BITSPARSE: COMPACT ACTIVATION STORAGE FOR ReLU FEED-FORWARD NETWORKS

Anonymous authors
Paper under double-blind review

ABSTRACT

Reducing memory use in large language model (LLM) training is an ongoing challenge. Activation checkpointing saves memory by recomputation, while lossy compression changes the stored activations. The reemergence of sparse ReLU and ReLU^2 activations creates a third opportunity: losslessly compressing the activation cache itself. We present BitSparse, a GPU-resident lossless compression format for sparse activations. Compression and reconstruction run in custom Triton kernels fused into an autograd function, shrinking the activation cache while recovering it exactly for the backward pass. Bfloat16 (BF16) and FP8 activations are supported. Optional sign-bit packing further increases compression up to close to the theoretical limit. We analyze its compression ratio and compare it to the theoretical optimum. BitSparse lowers per-token VRAM by 31% and 30% Modded-NanoGPT and Nemotron-H respectively, with overheads of no more than 1% at practical sequence lengths. Additionally, a new preallocated buffer system avoids host synchronization during training when variable size tensors are allocated.

1 INTRODUCTION

The memory required to train large language models limits the batch sizes and sequence lengths that fit on an accelerator. Alongside weights, gradients, and optimizer states, backpropagation requires activations retained from the forward pass. Parameter-related storage is largely independent of input size, whereas saved activations and some temporary buffers grow with the number of tokens processed simultaneously. Reducing this input-dependent memory cost is therefore important even when the model weights already fit on the device.

Feed-forward layers are a natural target. They expand each token's hidden representation, typically by a factor of 3–5, and retain intermediate values for computing weight gradients. While the original Transformer Vaswani et al. (2017) used quadratic memory, more recent memory-efficient attention (Rabe & Staats, 2022; Dao et al., 2022) and state-space architectures such as Mamba (Gu & Dao, 2024) are both linear in the number of tokens, so the feedforward layers make up a significant portion of the memory cost.

Certain activations produce sparse outputs that can be exploited to reduce memory usage. The ReLU activation (ReLU) maps nonpositive inputs to exact zeros (Nair & Hinton, 2010). Large language models (LLMs) have used alternative activations such as GELU Hendrycks & Gimpel (2016) and gated activations such as SwiGLU. However, recently squared ReLU (ReLU^2) activations have demonstrated competitive efficiency in LLMs (So et al., 2021) and are used in the Nemotron family and Modded-NanoGPT (Blakeman et al., 2025; Basant et al., 2025; Adler et al., 2024; Jordan et al., 2024a). Sparsity naturally emerges in training without any sparsity-inducing objective (Li et al., 2023; Mirzadeh et al., 2024). In our measurements, the pretrained Nemotron-H model has approximately 92% feed-forward activation sparsity.

Exploiting this redundancy requires an inexpensive enough representation to use during backpropagation with minimal latency. Alternatives such as activation checkpointing reduce storage by recomputing forward intermediates (Chen et al., 2016), while lossy activation compression trades accuracy for memory (Chen et al., 2021; Shamshoum et al., 2025). This raises a simple question: **can the**

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.

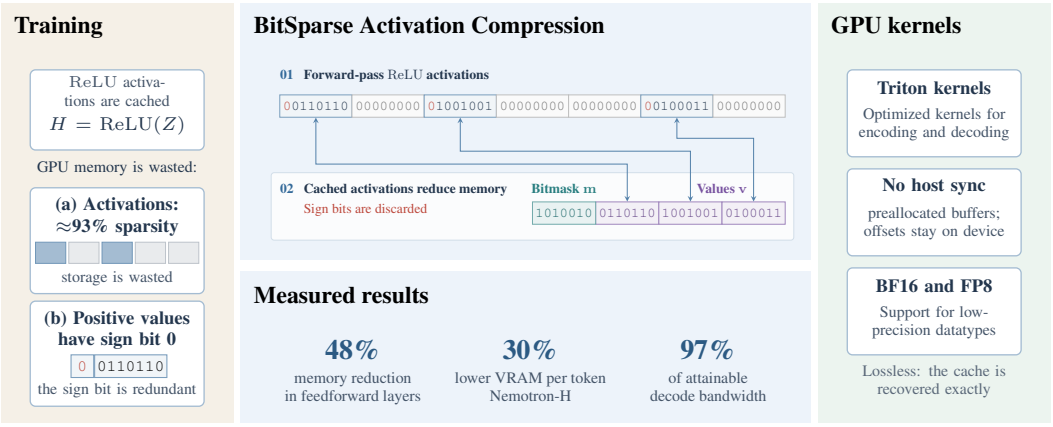


Figure 1: Overview of BitSparse. Caching ReLU activations for backpropagation is expensive: most entries are exact zeros, and every stored value carries a redundant sign bit. BitSparse caches only the nonzero values and reconstructs them with optimized Triton kernels.

exact zeros already produced by ReLU-family LLMs be converted into activation-memory savings without recomputation or approximation?

We present *BitSparse*, a GPU-native format for lossless compression of sparse activations using a bitmask and compact nonzero values. Custom Triton kernels compress activations during the forward pass and reconstruct them as needed during backward. A shared value buffer avoids host synchronization. BitSparse supports the ReLU and ReLU^2 activations with optimized kernels for minimum overhead. Figure 1 illustrates the approach. Our contributions are:

- A GPU activation-storage scheme for ReLU and squared ReLU, with tile-parallel reconstruction, integration into feed-forward backpropagation and a preallocated buffer to avoid host synchronization.
- An information-theoretic analysis of sparse activation storage, together with compression ratios and break-even sparsity thresholds for the implemented representation.
- Evaluations spanning activation sparsity, codec throughput, MLP comparisons, and language-model forward/backward workloads. BitSparse reduces peak allocated memory by up to 48% in an MLP, and reduces the memory per token by approximately 31% for Modded-NanoGPT and 30% for Nemotron-H, with minimal runtime overhead.

2 Background

2.1 ReLU activations and their gradients

We review backpropagation through a ReLU^2 activation followed by a linear projection; the ReLU derivation is given in Appendix C.1. The input is $Z \in \mathbb{R}^{M\times N}$ and the weight is $W \in \mathbb{R}^{D\times N}$, where M is the number of activation rows (for example, tokens flattened across a batch), and N and D are the input and output feature dimensions, respectively. Let $\mathrm{ReLU}(z) = \max(0, z)$ act elementwise, $G = \partial\mathcal{L}/\partial Y \in \mathbb{R}^{M\times D}$ be the upstream gradient, and $\odot$ denote elementwise multiplication. For the forward pass

$$H = \mathrm{ReLU}(Z), \qquad \tilde{H} = \mathrm{ReLU}^2(Z) = H^2, \qquad Y = \tilde{H}W^\top, \tag{1}$$

the gradients are

$$\frac{\partial\mathcal{L}}{\partial W} = G^\top \tilde{H}, \qquad \frac{\partial\mathcal{L}}{\partial \tilde{H}} = GW, \qquad \frac{\partial\mathcal{L}}{\partial Z} = 2\frac{\partial\mathcal{L}}{\partial \tilde{H}} \odot \mathbb{I}[Z > 0] \odot Z, \tag{2}$$

The activation-dependent information required for these gradients is contained in the positive values of Z and their positions. The same principle applies to differentiable pointwise functions of $\mathrm{ReLU}(Z)$, such as $\mathrm{ReLU}^3(Z)$ or $\sin(\mathrm{ReLU}(Z))$, although these are less popular.

2.2 ENTROPY OF SPARSE ReLU ACTIVATIONS

We wish to losslessly compress the activations H. Let $L = MN$ be the number of logical entries, excluding padding, and let $s = \|H\|_0$ denote the number of nonzero values. We denote the zero fraction, or *activation sparsity*, by $\rho = 1 - s/L$. Let d denote the storage width of the numerical format in bits, with $d = 16$ for BF16 and $d = 8$ for FP8. Each positive value has a redundant sign bit, leaving $d - 1$ non-sign bits.

For independent symbols, the average lossless code length per symbol can approach the Shannon entropy for long sequences. The following idealized model is used to obtain a reference entropy.

Assumption 2.1 (Abstract sparse payload model)**.** The entries of H are independent and identically distributed abstract symbols. Each entry is the distinguished zero symbol with probability ρ, where $0 < \rho < 1$, and otherwise carries a uniformly distributed $(d-1)$-bit payload. The 2^{d-1} payload symbols are distinct from the zero symbol.

All logarithms are base two and h_2 denotes the binary entropy function; Appendix B gives its definition together with the proof of the following result.

Theorem 2.2 (Entropy of sparse activations)**.** *Under Assumption 2.1, the entropy and optimum asymptotic compression ratio relative to dense d-bit storage are*

$$\mathcal{H}(H) = L\big[h_2(\rho) + (1-\rho)(d-1)\big], \qquad R(\rho) = \frac{dL}{\mathcal{H}(H)}. \tag{3}$$

The entropy per entry consists of the support entropy $h_2(\rho)$ and the expected nonzero payload cost $(1-\rho)(d-1)$. Lemma B.1 shows that this is an upper bound for actual nonnegative floating-point activations: restricting payloads to valid positive values, using nonuniform distributions, or introducing correlations across positions can only lower the entropy.

3 METHODOLOGY

3.1 OVERVIEW

We propose ***BitSparse***, an activation compression method for ReLU-like activations (e.g. ReLU^2) that adopts the notation and storage conventions of Section 2.2. BitSparse stores only the nonzero values, together with small index buffers for reconstruction, so the activation cache shrinks while the feed-forward projection is still computed densely. Activations are compressed before caching and reconstructed during the backward pass, preserving values in the selected storage format.

3.2 TILE-WISE SPARSE ACTIVATION FORMAT

BitSparse is a tile-based sparse representation for moderately sparse tensors. It partitions H into a regular grid of $B \times B$ tiles, with $B = 64$ by default, so the grid has

$$G_M = \left\lceil \frac{M}{B} \right\rceil, \qquad G_N = \left\lceil \frac{N}{B} \right\rceil \tag{4}$$

rows and columns. Tiles are ordered row-major, with tile (u, v) identified by $t = uG_N + v$, giving $T = G_M G_N$ tiles. Boundary tiles are logically padded with zeros so every tile has $B^2 = 4096$ entries. The representation of H consists of three arrays: a bitmask $\mathbf{m}$ with one bit per activation position (including padding), a compact value array $\mathbf{v}$ holding only positive values, and a 32-bit prefix array $\mathbf{p}$. For a tile t, the bitmask marks the nonzero entries and the value array collects the corresponding positives; the full arrays concatenate the per-tile arrays in tile order, following row-major order within each tile,

$$\mathbf{m}_t = \mathbb{I}[H_t > 0] \in \{0,1\}^{B \times B}, \tag{5}$$

$$\mathbf{v}_t = \mathrm{concat}\{h \in H_t : h > 0\} \in \mathbb{F}_d^{s_t}, \tag{6}$$

$$\mathbf{m} = \mathrm{concat}\{\mathbf{m}_t : t \in [0, \ldots, T-1]\} \in \{0,1\}^{TB^2}, \tag{7}$$

$$\mathbf{v} = \mathrm{concat}\{\mathbf{v}_t : t \in [0, \ldots, T-1]\} \in \mathbb{F}_d^{s}. \tag{8}$$

Here $\mathbb{F}_d$ denotes the storage format of Section 2.2, s_t is the number of nonzero values in tile t, and $\sum_{t=0}^{T-1} s_t = s$. Together, $\mathbf{m}$ and $\mathbf{v}$ determine H exactly. In memory the mask is packed into unsigned bytes, one bit per padded position, which avoids the eight bits per element of a PyTorch Boolean tensor. To locate each tile's values directly, a prefix array holds their starting logical offsets,

$$p_t = \sum_{q=0}^{t-1} s_q \quad (0 \le t < T), \qquad \mathbf{p} \in \mathbb{I}_{32}^{T}, \tag{9}$$

where $\mathbb{I}_{32}$ is the set of 32-bit unsigned integers and the empty sum gives $p_0 = 0$. Consequently, each 64×64 tile requires 512 bytes of mask metadata and 32 bits of prefix metadata; the full sparse object additionally records the logical shape, tile geometry and stored dtype on the CPU.

3.3 Sign-bit packing

Because the output of ReLU activations is always nonnegative, our method can optionally remove the redundant sign bit. BF16 can be reduced to 15 bits and FP8 values to 7 bits. Zero entries are reconstructed as positive zero. We refer to BitSparse with sign-bit packing enabled as ***BitSparse-sb***.

GPU memory is byte-aligned, so a custom Triton kernel is written for packing individual bits. Let u_i be the unsigned bit pattern of the ith compact value v_i with its sign bit removed, and let w_j be the jth output byte. The corresponding input index is $i = \lfloor 8j/(d-1) \rfloor$, with bit offset $k = 8j \bmod (d-1)$ within u_i. The output byte is

$$w_j = [(u_i \gg k) | (u_{i+1} \ll (d-k-1))] \;\&\; (11111111)_2. \tag{10}$$

Here i indexes compact values, j indexes packed bytes, and k is a bit offset; out-of-range input words are treated as zero. The sign-bit packing reads two words and writes one byte at a time without atomics for high speed. Decoding works in an analogous way, also without atomics.

3.4 GPU compression and reconstruction

Compression is performed entirely on the GPU in three stages. First, a Triton tile kernel loads one tile, evaluates $H_t > 0$, constructs $\mathbf{m}$ and the count s_t. Second, a scan over the per-tile counts constructs the starting offsets $\mathbf{p}$ in Equation (9). Third, a second Triton kernel expands each tile mask and scatters positive values into $\mathbf{v}$ according to Equation (8). Optionally, a final Triton kernel strips the sign bits and packs values into bytes according to Equation (10).

Reconstruction is performed by a single Triton kernel. Each tile identifies the location of its values using p_t and loads the relevant values along with the bitmask. Next, a prefix sum is used to compute the location of each value in the final array. Finally, a masked gather is used to produce the decompressed tensor H, with sign-bit unpacking if necessary. Optionally, the values may be squared during decoding for the ReLU^2 activation.

Two further kernels fuse the backward-pass operations. The ReLU backward pass requires only the support mask, so a single kernel multiplies the incoming gradient by that mask in place, without reconstructing the activation. The ReLU^2 pass performs the same operation with an additional dependence on Z, again in a single fused kernel. Furthermore, our blockwise structure allows for computing the backward matmul (Equations 19, 2) in blocks to avoid materializing the full activation to reduce peak memory further.

3.5 Shared value buffers and implementation details

The size of the value buffer $\mathbf{v}$ depends on the data, so allocating it on the host requires reading the device-resident count s, which interrupts asynchronous kernel submission. To avoid this, BitSparse optionally appends the values of different activations to a `TensorBuffer`, a large preallocated block with device memory offsets. Compressing a tensor appends its values and saves the starting offset on the device; decoding reads from the buffer using that offset, implemented in the kernels as `val_idx = tl.load(prefix_ptr + tile_id) + tl.load(vals_offset_ptr)`. This allows BitSparse to operate completely asynchronously. The buffer is reset at the end of every train step, and must be sized to not overflow. We recommend setting it as large as possible to use the unallocated memory.

Table 1: Theoretical compression ratios of BitSparse, BitSparse-sb, and the theoretical optimum under Assumption 2.1. Boundary padding overheads are neglected.

	BF16			FP8		
Sparsity ρ	BitSparse	BitSparse-sb	Ideal	BitSparse	BitSparse-sb	Ideal
50%	1.776×	1.881×	1.882×	1.598×	1.775×	1.778×
75%	3.195×	3.363×	3.508×	2.660×	2.901×	3.123×
90%	6.135×	6.380×	8.126×	4.425×	4.684×	6.843×

The buffer gives a latency improvement for small sizes which are CPU bound. Appendix E.6 shows a 5.2% runtime reduction at 32 tokens, diminishing as batch size increases and becoming negligible beyond 128 tokens. Furthermore, this allows BitSparse to work with `torch.compile`.

3.6 Sparse activation checkpointing for feed-forward layers

Section 2.1 shows that the positive preactivation values and their positions contain the activation-dependent information required for backpropagation. We therefore cache a compressed representation of $\mathrm{ReLU}(Z)$ for both activation functions. For ReLU^2, the unsquared activation is stored to avoid a more unstable square root and keep numerical consistency with the forward pass.

Our method is implemented in PyTorch as a custom `torch.autograd.Function` class that combines the ReLU or ReLU^2 activation along with a linear matrix multiplication; Algorithm 1 gives its forward and backward passes. Sign-bit packing and `TensorBuffer` are optional, with support for Bfloat16 and FP8 (both e5m2 and e4m3) with scaling Micikevicius et al. (2022).

The overhead of compressing and decompressing is fairly minimal for typical LLM training. Besides using highly optimized Triton kernels, the overhead is asymptotically reduced as the LLM's hidden dimension increases. To see this, observe that the FLOP complexity for the linear layer $HW^\top$ is $\mathcal{O}(MND)$. Since the size of H is MN, the ratio overhead from compressing H to the matrix multiplication is $\mathcal{O}(1/D)$.

4 Theoretical compression analysis

Theorem 4.1 (BitSparse compression ratios)**.** *For $B = 64$ and negligible boundary padding and per-tensor overhead, the compression ratios without and with sign-bit packing are, respectively,*

$$R_{\mathrm{BitSparse}} \simeq \frac{d}{d(1-\rho)+1+1/128}, \qquad R_{\mathrm{sb}} \simeq \frac{d}{(d-1)(1-\rho)+1+1/128}. \tag{11}$$

To see this result, observe the storage format has three components. Each entry requires on average one mask bit, $32/B^2 = 1/128$ prefix bits and $(1-\rho)$ nonzero values at d bits unpacked or $d-1$ bits packed. Dividing the dense cost d by this gives the ratios above. A full derivation is given in Appendix B. This is exact when there no padding and alignment mismatches.

Table 1 compares the ratios with and without sign-bit packing against the ideal rates of Equation (3). Compared with the ideal rate in Theorem 2.2, the BitSparse-sb has an excess cost per entry of

$$\frac{S_{\mathrm{BS}}}{L} - \big[h_2(\rho) + (1-\rho)(d-1)\big] \simeq 1 - h_2(\rho) + \frac{1}{128}. \tag{12}$$

At $\rho = 1/2$ the excess is exactly the prefix cost, $1/128$. Storage is reduced whenever sparsity exceeds $\rho > (1+1/128)/d$ without packing, or $\rho > 1/[128(d-1)]$ with packing. These thresholds are small, roughly 6% and 13% for unpacked BF16 and FP8 and below 0.2% once packed, so any realistic activation sparsity reduces storage. The benefit of sign-bit packing is most notable when there are many nonzero values, and almost entirely compensates for the extra bitmask storage.

5 Related work

Lossy activation compression and low-precision training. Multiple methods have investigated lossy compression for activations. ActNN (Chen et al., 2021) uses randomized mixed-precision

quantization and analyzes the resulting gradient variance. GACT (Liu et al., 2022) extends activation compression to generic network architectures, dynamically choosing tensor compression ratios. CompAct (Shamshoum et al., 2025) uses random projections to store low-rank activations.

Low precision datatypes such as FP8 halves the storage per value compared to BF16 and offers speed-ups with hardware support (Micikevicius et al., 2022). COAT (Xi et al., 2025) targets both activation and optimizer-state memory through FP8 quantization. Hardware-accelerated 2:4 sparsity for ReLU^2 activations has been investigated by Haziza et al. (2025) for training and inference where activations are pruned so only 2 of every 4 activations are nonzero.

Lossless memory reduction techniques. A few methods have explored lossless activation storage with ReLU activations. BitTrain (Hosny et al., 2021) stores the nonzero activation values for training convolutional neural networks (CNNs) on CPU-based edge devices. cDMA (Rhu et al., 2018) exploits activation sparsity to reduce transfers between GPU and host memory during training for host activation caching, also for CNNs. Gist (Jain et al., 2018) compresses sparse activations with a 8-bit CSR format. BitSparse extends these methods in several ways, including the standard GPU training setup without host synchronization, dedicated fused kernels for low overhead, low precision support and sign-bit packing for higher compression. We view these as essential for adoption in the modern LLM training era.

In addition, nvCOMP (NVIDIA Corporation, 2026) provides optimized codecs for general and floating data, which can be applied to activations. Activation checkpointing recomputes activations during backpropagation (Chen et al., 2016), sacrificing speed for lower memory. Korthikanti et al. (2023) showed this to be effective in large LLM training setups with thousands of GPUs.

Activation sparsity for efficient computation. The lazy neuron study (Li et al., 2023) documents the emergence of sparse ReLU activations in trained transformers. Exploiting the sparsity in ReLU networks for inference has long been explored in CNNs Parashar et al. (2017). More recently, ReLU Strikes Back (Mirzadeh et al., 2024) investigated ReLU-based LLMs, exploiting their activation patterns to reduce inference computation. These methods avoid loading weights corresponding to zero activations. Unfortunately, they do not generalize to training, different tokens activate different channels.

6 Experiments

6.1 Sparsity in LLMs

We first examine activation sparsity in the feed-forward layers of two LLMs using ReLU^2: a 162M-parameter Modded-NanoGPT transformer (Jordan et al., 2024a) and the 8B-parameter Nemotron-H-8B-Base-8K model. These models differ in both size and architecture: Modded-NanoGPT uses transformer blocks, whereas Nemotron-H combines attention, Mamba, and MLP blocks. Full details are given in Appendix E.1.

The Modded-NanoGPT model is trained following the Track 3 baseline procedure. Activation sparsity is measured on the validation set every 300 steps for every layer. Figure 5 shows an initial sparsity of approximately $\rho = 0.5$, rapidly increasing to $\rho = 0.82$. This phenomenon has been observed in other works Haziza et al. (2025).

Next, we evaluate the pretrained Nemotron-H model. The mean activation sparsity is 0.9273, with minimum batch-wise mean sparsity over 256 evaluation batches of 0.9176, corresponding to 92.73% and 91.76% sparsity, respectively. The sparsity is extremely consistent, with the minimum batch-wise mean sparsity only 0.97 percentage points below the mean, meaning activation compression can be used effectively.

6.2 Benchmarking and Ablations

BisSparse is benchmarked in Table 2, which measures encoding and decoding time for a 1 GiB tensor with $\rho = 0.5$ and $\rho = 0.9$, with and without sign-bit packing. Decoding reaches 97.4% of the attainable bandwidth at $\rho = 0.5$ without sign-bit packing, while encoding is slower at 60.9%. BitSparse-sb is slower, since an additional kernel is needed. Higher sparsity reduces encoding time, although effective bandwidth utilization is lower since less data is moved.

Table 2: Timings for encoding and decoding a 1 GiB sparse BF16 tensor ($65{,}536 \times 8{,}192$) with and without sign-bit packing. The bandwidth utilization (Memory BW) includes both read and write. Bandwidth utilization is compared with the attainable throughput measured with NVbandwidth, with the device's nameplate bandwidth (936 GB/s) in parentheses.

Sign-bit packing	ρ	Time / ms		Rate / GiB/s		Memory BW / %	
		Encode	Decode	Encode	Decode	Encode	Decode
No	0.5	3.52	2.20	284.1	454.5	60.9 (50.9)	97.4 (81.5)
Yes	0.5	7.03	2.79	142.2	358.4	29.9 (25.0)	75.3 (63.0)
No	0.9	3.31	2.22	302.1	450.5	52.3 (43.8)	78.0 (65.3)
Yes	0.9	4.84	2.92	206.6	342.5	35.4 (29.6)	58.7 (49.1)

Table 3: Peak allocated memory and time per step for the ReLU^2 MLP benchmark, using BF16 and FP8 activations at sparsity $\rho = 0.5$ and 0.9. CompAct is a lossy reference. Bold and italic values indicate the lowest and second-lowest lossless results. Dashes indicate unavailable results.

	Memory / MiB				Time / ms			
	BF16		FP8		BF16		FP8	
Method	0.5	0.9	0.5	0.9	0.5	0.9	0.5	0.9
Dense	12346	11344	9492	9492	*1827*	*1827*	*395*	395
BitSparse	8806	*5896*	*7180*	*6617*	**1822**	**1815**	**391**	*391*
BitSparse-sb	*8637*	**5860**	**7042**	**6551**	1844	1828	397	**391**
Checkpoint	**6713**	6713	7417	7417	2144	2144	507	507
LZ4	11086	8052	8777	8242	3565	2584	693	615
Zstd	10064	7517	8071	7664	4937	3311	990	784
Cascaded	12207	8462	9157	8921	1957	2185	469	461
Bitcomp	12285	9154	8870	8693	1926	2144	461	452
CSR	–	15859	–	–	–	2308	–	–
CompAct-2	14111	14767	10078	13284	2754	2754	516	516
CompAct-4	12581	13237	9238	12519	2037	2037	399	399
CompAct-8	11158	12472	8854	12138	1692	1692	346	346

To assess the benefit of our custom kernels, we compare our Triton implementation with a pure PyTorch implementation of BitSparse on a $65{,}536 \times 8{,}192$ BF16 tensor. Both versions produce identical outputs. The Triton implementation encodes $7.59\times$ faster and decodes $7.91\times$ faster, while reducing peak allocated memory by 37.5% and 21.9%, respectively (Table 12, Appendix E.4).

The fused backward gradient kernel is compared with the standard PyTorch version on a $16{,}000 \times 21{,}504$ BF16 tensor with 50% sparsity. BitSparse reduces peak allocated memory by 46.8% for ReLU and 60.9% for ReLU^2, while reducing runtime by 17.2% and 39.2%, respectively. Appendix E.5 gives the setup, and Table 13 reports the results.

6.3 Comparisons on training

We benchmark activation storage on a stack of eight residual MLP blocks (Appendix D.3), with BF16 parameters, BF16 or FP8 activation storage, on a forward and backward pass. Optimizer stats are not included, since neither BitSparse or any baselines affect optimizer buffers. The comparisons cover ReLU and ReLU^2 at sparsity $\rho = 0.5$ and 0.9 in both BF16 and FP8. The `TensorBuffer` is not used for these experiments so peak memory allocation can be measured.

Comparisons are made against activation checkpointing of each FFN block Chen et al. (2016); lossless codecs as implemented by nvCOMP NVIDIA Corporation (2026), LZ4 Collet (2011), Zstd Collet & Kucherawy (2021), Bitcomp, and Cascaded; compressed sparse row (CSR); and CompAct Shamshoum et al. (2025), a lossy reference. The peak allocated memory and time per step is recorded. Every method except CompAct produces identical gradients up to rounding.

Table 3 reports the ReLU^2 results in both BF16 and FP8; Appendix E.3 gives the corresponding ReLU results. At $\rho = 0.9$, BitSparse is the most memory-efficient lossless method, dropping

Table 4: Effect of batch size on peak allocated memory and time per step for the ReLU^2 MLP at sparsity $\rho = 0.9$. Bold and italic values indicate the lowest and second-lowest results, respectively.

Batch	Memory / MiB				Time / ms			
size	Dense	Checkpoint	BitSparse	BitSparse-sb	Dense	Checkpoint	BitSparse	BitSparse-sb
32	3054	2879	*2879*	**2879**	**12**	14	*12*	13
128	3094	2899	*2896*	**2896**	**18**	22	*19*	19
512	3264	2982	*2972*	**2971**	*64*	74	**64**	64
2000	3932	3294	*3252*	**3249**	243	284	**240**	*241*
8000	7107	4704	*4384*	**4368**	929	1086	**919**	*926*
32000	19811	10704	*8916*	**8846**	3640	4268	**3604**	*3628*

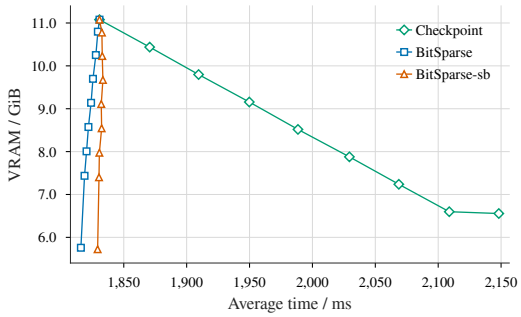


Figure 2: Memory–runtime trade-off for the ReLU^2 MLP at sparsity $\rho = 0.9$, varying the number of compressed or checkpointed layers from 0 to 8. Each point represents one layer-count configuration, lines connect configurations in layer-count order.

VRAM usage by up to 48% relative to dense storage. Sign-bit packing reduces memory further, at a small cost in time. For BF16, checkpointing uses the least memory at $\rho = 0.5$, but BitSparse overtakes it at $\rho = 0.9$. For FP8, BitSparse-sb uses the least memory at both sparsity levels. Checkpointing carries a substantially larger runtime penalty. The other lossless compression methods either have lower compression than BitSparse or run substantially slower.

To further investigate the memory–runtime trade-off between BitSparse and checkpointing, we vary the number of compressed layers in our model from 0 to 8. Figure 2 shows the frontier for ReLU^2 at $\rho = 0.9$. Appendix E.7 gives the other three configurations. BitSparse dominates the Pareto-frontier, particularly at higher sparsity levels. Also visible is the drop in VRAM on the last data point caused by lower transient memory usage from our fused pointwise gradient kernel. Furthermore, at high sparsity, our ReLU^2 kernel is faster than the standard implementation.

We also vary the batch size from 32 to 32,000 (Table 4). As the batch grows, the activations that BitSparse compresses account for more memory so memory savings increase from 6% at 32 rows to 55% at 32,000. Runtime stays within 1% of dense storage from 512 rows upward. BitSparse therefore occupies a different point on the memory–compute frontier from checkpointing: its advantage grows with naturally occurring activation sparsity, while avoiding recomputation.

6.4 Memory and runtime in LLMs

As a larger training example, we evaluate peak VRAM usage and per-step latency when training Modded-NanoGPT and Nemotron-H across a range of sequence lengths (Appendix D.4), comparing dense activation storage, BitSparse(-sb), and activation checkpointing. Again, no optimizer buffer is used. A numerical equality check confirms that all methods reproduce the gradients up to floating-point rounding.

Figures 3 and 4 report the memory and runtime results, respectively. Across the measured token counts, VRAM per token on Modded-NanoGPT decreases from 0.402 MiB to 0.277 MiB with BitSparse and 0.276 MiB with BitSparse-sb. On Nemotron-H, the corresponding values are 2.719,

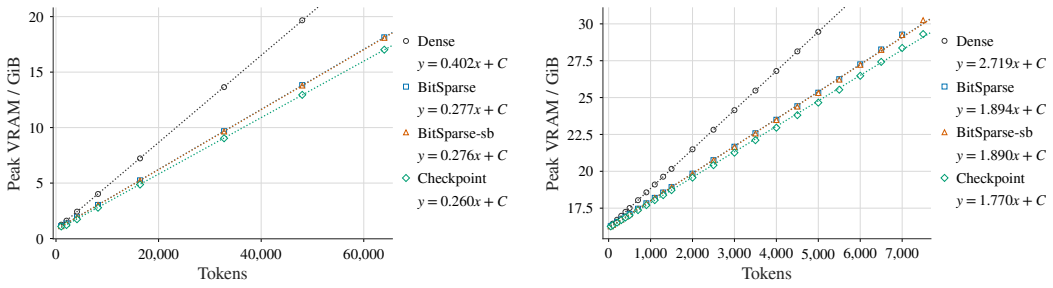


Figure 3: Peak GPU memory for Modded-NanoGPT (left) and Nemotron-H (right) versus the number of tokens processed per step. Markers denote measurements, dotted curves are least-squares linear fits. Legend entries give the fitted slope in units of MiB per token.

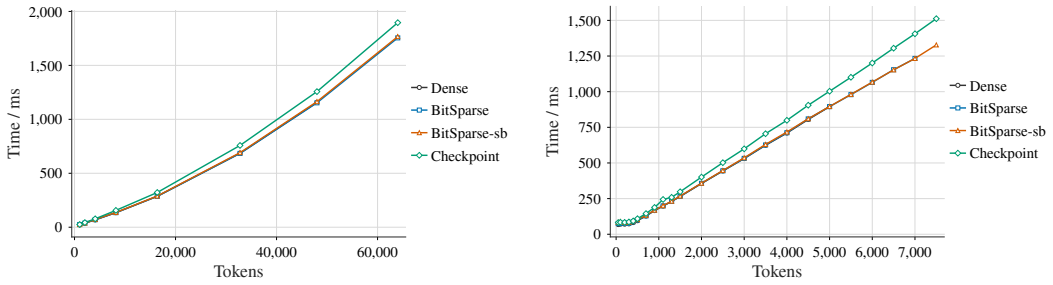


Figure 4: Step time for Modded-NanoGPT (left) and Nemotron-H (right) versus the number of tokens processed per step. Conventions follow Figure 3.

1.894, and 1.890 MiB. This corresponds to an asymptotic VRAM reduction of 31% on Modded-NanoGPT and 30% on Nemotron-H. BitSparse incurs a negligible performance penalty. Checkpointing uses less memory but generally incurs a notable runtime cost. Appendix A.1 and Table 5 give a detailed breakdown of the memory used per token for Nemotron-H.

6.5 Full training comparison

To demonstrate the full recipe, Modded-NanoGPT is trained with BitSparse-sb and TensorBuffer over the full 3,350 training steps. Figure 6 in Appendix E.2 shows that the validation loss remains virtually identical to that of the original recipe, ending at 3.2778 for standard training and 3.2780 for BitSparse-sb, a difference of 0.0002, both below the target of 3.28.

Table 7 in Appendix E.2 compares standard and BitSparse training at microbatch sizes of 32 and 64. At size 32, BitSparse is slightly slower (7.668 versus 7.428 seconds per step). Standard training runs out of memory at size 64, whereas BitSparse supports the larger microbatch and reduces its step time to 7.416 seconds, slightly faster than standard training at size 32.

7 Conclusion

We presented *BitSparse*, a GPU-resident tile-wise sparse representation for ReLU and ReLU^2 feed-forward activations that is lossless and can be used as a drop-in replacement to reduce memory usage. It is fast, using optimized fused Triton kernels, with overhead typically less than 1% for large sequence lengths while reducing memory by up to 30%. Because this sparsity arises naturally, BitSparse converts existing exact zeros into memory savings without approximation. BitSparse supports both BF16 and FP8 formats and can operate without host synchronization.

BitSparse depends on the continued success of ReLU-family activations, since its gains rest on exact structural zeros. The scope is limited to feed-forward activations; attention and Mamba layers are not affected. Future work should extend the representation to gated activations and co-design it with checkpointing, offloading, and distributed training.

AI USE STATEMENT

Generative AI assisted with proofreading this paper. The Triton kernels were optimized using AI tools for maximum performance. A simple Autoresearch Karpathy (2026) prompt was used in combination with manual inspection and direction. Our prompt is included in the supplementary materials. After the kernels were optimized, the rest of the supporting software (including PyTorch autograd integration, experimentation and accuracy validation) was written manually with minor help from AI tooling. Experiments were run manually to ensure accuracy.

REPRODUCIBILITY STATEMENT

The supplementary materials include the BitSparse implementation (`lib_sparse`), benchmark scripts (`experiments`), and language-model experiments (`nanogpt` and `nemotron`).

Section 3 describes the storage format, GPU kernels, and shared buffer; Algorithm 1 gives the forward and backward passes. Appendix B contains the proofs of the entropy and compression results. Appendix D specifies hardware, software versions, random seeds, and baselines. Appendices D.3 and D.4 detail the MLP and language-model workloads, precision settings, and timing and memory measurements. Appendix E.1 describes the data and aggregation used to measure activation sparsity, and Appendix E provides the full-training comparison, additional benchmark results, and ablations.

ETHICS STATEMENT

BitSparse compresses intermediate activations during model training and does not itself collect data or introduce a new data source. Reducing GPU memory requirements may make larger training workloads more accessible. Beyond this, any privacy, bias, and misuse risks depend on the use case, rather than BitSparse. Its environmental impact depends on the memory savings and runtime overhead of the complete training workload.

REFERENCES

Bo Adler et al. Nemotron-4 340B technical report, 2024.

Aarti Basant et al. NVIDIA Nemotron Nano 2: An accurate and efficient hybrid mamba-transformer reasoning model, 2025.

Aaron Blakeman et al. Nemotron-H: A family of accurate and efficient hybrid mamba-transformer models, 2025.

Jianfei Chen, Lianmin Zheng, Zhewei Yao, Dequan Wang, Ion Stoica, Michael Mahoney, and Joseph Gonzalez. ActNN: Reducing training memory footprint via 2-Bit activation compressed training. In *Proceedings of the 38th International Conference on Machine Learning*, Proceedings of Machine Learning Research. PMLR, 2021.

Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost, 2016.

Yann Collet. Lz4: Extremely fast compression algorithm, 2011.

Yann Collet and Murray S. Kucherawy. Zstandard compression and the 'application/zstd' media type, 2021.

Tri Dao, Daniel Y Fu, Stefano Ermon, Atri Rudra, and Christopher Re. Flashattention: Fast and memory-efficient exact attention with IO-awareness, 2022.

Albert Gu and Tri Dao. Mamba: Linear-time sequence modeling with selective state spaces, 2024.

Daniel Haziza, Timothy Chou, Dhruv Choudhary, Luca Wehrstedt, Francisco Massa, Jiecao Yu, Geonhwa Jeong, Supriya Rao, Patrick Labatut, and Jesse Cai. Accelerating transformer inference and training with 2:4 activation sparsity, 2025.

Dan Hendrycks and Kevin Gimpel. Gaussian error linear units (GELUs), 2016.

Abdelrahman Hosny, Marina Neseem, and Sherief Reda. BitTrain: Sparse bitmap compression for memory-efficient training on the edge. In *Proceedings of the Sixth ACM/IEEE Symposium on Edge Computing*, 2021.

Animesh Jain, Amar Phanishayee, Jason Mars, Lingjia Tang, and Gennady Pekhimenko. Gist: Efficient data encoding for deep neural network training. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018.

Keller Jordan, Jeremy Bernstein, Brendan Rappazzo, @fernbear.bsky.social, Boza Vlado, You Jiacheng, Franz Cesista, Braden Koszarsky, and @Grad62304977. modded-nanogpt: Speedrunning the NanoGPT baseline, 2024a. URL https://github.com/KellerJordan/modded-nanogpt.

Keller Jordan, Yuchen Jin, Vlado Boza, You Jiacheng, Franz Cesista, Laker Newhouse, and Jeremy Bernstein. Muon: An optimizer for hidden layers in neural networks, 2024b.

Andrej Karpathy. autoresearch, 2026. URL https://github.com/karpathy/autoresearch.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014.

Vijay Anand Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. Reducing activation recomputation in large transformer models. In *Proceedings of Machine Learning and Systems*, 2023.

Zonglin Li, Chong You, Srinadh Bhojanapalli, Daliang Li, Ankit Singh Rawat, Sashank J. Reddi, Ke Ye, Felix Chern, Felix Yu, Ruiqi Guo, and Sanjiv Kumar. The lazy neuron phenomenon: On emergence of activation sparsity in transformers. In *International Conference on Learning Representations*, 2023.

Xiaoxuan Liu, Lianmin Zheng, Dequan Wang, Yukuo Cen, Weize Chen, Xu Han, Jianfei Chen, Zhiyuan Liu, Jie Tang, Joey Gonzalez, Michael Mahoney, and Alvin Cheung. GACT: Activation compressed training for generic network architectures. In *Proceedings of the 39th International Conference on Machine Learning*. PMLR, 2022.

Kai Lv, Yuqing Yang, Tengxiao Liu, Qinghui Gao, Qipeng Guo, and Xipeng Qiu. Full parameter fine-tuning for large language models with limited resources, 2024.

Paulius Micikevicius, Dusan Stosic, Neil Burgess, Marius Cornea, Pradeep Dubey, Richard Grisenthwaite, Sangwon Ha, Alexander Heinecke, Patrick Judd, John Kamalu, Naveen Mellempudi, Stuart Oberman, Mohammad Shoeybi, Michael Siu, and Hao Wu. FP8 formats for deep learning, 2022.

Iman Mirzadeh, Keivan Alizadeh-Vahid, Sachin Mehta, Carlo C. del Mundo, Oncel Tuzel, Golnoosh Samei, Mohammad Rastegari, and Mehrdad Farajtabar. ReLU strikes back: Exploiting activation sparsity in large language models. In *International Conference on Learning Representations*, 2024.

Vinod Nair and Geoffrey E. Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on Machine Learning*, 2010.

NVIDIA Corporation. NVIDIA nvCOMP. https://docs.nvidia.com/cuda/nvcomp/, 2026.

Open Compute Project. Ocp microscaling formats (mx) specification, 2023.

Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Steve Keckler, and William Dally. SCNN: An accelerator for compressed-sparse convolutional neural networks. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, 2017.

Guilherme Penedo, Hynek Kydlíček, Loubna Ben allal, Anton Lozhkov, Margaret Mitchell, Colin Raffel, Leandro Von Werra, and Thomas Wolf. The fineweb datasets: Decanting the web for the finest text data at scale, 2024.

Markus N. Rabe and Charles Staats. Self-attention does not need $o(n^2)$ memory, 2022.

Minsoo Rhu, Mike O'Connor, Niladrish Chatterjee, Jeff Pool, Youngeun Kwon, and Steve Keckler. Compressing DMA engine: Leveraging activation sparsity for training deep neural networks. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018.

Yara Shamshoum, Nitzan Hodos, Yuval Sieradzki, and Assaf Schuster. CompAct: Compressed activations for memory-efficient LLM training. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, 2025.

David R. So, Wojciech Mańke, Hanxiao Liu, Zihang Dai, Noam Shazeer, and Quoc V. Le. Primer: Searching for efficient transformers for language modeling. In *Advances in Neural Information Processing Systems*, 2021.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2017.

Haocheng Xi, Han Cai, Ligeng Zhu, Yao Lu, Kurt Keutzer, Jianfei Chen, and Song Han. COAT: Compressing optimizer states and activation for memory-efficient FP8 training. In *International Conference on Learning Representations*, 2025.

A MEMORY USAGE DURING TRAINING

Reference model and assumptions. Here, we describe in more detail the memory usage involved in training LLMs.

We assume Bfloat16 storage is used everywhere for this analysis (2 bytes per element). Let P be the number of parameters in a model, W be its width, D be its depth and FFN expansion factor r. For LLM training, let the sequence length be n and total number of tokens be N, equal to sequence length times batch size.

The model parameters require storage of $2P$ bytes. The optimizer states require either a buffer for every parameter, like Muon Jordan et al. (2024b) and SGDM, or 2 buffers for optimizers such as Adam Kingma & Ba (2014). Hence at least $2P$ bytes are needed. If gradients are all stored, an additional $2P$ bytes of storage are needed. However, gradient overhead can be removed by fusing the optimizer into the backward step, where the momentum buffers are updated as soon as gradients are computed, removing the need for gradient storage Lv et al. (2024). Hence, in this setup for LLM training or finetuning with Muon, a static memory overhead of $4P$ is required.

The rest of the memory used is for the activations. These scale linearly with input size. While attention complexity is the square of sequence length, the peak memory required is only linear Rabe & Staats (2022). Furthermore, Flash attention Dao et al. (2022) provides a highly optimized attention implementation, where the intermediate computations are recomputed in the backward pass rather than stored. This reduces the cache size to $\mathcal{O}(N)$. State-space methods such as Mamba Gu & Dao (2024) also use fused checkpointed kernels for the backward pass for good memory efficiency without saving the full intermediate computation graph. The storage used is $\mathcal{O}(DWN)$.

The feedforward layers make up the other major portion of the activation cache. A feedforward block consists of an up projection, activation and down-projection, $y = \text{activation}(xW_1^\top)W_2^\top$. This projection to the high dimension activation requires a large amount of storage, since its hidden dimension is rW. The input to the FFN layer is cached. Since the activation is pointwise, it can be recomputed leading to total memory usage $O(D(r+1)WN)$.

Finally, we have pointwise and normalization operations, such as RMS and LayerNorm. These pointwise operations can be efficiently checkpointed with fused kernel implementations; hence these layers do not require significant storage.

A.1 NEMOTRON-H MEMORY USAGE

We will use Nemotron-H (Blakeman et al., 2025) as a reference for size and shapes. Its configuration contains 52 blocks: 24 MLP, 24 Mamba, and 4 attention blocks, with hidden width 4096, MLP width $21{,}504$ (an expansion factor of 5.25), and vocabulary size $131{,}072$, totaling approximately 8.1 billion parameters that occupy 15.089 GiB in Bfloat16. In this paper, we will use BF16 precision for training for simplicity.

Activations and intermediate tensors. Unless stated otherwise, the tensor sizes below are for a micro-batch of 4096 tokens, so that they can be compared directly with the per-token figures in Table 5. A single BF16 residual-width tensor occupies 0.03125 GiB at this length, whereas an expanded MLP tensor occupies 0.16406 GiB. Keeping one expanded tensor for each of the 24 MLPs costs 3.9375 GiB. A conventional ReLU^2 backward graph may retain both the unsquared ReLU output and the squared tensor consumed by the down-projection, requiring two such sets (7.875 GiB), in addition to the MLP inputs and normalization intermediates. The precise saved set depends on fusion and checkpointing.

Mamba's projected gate, convolution inputs, and time-step values span $18{,}560$ features per token; one projected tensor per Mamba block totals 3.398 GiB. Convolution outputs, scan intermediates, and normalization can require additional storage, depending on the kernel and chunking strategy. Attention Q, K, and V together span 6144 features per token, totaling 0.188 GiB across four blocks. Materializing one BF16 attention-score tensor per block would add 4 GiB at length 4096, or 16 GiB at length 8192; memory-efficient attention avoids retaining the full score matrices. Finally, the vocabulary projection produces 1 GiB of BF16 logits at 4096 tokens, and the implementation

Table 5: Analytical activation storage per token for Nemotron-H-8B-Base-8K, comparing dense storage, checkpointing of each feed-forward block, BitSparse, and BitSparse with sign-bit packing. Sizes are computed from the model's tensor shapes, not measured, and cover the forward-pass tensors retained for the backward pass. Components below 1 KiB per token are grouped under *Other*. Values are rounded to the nearest whole KiB, so columns may not sum exactly.

	Storage / (KiB per token)			
Activation	Dense	Checkpoint	BitSparse	BitSparse-sb
Raw inputs retained for block/final norms	424	424	424	424
Mamba packed projection (`gate`, `x`, `B`, `C`, `dt`)	870	870	870	870
Mamba scan output	384	384	384	384
FlashAttention Q/K/V	48	48	48	48
FlashAttention output	32	32	32	32
FFN unsquared ReLU values	1008	0	81	76
FFN masks, prefixes and offsets	0	0	63	63
Other (RMS and attention statistics, metadata)	2	1	2	2
Total	2768	1759	1905	1899

casts these to FP32, whose footprint is 2 GiB. Loss computation and the cast can introduce further temporary allocations.

Implementation optimizations. Two lossless changes to the Nemotron-H implementation keep its measured footprint close to the model's intrinsic memory use. The RMS normalization is fused into a single kernel, so its intermediate statistics and normalized values are recomputed during the backward pass rather than retained. The vocabulary projection head is evaluated in chunks along the vocabulary dimension, so the full FP32 logits tensor described above is never materialized at once and only the chunk being processed is live. Neither change alters the computation performed, only where its intermediates are stored, so the results match the unfused formulation up to floating-point rounding. Both are applied to the dense baseline as well as to BitSparse, so the comparisons in Section 6.4 reflect the choice of activation storage rather than incidental implementation differences.

Effect of activation compression. Using the mean observed sparsity $\rho = 0.9273$ as an illustrative cache sparsity, Equation (11) reduces one set of expanded BF16 MLP activations from 3.9375 GiB to approximately 0.534 GiB without sign-bit packing or 0.516 GiB with packing. These estimates include 0.246 GiB of masks and 0.00192 GiB of tile-prefix metadata. Using the minimum batch-wise mean sparsity $\rho = 0.9176$ instead gives 0.572 and 0.552 GiB, respectively. The sparsity measurements were taken after squaring, while the cache stores unsquared ReLU values, so these are estimates rather than exact cache measurements. Reconstruction can still require a dense temporary, and the attention, Mamba, residual, and output-head tensors are not compressed by this method.

Scaling and relation to the benchmark. At fixed precision, parameter-related storage is independent of input length and microbatch size. The linear-width activation footprints above scale by $M/4096$, where M counts all tokens in the microbatch; materialized attention scores instead scale with batch size times squared sequence length. Wider dense projections increase parameter counts approximately quadratically, while the listed feature tensors grow linearly with width when expansion ratios and head dimensions are fixed. The measurements in Appendix D.4 omit optimizer updates and release parameter gradients as they are produced. They therefore exclude optimizer state and a retained full-model gradient buffer, but transient gradients and operator workspaces still contribute to the measured peak. Inference KV and recurrent-state caches are disabled for these forward/backward runs. Since tensor lifetimes overlap only partially and allocator reservation may exceed live storage, the rows above should not be summed into a predicted peak-memory total.

B Analysis of storage usage

Binary entropy function. All logarithms are base two. The binary entropy function is

$$h_2(\rho) = -\rho \log_2 \rho - (1-\rho)\log_2(1-\rho), \tag{13}$$

with the convention $0 \log_2 0 = 0$ at the endpoints.

Proof of Theorem 2.2. For an abstract entry h, let S indicate whether it is a payload symbol rather than the distinguished zero symbol. Since S is determined by h, the entropy chain rule gives

$$\mathcal{H}(h) = \mathcal{H}(S) + \mathcal{H}(h \mid S) = h_2(\rho) + (1-\rho)(d-1),$$

because $h_2(1-\rho) = h_2(\rho)$, the zero symbol is deterministic and each payload has $d-1$ independent uniform bits. Independence of the entries makes their entropies additive, yielding the result. □

Lemma B.1 (The uniform-bit model is an upper bound)**.** *Let X be an actual activation drawn from an arbitrary distribution over the finite nonnegative values representable in a d-bit format, with $\Pr(X = 0) = \rho$, and let $\mathcal{V}$ denote the set of representable finite positive values. Then*

$$\mathcal{H}(X) \le h_2(\rho) + (1-\rho)\log_2|\mathcal{V}| \le h_2(\rho) + (1-\rho)(d-1). \tag{14}$$

Moreover, for any joint distribution $(X_1, \ldots, X_L)$ whose entries share these marginals,

$$\mathcal{H}(X_1, \ldots, X_L) \le L\big[h_2(\rho) + (1-\rho)(d-1)\big], \tag{15}$$

which is the value in Theorem 2.2.

Proof. Let $S = \mathbb{I}[X > 0]$, so that $\Pr(S = 0) = \rho$ and $\mathcal{H}(S) = h_2(\rho)$. Since S is determined by X, the chain rule gives

$$\mathcal{H}(X) = \mathcal{H}(S) + \mathcal{H}(X \mid S) = h_2(\rho) + (1-\rho)\,\mathcal{H}(X \mid S = 1),$$

where the conditional term retains the factor $1 - \rho$ because X is identically zero when $S = 0$, a deterministic event. Conditional on $S = 1$, the value X lies in $\mathcal{V}$, and the entropy of any distribution supported on a finite set is at most the logarithm of its cardinality, so $\mathcal{H}(X \mid S = 1) \le \log_2|\mathcal{V}|$. A d-bit format admits 2^d bit patterns, exactly 2^{d-1} of which carry a zero sign bit, and every positive value is one of those, so $|\mathcal{V}| \le 2^{d-1}$ and $\log_2|\mathcal{V}| \le d - 1$. This gives Equation (14).

Entropy is subadditive, so

$$\mathcal{H}(X_1, \ldots, X_L) \le \sum_{\ell=1}^{L} \mathcal{H}(X_\ell) \le L\big[h_2(\rho) + (1-\rho)(d-1)\big],$$

which is Equation (15). Subadditivity is an equality if and only if the entries are independent. The per-entry bound using $\log_2|\mathcal{V}|$ is attained when the positive values are uniform over $\mathcal{V}$; the looser bound using $d - 1$ is strict for actual floating-point values. □

Remarks. Lemma B.1 separates the three relaxations that Equation (14) permits. The support need not be full: $|\mathcal{V}|$ may be smaller than 2^{d-1} if some bit patterns are never produced. The positive values need not be uniform. Individual entries need not be independent, and subadditivity charges no penalty for dependence. Each relaxation can only lower the entropy, so the uniform, independent model of Assumption 2.1 is the worst case for compression and therefore a valid reference rate.

The bound is in fact strict. Zero is itself one of the 2^{d-1} patterns with a zero sign bit, so $|\mathcal{V}| \le 2^{d-1} - 1$ and $\log_2|\mathcal{V}| < d - 1$; formats that reserve patterns for infinities or NaN have still fewer finite positive values. The reference entropy in Theorem 2.2 therefore exceeds the actual tensor entropy by at least $L(1-\rho)\big[(d-1) - \log_2|\mathcal{V}|\big]$. This gap is exact when entries are independent and their positive values are uniform over $\mathcal{V}$; nonuniformity or dependence can increase it. The corresponding per-entry gap omits the factor L. We use the reference bound because it depends only on the format width.

Proof of Theorem 4.1. The representation in Section 3.2 stores $s = (1-\rho)L$ positive values, a TB^2-bit mask, and $32T$ bits of prefix metadata. Writing $c = d$ without packing and $c = d - 1$ with packing, its occupied storage is exactly

$$S_{\mathrm{BS}} = cs + TB^2 + 32T + \Delta, \qquad R_{\mathrm{BS}} = \frac{dL}{S_{\mathrm{BS}}}, \tag{16}$$

where Δ includes alignment, guard storage, and per-tensor metadata, including an optional terminal prefix entry. With padding factor $\alpha = TB^2/L$, the cost per logical entry is

$$\frac{S_{\mathrm{BS}}}{L} = c(1-\rho) + \alpha\left(1 + \frac{32}{B^2}\right) + \frac{\Delta}{L}. \tag{17}$$

For negligible padding and per-tensor overhead, $\alpha \simeq 1$ and $\Delta/L \simeq 0$. Substituting $B = 64$ gives a metadata cost of $1 + 32/4096 = 1 + 1/128$ bits per entry. Dividing the dense cost d by the resulting cost per entry, for each choice of c, gives Equation (11). □

C Fused activation–linear layer

C.1 ReLU gradient derivation

Using the notation of Section 2.1, the forward pass is

$$H = \mathrm{ReLU}(Z), \qquad Y = HW^\top. \tag{18}$$

Differentiating the loss gives

$$\frac{\partial \mathcal{L}}{\partial W} = G^\top H, \qquad \frac{\partial \mathcal{L}}{\partial H} = GW, \qquad \frac{\partial \mathcal{L}}{\partial Z} = \frac{\partial \mathcal{L}}{\partial H} \odot \mathrm{ReLU}'(Z) = \frac{\partial \mathcal{L}}{\partial H} \odot \mathbb{I}[Z > 0], \tag{19}$$

where we define $\mathrm{ReLU}'(0) = 0$. The gradients with respect to Z and W depend only on the positive values of H and their positions, which are cached from the forward pass.

C.2 Forward and backward algorithm

Algorithm 1 sets out the forward and backward passes of the fused ReLU–linear autograd function. The forward pass evaluates the activation in place, compresses it, and then performs a dense matrix multiplication, so the forward computation is unchanged from the dense baseline. Only the cached activation is compressed. The backward pass decompresses the activation and computes the weight gradient directly, and applies the activation derivative from the stored bitmask alone, so the dense activation is never reconstructed for the input gradient.

Algorithm 1 BitSparse fused ReLU–linear layer.

Require: preactivation $Z \in \mathbb{R}^{M\times N}$, weight $W \in \mathbb{R}^{D\times N}$
Ensure: output $Y \in \mathbb{R}^{M\times D}$, cache holding the compressed activation

procedure FORWARD(Z, W)
  $H \leftarrow \mathrm{ReLU}(Z)$ ▷ in place
  cache W
  $H_{\mathrm{s}} \leftarrow$ COMPRESS(H) ▷ per-tile bitmask, compact positive values, and prefix offsets; optional sign-bit packing
  $Y \leftarrow HW^\top$ ▷ dense matrix multiplication
end procedure

procedure BACKWARD(G) ▷ $G = \partial\mathcal{L}/\partial Y \in \mathbb{R}^{M\times D}$, the upstream gradient
  retrieve W and H_{s} from the cache
  $\partial W \leftarrow$ SPARSEPRODUCT($G^\top, H_{\mathrm{s}}$) ▷ dense–sparse product against the compressed activation
  if ∂Z is required **then**
    $\partial H \leftarrow GW$ ▷ dense matrix multiplication
    $\partial Z \leftarrow$ MASKWITHBITMASK($\partial H, \mathrm{bitmask}(H_{\mathrm{s}})$) ▷ in place, applying $\mathrm{ReLU}'(Z)$ from the bitmask alone
  else
    $\partial Z \leftarrow \varnothing$
  end if
  release H_{s}
end procedure

The stored values are BF16 or FP8. For FP8 the per-tensor scale is carried in the sparse metadata and applied to the matrix products; sign-bit packing is an independent option that removes the redundant sign bit, as described in Section 3.3. The ReLU^2 case follows the a similar structure, with the stored values squared when reconstruction is required and `MaskWithBitmask` mask replaced with a masked multiplication.

D EXPERIMENTAL SETUP

D.1 SYSTEM SETUP

All experiments were implemented using PyTorch 2.14 with Triton 3.7.1 in Python 3.14. All experiments were run on an NVIDIA RTX 3090 24GB with the exception of the FP8 MLP and Nemotron-H experiments, which were run on an RTX 5090 32GB, for the FP8 support and extra VRAM. CompAct used the official implementation. The nvCOMP baseline uses version 5.3.0.16.

Random seed 0 is used for all experiments. Timings are averaged over 5 runs, after warmup.

D.2 BASELINES

Activation Checkpointing. Activation checkpointing Chen et al. (2016) reduces the amount of memory required by recomputing activations in the backward pass, rather than storing them in the forward pass. This trades off storage for computation, since certain layers will have computations repeated twice in a training step. PyTorch supports activation checkpointing natively. In our method, we checkpoint the full feedforward block, including the RMS norm layer.

Lossless compression. Lossless compression can be used to reduce the size of the saved activations. The activations are compressed before being cached, and decompressed in the backward pass when needed. The nvCOMP NVIDIA Corporation (2026) library provides highly optimized GPU-native codecs for this task, including LZ4 Collet (2011), Zstd Collet & Kucherawy (2021), Bitcomp, and Cascaded. LZ4 uses dictionary matching to replace repeated byte sequences with short references. Zstd similarly exploits repeated patterns, while also using entropy coding to improve compression ratio. Bitcomp is a proprietary format designed for numerical data and can efficiently encode sparse values. Cascaded combines techniques such as run-length encoding, delta encoding, and bit-packing. While these are general lossless compression codecs, they benefit greatly from activation sparsity, since the many zeros are easy to compress.

Compressed Sparse Row. A standard format for sparse matrices is the CSR format. This stores the nonzero values, their column indices and a cumulative counter of row indices, and is well optimized for GPU multiplication, with native support on our GPUs. This method is however suboptimal for this setting, where sparsity is around 0.5 to 0.85, where the overhead from indices is significant. Currently, there is no FP8 support for CSR tensors.

CompAct. Lossy compression has been considered. We compare against a recent method CompAct Shamshoum et al. (2025), which uses random Gaussian matrices to compress the hidden pre-nonlinearity activations into a lower dimension space. This method exploits the low-rank nature of activations. Training using CompAct demonstrates lower memory, with a small increase in training loss due to its lossy nature. The amount of compression can be tuned. We use CompAct-f to denote a compression factor of f, meaning its compression dimension is $1/f$ times the activation. A higher f gives lower memory and faster compression, at the cost of lower accuracy.

D.3 MLP TRAINING BENCHMARK SETUP

Architecture and workload. The comparisons in Section 6.3 use a stack of eight residual MLP blocks. Each block applies RMS normalization, a bias-free linear expansion from 4096 to $21{,}504$ features (the same as Nemotron-H), a ReLU or ReLU^2 activation, and a bias-free projection back to 4096 features before adding the residual connection. Inputs are synthetic standard-normal batches with $16{,}000$ rows and 4096 features. Parameters and activations use BF16.

In this test, parameters are not updated since we are interested in the performance and memory for computing the gradients, not the optimizer. Each lossless method is validated by checking that its gradients match the dense baseline up to floating-point rounding.

FP8 activation storage and matrix multiplication For the FP8 experiment we use FP8 only for the matrix multiplications and for the activation cache; the model's parameters, inputs, and gradients remain in BF16. In each product the BF16 operand is cast to FP8 on the fly, so both the activation and the weight are represented in FP8-E4M3 for the GEMM, which accumulates in FP32 and returns a BF16 result. The activation cache stores the FP8 values themselves, together with the scale needed to interpret them. The same FP8 scaling is used as in Nemotron-H, with a tensor-wise scale γ:

$$\gamma = \max\left(\frac{\max_{i,j}|H_{ij}|}{q_{\max}}, 10^{-9}\right), \qquad H_{\mathrm{FP8}} = \mathrm{cast}_{\mathrm{FP8}}\left(\frac{H}{\gamma}\right), \tag{20}$$

where $q_{\max}$ is the largest finite value of the selected FP8 format, so γ places the largest magnitude at the top of the FP8 range. We use a separate tensor-wise scale for each operand, the weight and the activation. For the ReLU^2 path, conversion is applied to the saved $\mathrm{ReLU}(Z)$ values. The implementation currently converts tensors to E4M3 FP8. The activation scale is stored alongside the sparse cache so that the values can be interpreted and scaled back during reconstruction. The weight scale is transient, recomputed each time the weight is cast, so keeping weights in BF16 and casting them per product adds no persistent parameter storage. Dense FP8 products use PyTorch's scaled matrix multiplication primitive with tensor-wise scales. This scaling ensures the full dynamic range of FP8 values is used. Our method is also compatible with more advanced formats such as MXFP8 which use blockwise multiplication Open Compute Project (2023).

Setup details. All methods use the same weights and inputs, fixed using a random seed. In order to generate sparse activations, the weights of the networks are shifted until the activations have the desired sparsity level. Every method uses a custom backward implementation which fuses the RMS norm layer via `torch.autograd.Function`. This is to ensure maximum efficiency since we found the default PyTorch implementation to be inefficient in its activation caching. In particular, fusing the RMS norm layer reduced the memory consumption of every method.

Timing and memory. Each configuration is measured five times. Each measurement excludes one warmup forward/backward pass and is followed by three timed steps that are averaged for the displayed time. GPU execution is synchronized at the timing boundaries. We record peak GPU memory allocated during the final measured pass, including model parameters, activations, and temporary tensors, rather than only the compressed activation footprint.

D.4 Language-model memory and runtime setup

Models and data. We evaluate two models. The first is the 162M-parameter Modded-NanoGPT checkpoint from Section 6.1, trained by us according to the Track 3 baseline and evaluated at step 3300 (12 layers, 768 hidden dimensions, 50,304-token vocabulary). The second is NVIDIA's official pretrained Nemotron-H-8B-Base-8K model. Both run in BF16 on a single GPU: an RTX 3090 with 24 GiB for Modded-NanoGPT and an RTX 5090 with 32 GiB for Nemotron-H. Models are evaluated on text from the FineWeb10B validation dataset Penedo et al. (2024), truncated to the target length.

Batch size. Modded-NanoGPT is evaluated with a fixed batch of 4 sequences, sweeping the sequence length from 256 to 16,000 tokens, so the number of tokens per step ranges from 1,024 to 64,000. Nemotron-H is evaluated one sequence at a time at each length.

Configurations. Each model is measured with dense activation storage, activation checkpointing on every feed-forward block, BitSparse, and BitSparse with sign-bit packing. Both models use ReLU^2 feed-forward activations. BitSparse compresses each activation at the sparsity the model exhibits naturally (Section 6.1). We modify both models to use the same fused RMS normalization, and chunk the Nemotron-H vocabulary projection head, to keep incidental memory overhead low (Appendix A.1).

Workload. Each step performs a forward pass and the backward pass of the language-model loss over that step's tokens. Model parameters are never updated and gradients are released as they are produced, so no full-model parameter-gradient buffer or optimizer state is retained. Transient parameter gradients still contribute to peak memory.

Timing and memory. The timing and peak memory consumption in a training step are measured. Times are averaged per step over repeated iterations after warmup. Memory is tracked using `torch.cuda.max_memory_allocated("cuda")`. As in Appendix D.3, peak memory covers model parameters, activations, and temporary tensors. Warmup steps are used to ensure accurate timing.

E Additional results

E.1 Activation sparsity measurement in LLMs

The experiments in Section 6.1 characterize feed-forward activation sparsity during Modded-NanoGPT training and in a pretrained Nemotron-H model. We evaluate both models on contiguous sequences from the FineWeb10B validation set, with fixed weights and without BitSparse compression. Both experiments use 256 evaluation batches, but the batch composition differs: the Modded-NanoGPT runs place four 4096-token sequences in each batch, while the Nemotron-H runs process a single sequence per batch.

Measurement and aggregation. Following Section 2.2, we report activation sparsity ρ; larger values indicate a greater fraction of zeros. For each layer ℓ and evaluation batch j, we measure $\rho_{\ell,j}$ over all tokens and feature dimensions of the ReLU^2 output, counting exact zeros without a magnitude threshold. With Q evaluation batches and measured layers $\mathcal{I}$, we compute

$$\overline{\rho}_\ell = \frac{1}{Q}\sum_{j=1}^{Q}\rho_{\ell,j}, \qquad \rho_{\ell,\min} = \min_{1\le j\le Q}\rho_{\ell,j}, \tag{21}$$

$$\rho_{\mathrm{batch,min}} = \min_{1\le j\le Q}\frac{1}{|\mathcal{I}|}\sum_{\ell\in\mathcal{I}}\rho_{\ell,j}. \tag{22}$$

These are the per-layer mean sparsity, the per-layer minimum sparsity, and the minimum batch-wise mean sparsity across layers, respectively. The last quantity captures the densest observed batch averaged over layers; individual layers may reach their minima on different batches. Layers receive equal weight. All minima refer to the evaluated batches, rather than bounds on unseen inputs. Measurements are taken after squaring, so finite-precision underflow may produce additional zeros relative to the unsquared activations saved by BitSparse.

Modded-NanoGPT during training. We evaluate all 12 feed-forward layers at training steps $0, 300, \ldots, 3300$. Each checkpoint is evaluated on the same 256 batches, each containing four sequences of 4096 tokens, for a total of $1{,}024$ sequences and $4{,}194{,}304$ input-token positions. Four preliminary warmup batches are excluded. The model uses BF16 activations. We report each layer's mean and minimum activation sparsity to show how sparsity develops during training and how much it varies across batches. In the training-evolution plot, solid and dashed lines denote the mean and minimum, respectively, with a shared color for each layer.

Pretrained Nemotron-H. We evaluate the pretrained Nemotron-H model in BF16. Measurements cover all feed-forward MLP blocks in the hybrid architecture, excluding attention and Mamba blocks; layer indices refer to positions in the full model. Evaluation comprises 256 sequences of 4096 tokens, processed one sequence per batch, for $1{,}048{,}576$ input-token positions. One warmup pass is excluded from the statistics. We report per-layer mean and minimum sparsities, together with the mean sparsity across layers, the smallest individual layer–batch sparsity, and the minimum batch-wise mean sparsity defined above.

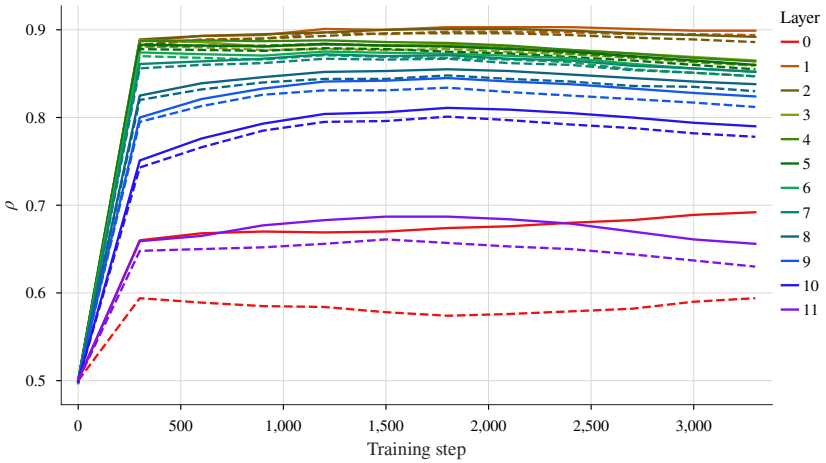


Figure 5: Activation sparsity during Modded-NanoGPT training for all 12 feed-forward layers. Solid lines show per-layer mean sparsity and dashed lines show per-layer minimum sparsity over 256 evaluation batches. The vertical axis reports the zero fraction ρ, so higher values indicate greater sparsity.

Table 6: Activation sparsity ρ for selected Nemotron-H-8B-Base-8K layers over 256 validation batches. Per-layer minima correspond to each layer's densest observed batch. The overall row reports mean sparsity across all measured layers and batches, and the minimum batch-wise mean sparsity across all measured layers. Higher values indicate greater sparsity.

Layer	Mean sparsity	Minimum sparsity
1	0.9718	0.9673
3	0.9335	0.9235
10	0.9590	0.9543
27	0.8649	0.8408
32	0.9160	0.9023
47	0.9616	0.9383
51	0.9357	0.8789
Overall	0.9273	0.9176

E.2 Full training comparison

E.3 MLP benchmark results

Table 8 reports the ReLU MLP benchmark at sparsity $\rho = 0.5$ and 0.9 for both BF16 and FP8. Table 3 gives the corresponding ReLU^2 results discussed in the main text.

Batch-size variation. Tables 9, 10, and 11 complete the batch-size sweep of Table 4 for its three remaining activation and sparsity settings: ReLU^2 at $\rho = 0.5$, and ReLU at $\rho = 0.5$ and 0.9. The saving scales with sparsity and with the size of the dense baseline, not with the activation alone. At $32{,}000$ rows BitSparse reduces peak memory by 36% for ReLU^2 at $\rho = 0.5$, 26% for ReLU at the same sparsity, and 50% for ReLU at $\rho = 0.9$. The larger reduction for ReLU^2 comes from the reduced transient storage used to compute activations in the backward pass from our optimized fused Triton kernel. The crossover at which sign-bit packing becomes the smallest configuration depends on both factors: it occurs at $8{,}000$ rows for both activation functions at $\rho = 0.9$, and not at all at $\rho = 0.5$. Runtime stays within about 1.5% of dense storage at $32{,}000$ rows in all three settings.

Table 7: Memory usage and time per training step for Modded-NanoGPT at two microbatch sizes. OOM denotes an out-of-memory failure.

Method	Microbatch size	Memory / MiB	Time / s
Normal	32	15229.6	7.428
Normal	64	OOM	OOM
BitSparse	32	13174.7	7.668
BitSparse	64	21822.1	7.416

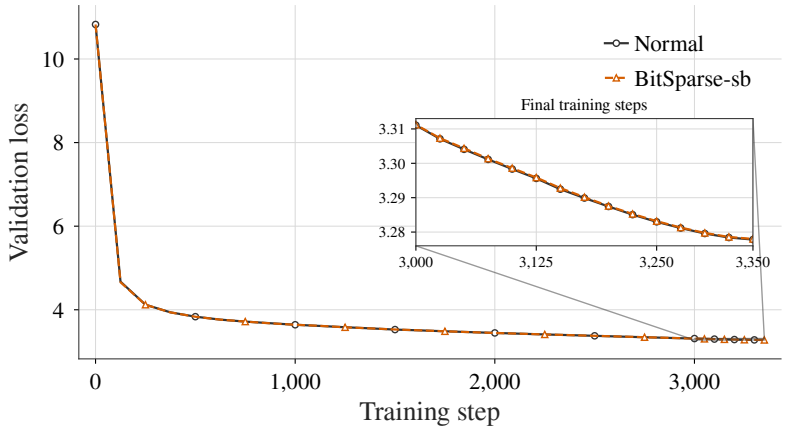


Figure 6: Validation loss during standard (Normal) and BitSparse-sb training of Modded-NanoGPT. The inset zooms in on steps 3,000–3,350.

E.4 Codec implementation ablation

Table 12 compares encoding and decoding with a pure PyTorch implementation of BitSparse and our custom Triton kernels. Both versions do the same operation, producing bit-identical outputs. The PyTorch version is optimized extensively using our LLM agent to ensure a fair comparison. Both operate on a $65{,}536 \times 8{,}192$ BF16 tensor (1 GiB in dense form). We report peak allocated GPU memory and average runtime for each operation.

E.5 Activation backward ablation

Table 13 isolates the pointwise activation derivative. The BF16 input shares the size of our MLP tests (16,000 rows and 21,504 features) and has around 50% sparsity. The PyTorch baseline is optimized to use fused inplace operations whenever possible for fast speed and as low a memory overhead as possible. Meanwhile, BitSparse fuses reconstruction of the required values and the derivative into an in-place update, using compact BF16 values. Peak allocated memory includes the gradient, retained activation representation, and any backward temporaries. Results are averaged over 32 iterations after 5 warmup iterations.

E.6 Shared value buffer ablation

Table 14 compares BitSparse with and without the shared value buffer described in Section 3.5. The comparison uses the same eight-block MLP setup as the main benchmark (Appendix D.3), with ReLU^2 activations at sparsity $\rho = 0.5$, and varies the input size from 32 to 1024 tokens. The buffer reduces runtime most at the smallest input size, from 12.563 to 11.904 ms at 32 tokens. At 256–1024 tokens, the runtime differences are below 0.4% in magnitude, with a small slowdown at 256 tokens rather than a consistent improvement.

Table 8: Peak allocated memory and time per step for the ReLU MLP benchmark, using BF16 and FP8 activations at sparsity $\rho = 0.5$ and 0.9. CompAct is a lossy reference. Bold and italic values indicate the lowest and second-lowest lossless results. Dashes indicate unavailable results.

	Memory / MiB				Time / ms			
	BF16		FP8		BF16		FP8	
Method	0.5	0.9	0.5	0.9	0.5	0.9	0.5	0.9
Dense	11046	10039	8500	8500	**1748**	**1748**	**323**	333
BitSparse	8800	*5742*	7165	6495	*1794*	*1776*	*324*	**328**
BitSparse-sb	*8631*	**5715**	*7030*	*6443*	1799	1786	330	*331*
Checkpoint	**6928**	6057	**6105**	**6105**	2047	2047	441	452
LZ4	10721	7324	8428	7700	3334	2384	644	544
Zstd	9707	6942	7727	7188	4702	3046	934	691
Cascaded	11827	7533	8825	8281	1910	1943	424	411
Bitcomp	11946	7886	8534	8153	1855	1919	414	404
CSR	–	15984	–	–	–	2014	–	–
CompAct-2	11486	11486	10078	10078	2653	2653	485	485
CompAct-4	9955	9955	9238	9238	1976	1976	374	374
CompAct-8	9190	9189	8854	8854	1636	1636	324	324

Table 9: Effect of batch size on peak allocated memory and time per step for the ReLU^2 MLP at sparsity $\rho = 0.5$. Bold and italic values indicate the lowest and second-lowest results, respectively.

Batch	Memory / MiB				Time / ms			
size	Dense	Checkpoint	BitSparse	BitSparse-sb	Dense	Checkpoint	BitSparse	BitSparse-sb
32	3054	**2879**	2883	*2882*	**13**	14	14	*13*
128	3094	**2899**	2912	*2911*	**19**	22	*19*	20
512	3264	**2982**	3033	*3028*	*64*	75	**64**	67
2000	3932	**3294**	3492	*3471*	*242*	283	**241**	244
8000	7107	**4704**	5337	*5253*	**925**	1088	*927*	938
32000	19811	**10704**	12729	*12394*	*3636*	4270	**3618**	3660

E.7 Layer-count trade-offs

Figures 7, 8, and 9 give the three remaining activation and sparsity settings for the layer-count sweep of Figure 2.

Whenever at least one layer is compressed or checkpointed, BitSparse completes the step faster than checkpointing, because checkpointing pays for recomputation while BitSparse adds only a codec pass. Checkpointing nevertheless attains the lowest VRAM in the two $\rho = 0.5$ settings, and the margin there is large: for ReLU at 8 layers, BitSparse uses 29% more memory than checkpointing at $\rho = 0.5$, but 5% less at $\rho = 0.9$. Both $\rho = 0.9$ settings therefore cross below checkpointing at the largest layer count.

The steepest fall in VRAM occurs at the final layer count, and it is far sharper for ReLU^2 than for ReLU: moving from 7 to 8 compressed layers lowers BitSparse's VRAM by 23% at $\rho = 0.9$ and 15% at $\rho = 0.5$ for ReLU^2, but by only 7% and 1% for ReLU. Sign-bit packing tracks BitSparse closely in every setting, saving slightly more memory at a small cost in time.

Table 10: Effect of batch size on peak allocated memory and time per step for the ReLU MLP at sparsity $\rho = 0.5$. Bold and italic values indicate the lowest and second-lowest results, respectively, including ties.

Batch	Memory / MiB				Time / ms			
size	Dense	Checkpoint	BitSparse	BitSparse-sb	Dense	Checkpoint	BitSparse	BitSparse-sb
32	3054	**2878**	2883	*2882*	**12**	15	13	*13*
128	3094	**2894**	2912	*2911*	**18**	21	*19*	19
512	3264	**2960**	3033	*3028*	**62**	73	*64*	65
2000	3889	**3212**	3490	*3470*	**234**	273	*236*	239
8000	6451	**4376**	5334	*5251*	**887**	1037	*905*	912
32000	17186	**9392**	12717	*12383*	**3478**	4069	*3530*	3571

Table 11: Effect of batch size on peak allocated memory and time per step for the ReLU MLP at sparsity $\rho = 0.9$. Bold and italic values indicate the lowest and second-lowest results, respectively.

Batch	Memory / MiB				Time / ms			
size	Dense	Checkpoint	BitSparse	BitSparse-sb	Dense	Checkpoint	BitSparse	BitSparse-sb
32	3054	**2878**	2879	**2878**	**12**	13	*12*	12
128	3094	**2894**	*2895*	*2895*	**18**	21	*19*	19
512	3264	**2960**	2966	*2965*	**61**	72	*63*	63
2000	3889	**3212**	3233	*3230*	**232**	271	*235*	237
8000	6451	4376	*4307*	**4293**	**893**	1040	*904*	908
32000	17186	9392	*8607*	**8555**	**3482**	4075	*3527*	3546

Table 12: BitSparse codec implementation ablation on a $65{,}536 \times 8{,}192$ BF16 tensor. The baseline implements BitSparse in pure PyTorch; the proposed implementation uses custom Triton kernels. Lower peak memory and runtime are better.

Operation	Implementation	Peak memory / MiB	Time / ms
Encode	Pure PyTorch	2560.00	28.010
	Triton (BitSparse)	**1600.50**	**3.690**
Decode	Pure PyTorch	2048.53	17.168
	Triton (BitSparse)	**1600.50**	**2.170**

Table 13: Activation backward ablation on a $16{,}000 \times 21{,}504$ BF16 tensor with approximately 50% sparsity. Peak memory includes the saved representation, gradient, and backward temporaries; time covers only the activation-derivative computation. Lower is better.

Activation	Method	Peak memory / MiB	Time / ms
ReLU	PyTorch	1312.50	2.467
	BitSparse	**698.25**	**2.043**
ReLU^2	PyTorch	2625.00	4.200
	BitSparse	**1025.70**	**2.552**

Table 14: Effect of the shared value buffer on BitSparse runtime as input size increases. Times retain three decimal places to show the small differences between configurations. Lower is better.

Tokens	Without buffer / ms	With buffer / ms
32	12.563	11.904
64	17.622	17.136
128	20.150	19.954
256	37.973	38.097
512	66.539	66.308
1024	130.807	130.518

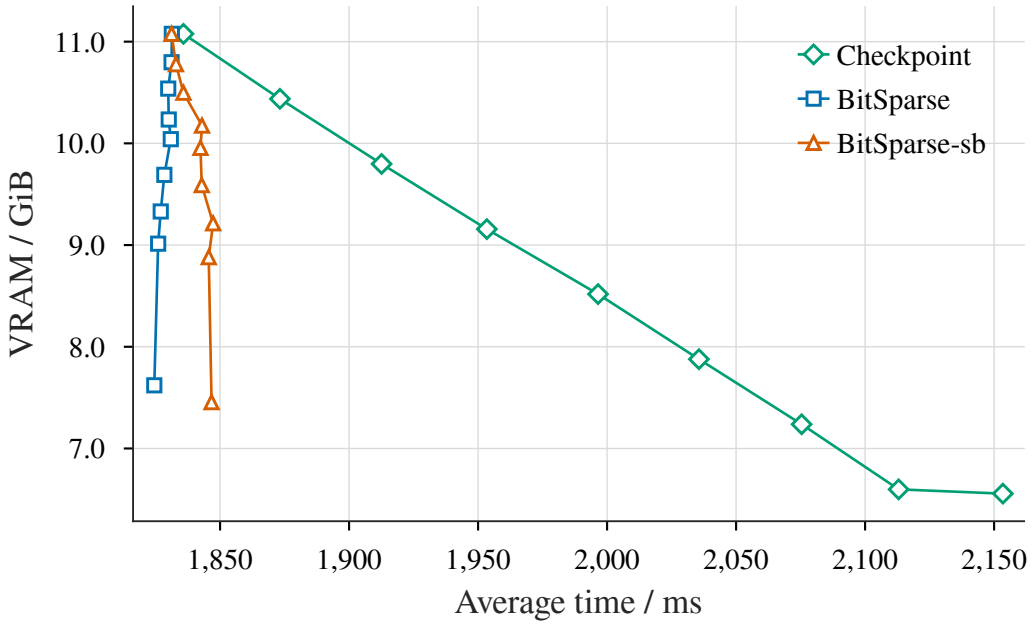


Figure 7: Memory–runtime trade-off for the ReLU^2 MLP at sparsity $\rho = 0.5$. Conventions follow Figure 2.

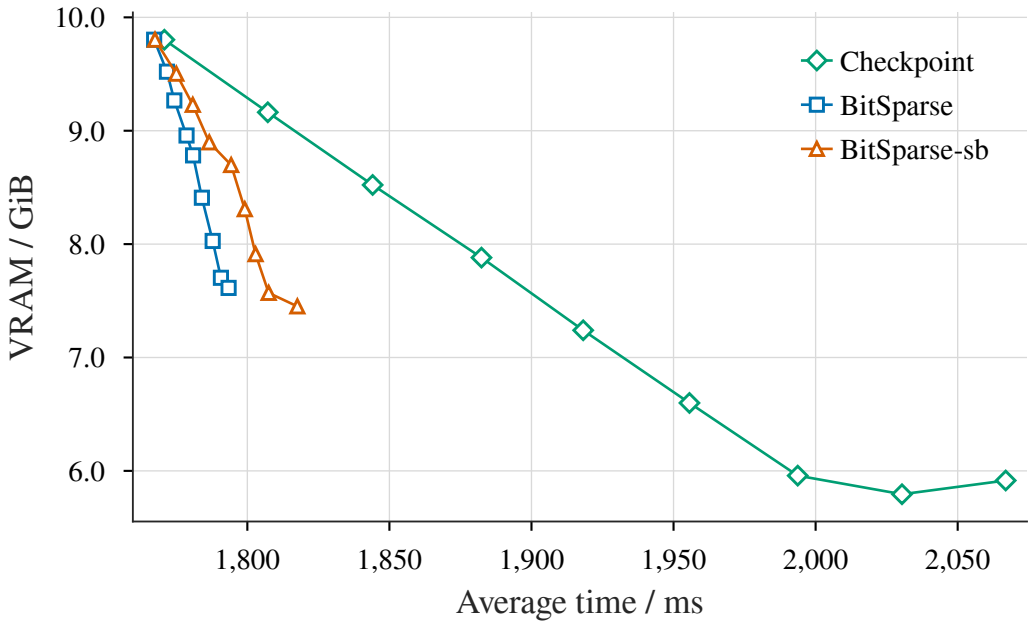


Figure 8: Memory–runtime trade-off for the ReLU MLP at sparsity $\rho = 0.5$. Conventions follow Figure 2.

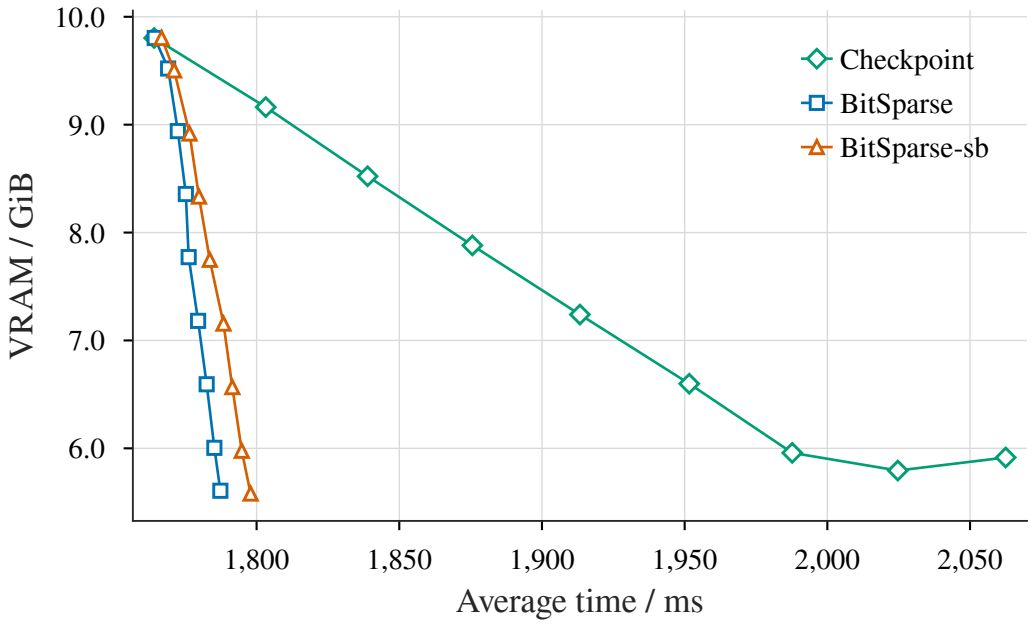


Figure 9: Memory–runtime trade-off for the ReLU MLP at sparsity $\rho = 0.9$. Conventions follow Figure 2.