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

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

[1]Department of Computer Science, Rice University
[2]xMAD.ai
[3]Department of Computer and Data Sciences, Case Western Reserve University

Abstract

Large Language Models (LLMs) 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 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 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) decomposition of memory-intensive lookup tables (LUTs) into compact LUTs that fit in GPU SRAM, (ii) a two-phase kernel for coordinating thread read/write positions using lightweight auxiliary variables, and (iii) transformer-block-level decompression to minimize latency. Experiments on recent models, including Llama-3.1, Qwen-2.5, and Gemma-3, validates our hypothesis that DFloat11 achieves around 30% model size reduction while preserving bit-for-bit exact outputs. Compared to a potential alternative of offloading parts of an uncompressed model to the CPU to meet memory constraints, DFloat11 achieves 1.9–38.8× higher throughput in token generation. With a fixed GPU memory budget, DFloat11 enables 5.3–13.17× longer context 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. Our code and models are available at `https://github.com/LeanModels/DFloat11`.

1 Introduction

Large Language Models (LLMs) have demonstrated remarkable capabilities across a wide range of natural language processing (NLP) tasks [38]. However, their rapidly increasing sizes create substantial obstacles to efficient deployment and inference, especially in environments with limited computational or memory resources. For example, Llama-3.1-405B [11] 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

Preprint. Under review.

Table 1: Comparison of running LLMs on specific GPUs using BFloat16 and our proposed DFloat11 format. Note that **DFloat11 produces outputs that are 100% identical to those of the original BFloat16 models.**

Model	GPU Type	Method	Successfully Run?	Required Memory
Llama-3.1-405B-Instruct	8×H100-80G	BFloat16	✘	811.71 GB
		DFloat11 (Ours)	✔	551.22 GB
Llama-3.3-70B-Instruct	1×H200-141G	BFloat16	✘	141.11 GB
		DFloat11 (Ours)	✔	96.14 GB
QwQ-32B	1×A6000-48G	BFloat16	✘	65.53 GB
		DFloat11 (Ours)	✔	45.53 GB
Qwen2.5-32B-Instruct	1×A6000-48G	BFloat16	✘	65.53 GB
		DFloat11 (Ours)	✔	45.53 GB
DeepSeek-R1-Distill-Llama-8B	1×RTX 5080-16G	BFloat16	✘	16.06 GB
		DFloat11 (Ours)	✔	11.23 GB

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.**

1.1 Lossy Compression Degrades Output Quality, and Lossless Compression Lacks GPU Efficiency

To address the growing model size of LLMs, quantization techniques [7, 21] are commonly employed, converting high-precision weights into lower-bit representations. This significantly reduces memory footprint and computational requirements, facilitating faster inference and deployment in resource-constrained environments. However, quantization is inherently a *lossy compression* technique, introducing a fundamental drawback: **it inevitably alters the output distribution of LLMs, thus impacting model accuracy and reliability.**

In contrast, *lossless compression* techniques preserve the exact original weights of large-scale LLMs while effectively reducing their size, ensuring the model's output distribution remains identical to that of the uncompressed representation (e.g., BFloat16). However, existing lossless methods primarily focus on improving storage efficiency for LLMs, such as shrinking model checkpoints [13, 16] or optimizing performance for specialized hardware like FPGAs [40]. While these methods indeed benefit scenarios like efficient checkpoint rollbacks during training [32] or accelerated downloads from model repositories such as HuggingFace [16], their advantages typically do not extend effectively to general-purpose GPU-based LLM inference.

1.2 Dynamic-Length Float (DFloat): A Data Format for Lossless GPU Inference

Our key insight is that the BFloat16 (BF16) format used to store LLM weights is information-inefficient. Through an analysis of the frequency distribution of BFloat16 components (sign, exponent, and mantissa) in recent LLMs, **we leverage the observation that BFloat16 exponents carry significantly less information than their allocated bit width**, with an entropy of approximately 2.6 bits compared to the assigned 8 bits [13]. This highlights an opportunity for substantial lossless compression. Leveraging entropy coding techniques such as Huffman coding [17], which assign shorter codes to more frequent symbols, we can achieve roughly 30% compression without any loss in information. However, such a compressed representation introduces considerable challenges for efficient GPU inference.

Although theoretically appealing, efficient inference with entropy-coded weights on GPUs presents significant challenges. During inference, losslessly compressed weight matrices must be decompressed on-the-fly back into their original BFloat16 format for matrix multiplications, and this decompression step introduces a critical performance bottleneck. Traditional Huffman decoding algorithms rely on sequential, bit-by-bit traversal of a Huffman tree, making them inherently unsuitable for GPU's massively parallel architecture. Assigning a single GPU thread for decompression, however, results in severe under-utilization of GPU resources and increased latency. Existing entropy coding-based compression methods, ranging from established CNN-focused approaches [13] to recent

adaptations for LLMs [16, 14, 40], have yet to deliver significant inference efficiency improvements in GPU-based scenarios relevant to most end-users.

To enable efficient inference with entropy-coded weights on GPUs, we introduce a novel data representation called **Dynamic-Length Float (DFloat)**, along with a GPU kernel designed for fast, online decompression of 11-bit DFloat weights **(DFloat11)** in a massively parallel manner. The DFloat11 decompression kernel comprises three core components: ❶ *Efficient decoding of entropy-coded weights using compact lookup tables (LUTs) stored in GPU shared memory (SRAM).* A single, monolitic LUT for decoding 32-bit Huffman codes would require roughly 4.29 billion entries (as discussed in Section 3.3.1), making it prohibitively memory-intensive. To mitigate this, we decompose it into multiple compact LUTs that fit within GPU SRAM to enable fast access. ❷ *Precise and efficient identification of "READ" positions in encoded weights and "WRITE" positions for decoded weights.* Since entropy-encoded weights have variable bit widths and are tightly packed, determining correct read/write offsets for each thread is challenging. We resolve this through a two-phase kernel design, which employs a minimal set of auxiliary variables to efficiently coordinate thread-specific input and output positions. ❸ *Performing matrix decompression in batch for improved GPU resource utilization.* Decompressing individual weight matrices leads to poor GPU utilization due to their relatively small size. To address this, we decompress weights at the transformer-block level, significantly enhancing throughput and reducing inference latency.

We summarize our contributions as follows:

- We propose **Dynamic-Length Float (DFloat11)**, a data format that losslessly compresses BFloat16 weights of LLMs down to approximately 11 bits. By leveraging the information inefficiency of the BFloat16 representation, we achieve roughly a 30% "free" model-size reduction while preserving bit-for-bit identical outputs.
- We introduce optimized algorithmic designs to enable efficient GPU inference with DFloat11-compressed models. By carefully exploiting GPU memory and computational hierarchies, we develop hardware-aware algorithms for efficient online inference of DFloat11-compressed models.
- We extensively evaluate our method on popular LLMs and large reasoning models, including Llama-3.1, Qwen2.5, QwQ-32B, Mistral, Gemma-2, Gemma-3, DeepSeek-R1-Distill-Qwen, and DeepSeek-R1-Distill-Llama [11, 36, 31, 30, 27, 26, 12]. Experimental results demonstrate that our method consistently achieves about 30% compression without altering outputs. For example, our approach reduces the hardware requirements for running *Llama-3.1-405B from two GPU nodes down to a single node*, equipped with 8×80GB NVIDIA A100 GPUs, without any loss in accuracy or changes in output distribution.

2 Motivation: Is Lossless Compression of LLMs Worth Studying?

Much of the motivation behind our work lies in understanding **whether lossless compression of 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 [7, 21], 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 [10, 37, 18] 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 [11] and its 8-bit counterpart

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

(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 [12] 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 [23] reveals that quantizing DeepSeek-R1-Distill-Qwen-1.5B with 8-bit SmoothQuant [34] (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 D.

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 ∼70% of the memory footprint with many throughput benefits**, which is a unique and practical offering for resource-constrained deployment settings.

3 Method

In this section, we present our framework for lossless compression of LLM weights. We begin with background on BFloat16, entropy coding, and the GPU computation and memory model. Next, we motivate our approach by analyzing the information efficiency of BFloat16 in representing model weights. We then introduce our proposed floating-point format, Dynamic-Length Float (DFloat11), along with its custom decompression kernel designed for efficient GPU inference.

3.1 Preliminary

Brain Float (BFloat16) The weights of large language models (LLMs) are typically represented using floating-point number formats. Recent state-of-the-art LLMs predominantly employ the 16-bit Brain Floating Point format (BFloat16 or BF16), which balances numerical precision and 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. Compared to FP32, BF16 uses half the bytes to represent the same number of parameters while having a similar numerical range, due to having the same number of exponent bits. In contrast to FP16, BF16 offers a wider numerical range, reducing the risk of overflow during model training and inference.

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 [17], arithmetic coding [20], and Asymmetric Numeral Systems (ANS) [5]. Among these, Huffman coding is one of the earliest and 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. Huffman coding is widely used in file compression and data transmission. However, decoding in a massively parallel manner remains 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

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

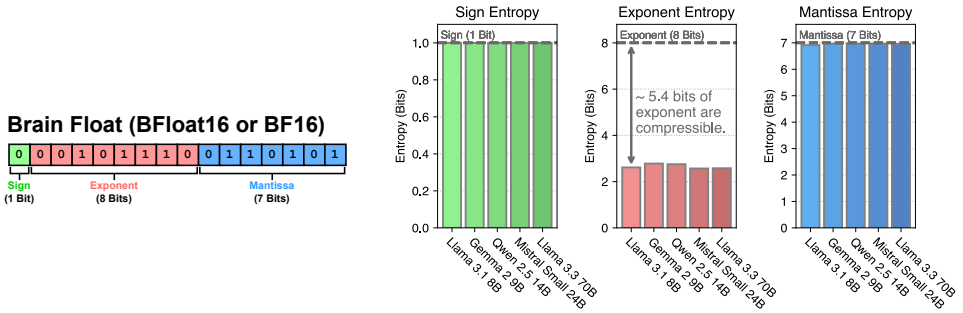

Figure 1: (Left) The allocation of bits for the components of BFloat16. (Right 3) The information content, as measured by Shannon Entropy, of the components (sign, exponent, mantissa) of BFloat16 weights in various LLMs.

blocks and executed on streaming multiprocessors (SMs). Each block of threads has access to a small, fast on-chip memory known as shared memory or SRAM. This memory offers low latency and high throughput, making it much more efficient for frequent read and write operations compared to the global high-bandwidth memory (HBM), which is off-chip and shared across all SMs. 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 losslessly compressed weights during inference.

3.2 Technical Motivation: BFloat16 Representation is Information Inefficient

To motivate the lossless compression of LLM weights, we analyze the compressibility of the BFloat16 components (sign, exponent, and mantissa) in the weights of recent large language models. Specifically, we use Shannon entropy to quantify the information content of all parameters within the 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 components 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 better understand this discrepancy, we visualize the relative frequency for all values of BFloat16 components in Figure 7 in the Appendix, and plot the ranked frequency distribution of exponent values in Figure 8 in the Appendix. As shown, the sign and mantissa values are distributed relatively uniformly across the entire value ranges. However, the distribution of exponent values is highly imbalanced: out of the 256 possible 8-bit values, only around 40 are used, with the rest never appearing in the weights. Moreover, the ranked frequencies decay rapidly towards zero. These findings explain the low entropy of the exponent and highlight the potential for significant compression. In the following sections, we leverage the low information content of BFloat16 exponents to design a lossless compression framework that enables efficient GPU inference.

3.3 Dynamic-Length Float: A Lossless LLM Compression Framework 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 from all BFloat16 weights within the linear projection matrices of an LLM. We then compress the exponents using Huffman coding, while preserving the original signs and mantissas. Exponents are encoded and

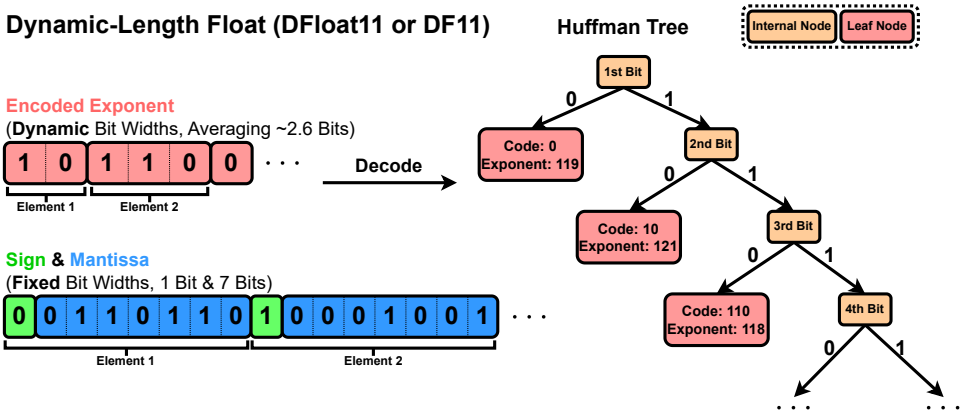

Figure 2: An illustration of our proposed format Dynamic-Length Float for compressing the BFloat16 weights of LLMs losslessly.

tightly bit-packed into a byte array, $\mathbf{EncodedExponent}$, while the sign and mantissa are left uncompressed and stored in a separate byte array $\mathbf{PackedSignMantissa}$. Figure 2 illustrates Dynamic-Length Float (DFloat11 or DF11), our proposed format for compactly representing floating-point model parameters.

The Core Challenge: Efficient GPU Inference with Compressed Weights Although Dynamic-Length Float enables effective lossless compression of LLMs, a key challenge remains: *how to perform efficient GPU inference with these compressed weights*. Entropy-coded weights, due to their variable-length encoding, cannot be directly used in matrix multiplications. Therefore, each weight matrix is decompressed on-the-fly to its original BFloat16 representation when needed during inference, and immediately discarded after the matrix multiplication completes to save memory. Traditional Huffman decoding involves bit-by-bit traversal of the Huffman tree to decode each element, a process that is inherently sequential and fits poorly with the massively parallel execution model of GPUs. Naively assigning a single thread to decode each value results in severe underutilization of GPU resources and high latency. Overcoming this bottleneck is critical for making compressed inference practical.

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:

1. Decomposing a monolithic prefix-free lookup table (LUT) into multiple compact LUTs that fit within GPU SRAM;
2. Introducing a two-phase kernel design that leverages lightweight auxiliary variables to efficiently coordinate read/write operations of threads;
3. Performing decompression at the transformer block level to boost throughput and minimize latency.

3.3.1 Efficient Decoding with Compact LUTs

Huffman codes can be efficiently decoded using a lookup table (LUT)–based approach [35]. We construct a lookup table, denoted as $\mathbf{LUT}$, of size 2^L, where L is the maximum bit length of any Huffman code in the codebook. Each entry in $\mathbf{LUT}$ maps an L-bit sequence to the decoded symbol whose Huffman code is a prefix of that sequence. Due to the prefix-free property of Huffman codes, this mapping is guaranteed to be unambiguous.

To decode, we read the next L bits from the encoded bitstream and use them as an index into $\mathbf{LUT}$ to retrieve the next decoded symbol. To determine the exact number of bits consumed in the decoding process (i.e., the length of the matched Huffman code), we use a second lookup table,

Algorithm 1 GPU kernel for decompressing DFloat11 to BFloat16

procedure DFLOATTOBFLOAT
require:
- $\mathbf{EncodedExponent}, \mathbf{PackedSignMantissa}$: byte arrays
- $\mathbf{LUT}_1, \mathbf{LUT}_2, \mathbf{LUT}_3, \mathbf{LUT}_4, \mathbf{CodeLengths}$: 8-bit unsigned integer arrays of size 256
- $\mathbf{Gaps}$: 5-bit unsigned integer array (one entry per thread in each block)
- $\mathbf{BlockOutputPos}$: 32-bit unsigned integer array (one entry per block)
- $\mathbf{Outputs}$: BFloat16 array, for storing results
- B, T, n, R: the number of blocks, number of threads, number of bytes in $\mathbf{EncodedExponent}$ processed by each thread, a reserved value, respectively

  Divide $\mathbf{EncodedExponent}$ into chunks: $\mathbf{EncodedExponent}_1, \ldots, \mathbf{EncodedExponent}_B$ of size nT bytes each
  for all $b \leftarrow 1, \ldots, B$ **(in parallel across blocks) do**
    Load $\mathbf{EncodedExponent}_b$ into SRAM
    Divide $\mathbf{EncodedExponent}_b$ into chunks: $\mathbf{EncodedExponent}_{b,1}, \ldots, \mathbf{EncodedExponent}_{b,T}$ of size n bytes each
    Load $\mathbf{LUT}_1, \mathbf{LUT}_2, \mathbf{LUT}_3, \mathbf{LUT}_4, \mathbf{CodeLengths}$ into SRAM
    Initialize integer arrays $\mathbf{NumElements}[1 \ldots T], \mathbf{ThreadOutputPos}[1 \ldots T]$ with all 0s
    for all $t \leftarrow 1, \ldots, T$ **(in parallel across threads) do**
    ▷ Phase 1: Each thread determines its initial output position
      $\text{BitOffset} \leftarrow \mathbf{Gaps}[bT + t]$
      while $\text{BitOffset} < 8n$ **do**
        Read the next 4 bytes of $\mathbf{EncodedExponent}_{b,t}$, starting from the BitOffset-th bit, into $\text{Byte}_{1\ldots4}$
        $\text{Exponent} \leftarrow R$
        $i \leftarrow 1$
        while $\text{Exponent} = R$ **do**
          $\text{Exponent} \leftarrow \mathbf{LUT}_i[\text{Byte}_i]$
          $i \leftarrow i + 1$
        end while
        $\text{BitOffset} \leftarrow \text{BitOffset} + \mathbf{CodeLengths}[\text{Exponent}]$
        $\mathbf{NumElements}[t] \leftarrow \mathbf{NumElements}[t] + 1$
      end while
      Thread Synchronization Barrier
      $\mathbf{ThreadOutputPos}[t] \leftarrow \mathbf{BlockOutputPos}[b] + \sum_{i=1}^{t-1} \mathbf{NumElements}[i]$
    ▷ Phase 2: Writing decoded BFloat16s to the appropriate positions
      $\text{BitOffset} \leftarrow \mathbf{Gaps}[bT + t]$
      while $\text{BitOffset} < 8n$ **do**
        Read the next 4 bytes of $\mathbf{EncodedExponent}_{b,t}$, starting from the BitOffset-th bit, into $\text{Byte}_{1\ldots4}$
        $\text{Exponent} \leftarrow R$
        $i \leftarrow 1$
        while $\text{Exponent} = R$ **do**
          $\text{Exponent} \leftarrow \mathbf{LUT}_i[\text{Byte}_i]$
          $i \leftarrow i + 1$
        end while
        $\text{Byte} \leftarrow \mathbf{PackedSignMantissa}\big[\mathbf{ThreadOutputPos}[t]\big]$
        $\text{Sign} \leftarrow \text{Byte}$ `bitwise_and 0b10000000`
        $\text{Mantissa} \leftarrow \text{Byte}$ `bitwise_and 0b01111111`
        $\mathbf{Outputs}[\mathbf{ThreadOutputPos}[t]] \leftarrow$ (Sign `bitwise_left_shift` 8) `bitwise_or` (Exponent `bitwise_left_shift` 7) `bitwise_or` Mantissa
        $\text{BitOffset} \leftarrow \text{BitOffset} + \mathbf{CodeLengths}[\text{Exponent}]$
        $\mathbf{ThreadOutputPos}[t] \leftarrow \mathbf{ThreadOutputPos}[t] + 1$
      end while
    end for
  end for
end procedure

$\mathbf{CodeLengths}$, which maps each symbol to the length of its Huffman code. We then advance the bitstream by that length and repeat the process to decode subsequent symbols.

For decoding the exponents of DFloat11, we constrain the maximum code length L to 32 bits for each model, allowing us to read 32 bits from the encoded bitstream in each decoding step. Empirically, we observe that most models naturally produce maximum code lengths $L \leq 32$, or only slightly above this threshold. For models where $L > 32$, we enforce the length constraint by reducing the frequencies of the least common exponents to 1 and rebuilding the Huffman tree. This results in a more balanced structure in the tail of the Huffman tree, assigning equal-length codes to the rarest exponents and reducing the maximum code length to 32 bits.

However, when $L = 32$, a direct lookup table would require $2^{32} \approx 4.29$ billion entries, which is prohibitively memory-intensive. To address this, we propose partitioning the monolithic $\mathbf{LUT}$ into four disjoint and memory-efficient lookup tables. Specifically, we replace the 2^{32}-entry array with four 2^8-entry arrays: $\mathbf{LUT}_1$, $\mathbf{LUT}_2$, $\mathbf{LUT}_3$, and $\mathbf{LUT}_4$. Each entry in these tables consumes a single byte, resulting in a total memory footprint of $4 \times 2^8 = 1024$ bytes. The $\mathbf{CodeLengths}$ table adds another $2^8 = 256$ bytes, bringing the total to 1280 bytes. This fits in GPU SRAM and enables fast access.

Decoding a 32-bit encoded sequence using the four compact LUTs proceeds as follows. We start by reading the first byte (8 bits) of the sequence and using it as an index into $\mathbf{LUT}_1$. If the corresponding Huffman code has a length of 8 bits or fewer, $\mathbf{LUT}_1$ returns a valid decoded symbol, and the decoding step is complete. Otherwise, it returns a reserved value R, indicating that the code has not yet been fully resolved. In that case, we read the next byte from the sequence and use it to index into $\mathbf{LUT}_2$. This process continues with $\mathbf{LUT}_3$ and $\mathbf{LUT}_4$ as needed, until a non-R value is returned, signaling that the Huffman code has been successfully decoded. The first non-R value returned is the decoded symbol corresponding to a prefix of the 32-bit encoded sequence.

While this approach significantly reduces memory usage, it introduces potential ambiguity. Specifically, if two different Huffman codes with distinct prefixes both map to the reserved value R within the same LUT, the remaining bits may no longer be sufficient to unambiguously identify the correct decoded symbol in the subsequent LUTs. Fortunately, such conflicts are rare in practice and can be resolved. Due to the highly imbalanced distribution of exponent frequencies, the resulting Huffman trees are often skewed. As a result, each compact LUT typically maps at most one index to the value R, thereby avoiding ambiguity. In the rare cases where ambiguity does arise, it can be resolved by slightly adjusting the frequency distribution. Specifically, we increase the frequency of the more common exponent associated with the conflicting Huffman code and rebuild the Huffman tree. This shortens the code for that exponent, eliminating the conflict and ensuring that each compact LUT maps at most one index to the reserved value R.

3.3.2 Two-Phase Kernel and Lightweight Auxiliary Variables

To enable massively parallel decoding of entropy-coded exponents in DFloat11, we assign each thread a fixed number of bytes from the encoded sequence to process. However, this approach introduces two key challenges: 1. Because Huffman codes have variable bit widths and are tightly packed, the starting bit position for each thread to begin decoding is unclear. 2. Except for the first thread, the index of the elements being decoded is unknown, making it difficult to determine the correct output location for storing the results.

To address the first issue, we use a gap array [35] to determine the starting bit position for each thread. The gap array $\mathbf{Gaps}$ contains one entry per thread, and each entry specifies the bit offset of the first valid Huffman code relative to the thread's assigned starting byte. Since the maximum code length is 32 bits, each offset lies in the range $[0, 31]$. To maintain memory efficiency, we encode each entry using 5 bits.

To address the second issue, the most straightforward approach is to maintain an array that stores the output position of the first decoded element for each thread. However, this incurs substantial storage overhead. Given the large size of weight matrices, each output position must be represented as a 32-bit integer. Since decoding typically involves tens of thousands of threads per weight matrix, storing these positions results in significant memory consumption. This overhead undermines the compression ratio achieved by DFloat11.

To reduce storage overhead, we store the output position only for the first element of each thread block, rather than for every individual thread. Since each block typically contains hundreds to thousands of threads, this reduces the overhead from one 32-bit integer per thread to one per block, resulting in a negligible memory overhead.

To make decoding possible with block-level output positions, we adopt a *two-phase* kernel design. In the **first phase**, all threads within a block decode their assigned portions of the encoded sequence in parallel, but without writing any output to global memory. Instead, each thread counts how many elements it will decode. After this pass, we synchronize the threads within each block and compute the output position for every thread by calculating prefix sums over the counts, starting from the known output position of the block. In the **second phase**, each thread re-decodes the same portion of the encoded sequence, this time writing the decoded results to the HBM at the correct output positions. To avoid redundant memory read access to HBM during the two passes, we load the encoded exponents into SRAM. The pseudocode for the two-phase kernel is presented in Algorithm 1.

3.3.3 Transformer-Block-Level Decompression

We now have a complete method for decompressing entropy-coded exponents in a massively parallel manner. The weights of the LLM are stored in the DFloat11 format, accompanied by lightweight auxiliary data: thread-level gap offsets and block-level output positions, which determine the read and write locations for each thread. Both the compressed weights and auxiliary variables reside entirely on the GPU during inference. 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. In the DFloat11 decompression kernel, we set the number of bytes processed per thread to $n = 8$, the number of threads per block to $T = 256$, and the number of thread blocks to $B = \lceil \frac{|\mathbf{EncodedExponent}|}{nT} \rceil$, where $|\mathbf{EncodedExponent}|$ is the total number of bytes in the encoded exponents. As the size of the DFloat11 weights increases, more thread blocks are utilized, resulting in higher decompression throughput. This effect is illustrated in Figure 6, which shows that decompression throughput improves significantly with larger matrix sizes. To capitalize on this, we propose batching the decompression of multiple matrices together to improve throughput and hide latency. Specifically, we batch the decompression of all DFloat11 weight matrices within a single transformer block. Before executing any computation in a transformer block, we first decompress all its associated weights.

This technique significantly reduces decompression latency and improves overall inference efficiency. The latency breakdown for DFloat11-compressed Llama-3.1-8B-Instruct [11] across different batch sizes is shown in Figure 5.

4 Experiments

In this section, we empirically evaluate the effectiveness of DF11 compression and its inference efficiency on GPUs. We compress a wide selection of recent LLMs from their original BFloat16 representation into our proposed DF11 format and report the resulting compression factors. We then compare the inference performance of the DF11-compressed models running on different GPUs with that of uncompressed models. Finally, we conduct an ablation study to analyze the effects of compression.

Software and Hardware We implement the DF11 decompression kernel in CUDA and C++, and integrate it into the Transformers [33] inference framework. We evaluate the inference efficiency of our losslessly compressed models against their uncompressed counterparts. For the uncompressed baseline, 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.

Table 2: Lossless compression statistics for various models. Model sizes are shown before and after compression.

Model	Original → Losslessly Compressed	Compression Ratio	Compressed Bit Width
Llama-3.1-8B-Instruct	16.06 GB → 11.24 GB	69.98%	11.20
Meta-Llama-3-8B	16.06 GB → 11.26 GB	70.12%	11.22
Llama-3.3-70B-Instruct	141.11 GB → 96.14 GB	68.13%	10.90
Llama-3.1-405B-Instruct	811.71 GB → 551.22 GB	67.91%	10.87
Qwen2.5-14B-Instruct	29.54 GB → 20.81 GB	70.44%	11.27
Qwen2.5-32B-Instruct	65.53 GB → 45.53 GB	69.48%	11.12
QwQ-32B	65.53 GB → 45.53 GB	69.48%	11.12
Mistral-Nemo-Instruct-2407	24.50 GB → 17.04 GB	69.55%	11.13
Mistral-Small-24B-Instruct-2501	47.14 GB → 32.30 GB	68.52%	10.96
Codestral-22B-v0.1	44.49 GB → 30.24 GB	67.96%	10.87
gemma-2-9b-it	20.32 GB → 14.59 GB	71.81%	11.49
gemma-3-12b-it	25.55 GB → 18.21 GB	71.27%	11.40
gemma-3-27b-it	56.84 GB → 39.95 GB	70.28%	11.25
DeepSeek-R1-Distill-Qwen-7B	15.23 GB → 10.73 GB	70.48%	11.28
DeepSeek-R1-Distill-Llama-8B	16.06 GB → 11.23 GB	69.95%	11.19

Table 3: Comparison of accuracy and perplexity for the original (BF16) and losslessly compressed (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
Qwen2.5-14B-Instruct	BF16	78.885 ± 0.330	51.775 ± 1.749	6.661	23.093
	DF11 (Ours)	78.885 ± 0.330	51.775 ± 1.749	6.661	23.093

4.1 Results

4.1.1 DF11 Compresses LLMs to 70% Size

Table 2 presents the compression factors of DF11 for a wide selection of recent LLMs. Specifically, we apply compression to all linear projection layers by converting their parameters from BF16 to DF11. The models we compress include LLaMA 3/3.1/3.3 [11], Qwen 2.5 [36], QwQ [31], Mistral Nemo/Small/Codestral [29, 30, 28], Gemma 2/3 [27, 26], and DeepSeek R1 Distilled [12]. DF11 achieves approximately 70% compression across all models, corresponding to an effective bit width of around 11 bits.

4.1.2 Accuracy and Perplexity Evaluations Confirm DF11 Is Perfectly 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` [9], reporting accuracy on MMLU [15] and TruthfulQA [22], and word-level perplexity on WikiText [24] and C4 [25]. The results are shown in Table 3. As demonstrated, the compressed models achieve identical accuracy and perplexity to their original BF16 counterparts. To further validate the losslessness property, we compare the decompressed BF16 weight matrices from DF11 against the originals for each model in Table 2, confirming exact bit-level equivalence.

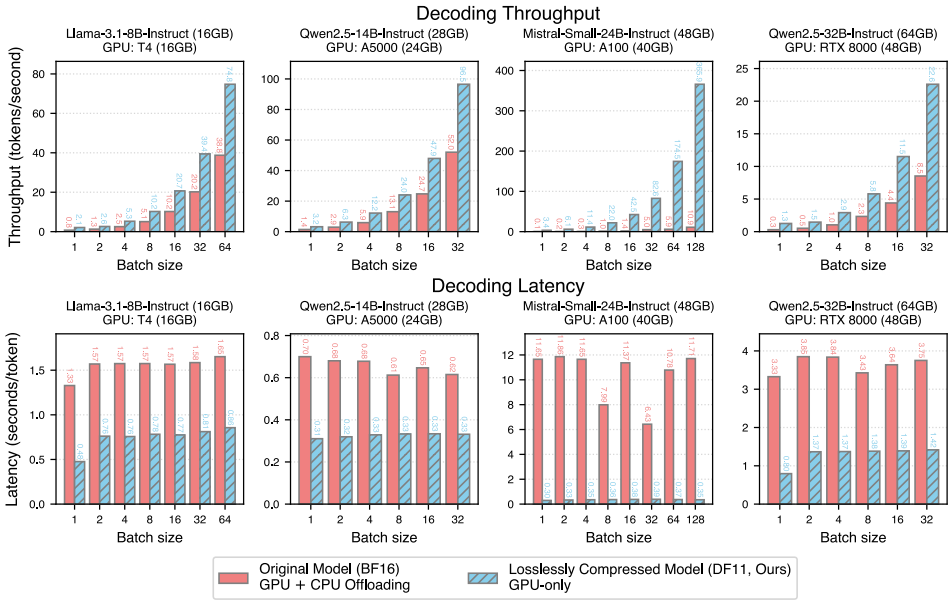

Figure 3: Comparison of average latency and throughput for token decoding between the original (BF16) models and their losslessly compressed (DF11) counterparts. Portions of the BF16 models are offloaded to the CPU due to GPU memory constraints.

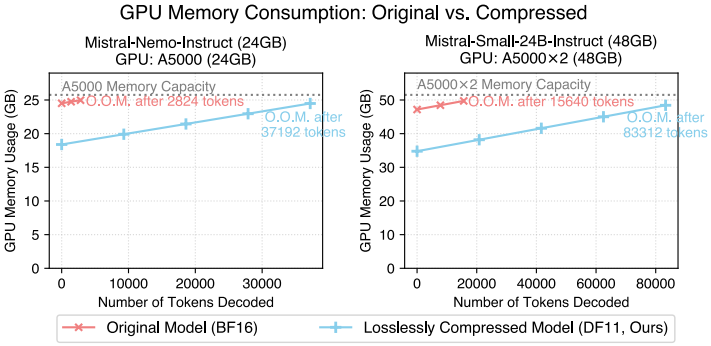

Figure 4: Comparison of GPU memory consumption between the original (BF16) models and their losslessly compressed (DF11) counterparts. The DF11 models support 5.33–13.17× longer context lengths by allowing more GPU memory to be used for storing the KV cache. "O.O.M." means out of memory.

4.1.3 DF11 Outperforms CPU Offloading in Inference Efficiency

We compare the inference efficiency of DF11 and BF16 models across different hardware platforms. The uncompressed BF16 models exceed the memory limits of a single GPU, whereas the losslessly compressed DF11 models fit within those limits. For the BF16 models, we retain most of the model and the computation in the GPU while offloading some components and their associated computations to the CPU. To measure latency and throughput, we first perform a warm-up run by processing 100 tokens. In the actual evaluation, we decode 100 tokens starting from an empty prompt, using varying batch sizes. Each configuration is run five times, and we report the average latency and throughput across these runs. The results across different models, GPUs, and batch sizes are presented in Figure 3. As shown, DF11 models consistently outperform the BF16 models with CPU offloading, achieving

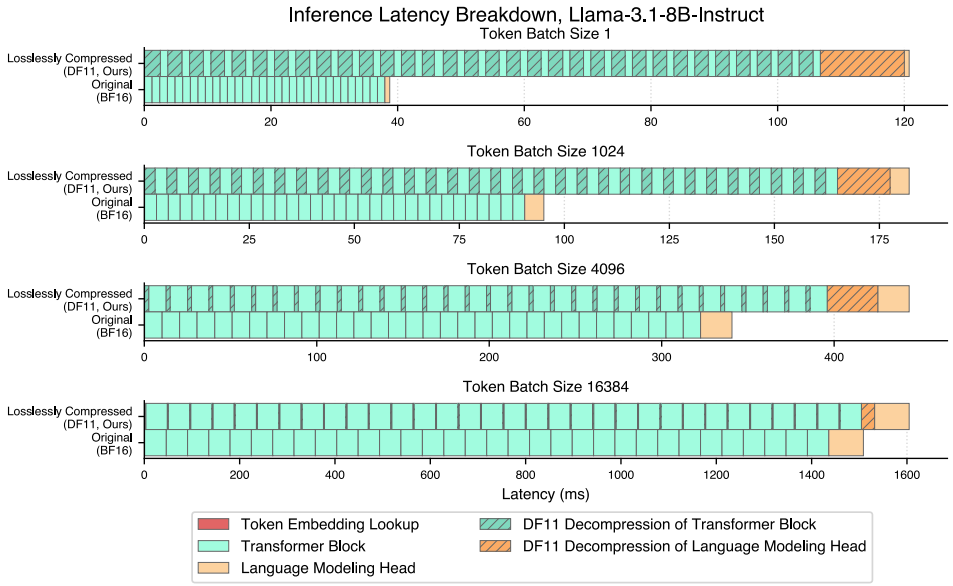

Figure 5: Comparison of latency breakdown for DFloat11 and BFloat16 Llama-3.1-8B-Instruct during GPU inference for different token batch sizes, using one A100-40GB GPU.

1.85–38.83× lower latency or higher throughput. For comparison using multiple GPUs, Figure 9 in the Appendix shows the performance of DF11 models running on a single GPU versus BF16 models running on two GPUs.

4.1.4 Memory Savings from DF11 Enable Longer Generation Lengths

The memory savings provided by DF11 compression not only reduce the number of GPUs required for inference but also enable longer generation lengths. During inference, the KV cache grows lienarly with the number of decoded tokens and quickly becomes a bottleneck for GPU memory. In Figure 4, we show the GPU memory consumption of DF11 and BF16 models during inference with a batch size of 1, as the number of decoded tokens increases. As shown, DF11 compression significantly extends the token generation length, allowing 5.33–13.17× more tokens to be decoded before reaching the GPU memory limit compared to BF16 models.

4.2 Ablation Study

4.2.1 Latency Breakdown Shows Decompression Overhead Is Amortized at Larger Batch Sizes

We compare the latency breakdown of Llama-3.1-8B-Instruct in BF16 and DF11 formats using varying token batch sizes on a single A100-40GB GPU. For each configuration, we measure the latency of each component during the forward pass over 10 runs and report the average in Figure 5. Compared to the original model, the DF11-compressed version introduces additional latency due to the decompression of transformer blocks and the language modeling head. This decompression adds a constant overhead that is independent of the token batch size. As a result, increasing the batch size amortizes the decompression cost, leading to a substantially smaller gap in overall inference time.

4.2.2 DF11 Decompression Is Significantly Faster Than CPU-to-GPU Transfer and ANS

We compare the latency and throughput of the DF11 decompression kernel against two baselines: CPU-to-GPU transfer and ANS (Asymmetric Numeral System) decompression [5] from the NVIDIA nvCOMP library [1], using weight matrices of varying sizes. The motivation is to evaluate two strategies for memory-constrained inference: offloading weight matrices to CPU memory and trans-

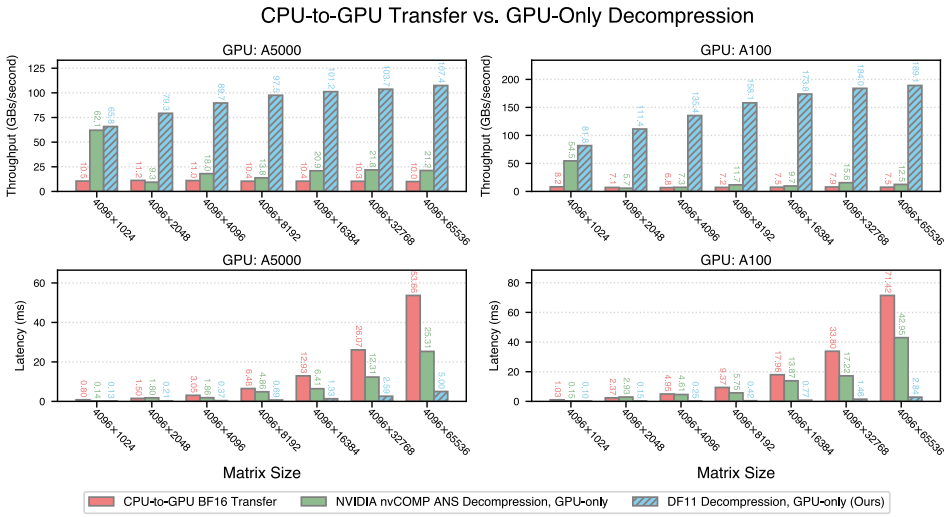

Figure 6: Throughput (top two) and latency (bottom two) comparisons between transferring BFloat16 matrices from CPU to GPU and decompressing the same matrices using the NVIDIA nvCOMP ANS library and our proposed DFloat11 kernel, across matrix sizes and GPU types.

ferring them to the GPU as needed, or storing compressed matrices on the GPU and decompressing them on demand. For this study, we use the weight matrix of the language modeling head from Llama-3.1-8B-Instruct and slice it into varying sizes. The results are shown in Figure 6. As illustrated, DF11 decompression is significantly more efficient than both baselines, achieving up to 24.87× higher throughput or lower latency than CPU-to-GPU transfer, and up to 15.12× faster than NVIDIA nvCOMP decompression. In addition to speed, DF11 also provides better compression ratio than nvCOMP, reducing model size by approximately 70% compared to around 78% for nvCOMP. Notably, DF11 decompression throughput improves with larger matrix sizes due to increased GPU thread utilization.

5 Related Works

Data Formats for Model Weights LLM weights are typically stored in compact floating-point formats such as FP16 or BF16. FP16 allocates 1 sign bit, 5 exponent bits, and 10 mantissa bits, whereas BF16 uses 1 sign bit, 8 exponent bits, and 7 mantissa bits. Compared to FP16, BF16 offers a wider dynamic range at the cost of precision, which improves numerical stability and mitigates overflow issues during training [8, 19].

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()` [2] and its following works. Formats with a stronger emphasis on efficiency, such as FP4, INT4, NF4 [3], and AF4 [39], use only 4 bits. In this work, we primarily focus on formats with $\geq$8 bits, as benchmark literature [37, 10, 23] often suggests that 8-bit quantization results in negligible performance drop—though we show in Section 2 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 [6, 21, 7] 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* [13] is a foundational work, applying Huffman coding [17] to quantized CNN models and achieving an additional ∼22% compression gain for model checkpoints. *ZipNN* [16] extended this idea to language models, comparing its results to

classic lossless compression tools such as zlib [4] and zstd[3] and demonstrated superior compression gains. However, this line of work 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 [32]—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* [14] 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"*.[4] 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 6).

Another work referencing NeuZip is *Huff-LLM* [40], 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.

6 Conclusion

In this work, we present *Dynamic-Length Float* (DFloat) as a lossless compression data format for LLM weights. To the best of our knowledge, DFloat is the only data format capable of reducing memory footprint while remaining compatible with efficient GPU inference. Specifically, we evaluate several popular LLMs using the 11-bit DFloat format (DF11), alongside custom-developed GPU kernels tailored for this format. Empirical results suggest that DF11-based compression significantly lowers hardware demands for serving LLMs, while introducing reasonable processing overhead in most practical use cases.

Acknowledgements

We thank Oscar Wu for insightful discussions and `xMAD.ai` for providing computational resources.

References

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

[2] 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.

[3] 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.

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

[3] `https://github.com/facebook/zstd`

[4] `https://developer.nvidia.com/nvcomp`

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

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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*.

[16] 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.

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

[18] 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.

[19] 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.

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

[21] 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.

[22] 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.

[23] 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.

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

[25] 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.

[26] Gemma Team, Aishwarya Kamath, Johan Ferret, Shreya Pathak, Nino Vieillard, Ramona Merhej, Sarah Perrin, Tatiana Matejovicova, Alexandre Ramé, Morgane Rivière, et al. Gemma 3 technical report. *arXiv preprint arXiv:2503.19786*, 2025.

[27] Gemma Team, Morgane Riviere, Shreya Pathak, Pier Giuseppe Sessa, Cassidy Hardin, Surya Bhupatiraju, Léonard Hussenot, Thomas Mesnard, Bobak Shahriari, Alexandre Ramé, et al. Gemma 2: Improving open language models at a practical size. *arXiv preprint arXiv:2408.00118*, 2024.

[28] Mistral AI Team. Codestral. https://mistral.ai/news/codestral, May 2024.

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

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

[31] Qwen Team. QwQ: Reflect Deeply on the Boundaries of the Unknown. https://qwenlm.github.io/blog/qwq-32b-preview/, November 2024.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

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

[40] 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.

Appendix

A Frequency Distribution of BFloat16 Values

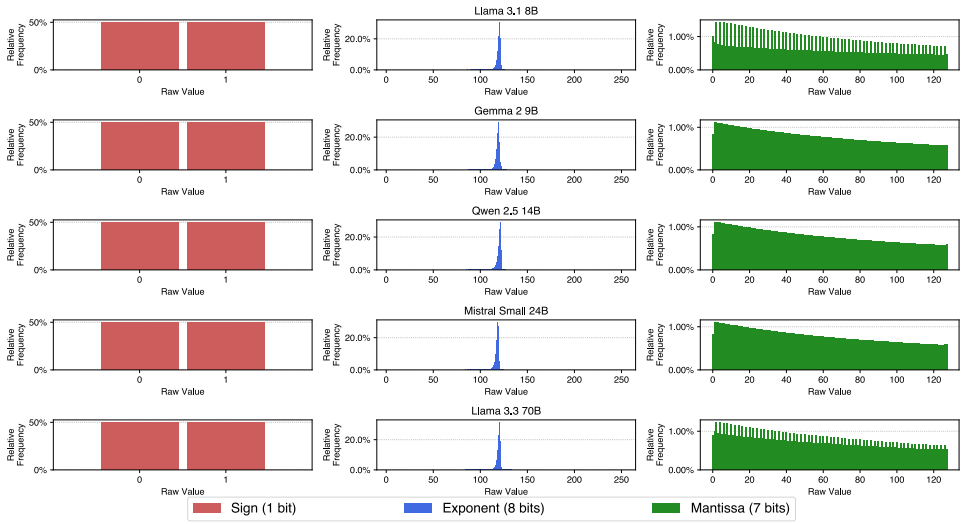

Figure 7: Relative frequency distribution of sign, exponent, and mantissa values in the BFloat16 weights of all linear projection layers across various LLMs.

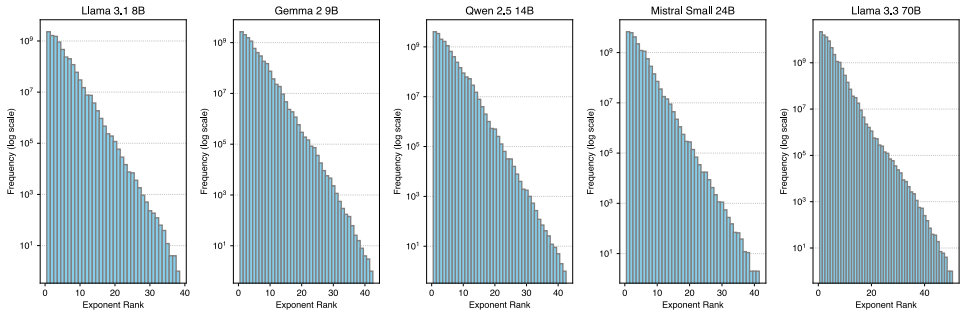

Figure 8: Distribution of BFloat16 exponent values across various models. The frequency of exponent values (shown in log scale) decays rapidly with exponent rank.

B Hardware for Experiments

Table 4: System specifications of servers used for experiments.

	GPU	GPU Memory	CPU	CPU Memory
Server 1	NVIDIA Tesla T4	15360MiB	Intel Xeon Platinum 8259CL	187GB
Server 2	NVIDIA RTX A5000	24564MiB	AMD EPYC 7513 32-Core	504GB
Server 3	NVIDIA A100	40960MiB	AMD EPYC 7742 64-Core	1.48TB
Server 4	NVIDIA Quadro RTX 8000	49152MiB	AMD EPYC 7742 64-Core	1.48TB

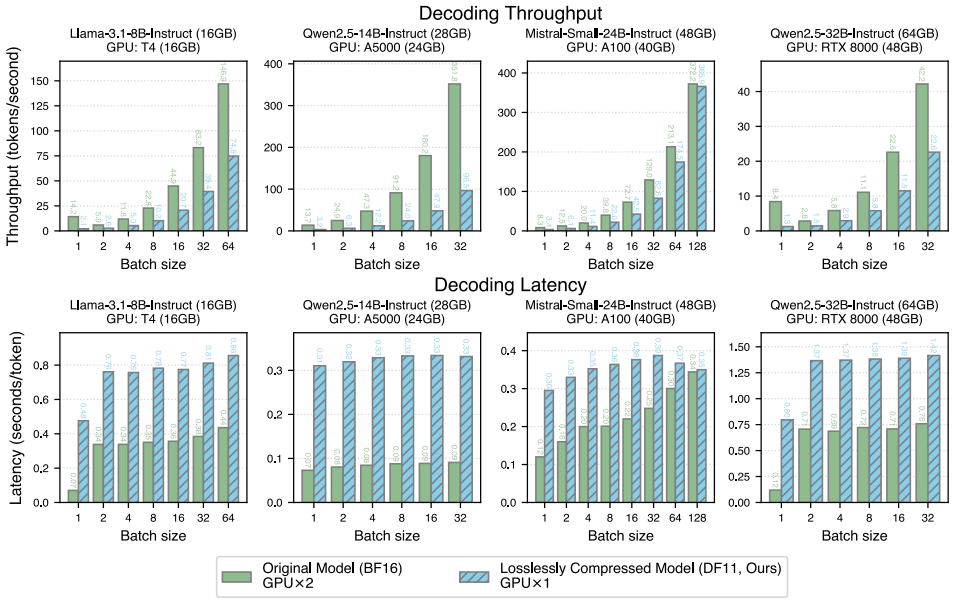

Figure 9: Comparison of average latency and throughput for token decoding between the original (BF16) models and their losslessly compressed (DF11) counterparts. The DF11 models are run on a single GPU, while the BF16 models require two GPUs due to memory constraints.

C GPU Inference Efficiency Comparison: BF16 vs. DF11

We present the GPU inference efficiency of BF16 and DF11 models in Figure 9, for various models, GPUs, and batch sizes. Due to GPU memory constraints, the BF16 models are run with two GPUs while the DF11 models are run with a single GPU.

D Impact of Lossy Quantization

Table 5: 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