Waiting to Decompress: The Economics of LLM-Based Compression

Andreas Kipf, Tobias Schmidt*, Ping-Lin Kuo, Skander Krid, Moritz Rengert, Luca Heller, Andreas Zimmerer, Mihail Stoian, Varun Pandey, Alexander van Renen
University of Technology Nuremberg *Technical University of Munich

ABSTRACT

The latest AI summer brought us large language models (LLMs)—generative powerhouses with surprising reasoning, coding, and text generation abilities. Unsurprisingly, the database community started applying them to every aspect of data processing, including data compression. However, these methods come with painfully low inference speed—and recent papers have conveniently left out runtime considerations.

In this paper, we address this gap and ask: When will LLM-based compression methods become economically viable? To answer this, we first benchmark existing approaches and optimize them for speed. We then introduce a cost model showing that, given current compute and storage costs, LLM-based compression would take 10 years to amortize its computational expense. Compared to traditional compression methods, LLM-based compression would take 120 years to break even. Finally, we project future trends and speculate that LLM-based compression could become economically viable within the next decade with anticipated improvements in hardware and model efficiency.

1 INTRODUCTION

Data storage is a major driver of cloud cost, especially for infrequently accessed (cold) data. Services like Amazon S3 charge around 20 USD per terabyte per month. For many organizations, cold data is retained for compliance or reproducibility, yet it may only be read a few years down the road or never. A petabyte of such archival data incurs over 200,000 USD annually in storage fees, regardless of how little it is used.

Modern data warehouses mitigate storage costs using columnar formats with efficient lightweight encodings (e.g., FSST [3]) in combination with fast general-purpose compression algorithms such as Snappy, LZ4, or Zstandard (zstd) when storing data on object storage services like Amazon S3 or Azure Blob Storage.

Recent work on adaptive compression [35, 38] has shown that varying compression effort based on data access frequency, i.e., compressing cold data more aggressively, can yield significant storage savings without compromising performance. However, even these adaptive approaches remain fundamentally limited by the capabilities of traditional compression algorithms.

In this paper, we explore a more radical design point: using large language models (LLMs) as text compressors. By leveraging the predictive capabilities of LLMs, we can compute a probability distribution over the next token in a sequence and encode text using arithmetic coding [29]. On English text from Wikipedia, this approach achieves compression ratios that outperform traditional baselines by more than 2×.

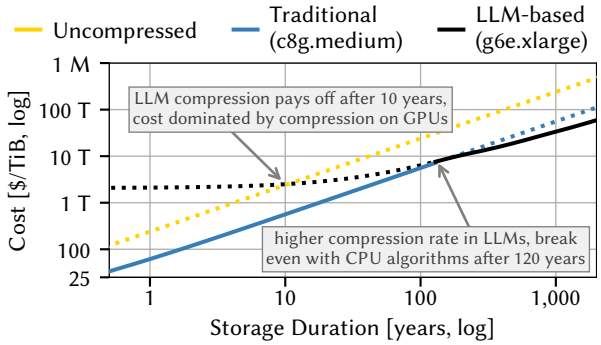


Figure 1: The cost of storing 1 TiB of (Wikipedia) data on Amazon S3 over time using traditional and LLM-based compression algorithms. The price considers the S3 storage cost ($240 per TiB per year) plus the cost of renting an EC2 instance for compressing the data.

However, this improvement comes at a cost: LLM inference is orders of magnitude slower and more expensive than traditional compression. We therefore do not claim that LLM-based compression is practical today for general-purpose use. Instead, we ask two forward-looking systems questions:

- **Q1:** If we compress data today and plan to decompress it in x years, which compression method minimizes the total cost over that time horizon, accounting for both storage and compute costs?
- **Q2:** Given current trends in GPU pricing and performance (e.g., TFLOPS per USD), when will LLM-based compression become economically practical?

To answer these questions, we develop a cost model that incorporates cloud storage and compute costs, along with the inference efficiency of LLMs. As shown in Figure 1, with today's hardware, LLM-based compression is only economically viable for datasets that remain cold for extended periods. Notably, both compression and decompression can be parallelized using cloud-native, serverless infrastructure. In such settings, (de)compression latency becomes a matter of dollars rather than seconds.

Section 3 demonstrates how LLMs can be applied to data compression and explores strategies to make this approach practical. Today, the main limitations of LLM-based compression are GPU inference speed and hardware cost. To mitigate these bottlenecks, we introduce several system-level optimizations that accelerate inference by more than an order of magnitude. Section 4 presents a

 16th Annual Conference on Innovative Data Systems Research (CIDR '26). January 18-21, Chaminade, USA

comparative analysis between traditional CPU-based compression algorithms and LLM-based approaches. Finally, Section 5 discusses further opportunities and examines current trends in GPU development and specialized inference hardware.

We conclude that LLM-based compression is not yet ready to replace lightweight, CPU-based compression algorithms for hot or frequently accessed data. Nevertheless, it introduces a promising new dimension to the compression design space: as inference becomes faster and more affordable, LLM-based methods could become a compelling option for long-term archival of cold datasets.

2 RELATED WORK

Compression can be considered as a two-step process which includes modeling and coding [30]. Modeling analyzes data to find redundant information, uses it to build a model and combines the model with encoders that helps the compressor reach closer to the entropy [31]. Popular entropy encoders include Huffman [15], arithmetic encoding [29], range encoding [23], and asymmetric numeral systems (ANS) [10]. Several statistical models, in particular *probabilistic* ones, have been proposed in the past [5, 17, 21] that take into account the past symbols to generate probabilities which are subsequently encoded.

Large language models, specifically those utilizing the transformer [34] architecture, resemble these *probabilistic* models as they perform the identical task of predicting the subsequent token (or symbol) based on the preceding symbols within the context window of the transformer. As a result, recent research has investigated the potential of LLMs for data compression.

LLMZip [33] was the first work to propose using large language models as compressors. The authors used LLaMA-7B as the predictive model and integrated it with zlib, Huffman coding, and arithmetic coding. FineZip [25] applies parameter-efficient fine-tuning [22] as an online memorization step to make the text being compressed more probable for LLMs. ALCZip [37] exploits the LLM's tokenizer to build a frequency dictionary which is subsequently encoded using Huffman coding. L3TC [41] uses a combination of RWKV [28] as the predictor LLM, an outlier-aware tokenizer, and high-rank reparameterization strategy to achieve effective compression. Although these studies have primarily focused on text compression, there are several works that have also investigated different modalities. In [7], the authors show that LLMs are capable of compressing data of several modalities, not just text.

Inference. LLM inference has high computational and memory requirements. To date, various techniques have been proposed to make inference more efficient by reducing memory and computational costs. These techniques can be divided into algorithmic and system-related methods of inference optimization. Algorithm-level improvements include speculative decoding and KV-cache optimization [36] as well as model-level optimizations such as quantization, attention mechanisms, and model compression [42]. As for system-level improvements, Wan et al. [36] propose a specific hardware configuration; we will cover a subset of these in Section 5. Notably, previous work on data compression with LLMs has focused on improving compression ratio rather than compression speed [37]. Some works dealt with speeding up the inference, e.g., Mittu et al. [26] quantize the model to speed up the compression.

3 LANGUAGE MODELING IS COMPRESSION

Language models, e.g., transformers [34], assign probabilities to tokens, which can be converted into compressed representations using arithmetic coding [29]. The better a model is at assigning a high probability to the correct next token, the fewer bits are needed to encode the data.

3.1 Reducing Cross Entropy

Language modeling is compression in a very literal, information-theoretic sense. As a language model improves, its predicted probability distribution becomes closer to the true data distribution. This does not reduce the true entropy of the language itself (which is fixed), but it reduces the cross-entropy between the model and the data. In other words, a better model assigns higher probability to the correct tokens, which means fewer bits are needed to encode each token—enabling better compression.

If a model could always predict the correct next token, the effective cross-entropy would be zero—and we would not need to store any additional information beyond the first token to reconstruct the original data, since the model would already "know" exactly what comes next. In practice, however, models are not perfect: they assign high but not total probability to the correct token (e.g., 0.9) and some probability to incorrect alternatives (e.g., 0.1). As a result, we still need to encode which token was actually chosen, which requires a few additional bits per token for a lossless reconstruction of the original data.

3.2 Lossless Model-Based Compression

Figure 2 shows the compression (encoding) process. For every input token, we perform the following two steps: (1) we use a large language model to compute the probability of the token based on preceding tokens (context), and then (2) encode this probability using arithmetic coding. The arithmetic encoder produces a compressed bitstream that encodes the probability of every input token, depending on the context. During decompression, we extract these probabilities to determine the correct tokens using the model.

Token Prediction. We rely on the base variants of language models trained with a causal language modeling objective[1] which preserves the left-to-right predictive structure needed for entropy coding. Given some input tokens as context, the language model predicts the probability distribution for the next token. This process is deterministic; therefore, the LLM computes the same probabilities every time it is invoked for the same context.

Consider the example in Figure 2: we want to compress the text "`the quick brown fox`". The text is split into four tokens "`the`", "`quick`", "`brown`", and "`fox`". For encoding the last token, we take the preceding tokens ("`the quick brown`") as context and task the model to compute the probabilities of all possible tokens that follow these three tokens. For the given example, "`fox`" has the highest probability as it is a common phrase that the models have encountered during training. Given a probability and the context, the input token can be reconstructed by recomputing the probabilities using the model. For "`fox`", we store the probability

[1] Causal language modeling trains models to predict the next token given all previous tokens, i.e., $P(x_t \mid x_{<t})$. This is the standard objective used in autoregressive models like GPT [4].

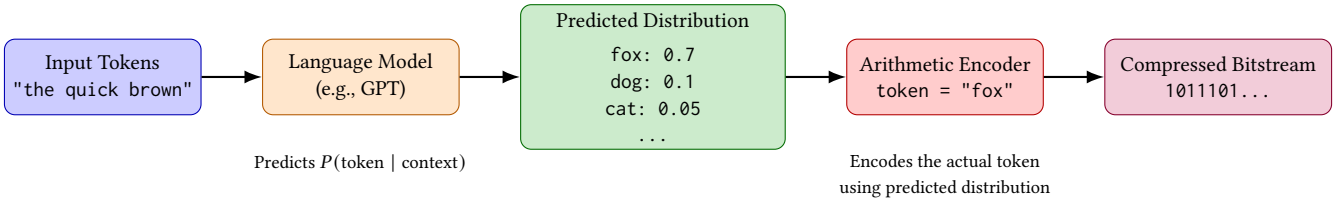


Figure 2: Encoding process using a language model and arithmetic coding. Given a context (e.g., "`the quick brown`"), the model predicts a probability distribution over the next token. Arithmetic coding uses this distribution to encode the actual next token (e.g., "`fox`"), producing a compact binary representation.

0.7, and we can map this probability back to the input token by invoking the model again. Note that, for a one-to-one mapping, it is necessary to store the cumulative probability range for the token. For instance, for "`fox`", we encode $[0.0, 0.7)$, while for "`dog`" or "`cat`", we need to use $[0.7, 0.8)$ and $[0.8, 0.85)$, respectively.

Instruction-tuned or chat-optimized models, such as those used in public APIs like ChatGPT, modify the original output distribution to follow prompts or align with user intent. This process often introduces non-determinism and thus distorts the original token probabilities needed for lossless coding. In public APIs, this non-determinism is often controlled by the *temperature* parameter, which adds stochastic noise and prevents the model from always choosing the most likely next token. Crucially, we bypass this high-level, non-deterministic interface. Instead, we utilize the underlying models at a low level to ensure deterministic behavior.

Another source of non-determinism is batch-size variance: when the batch size changes, the shape of the matrices being multiplied in the GPU changes. The GPU handles these shapes differently, which can lead to a different execution order. This results in small numerical differences due to floating-point arithmetic and hence can cause the model to output slightly different probabilities. This point has been highlighted in a recent blog post by He and Thinking Machines Lab [12]. Fixing the batch size ensures deterministic behavior, enabling lossless compression and decompression.

Arithmetic Coding. Arithmetic coding is a technique that leverages the model's predicted probabilities to compress data extremely efficiently. It approaches the information-theoretic lower bound [31], achieving compression rates that reflect the model's cross-entropy with the true distribution. It takes the cumulative probability distribution of the token probabilities computed by the model and encodes it. For the given example, encoding "`fox`" will require fewer bits than "`dog`" or "`cat`", as it is more likely.

In a nutshell, arithmetic coding [29] works like this: it represents the entire sequence as a subinterval of the real number line between 0 and 1. Starting with the full interval $[0, 1)$, the encoder progressively narrows this range based on the predicted probability distribution of each token. For each step, the interval is partitioned according to the probabilities, and the subinterval corresponding to the actual next token is selected. More probable tokens result in smaller updates to the interval and contribute fewer bits to the final encoding. The process continues token by token until the sequence is fully encoded. The result is a compact binary representation whose length reflects the total cross-entropy between the model's predicted distribution and the actual sequence.

Decompression. The steps required to decompress (decode) data closely mirror those used during compression. To reconstruct the current token, we provide the LLM with all preceding, already decompressed tokens as context. The LLM then computes the conditional probability distribution for the next expected token. The arithmetic decoder uses the decoded probability from the bitstream and the model's computed probability distribution to reconstruct the correct token, effectively mapping the extracted probability back to its corresponding token. Finally, we append the newly decompressed token to the context and repeat the entire process for the subsequent token.

3.3 Custom Models vs. Foundation Models

When training a language model from scratch on a specific dataset, we directly approximate its true distribution. With sufficient capacity and training, custom models can approach the dataset's entropy limit [13]. In contrast, foundation models (e.g., GPT [4]) are trained on diverse, general-purpose data and are not tailored to any single dataset. This leads to a distributional mismatch, which makes their predictions less accurate for compression than custom models.

Nevertheless, foundation models often outperform traditional compressors such as zstd, thanks to their ability to capture long-range and semantic structure, even without fine-tuning [7]. Compared to custom-trained models—which require storing additional model parameters (analogous to the dictionary in zstd)—we often treat foundation models as "free" in terms of storage, assuming they are already available and not counted against compression size. So while foundation models do not perfectly model any specific dataset, they still offer a strong starting point for compression.

An important caveat, however, is that deterministic decompression requires access to the exact same model used during encoding—including its weights, tokenizer, and preprocessing pipeline. Any change, even in floating-point arithmetic or token handling, could result in mismatches during decoding, making lossless reconstruction impossible.

3.4 System-Level Considerations

For an efficient implementation of LLM-based compression, we must address both the computational demands of large language models and the practical constraints of real-world hardware.

Model Choice. As we will show later, the choice of language model directly impacts compression rate, latency, and resource usage. Smaller models are faster but less accurate, while larger ones may offer better compression at the cost of increased memory and

compute. Precision format (e.g., FP16) and quantization techniques also impact performance. We limit our experiments to relatively small LLM variants with up to 8B parameters to ensure they fit on a single GPU machine.

Parallelism and GPU Utilization. Although LLM compression for a single stream of text is inherently sequential, compression throughput can be improved by chunking the data and processing it in batches. For instance, given an input text with 100,000 tokens, we can divide it into 10 chunks with 10,000 tokens each. The GPU can process these chunks in 10 batches in parallel, requiring only 9,999 calls that predict the next tokens for all batches concurrently. In contrast, processing the entire file sequentially would entail 99,999 steps. Theoretically, parallelizing scales linearly on a GPU, and we can fully exploit its computational power. Crucially, for each token, the model must only see the preceding tokens within the same chunk as context—this ensures that decompression can be parallelized in the same way. In addition, the compressed file must include the first token of each chunk (uncompressed) in its header.

Decoupling Arithmetic Coding. In practice, to really saturate the GPU and avoid idle time, it is crucial to minimize the overhead of the CPU work. In the context of LLM-based compression, the most CPU-heavy computation is arithmetic coding. To avoid bottlenecks, it should be decoupled from the GPU inference loop, allowing the two to run concurrently. One approach is to queue the predicted probability distributions and process them asynchronously.

KV Caching. With LLM inference, a KV cache can be used to store the key/value matrices for each layer to avoid recomputing the previously processed context. However, without proper management, the cache can grow unbounded and consume excessive memory. In our implementation, when the processing context length exceeds the predefined context window size, the cache is discarded and computation restarts with only 10% of the context window size. Doing so not only prevents excessive memory usage, it also improves inference throughput. However, this slightly impacts the compression factor because of the discarded context.

Tunable Parameters. Compression rate and throughput depend on several key parameters, which will be explored in detail in the next section. These include (a) the batch size, which affects GPU utilization, and (b) the context window size, since longer windows improve compression but require more memory and computation. In addition, hardware-level parameters—such as available GPU memory and memory bandwidth—constrain the choice of model and influence the overall throughput of the compression pipeline.

4 EVALUATION

Setup. The LLM compression experiments[2] use a `g6e.xlarge` virtual machine instance equipped with an NVIDIA L40S GPU, 4 vCPUs, and 32 GiB of system memory; the L40S provides 48 GiB of GPU memory and is optimized for generative AI workloads. For the traditional algorithms, we use a `c8g.medium` EC2 instance with an AWS Graviton4 Processor and 2 GiB of memory. Note that we rent a virtual instance that allocates only 1 vCPU to accommodate for the single-threaded standard compression algorithms. The GPU instance costs $0.804 per hour and the Graviton instance $0.019 per hour, when reserving the instance for three years.

[2]https://github.com/utndatasystems/compression-economics

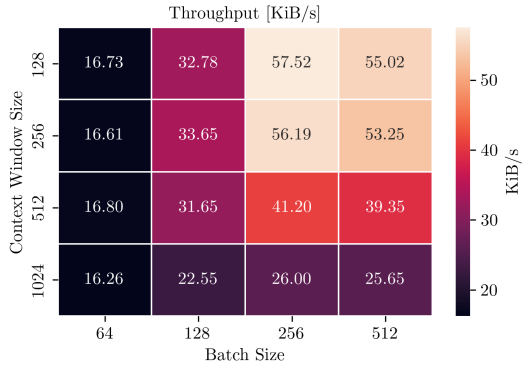


Figure 3: Model inference throughput for the first 1M tokens of text8 using the LLM Qwen2.5-0.5B, with batch sizes ∈ {64, 128, 256, 512} and context window lengths ∈ {128, 256, 512, 1024}. Higher batch sizes and smaller context windows increase inference throughput.

Datasets. We utilize two datasets coming from different contexts: English language, namely the well-established `text8` dataset extracted from Wikipedia (100 MiB) and Python code, namely the first 100 MiB from PyTorrent [2], which is a corpus of Python libraries.

Hyper Parameter Optimization. Our approach exhibits a trade-off in throughput, influenced by batch size and context size. We perform a grid search over both parameters to identify the optimal combination. The results are shown as a heatmap in Figure 3. A context window of 128 tokens combined with 256 batches yields the best results. All experiments leverage KV caching, which boosts performance significantly: from 2.58 KiB/s without caching to 57.52 KiB/s with it. The context size also directly affects the compression factor; larger windows improve the model's accuracy to predict the next token. For instance, increasing the window size from 128 to 1024 tokens improves the compression factor from 7.77 to 8.01. However, this improvement comes at a cost: the throughput reduces by more than a factor of two.

Are We There Already? Figure 4 depicts the results of our experiments for both datasets with various compression techniques. For LLM-based compressors, we further evaluated different models and model sizes. In general, LLM-based techniques achieve higher compression factors, with larger models taking the lead. One notable exception here is `distilgpt2` on the Code dataset, which achieves a worse compression factor compared to traditional techniques at a much higher cost.

The better compression factor of LLM-based techniques comes at a higher cost both in terms of compression time and, therefore, also compression costs: Disregarding `distilgpt2`, even the cost to compress each dataset with the cheapest LLM-based compressor (`qwen2.5 0.5B`) is more than two orders of magnitude higher than that of the most expensive traditional technique (`xz -9e`).

It is apparent that current costs are not in favor of LLM-based compressors. However, when we examine the development of GPU and storage costs in the cloud over the past 5 years and extrapolate

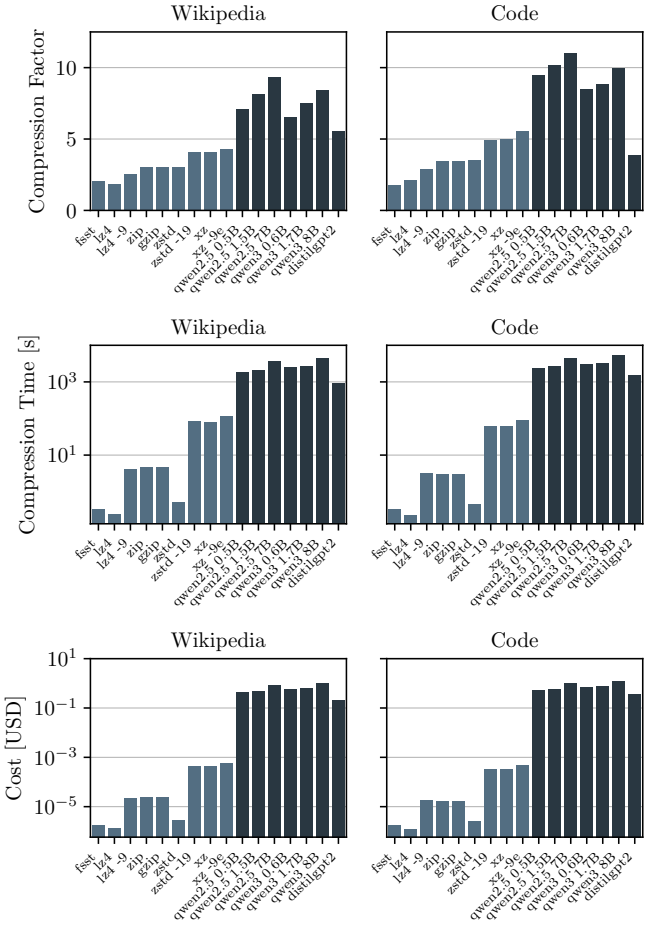
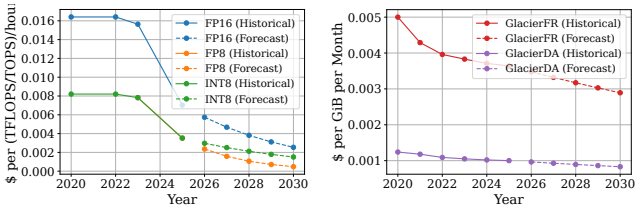


Figure 4: Compression factor, time, and cost for compressing 100 MiB of English Wikipedia and 100 MiB of Python code using various compressors. Classical methods were run on AWS `c8g.medium` ($0.019 / h), and LLM-based methods on AWS `g6e.xlarge` with a GPU ($0.804 / h), assuming costs for 3-year reserved instances.



(a) Compute Cost **(b) Storage Cost**

Figure 5: Compute and storage costs for the past 5 years and projected for the next 5 years.

them for the next 5 years, as we did in Figure 5, we can see a clear trend that might make storing data compressed with LLMs pay off earlier in the future: Despite, or because of, the increased demand for GPUs since the surge of LLMs, normalized GPU costs have dropped considerably, especially in the last two years. In contrast to that, storage cost pricing for archival storage is dropping at a much slower rate. If both trends persist, compressing with LLM-based techniques will become significantly cheaper over time, while the cost of storing data will remain relatively constant, allowing for a shorter amortization period.

5 DISCUSSION

Model-based compression with entropy coding shows strong potential as a future alternative to traditional methods. However, its practical viability depends on a range of factors—model efficiency, system-level optimizations, and hardware constraints—which we explore in the following discussion.

Specialized Models and Fine-Tuning. While our evaluation is based on standard, general-purpose language models, one could improve compression by using custom, domain-specific models tailored to the target data. For instance, Qwen has released a code-focused variant, but not the corresponding base model—limiting its applicability for our method.

Another approach is to fine-tune a foundation model on the target dataset to better match its distribution. This can improve compression rates by reducing the cross-entropy between the model and the data. However, fine-tuning introduces additional parameters—such as adapter weights—which must be stored alongside the compressed data. Unlike the foundation model, which we treat as "free" under the assumption that it is already available, these task-specific parameters represent a non-negligible overhead and must be accounted for in the total compression cost.

Recent approaches based on parameter-efficient fine-tuning have shown to be particularly promising for adapting large language models without incurring the cost of storing full model weights. For example, using LoRA [14] to fine-tune a 7B parameter model typically introduces only 0.1–0.5% additional parameters—roughly 7–35 million parameters, depending on the configuration. While this is significantly smaller than a full model checkpoint, it still adds tens of megabytes that must be stored and transmitted to enable lossless decompression.

Diffusion Models. Modern diffusion-based language models [8, 9, 16] offer a compelling alternative to autoregressive architectures by enabling parallel token generation, which significantly improves inference speed. Score Entropy Discrete Diffusion (SEDD) [8] matches GPT-2 on standard NLP benchmarks while offering up to 4.5× lower latency, though it lags slightly on short-context tasks. Concurrently, Deschenaux and Gulcehre [9] propose a self-distillation method enabling diffusion models to generate 32 tokens in parallel, achieving up to 8× faster inference than KV-cached autoregressive models without compromising quality. Together, these advances position diffusion models as a promising path for efficient compression.

Early-Exit Networks. A promising direction to further increase the efficiency of inference is early-exit networks [32]. The key idea is that not every token needs to go through all layers of a model. This is achieved by adding exit heads, which are small classifiers trained at intermediate layers which determine whether a token should proceed to the next layer.

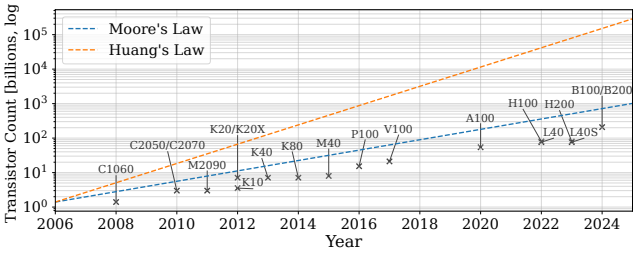


Figure 6: Server-grade NVIDIA GPUs follow Moore's Law.

One issue with this strategy is divergence in execution when computing the probabilities for multiple tokens in parallel. Some tokens within the same batch go through all layers, while others exit early. This adds control flow overhead, causes irregular memory access patterns, and most importantly, leads to idle batch slots, and hence GPU underutilization. This is the reason why current implementations only see a reduction in FLOPs but not in wall-clock time [20]. Current LLM frameworks such as vLLM implement advanced scheduling techniques to optimize throughput for requests that produce different numbers of tokens through continuous batching [40]. These optimizations are designed for models where all tokens undergo the same amount of processing. Early-exit networks break this uniformity by allowing tokens to stop after varying numbers of layers.

This issue is not new to the data management community. A similar issue exists when processing data with SIMD instructions on modern CPUs. Lang et al. [19] addressed this issue by refilling SIMD lanes with new data items.

Quantization and Pruning Techniques. The current drawback of LLM-based compression is inference latency. Two promising directions to mitigate this bottleneck—while also expanding the practical applicability of such methods—are quantization and pruning, which are orthogonal techniques targeting different aspects of inference efficiency.

On the one hand, quantization reduces the precision of the weights and activations, e.g., from FP16 to FP8. This reduces both model size and enables faster inference on specialized hardware [24], or via optimized libraries, e.g., NVIDIA's TransformerEngine [6]. On the other hand, pruning deals with the actual values of the weights, e.g., weights or layers that do not influence the actual model accuracy. Similar to quantization, this can lower the computational and memory footprint of the model. However, the resulting sparsity patterns are to be supported by the hardware itself (unless more regular sparsity patterns are employed [39]).

Hardware Trends. There are several notable trends in recent GPU development. First, Figure 6 shows that high-end data center GPUs continue to achieve transistor counts and compute densities that track closely with Moore's Law projections, in contrast to Huang's Law, which indicated a faster growth.

This growth increasingly relies on architectural innovations that go beyond simple transistor miniaturization, such as using multi-die packages and large data-center form factors, which allow GPUs to occupy a larger silicon area. For example, NVIDIA's B200 GPU integrates 1600 mm^2 silicon across two dies within a single package to achieve approximately 208 billion transistors [27].

Wafer scale processors push this even further as evident by Cerebras's third-generation Wafer Scale Engine (WSE-3). The WSE-3 is a 46,225 mm^2 full wafer chip containing 4 trillion transistors [18] representing fundamentally different scaling, vastly exceeding conventional GPU die areas. These extreme-scale transistor counts allow for more specialized hardware features (e.g., tensor cores, sparsity aware units) helping accelerate inference workloads.

Another notable trend is the divergence between general-purpose and domain-specific accelerators. NVIDIA continues to design general-purpose GPUs integrating CUDA, ray-tracing, and tensor cores in a unified architecture. This allows it to cater to multiple application domains, including graphics, scientific computing, AI, robotics, and autonomous driving. In contrast, cloud providers have developed domain-specific AI accelerators, including Google TPUs [11] and the AWS Inferentia [1] family, which are optimized specifically for matrix and tensor operations.

If these trends in GPU hardware innovation continue to hold, we can expect compression throughput and cost to drop significantly over the next few hardware generations, narrowing the gap between LLM-based and traditional methods.

6 CONCLUSIONS

While prior work on LLM-based compression has primarily focused on improving or evaluating compression rates—often neglecting runtime and system-level considerations—we have presented one of the first comprehensive studies that offers a holistic view, addressing both compression rate and real-world performance. We show that, although LLM-based compressors can surpass traditional algorithms in terms of compression ratio, their economic feasibility is currently hindered by high inference latency and compute costs.

However, based on current trends in compute and storage costs and the continued adherence to Moore's Law in GPU development, we anticipate that the economic break-even point for LLM-based compression could be reached within the next decade.

ACKNOWLEDGMENTS

The authors acknowledge the use of OpenAI's ChatGPT for editing and polishing the text and figures for spelling, grammar, and stylistic improvements. Additionally, ChatGPT was utilized for support in basic coding tasks.

REFERENCES

[1] Amazon Web Services. 2025. *AWS Inferentia.* https://aws.amazon.com/ai/machine-learning/inferentia/

[2] Mehdi Bahrami, N. C. Shrikanth, Shade Ruangwan, Lei Liu, Yuji Mizobuchi, Masahiro Fukuyori, Wei-Peng Chen, Kazuki Munakata, and Tim Menzies. 2021. PyTorrent: A Python Library Corpus for Large-scale Language Models. arXiv:2110.01710 [cs.SE] https://arxiv.org/abs/2110.01710

[3] Peter A. Boncz, Thomas Neumann, and Viktor Leis. 2020. FSST: Fast Random Access String Compression. *Proc. VLDB Endow.* 13, 11 (2020), 2649–2661.

[4] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. In *NeurIPS.*

[5] John G. Cleary and Ian H. Witten. 1984. Data Compression Using Adaptive Coding and Partial String Matching. *IEEE Trans. Commun.* 32, 4 (1984), 396–402. doi:10.1109/TCOM.1984.1096090

[6] NVIDIA Corporation. 2022. NVIDIA Transformer Engine. https://docs.nvidia.com/deeplearning/transformer-engine/user-guide/. Accessed: 2025-08-05.

[7] Grégoire Delétang, Anian Ruoss, Paul-Ambroise Duquenne, Elliot Catt, Tim Genewein, Christopher Mattern, Jordi Grau-Moya, Li Kevin Wenliang, Matthew Aitchison, Laurent Orseau, Marcus Hutter, and Joel Veness. 2024. Language Modeling Is Compression. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net. https://openreview.net/forum?id=jznbgiynus

[8] Justin Deschenaux and Caglar Gulcehre. 2024. Promises, Outlooks and Challenges of Diffusion Language Modeling. *CoRR* abs/2406.11473 (2024).

[9] Justin Deschenaux and Caglar Gulcehre. 2025. Beyond Autoregression: Fast LLMs via Self-Distillation Through Time. In *ICLR*. OpenReview.net.

[10] Jarek Duda. 2009. Asymmetric numeral systems. *CoRR* abs/0902.0271 (2009). arXiv:0902.0271 http://arxiv.org/abs/0902.0271

[11] Google Cloud. 2025. *Introduction to Cloud TPU*. https://docs.cloud.google.com/tpu/docs/intro-to-tpu

[12] Horace He and Thinking Machines Lab. 2025. Defeating Nondeterminism in LLM Inference. *Thinking Machines Lab: Connectionism* (2025). doi:10.64434/tml.20250910 https://thinkingmachines.ai/blog/defeating-nondeterminism-in-llm-inference/.

[13] Tom Henighan, Jared Kaplan, Mor Katz, Mark Chen, Christopher Hesse, Jacob Jackson, Heewoo Jun, Tom B. Brown, Prafulla Dhariwal, Scott Gray, Chris Hallacy, Benjamin Mann, Alec Radford, Aditya Ramesh, Nick Ryder, Daniel M. Ziegler, John Schulman, Dario Amodei, and Sam McCandlish. 2020. Scaling Laws for Autoregressive Generative Modeling. *CoRR* abs/2010.14701 (2020).

[14] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2022. LoRA: Low-Rank Adaptation of Large Language Models. In *ICLR*. OpenReview.net.

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

[16] Mohamed Amine Ketata, David Lüdke, Leo Schwinn, and Stephan Günnemann. 2025. Joint Relational Database Generation via Graph-Conditional Diffusion Models. *CoRR* abs/2505.16527 (2025). arXiv:2505.16527 doi:10.48550/ARXIV.2505.16527

[17] Byron Knoll. 2014. CMIX: a lossless data compression program. https://www.byronknoll.com/cmix.html

[18] Yudhishthira Kundu, Manroop Kaur, Tripty Wig, Kriti Kumar, Pushpanjali Kumari, Vivek Puri, and Manish Arora. 2025. A Comparison of the Cerebras Wafer-Scale Integration Technology with Nvidia GPU-based Systems for Artificial Intelligence. *CoRR* abs/2503.11698 (2025). arXiv:2503.11698 doi:10.48550/ARXIV.2503.11698

[19] Harald Lang, Linnea Passing, Andreas Kipf, Peter Boncz, Thomas Neumann, and Alfons Kemper. 2020. Make the most out of your SIMD investments: counter control flow divergence in compiled query pipelines. *VLDB J.* 29, 2-3 (2020), 757–774. doi:10.1007/S00778-019-00547-Y

[20] Xuan Luo, Weizhi Wang, and Xifeng Yan. 2025. Adaptive Layer-skipping in Pre-trained LLMs. *CoRR* abs/2503.23798 (2025). arXiv:2503.23798 doi:10.48550/ARXIV.2503.23798

[21] Matthew V Mahoney. 2002. The PAQ1 data compression program. *Draft, Jan* 20 (2002), 20.

[22] Sourab Mangrulkar, Sylvain Gugger, Lysandre Debut, Younes Belkada, Sayak Paul, and Benjamin Bossan. 2022. PEFT: State-of-the-art Parameter-Efficient Fine-Tuning methods. https://github.com/huggingface/peft.

[23] G Nigel N Martin. 1979. Range encoding: an algorithm for removing redundancy from a digitised message. In *Proc. Institution of Electronic and Radio Engineers International Conference on Video and Data Recording*, Vol. 2.

[24] Paulius Micikevicius, Dusan Stosic, Neil Burgess, Marius Cornea, Pradeep Dubey, Richard Grisenthwaite, Sangwon Ha, Alexander Heinecke, Patrick Judd, John Kamalu, Naveen Mellempudi, Stuart F. Oberman, Mohammad Shoeybi, Michael Y. Siu, and Hao Wu. 2022. FP8 Formats for Deep Learning. *CoRR* abs/2209.05433 (2022). arXiv:2209.05433 doi:10.48550/ARXIV.2209.05433

[25] Fazal Mittu, Yihuan Bu, Akshat Gupta, Ashok Devireddy, Alp Eren Ozdarendeli, Anant Singh, and Gopala Anumanchipalli. 2024. FineZip : Pushing the Limits of Large Language Models for Practical Lossless Text Compression. *CoRR* abs/2409.17141 (2024). arXiv:2409.17141 doi:10.48550/ARXIV.2409.17141

[26] Fazal Mittu, Yihuan Bu, Akshat Gupta, Ashok Devireddy, Alp Eren Ozdarendeli, Anant Singh, and Gopala Anumanchipalli. 2024. FineZip: Pushing the Limits of Large Language Models for Practical Lossless Text Compression. arXiv:2409.17141 [cs.CL] https://arxiv.org/abs/2409.17141

[27] NVIDIA Corporation. 2025. *NVIDIA Blackwell Architecture Technical Overview*. https://resources.nvidia.com/en-us-blackwell-architecture

[28] Bo Peng, Eric Alcaide, Quentin Anthony, Alon Albalak, Samuel Arcadinho, Stella Biderman, Huanqi Cao, Xin Cheng, Michael Chung, Leon Derczynski, Xingjian Du, Matteo Grella, Kranthi Kiran GV, Xuzheng He, Haowen Hou, Przemyslaw Kazienko, Jan Kocon, Jiaming Kong, Bartlomiej Koptyra, Hayden Lau, Jiaju Lin, Krishna Sri Ipsit Mantri, Ferdinand Mom, Atsushi Saito, Guangyu Song, Xiangru Tang, Johan S. Wind, Stanislaw Wozniak, Zhenyuan Zhang, Qinghua Zhou, Jian Zhu, and Rui-Jie Zhu. 2023. RWKV: Reinventing RNNs for the Transformer Era. In *Findings of the Association for Computational Linguistics: EMNLP 2023, Singapore, December 6-10, 2023*. Association for Computational Linguistics, 14048–14077. doi:10.18653/V1/2023.FINDINGS-EMNLP.936

[29] Jorma Rissanen and Glen G. Langdon Jr. 1979. Arithmetic Coding. *IBM J. Res. Dev.* 23, 2 (1979), 149–162. doi:10.1147/RD.232.0149

[30] Jorma Rissanen and Glen G. Langdon Jr. 1981. Universal modeling and coding. *IEEE Trans. Inf. Theory* 27, 1 (1981), 12–22. doi:10.1109/TIT.1981.1056282

[31] Claude E. Shannon. 2001. A mathematical theory of communication. *ACM SIGMOBILE Mob. Comput. Commun. Rev.* 5, 1 (2001), 3–55. doi:10.1145/584091.584093

[32] Surat Teerapittayanon, Bradley McDanel, and H. T. Kung. 2016. BranchyNet: Fast inference via early exiting from deep neural networks. In *23rd International Conference on Pattern Recognition, ICPR 2016, Cancún, Mexico, December 4-8, 2016*. IEEE, 2464–2469. doi:10.1109/ICPR.2016.7900006

[33] Chandra Shekhara Kaushik Valmeekam, Krishna Narayanan, Dileep Kalathil, Jean-François Chamberland, and Srinivas Shakkottai. 2023. LLMZip: Lossless Text Compression using Large Language Models. *CoRR* abs/2306.04050 (2023). arXiv:2306.04050 doi:10.48550/ARXIV.2306.04050

[34] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*. 5998–6008. https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html

[35] Lukas Vogel, Alexander van Renen, Satoshi Imamura, Viktor Leis, Thomas Neumann, and Alfons Kemper. 2020. Mosaic: A Budget-Conscious Storage Engine for Relational Database Systems. *Proc. VLDB Endow.* 13, 11 (2020), 2662–2675.

[36] Zhongwei Wan, Xin Wang, Che Liu, Samiul Alam, Yu Zheng, Jiachen Liu, Zhongnan Qu, Shen Yan, Yi Zhu, Quanlu Zhang, Mosharaf Chowdhury, and Mi Zhang. 2024. Efficient Large Language Models: A Survey. *Transactions on Machine Learning Research* (2024). https://openreview.net/forum?id=bsCCJHbO8A Survey Certification.

[37] Jiawei Wang and Qingxin Zhang. 2024. ALCZip: Fully Exploitation of Large models in Lossless Text Compression. In *2024 4th International Conference on Electronic Information Engineering and Computer Communication (EIECC)*. 1007–1012. doi:10.1109/EIECC64539.2024.10929277

[38] Leon Windheuser, Christoph Anneser, Huanchen Zhang, Thomas Neumann, and Alfons Kemper. 2024. Adaptive Compression for Databases. In *EDBT*. OpenProceedings.org, 143–149.

[39] Mengzhou Xia, Tianyu Gao, Zhiyuan Zeng, and Danqi Chen. 2024. Sheared LLaMA: Accelerating Language Model Pre-training via Structured Pruning. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net. https://openreview.net/forum?id=09iOdaeOzp

[40] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. 2022. Orca: A Distributed Serving System for Transformer-Based Generative Models. In *OSDI*. USENIX Association, 521–538.

[41] Junxuan Zhang, Zhengxue Cheng, Yan Zhao, Shihao Wang, Dajiang Zhou, Guo Lu, and Li Song. 2025. L3TC: Leveraging RWKV for Learned Lossless Low-Complexity Text Compression. In *AAAI-25, Sponsored by the Association for the Advancement of Artificial Intelligence, February 25 - March 4, 2025, Philadelphia, PA, USA*, Toby Walsh, Julie Shah, and Zico Kolter (Eds.). AAAI Press, 13251–13259. doi:10.1609/AAAI.V39I12.33446

[42] Zixuan Zhou, Xuefei Ning, Ke Hong, Tianyu Fu, Jiaming Xu, Shiyao Li, Yuming Lou, Luning Wang, Zhihang Yuan, Xiuhong Li, Shengen Yan, Guohao Dai, Xiao-Ping Zhang, Yuhan Dong, and Yu Wang. 2024. A Survey on Efficient Inference for Large Language Models. arXiv:2404.14294 [cs.CL] https://arxiv.org/abs/2404.14294