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

Tianyi Zhang[1], **Mohsen Hariri**[2], **Shaochen (Henry) Zhong**[1], **Vipin Chaudhary**[2], **Yang Sui**[1], **Xia Hu**[1], and **Anshumali Shrivastava**[1,3]

[1]Department of Computer Science, Rice University
[2]Department of Computer and Data Sciences, Case Western Reserve University
[3]Ken Kennedy Institute

{tz21, henry.zhong, yang.sui, xia.hu, anshumali}@rice.edu, {mohsen.hariri, vipin}@case.edu

Code: https://github.com/LeanModels/DFloat11
Models: https://huggingface.co/DFloat11

Abstract

Large-scale AI models, such as Large Language Models (LLMs) and Diffusion Models (DMs), have grown rapidly in size, creating significant challenges for efficient deployment on resource-constrained hardware. In this paper, we introduce *Dynamic-Length Float* (DFloat11), a lossless compression framework that reduces LLM and DM size by 30% while preserving outputs that are bit-for-bit identical to the original model. DFloat11 is motivated by the low entropy in the BFloat16 weight representation of LLMs, which reveals significant inefficiency in the existing storage format. By applying entropy coding, DFloat11 assigns dynamic-length encodings to weights based on frequency, achieving near information-optimal compression without any loss of precision. To facilitate efficient inference with dynamic-length encodings, we develop a custom GPU kernel for fast online decompression. Our design incorporates the following: (i) compact, hierarchical lookup tables (LUTs) that fit within GPU SRAM for efficient decoding, (ii) a two-phase GPU kernel for coordinating thread read/write positions using lightweight auxiliary variables, and (iii) transformer-block-level decompression to minimize latency. Experiments on Llama 3.3, Qwen 3, Mistral 3, FLUX.1, and others validate our hypothesis that DFloat11 achieves around 30% model size reduction while preserving bit-for-bit identical outputs. Compared to a potential alternative of offloading parts of an uncompressed model to the CPU to meet memory constraints, DFloat11 achieves 2.3–46.2× higher throughput in token generation. With a fixed GPU memory budget, DFloat11 enables 5.7–14.9× longer generation lengths than uncompressed models. Notably, our method enables lossless inference of *Llama 3.1 405B*, an 810GB model, on a single node equipped with 8×80GB GPUs.

1 Introduction

Foundation models, such as Large Language Models (LLMs) and Diffusion Models (DMs), have demonstrated remarkable capabilities across a wide range of Natural Language Processing (NLP) [56] and Computer Vision (CV) tasks [57]. However, their huge model sizes create substantial obstacles

39th Conference on Neural Information Processing Systems (NeurIPS 2025).

for efficient deployment, especially in memory-constrained environments. For example, a competitive recent LLM, *Llama 3.1 405B* [20], has 405 billion parameters in 16-bit Brain Float (BFloat16) format and requires about 810 GB of memory for full inference, exceeding the capacity of a typical high-end GPU server (e.g., DGX A100/H100 with 8×80GB GPUs). As a result, deploying this model requires multiple nodes, making it expensive and inaccessible. **In this work, we present a solution that compresses any BFloat16 model to approximately 70% of its original size while preserving 100% of its accuracy on any task.**

Model compression via quantization has limitations. Quantization is a type of *lossy* compression method that lowers the precision of model weights by converting them into lower bit-width representations [15, 37, 36, 43]. Although it can significantly reduce memory usage and often improve inference speed, quantization is not a one-size-fits-all solution and presents several key limitations: ❶ *Accuracy degradation*. By design, quantization introduces approximation errors. The degree of accuracy loss depends on multiple factors, including the base model, quantization method, evaluation benchmark, and target bit-width [35]. These interactions make it difficult to predict or quantify the impact comprehensively. Even mild quantization can noticeably degrade performance. For example, applying 8-bit SmoothQuant [51] to *DeepSeek-R1-Distill-Qwen-1.5B* [21] results in a 9.09% drop in average accuracy across reasoning tasks [39]. ❷ *Behavioral shifts.* Even when overall accuracy metrics appear roughly unchanged, quantized models may behave differently from their full-precision counterparts. For instance, Dutta et al. [13] observe a phenomenon called *flips*, where quantized models produce answers that change from correct to incorrect and vice versa. This indicates that quantization can significantly alter model behavior, even when standard accuracy metrics show minimal change. For example, the W8A16 GPTQ-quantized Qwen2-1.5B[15, 54] exhibits only a 0.3% drop in GSM8K (8-shot) accuracy [5], yet 6.37% of its answers flip in correctness [13]. ❸ *Compliance and reliability concerns.* In domains like finance or healthcare, quantized models may not satisfy regulatory or reliability standards, as their outputs may differ from those of the original models [31]. We refer readers to Appendix A for a more detailed discussion on quantization.

Existing lossless model compression does not support efficient GPU inference. Unlike lossy compression, *lossless compression* reduces model size while preserving the full precision of the original weights. This ensures the model's output distribution remains identical to that of the uncompressed counterpart. However, most existing lossless methods focus on storage efficiency, such as compressing model checkpoints [22, 25], or target specialized hardware like FPGAs [59], rather than accelerating inference on general-purpose GPUs. While useful for tasks like checkpoint rollback during large-scale training [47] or reducing download time from model hubs [25], these methods offer little to no benefit for GPU-based inference.

Our proposal, Dynamic-Length Float (DFloat11), is a lossless compression framework optimized for efficient GPU inference. We identify a key inefficiency in the commonly used BFloat16 format: its 8-bit exponent field carries only about 2.6 bits of actual information. This redundancy is consistent across a wide range of LLMs, as shown in Section 2.2. To exploit it, we apply Huffman coding [28] to the exponent bits of BFloat16 weights, while leaving the sign and mantissa bits uncompressed. The resulting exponents have dynamic-length encodings: frequent values are assigned shorter codes, while rarer ones use longer codes. However, standard Huffman decoding relies on sequential bit-by-bit tree traversal, which is inefficient on GPUs due to limited parallelism. Assigning one GPU thread per decompression task leads to severe hardware underutilization and high latency. To overcome this, we design a hardware-aware algorithm that enables efficient online decompression of dynamic-length floats on GPUs. Our solution includes three key components: 1. compact, hierarchical lookup tables (LUTs) that fit in GPU SRAM to support fast, table-based Huffman decoding, 2. a two-phase GPU kernel that uses lightweight auxiliary variables to coordinate thread-level read and write operations, and 3. batched decompression at the transformer-block level to maximize throughput. We summarize our contributions as follows:

1. We propose **Dynamic-Length Float (DFloat11)**, a losslessly compressed floating-point format that reduces BFloat16 weights to approximately 11 bits. This yields around 30% model size reduction with bit-for-bit identical outputs.
2. We develop optimized, hardware-aware algorithms for efficient GPU inference with DFloat11-compressed models by leveraging GPU memory and compute hierarchies.

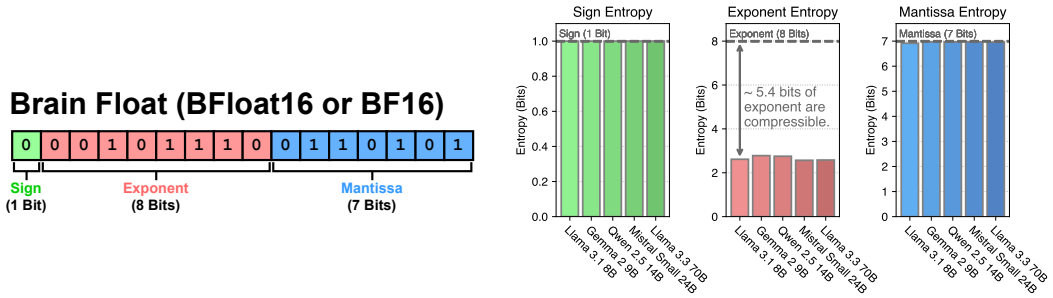

Figure 1: **(Left)** The allocation of bits for the components of BFloat16. **(Right 3)** The Shannon entropy of the components (sign, exponent, mantissa) of BFloat16 weights in various LLMs.

3. We evaluate DFloat11 across popular LLMs and diffusion transformers, including Llama 3, Qwen 3, Mistral 3, DeepSeek R1 Distilled, FLUX.1, and Stable Diffusion 3.5 [20, 46, 45, 21, 32, 2]. Our method consistently achieves 30% compression without altering original outputs at all. Notably, it enables running *Llama-3.1-405B on a single node* (8×80GB A100 GPUs), reducing hardware requirements by half without accuracy loss.

2 Method

In this section, we introduce our proposed floating-point format, Dynamic-Length Float (DFloat11), along with its custom decompression kernel designed for efficient GPU inference.

2.1 Preliminary

Brain Float (BFloat16) Recent state-of-the-art LLMs predominantly employ the 16-bit Brain Float format (BFloat16 or BF16) for storing weights, due to its balance of numerical precision with memory efficiency. BF16 allocates its 16 bits as follows: 1 *sign* bit, 8 *exponent* bits, and 7 *mantissa* bits. The numerical value represented by a BF16 number is computed as:

$$(-1)^{\text{sign}} \times 2^{\text{exponent}-127} \times (1.\text{mantissa}), \tag{1}$$

where mantissa is interpreted as a binary fractional value.

Entropy Coding Entropy coding is a core technique in lossless data compression that leverages statistical redundancy to reduce data size. Several widely used methods fall under this category, including Huffman coding [28], arithmetic coding [33], and Asymmetric Numeral Systems (ANS) [12]. Among these, Huffman coding is one of the most widely adopted, which uses variable-length encoding to minimize the size of encoded data. It assigns shorter binary codes to more frequent symbols and longer codes to less frequent ones. The codes are decoded using a prefix-free binary tree, known as a Huffman tree. Due to the prefix-free property of Huffman codes, no code is a prefix of any other, which ensures unique decodability of the encoded bitstream without the need for delimiters. The tree is constructed based on symbol frequencies and is provably optimal for any given frequency distribution. However, decoding Huffman codes in a massively parallel manner is challenging due to its inherently sequential nature.

GPU Computation and Memory Paradigm GPUs are designed to perform computations in a massively parallel manner. A modern GPU consists of thousands of threads, which are organized into blocks and executed on streaming multiprocessors (SMs). Each block has access to a small, fast, on-chip memory called shared memory (often referred to as SRAM), which provides much lower latency and higher bandwidth than the off-chip global memory, commonly known as high-bandwidth memory (HBM). The capacity of shared memory is limited, typically having up to 100 KB per block. In this work, we leverage the fast access characteristics of SRAM to enable efficient on-the-fly decompression of compressed weights during inference.

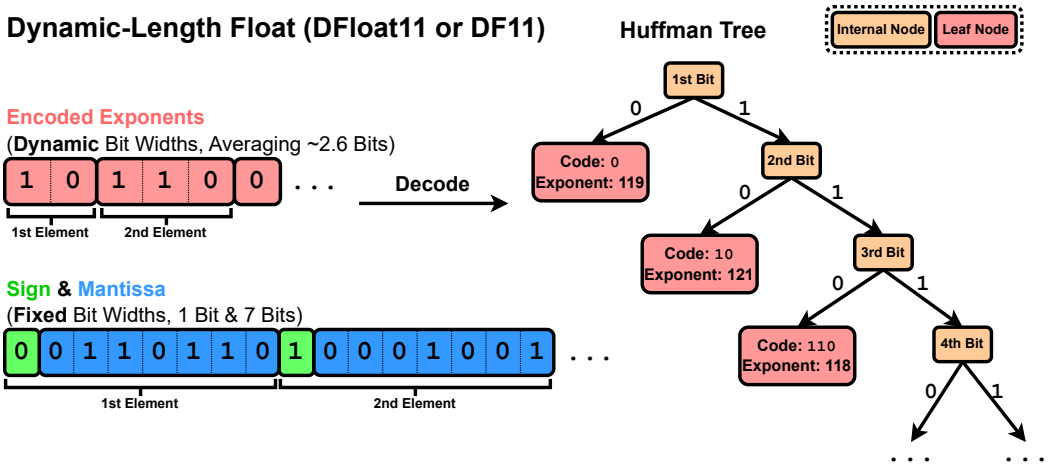

Figure 2: Our proposed format *Dynamic-Length Float* for compressing BFloat16 weights of LLMs losslessly down to 11 bits. The exponents are compressed via Huffman coding, while the sign and mantissa bits remain uncompressed.

2.2 Motivation: BFloat16 Representation is Information Inefficient

To motivate the lossless compression of LLM weights, we analyze the compressibility of the BFloat16 weights of recent LLMs. Specifically, we use Shannon entropy to quantify the information content of BFloat16 components (sign, exponent, and mantissa) for all linear projection matrices of an LLM. The Shannon entropy $H(\cdot)$ is defined as:

$$H(X) = -\sum_{x \in \mathcal{X}} p(x) \log_2 p(x) \tag{2}$$

where X is a discrete random variable with support $\mathcal{X}$, and $p : \mathcal{X} \to [0, 1]$ denotes its probability mass function. We present the computed entropy values in Figure 1. As shown, the entropy of the sign and mantissa bits is close to their respective bit widths, indicating limited potential for compression. In contrast, the exponent exhibits significantly lower entropy, approximately 2.6 bits versus its allocated 8 bits, suggesting substantial opportunities for lossless compression.

To understand this discrepancy, we visualize the frequency distribution of all BFloat16 components in Figure 8 and the ranked frequency of exponent values in Figure 9, both in the Appendix. The sign and mantissa values are relatively uniform across their ranges, but the exponent distribution is highly imbalanced: only about 40 of the 256 possible 8-bit values are used, with the rest never appearing. Ranked frequencies also decay rapidly. These observations reveal the low entropy of the exponent and its potential for compression.

2.3 Dynamic-Length Float: Lossless LLM Compression for Efficient GPU Inference

To address the substantial information inefficiency in the BFloat16 representation of LLM weights, we propose a lossless compression framework that encodes floating-point parameters using entropy coding. Specifically, we build a Huffman tree based on the distribution of exponents in model weights. We then compress the exponents using Huffman coding, while preserving the original signs and mantissas. Exponents are encoded and tightly bit-packed into a byte array, EncodedExponent, while the sign and mantissa are left uncompressed and stored in a separate byte array PackedSignMantissa. Figure 2 illustrates Dynamic-Length Float (DFloat11 or DF11), our proposed format for compactly representing BFloat16 model parameters.

The Core Challenge: Efficient GPU Inference with Compressed Weights While DFloat11 enables lossless compression of LLM weights, efficient GPU inference remains a key challenge. Entropy-coded weights use variable-length encoding and cannot be directly used in matrix multiplications. As a result, each weight matrix must be decompressed on-the-fly to its original BFloat16 format

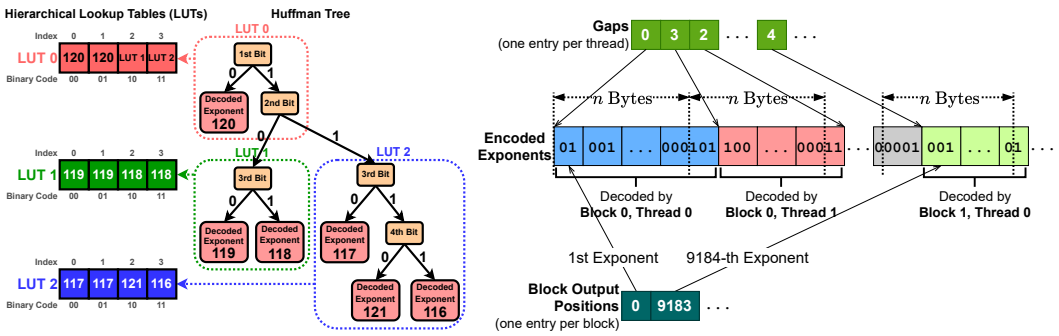

Figure 3: **(Left)** The Huffman tree is decomposed into a set of non-overlapping subtrees, each corresponding to a compact lookup table (LUT). These hierarchical LUTs reside in GPU SRAM to enable efficient Huffman decoding via array lookups. **(Right)** Each thread decodes n bytes of encoded exponents. The array *Gaps* stores the bit offset of the first element assigned to each thread, while the array *Block Output Positions* stores the index of the first element for each thread block.

before matrix multiplication, then discarded immediately after use to conserve memory. However, traditional Huffman decoding is inherently sequential, requiring bit-by-bit tree traversal for each element, which is ill-suited for GPUs' parallel architecture. Naively assigning a single thread for decompression leads to poor utilization and high latency. Addressing this bottleneck is essential for practical compressed inference.

In the following paragraphs, we present our solution in detail: a set of hardware-aware algorithmic designs tailored for low-latency decoding of entropy-coded weights in a massively parallel manner. Our approach consists of three key components: ❶ leveraging compact lookup tables that fit within GPU SRAM for efficient, lookup-based decoding, ❷ introducing a two-phase kernel design to coordinate read/write operations for all threads using lightweight auxiliary variables, and ❸ performing decompression at the transformer block level to minimize latency.

2.3.1 Efficient Decoding with Hierarchical Lookup Tables

The traditional approach to decoding Huffman codes involves reading the encoded bitstream bit by bit and traversing the Huffman tree accordingly. However, this method is inefficient on GPUs due to frequent branching and limited parallelism. To enable efficient decoding on GPUs, we adopt a lookup-table-based approach [53].

Assume the maximum Huffman code length is L, and we construct a lookup table LUT of size 2^L. At each index i, LUT stores the decoded exponent whose Huffman code matches the prefix of the binary representation of i. To decode the next exponent, we read the next L bits from the encoded bitstream, interpret them as an index into LUT, and retrieve the corresponding value. To determine how many bits to advance in the stream, we use a secondary lookup table $\mathsf{CodeLengths}$, which maps each exponent to the length of its Huffman code. A detailed example of this decoding process is provided in Section I of the Appendix.

In practice, the value of L can be large. For LLMs, L typically ranges from 24 to 32, resulting in a LUT with up to 2^{32} entries, which cannot fit within GPU SRAM for fast lookups. To address this, we decompose the monolithic LUT into a hierarchy of compact lookup tables [53]. We first partition the Huffman tree into non-overlapping subtrees of height 8. Each subtree corresponds to a compact LUT that decodes 8 bits, requiring only $2^8 = 256$ entries.

Figure 3 shows an example of how a Huffman tree of height 4 can be decomposed into a hierarchy of compact LUTs, each with 4 entries. Because the LUTs are organized hierarchically, some entries must serve as references to other LUTs lower in the hierarchy. We take advantage of the sparsity in 8-bit exponent usage: although 256 values are available, typically only around 40 are used in LLMs (see Figure 9 in the Appendix). We repurpose unused values (specifically, the range 240 to 255) as pointers to other LUTs. These values correspond to extremely large magnitudes ($\pm 2^{113}$ to $\pm 2^{128}$) that do not occur in LLM weights, making them safe for use as internal markers.

We use k to denote the number of compact LUTs. In our experiments, we observe that k ranges from 4 to 8 for the Huffman trees built from BFloat16 exponent values. Combined with CodeLengths, these LUTs occupy at most $(8 + 1) \times 256$ bytes of memory, which easily fits within SRAM and allows for fast repeated lookups.

2.3.2 Two-Phase Kernel and Lightweight Auxiliary Variables

To leverage the parallel processing capabilities of GPUs, we assign each thread to a contiguous, non-overlapping block of encoded exponents consisting of n bytes ($n = 8$ in our experiments). Each thread decodes elements whose Huffman codes begin within its assigned block. Since Huffman codes are variable-length, a thread may need to skip some bits at the start before decoding the first element. Similarly, the last element may span beyond the assigned byte range.

This approach introduces two key challenges: 1. The starting bit position for each thread is unclear due to the variable-length nature of Huffman codes. 2. Except for the first thread, the index of decoded elements is unknown, making it difficult to determine their correct output locations.

To address the first issue, we use a gap array [53] to specify the starting bit offset for each thread. The array Gaps has one entry per thread, where each entry indicates the offset of the first valid Huffman code relative to the thread's assigned starting byte. With a maximum code length of 32 bits, each offset lies in $[0, 31]$ and is stored using only 5 bits.

For the second issue, maintaining an output position for each thread is straightforward but memory-intensive. Each position requires a 32-bit integer, and with tens of thousands of threads per weight matrix, this leads to significant overhead, undermining DFloat11's compression benefits. To reduce this overhead, we store the output position only for the first element of each thread block rather than for every thread. Since each block typically contains hundreds to thousands of threads, this optimization reduces the overhead from one 32-bit integer per thread to one per block, making the memory cost negligible. Figure 3 illustrates how the *gap* and *block-level output position* arrays encode the metadata associated with the encoded exponents.

To support this design, we implement a *two-phase* kernel. In the **first phase**, each thread decodes its assigned block and counts the number of elements, without writing to the HBM. Afterward, threads within a block synchronize to compute per-thread output positions via a prefix sum over the element counts. We use the Blelloch algorithm [4] for this step. In the **second phase**, each thread re-decodes the same block, this time writing decoded values to a write buffer in SRAM at the calculated positions. To avoid redundant global memory access, the encoded exponents are loaded into SRAM before the first pass. Once all decoded exponents are written to SRAM, a single batch of coalesced writes is issued to HBM. Pseudocode for the two-phase kernel is provided in Algorithm 1 of the Appendix.

2.3.3 Transformer-Block-Level Decompression

We now have a complete recipe for decompressing entropy-coded exponents in a massively parallel manner. During inference, the LLM weights stored in DFloat11 format, along with auxiliary variables (the thread-level gap array and block-level output position array), reside entirely in GPU memory. When a weight matrix is needed for matrix multiplication, it is decompressed on-the-fly into the original BFloat16 format. Once the matrix multiplication is complete, the BFloat16 matrix is immediately discarded to conserve GPU memory.

In practice, decompressing a single weight matrix often underutilizes GPU resources due to its relatively small size. As the matrix size increases, decompression throughput improves. Figure 7 illustrates this trend, showing how DFloat11 decompression scales with matrix size. To capitalize on this, we propose batching the decompression of multiple matrices together to improve throughput and hide latency. Specifically, we decompress all DFloat11 weight matrices within a transformer block as a single batch. This batched decompression occurs right before the forward pass of the transformer block. We also compress the token embedding and language modeling head of LLMs. Since these matrices are large enough to saturate GPU resources, batching their decompression is unnecessary.

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

3 Experiments

We empirically evaluate the effectiveness of DF11 compression and its GPU inference efficiency. A range of recent LLMs and DMs are compressed from their original BFloat16 format into DF11, and we report the resulting compression ratios. We then compare the inference performance of DF11-compressed models against their uncompressed counterparts across different GPUs, followed by an ablation study to analyze the impact of compression.

Software and Hardware We implement the DF11 decompression kernel in CUDA and C++, and integrate it into the HuggingFace Transformers [48] inference framework. We evaluate the inference efficiency of our DF11 models against the original BF16 counterparts. We use the HuggingFace Accelerate framework to support CPU offloading and multi-GPU inference. To assess the performance of the DF11 kernel across different hardware configurations, we run experiments on multiple machines with varying GPU and CPU setups. The hardware specifications for all experimental machines are provided in Table 4 in the Appendix.

3.1 Results

DF11 compresses models to 70% size. Table 1 presents the compression factors of DF11 for a wide selection of recent LLMs and DMs. Specifically, we apply compression to all weight matrices and token embeddings in LLMs and all weight matrices in the transformer blocks of DMs. The models we compress include Llama 3.1/3.3 [20], Qwen 3 [54], Mistral Nemo/Small [44, 45], Phi 4 [1], DeepSeek R1 Distilled [21], Stable Diffusion 3.5 [2], FLUX.1 [32]. DF11 achieves approximately 70% compression across all models, corresponding to an effective bit width of around 11 bits.

Accuracy and perplexity evaluations confirm DF11 compression is lossless. We verify the lossless property of DF11 compression through a series of accuracy and perplexity evaluations on standard benchmarks. Evaluations are conducted using `lm_evaluation_harness` [18], reporting accuracy on MMLU [24] and TruthfulQA [38], and word-level perplexity on WikiText [41] and C4 [42]. The results are shown in Table 2. As demonstrated, the compressed model achieves identical accuracy and perplexity to the original BF16 counterpart. We also present the text-to-image

Table 3: Comparison of peak GPU memory usage and text-to-image generation time for diffusion transformers in BF16 and DF11, using a single A5000 GPU.

	Peak GPU Memory (GB)		Generation Time (s)	
Model	BF16	DF11 (Ours)	BF16	DF11 (Ours)
Stable Diffusion 3.5 Large	16.44	11.78	66.36 ± 0.13	69.08 ± 0.11
FLUX.1 dev	23.15	16.72	74.41 ± 0.15	78.53 ± 0.18

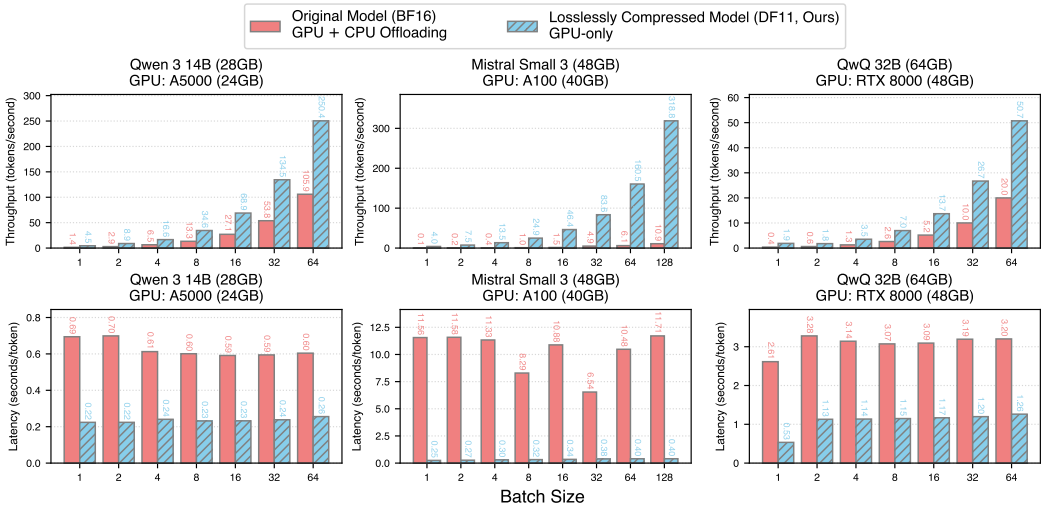

Figure 4: Comparison of throughput **(top row)** and latency **(bottom row)** for token decoding using the original BF16 models and their DF11-compressed counterparts. Portions of the BF16 models are offloaded to the CPU due to GPU memory constraints.

generation results of BF16 and DF11 Stable Diffusion 3.5 Large model in Appendix J. Given the same random seed and text prompt, the image generated are pixel-wise identical with the original model.

DF11 outperforms CPU offloading in inference efficiency. We compare the inference performance of DF11 and BF16 models across various hardware platforms. Due to memory constraints, BF16 models exceed the capacity of a single GPU and require partial CPU offloading, while DF11 models fit entirely within GPU memory. For fair comparison, we retain most computation on the GPU for BF16 models and offload only necessary components. Latency and throughput are measured after a 100-token warm-up run, followed by decoding 100 tokens from an empty prompt across varying batch sizes. Each configuration is run five times, and we report the average results. As shown in Figure 4, DF11 consistently outperforms BF16 with CPU offloading, achieving 2.31–46.24× lower latency or higher throughput. Multi-GPU comparisons are shown in Figure 10 in the Appendix.

DF11 reduces memory usage for diffusion transformers with minimal latency impact. We assess the impact of DF11 compression on diffusion transformer models by measuring peak GPU memory usage and text-to-image generation latency for an 1024×1024 image across five runs. Neither the BF16 nor DF11 models employ CPU offloading. As shown in Table 3, DF11 reduces memory consumption by 28.3% for Stable Diffusion 3.5 and 27.8% for FLUX.1. The relative increase in latency is small: 4.1% for Stable Diffusion and 5.5% for FLUX.1.

DF11 memory savings enable longer generation lengths. DF11 compression not only can reduce the number of GPUs needed for inference but can also support longer generation under the same VRAM budget. During decoding, the KV cache grows linearly with the number of tokens and quickly becomes a memory bottleneck. Figure 5 shows GPU memory usage for DF11 and BF16 models with batch size 1 as token count increases. DF11 allows 5.70 to 14.86× more tokens to be decoded before reaching memory limits.

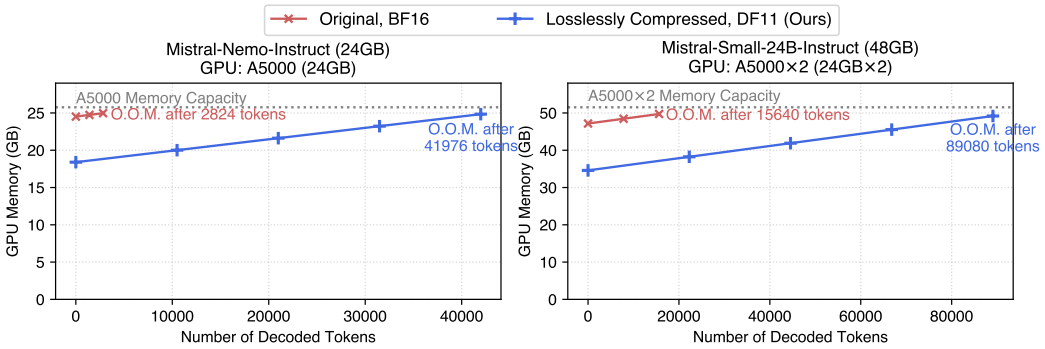

Figure 5: Comparison of GPU memory consumption between BF16 models and DF11 counterparts. The DF11 models support 5.70–14.86× longer context lengths by allowing more GPU memory to be used for storing the KV cache. "O.O.M." means out of memory.

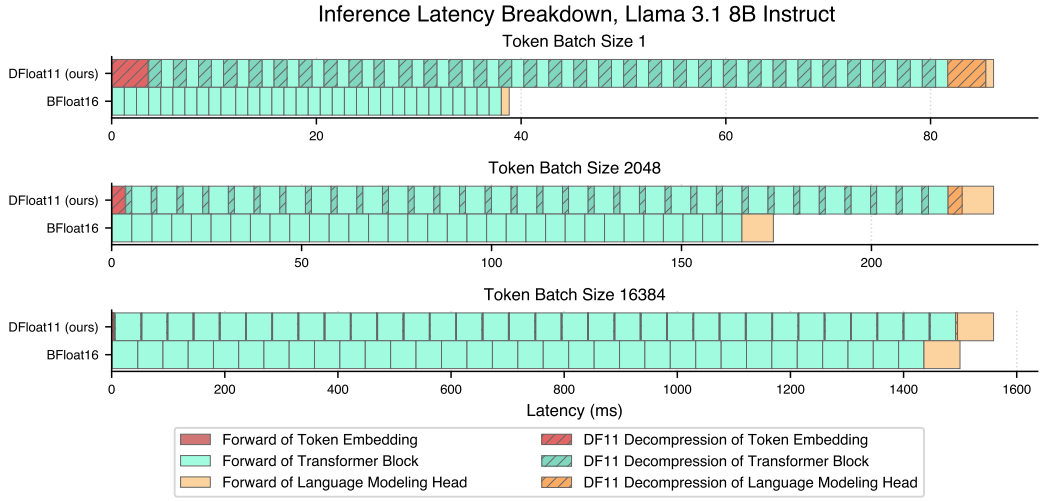

Figure 6: Comparison of latency breakdown for DF11 and BF16 Llama 3.1 8B Instruct during GPU inference for different token batch sizes, using one A100-40GB GPU.

3.2 Ablation Study

Latency breakdown shows decompression overhead is amortized at larger batch sizes. We analyze the latency of *Llama 3.1 8B Instruct* in BF16 and DF11 formats across varying token batch sizes on an A100-40GB GPU. For each setting, we measure the average latency of each component over 10 runs, as shown in Figure 6. DF11 introduces additional latency from decompressing the token embedding, transformer blocks, and language modeling head. This overhead is constant and independent of batch size, so increasing the token batch size effectively amortizes the cost.

DF11 decompression is significantly faster than CPU-to-GPU transfer and nvCOMP ANS. We compare DF11 decompression latency and throughput with two baselines: CPU-to-GPU weight transfer and ANS decompression [12] from NVIDIA's nvCOMP [6], using sliced weight matrices from the Llama 3.1 8B Instruct language modeling head. As shown in Figure 7, DF11 achieves up to 34.95× higher throughput than CPU transfer and up to 20.97× faster decompression than nvCOMP. DF11 also offers a better compression ratio (68%) compared to nvCOMP (79%). Moreover, DF11 decompression throughput improves with larger matrix sizes due to better GPU utilization.

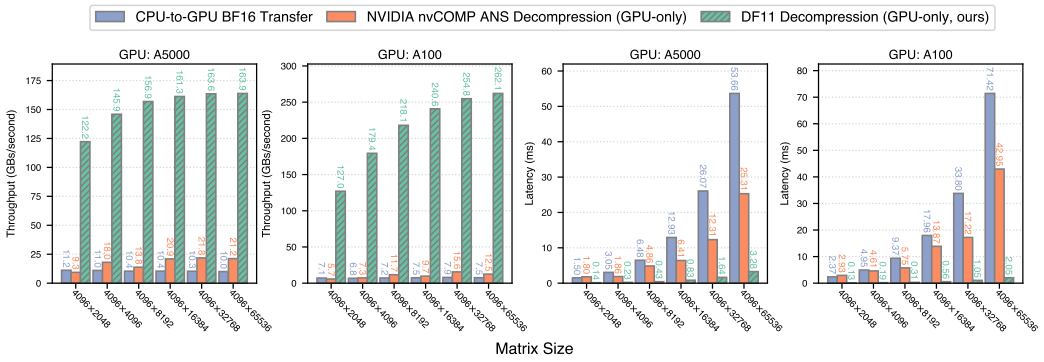

Figure 7: Throughput **(left two)** and latency **(right two)** comparisons between transferring BF16 matrices from CPU to GPU and decompressing the same matrices on GPU using the NVIDIA nvCOMP ANS library and our proposed DF11 kernel, across matrix sizes and GPU types.

4 Related Works

Data Formats for Model Weights Full-precision model weights are typically stored in formats such as BF16, FP16, or FP32. Several works have proposed 4-bit compressed formats, including FP4, INT4, NF4 (NormalFloat) [9], AF4 (AbnormalFloat) [58], and SF4 (Student Float) [11], which represent each parameter with 4 bits. Unlike these lossy formats, the proposed DF11 format compresses weights losslessly.

Lossless Model Compression While lossy compression methods such as pruning [14] and quantization [37, 15] are well-studied, lossless compression remains less explored. Four prior works have addressed this area. *Deep Compression* [22] applied Huffman coding [28] to quantized CNNs, achieving 22% additional compression. *ZipNN* [25] extended this approach to language models with improved compression over classical methods. However, both techniques target storage efficiency and do not support inference-time gains. *NeuZip* [23] is the only prior work supporting GPU inference. It uses Asymmetric Numeral Systems (ANS) with layer-wise decompression and relies on NVIDIA's nvCOMP for GPU-based operations. nvCOMP is no longer open source, and its binary-only distribution limits adoption. Moreover, as shown in Figure 7, nvCOMP ANS incurs higher latency and lower throughput compared to our DFloat11 kernel. *Huff-LLM* [59] is designed for FPGA-like hardware and is not applicable to GPUs. Additional discussion of related formats is presented in Appendix B.

5 Conclusion

We introduce *Dynamic-Length Float* (DFloat11), a lossless compression framework designed for efficient GPU inference of BFloat16 models, including both large language models (LLMs) and diffusion models (DMs). DFloat11 exploits the information redundancy inherent in foundation model weights through entropy-coded, dynamic-length encoding, achieving compression rates close to the information-theoretic limit. To enable efficient deployment, we develop hardware-aware algorithms that support high-speed inference directly on compressed weights. Extensive experiments demonstrate that DFloat11 significantly reduces GPU memory requirements for LLMs and DMs, allowing for longer generation lengths, while maintaining bit-exact accuracy and incurring only negligible decompression overhead.

Acknowledgements

This work was supported by National Science Foundation SHF-2211815 and Ken Kennedy Institute Cluster Grants. Additionally, Henry and Xia are supported by ITE-2429680, IIS-2310260, and US Department of Transportation (USDOT) Tier-1 University Transportation Center (UTC) Transportation Cybersecurity Center for Advanced Research and Education (CYBER-CARE) grant #69A3552348332. Mohsen and Vipin are supported by OAC-2320952, OAC-2112606, and OAC-2117439. The views and conclusions in this paper are those of the authors and do not represent the views of any funding or supporting agencies.

References

[1] Marah Abdin, Jyoti Aneja, Harkirat Behl, Sébastien Bubeck, Ronen Eldan, Suriya Gunasekar, Michael Harrison, Russell J Hewett, Mojan Javaheripi, Piero Kauffmann, et al. Phi-4 technical report. *arXiv preprint arXiv:2412.08905*, 2024.

[2] Stability AI. Introducing stable diffusion 3.5. `https://stability.ai/news/introducing-stable-diffusion-3-5`, October 2024. Accessed: May 15, 2025.

[3] Anonymous. FAFO: Lossy KV cache compression for lossless inference acceleration via draftless fumble decoding. In *Submitted to The Fourteenth International Conference on Learning Representations*, 2025. under review.

[4] Guy E Blelloch. Scans as primitive parallel operations. *IEEE Transactions on computers*, 38(11):1526–1538, 1989.

[5] 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. *arXiv preprint arXiv:2110.14168*, 2021.

[6] NVIDIA Corporation. nvCOMP: Gpu-accelerated compression and decompression library. `https://developer.nvidia.com/nvcomp`, 2025. Accessed: April 11, 2025.

[7] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in neural information processing systems*, 35:16344–16359, 2022.

[8] 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.

[9] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. Qlora: Efficient finetuning of quantized llms. *Advances in neural information processing systems*, 36:10088–10115, 2023.

[10] P. Deutsch and J.-L. Gailly. Rfc1950: Zlib compressed data format specification version 3.3, 1996.

[11] Jordan Dotzel, Yuzong Chen, Bahaa Kotb, Sushma Prasad, Gang Wu, Sheng Li, Mohamed S Abdelfattah, and Zhiru Zhang. Learning from students: Applying t-distributions to explore accurate and efficient formats for llms. In *International Conference on Machine Learning*, pages 11573–11591. PMLR, 2024.

[12] Jarek Duda. Asymmetric numeral systems: entropy coding combining speed of huffman coding with compression rate of arithmetic coding. *arXiv preprint arXiv:1311.2540*, 2013.

[13] Abhinav Dutta, Sanjeev Krishnan, Nipun Kwatra, and Ramachandran Ramjee. Accuracy is not all you need. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024.

[14] Elias Frantar and Dan Alistarh. Sparsegpt: Massive language models can be accurately pruned in one-shot. In *International Conference on Machine Learning*, pages 10323–10337. PMLR, 2023.

[15] 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.

[16] Yichao Fu, Peter Bailis, Ion Stoica, and Hao Zhang. Break the sequential dependency of LLM inference using lookahead decoding. In *Forty-first International Conference on Machine Learning*, 2024.

[17] Kazuki Fujii, Taishi Nakamura, and Rio Yokota. Balancing speed and stability: The trade-offs of fp8 vs. bf16 training in llms. *arXiv preprint arXiv:2411.08719*, 2024.

[18] 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, 07 2024.

[19] Ruihao Gong, Yang Yong, Shiqiao Gu, Yushi Huang, Chengtao Lv, Yunchen Zhang, Xianglong Liu, and Dacheng Tao. Llmc: Benchmarking large language model quantization with a versatile compression toolkit. *arXiv preprint arXiv:2405.06001*, 2024.

[20] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

[21] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

[22] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

[23] Yongchang Hao, Yanshuai Cao, and Lili Mou. Neuzip: Memory-efficient training and inference with dynamic compression of neural networks. *arXiv preprint arXiv:2410.20650*, 2024.

[24] Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. Measuring massive multitask language understanding. In *International Conference on Learning Representations*.

[25] Moshik Hershcovitch, Andrew Wood, Leshem Choshen, Guy Girmonsky, Roy Leibovitz, Ilias Ennmouri, Michal Malka, Peter Chin, Swaminathan Sundararaman, and Danny Harnik. Zipnn: Lossless compression for ai models. *arXiv preprint arXiv:2411.05239*, 2024.

[26] Coleman Richard Charles Hooper, Sehoon Kim, Hiva Mohammadzadeh, Michael W. Mahoney, Sophia Shao, Kurt Keutzer, and Amir Gholami. KVQuant: Towards 10 million context length LLM inference with KV cache quantization. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024.

[27] Edward J Hu, yelong shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*, 2022.

[28] David A Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[29] Renren Jin, Jiangcun Du, Wuwei Huang, Wei Liu, Jian Luan, Bin Wang, and Deyi Xiong. A comprehensive evaluation of quantization strategies for large language models. In *Findings of the Association for Computational Linguistics ACL 2024*, pages 12186–12215, 2024.

[30] Dhiraj Kalamkar, Dheevatsa Mudigere, Naveen Mellempudi, Dipankar Das, Kunal Banerjee, Sasikanth Avancha, Dharma Teja Vooturi, Nataraj Jammalamadaka, Jianyu Huang, Hector Yuen, et al. A study of bfloat16 for deep learning training. *arXiv preprint arXiv:1905.12322*, 2019.

[31] Artyom Kharinaev, Viktor Moskvoretskii, Egor Shvetsov, Kseniia Studenikina, Bykov Mikhail, and Evgeny Burnaev. Investigating the impact of quantization methods on the safety and reliability of large language models. *arXiv preprint arXiv:2502.15799*, 2025.

[32] Black Forest Labs. Flux. `https://github.com/black-forest-labs/flux`, 2024.

[33] G. G. Langdon. An introduction to arithmetic coding. *IBM Journal of Research and Development*, 28(2):135–149, 1984.

[34] Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding. In *Fortieth International Conference on Machine Learning*, 2023.

[35] Shiyao Li, Xuefei Ning, Luning Wang, Tengxuan Liu, Xiangsheng Shi, Shengen Yan, Guohao Dai, Huazhong Yang, and Yu Wang. Evaluating quantized large language models. In *International Conference on Machine Learning*, pages 28480–28524. PMLR, 2024.

[36] Xiuyu Li, Yijiang Liu, Long Lian, Huanrui Yang, Zhen Dong, Daniel Kang, Shanghang Zhang, and Kurt Keutzer. Q-diffusion: Quantizing diffusion models. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 17535–17545, 2023.

[37] Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. Awq: Activation-aware weight quantization for on-device llm compression and acceleration. *Proceedings of Machine Learning and Systems*, 6:87–100, 2024.

[38] Stephanie Lin, Jacob Hilton, and Owain Evans. Truthfulqa: Measuring how models mimic human falsehoods. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 3214–3252, 2022.

[39] Ruikang Liu, Yuxuan Sun, Manyi Zhang, Haoli Bai, Xianzhi Yu, Tiezheng Yu, Chun Yuan, and Lu Hou. Quantization hurts reasoning? an empirical study on quantized reasoning models. *arXiv preprint arXiv:2504.04823*, 2025.

[40] Zirui Liu, Jiayi Yuan, Hongye Jin, Shaochen Zhong, Zhaozhuo Xu, Vladimir Braverman, Beidi Chen, and Xia Hu. KIVI: A tuning-free asymmetric 2bit quantization for KV cache. In *Forty-first International Conference on Machine Learning*, 2024.

[41] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models. In *International Conference on Learning Representations*, 2017.

[42] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, 21(140):1–67, 2020.

[43] Yang Sui, Yanyu Li, Anil Kag, Yerlan Idelbayev, Junli Cao, Ju Hu, Dhritiman Sagar, Bo Yuan, Sergey Tulyakov, and Jian Ren. Bitsfusion: 1.99 bits weight quantization of diffusion model. *arXiv preprint arXiv:2406.04333*, 2024.

[44] Mistral AI Team. Mistral NeMo. `https://mistral.ai/news/mistral-nemo`, July 2024.

[45] Mistral AI Team. Mistral Small 3. `https://mistral.ai/news/mistral-small-3`, January 2025.

[46] Qwen Team. Qwen3: Think deeper, act faster, April 2025.

[47] Zhuang Wang, Zhen Jia, Shuai Zhang, Zhen Zhang, Mason Fu, T. S. Eugene Ng, and Yida Wang. Gemini: Fast failure recovery in distributed training with in-memory checkpoints. 2023.

[48] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 conference on empirical methods in natural language processing: system demonstrations*, pages 38–45, 2020.

[49] Heming Xia, Tao Ge, Peiyi Wang, Si-Qing Chen, Furu Wei, and Zhifang Sui. Speculative decoding: Exploiting speculative execution for accelerating seq2seq generation. *arXiv preprint arXiv:2203.16487*, 2022.

[50] Heming Xia, Zhe Yang, Qingxiu Dong, Peiyi Wang, Yongqi Li, Tao Ge, Tianyu Liu, Wenjie Li, and Zhifang Sui. Unlocking efficiency in large language model inference: A comprehensive survey of speculative decoding. *arXiv preprint arXiv:2401.07851*, 2024.

[51] Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. Smoothquant: Accurate and efficient post-training quantization for large language models. In *International Conference on Machine Learning*, pages 38087–38099. PMLR, 2023.

[52] Mengwei Xu, Wangsong Yin, Dongqi Cai, Rongjie Yi, Daliang Xu, Qipeng Wang, Bingyang Wu, Yihao Zhao, Chen Yang, Shihe Wang, et al. A survey of resource-efficient llm and multimodal foundation models. *arXiv preprint arXiv:2401.08092*, 2024.

[53] Naoya Yamamoto, Koji Nakano, Yasuaki Ito, Daisuke Takafuji, Akihiko Kasagi, and Tsuguchika Tabaru. Huffman coding with gap arrays for gpu acceleration. In *Proceedings of the 49th International Conference on Parallel Processing*, pages 1–11, 2020.

[54] An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, et al. Qwen2. 5 technical report. *arXiv preprint arXiv:2412.15115*, 2024.

[55] Ge Yang, Changyi He, Jinyang Guo, Jianyu Wu, Yifu Ding, Aishan Liu, Haotong Qin, Pengliang Ji, and Xianglong Liu. LLMCBench: Benchmarking large language model compression for efficient deployment. In *The Thirty-eight Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2024.

[56] Jingfeng Yang, Haongye Jin, Ruixiang Tang, Xiaotian Han, Qizhang Feng, Haoming Jiang, Shaochen Zhong, Bing Yin, and Xia Hu. Harnessing the power of llms in practice: A survey on chatgpt and beyond. 2024.

[57] Ling Yang, Zhilong Zhang, Yang Song, Shenda Hong, Runsheng Xu, Yue Zhao, Wentao Zhang, Bin Cui, and Ming-Hsuan Yang. Diffusion models: A comprehensive survey of methods and applications. *ACM Computing Surveys*, 56(4):1–39, 2023.

[58] Davis Yoshida. Nf4 isn't information theoretically optimal (and that's good). *arXiv preprint arXiv:2306.06965*, 2023.

[59] Patrick Yubeaton, Tareq Mahmoud, Shehab Naga, Pooria Taheri, Tianhua Xia, Arun George, Yasmein Khalil, Sai Qian Zhang, Siddharth Joshi, Chinmay Hegde, et al. Huff-llm: End-to-end lossless compression for efficient llm inference. *arXiv preprint arXiv:2502.00922*, 2025.

[60] Haochen Zhang, Junze Yin, Guanchu Wang, Zirui Liu, Lin Yang, Tianyi Zhang, Anshumali Shrivastava, and Vladimir Braverman. Breaking the frozen subspace: Importance sampling for low-rank optimization in LLM pretraining. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025.

[61] Tianyi Zhang and Anshumali Shrivastava. Leanquant: Accurate and scalable large language model quantization with loss-error-aware grid. In *The Thirteenth International Conference on Learning Representations*, 2025.

[62] Tianyi Zhang, Junda Su, Aditya Desai, Oscar Wu, Zhaozhuo Xu, and Anshumali Shrivastava. Sketch to adapt: Fine-tunable sketches for efficient LLM adaptation. In *Forty-second International Conference on Machine Learning*, 2025.

[63] Tianyi Zhang, Jonah Wonkyu Yi, Zhaozhuo Xu, and Anshumali Shrivastava. KV cache is 1 bit per channel: Efficient large language model inference with coupled quantization. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024.

[64] Tianyi Zhang, Jonah Wonkyu Yi, Bowen Yao, Zhaozhuo Xu, and Anshumali Shrivastava. NoMAD-attention: Efficient LLM inference on CPUs through multiply-add-free attention. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024.

[65] Jiawei Zhao, Zhenyu Zhang, Beidi Chen, Zhangyang Wang, Anima Anandkumar, and Yuandong Tian. Galore: Memory-efficient LLM training by gradient low-rank projection. In *Forty-first International Conference on Machine Learning*, 2024.

Appendix

A Discussion: Is Quantization a Universal Solution?

Much of the motivation behind our work lies in understanding **whether lossless compression of large-scale models such as LLMs, which preserves 100% identical output behavior compared to the original uncompressed model, is a practical direction worthy of further study.** Specifically, how does DFloat11, which compresses LLMs to approximately 11 bits, compare to widely used lossy quantization techniques [15, 37], where models are typically reduced to even lower bit-widths (e.g., 8-bit or 4-bit)?

The answer is far more nuanced than a simple "Yes/No" or a one-size-fits-all judgment about which approach is better. For instance, existing benchmark studies like [19, 55, 29] often suggest that 8-bit (weight-only or not) quantization is a relatively "safe" compression scheme. Although technically lossy, 8-bit models can often maintain strong task performance across a range of standard benchmarks. However, we must note these benchmarks typically focus on a narrow set of tasks (e.g., WikiText2 perplexity, MMLU, Commonsense Reasoning), and thus fail to offer a comprehensive view of real-world LLM usage, especially from the perspective of end-users.

That being said, the argument that "current benchmarks fail to capture the performance gap between 8-bit compressed and 16-bit uncompressed models" is itself constrained by the limitations of the current benchmarking landscape, making it difficult to produce abundant supporting evidence. Nonetheless, some reports have begun to highlight such gaps. For example, human evaluations on LLM Arena[1] show a notable performance drop between Llama-3.1-405B-Instruct [20] and its W8A8 counterpart (Llama-3.1-405B-Instruct-FP8), particularly under coding (1293 vs. 1277) and long-query (1282 vs. 1275) tasks. Similarly, quantizing DeepSeek-R1-Distill-Llama-70B [21] from 16 bits to 8 bits results in a 23.7% drop on GPQA (from 9.51% to 7.25%).[2] Furthermore, reasoning, a core capability of modern LLMs, appears especially sensitive to compression loss. Recent benchmark [39] reveals that quantizing DeepSeek-R1-Distill-Qwen-1.5B with 8-bit SmoothQuant [51] (for weight, attention, and KV cache) leads to an average 9.09% drop in reasoning tasks (48.82% to 44.29%) across datasets like AIME, MATH-500, GPQA-Diamond, and LiveCodeBench. We leave more evidence exploring the performance gap between 8-bit quantized and uncompressed model in Appendix H.

Although the broader question: "Which specific task, on which model, using which quantization technique, under what conditions, will lead to a noticeable drop compared to FP16/BF16?" is likely to remain open-ended simply due to the sheer amount of potential combinations. It is fair to say that lossy quantization introduces complexities that some end-users would prefer to avoid, since it creates uncontrolled variables that must be empirically stress-tested for each deployment scenario.

To eliminate this burden, DFloat11 offers a compelling alternative: delivering **100% identical performance to the original model, while consuming only $\sim$70% of the memory footprint with throughput benefits**, which is a unique and practical offering for resource-constrained deployment settings.

B Extended Related Works

Data Formats for Model Weights LLM weights are typically stored in compact floating-point formats such as FP16 or BFloat16 (officially stylized as *bfloat16*[3]). FP16 allocates 1 sign bit, 5 exponent bits, and 10 mantissa bits, whereas BFloat16 uses 1 sign bit, 8 exponent bits, and 7 mantissa bits. Compared to FP16, BFloat16 offers a wider dynamic range at the cost of precision, which improves numerical stability and mitigates overflow issues during training [17, 30].

Compressed data formats typically aim for lower bit-widths. For example, FP8—which comes in both `E4M3` (4 exponent bits, 3 mantissa bits, plus 1 sign bit) and `E5M2` configurations—has seen reasonable adoption in LLM training and development. Integer formats like INT8 have also been well explored, as in `LLM.int8()` [8] and its following works. Formats with a stronger emphasis on efficiency, such

[1] `https://x.com/lmarena_ai/status/1835760196758728898`

[2] `https://huggingface.co/RedHatAI/DeepSeek-R1-Distill-Llama-70B-quantized.w8a8`

[3] `https://cloud.google.com/blog/products/ai-machine-learning/bfloat16-the-secret-to-high-performance-on-cloud-tpus`

as FP4, INT4, NF4 [9], and AF4 [58], use only 4 bits. In this work, we primarily focus on formats with $\geq$8 bits, as benchmark literature [55, 19, 39] often suggests that 8-bit quantization results in negligible performance drop—though we show in Section A that this claim is likely skewed due to evaluation selectiveness and benchmark limitations.

Lossless Model Compression While lossy model compression techniques such as pruning and quantization [14, 37, 15] have received widespread attention, lossless model compression remains a relatively underexplored area. Upon careful investigation, we identified roughly four prior works that have made meaningful efforts in this space. *Deep Compression* [22] is a foundational work, applying Huffman coding [28] to quantized CNN models and achieving an additional $\sim$22% compression gain for model checkpoints. *ZipNN* [25] extended this idea to language models, comparing its results to classic lossless compression tools such as zlib [10] and zstd[4] and demonstrated superior compression gains. However, this line of work—including their industry counterparts, such as ezm7[5]—is limited in that its efficiency gains only apply to storage (reducing the size of model checkpoints) but offer no benefits during inference. While such storage savings are meaningful in large-scale training settings—where frequent snapshotting and checkpoint rollbacks are needed [47]—they have limited impact for everyday LLM end-users. Model downloading is typically a one-time cost, so even if a model checkpoint is compressed by 50%, it only cuts the download time at most by half, presumably over the model's entire lifecycle of deployment. Furthermore, checkpoints are usually stored on disk, where terabytes of capacity are easily available, making up a much looser constraint compared to GPU HBM (High Bandwidth Memory); one of the main resource constraints during inference.

We argue that a lossless compression technique would be substantially more impactful if it could deliver efficiency gains during inference—particularly on GPU-based systems, which is the default setup for LLM serving. In this context, *NeuZip* [23] is the only prior work we identify that supports GPU inference. NeuZip applies entropy encoding with layer-wise decompression to maintain a reduced memory footprint throughout serving. However, it is built on NVIDIA's nvCOMP: *"a high-speed data compression and decompression library optimized for NVIDIA GPUs"*.[6] Unfortunately, nvCOMP is no longer open-source (only binary executables are available), which hinders future research. Moreover, we empirically find that nvCOMP's inference throughput and latency are significantly worse than our proposed DFloat11 kernel, resulting in a pipeline that trades memory efficiency for substantial inference overhead (see Figure 7).

Another work referencing NeuZip is *Huff-LLM* [59], which also aims to reduce memory costs while maintaining efficient inference. However, its contributions are specific to FPGA-like architectures and do not apply to GPUs. To the best of our knowledge, the DFloat data format we presented (and its respective kernel support in DFloat11) shall serve as the only GPU-inference-friendly data format with lossless compression benefits.

Efficient LLM Inference LLMs are computationally intensive and resource-demanding, making the efficiency of LLM inference a key research focus [52]. FlashAttention [7] accelerates exact attention computation on GPUs through kernel fusion, while NoMAD Attention [64] speeds up attention on CPUs using in-register lookups. Model compression is another effective strategy to reduce resource requirements for serving LLMs and diffusion models. Quantization methods such as GPTQ [15], AWQ [37], SmoothQuant [51], LeanQuant [61], CQ [63], KVQuant [26], and KIVI [40] lower memory usage and enhance efficiency by compressing model weights, activations, or KV cache. Compression is also applied in fine-tuning: methods like LoRA [27], QLoRA [9], and SketchTune [62] compress model weight deltas, whereas GaLore [65] and SARA [60] compress optimizer states during training. One additional line of work relevant to efficient LLM inference would be *lossless efficient decoding*, where paradigms such as *speculative decoding* [49, 34, 50] and *n-gram candidate decoding* [16, 3] offer lossless generation quality with improved latency. DFloat11 mainly differs from these works in that it provides substantial savings in memory footprint while maintaining lossless generation quality, whereas most—if not all—lossless efficient decoding methods require memory consumption equal to or greater than that of the original model.

[4] https://github.com/facebook/zstd

[5] https://github.com/liuliu/s4nnc/pull/11 https://encode.su/threads/4067-Good-Compressors-for-16-bit-floats

[6] https://developer.nvidia.com/nvcomp

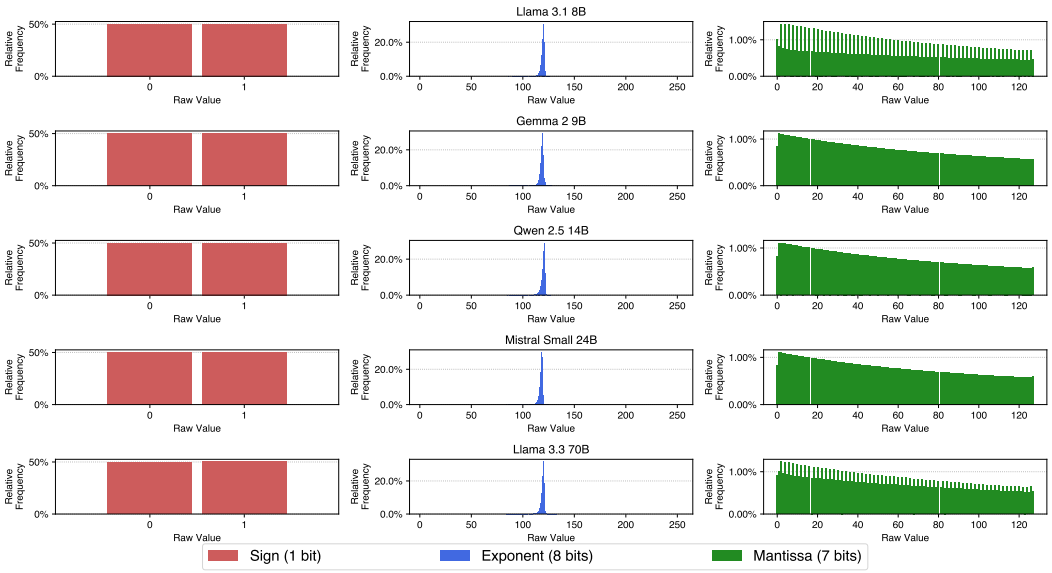

Figure 8: Relative frequency distribution of sign, exponent, and mantissa values in the BFloat16 weights of all linear projection layers across various LLMs.

C Frequency Distribution of BFloat16 Values

Figure 8 presents the frequency distribution for distinct values of sign, exponent, and mantissa bits in the BFloat16 weights of LLMs. Figure 9 shows the sorted frequency of exponent values of LLM weights.

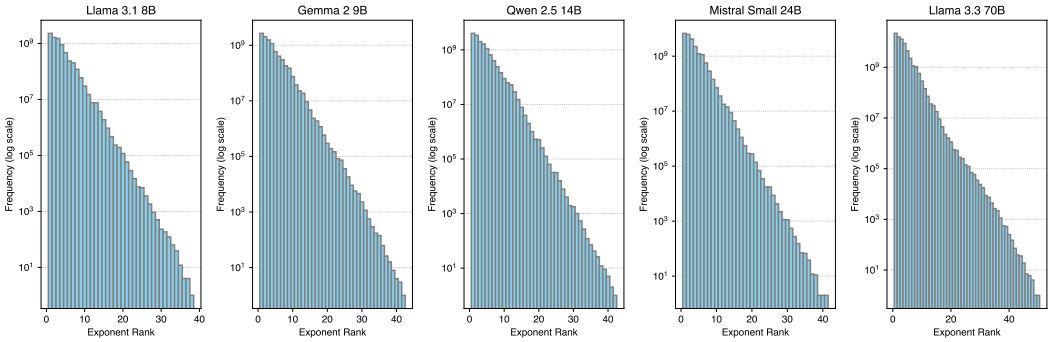

Figure 9: Distribution of BFloat16 exponent values across various models. The frequency of exponent values (shown in log scale) decays rapidly with exponent rank.

D Pseudo-code of the GPU kernel for DFloat11 Decompression

Algorithm 1 presents the pseudo-code of the two-phase GPU kernel for decompressing DFloat11 to BFloat16.

Table 4: System specifications of servers used for experiments.

	GPU	GPU Memory	CPU	CPU Memory
Server 1	NVIDIA RTX A5000	24564MiB	AMD EPYC 7513 32-Core	504GB
Server 2	NVIDIA A100	40960MiB	AMD EPYC 7742 64-Core	1.48TB
Server 3	NVIDIA Quadro RTX 8000	49152MiB	AMD EPYC 7742 64-Core	1.48TB

Algorithm 1 GPU kernel for decompressing DFloat11 to BFloat16

procedure DF11ToBF16
require:
- $\mathsf{EncodedExponent}, \mathsf{PackedSignMantissa}$: byte arrays
- $\mathsf{LUT}_1, \ldots, \mathsf{LUT}_k, \mathsf{CodeLengths}$: 8-bit unsigned integer arrays of size 256
- Gaps: 5-bit unsigned integer array (one entry per thread in each block)
- $\mathsf{BlockOutputPos}$: 32-bit unsigned integer array (one entry per block)
- $\mathsf{Outputs}$: BFloat16 array, for storing results
- B, T, n, k: the number of thread blocks, number of threads, number of bytes processed by each thread, number of compact LUTs, respectively

Divide $\mathsf{EncodedExponent}$ into chunks: $\mathsf{EncodedExponent}_1, \ldots, \mathsf{EncodedExponent}_B$ of size nT bytes each
for all $b \leftarrow 1, \ldots, B$ **(in parallel across blocks) do**
  Load $\mathsf{EncodedExponent}_b$ into SRAM
  Divide $\mathsf{EncodedExponent}_b$ into chunks: $\mathsf{EncodedExponent}_{b,1}, \ldots, \mathsf{EncodedExponent}_{b,T}$ of size n bytes each
  Load $\mathsf{LUT}_1, \ldots, \mathsf{LUT}_k, \mathsf{CodeLengths}$ into SRAM
  Initialize integer arrays $\mathsf{NumElements}[1 \ldots T], \mathsf{ThreadOutputPos}[1 \ldots T]$ with all 0s
  Initialize BFloat16 write buffer $\mathsf{WriteBuffer}$ in SRAM
  for all $t \leftarrow 1, \ldots, T$ **(in parallel across threads) do**
▷ Phase 1: Each thread determines its initial output position
    $\mathrm{BitOffset} \leftarrow \mathsf{Gaps}[bT + t]$
    while $\mathrm{BitOffset} < 8n$ **do**
      Read the next 4 bytes of $\mathsf{EncodedExponent}_{b,t}$, starting from the $\mathrm{BitOffset}$-th bit, into $\mathrm{Byte}_{1\ldots4}$
      $i \leftarrow 1$
      $\mathrm{Exponent} \leftarrow \mathsf{LUT}_1[\mathrm{Byte}_i]$
      while $\mathrm{Exponent} \geq 240$ **do**
        ▷ $\mathrm{Exponent} \geq 240$ means that it is a pointer to the next LUT
        $i \leftarrow i + 1$
        $\mathrm{Exponent} \leftarrow \mathsf{LUT}_{(257-\mathrm{Exponent})}[\mathrm{Byte}_i]$
      end while
      $\mathrm{BitOffset} \leftarrow \mathrm{BitOffset} + \mathsf{CodeLengths}[\mathrm{Exponent}]$
      $\mathsf{NumElements}[t] \leftarrow \mathsf{NumElements}[t] + 1$
    end while
    Thread Synchronization Barrier
    ▷ Compute prefix-sum using Blelloch's Algorithm:
    $\mathsf{ThreadOutputPos}[t] \leftarrow \mathsf{BlockOutputPos}[b] + \sum_{i=1}^{t-1} \mathsf{NumElements}[i]$
▷ Phase 2: Writing decoded BFloat16s to the appropriate positions
    $\mathrm{BitOffset} \leftarrow \mathsf{Gaps}[bT + t]$
    while $\mathrm{BitOffset} < 8n$ **do**
      Read the next 4 bytes of $\mathsf{EncodedExponent}_{b,t}$, starting from the $\mathrm{BitOffset}$-th bit, into $\mathrm{Byte}_{1\ldots4}$
      $i \leftarrow 1$
      $\mathrm{Exponent} \leftarrow \mathsf{LUT}_1[\mathrm{Byte}_i]$
      while $\mathrm{Exponent} \geq 240$ **do**
        ▷ $\mathrm{Exponent} \geq 240$ means that it is a pointer to the next LUT
        $i \leftarrow i + 1$
        $\mathrm{Exponent} \leftarrow \mathsf{LUT}_{(257-\mathrm{Exponent})}[\mathrm{Byte}_i]$
      end while
      $\mathrm{Byte} \leftarrow \mathsf{PackedSignMantissa}\big[\mathsf{ThreadOutputPos}[t]\big]$
      $\mathrm{Sign} \leftarrow \mathrm{Byte}$ `bitwise_and` `0b10000000`
      $\mathrm{Mantissa} \leftarrow \mathrm{Byte}$ `bitwise_and` `0b01111111`
      $\mathsf{WriteBuffer}[\mathsf{ThreadOutputPos}[t] - \mathsf{BlockOutputPos}[b]] \leftarrow$
        $(\mathrm{Sign}$ `bitwise_left_shift` $8)$ `bitwise_or`
        $(\mathrm{Exponent}$ `bitwise_left_shift` $7)$ `bitwise_or` $\mathrm{Mantissa}$
      $\mathrm{BitOffset} \leftarrow \mathrm{BitOffset} + \mathsf{CodeLengths}[\mathrm{Exponent}]$
      $\mathsf{ThreadOutputPos}[t] \leftarrow \mathsf{ThreadOutputPos}[t] + 1$
    end while
  end for
  ▷ Perform coalesced writes to HBM:
  $\mathsf{Outputs}[\mathsf{BlockOutputPos}[b] \ldots (\mathsf{BlockOutputPos}[b+1] - 1)] \leftarrow$
    $\mathsf{WriteBuffer}[0 \ldots (\mathsf{BlockOutputPos}[b+1] - \mathsf{BlockOutputPos}[b] - 1)]$
end for
end procedure

E Hardware for Experiments

Table 4 presents the hardware configuration of servers used for experiments.

F DFloat11 Compression Time

Table 5: Compression time per transformer block for different models.

Model	Compression Time per Transformer Block (s)
Llama 3.1 8B Instruct	191
Llama 3.3 70B Instruct	547
Llama 3.1 405B Instruct	2133

Table 5 reports the time required to compress a single transformer block for models of different sizes. Compression is a one-time preprocessing step for each model and is performed using a single CPU thread. Since transformer blocks are independent in terms of weight storage, their compression can be parallelized across multiple CPU threads, making the overall process highly scalable and efficient.

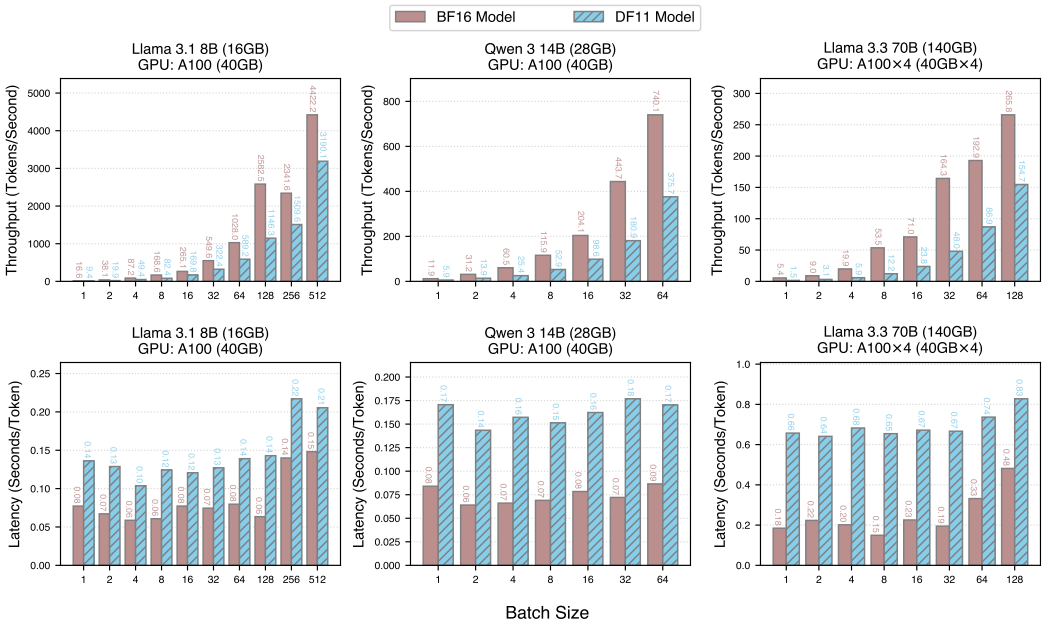

Figure 10: Comparison of average latency and throughput for token decoding between the original (BF16) models and their losslessly compressed (DF11) counterparts. The BF16 and DF11 models are run on the same GPU configurations, with Flash Attention [7] turned on for both methods.

G GPU Inference Efficiency Comparison: BF16 vs. DF11

We present the GPU inference efficiency of BF16 and DF11 models in Figure 10, for various models and batch sizes on A100 GPUs.

H Impact of Lossy Quantization

An accuracy comparison of the original and INT8-quantized Llama model is presented in table 6.

Table 6: INT8 quantization error on different tasks. "Math" denotes MATH Hard with 2 shots. "GPQA CoT" is with 2 shots. "Δ" denotes the error gap via INT8 quantization.

Model	Data Type	Math	GPQA CoT
Llama-3.1-8B-Instruct	BF16	23.92	15.18
	INT8	19.92	14.06
	Δ	4.0	1.12

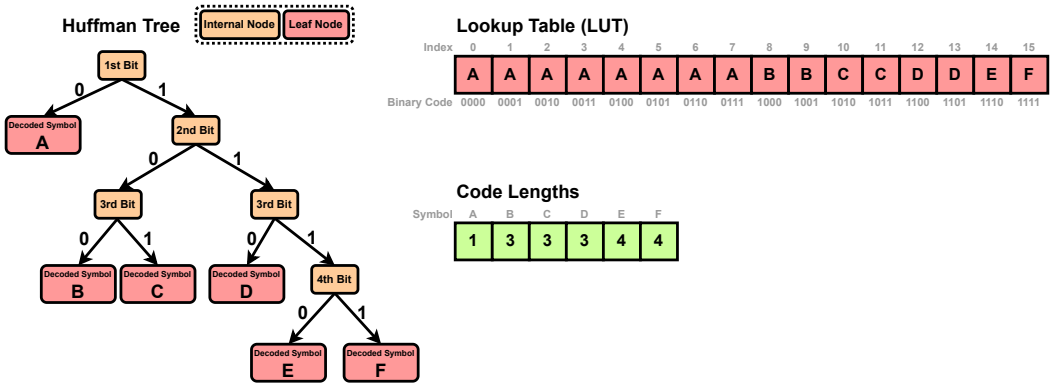

Figure 11: Decoding Huffman codes can be performed either by traversing the Huffman tree or by using two lookup tables: one that maps each L-bit binary code to its corresponding symbol, and another that stores the code length for each symbol.

I Efficient Decoding of Huffman Codes Using Compact Lookup Tables

I.1 The Dual Lookup Table Approach

Huffman decoding can be performed by traversing the Huffman tree: starting from the root, each bit of the encoded bitstream determines the branch to follow, and the symbol is fully decoded upon reaching a leaf node. While this bit-by-bit traversal is conceptually simple, it is inefficient in practice. Each branching decision depends on the previous one, leading to frequent memory accesses and conditional jumps. This pattern is especially problematic on GPUs, where it causes branch divergence and limits instruction-level parallelism. A widely adopted alternative is *lookup-table-based decoding* [53], which flattens the Huffman tree into two compact lookup tables. This enables decoding of each symbol using just two array lookups and a bit shift, significantly improving throughput.

We employ two lookup tables, `LUT` and `CodeLengths`, to achieve efficient, branch-free Huffman decoding. Let L denote the length of the longest codeword in the Huffman codebook. We construct the primary lookup table `LUT` as an array of size 2^L, where each entry maps an L-bit binary sequence to the first symbol it encodes.

Figure 11 shows an example with $L = 4$ and a set of symbols `A, B, C, D, E, F`. For clarity, we use letters to represent symbols, though in practice these correspond to exponent values in BFloat16 weights. The lookup table `LUT` contains $2^4 = 16$ entries, indexed by all possible 4-bit binary sequences. Each entry in `LUT` stores the symbol whose Huffman code matches the prefix of that index. If a symbol's Huffman code is shorter than L bits, it will fill multiple consecutive entries. For example, if symbol `A` is encoded as the single bit `0`, then all binary sequences from `0000` to `0111` begin with `0`, so entries 0 through 7 in `LUT` are assigned to `A`. In contrast, symbols with Huffman codes of length L occupy exactly one entry each. For instance, `E` $=$ `1110` and `F` $=$ `1111` map to entries 14 and 15, respectively. This construction yields a dense prefix table that allows decoding a symbol with a single array lookup using an L-bit segment from the encoded bitstream.

To advance the encoded bitstream for decoding the next symbol, we also store the code lengths of all symbols. The second lookup table, `CodeLengths`, maps each symbol to its Huffman code length. In

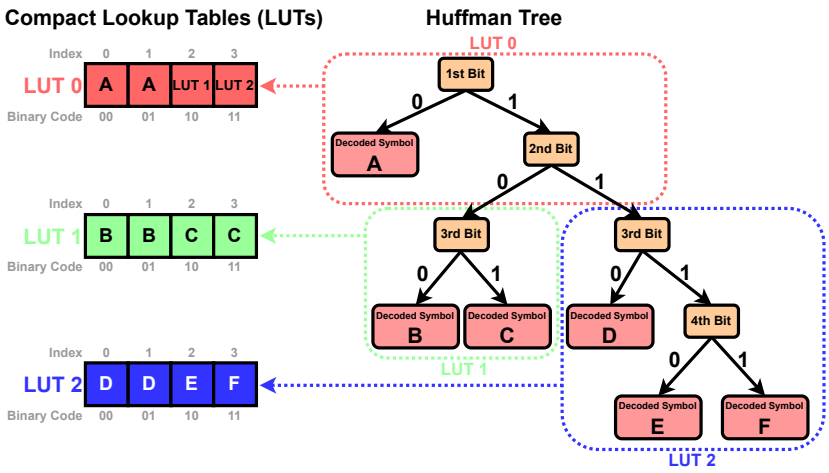

Figure 12: A Huffman tree can be decomposed into a hierarchy of subtrees, each represented by a compact lookup table (LUT). Each LUT may reference another lower-level LUT in the hierarchy. This hierarchical decoding approach is functionally equivalent to using a single monolithic LUT, but significantly more memory efficient.

the example, the lengths are: `A:1, B:3, C:3, D:3, E:4, F:4`. Together, these two tables allow fast, deterministic decoding by repeating the following steps:

1. Use the next L bits from the encoded bitstream to index LUT and retrieve the decoded symbol.
2. Look up the code length of the decoded symbol from $\mathsf{CodeLengths}$ to determine how many bits to consume.
3. Advance the encoded bitstream and repeat.

This approach eliminates conditional branches and pointer chasing during decoding, making it highly suitable for parallel computation on GPUs.

I.2 Decomposing LUT into Hierarchical, Compact Lookup Tables

The primary lookup table LUT contains 2^L entries, where L is the maximum code length in the Huffman codebook. While this enables constant-time decoding, the table size grows exponentially with L. In practice, L ranges from 24 to 32 for Huffman trees built with BFloat16 exponents. This results in table sizes of 2^{24} to 2^{32} entries, which far exceeds the capacity of GPU SRAM. To address this, we decompose LUT into multiple smaller lookup tables that fit within on-chip memory, while still enabling fast decoding.

Hierarchical Table Structure Instead of storing a single flat table of size 2^L, we decompose LUT into a hierarchy of compact lookup tables. Each table corresponds to a subtree of the Huffman tree and is responsible for decoding b bits. Each table processes the next b bits and either (i) directly returns a decoded symbol, or (ii) delegates to a table next in the hierarchy for decoding the next b bits. This hierarchical organization mirrors the structure of the original Huffman tree and significantly reduces total memory usage.

Figure 12 illustrates an example where the Huffman tree is partitioned into three subtrees, each mapped to a separate lookup table responsible for 2 bits. The decoding process using these three LUTs proceeds as follows:

- LUT_0: Uses the first and second bits of the encoded bitstream to determine how to proceed, leading to 3 possible cases:
  - `00`, `01` $\rightarrow$ decode the next symbol as `A`.
  - `10` $\rightarrow$ delegate to LUT_1 .

  - `11` → delegate to LUT_2.
- LUT_1: Uses the third and fourth bits of the encoded bitstream to continue decoding:
  - `00`, `01` → decode the next symbol as `B`
  - `10`, `11` → decode the next symbol as `C`
- LUT_2: Uses the third and fourth bits of the encoded bitstream to continue decoding:
  - `00`, `01` → decode the next symbol as `D`
  - `10` → decode the next symbol as `E`
  - `11` → decode the next symbol as `F`

For decoding Huffman-coded BFloat16 exponents, we decompose the LUT into multiple compact lookup tables, each responsible for decoding 8 bits (i.e. $b = 8$). This allows us to read the next byte from the encoded bitstream and perform an array lookup from a 256-entry array in each step. In practice, the decomposition of LUT leads to 4 to 8 compact LUTs, each with 256 entries, which comfortably fits within fast SRAM.

J Text-to-image Results of BF16 and DF11 Diffusion Models

Figure 13: Images generated by Stable Diffusion 3.5 Large in the original BFloat16 precision (**top 5**) are pixel-wise identical to those produced by the DFloat11-compressed model (**bottom 5**), using the same prompt and random seed.

Figure 13 presents the comparison of images generated using Stable Diffusion 3.5 Large in BFloat16 and DFloat11 weight format. The images are pixel-wise identical, when using the same prompt and random seed.

K Limitations

This work focuses exclusively on losslessly compressing BFloat16 weights. We do not consider other formats such as FP32, FP16, or FP8, which may require different compression strategies. While DF11 improves memory efficiency, it introduces a small but non-zero latency overhead due to decompression. This overhead is amortized at larger batch sizes but may impact latency-sensitive applications with small batches. Our evaluation is limited to GPUs. We do not assess performance on other hardware such as CPUs, TPUs, or custom accelerators, which may require platform-specific optimizations.

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The main claims accurately reflect the paper's contributions and scope, as supported by evidences in the Method and Experiments section.

   Guidelines:

   - The answer NA means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A No or NA answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: We have included a Limitations section.

   Guidelines:

   - The answer NA means that the paper has no limitation while the answer No means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [NA]

Justification: This paper does not include theoretical results.

Guidelines:

- The answer NA means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: We have fully disclosed all information needed for result reproduction.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - If the paper includes experiments, a No answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
   - If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
   - Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
   - While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
     (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
     (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
     (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
     (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

Answer: [Yes]

Justification: Our code and models are publicly available.

Guidelines:

- The answer NA means that paper does not include experiments requiring code.
- Please see the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- While we encourage the release of code and data, we understand that this might not be possible, so "No" is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
- The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
- The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
- At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
- Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer, etc.) necessary to understand the results?

   Answer: [Yes]

   Justification: This paper does not include training experiments. All evaluation details necessary to reproduce and understand the results are provided in the full paper.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [Yes]

   Justification: We include error bars for experiments where appropriate.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The authors should answer "Yes" if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
   - The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
   - The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
   - The assumptions made should be given (e.g., Normally distributed errors).

- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g. negative error rates).
- If error bars are reported in tables or plots, The authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: We have provided detailed information on the computational resources required to reproduce our results.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: After reviewing the NeurIPS Code of Ethics, we have verified that our research is in compliance.

   Guidelines:

   - The answer NA means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer No, they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [NA]

    Justification: This work focuses on improving the inference memory efficiency of existing pre-trained models and does not involve the development of new model, training data, or training method, which limits its direct societal impact.

    Guidelines:

    - The answer NA means that there is no societal impact of the work performed.
    - If the authors answer NA or No, they should explain why their work has no societal impact or why the paper does not address societal impact.

- Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pretrained language models, image generators, or scraped datasets)?

    Answer: [NA]

    Justification: We do not introduce new data or models.

    Guidelines:

    - The answer NA means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: The assets used are properly credited.

    Guidelines:

    - The answer NA means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.
    - The name of the license (e.g., CC-BY 4.0) should be included for each asset.
    - For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
    - If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.

- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [NA]

    Justification: No new assets are introduced by this paper.

    Guidelines:

    - The answer NA means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [NA]

    Justification: This paper does not involve crowdsourcing or research with human subjects.

    Guidelines:

    - The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [NA]

    Justification: This paper does not involve crowdsourcing or research with human subjects.

    Guidelines:

    - The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
    - Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
    - We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
    - For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does not impact the core methodology, scientific rigorousness, or originality of the research, declaration is not required.

Answer: [NA]

Justification: LLMs are not involved in any originl or important parts of this research.

Guidelines:

- The answer NA means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
- Please refer to our LLM policy (`https://neurips.cc/Conferences/2025/LLM`) for what should or should not be described.