Seeing What Matters: Adaptive Visual Token Compression for Efficient CLIP Inference

CME 213 Final Project Report

Aarya Sumuk

Abstract

CLIP image inference processes an image as a sequence of visual patch tokens. This is accurate, but expensive because later transformer blocks attend over many tokens, including redundant background patches. I study a training-free way to compress CLIP visual tokens after an early transformer block, then evaluate the quality and performance tradeoff with CUDA and MPI. The project implements fixed and adaptive token compression, a custom CUDA selected-token compaction kernel, and a multi-GPU MPI evaluation pipeline. On ImageNetV2, full CLIP-ViT-B/16 reaches 55.48% top-1 accuracy. Keeping 75% of patch tokens after block 1 preserves most quality, reaching 53.55% top-1 accuracy and 0.985 feature cosine similarity. Adaptive entropy + fusion uses 66.76% output tokens on average and reaches 51.26% top-1 accuracy. The CUDA kernel exactly matches `torch.gather` and is modestly faster, but the main bottleneck is the remaining transformer computation. MPI scaling is strong: 4 GPUs reach 3.53× speedup and 88.4% efficiency on 1000 images, while weak scaling remains at 95.4% efficiency with 250 images per rank.

1 Problem and approach

CLIP-ViT-B/16 splits an image into 196 patch tokens plus a CLS token. The vision transformer then processes this full sequence through all blocks. The main question in this project is whether we can reduce the number of visual tokens after an early block while keeping the final CLIP prediction and image representation close to the full-token baseline.

The input is a batch of images from ImageNetV2. The output is a compressed CLIP image feature, zero-shot prediction, and timing/accuracy measurements. I do not train or fine-tune CLIP; all methods are training-free and operate inside the pretrained vision encoder. This makes the project a systems study rather than a new pruning model.

The implemented pipeline is:

1. run CLIP through vision transformer block 1,
2. score patch tokens using CLS-to-patch attention,
3. select a fixed or adaptive token budget,
4. optionally fuse dropped patches into one summary token,
5. compact selected tokens with CUDA,
6. run the remaining CLIP blocks on the shorter sequence.

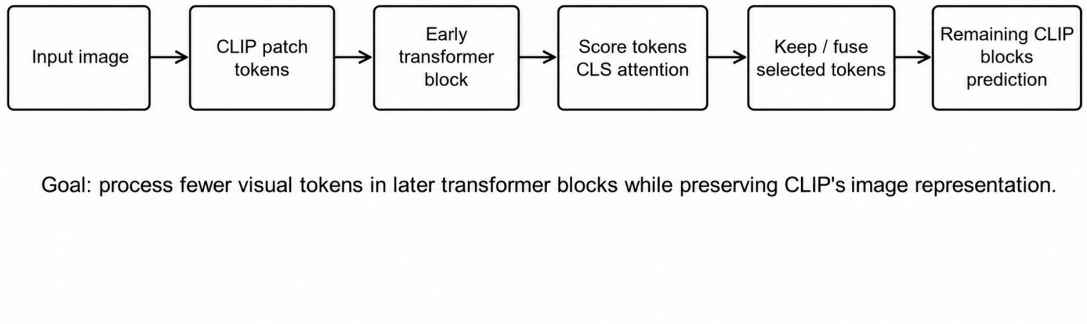


Figure 1: Token compression pipeline. Patch tokens are scored after an early CLIP block, compacted, and passed through the remaining transformer blocks.

2 Related work and design choice

CLIP learns transferable image and text representations through contrastive image-text pretraining [2]. Its image encoder follows the Vision Transformer design, where an image is represented as a sequence of patch tokens [1]. I evaluate on ImageNetV2, a re-collected ImageNet-style validation set used to test generalization under dataset shift [3].

Prior efficient-transformer methods show that visual tokens are often redundant. DynamicViT learns to sparsify tokens dynamically [4], EViT reorganizes and fuses tokens [5], and Token Merging merges similar tokens rather than dropping them directly [6]. These methods are useful reference points because they show that token reduction can work, but they usually require training, architectural changes, or a different token-update rule.

My design is intentionally simpler. I use direct top-K token selection inside a pretrained CLIP model, without retraining or changing the architecture. The goal is not to claim a new state-of-the-art pruning method. Instead, the goal is to measure the systems tradeoff: how much CLIP quality is preserved, how much the CUDA compaction step costs, and how well the resulting evaluation pipeline scales with MPI.

3 CUDA and MPI implementation

CUDA compaction. After token selection, the compaction step gathers selected embeddings:

$$\text{output}[b, k, d] = \text{tokens}[b, \text{indices}[b, k], d].$$

The input token tensor has shape $B \times N \times D$, the selected index tensor has shape $B \times K$, and the output has shape $B \times K \times D$. The CUDA kernel flattens the (b, k, d) space and uses one thread per output scalar. Writes are contiguous over the embedding dimension, while input reads are irregular in the selected token index. This is a memory-movement kernel, not a compute-heavy kernel, so I do not use shared memory or tensor cores.

MPI evaluation. The MPI implementation is data parallel. Each rank loads CLIP on one GPU and processes a cyclic shard of the dataset:

$$r,\ r+P,\ r+2P,\ldots$$

for rank r out of P ranks. Ranks do not exchange image tensors, token tensors, or activations. At the end, they combine scalar counts and sums using `MPI_Allreduce`. Timing uses the maximum rank time, since the slowest rank determines distributed runtime. I verified that the 4-rank run maps ranks 0–3 to GPUs 0–3.

This layout is intentionally communication-light. It matches the natural structure of the workload: images are independent, while final metrics are tiny.

4 Performance model and measurement

For distributed evaluation, a simple model is

$$T(P,N) \approx T_{\text{setup}} + \frac{N}{P} t_{\text{img}} + T_{\text{comm}}(P), \qquad T_{\text{comm}}(P) \approx \alpha \log P + \beta m,$$

where P is the number of GPUs, N is the number of images, and m is the number of scalar metrics reduced at the end. Since m is small and no large tensors are communicated, the model predicts that MPI communication should be negligible. Strong scaling should be close to linear until fixed overheads dominate. Weak scaling should keep wall time nearly constant when the number of images per rank is fixed.

The CUDA model is different. Compaction performs little arithmetic: each output scalar needs an index lookup, one read, and one write. The expected bottleneck is memory access, especially the irregular input gather. This predicts only modest speedup over PyTorch's optimized `torch.gather`.

I measured zero-shot accuracy, top-1 consistency with full CLIP, feature cosine similarity, CUDA compaction time, component timing, MPI wall time, speedup, efficiency, and communication time. CUDA timings used synchronized GPU/PyTorch timing after warmup. MPI timings used wall-clock timing around distributed evaluation and a separate timing around the final reductions.

All experiments were run on the CME 213 teaching cluster using NVIDIA Quadro RTX 6000 GPUs, PyTorch, CUDA 12.3, and mpi4py with one MPI rank per GPU.

5 Results and analysis

5.1 Compression quality

Full CLIP reaches 55.48% top-1 and 81.30% top-5 accuracy on ImageNetV2. Fixed 75% compression preserves most of that quality. Fixed 50% saves more tokens but loses more accuracy, while 25% is too aggressive.

Table 1: Fixed-budget token compression after CLIP vision transformer block 1.

Kept tokens	K	Top-1	Top-5	Consistency	Feature cosine
25%	49	14.96%	30.05%	18.99%	0.695
50%	98	45.42%	71.42%	63.73%	0.916
75%	147	53.55%	79.66%	84.15%	0.985

I also tested adaptive entropy + fusion. The entropy policy assigns different budgets based on the spread of CLS-attention scores, and the fusion step adds one weighted summary token from dropped patches. This does not beat fixed 75%, but it gives a useful middle point: fewer tokens than fixed 75%, with much better quality than fixed 50%.

Table 2: Adaptive entropy + fusion compared with fixed compression.

Method	Avg. output tokens	Top-1	Consistency	Feature cosine
Full CLIP	100.00%	55.48%	100.00%	1.000
Fixed 50%	50.00%	45.42%	63.73%	0.916
Adaptive entropy + fusion	66.76%	51.26%	77.43%	0.962
Fixed 75%	75.00%	53.55%	84.15%	0.985

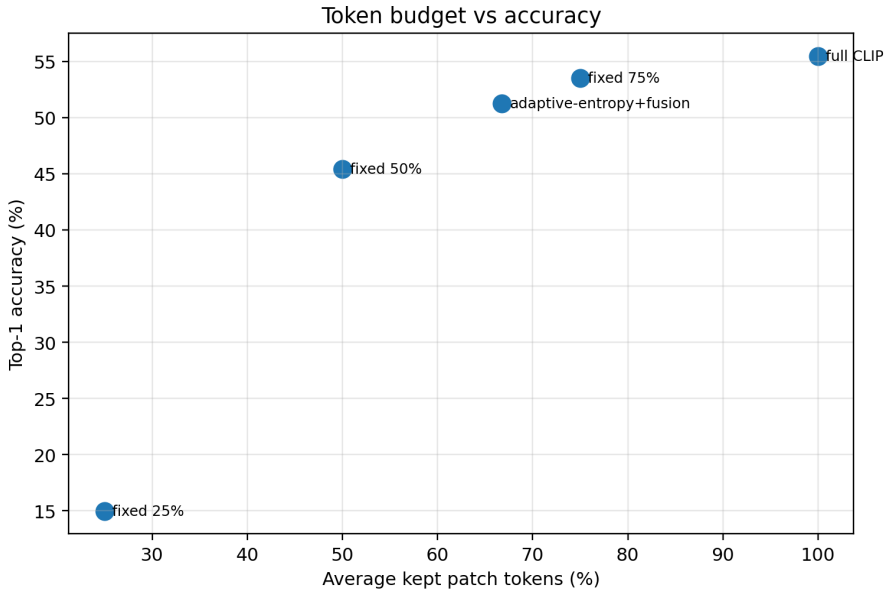


Figure 2: Accuracy versus token budget for fixed and adaptive compression.

5.2 CUDA benchmark

The custom CUDA kernel exactly matches `torch.gather`, with maximum absolute error 0.0. It is consistently faster, but the speedup is small because compaction is memory-bound and sub-millisecond.

Table 3: CUDA compaction compared with PyTorch `torch.gather` on real CLIP tokens.

Kept tokens	K	CUDA	`torch.gather`	Speedup
25%	49	0.059 ms	0.069 ms	1.16×
50%	98	0.103 ms	0.111 ms	1.08×
75%	147	0.147 ms	0.154 ms	1.04×

The end-to-end bottleneck is not compaction. For 75% compression, gather takes about 0.160 ms,

while the remaining transformer blocks take about 140.9 ms per batch. This is consistent with the GPU memory model: the custom kernel improves a small memory-movement primitive, but most time is spent in transformer computation.

5.3 MPI scaling

The MPI results match the performance model. Since each rank processes independent images and only reduces scalar metrics, communication is tiny. Strong scaling is close to linear through 4 GPUs, with some efficiency loss from fixed overheads such as Python iteration, synchronization, dataloader overhead, and kernel launches.

Table 4: MPI strong scaling with fixed total problem size of 1000 images.

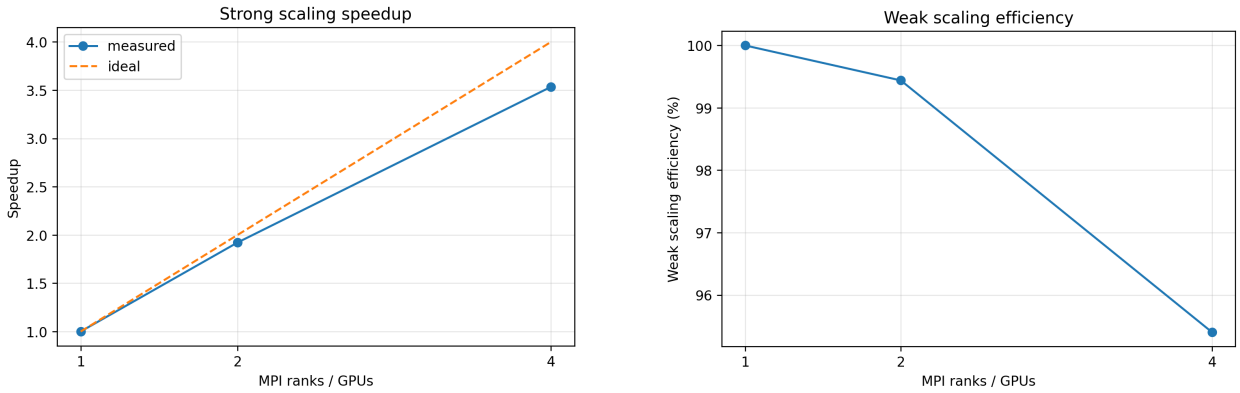
GPUs	Wall time	Throughput	Speedup	Efficiency
1	11.13 s	89.82 img/s	1.00×	100.0%
2	5.79 s	172.60 img/s	1.92×	96.1%
4	3.15 s	317.48 img/s	3.53×	88.4%

Weak scaling keeps the local problem size fixed at 250 images per rank. Wall time remains nearly flat from 1 to 4 GPUs, which confirms that communication is not the limiting cost at this scale.

Table 5: MPI weak scaling with 250 images per rank.

GPUs	Total images	Wall time	Throughput	Weak efficiency
1	250	3.12 s	80.02 img/s	100.0%
2	500	3.14 s	159.14 img/s	99.4%
4	1000	3.27 s	305.35 img/s	95.4%

This behavior is consistent with Amdahl's law and the lecture model for collectives. The parallel part is the image evaluation, which scales well, while the non-scaling part is fixed overhead and final synchronization. Figure 3 summarizes both effects: strong scaling reaches 3.53× on 4 GPUs, and weak scaling remains above 95% efficiency. The measured final reduction time is below 1 ms in the clean runs, so the main bottleneck is local model execution, not MPI communication.



(a) Strong scaling speedup.

(b) Weak scaling efficiency.

Figure 3: MPI scaling summary.

5.4 Algorithmic variants

The main variants were fixed budgets, adaptive entropy allocation, fused dropped-token summaries, token scoring rules, and CUDA versus PyTorch compaction. The most useful quality variant was adaptive entropy + fusion. The scoring ablation also showed that CLS attention works best at the 75% budget, but random selection is surprisingly competitive at 50%. This suggests that CLIP patch tokens are redundant under moderate pruning, while attention-based selection matters more when trying to preserve high-quality predictions.

6 Correctness and limitations

I checked correctness at three levels. First, the CUDA compaction kernel was compared directly with `torch.gather`; the maximum absolute error was 0.0. Second, the model-level results behave as expected: accuracy, consistency, and feature cosine all improve as more tokens are kept. Third, the MPI implementation gives stable aggregate metrics across 1, 2, and 4 ranks on the same 1000-image evaluation, and the 4-rank run maps ranks to separate GPUs.

The main limitation is that the adaptive policy is simple. Entropy-based token allocation is easy to implement, but the attention entropy values are highly concentrated, so the signal is weak. A better policy could use CLIP prediction margin, model confidence, or a learned lightweight selector. On the systems side, a full Nsight trace would give more detail about GPU utilization and kernel-launch overhead. Still, the current measurements are enough to identify the main conclusion: token compression can reduce transformer work, CUDA compaction is correct but not the main bottleneck, and MPI scales well because the workload is naturally data parallel.

Acknowledgment of assistance

I used ChatGPT as a writing and debugging aid for this project. I used it to help organize implementation steps, debug CUDA/MPI environment issues, and revise parts of the writeup. I ran the experiments myself, checked the outputs, and verified the final results before using them in this report.

References

[1] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, and N. Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. In *International Conference on Learning Representations*, 2021.

[2] A. Radford, J. W. Kim, C. Hallacy, A. Ramesh, G. Goh, S. Agarwal, G. Sastry, A. Askell, P. Mishkin, J. Clark, G. Krueger, and I. Sutskever. Learning transferable visual models from natural language supervision. In *International Conference on Machine Learning*, 2021.

[3] B. Recht, R. Roelofs, L. Schmidt, and V. Shankar. Do ImageNet classifiers generalize to ImageNet? In *International Conference on Machine Learning*, 2019.

[4] Y. Rao, W. Zhao, B. Liu, J. Lu, J. Zhou, and C.-J. Hsieh. DynamicViT: Efficient vision transformers with dynamic token sparsification. In *Advances in Neural Information Processing Systems*, 2021.

[5] Y. Liang, C. Ge, Z. Tong, Y. Song, J. Wang, and P. Xie. EViT: Expediting vision transformers via token reorganizations. In *International Conference on Learning Representations*, 2022.

[6] D. Bolya, C.-Y. Fu, X. Dai, P. Zhang, C. Feichtenhofer, and J. Hoffman. Token Merging: Your ViT but faster. In *International Conference on Learning Representations*, 2023.

[7] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, 2019.

Appendix: Optional figures

These figures are useful backup material but are not required for the main argument.

Figure 4: Example ImageNetV2 images.

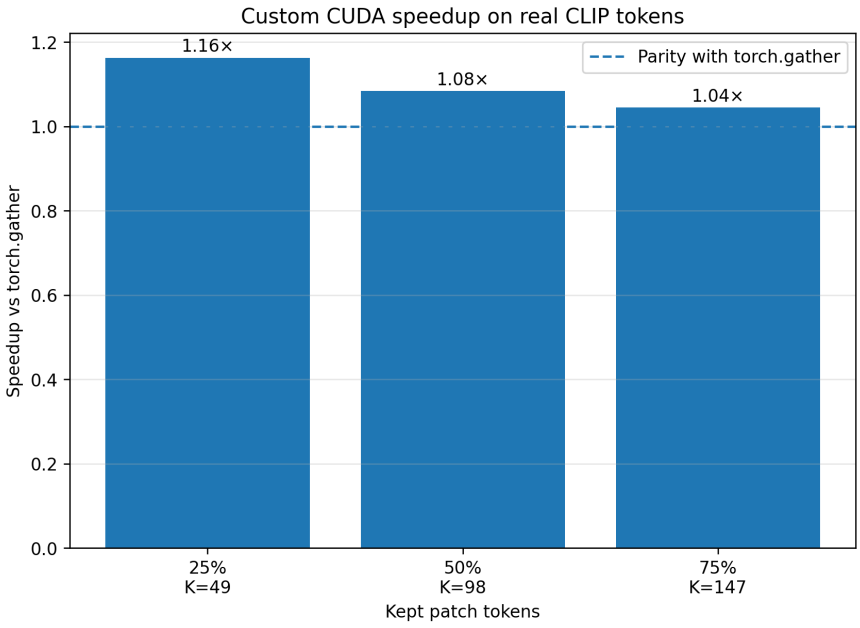


Figure 5: Custom CUDA compaction speedup over `torch.gather`.

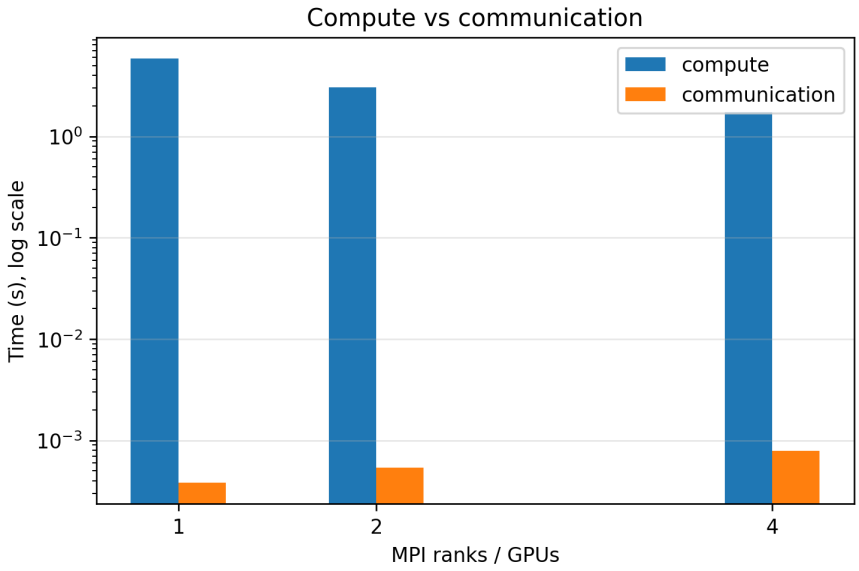


Figure 6: Compute versus communication time. Communication is below 1 ms in the clean runs.

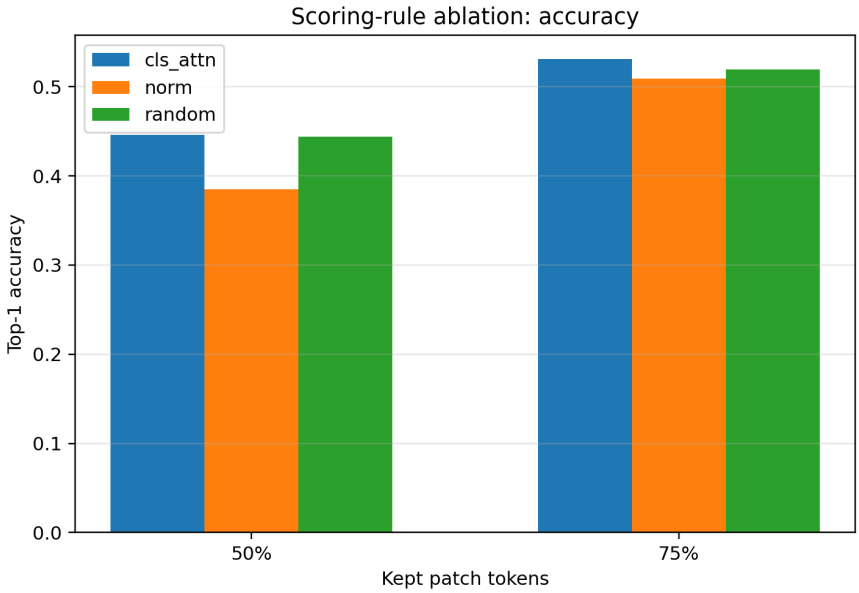


Figure 7: Token scoring ablation.

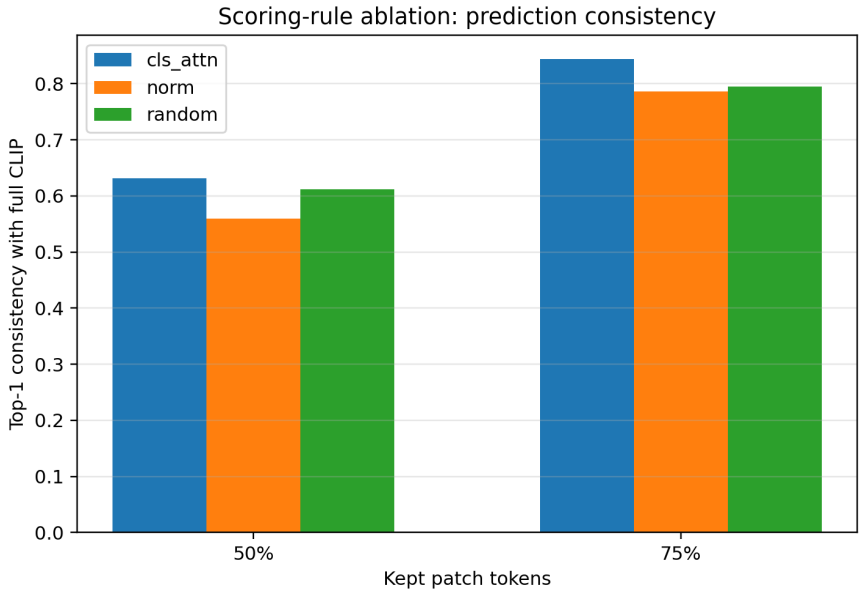


Figure 8: Top-1 consistency for token scoring rules.

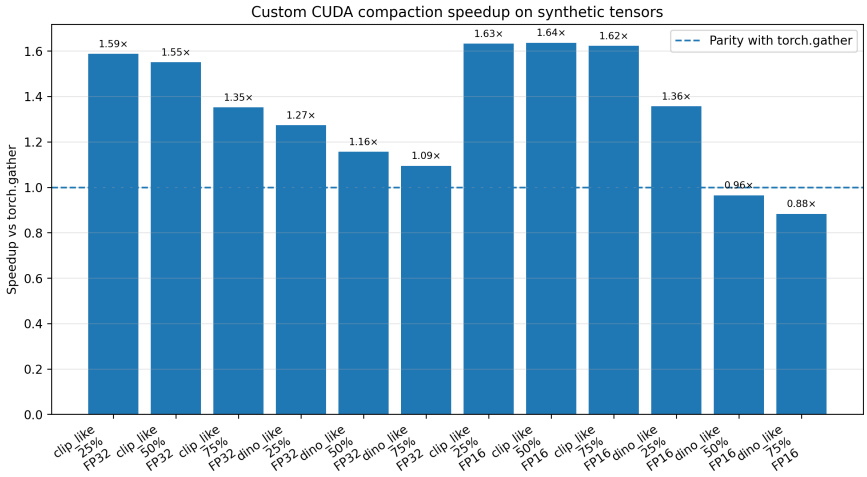


Figure 9: Synthetic CUDA compaction benchmark.