on the six reference-shape points gives $+1.13$ (SE 0.05) and -0.12 (SE 0.06) with three residual degrees of freedom. The variance inflation factor of both regressors is 4.4. Bandwidth adds 1.7% of partial R^2 given $\mathrm{SM}\cdot f$ and density, while $\mathrm{SM}\cdot f$ adds 93.5% given bandwidth and density. With log density as the only other term, SM count alone explains $R^2 = 0.94$, clock alone 0.42, their product 0.985, bandwidth 0.78 and dense throughput 0.78. The data therefore favor per-SM scaling clearly, but they bound the bandwidth coefficient only to $[-0.23, 0.30]$, not to zero.

How far the kernel is from peak. At the reference point the kernel reaches 3.3% of the datasheet dense `bf16` tensor-core peak on the A100, 7.8% on the A40, 8.1% on the L40S and 15.7% on the L40, while cuBLAS reaches 58–83% on the same four cards. The L40's datasheet rate for `bf16` with fp32 accumulation is half the L40S's (181 against 362 TFLOP/s) on the same die, which halves its dense baseline but not our kernel, and explains most of the L40's advantage. The kernels use `wmma` loads through shared memory without asynchronous copies (`cp.async`) or `ldmatrix`, and `cuobjdump --dump-resource-usage` on the sm_86 build reports 56 registers per thread and 16 KB of shared memory for the forward kernel and 64 registers for the block-SDDMM, with no spills; at 512 threads per block, registers allow two blocks, that is 32 resident warps per SM on every card (two-thirds of the 48-warp limit, half of the A100's 64). Occupancy is thus fixed per SM, which is consistent with throughput proportional to SM count, and a kernel this far from peak can be limited by latency or instruction issue rather than by any memory level. The law and cost model describe this kernel. cuSPARSE's Blocked-ELL kernel, which keeps the tensor cores busier, does not follow the same scaling on five of these cards (Appendix B.9).

Prediction for Hopper. Our kernels compile for sm_90 but use no Hopper-specific instructions. With $k_0 = 0.078$, an H100 SXM (132 SMs, 1.98 GHz) would run the kernel at about 20 TFLOP/s; against a dense `bf16` throughput of 600–700 TFLOP/s this gives $c \approx 0.03$ and a break-even near 97% sparsity, outside the range where the method trains well. We have not measured an H100, so this is a testable prediction rather than a result.

For the A40 (1740 MHz), L40S (2520 MHz) and RTX PRO 6000 (2430 MHz) we use the maximum SM clock reported by the driver; for the A100, L40 and RTX A500 we use datasheet boost clocks.

The sustained clock under load was not recorded during the cluster sweeps, and a laptop A500 or a 250 W A100-PCIE may not hold its boost clock throughout; Appendix B.4 measures the A500's sustained clock and the sensitivity of the fit to it.

B.4 Sustained clock and repeatability

We reran the full RTX A500 sweep while logging `clocks.sm` and `power.draw` every 250 ms. Under load the card held a median SM clock of 1387 MHz (10th–90th percentile 1327–1822) at 44 W, 22% below the 1.77 GHz boost clock used for its $\mathrm{SM}\cdot f$. The rerun, weeks after the released sweep, reproduces every row: kernel throughput to within 3.4% (median 0.5%), dense throughput to within 8.0% (median 1.4%) and c to within 7.1% (median 0.9%). Using the sustained clock for the A500 changes the regression of Equation (4) to $\hat{\beta}_{\mathrm{SM}\cdot f} = +0.93$ and $\hat{\beta}_B = -0.02$ (wild-bootstrap intervals $[0.73, 1.13]$ and $[-0.31, 0.27]$) and the leave-one-card-out error at the reference shape from 0.91–$1.19\times$ to 0.80–$1.25\times$. The conclusions do not depend on which clock is used for this card; the cluster cards' sustained clocks were not logged (`data/clock/`, `experiments/run_review_local2.sh`).

Two cluster cards were also swept a second time, a month after the released sweep and on the same card types (`data/rerun/`). On the RTX PRO 6000 every row's c reproduces within 5.4% (median 1.4%). On the L40S the reference row reproduces within 3% and the median row within 6%, but three of the nine rows differ by 11–16% in c, one of them by 39% in kernel throughput. Single timings on the cluster therefore carry errors of this size on some rows, comparable to the spread of the leave-one-card-out errors across shapes (0.78–$1.66\times$), part of which is timing noise.

B.5 Is the forward-only sweep a fair proxy for a training step?

The six-card sweep times the forward product. Five of the six cards also have per-layer benchmarks that time forward and backward passes on the same shapes (Table 7); the RTX A500 row was added for this revision.

Table 7: Forward-only proxy against forward+backward timings at 8192^3, block 64, $d = 0.102$, batch 8192. Generated by `figures/fwd_vs_fwdbwd.py`.

card	forward only	forward+backward	ratio
A100	0.48×	0.57×	1.19×
A40	0.92×	0.93×	1.01×
L40S	1.38×	1.38×	1.00×
RTX PRO 6000	0.89×	0.93×	1.04×
RTX A500	1.00×	1.10×	1.10×

The proxy is accurate to 1.00–$1.19\times$ (mean $1.07\times$) and preserves the ordering of the five cards; where it errs, the forward product understates the training-step speedup. Including the backward pass moves the A100 from $0.48\times$ to $0.57\times$, still well below break-even, and the RTX A500 from $1.00\times$ to $1.10\times$. The L40, the card at the top of the headline range, has no per-layer benchmark, so its forward-only figure is not validated.

B.6 Testing the bandwidth hypothesis

We tested the bandwidth hypothesis with timing alone. Our shapes are chosen so that the working set exceeds twice the largest L2 cache; shrinking the problem until the working set fits in L2 removes nearly all dynamic random-access memory (DRAM) traffic without changing the kernel, so a jump in throughput at the L2 boundary would indicate a DRAM bound. We ran this on an L40 over six sizes with working sets (WS) of 24–1063 MB, straddling its 96 MB L2, at three densities each.

The experiment does not separate the two explanations. Throughput rises with problem size (from 4.2 to 24.3 TFLOP/s over this range), and on one GPU the working set cannot be varied without varying problem size, so "fits in L2" coincides with "too small to fill the machine". Fitting $\log(\mathrm{TFLOP/s}) = a + b\log(\mathrm{WS}) + c\log d + s\,\mathbf{1}[\mathrm{WS} < L2]$ gives $s = -0.47$ (95% confidence interval (CI) $[-0.87, -0.07]$), but adding a quadratic term in $\log(\mathrm{WS})$ gives $s = -0.03$ ($[-0.29, +0.23]$)

and raises R^2 from 0.89 to 0.97, so the apparent step is curvature. Varying cache residency at a fixed problem size needs a profiler, a locked memory clock, or a second card with a different L2. This also bears on the attained fraction η of Section 4.1: the traffic model charges every activation re-read to DRAM, so $\eta > 1$ on the large-L2 AD102 cards (123–128%) may reflect L2 hits rather than an impossible bandwidth, and we use it only as evidence that the DRAM traffic model is wrong. Resource usage from the compiled object is reported in Appendix B.3.

B.7 Figure 1 at 90% sparsity

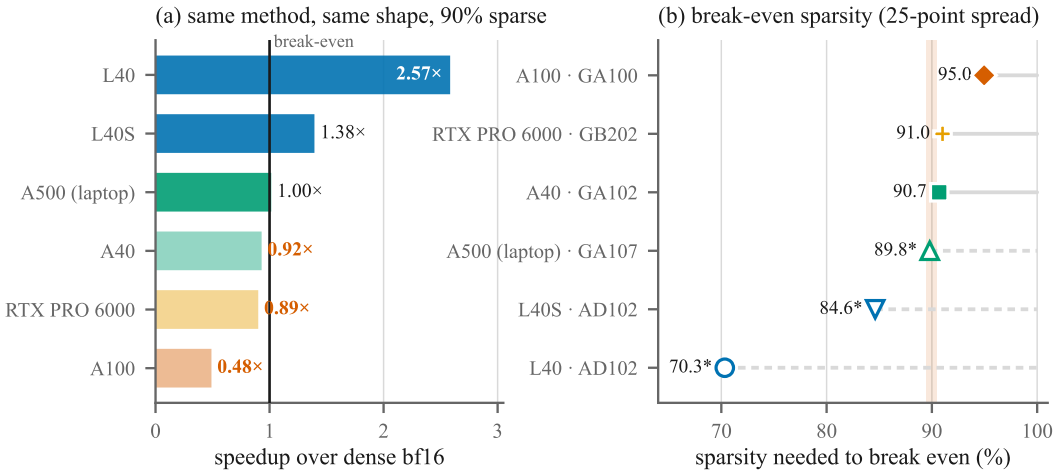

Figure 7: Figure 1 at 90% sparsity: **(a)** Same kernel and 8192^3 shape: four of the six accelerators are at or below break-even, from 2.57× on the L40 to 0.48× on the A100. **(b)** The same break-even values as Figure 1(b), with 90% highlighted.

The main text shows 95% sparsity, the operating point the method is designed for. At 90%, the same kernel wins on only two of the six cards, so the figure answers whether there is a speedup at all rather than how large it is.

B.8 Break-even depends on shape as well as card

Since $c \equiv \text{kernel TFLOP/s}/D$, c inherits every dependence of the dense baseline, and on one L40 the dense throughput varies 2.0× across square shapes (Table 8). It peaks around 2880–4096^3 and falls again, because cuBLAS is tuned for a band of shapes. A break-even sparsity measured at one shape therefore does not carry over to another shape on the same card, and the per-card values in Table 1 hold at the 8192^3 reference shape.

Table 8: Dense `bf16` throughput D on one L40 across square shapes.

shape	1024^3	1408^3	2048^3	2880^3	4096^3	5760^3
D (TFLOP/s)	76.6	106.6	138.4	141.1	138.2	107.9

B.9 A second kernel: cuSPARSE Blocked-ELL on five cards

The format study of Appendix C timed cuSPARSE's Blocked-ELL SpMM against a cuBLAS GEMM with fp16 inputs and fp32 accumulation, on the RTX A500 and, through our per-card kernel job, on the A100, A40, L40S and RTX PRO 6000; that job never landed on an L40. Each card has 90 configurations (weight $M \times K$ with $M = K \in \{1024, 2048, 4096\}$, batch $N \in \{512, 2048\}$, blocks of 16, 32 and 64, five densities), timed as the mean of 50 iterations after 10 warm-up iterations, one run per cell. The sparse kernel reads and writes fp16 and accumulates in fp32. We took these timings for the format study, before the cost model existed, so they were not collected to test it. We use the

18 configurations at $M = K = 4096$ with densities 0.25, 0.1 and 0.05, because the smaller products last only tens of microseconds. Table 9 gives the reference configuration, batch 2048 and $b = 64$ at density 0.094 (6 of 64 block-columns), next to our kernel at its own reference point.

Table 9: cuSPARSE Blocked-ELL on five cards at weight 4096^2, batch 2048, $b = 64$, density 0.094. D is the dense fp16 GEMM with fp32 accumulation on the same shape; the last column is our kernel at 8192^3, $d = 0.102$ (Table 1). Throughput per SM-GHz and per GB/s is in GFLOP/s. Generated by `figures/cusparse_crosscard.py`.

card	SM$\cdot f$	B (GB/s)	D (TFLOP/s)	cuSPARSE (TFLOP/s)	per SM-GHz	per GB/s	speedup	ours, per SM-GHz
RTX A500	28	107	21.9	5.5	194	51	2.68×	77
A40	146	584	103.1	34.6	237	59	3.58×	80
A100	152	1146	257.0	69.5	**456**	61	2.88×	68
L40S	358	720	241.5	82.2	230	114	3.63×	82
RTX PRO 6000	457	1482	356.4	167.8	367	113	5.02×	85

Our kernel runs at 68–85 GFLOP/s per SM-GHz on these five cards, cuSPARSE at 194–456, a $2.35\times$ range. Over the 18 configurations the coefficient of variation of cuSPARSE's throughput per SM-GHz across the five cards has a median of 33% (22–50%), against 8.7% (5.6–17.4%) for our kernel over the nine shape–density rows the five cards share. Leave-one-card-out with one fitted constant, as in Table 6, gives 0.56–$1.66\times$ at the reference configuration for $c = k_0(\mathrm{SM}\cdot f)/D$ and 0.59–$1.61\times$ for $c = kBA/D$ (with A from Equation (3) for 2-byte elements). Over all 150 rows at $M = K = 4096$ (30 per card, with a scale fitted per configuration on the other four cards) the SM-count model's 10–90% error range is 0.61–$1.53\times$ and its full range 0.46–$3.14\times$. For our kernel the same procedure gives 0.91–$1.19\times$ and 0.78–$1.66\times$.

The A100 and A40 have almost the same SM count $\times$ clock (152 and 146 SM-GHz), while the A100 has twice the tensor-core width per SM and twice the bandwidth. Over the 18 configurations cuSPARSE runs $1.46\times$ faster on the A100 (median; 1.19–$2.11\times$, and 1.7–$2.1\times$ with blocks of 32 and 64 at batch 2048), and our kernel $0.95\times$ as fast (0.88–$1.13\times$ over nine rows). Neither SM count $\times$ clock nor DRAM bandwidth describes cuSPARSE alone. On the three Ampere cards it runs at 51–61 GFLOP/s per GB/s while its rate per SM-GHz varies $2.4\times$, and on the L40S and RTX PRO 6000, whose L2 caches (96 and 128 MB) are several times the 37 MB working set of the reference configuration, it runs at 113–114 GFLOP/s per GB/s. That pattern fits a kernel limited by memory traffic, but on these cards bandwidth and tensor-core width move together, so we do not claim a mechanism for it.

The comparison is not fully controlled. Our sweep keeps every working set above twice the L2 size, while these smaller fp16 products fit in the L2 of the A100 (40 MB), L40S and RTX PRO 6000 but not of the A40 (6 MB) or RTX A500, so part of cuSPARSE's A100 advantage may come from cache residency rather than from the kernel. The L40S and RTX PRO 6000 do not separate the two kernels: per SM-GHz, cuSPARSE runs $1.26\times$ faster on the RTX PRO 6000 (median over the 18 configurations), and our kernel $1.25\times$ faster over its nine DRAM-sized rows. We therefore read this section as showing that the SM-count constants do not transfer to another kernel and shape regime, not as identifying what limits cuSPARSE.

The dependence on the card holds for cuSPARSE as well. Its speedup at the reference configuration runs from $2.68\times$ to $5.02\times$, and its constant c spans $1.87\times$ across the five cards (median over the 18 configurations $1.99\times$, range 1.50–$3.02\times$). For our kernel on the same five cards the spread is $2.88\times$ at its reference point (median over rows $2.19\times$), so the two kernels depend on the card by similar amounts for different reasons. Fitted over densities 0.25 to 0.02, cuSPARSE's exponent is 0.63 ± 0.11 over 30 fits, lower than our 0.84, partly because at 98% sparsity these products last a few microseconds and fixed costs flatten the speedup; we do not use it elsewhere. The RTX A500 file was produced under Windows with CUDA 12.6 and the cluster files under Linux with CUDA 12.8, so the cuSPARSE versions differ.

C Choice of sparse format

We first measured which sparse formats in NVIDIA's libraries turn sparsity into speed. The benchmark (`bench/kernels/block_spmm_bench.cu`) computes $C = A \cdot B$ on an RTX A500 with A block-sparse at a constant number of blocks per block-row, over 90 configurations (six shapes, three

block sizes, five densities), with 10 warm-up and 50 timed iterations. Every sparse result matches the cuBLAS dense result to within 2.2×10^{-6} relative error. Each sparse kernel is compared with a dense baseline of the same precision: cuSPARSE CSR SpMM and legacy BSR (`bsrmm`) run in fp32 against cuBLAS single-precision GEMM (SGEMM), and cuSPARSE Blocked-ELL SpMM runs on tensor cores in fp16 against cuBLAS half-precision GEMM (HGEMM).

Table 10: Speedup of three cuSPARSE formats over a dense GEMM of the same precision by sparsity (RTX A500, 4096×4096 weight, batch 2048, 64×64 blocks). CSR and BSR run in fp32 without tensor cores; Blocked-ELL runs in fp16 on tensor cores. Generated by `figures/format_feasibility.py` from `bench/results/spmm_sweep.csv`.

format	50%	75%	90%	95%	98%
CSR (fp32)	0.05×	0.10×	0.25×	0.48×	1.50×
BSR (fp32)	0.21×	0.42×	1.09×	2.17×	6.26×
Blocked-ELL (fp16)	0.82×	**1.26×**	**2.68×**	**4.30×**	**7.93×**

Across all 90 configurations, CSR beats its dense baseline in 11 cases, all at 98% sparsity (best $1.58\times$); BSR in 47 cases, from 90% sparsity (best $6.47\times$); and Blocked-ELL in 74 cases (best $7.93\times$). Unstructured CSR is $4\times$ slower than dense at 90% sparsity, BSR does not use tensor cores, and Blocked-ELL is the only format that is fast in the 90–98% range where DST operates. Its single requirement, a constant number of blocks per block-row, became our invariant. We wrote our own kernels on this layout rather than calling cuSPARSE because the training loop needs all three products and cuSPARSE provides no block-SDDMM for the weight gradient. The same benchmark also ran on four of the cluster cards; Appendix B.9 uses those timings to test the cost model on cuSPARSE's kernel.

Other ways of exploiting sparsity are compared at the training level in Section 5: unstructured masks (SET), constant fan-in (SRigL) and diagonal structure (DynaDiag), all of which run at dense speed in their standard implementations, and hardware 2:4 sparsity on the language models (Appendix R). At the kernel level, Appendix G compares our kernel with blocked compressed sparse column (CSC) storage (BLaST) and PyTorch's Triton block-sparse (BSR) kernel.

D Block regularization

Group lasso (Yuan & Lin, 2006), a penalty on the norm of each block, is the usual way to make a dense network block-sparse (Wen et al., 2016). We tested whether adding it to BlockDST helps, with cross-entropy loss $\mathcal{L}_{\text{CE}} + \lambda \sum_{\text{layers}} \sum_{\text{live } j} \|W_j\|_F$ over the live blocks W_j. The setup is the MLP of Table 3 at width 4096 (to fit a 4 GB GPU), 90% sparsity, block 64, batch 2048, SGD with the hyperparameters of Table 22, 60 epochs, three seeds, on the RTX A500 (Table 11).

Table 11: Adding a group-lasso penalty on the live blocks. Best test accuracy (mean ± standard deviation over three seeds), training step rate, and mean Frobenius norm of the live blocks at the end of training. Generated by `experiments/run_block_reg.py`.

λ	test accuracy (%)	steps/s	live-block norm
0 (BlockDST)	$\mathbf{59.78 \pm 0.26}$	**41.3**	1.076
10^{-5}	59.76 ± 0.26	39.1	1.072
10^{-4}	59.60 ± 0.31	39.1	1.030
10^{-3}	58.04 ± 0.12	39.1	0.652
10^{-2}	48.27 ± 0.48	39.0	0.031

The penalty does not help. Up to $\lambda = 10^{-4}$ accuracy is unchanged within one standard deviation; at 10^{-3} it drops by 1.7 percentage points and at 10^{-2} by 11.5, as the penalty shrinks the live blocks toward zero. Computing it also lowers the step rate by about 5%. The reason is that the invariant already enforces the block structure exactly, which is what group lasso is meant to induce, and the rewiring step already removes the weakest blocks by norm. The penalty adds nothing to either and acts only as extra shrinkage on the weights the model keeps. We therefore train without it.

E SMALL-SCALE ABLATIONS

These runs use the RTX A500 and the MLP of Table 3 at smaller widths, CIFAR-10, 90% sparsity, SGD with learning rate 0.05, batch 256, 40 epochs, `bf16` autocast and three seeds, matched in epochs. Every arm, BlockDST included, runs through the same masked implementation here, so these tables measure accuracy only, and each arm is pinned to the density BlockDST reaches with $b = 64$.

Table 12: Best test accuracy (%, mean ± sd over three seeds) by width, epoch-matched at batch 256 and 90% sparsity, with rewiring every ΔT steps. BlockDST uses $b = 64$ unless marked.

width	dense	BlockDST, ΔT=100	BlockDST, 1000	BlockDST b=32	SET, 100	SET, 1000	RigL, 100	RigL, 1000
1024	63.27 ± 0.17	58.03 ± 0.76	58.47 ± 1.00	58.70 ± 0.20	60.63 ± 0.22	59.61 ± 0.05	58.48 ± 0.18	61.27 ± 0.32
2048	64.49 ± 0.03	60.23 ± 0.12	60.47 ± 0.34	61.06 ± 0.06	62.19 ± 0.45	61.33 ± 0.03	60.49 ± 0.18	63.12 ± 0.14
4096	64.98 ± 0.26	62.60 ± 0.36	62.69 ± 0.38	62.62 ± 0.20	63.18 ± 0.20	61.76 ± 0.06	61.24 ± 0.28	64.37 ± 0.54
8192	65.38 ± 0.17	64.57 ± 0.26	64.37 ± 0.27	63.57 ± 0.24	63.99 ± 0.08	62.25 ± 0.24	62.69 ± 0.15	65.24 ± 0.19

Table 12 changes the reading of Section 5. Per epoch at batch 256, unstructured masks are at least as accurate as BlockDST: RigL with $\Delta T = 1000$ is ahead at every width, by 2.8 pp at 1024 and 0.7 pp at 8192, and SET is ahead up to width 4096. The gap narrows as the hidden layer's k grows (2 at width 1024, 13 at 8192), which is the regime the method is designed for. BlockDST is insensitive to ΔT (≤ 0.4 pp), RigL gains 2.6–3.1 pp from the longer interval, and SET loses up to 1.7 pp. Block size 32 against 64 changes BlockDST by at most 1.0 pp in either direction. An Erdős–Rényi–kernel layer-wise density, which RigL uses, changes neither SET nor RigL by more than 0.4 pp at widths 1024 and 4096. The 7.9–13.5 pp margins of Section 5 therefore belong to the large-batch, width-16,384 configuration and its shared learning rate, not to the patterns; Table 13 tests the two explanations.

Table 13: Further ablations on the RTX A500. **Top:** accuracy at width 4096, 90% sparsity, the setting of Table 12 ($\Delta T = 100$, three seeds); "no invariant" keeps BlockDST's blocks, budget and criteria but lets block-rows gain or lose blocks. **Bottom:** time-to-target protocol of Table 2 at batch 8192 and 95% sparsity (`bf16` storage for BlockDST), seconds and steps to 0.55 accuracy; three seeds at learning rate 0.05, one seed otherwise.

accuracy (%), width 4096, 90% sparsity			
BlockDST, $b = 64$ (constant k)	density 9.8%	62.60 ± 0.36	
no invariant, $b = 64$	density 9.8%	62.24 ± 0.56	
BlockDST, $b = 16$	density 10.0%	62.41 ± 0.16	
BlockDST, $b = 32$	density 9.8%	62.62 ± 0.20	
BlockDST, $b = 128$	density 10.7%	62.67 ± 0.23	
arm	learning rate	seconds to 0.55	steps to 0.55
width 8192			
BlockDST	0.05	124.9 ± 1.9	793
	0.1 / 0.2	63.1 / 52.8	399 / 335
dense	0.05	83.4 ± 15.3	256
	0.1 / 0.2	45.2 / 44.2	139 / 139
SET	0.05 / 0.1	604.3 / 333.8	1839 / 1015
	0.2 / 0.4	177.8 / 128.2	540 / 388
dense, width 1119 (same live parameters)	0.05	$\mathbf{12.6 \pm 1.1}$	309
width 16,384			
BlockDST	0.05	292.9 ± 31.6	613
dense, width 2734 (same live parameters)	0.05	$\mathbf{21.7 \pm 4.2}$	276

The invariant costs no measurable accuracy: removing it changes BlockDST by -0.36 pp, within one standard deviation, and block sizes from 16 to 128 agree to within 0.3 pp. The shared learning rate of 0.05 is too low for every arm at batch 8192, and it penalizes SET most: from 0.05 to 0.4 SET needs $4.7\times$ fewer steps, BlockDST $2.4\times$ fewer from 0.05 to 0.2, so at each arm's best tested rate SET needs only $1.16\times$ more steps than BlockDST, against $2.3\times$ at the shared rate. The large-batch margins over the masked baselines in Section 5 are therefore partly a tuning effect, although BlockDST remains $2.4\times$ faster than tuned SET in time through its step rate. A dense model with the same number of live

parameters is the strongest arm by far: it reaches the target 10–14× sooner than BlockDST at both widths, in a similar number of steps, and with less memory (about 900 against 1540 MB at width 8192). On this card BlockDST is also slower than the wide dense model, consistent with the A500's break-even near 90%. Time-to-target on the A500 is not comparable in absolute terms with the L40 of Table 2.

Does rewiring help? Table 14 separates the kernel from the rewiring. The static arm runs the same kernels on the same constant-k random block layout but never rewires ($\Delta T = 0$); the dynamic arm is BlockDST as in the rest of the paper, rewiring at step 0 and every 1000 steps. We added a higher target, 0.60, which BlockDST reaches just after its rewirings at steps 1000 and 2000. The static layout is faster at every target: 1.19× to 0.55 and 1.36× to 0.60, with seed ranges that do not overlap, at the same step rate. The step-0 rewiring roughly doubles the steps to 0.45 (134 against 69), which suggests the cause: it replaces 30% of each row's initialized blocks with zero blocks. In these runs, therefore, the time-to-target gains come from executing a fixed block layout, and rewiring costs time rather than saving it. To separate the step-0 rewiring from rewiring as such, we delayed the first rewiring to step 1000 (`--rewire-start 1000`). Another job started on the same GPU during this run, so we use only seed 0, whose step-rate calibration (which sizes the rewiring schedule) preceded it, and compare steps, which do not depend on GPU speed. Until step 1000 this arm is the static layout, and it reaches 0.55 in 677 steps (static 670–675). The rewiring at step 1000 then costs time: it needs 1920 steps to 0.60, 1.29× the static layout's 1482–1485 and close to the 1986–2027 of the arm that also rewires at step 0. With one seed this is an observation rather than a result, but the static and dynamic arms vary by under 3% across seeds, and it indicates that rewiring itself, not only the zeroed initial blocks, slows convergence over this horizon (`figures/review_v66.py`). We have not tested whether rewiring pays in much longer training, as it does for unstructured DST, or with a gradient growth criterion on the kernel path. The dense arms repeat Table 13 with the new target and seeds.

Table 14: Static against dynamic block layout on the RTX A500: seconds to first reach each target (mean ± sample sd, three seeds). Width 8192, batch 8192, 95% sparsity, learning rate 0.05, `bf16` storage; every arm reached every target on every seed. Generated by `figures/review_static.py`.

arm	steps/s	$t\to0.45$	$t\to0.50$	$t\to0.55$	$t\to0.60$
BlockDST, rewired ($\Delta T = 1000$)	6.33	21.1 ± 0.2	48.9 ± 5.8	125.1 ± 1.1	316.6 ± 2.8
same layout, never rewired	6.39	10.8 ± 0.0	39.4 ± 4.7	105.3 ± 0.4	232.4 ± 1.0
dense, width 8192	3.06	10.7 ± 3.2	30.7 ± 4.5	84.4 ± 15.1	218.1 ± 18.3
dense, width 1119 (same live parameters)	24.67	$\mathbf{2.2 \pm 0.5}$	$\mathbf{5.0 \pm 0.7}$	$\mathbf{12.5 \pm 1.2}$	$\mathbf{29.2 \pm 2.5}$

F Reporting recommendation

Because the constant can be predicted once the right resource is known, we suggest reporting the three quantities that let a reader compute a result on other hardware:

1. The accelerator's SM count and boost clock, and D, the dense baseline's measured throughput on the shapes used. With these, $c = k_0(\mathrm{SM}\cdot f)/D$ predicts the constant on a held-out card of ours to within 0.91–1.19× at the reference shape and 0.78–1.66× over all shapes (Table 6). D is not on a datasheet but costs a single GEMM.
2. The fitted exponent and the densities it was fitted on: ours is 0.84 on average rather than the 1 a roofline argument suggests, and on the L40 the fitted 0.88 moves the extrapolated break-even by 3.6 points relative to $p = 1$. It should be fitted over at least three densities, and break-even should be anchored at a measured speedup (Equation (1)).
3. The density range and the shape: break-even is often an extrapolation (in our data on three of six cards, by up to 2.9× beyond the densest point measured), and it depends on shape through D (Appendix B.8), so both should be stated.

The attained fraction of the bandwidth roofline, η, is not a useful number to report. It is the reciprocal of the bandwidth model's error, so it describes that model rather than the kernel.

Halving the weight traffic with bf16 storage speeds the kernel up by only 1.12–$1.25\times$ rather than $2\times$, as expected for a kernel that is not bandwidth-bound (Appendix G.1).

G KERNEL BENCHMARKS

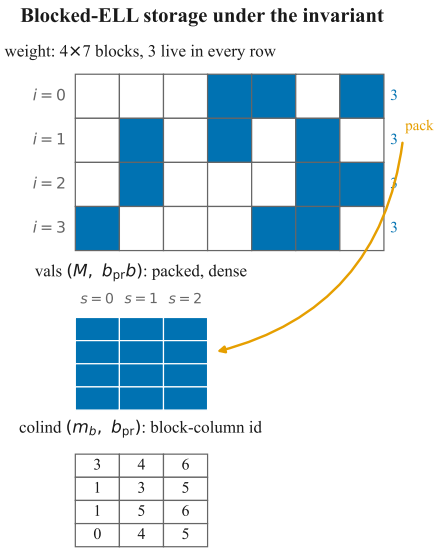

Figure 8: Blocked-ELL storage under the invariant. Live blocks sit at arbitrary block-columns, but every block-row holds exactly k of them, so the values form a dense (M, kb) array and the only index is one (m_b, k) array of block-column ids, with no row pointers or per-row lengths.

All comparisons run in one process, on one GPU, over the same live blocks; Figure 8 shows the layout our kernels read. Compared with BLaST's BSpMM, our kernel is 1.4–$2.4\times$ faster at 98% sparsity, similar at 95%, and slower at 90% on the largest shape. Compared with PyTorch's Triton BSR kernel, it is 1.6–$2.7\times$ faster at 98%, 1.9–$3.0\times$ at 95% and 1.9–$3.2\times$ at 90%, over 16 shape and block-size combinations reproduced in three independent jobs. In both comparisons our kernel reads `fp32` while theirs reads `bf16`. The Triton path is not autotuned for these shapes, BLaST's kernel targets a blocked-CSC layout and a multi-GPU large language model (LLM) workload, and neither comparison was run on the A100.

G.1 NUMERICAL PRECISION

What each arm computes in. The dense and masked arms train with bfloat16 (bf16) autocast, that is, bf16 tensor-core products with 32-bit floating-point (fp32) master weights and accumulation (Micikevicius et al., 2018; Kalamkar et al., 2019). Our kernel stores the block values in fp32 and multiplies on tensor cores with bf16 inputs and fp32 accumulation; the "BlockDST+bf16" arm stores the block values in bf16 as well.

Why it matters for the comparison. Precision changes the dense baseline far more than it changes accuracy. A bf16 GEMM runs several times faster than an fp32 one on tensor cores, so a sparse kernel compared against an fp32 dense baseline, or a half-precision sparse kernel against an fp32 dense one, looks faster by that factor for reasons unrelated to sparsity. Every speedup in this paper is therefore measured against a dense GEMM of the same or higher arithmetic precision: our kernel against bf16 cuBLAS, and in Table 10 each cuSPARSE format against cuBLAS in its own precision.

Effect on accuracy. Storing the block values in bf16 changes results only by rounding: the two paths agree to 3×10^{-2} relative error on y, dx and dw (`tests/test_wmma_path.py`). Over a training run, rounding shifts the step at which a target is crossed by about 9% (for example 485 and

445 steps on the A40 at 90%), but the time to 0.55 accuracy is within seed variation: 29.4 against 29.1 s at 95% sparsity and 37.7 against 43.9 s at 90% (Table 2).

Effect on wall-clock time and memory. bf16 storage makes the kernel 1.12–1.25× faster across four GPUs, rather than the 2× that halving the weight traffic would give a bandwidth-bound kernel, which is consistent with per-SM scaling (Section 4.1). In training on the L40 it raises the step rate by 1.14× at 90% sparsity (11.4 to 13.0 steps/s) and not at all at 95% (21.7 and 21.4 steps/s), where the sparse products are a smaller share of the step. It does not lower peak memory (2793 against 2459 MB at 95%), so its benefit is throughput at moderate sparsity only.

G.2 Is the kernel itself competitive?

Against the two released implementations we could run on the same GPU and the same blocks, our kernel is 1.4–2.4× faster than BLaST at 98% sparsity and 1.9–3.0× faster than the Triton BSR path at 95%, but slower than BLaST at 90% on the largest shape. Against cuSPARSE's Blocked-ELL SpMM at the shapes of Table 10 on the RTX A500 (weight 4096×4096, batch 512 and 2048, $b = 64$, 90–98% sparsity), our forward kernel is slower: 2.9–3.3× with `bf16` block storage against cuSPARSE's fp16, and 2.1–2.4× with fp32 storage (`bench/bench_vs_cusparse_ell.py`). Our kernels use `wmma` without asynchronous copies (Appendix B.3), and this gap is the main reason to read the speedup constants as properties of our kernel: a forward product as fast as cuSPARSE's would raise c by up to 3× and lower every break-even. cuSPARSE cannot replace the whole kernel set, because it provides no block-SDDMM for the weight gradient, but it could serve the forward product in the same layout.

H From kernel speedup to training speedup

At 95% sparsity on the L40 our kernel is 5.13× faster than a dense `bf16` GEMM, while training on the same card is 1.44× faster. Figure 9 splits the gap into two measured factors:

$$\underbrace{\frac{t_{\text{dense}}}{t_{\text{ours}}}}_{\text{end-to-end}} = \underbrace{S_{\text{kernel}}}_{\text{GEMM}} \times \underbrace{\frac{S_{\text{step}}}{S_{\text{kernel}}}}_{\text{Amdahl}} \times \underbrace{\frac{1}{R_{\text{steps}}}}_{\text{step efficiency}}, \tag{5}$$

where S_{step} is the ratio of training step rates and R_{steps} the factor by which the sparse model needs more steps to reach the target (Table 15).

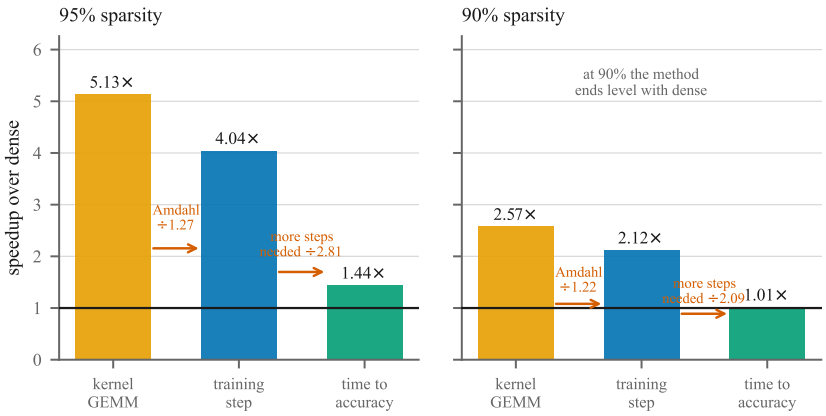

Figure 9: How a kernel speedup shrinks into a training speedup. Same L40 and runs as Table 2. Two multiplicative losses separate the kernel figure from the end-to-end result, and at 90% sparsity they remove the advantage.

Inverting $S_{\text{step}} = 1/[(1-f) + f/S_{\text{kernel}}]$ gives $f = 93\%$ at 95% sparsity: nearly all of a training step of a width-16,384 three-layer MLP is sparsifiable GEMM, close to the best case. In a transformer,

Table 15: Decomposition of the end-to-end speedup on the L40 (Equation (5)), time to 55% accuracy, three seeds.

sparsity	kernel	step rate	Amdahl	steps to target	end-to-end
95%	5.13×	4.04×	÷1.27	×2.81	**1.44**×
90%	2.57×	2.12×	÷1.22	×2.09	1.01×

attention, normalization and the optimizer are not sparsified, f is lower, and this factor alone can remove a large kernel speedup, which is consistent with our transformer results (Section 5.5).

At 95% sparsity the model needs 2.81× more steps to reach 55% accuracy. This is the larger of the two losses, and no kernel improvement changes it, because it is what sparsity costs the optimizer. A throughput benchmark cannot see it, which is why we measure time to a fixed accuracy.

H.1 Transfer to a second accelerator

If the decomposition is meaningful, the Amdahl and step-efficiency factors describe the model and the method, and should carry over to another card, leaving the kernel speedup as the only hardware-dependent term. We fixed this prediction before running any training on a card other than the L40 (`figures/predict_crosscard_t2t.py`) and tested it on an A40 (Tables 16 and 17).

Table 16: Time-to-target on a second accelerator. Seconds to 55% test accuracy on an A40 (one seed), same protocol as Table 2. End-to-end is dense time divided by BlockDST time, so values below 1 mean BlockDST is slower.

sparsity	BlockDST	dense	end-to-end
95%	56 s	39 s	0.70×
90%	79 s	38 s	0.48×

Table 17: Predicted and measured factors on the A40, with the L40 factors as the prediction (seed 0).

	95% sparsity		90% sparsity		
factor	predicted	measured	predicted	measured	difference
steps to target	2.81×	2.76×	2.09×	2.15×	−2%, +3%
kernel ÷ Amdahl	1.51×	1.91×	0.75×	1.03×	+26%, +37%
end-to-end	0.54×	0.70×	0.36×	0.48×	+30%, +34%

The step-efficiency factor transfers to within 3%, so the optimizer cost of sparsity is a property of the model. The predicted direction of the end-to-end result, a loss on the A40, also holds. The kernel factor does not transfer. Inverting the measured step rate gives kernel speedups of 2.41× and 1.26× at the *training* shape, against 1.91× and 0.92× at 8192^3. Since the dense baseline varies 2.0× across shapes on one card (Appendix B.8), the kernel factor must be measured at the shape the model trains at.

The dense arm is run once per density block but does not depend on density, so these runs are replicates of the same configuration and seed. They are not identical: on the L40, seed 0 needs 187 steps to reach 0.55 in one block and 230 in the other, so nondeterministic GPU arithmetic, not the seed, sets most of the spread. Across the six L40 dense runs the number of steps to the target varies by 13% (coefficient of variation) and the step rate by 2%, so the dense arm's spread in time-to-target is optimization variance rather than a timing artifact; on the A40 the replicates agree to within 2%. The step-rate and step-count factors are therefore more precise than the end-to-end ratio, and `figures/check_crosscard_t2t.py` scores them separately.

I ACCURACY AGAINST TRAINING FLOPS

Figure 10 plots test accuracy against cumulative training floating-point operations (FLOPs) for the runs of Table 2. For the MLP, one training step costs $6B\,[\,d\,(3072\,W + W^2) + 10\,W\,]$ FLOPs for batch B, width W and density d: two FLOPs per multiply-accumulate, and forward, input-gradient and weight-gradient products of equal size. We count only live weights for every sparse method, as the DST literature does. BlockDST's rewiring grows blocks at random (Appendix M), so it computes no gradient at dead blocks and adds only a per-block norm and a sort, negligible against a training step.

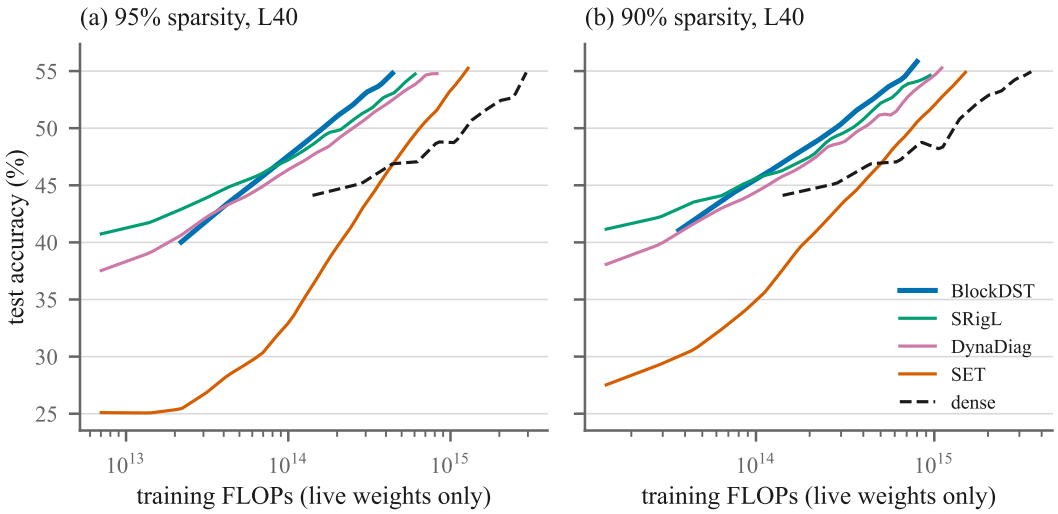

Figure 10: Test accuracy against training FLOPs (live weights only), L40, width 16,384, batch 8192, mean of three seeds. Dense is dashed. The masked baselines execute dense FLOPs in practice, so this axis describes their arithmetic cost only in principle.

Table 18: Training FLOPs (live weights only) to first reach 0.55 test accuracy, mean of three seeds, and the reduction relative to dense. Generated by `figures/make_fig_flops.py`.

	95% sparsity		90% sparsity	
arm	FLOPs	vs dense	FLOPs	vs dense
BlockDST	$\mathbf{4.9 \times 10^{14}}$	**7.1**× fewer	$\mathbf{8.0 \times 10^{14}}$	**4.7**× fewer
SRigL	6.8×10^{14}	5.2× fewer	1.1×10^{15}	3.4× fewer
DynaDiag	9.1×10^{14}	3.9× fewer	1.2×10^{15}	3.3× fewer
SET	1.3×10^{15}	2.7× fewer	1.7×10^{15}	2.3× fewer
dense	3.5×10^{15}	—	3.8×10^{15}	—

Per unit of arithmetic, BlockDST reaches the target with the fewest FLOPs of any method at both sparsities, so its accuracy per FLOP is at least as good as that of the finer-grained patterns (Table 18). The FLOP counts of all sparse methods are 2–7× below dense, yet only BlockDST turns this into less time: the masked baselines execute the dense product and take 2.7–7.0× longer than dense (Table 2), and BlockDST saves 1.44× in time against 7.1× in FLOPs, the gap explained in Appendix H. FLOPs therefore rank methods by algorithmic efficiency but not by training time.

J STATISTICAL TREATMENT

We report margins with Welch's t (Welch, 1947) (Satterthwaite degrees of freedom (Satterthwaite, 1946)) and confidence intervals on ratios computed on log time (Table 19). The Holm correction (Holm, 1979) accounts for the 30 pairwise comparisons in the equal wall-clock table.

Table 19: Main comparisons with confidence intervals and p-values.

comparison	effect	95% CI	t (df)	p	p (Holm)
BlockDST vs dense, $t\to0.55$, 95%	1.44×	[0.99, 2.07]	—	0.050	—
interpolated crossing, 95%	1.45×	[1.08, 1.95]	(2.8)	0.029	—
six pooled dense runs, 95%	1.48×	[1.21, 1.81]	(5.5)	0.004	—
BlockDST vs dense, $t\to0.55$, 90%	1.01×	[0.76, 1.35]	—	0.93	—
six pooled dense runs, 90%	0.98×	[0.85, 1.13]	(6.2)	0.70	—
BlockDST vs SRigL, $t\to0.55$, 95%	5.22×	[4.0, 6.9]	—	< 0.01	—
BlockDST vs dense, eq. wall-clock, 90%	+1.22 pp	[0.53, 1.91]	7.13 (3.2)	0.005	0.14
BlockDST vs dense, eq. wall-clock, 95%	+0.53 pp	[0.06, 1.00]	3.09 (3.8)	0.037	1.00

The $1.44\times$ time-to-target result cannot be distinguished from $1.0\times$ with three seeds ($p = 0.050$). Most of the variance is in the dense arm, whose three seeds take 34.7, 41.6 and 49.2 s. Two further analyses tighten it. Both were chosen after we had seen the pre-specified three-seed result, so they are post hoc and exploratory, and we do not use them to claim significance. The recorded crossing is the first evaluation above the target on a grid with 15% resolution; interpolating between the bracketing evaluations, which each file stores, gives $1.45\times$ ($[1.08, 1.95]$). The dense arm does not depend on the density setting, so its three runs in the 90% block replicate the configuration of the three in the 95% block; pooling all six gives $1.48\times$ ($[1.21, 1.81]$, $p = 0.004$). The pooled runs reuse seeds 0–2, so they are replicates of configuration rather than of seed, but a repeated seed does not repeat the run (seed 0 needs 187 and 230 steps; Appendix H.1). At 90% sparsity every version is a tie. The comparisons with the masked baselines have large margins and small variances.

The pooled within-arm standard deviation of accuracy is 0.187 pp, giving a minimum detectable effect of 0.57 pp at 80% power with n=3 per arm under the t distribution (0.43 pp under a normal approximation). Smaller margins, including +0.53 pp at 95% sparsity, are reported as ties. More seeds on the L40 (about 1 min per run) and on the A40 would settle the dense comparison directly.

The 54 cost-model measurements are one run per (card, shape, density), so the bracketed ranges in Appendix B are spreads across shapes rather than confidence intervals (Appendix B.4 repeats the RTX A500 sweep to measure run-to-run spread), and the peak-memory table uses one seed per cell.

Selection on the test set. CIFAR-10 has no official validation split, and we did not hold one out: the accuracy targets, the ViT learning rates and "best-epoch" accuracies are all read from the test set. The targets were fixed in the job script before any time-to-target run, as values below every arm's earlier equal wall-clock ceiling, and all three are reported; the learning-rate choices, however, favor arms with more tuning cells, which in Table 25 are the sparse arms.

K Peak memory

Memory is measured with `torch.cuda.max_memory_allocated`. A reading taken just before the model is built is subtracted so that the shared dataset is charged to no arm; this floor is 721.7 MB in 11 of the 12 files and 704.5 MB in the twelfth. Without the subtraction the BlockDST/dense ratio would be $1.85\times$ instead of $2.10\times$. The peak includes weights, gradients, the SGD momentum buffers and activations; BlockDST's rewiring forms no dead-block gradient. These memory runs use a 60 s cap; one SET run among them did not reach its target (0.4249 against 0.45) and is recorded as censored. The `bf16` storage variant uses *more* memory than the default (2793 against 2459 MB at 95%) because it keeps the `fp32` master block values and adds per-step `bf16` copies of the input, the block values and the output gradient: it reduces memory traffic in the kernels, not the memory footprint.

The baselines are our implementations of the patterns, each holding a dense weight, a mask, a gradient and the masked product, as most released DST code does. The comparison is therefore about this standard implementation, not about SET, SRigL or DynaDiag as methods: MEST (Yuan et al., 2021) and SMET (Xiao et al., 2026) reduce DST memory, and truly sparse SET implementations avoid the dense weight. Dense memory is the same at both densities, as it should be.

Table 20: Peak device memory during training. One L40, width 16,384, batch 8192, one seed per cell; the last column is at 95% sparsity.

arm	95% sparsity	90% sparsity	vs dense at 95%
BlockDST	**2459 MB**	**2587 MB**	**2.10× less**
BlockDST+bf16	2793 MB	2970 MB	1.85× less
dense	5156 MB	5156 MB	—
SRigL	7345 MB	7378 MB	**1.42× more**
SET	9055 MB	9120 MB	**1.76× more**
DynaDiag	11412 MB	11412 MB	**2.21× more**

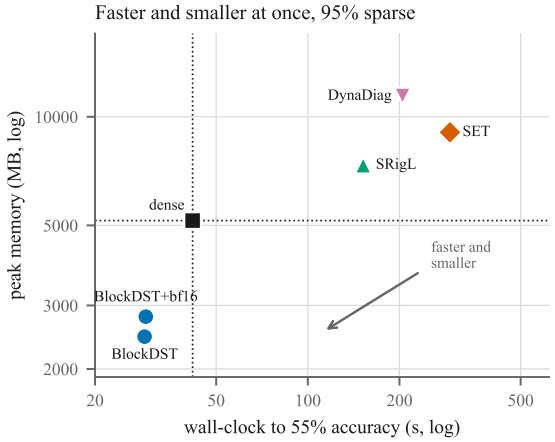

Figure 11: Time and memory together: of the implementations compared, only BlockDST is faster and smaller than dense at once; the masked baselines use 1.4–2.2× more memory than dense because each stores a dense weight, a mask, a gradient and the masked product. L40, width 16,384, batch 8192, 95% sparsity; times are the means of Table 2, memory is from the separate 60 s runs.

L Equal wall-clock results

Table 21: Accuracy after an equal wall-clock budget (width 16,384, 95% sparsity, 600 s, 3 seeds, one job on one L40). SRigL's constant fan-in already balances rows, so its margin isolates the effect of block granularity.

method	accuracy	steps/s	gap
BlockDST (ours)	**0.6944** ± 0.0020	**21.4**	—
dense	0.6891 ± 0.0022	5.6	+0.53 pp (tie)
SRigL	0.5942 ± 0.0011	5.9	−10.02 pp
DynaDiag	0.5850 ± 0.0026	5.9	−10.94 pp
SET	0.5595 ± 0.0008	5.8	−13.49 pp

At the same density an unstructured mask is a superset of our constant-blocks-per-row mask, and the literature treats unstructured DST as an upper bound on accuracy for structured methods, yet SET is 13.5 pp below dense here. Likely causes are layer collapse from global magnitude pruning at 95% on a 16,384-wide layer, or a shared learning rate that suits the block pattern better than the unstructured one. Appendix E supports the second: at batch 8192 and width 8192, raising SET's learning rate from 0.05 to 0.4 cuts its steps to 0.55 by 4.7×, far more than any other arm gains. Comparisons with SET should therefore not be read as evidence about the sparsity pattern.

These files predate device-name logging, and one scheduler label maps to two different GPUs (Appendix N). We verify that each batch ran on one card with the dense arm: dense does not depend on the density setting, so all its runs in a batch must share one step rate. In the released set they do (5.55–5.61 step/s, a 1.01× spread). Batches whose dense runs split into two step rates are excluded;

one such batch, whose 95% and 98% blocks ran about 1.7× faster than its 90% block, was excluded on this test (`verification/verify_eqwc_quarantine.py`).

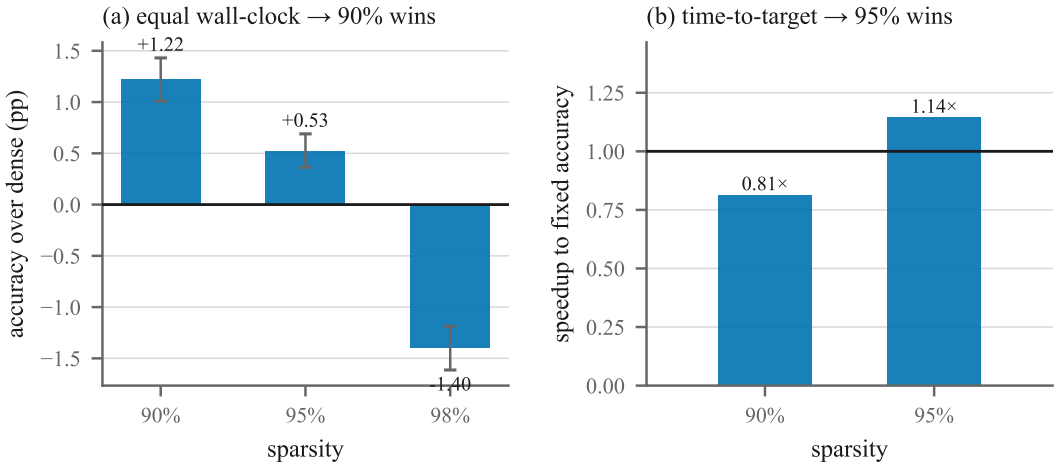

Figure 12: The two protocols favor different sparsities. **(a)** With equal wall-clock budgets, 90% sparsity looks best. **(b)** With time-to-target, 95% looks best. The margins in (a) are not individually significant after correction (Appendix J); the claim is the direction of the disagreement.

Accuracy keeps improving well into the 600 s budget. BlockDST is at 0.548 after about 92 epochs and 0.694 after about 2100. The case for time-to-target therefore does not rest on over-training; it rests on the fact that a fixed clock cannot express a throughput advantage except as extra accuracy.

M HYPERPARAMETERS

Table 22: Training hyperparameters, shared by all arms.

setting	value
task	CIFAR-10, 50,000 train / 10,000 test
model	3-layer MLP, width 16,384 (1024–8192 in the ablations of Appendix E)
batch	8192
optimizer	SGD, momentum 0.9, constant learning rate 0.05
block size b	64
live blocks per row k	input layer 2, hidden layer 13 at 95% (overall 95.07% sparse); 5 and 26 at 90%; 1 and 5 at 98%
rewiring interval ΔT	1000 steps (100 for the language models)
rewiring fraction	0.3, cosine-decayed to zero at 75% of the steps the calibrated step rate gives for the time cap
drop criterion	Frobenius norm of the live block
growth criterion	uniform at random among the dead blocks of that block-row
grown blocks	initialized to zero, momentum reset
precision	`bf16` autocast for all arms; kernel storage `fp32` unless stated
seeds	0, 1, 2
time-to-target	targets 0.45, 0.50, 0.55; cap 600 s (60 s for the memory runs)
evaluation grid	geometric, $t_0 = 0.3$ s, growth 1.15

One training step. (1) Forward: $Y = W_{\text{blk}} X$ with the blocked-ELL kernel, k blocks per block-row. (2) Input gradient: $\nabla X = W_{\text{blk}}^{\top} \nabla Y$ through the block-column index. (3) Weight gradient: block-SDDMM, ∇W at live blocks only. (4) Optimizer step on the (N, kb) value array. (5) If the step is a rewiring step: per block-row, drop the $n = \lfloor f(t)\, k \rceil$ live blocks with the smallest Frobenius norm, grow n dead blocks chosen uniformly at random, zero the grown values and their momentum, and rebuild the block-column index. $f(t)$ is the cosine-decayed rewiring fraction.

Growth criterion. The kernel implementation (`dst/blockdst_fast.py`), which produces every timed result in the paper, grows at random, so it needs no gradient at dead blocks and forms nothing dense at any step. Our masked reference implementation (`dst/methods.py`) also supports growth by the largest dead-block gradient norm, as RigL does for single weights; the accuracy-only width and interval ablations of Appendix E use that criterion. A gradient criterion on the kernel path would need ∇W at dead blocks, that is, one dense $\nabla Y X^\top$ per rewiring; computed one block-row panel at a time it needs no dense buffer, and at $\Delta T = 1000$ it would cost about one dense weight-gradient product per thousand steps, well under 1% of training time on the L40, but we have not implemented it. The two implementations therefore differ in growth rule as well as in speed, and Table 14 shows that for the time-to-target runs the rewiring itself does not help. Rewiring happens at step 0 and then every ΔT steps. Because the runs are capped in time rather than steps, the schedule's length is the number of steps the 20 calibration steps predict for the cap (600 s in Table 2); every arm crosses its targets well before 75% of it. BlockDST reaches 0.55 accuracy in about 590 steps at 95% sparsity and 500 at 90%, so in the time-to-target runs it rewires once, at initialization, and then trains a fixed block layout; the 600 s equal wall-clock runs (about 20,000 steps) rewire 16 times. The update interval matters little for BlockDST: $\Delta T = 100$ and 1000 give accuracies within 0.2 pp at width 4096–8192 and 0.4 pp at 1024 (Appendix E). The baselines use the same schedule, density and optimizer and differ only in the mask pattern, without per-arm learning-rate tuning; Appendix E checks how much that costs SET.

N Measurement protocol

On our cluster one generic scheduler resource label resolves to either an L40 (300 W) or an L40S (350 W), which differ about 2× in `bf16` throughput and cannot be told apart from the resource or feature strings. All arms of a comparison therefore run sequentially in one job on one GPU, and the time-to-target runs record `torch.cuda.get_device_name`.

The dense arm's throughput does not depend on sparsity, so variation between its runs measures noise and exposes runs on a different card. It differs by 3–4% between Table 2 (5.4 steps/s) and Table 21 (5.6 steps/s), which were run at different times under different protocols.

Recorded times are the first evaluation after a target is crossed, on a geometric time grid (start 0.3 s, growth 1.15); using the interpolated crossing that each file also stores changes the ratios to dense, taken as the ratio of mean times, from 1.44 to 1.46 at 95% sparsity and from 1.01 to 1.04 at 90% (Table 19 reports the ratio of geometric-mean times, 1.45× at 95%, with its interval). Ratios against SET grow with the target (22.8×, 14.7× and 10.1× at 0.45, 0.50 and 0.55) because SET converges to 0.5595 ± 0.0008 (Appendix L), just above the 0.55 target, where the time needed grows quickly; the comparison with SRigL, whose ceiling of 0.5942 is well above the target, is the more reliable one.

Before the clock starts, each arm runs 5 warm-up steps and 20 calibration steps, which absorb cuBLAS heuristic selection, allocator growth and first-call compilation; the clock is paused for every evaluation. Runs whose first evaluation already exceeds the target are rejected, since their crossing time would be set by the evaluation grid. Runs that do not reach a target within the cap are recorded as censored rather than dropped; none of the 600 s runs is.

The runners skip completed output files but refuse to resume if the set of recorded fields has changed, so results from different runs are never mixed.

NVML ignores `CUDA_VISIBLE_DEVICES`, so the energy probe maps devices explicitly. A preflight step checks the kernel exports before each job, so a failed build cannot load an outdated kernel.

O Comparison with related methods

O.1 Kernel-accelerated baselines

The masked baselines of Section 5 (SET, SRigL and DynaDiag) have no sparse kernel in their standard implementations: SRigL reports speedups for inference only, and DynaDiag, which reports a 1.59× training speedup, has not released its kernels. They therefore measure what each pattern costs in accuracy, not what a kernel-based competitor achieves in time. We compare against accelerated sparse kernels in three places, all on the same GPU and the same matrices: cuSPARSE's CSR, BSR

and Blocked-ELL kernels (Appendix C), and BLaST's blocked-CSC kernel and PyTorch's Triton block-sparse kernel (Appendix G). The sparsity that NVIDIA hardware accelerates directly is 2:4 (Mishra et al., 2021), and 2:4 pre-training methods use it to speed up training at a fixed 50% sparsity (Hu et al., 2024a;b). Because 2:4 halves the number of multiply-accumulates, its speedup on the sparsified products is bounded by $2\times$, whereas our kernel reaches 1.08–$5.13\times$ at 95% sparsity and 1.99–$9.37\times$ at 98% (Figure 4(c)); on the other hand, 2:4 loses little accuracy at 50%, while our method targets the 90–98% range. Our own 2:4 language-model arm uses a mask and measures accuracy only. At the kernel level, PyTorch's cuSPARSELt path for 2:4 runs the forward product at $0.55\times$ and $0.97\times$ the speed of dense `bf16` on the RTX PRO 6000 (batch 8192, weights 4096^2 and 8192^2; one run, `data/kernel/kcmp24_rtxpro6000_559793.txt`), so on that card and software stack the one accelerated sparse-training path that ships with PyTorch does not speed up the forward product either. A full training run with accelerated 2:4 kernels on the same model and GPU is left for future work.

O.2 Other related work

Pruning at initialization (SNIP (Lee et al., 2019), GraSP (Wang et al., 2020), SynFlow (Tanaka et al., 2020)) fixes a sparse mask before training; BlockDST in our time-to-target runs is close to this regime, since it rewires once at initialization and then trains a fixed block layout (Appendix M), but with a mask that the kernel can execute. AC/DC (Peste et al., 2021) alternates dense and sparse phases and reduces inference cost rather than training time. Truly sparse SET (Curci et al., 2021) and SparseProp (Nikdan et al., 2023) avoid the dense weight on CPUs through unstructured sparse kernels, the memory saving our masked baselines forgo. Block-structured training on GPUs also includes Pixelated Butterfly and Monarch (Chen et al., 2022; Dao et al., 2022), which fix a structured product form rather than rewire, fine-grained block structures (Jiang et al., 2022), channel-level DST (Yin et al., 2023), and transposable N:M masks that make the backward pass sparse on 2:4 hardware (Hubara et al., 2021); none keeps a constant block count per row for a blocked-ELL kernel with a sparse weight gradient.

O.3 Reported speedups

Table 23 lists headline speedups reported by recent structured-sparsity work. The entries differ in accelerator, shape, baseline and training phase, so the table is not a ranking. Most are kernel or inference measurements. On our best card the gap between kernel and end-to-end speedup is $3.6\times$ at 95% sparsity and $2.5\times$ at 90%, where a $2.57\times$ kernel becomes a tie, so kernel speedup, step-rate speedup and time to a fixed accuracy are different quantities, and we report all three. Our cross-card rows are forward kernel measurements, with end-to-end rows given separately.

We do not compare directly with HASTE, VENOM, Samoyeds or MegaBlocks, because each addresses a different setting. HASTE is a sparse training method for the output layer of extreme multi-label classifiers, a layer type and task our benchmarks do not contain. VENOM and Samoyeds are sparse tensor-core kernels for N:M-style patterns, which fix the number of nonzeros per small group of weights rather than keeping a block mask that is rewired during training, and they provide no weight-gradient product for our layout. MegaBlocks makes mixture-of-experts routing block-sparse and does not sparsify the weights of a dense layer. Running any of them would therefore test a different method rather than a different kernel for ours. Their headline numbers also come from other accelerators, shapes and baselines, and the card alone changes our speedup by $5.4\times$ (Section 4), so Table 23 reports them for context only. Where a kernel accepts our layout, we compare on the same GPU and matrices: BLaST and PyTorch's Triton block-sparse kernel in Appendix G, and cuSPARSE in Appendix C.

P Provenance

Table 24 maps each result to the script and files that produce it.

The time-to-target files record the device name. The equal wall-clock, language-model and ViT files do not; their single-card provenance is checked with the dense-arm test of Appendix L (every language-model arm's step rate is constant across seeds to within 0.2%), and the ViT sweep is

Table 23: Headline speedups in recent structured-sparsity work, as reported by each paper, and ours for comparison.

work	reported speedup	sparsity	phase measured
SRigL (Lasby et al., 2024)[a]	1.7–13.0×	90%	inference, linear layer
DynaDiag (Tyagi et al., 2025a)[b]	3.13× / 1.59×	90%	inference / training
BLaST (Okanovic et al., 2025)[c]	2.2×	up to 95%	inference, 16 GPUs
HASTE (Ullah et al., 2026)[d]	4.4× / 25×	—	forward / backward
VENOM (Castro et al., 2023)[e]	up to 37×	N:M	kernel vs cuBLAS
Samoyeds (Wu et al., 2025)[f]	1.99× / 1.58×	N:M	kernel / model
MegaBlocks (Gale et al., 2023)[g]	up to 2.4×	mixture-of-experts (MoE) routing	training
BlockDST on L40	2.57×	90%	forward kernel
BlockDST on L40S	1.38×	90%	forward kernel
BlockDST on A100	0.48×	90%	forward kernel
BlockDST on L40, training step	2.12× / 4.04×	90% / 95%	training step
vs. BLaST BSpMM kernel	1.4–2.4×	98%	forward, same GPU
vs. PyTorch Triton BSR	1.9–3.0×	95%	forward, same GPU

Sources: [a]ICLR 2024, arXiv:2305.02299, 90% sparse linear layer, batch 256. [b]arXiv:2506.11449, 90% sparse ViT linear layers. [c]arXiv:2507.03117, Llama 3.2 on 16 GPUs. [d]arXiv:2606.01117, against fixed fan-in, extreme multi-label output layer. [e]SC 2023, arXiv:2310.02065, Spatha against cuBLAS. [f]arXiv:2503.10725, kernel / end-to-end model. [g]MLSys 2023, arXiv:2211.15841, dMoE against dense Megatron-LM.

matched in epochs, so its accuracies do not depend on the card. The width-32,768 runs, the L40S training runs and the BLaST and Triton comparisons are not part of the released files.

Reproducing the results. The following commands regenerate every figure and table that has a released source:

```
cd figures
python make_figures.py         # hardware and protocol figures
python make_figures_new.py     # SM-count and exponent figures
python fit_law.py              # speedup law, time-to-target table
python model_compare.py        # regression, model comparison
python fwd_vs_fwdbwd.py        # forward-only proxy check
python format_feasibility.py   # choice of sparse format
python check_crosscard_t2t.py # A40 results, repeatability
python ../verification/verify_eqwc_quarantine.py  # one card
python make_fig_flops.py       # accuracy against FLOPs
python review_stats.py         # break-even, intervals, models
python review_t2t.py           # pooled, interpolated t2t
python review_static.py        # static vs dynamic, 0.60 target
python review_v66.py           # delayed rewiring (one seed)
python cusparse_crosscard.py   # second kernel, five cards
```

Q Further limitations

- All training results in Section 5.1 are from the L40. On an A40 the method is slower than dense end-to-end (Table 16). The A100 is the weakest card: at 90% sparsity the kernel runs at 0.48× dense `bf16`, break-even is at 95.0%, and the bandwidth model overpredicts it by 3.3×. Our model predicts a break-even near 97% on an H100 (Appendix B.3), which we have not measured.
- The six per-card sweeps time the forward product, so the weight-gradient SDDMM is absent from Table 1 and Figure 1. On five cards the proxy agrees with forward+backward timings to within 1.00–1.19× (Appendix B.5), but the L40 is not among them.
- The 1.44× gain over dense has $p = 0.050$ with three seeds; it is significant only in the post hoc analyses (pooling the six dense replicates, or interpolating the crossing), and the A40 result is a single seed (Appendix J).
- The ViT-3072 experiment is inconclusive as a whole: two arms ran on a slower code path at 95%, the dense arm's learning rate lies at the edge of its grid, and SET is more accurate at

95% than at 90%. No exclusion rule was fixed in advance, so we report every cell and use none of them for a claim (Appendix R); it needs a rerun.

- Adoption requires our CUDA kernels and a custom autograd function; we have not tested `torch.compile`, which would treat the kernels as opaque calls, or multi-GPU training. Under data parallelism the block values are ordinary dense tensors of shape (N, kb), so gradient all-reduce is unchanged. The mask update needs no communication: after the all-reduce every replica holds the same block values, hence the same drop set, and a shared random seed gives the same grown blocks. The update itself is cheap: the rewired and never-rewired arms of Table 14 run at 6.33 and 6.39 steps/s. Layer-wise non-uniform block budgets (for example Erdős–Rényi–kernel) are untested for BlockDST.
- SET is 13.5 pp below dense despite being the less constrained pattern (Appendix L). Epoch-matched at batch 256, SET and RigL are as accurate as BlockDST or more (Appendix E), so the large margins over the masked baselines in Section 5 belong to the large-batch configuration and do not isolate the effect of the pattern.
- The baselines run their patterns, not their own kernels; DynaDiag's CUDA kernels are not publicly released. Using DynaDiag's published $1.59\times$ training speedup at 90% (Tyagi et al., 2025a) and our measured sensitivity (giving dense $1.70\times$ more steps gained 1.24 pp), a $1.59\times$ kernel would be worth about 1.1 pp to DynaDiag, an eighth of the 8.6 pp gap at 90%. The ordering holds, but the margin is approximate.
- On the ViT, SET and RigL are more accurate than BlockDST, and at 90% sparsity on a wider ViT diagonal sparsity beats block sparsity by 1.99 pp in an experiment we treat as inconclusive; on the language models, dense and 2:4 reach lower perplexity (Appendix R). Tasks are CIFAR-10 and WikiText-103, without ImageNet (Deng et al., 2009).
- At 98% sparsity dense is ahead by 1.40 pp on the L40 and 2.71 pp on the L40S, since $k = 1$ and 5 leave too few blocks per row for rewiring. At width 32,768 dense leads by 2.20 pp: cuBLAS reaches 289 TFLOP/s while our kernel does not scale with it.
- The main experiments use block size 64. The invariant and block-size ablations (removing the constant-k constraint: -0.36 pp; $b = 16$–128 within 0.3 pp; Table 13) are epoch-matched at width 4096 on the RTX A500, not timed, because a pattern without the invariant has no load-balanced kernel.
- At 95% sparsity the width-16,384 model has the live parameters of a dense model of width 2734. That dense model reaches 0.55 $13.5\times$ sooner on the RTX A500 (Table 13); it has not been run on the L40, so the method beats the dense model it was derived from, not the best use of that parameter budget.
- SM count $\times$ clock is collinear with the other per-SM resources on our cards (Appendix B.3), and hardware counters and locked clocks need administrator rights we do not have, so the limiting resource is not identified (Appendix B.6); a locked-clock sweep on one card, run with those rights, would separate SM clock from memory clock. The kernel runs at 3–16% of tensor-core peak, and cuSPARSE's Blocked-ELL kernel does not follow the same scaling (Appendix B.9), so the law and its constants are properties of our kernel. The cuSPARSE test uses smaller fp16 matrices than our sweep, one run per cell and no L40. All six accelerators are NVIDIA GPUs; whether the result holds for other memory systems and instruction sets is untested.
- Break-even depends on shape as well as card (Appendix B.8), and our forward kernel is about $3\times$ slower than cuSPARSE's Blocked-ELL SpMM on the RTX A500 (Appendix G.2). Newer methods such as ShuffleSparse (Tyagi et al., 2025b), HASTE (Ullah et al., 2026) and SMET (Xiao et al., 2026) target different settings and are compared only through their published figures.

R Transformer and vision-model results

R.1 Vision transformer with matched epochs

A six-block ViT of width 2048 with 302M parameters, 99.9% of them in sparsifiable linear layers, trained on CIFAR-10 for 40 epochs with AdamW, block size 32 and 90% sparsity (Table 25). The runs are matched in epochs, so accuracy does not depend on the card and no timing is claimed.

BlockDST matches dense ($+0.61$ pp; Welch $t \approx 2.2$ with three seeds per arm, not significant) and has the smallest spread of any arm, but it is behind the other two sparse methods: SET is 1.56 pp and RigL 1.07 pp more accurate. The sweep does not bracket every optimum. Dense improves steadily as the rate falls (39.9, 63.9, 79.8%), so its best rate may lie below 10^{-4}, while BlockDST and SET peak at the largest rate tried; we therefore describe the result as parity.

R.2 Language models

LLaMA-style decoders with 6 layers, sequence length 1024 and batch size 8, trained on WikiText-103 for 3600 s with 95%-sparse FFNs and three seeds (Table 26). The stored parameter counts are 214M and 302M; the corresponding dense models have about 411M and 604M parameters (untied embeddings, SwiGLU FFNs).

BlockDST is the only method with a faster step rate than dense, but dense leads it by 5.4 perplexity at d_{model}=2048 and 12.3 at 2560, and 2:4 matches or beats dense at both sizes. The extra 10–16% steps do not compensate for 95% FFN sparsity at this scale, and the gap widens with model size. BlockDST matches SET at 2048 while running faster, and at 2560 SET is 1.5 perplexity better, within one standard deviation. Inverting the step-rate gain with Equation (5) and the RTX PRO 6000's kernel speedup at 95% sparsity on the reference shape ($1.85\times$) gives a sparsifiable fraction of roughly $f \approx 0.2$–0.3, against 0.93 for the MLP; this is an estimate, since the FFN shapes and block size 32 differ from the reference shape. Sparsifying the attention projections as well would leave $k = 3$–4 blocks per row at 95% sparsity, the regime where rewiring degrades (Section 3). Each arm's step rate is constant across its three seeds to within 0.2%, so each model size ran on a single card.

R.3 Vision transformer with equal wall-clock budgets

A ViT with dimension 3072 trained for 600 s (Table 27). We report it in full but draw no conclusion from it, because it shows three anomalies we cannot resolve without a rerun. At 95% sparsity the BlockDST and DynaDiag arms ran on a slower code path (4.84 and 2.38 steps/s, against 8.03 and 5.84 in a second execution of the same configuration). The dense arm, at learning rate 10^{-4}, is 8.8–12.3 pp below every sparse arm at 90%, which suggests its rate is far from its optimum (on the ViT-2048 dense improves monotonically as the rate falls). SET is 6.3 pp more accurate at 95% than at 90%. We flagged the 95% column only after seeing the results, with no rule fixed in advance. The one comparison unaffected by the known code-path problem goes against us: at 90% sparsity diagonal sparsity (DynaDiag) is 1.99 pp ahead of BlockDST at an equal step rate.

Table 24: Source of each result: For each result, the script that produces it and the data files it reads. In the supplementary material, script paths are under `code/` (figure scripts under `figures/`) and data paths under `figures/`.

result	script, then data files
Table 1; Figures 1, 3 and 6	`bench/bandwidth_law.py` `data/*.json` (6 cards, 54 rows)
Table 2; Figure 5	`experiments/run_time_to_target.py` `data/time_to_target/` (36 runs, L40)
Table 16; Appendix H.1	`experiments/run_time_to_target.py` `data/time_to_target/*_A40.json` (12 runs)
Figure 11; Appendix K	`experiments/run_time_to_target.py` `data/time_to_target/` (12 runs, 60 s cap)
Figure 12; Appendix L	`experiments/run_equal_walltime.py` `data/equal_walltime/` (45 runs)
Table 25	ViT training script `data/vit_sweep/` (26 runs)
Table 26	`experiments/run_lm_equal_walltime.py` `data/lm/` (27 runs)
Table 27	`experiments/run_equal_walltime.py` `data/vit_equal_walltime/` (26 runs)
Table 10	`bench/kernels/block_spmm_bench.cu` `bench/results/spmm_sweep.csv`
Figure 10	`figures/make_fig_flops.py` `data/time_to_target/` (L40, width 16,384)
Table 11	`experiments/run_block_reg.py` `data/block_reg/` (15 runs)
Break-even (Table 1, Figure 1(b)); regression intervals; Table 6	`figures/review_stats.py`, `figures/model_compare.py` `data/*.json`
Pooled and interpolated time-to-target (Table 19)	`figures/review_t2t.py` `data/time_to_target/`
Table 12	`experiments/run_width_law*.py`, `run_erk_check.py` `data/ablation/width/`
Table 13	`experiments/run_review_local.sh` `data/ablation/local/`, `data/ablation/t2t_a500/`
Table 14	`experiments/run_review_local3.sh` `data/ablation/static/`
Delayed rewiring (Appendix E)	`experiments/run_review_local4.sh`, `figures/review_v66.py` `data/ablation/delayed/` (1 run, with the GPU watchdog log)
Section 4.2; Figure 3(c); Table 9	`bench/kernels/block_spmm_bench.cu`, `figures/cusparse_crosscard.py`, `data/cusparse/` (5 cards, 90 configurations each)
cuSPARSELt 2:4 forward product (Appendix O.1)	`bench/compare_sparse_kernels.py` `data/kernel/kcmp24_rtxpro6000_559793.txt` (one run)
Table 7	`bench/layer_step_bench.py` `data/fwdbwd/` (five cards)
Appendix B.4; cuSPARSE comparison (Appendix G.2)	`experiments/run_review_local2.sh` `data/clock/`, `data/kernel/`
Cluster reruns (Appendix B.4)	`bench/bandwidth_law.py`, `figures/cusparse_crosscard.py` `data/rerun/` (L40S, RTX PRO 6000)

Table 25: ViT at 90% sparsity: best-epoch test accuracy (%) for each learning rate. Bold marks each arm's best rate, run with three seeds (mean ± sd); other cells are single runs, except dense and RigL at 10^{-3}, which are means of two (dense diverged at that rate).

learning rate	BlockDST	dense	SET	RigL
1×10^{-3}	$\mathbf{80.41 \pm 0.10}$	39.92	$\mathbf{81.97 \pm 0.49}$	79.32
3×10^{-4}	78.84	63.89	79.54	$\mathbf{81.48 \pm 0.28}$
1×10^{-4}	75.07	$\mathbf{79.80 \pm 0.46}$	76.28	76.34

Table 26: Language models after one hour of training. Validation perplexity (mean ± sd over three seeds) and step rate.

d_{model}	arm	val. ppl	step/s	steps in 1 h	step rate vs dense
2048	dense	$\mathbf{29.53 \pm 0.75}$	6.93	24,937	1.00×
	2:4	$\mathbf{29.20 \pm 0.84}$	6.77	24,388	0.98×
	SET	34.80 ± 1.45	6.88	24,773	0.99×
	BlockDST	34.98 ± 0.65	**7.62**	27,437	1.10×
2560	dense	$\mathbf{37.46 \pm 1.72}$	4.92	17,719	1.00×
	2:4	$\mathbf{36.17 \pm 0.84}$	4.80	17,267	0.98×
	SET	48.23 ± 2.54	4.90	17,626	1.00×
	BlockDST	49.75 ± 3.39	**5.72**	20,594	1.16×

Table 27: ViT-3072 after 600 s: test accuracy (%) and step rate, three seeds. Inconclusive (see text); no claim in the paper rests on it.

	95% sparse		90% sparse	
arm	accuracy	step/s	accuracy	step/s
BlockDST	47.46 ± 0.50	4.84	67.69 ± 0.25	5.81
DynaDiag	41.24 ± 0.29	2.38	$\mathbf{69.68 \pm 0.51}$	5.83
SET	$\mathbf{72.50 \pm 0.90}$	5.80	66.24 ± 0.65	5.79
dense (10^{-4})	57.25 ± 0.93	6.05	57.42 ± 0.35	6.05