COMPRESSED SPARSE TILES FOR MEMORY-EFFICIENT SPARSITY

Mike Lasby[†1], **Max Zimmer**[2], **Sebastian Pokutta**[2], **Erik Schultheis**[†3,4]
[1]University of Calgary, [2]TU Berlin & Zuse Institute Berlin, [3]Aalto University, [3]IST Austria

ABSTRACT

Storing the weights of Large Language Models (LLMs) in GPU memory for local inference is challenging due to their size. While quantization has proven successful in reducing the memory footprint of LLMs, unstructured pruning introduces overhead by requiring the non-pruned weights' location to be encoded. This overhead hinders the efficient combination of quantization and unstructured pruning, especially for smaller batch sizes common in inference scenarios. To address this, we propose the CS256 storage format, which offers a better balance between space efficiency and hardware acceleration compared to existing formats. CS256 partitions the weight matrix into tiles and uses a hierarchical indexing scheme to locate non-zero values, reducing the overhead associated with storing sparsity patterns. Our preliminary results with one-shot pruning of LLMs show that CS256 matches the performance of unstructured sparsity while being more hardware-friendly. Our code is available at: https://github.com/mklasby/llm-compressor/tree/mklasby-cs256.

1 INTRODUCTION

In recent years, machine learning models have seen an unprecedented growth in size, far outstripping the increase in compute power predicted by Moore's law. Thus, training these foundation models requires increasingly longer durations and a growing number of accelerators in datacenters (Narayanan et al., 2021). While inference is significantly less resource intensive than training (e.g., except for a KV cache, no activations need to be persisted), even just storing the weights of such a model in GPU memory can be infeasible outside of datacenter deployments. For instance, LLama3 (Dubey et al., 2024) requires $\approx 754\text{GiB}$ in 16-bit precision for its 405 billion weights, equivalent to the capacity of 16 A6000 workstation GPUs. Consequently, model compression has become essential, converting the gigantic foundation models into smaller, locally runnable versions. Interestingly, a compressed version of a large model typically outperforms a memory-matched, uncompressed smaller model (Li et al., 2020).

Model compression can be broadly classified into two approaches: Storing less bits per weight (quantization, e.g. Dettmers et al. (2024); Frantar & Alistarh (2023)), and storing fewer weights at all (pruning, e.g. Han et al. (2015); Hoefler et al. (2021)). Quantization has had resounding success with LLMs: reducing the bit-width to eight bits per weight can be achieved without noticeable quality loss (Dettmers et al., 2022), and even reducing to four bits is possible with minimal degradation (Frantar et al., 2022). Recent methods are even approaching the 1-bit boundary (Ma et al., 2024), although such extreme quantization does result in a significant drop in generation quality. In fact, quantization has been so successful that newer hardware generations have support for smaller and smaller data types built directly in silicone, with 8-bit floating point support in H100 (NVIDIA, 2022) and 4-bit types in B100 (NVIDIA, 2025).

However, a fundamental drawback of pruning is that, in order to implement it efficiently, one would like the pruned elements to be as structured as possible; pruning of entire layers (Gromov et al., 2024) or neurons can be implemented almost without overhead, but also quickly deteriorates the models performance. On the other hand, unstructured pruning generally allows for removing a larger fraction of weights, but this is paid for in two ways. First, unstructured calculations can only use a small fraction of a GPUs theoretical maximum FLOPs (Gale et al., 2020), and second, one needs to encode the location of the non-pruned weights, thus introducing memory overhead. Nonetheless, some limited form of sparsity support has also

Correspondence to mklasby@ucalgary.ca and erik.schultheis@aalto.fi
† Equal contribution.

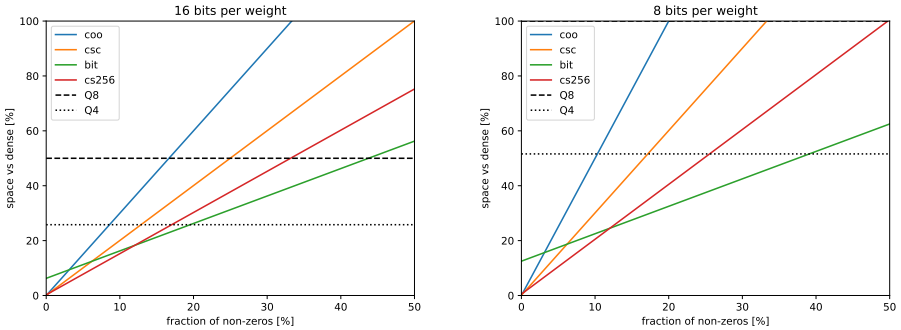


Figure 1: Compression ratio of several sparse storage at 16 and 8 bit weight precision, compared with 8 and 4 bit quantization (assuming effectively 4.127 bits per parameter (Dettmers et al., 2024)).

found its way into recent hardware in the form of 2:4 sparsity (Mishra et al., 2021). Due to the strong structural constraint, 2:4 sparsity has an overhead of only two bits per nonzero weight, but at the same time, also does not allow for pruning more than 50% of the weights.

In this paper, we take the stance that the second problem is much more problematic, because at small batch sizes, such as during inference, the computations will be memory-bandwidth bound instead of compute-bound. Even more importantly, there is a steep performance cliff between being able to fit a model fully on device memory, and having to stream its weights through the slow PCIe connection. This makes the overhead of sparse data formats the main obstacle, from a computational perspective, to the adoption of sparsity at a similar rate as quantization. To make matters worse, metadata overhead is independent of the weights' precision, preventing efficient combination of quantization with unstructured sparsity.

Figure 1 illustrates the storage overhead for different sparsity formats. Traditional sparsity formats such as coordinate (COO) or compressed-sparse row (CSR) formats, which are natively supported by PyTorch, come with significant overhead. To achieve a 50% reduction in memory, sparsities of 83.5% (90% for 8-bit weights) and 75% (83%) are required, beyond what current pruning algorithms can provide with acceptable loss in model quality. On the other hand, bitmasks are highly efficient at the intermediate sparsity regime most interesting to ML research, yet they require sequential decoding, anathema to current parallel hardware.

One way of achieving a sparsity format that is both space-efficient and amenable to hardware acceleration is by imposing constraints on the supported sparsity patterns (Gray et al., 2017; Castro et al., 2023; Yu et al., 2024; Okanovic et al., 2024; Lin et al., 2023). Most famously, 2:4 sparsity is natively supported in NVidia hardware since Ampere. Not only does this format lead to a true speedup of almost $2\times$ (when doing matrix-matrix multiplications using tensor cores), it also requires only 4 bits of indexing for each block of 4 weights, making it as memory-efficient as a bitmask representation. Another alternative are block-sparse formats, where a single piece of indexing information pertains to an entire block of weights.

Below, we propose the CS256 storage format, which offers a better balance between space efficiency and hardware acceleration compared to existing formats: it is (essentially) unstructured, but with less overhead than CSC, and better parallelizability than bitmasks.

2 Compressed Sparse Tiles

In COO format, each weight has its location identified by explicitly specifying its row and column, requiring two sufficiently large (typically 16-bit) integers, so that a single sparse weight is three times as expensive as a single weight in a dense matrix in 16-bit precision. The overhead is reduced in CSR format by observing that, for non-extreme sparsity levels, many weights will share their row index. By storing just the the column index, and an additional data structure that points to the beginning of each row, the overhead can be almost halved. The key improvement in CSR is to perform the localization of weights hierarchically: First, group weights together by coarse location (i.e., row), then locate them within (i.e., column). However, the coarse location is still large enough that the inner locator needs to have 16 bits.[1]

[1] Of course, for smaller matrices, one might get away with fewer bits, i.e., 12 bits would be enough for 4096 columns, but non-multiples of 8 make implementations much more difficult.

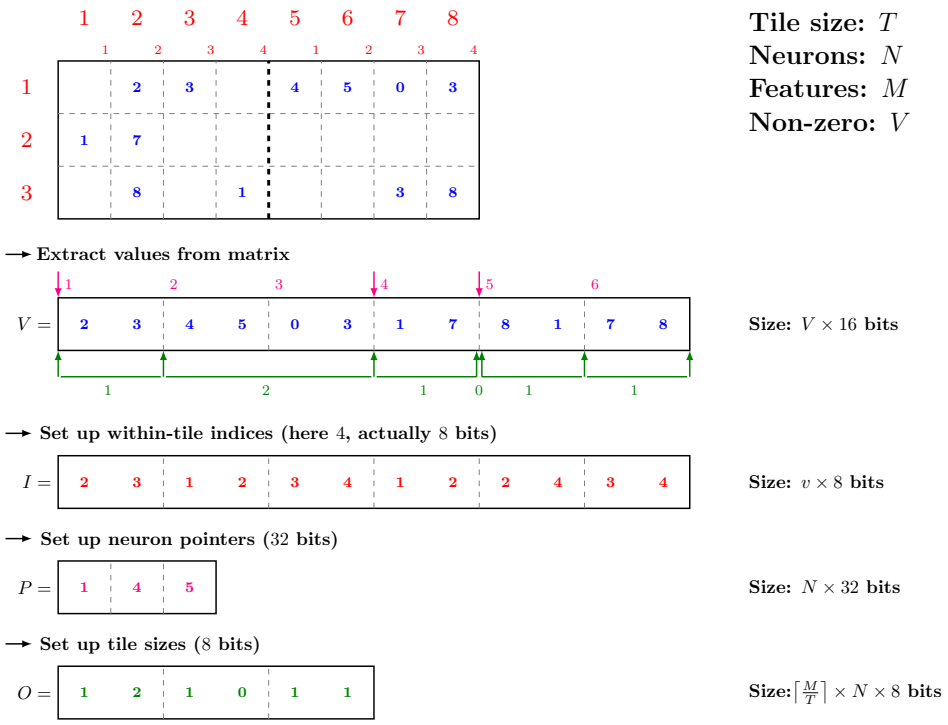


Figure 2: Visualization of the CS256 format, using a smaller tile size of 4 for illustration.

In order to reduce the memory overhead further, one needs to reduce the size of the regions such that 8 bits are sufficient to locate a weight. Thus, a straightforward improvement over CSR is to partition the matrix into rectangular tiles with an area of 256 entries, replace the pointers to row starts with pointers to tile starts, and use 8-bit indices to identify weights within a tile. Of course, we have now traded the reduced number of bits per weight with a larger number of pointers to tiles; a favorable trade at medium sparsities, but inefficient for extremely sparse matrices. This problem becomes even more apparent if we were to attempt and reduce the bits per weight further. Going down to only 2 bits per weight, we require tiles of size 4; but then, an $n \times n$ matrix would require $n^2/4$ tile pointers, which would consume a huge amount of memory in themselves.

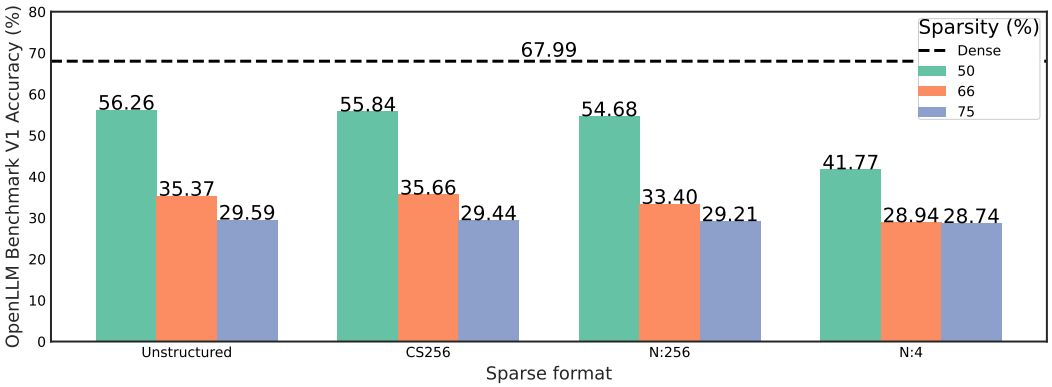
One way to address this is to make tiles uniform; if each tile has exactly the same number of non-zero entries, there is no need to store any tile pointers at all. For n entries per tile, the kth weight belong into tile $\lfloor k/n \rfloor$. For tiles sizes of four and $n=2$, this results in the hardware-accelerated $2:4$ sparsity format. However, the space savings are achieved by placing a strong structural constraint on the sparsity pattern.

In order to reduce the pointer overhead without introducing additional constraints, we can introduce another level in the location hierarchy: Group tiles together into larger super-tiles, have one 32-bit pointer per such super-tile, and then user smaller-sized offsets to identify the location of the tile inside the larger area.

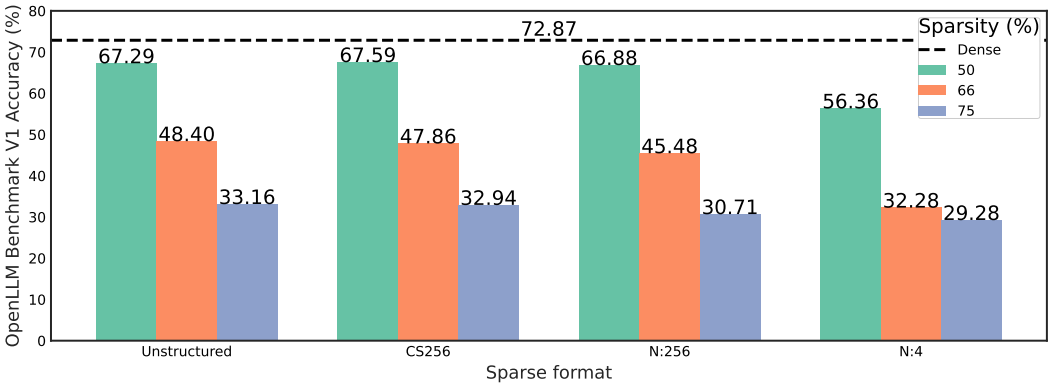
Let us focus, for now, on tiles of shape 1×256, that is, a tile spans 256 columns in one row. As there cannot be more than 256 non-zeros per tile, the size of the tile can be saved in 8 bit. Saving tile sizes, as opposed to offsets to their start, has the disadvantage that a prefix-sum needs to be computed in order to identify each particular tile. However, a typical matrix multiplication algorithm will iterate over the k dimension (i.e., the weights for one particular neuron) within one compute unit[2], so that the tile offset can be calculated without overhead.

This leaves us with the following storage format, illustrated in Figure 2. Weight values are stored in one continuous array `Values` (V in the figure). A second array `RowPointers` (P) of 32-bit indices indicates the start of each row within the weight array. For each tile of 256 columns within one row, its size is saved in an 8-bit value `BlockIncrements` (O). Finally, for each structural non-zero, one 8-bit index within the 256-wide tile is stores in `Indices` (I). While this raw definition of the format imposes no structural requirements on the distribution of non-zero elements, in order to enable efficient implementation,

[2]with the exception of split-k algorithms



(a) **Llama-3.1-8B-Instruct**



(b) **Qwen-2.5-7B-Instruct**

Figure 3: **Average accuracy on OpenLLM Leaderboard V1** benchmarks for various sparsities and formats for Llama-3.1-8B-Instruct (fig. 3a) and Qwen-2.5-7B-Instruct (fig. 3b). CS256 matches unstructured sparsity. For N:M, $M = 256$ yields considerable improvements over the 1:4, 2:6, and 2:4 formats.

we want to ensure data alignment for vectorized memory access. This is achieved by requiring that the number of non-zeros within each tile is a multiple of 4/8/16 (but can be different in different tiles).[3]

3 Experimental Results

We extend SparseGPT (Frantar & Alistarh, 2023) such that the pruned weights conform to the CS256 format, that is, we prune such that the number of active weights in each tile is a multiple of 8. We prune Llama-3.1-8B-Instruct (Llama Team, AI @ Meta, 2024) with a block size of 256 and dampening fraction of 0.01 to unstructured, CS256, and two N:M formats – $M \approx 4$ and $M = 256$ – with 50%, 66%, and 75% sparsity. For calibration data we randomly select 512 samples from UltraChat-200K[4] (Ding et al., 2023) truncated to a maximum sequence length of 2048. We evaluate the pruned models on OpenLLM Leaderboard V1 tasks using the EleutherAI evaluation harness (Gao et al., 2023) in Figure 3. These tasks include: 25-shot ARC-C (Clark et al., 2018), 5-shot strict exact match GSM8k (Cobbe et al., 2021), 10-shot HellaSwag (Zellers et al., 2019), 5-shot MMLU (Hendrycks et al., 2021), 5-shot Winogrande (Sakaguchi et al., 2019), and 0-shot multi-true (MC2) TruthfulQA (Lin et al., 2022). We report byte-length normalized accuracies for ARC-C and HellaSwag.

Having little structural constraints, we find that CS256 matches the performance of unstructured sparsity, outperforming the more structured formats. We posit that these improvements may be attributed to the non-uniform allocation of parameters across rows of the weight matrix. These preliminary results suggest

[3]One could always fill up the additional elements with explicit zeros, but that does not make sense for a ML application. If we have to pay the memory cost, we may as well put these weights to good use.

[4]https://huggingface.co/datasets/HuggingFaceH4/ultrachat_200k

that CS256 may offer generalization performance comparable to unstructured sparsity while providing practical benefits on commodity hardware.

Testing on a 16 GiB RTX A4000 card, running the default (bfloat16) implementation of `transformers`, a few of the model layers end up getting offloaded to CPU, and text generation, using a batch size of 1, proceeds at a speed of 0.41 sec/tok. Converting to 66% sparse CS256 format and using our custom kernel for `spmv` (see A.1), the model fits comfortably in device memory (see A.2), and generation speed increases to 0.21 sec/tok.

4 Conclusions and Outlook

This work introduces the CS256 format, a novel approach to LLM compression that offers flexibility and efficiency. Our analysis shows that CS256 enables significant memory reduction without compromising accuracy, making it suitable for deploying large models on resource-constrained devices. While further empirical validation is necessary, our preliminary findings suggest that CS256 is a promising addition to the LLM compression toolkit, opening possibilities for wider deployment and accessibility of powerful language models.

As Figure 1 suggests, the CS256 format would still retain noticable compression at 66% sparsity when using 8-bit weights, a combination which we intend to explore in future work.

REFERENCES

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 *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–14, 2023.

Peter Clark, Isaac Cowhey, Oren Etzioni, Tushar Khot, Ashish Sabharwal, Carissa Schoenick, and Oyvind Tafjord. Think you have solved question answering? try arc, the ai2 reasoning challenge, 2018. URL https://arxiv.org/abs/1803.05457.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems, 2021. URL https://arxiv.org/abs/2110.14168.

Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. Gpt3. int8 (): 8-bit matrix multiplication for transformers at scale. *Advances in Neural Information Processing Systems*, 35: 30318–30332, 2022.

Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. Qlora: Efficient finetuning of quantized llms. *Advances in Neural Information Processing Systems*, 36, 2024.

Ning Ding, Yulin Chen, Bokai Xu, Yujia Qin, Zhi Zheng, Shengding Hu, Zhiyuan Liu, Maosong Sun, and Bowen Zhou. Enhancing chat language models by scaling high-quality instructional conversations, 2023.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

Elias Frantar and Dan Alistarh. SparseGPT: Massive Language Models Can Be Accurately Pruned in One-Shot, March 2023. URL http://arxiv.org/abs/2301.00774. arXiv:2301.00774 [cs].

Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. Gptq: Accurate post-training quantization for generative pre-trained transformers. *arXiv preprint arXiv:2210.17323*, 2022.

Trevor Gale, Matei Zaharia, Cliff Young, and Erich Elsen. Sparse gpu kernels for deep learning. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–14. IEEE, 2020.

Leo Gao, Jonathan Tow, Baber Abbasi, Stella Biderman, Sid Black, Anthony DiPofi, Charles Foster, Laurence Golding, Jeffrey Hsu, Alain Le Noac'h, Haonan Li, Kyle McDonell, Niklas Muennighoff, Chris Ociepa, Jason Phang, Laria Reynolds, Hailey Schoelkopf, Aviya Skowron, Lintang Sutawika, Eric Tang, Anish Thite, Ben Wang, Kevin Wang, and Andy Zou. A framework for few-shot language model evaluation, 12 2023. URL https://zenodo.org/records/10256836.

Scott Gray, Alec Radford, and Diederik P Kingma. Gpu kernels for block-sparse weights. *arXiv preprint arXiv:1711.09224*, 3(2):2, 2017.

Andrey Gromov, Kushal Tirumala, Hassan Shapourian, Paolo Glorioso, and Daniel A Roberts. The unreasonable ineffectiveness of the deeper layers. *arXiv preprint arXiv:2403.17887*, 2024.

Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural networks. In C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett (eds.), *Advances in Neural Information Processing Systems*, volume 28. Curran Associates, Inc., 2015. URL https://proceedings.neurips.cc/paper/2015/file/ae0eb3eed39d2bcef4622b2499a05fe6-Paper.pdf.

Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. Measuring massive multitask language understanding, 2021. URL https://arxiv.org/abs/2009.03300.

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. *arXiv preprint arXiv:2102.00554*, January 2021.

Zhuohan Li, Eric Wallace, Sheng Shen, Kevin Lin, Kurt Keutzer, Dan Klein, and Joey Gonzalez. Train big, then compress: Rethinking model size for efficient training and inference of transformers. In *International Conference on machine learning*, pp. 5958–5968. PMLR, 2020.

Bin Lin, Ningxin Zheng, Lei Wang, Shijie Cao, Lingxiao Ma, Quanlu Zhang, Yi Zhu, Ting Cao, Jilong Xue, Yuqing Yang, et al. Efficient gpu kernels for n: m-sparse weights in deep learning. *Proceedings of Machine Learning and Systems*, 5:513–525, 2023.

Stephanie Lin, Jacob Hilton, and Owain Evans. Truthfulqa: Measuring how models mimic human falsehoods, 2022. URL https://arxiv.org/abs/2109.07958.

Llama Team, AI @ Meta. The Llama 3 Herd of Models, July 2024. URL http://arxiv.org/abs/2407.21783. arXiv:2407.21783 [cs].

Shuming Ma, Hongyu Wang, Lingxiao Ma, Lei Wang, Wenhui Wang, Shaohan Huang, Li Dong, Ruiping Wang, Jilong Xue, and Furu Wei. The era of 1-bit llms: All large language models are in 1.58 bits. *arXiv preprint arXiv:2402.17764*, 2024.

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

Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–15, 2021.

NVIDIA. Nvidia h100 tensor core gpu architecture, 2022. URL https://resources.nvidia.com/en-us-tensor-core/gtc22-whitepaper-hopper.

NVIDIA. Nvidia rtx blackwell gpu architecture, 2025. URL https://images.nvidia.com/aem-dam/Solutions/geforce/blackwell/nvidia-rtx-blackwell-gpu-architecture.pdf.

Patrik Okanovic, Grzegorz Kwasniewski, Paolo Sylos Labini, Maciej Besta, Flavio Vella, and Torsten Hoefler. High performance unstructured spmm computation using tensor cores. In *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–14. IEEE, 2024.

Keisuke Sakaguchi, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. Winogrande: An adversarial winograd schema challenge at scale, 2019. URL https://arxiv.org/abs/1907.10641.

Seungmin Yu, Xiaodie Yi, Hayun Lee, and Dongkun Shin. Toward efficient permutation for hierarchical n: M sparsity on gpus. *arXiv preprint arXiv:2407.20496*, 2024.

Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. Hellaswag: Can a machine really finish your sentence?, 2019. URL https://arxiv.org/abs/1905.07830.

A Appendix

A.1 CS256 SPMV kernel

Below is the implementation of the sparse-matrix-vector implementation for CS256 sparse matrices:

```
template<class Scalar, int VecSize>
__global__ void spmv(CS256<const Scalar, VecSize> cs256, const Scalar* B,
                             Scalar* C) {
  constexpr const int BLK_UNROLL = 4;
  constexpr const int BLOCK_DIM = 256;
  int m = cs256.m;
  int n = cs256.n;
  int blk_per_row = (n + 255) / 256;
  int blk_offset = ((blk_per_row + 3) / 4) * 4;

  __shared__ Scalar B_buffer[BLK_UNROLL][256];

  // block-index among unrolled blocks
  int block_offset = threadIdx.x % 4;
  int neuron_offset = threadIdx.x / 4;
  int neurons_per_block = BLOCK_DIM / 4;

  assert(m % 64 == 0);

  using WeightVec = GenericVector<Scalar, VecSize>;
  using IdxVec = GenericVector<std::uint8_t, VecSize>;
  using FtrVec = GenericVector<Scalar, 4>;

  for (int i = blockIdx.x * neurons_per_block + neuron_offset; i < m;
       i += gridDim.x * neurons_per_block) {
    int start = cs256.RowPointers[i];
    float s = 0.f;
    FtrVec ftr;
    int idx = threadIdx.x * 4;
    if (idx < cs256.n) {
      ftr = FtrVec::load(B + idx);
    } else {
      ftr = FtrVec::zeros();
    }

    for (int blk_o = 0; blk_o < blk_offset; blk_o += BLK_UNROLL) {
      auto increments = GenericVector<std::uint8_t, BLK_UNROLL>::load(
          cs256.BlockIncrements + i * blk_offset + blk_o);

      __syncthreads();
      *reinterpret_cast<FtrVec*>(&B_buffer[threadIdx.x * 4 / 256][idx % 256]) =
          ftr;
      idx = (blk_o + BLK_UNROLL) * 256 + threadIdx.x * 4;
      if (idx < cs256.n) {
        ftr = FtrVec::load(B + idx);
      } else {
        ftr = FtrVec::zeros();
      }
      __syncthreads();

      int blk = blk_o + block_offset;
      if (blk >= blk_per_row) continue;

      int end = start;
      for (int j = 0; j <= block_offset; ++j) {
        start = end;
        end += increments[j];
      }
```

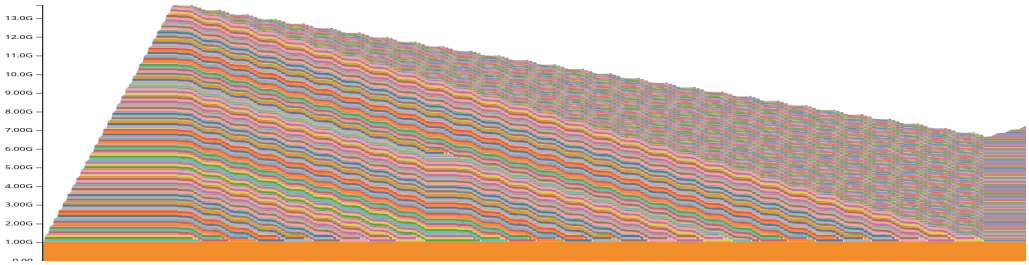


Figure 4: Memory profile when replacing dense weights with CS256 at 66% sparsity.

```
      for (int t = start; t != end; t += VecSize) {
        IdxVec idxs = IdxVec::load(cs256.Indices + t);
        WeightVec vals = WeightVec::load(cs256.Values + t);
        for (int tt = 0; tt < VecSize; ++tt) {
          int idx = idxs[tt];
          float feature = B_buffer[block_offset][idx];
          s += (float) vals[tt] * (float) feature;
        }
      }

      for (int j = block_offset; j < 4; ++j) { start += increments[j]; }
    }

    // reduction for split-k
    {
      cooperative_groups::thread_block block =
          cooperative_groups::this_thread_block();
      cooperative_groups::thread_block_tile<4> tile =
          cooperative_groups::tiled_partition<4>(block);
      s = cooperative_groups::reduce(tile, s,
                                     cooperative_groups::plus<float>{});
      if (tile.thread_rank() == 0) { C[i] = s; }
    }
  }
}
```

A.2 Memory profile

In Figure 4, we present the memory trace as extracted with PyTorch's memory profiler, when substituting weights in LLama-8B from dense to 66% sparse CS256 weights when running on an RTX A4000 GPU. Most of the dense layers fit on the GPU, so when these initial layers are sparsified, the GPU memory consumption decreases. However, the last few layers of the model are offloaded to CPU, so when these are replaced, GPU memory consumption actually increases. The large orange chunk at the bottom of the diagram corresponds to embedding/LM head weights, which remain untouched in our sparsification procedure.