70% Size, 100% Accuracy: Lossless LLM Compression for Efficient GPU Inference via Dynamic-Length Float (DFloat11)

Tianyi Zhang, Mohsen Hariri, Shaochen Zhong, Vipin Chaudhary, Yang Sui, Xia Hu, Anshumali Shrivastava

Limitations of Existing Compression Algorithms

Large Language Models (LLMs) and Diffusion Models (DMs) are large (GBs to TBs) and memory-intensive.

Existing lossy model compression methods (quantization, pruning, etc.) → tradeoffs between model quality and compression factor

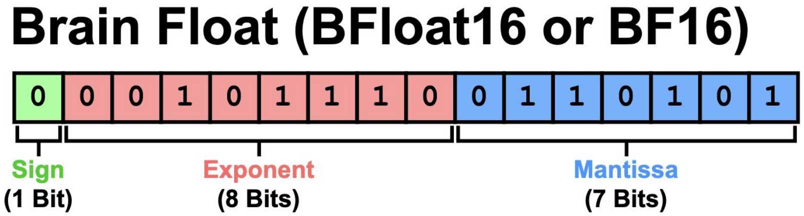
- Accuracy & performance degradations
- Model behavior changes (*flips*[1])

Existing lossless compression (lossless encoding) → does not support GPU inference

[1] Dutta, Abhinav, et al. "Accuracy is not all you need." Advances in Neural Information Processing Systems 37 (2024): 124347-124390.

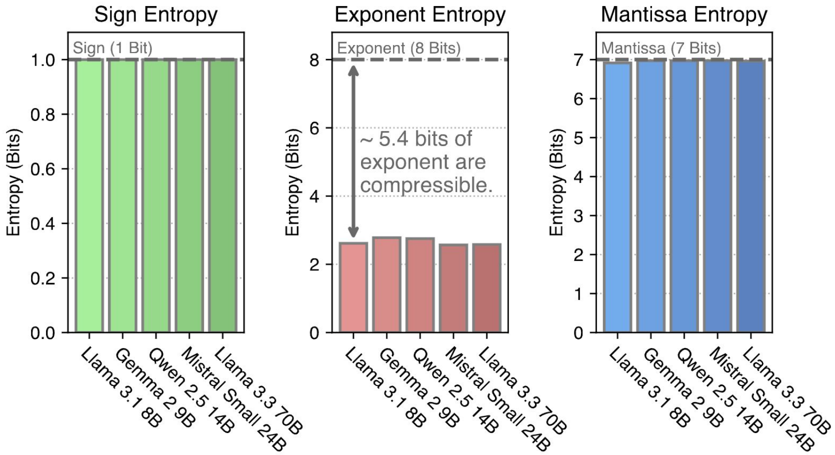
BFloat16 & Its Inefficiency

Foundation models predominantly use BFloat16



Our Insight: Low entropy in BFloat16 exponents

- Real entropy vs. allocated bit width: **2.6 bits** vs. **8 bits**
- Significant potential for lossless compression!



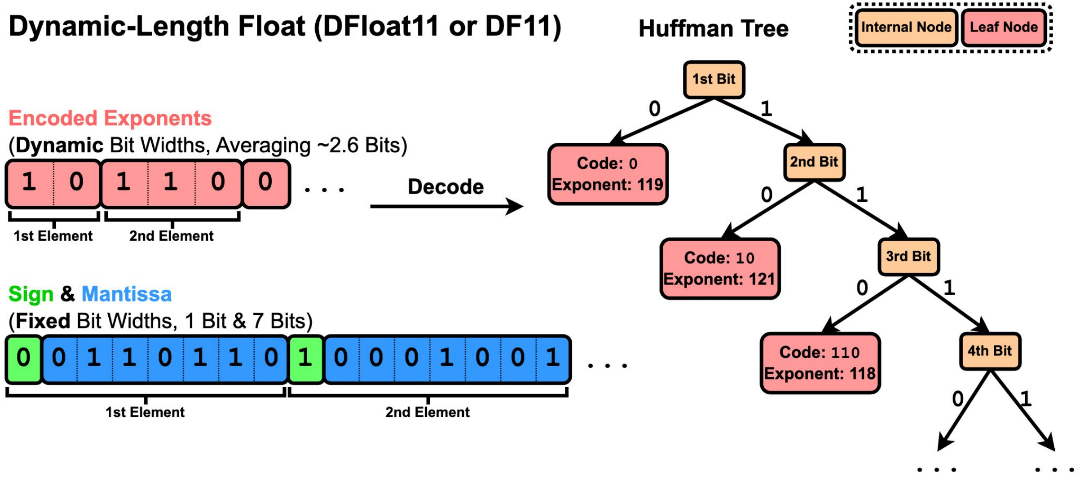
Lossless Compression via Huffman Coding

Huffman coding[1]

- Prefix-free binary tree
- Guaranteed lossless
- Information optimal for any given frequency distribution

Our proposal: Dynamic-Length Float (DFloat11)

- Huffman coding on BFloat16 exponents
- Pack *signs* and *mantissas* tightly



[1] Huffman, David A. "A Method for the Construction of Minimum-Redundancy Codes." Proceedings of the IRE, vol. 40, no. 9, Sept. 1952, pp. 1098–1101.

Core Challenge: GPU Inference with Dynamic-Length Float

GPUs are

- Massively parallel
- Inefficient at branching

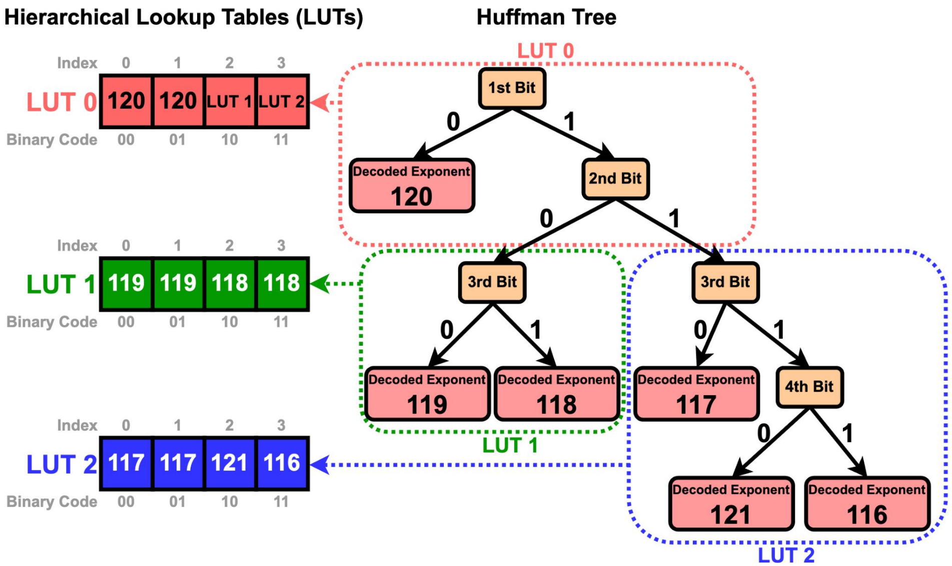
Dynamic-Length Float

- Variable-length encodings → difficult to coordinate GPU threads
- Binary-tree-based decoding → not suitable for GPU computation

Our Solution: Hardware-aware Algorithmic Designs

Technique #1 — Decoding Huffman codes via hierarchical lookups

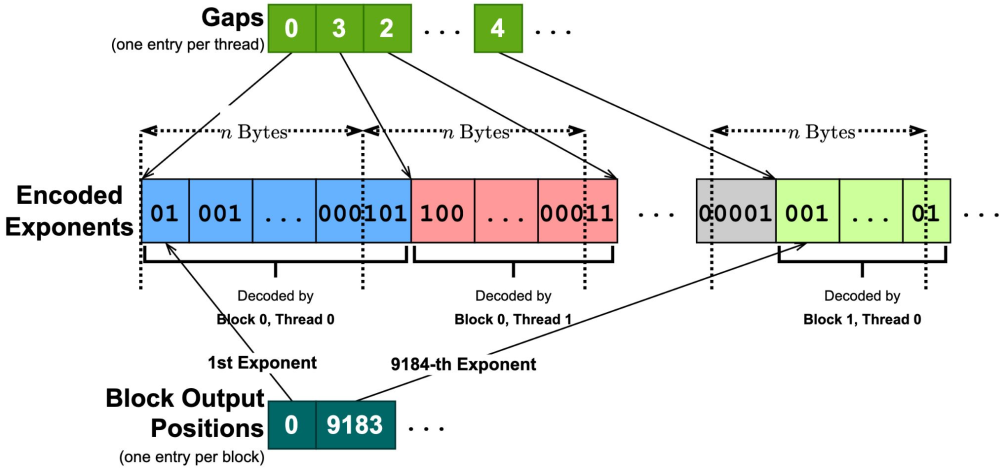
- Use lookups instead of binary tree traversal
- Decompose the monolithic lookup table (LUT) into compact LUTs



Our Solution: Hardware-aware Algorithmic Designs

Technique #2 — Two-phase Kernel for Thread Coordination

- Use thread-level *gap arrays* and block-level *output-position arrays* to determine the bit position
- Phase 1 — count the number of decoded elements, and synchronize threads
- Phase 2 — write the decoded elements to memory



Our Solution: Hardware-aware Algorithmic Designs

Technique #3 — Transformer-block level decompression

- More data to decompress at once → better GPU utilization → higher throughput
- Batch the decompression of all matrices in a transformer block

32% Size Reduction, 0 Accuracy Loss

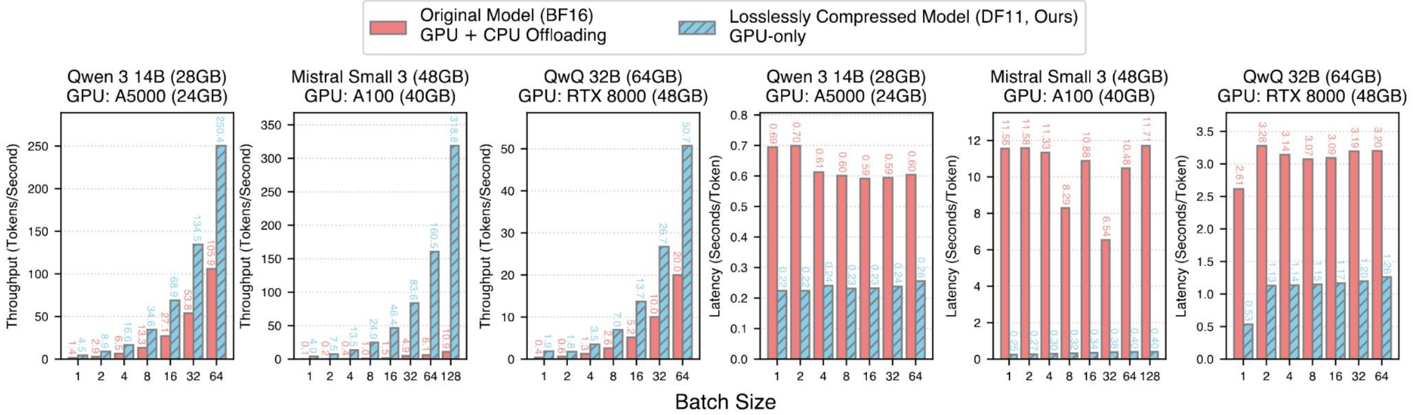
Table 1: DF11 statistics for various models. Model sizes are shown before and after compression.

Model	Original → DF11 Compressed	Compression Ratio	Avg. Bit Width
Large Language Models			
Llama 3.1 8B Instruct	16.06 GB → 10.90 GB	67.84%	10.85
Llama 3.3 70B Instruct	141.11 GB → 95.40 GB	67.61%	10.82
Llama 3.1 405B Instruct	811.71 GB → 551.22 GB	67.91%	10.87
Qwen 3 14B	29.54 GB → 20.14 GB	68.17%	10.91
QwQ 32B	65.53 GB → 44.65 GB	68.14%	10.90
Mistral Nemo Instruct	24.50 GB → 16.59 GB	67.74%	10.84
Mistral Small 3	47.14 GB → 31.86 GB	67.58%	10.81
Phi 4 Reasoning Plus	29.32 GB → 19.83 GB	67.64%	10.82
DeepSeek R1 Distill Llama 8B	16.06 GB → 10.89 GB	67.81%	10.85
Diffusion Transformers			
FLUX.1 dev	23.80 GB → 16.33 GB	68.61%	10.98
FLUX.1 schnell	23.78 GB → 16.31 GB	68.58%	10.97
Stable Diffusion 3.5 Large	16.29 GB → 11.33 GB	69.52%	11.12

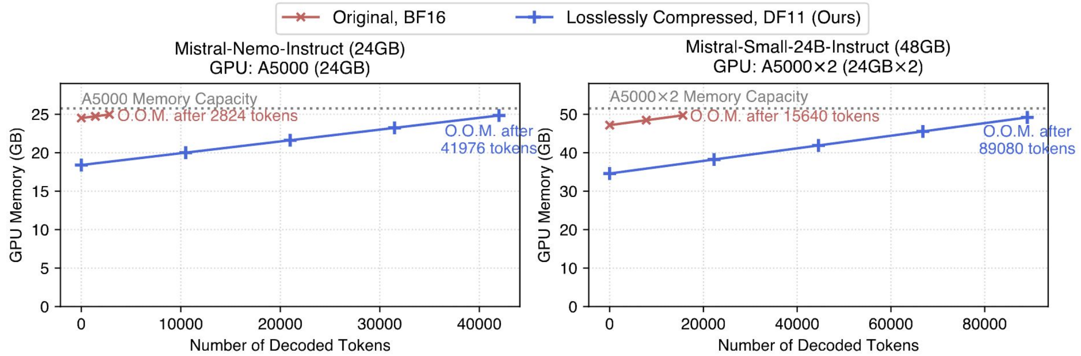
Table 2: Comparison of accuracy and perplexity for the BF16 and DF11 models on different benchmarks. DF11 compression results in absolutely no loss in accuracy or perplexity.

		Accuracy		Perplexity	
Model	**Data Type**	MMLU	TruthfulQA	WikiText	C4
Llama 3.1 8B Instruct	BF16	68.010 ± 0.375	36.965 ± 1.690	8.649	21.677
	DF11 (Ours)	68.010 ± 0.375	36.965 ± 1.690	8.649	21.677

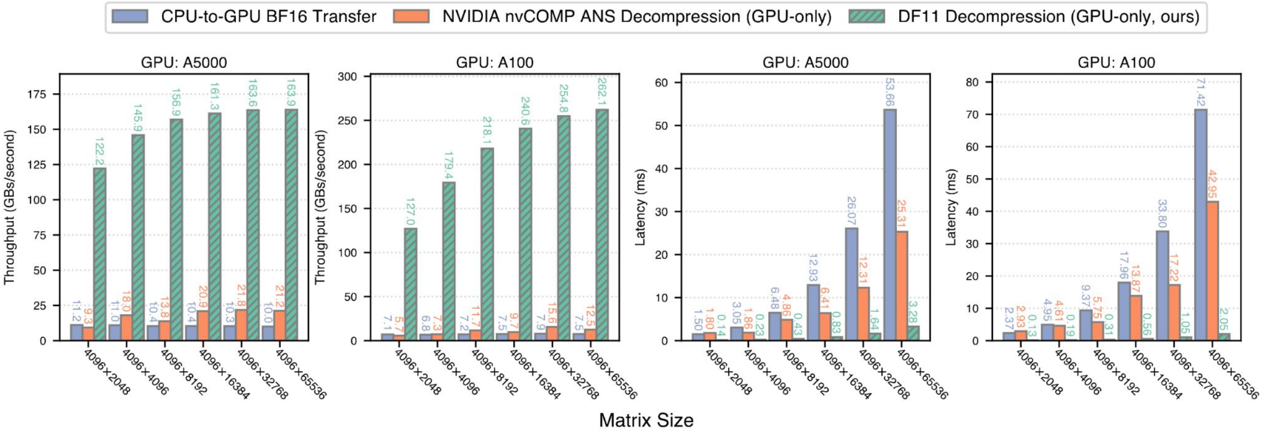
2.31—46.24× Faster Than Offloading



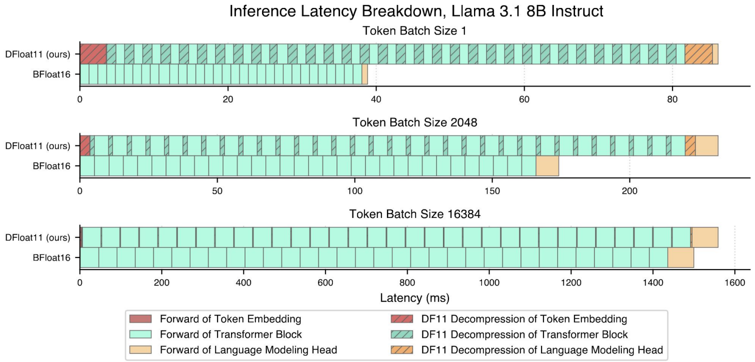
5.70—14.86× Longer Context Length



20.97× Faster Than NVIDIA nvCOMP



Negligible Decompression Overhead at Large Batch Size



Thank you

Paper

Code