Fast and Memory-Efficient Multi-Sequence Generation via Structured Masking

Anonymous Authors[1]

Abstract

Many applications of large language models (LLM) require drawing multiple samples from a single prompt, also known as multi-sequence generation. Current open-source approaches (e.g., HuggingFace) achieve this by replicating the prompt multiple times and treating each replication as an independent prompt within a batch. This approach is highly memory-inefficient, because the key-value (KV) cache will keep multiple copies for the repeated prompts. In this work, we present MultiGen, an alternative exact and memory-efficient strategy for multi-sequence generation that only requires storing each prompt once. To achieve exactness, we design a structured masking strategy that ensures newly sampled tokens for each generation only attend to their predecessor tokens in the same sequence. Further, we propose a novel attention computation algorithm based on intermixing matrix multiplications and diagonalized matrices that has the same theoretical runtime as the baseline approach and is generally faster in practice. Empirically, we demonstrate that MultiGen achieves consistent improvements in both generation time and memory consumption on a range of generation scenarios carefully controlled for prompt lengths, generation lengths, and number of sequence generations. Our core technique will be open-sourced and can be implemented in less than 50 lines of PyTorch.

1. Introduction

Large language models (LLMs) learn probabilistic generative models of text and are growing in popularity across a wide range of applications (Radford et al., 2019; Touvron et al., 2023). In many usecases, from complex reasoning to uncertainty quantification, there is an inherent need to sample multiple sequences for a given prompt. For example, Malinin and Gales (2020) and Lin et al. (2023) sample repeatedly from LLMs for the same prompts to estimate various notions of uncertainty. Similarly, Manakul et al. (2023) use multi-sampling to detect hallucinations in LLMs. Notably, repeated sampling is also a key step for self-consistency (Wang et al., 2022), a popular decoding strategy that improves the reasoning abilities of LLMs on complex tasks spanning arithmetic, commonsense, and symbolic reasoning. Across all these applications, it is important to efficiently sample multiple sequences for a given prompt, termed as the multi-sequence generation problem.

The standard technique for multi-sequence generation from an LLM, say k times for every prompt, is to simply repeat the prompt a desired number of times and create a pseudo-batch of (repeated) prompts. Once created, we can pass this batch to any LLM inference API, such as HuggingFace Transformers (Wolf et al., 2019). For autoregressive LLMs parameterized via decoder-only transformers (Vaswani et al., 2017; Radford et al., 2019), LLM APIs critically exploit the use of a key-value (KV) cache for accelerate inference. The key idea here is to exploit the triangular masked structure of attention matrices to avoid repeated computation of the attention between the augmented prompt tokens (initial prompt and generated tokens until current timestep) by caching their attention values in their previous time steps. While this general technique significantly accelerates inference, it comes at the cost of high memory usage to store the key and value embeddings. This seems especially wasteful where the KV cache size grows linearly with k, while incurring redundance in storing repeated copies of the key and value embeddings.

Motivated by the above limitations, we present MultiGen, an inference technique for exact, memory-efficient multi-sequence generation for large language models. The core idea is to use a single prompt and generate blocks of k tokens at every sampling step. We first introduce a structured mask wherein for any query for a specific generated sequence, we only attend to tokens from the same sequence at previous sampling steps, along with the initial prompt tokens. Additionally, we modify the positional encodings for the generated tokens to increment in blocks of k, similar to what they would have been in the original scheme

[1]Anonymous Institution, Anonymous City, Anonymous Region, Anonymous Country. Correspondence to: Anonymous Author <anon.email@domain.com>.

Preliminary work. Under review by the International Conference on Machine Learning (ICML). Do not distribute.

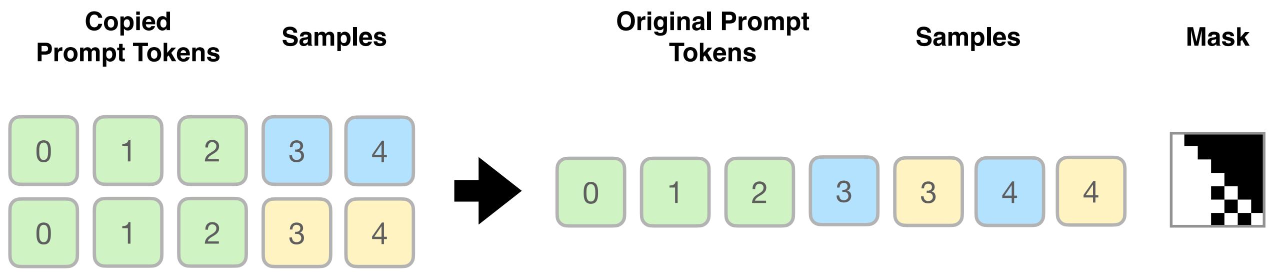

Figure 1: Illustration of MultiGen for $k = 2$ generations given a prompt of length $n = 3$ tokens (green). For visualization, we assume that the generation length is $m = 2$ tokens (blue, yellow). The number inside each box denotes the positional encodings. **Left:** Current approaches to multi-sample generation repeat the same prompts k times and treat the problem as one of batched generation. **Right:** MultiGen uses a structured mask and blocked positional encodings to isolate the k generations without replicating the prompts and thus avoids redundancy in the KV cache.

of repeating prompts. The above modifications guarantee correctness, and we illustrate this scheme in Figure 1. Next, for improving computational efficiency, we make the observation that the increase in attention computation over the baseline approach is entirely due to attention between tokens of different sequences. We decompose the attention computation into two stages: a dense attention between the new query tokens in a block and the initial prompt keys, along with sparse attention between only between the queries and keys belonging to the same sequence.

Empirically, we implement our MultiGen framework in PyTorch and benchmark against the most widely-used, open-source implmentation in HuggingFace Transformers. Since time to generation and memory usage can depend on a variety of inter-related factors and workloads, we conduct careful controlled experiments varying the initial prompt length, the generation length, and the number of sequences individually. In all scenarios, we find our approach to outperform the HuggingFace method, as well as other ablation baselines. Notably, our implementation can be easily implemented in less than 50 lines of PyTorch code and we are committed to open-sourcing our code and benchmarks.

2. Background

In this section, we review the basic architecture of a transformer, as well as key concepts related to latency and throughput in performing inference with large language models (LLM) parameterized via decoder-only transformers.

2.1. Transformer Architecture

Current autoregressive LLMs use the decoder-only Transformer (Vaswani et al., 2017; Radford et al., 2019) architecture. The core component of the Transformer is self-attention. Consider a sequence input of length n. Self-attention operates on input sequences $X \in \mathbb{R}^{n\times d}$ and is parameterized with matrices $W^Q, W^K, W^V \in \mathbb{R}^{d\times h}$. We can write self-attention as follows

$$\text{SA}(X) = \text{softmax}(A)XW^V.$$

where $A = \frac{(XW^Q)(XW^K)^\top}{\sqrt{d}}$ is an $n \times n$ attention matrix.

Thus, a Transformer forward pass will have an $\mathcal{O}(n^2)$ runtime. To preserve autoregressive dependencies, an $n \times n$ triangular mask M is applied to A such that "past" tokens cannot attend to "future" tokens. Such transformers are also referred to as causal, decoder-only, or autoregressive transformers. Finally, while attention itself is permutation-equivariant, the inputs X typically incorporate positional information through the use of positional embeddings.

2.2. LLM Inference via Prefilling and Decoding

Formally, LLM inference can be separated into two distinct phases: prefilling and decoding. Prefilling is a forward pass on the entire input sequence, in which the Key-Value (KV) cache can be processed in a concurrent manner. Prefilling is an operation on requires computing the outer product between $n \times d$ matrix Q and $n \times d$ matrix $K^\top$, resulting in an $n \times n$ self attention matrix. Once prefilling is complete, autoregressive generation begins, and it can be instantiated with a number of "decoding" strategies. Because decoding is autoregressive, it operates sequentially on a token-by-token basis. Thus, at iteration $t + 1$ of decoding, a $1 \times d$ vector Q is multiplied with an $d \times (n + t)$ matrix $K^\top$. The product is attention scores of dimension in an $1 \times (n + t)$. Crucially, the keys and values at every layer of the transformer is cached, so a significant amount of computation is saved.

Because prefilling can be parallelized, it is a compute-bound operation; meanwhile, decoding is memory bound because each iteration is not compute intensive, but the process requires storing a cache.

3. Method

Multi-sequence generation, as the name suggests, is the process of generating multiple sequences from a single source prompt. To motivate our approach, we first present the challenges with existing solutions. For simplicity, we assume that we have a single prompt and we desire multiple generations for this prompt. If we have more than one prompt, we can trivially apply MultiGen to each prompt and run it through any batched inference algorithm. For the remainder of this paper, we also use n to denote the number of prompt tokens, k to denote the number of multi-generations, and t to denote the expected length of each prompt.

3.1. Method 1: Copy Approach

The pervasive Huggingface (HF) library handles multi-sequence generation in a straightforward manner. To generate k sequences, simply copy the prompt k times and batch decode in a conventional fashion. The downside of this approach is that it is very memory efficient. Storing the same prompt repeatedly k times in the cache is wasteful, and because decoding is a memory-bound operation, it is undesirable. The number of FLOPS of this approach is on the order of $\mathcal{O}(k(n+t)^2)$, and the memory usage is $\mathcal{O}(k(n+t))$.

3.2. Method 2: Single Prompt, Dense Attention

Rather than duplicate prompts and grow the batch dimension, a natural approach is to instead keep a single copy of the prompt and store new samples in the sequence dimension. Specifically, at every iteration, compute a forward pass on k queries and use a custom attention mask to prevent newly sampled tokens from attending to each other. This approach is more memory efficient, requiring only $\mathcal{O}(n+m)$ memory, where $m = tk$. However, the downside is that growing the input sequence length severely penalizes the runtime. Because the sequence length grows by k tokens every iteration, the quadratic Transformer runtime leads to an $\mathcal{O}((n+tk)^2) = \mathcal{O}(k^2(n+t)^2)$ runtime, which is larger than the Huggingface approach by a factor of k.

3.3. Method 3: Single Prompt, Hybrid Attention

Our goal is to maintain the low memory of Method 2 with the runtime of Method 1. Consider Method 2 at the second iteration. The cache is of length $n+k$, where the first n tokens are the prompt and the next k are sampled tokens that do not attend to each other. At this iteration we have sampled k new tokens and must compute the outer product between their queries Q and $n+k$ cached keys. We make the following observation: we do not need to fully compute the $k \times k$ product between the queries and the final k columns of the cache.

This motivates the following procedure. Divide the cache keys into two portions: an n long prompt cache K_p and tk long cache K_d corresponding to the decoded tokens. First, compute the attention scores in a standard fashion with the product $A_p = QK_p^\top$. Next, we can exploit the sparse structure of K_d and obtain entries that will have non-zero attention after the sparse mask is applied. We can collect these entries into K'_d, which we can multiply element-wise $Q \odot K'_d$. We can then reshape the result and obtain attention scores A_d, and then we concatenate A_p with A_d to obtain an attention matrix A. These operations result in an equivalent operation to masked self-attention. This decomposed attention operation will be referred to as Hybrid Attention, and the full code is given in the Appendix.

4. Experiments

In this section, we aim to show that the theoretical savings demonstrated above manifest as empirical gains in throughput and memory efficiency. With constraints on an academic budget, all experiments are conducted on a single NVIDIA 24GB A5000 GPU connected to a Colfax CX41060s-EK9 4U Rackmount Server with AMD EPYC (Genoa) 9124 processors. We use a lightweight sharded 1.3 Billion parameter version of Llama2 with a context length of 4096, loaded through Huggingface. In principle, larger models loaded onto larger GPUs will yield analogous results.

4.1. Speed

For fair evaluation, we benchmark our method on a range of different inputs. We consider three parameters as part of the input: number of generated sequences, maximum number of new tokens, and prompt length. In terms of our previous runtime analysis, these correspond to k, m, and n respectively. In order to thoroughly understand how these variables affect performance, we must conduct experiments exploring how different combinations of the variables affect generation time. Because the parameter space is exponentially large and LLM computation is prohibitively GPU intensive, we opt to test a subset in which we hold two constant and vary one. We fix and vary k, m, n in the following manner

1. Fix: $k = 50, m = 50$, Vary: $n = (1, 20, 40, ..., 200)$
2. Fix: $k = 50, n = 50$, Vary: $m = (1, 10, ..., 80)$
3. Fix: $k = 50, n = 50$, Vary: $k = (1, 10, ..., 80)$

Note that to avoid cherry-picked constants, we fix them

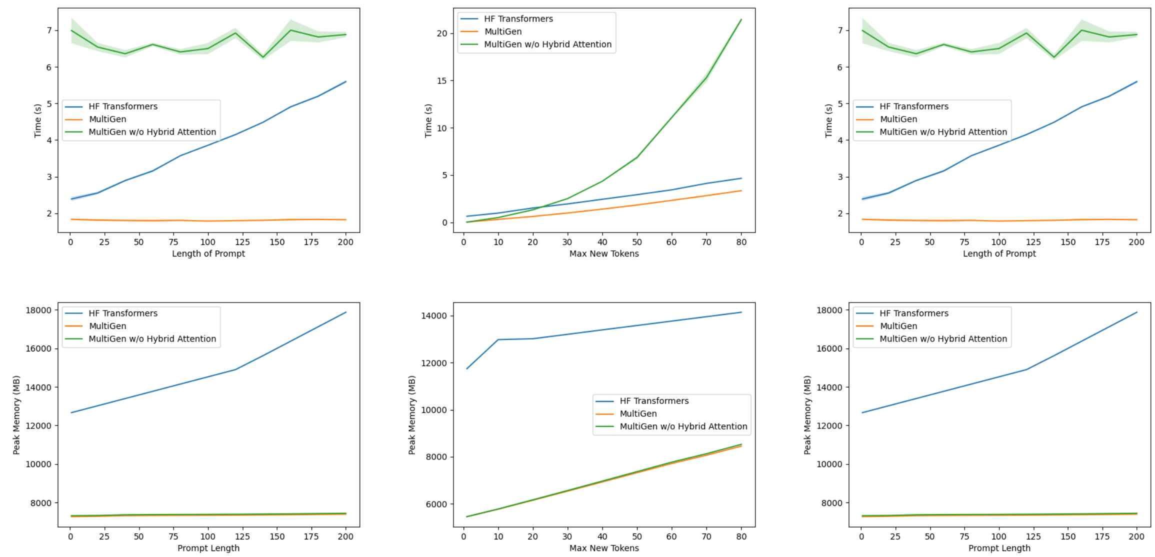

Figure 2: **Top:** We show the time usage in three different parameter regimes. In each, MultiGen outperforms the HF multi-sequence sampling approach. **Bottom:** For the same settings, we also show peak memory usage. In these plots, we see that MultiGen obtains the memory savings of Method 2 (no Hybrid Attention). Thus, it achieves significant memory improvement with no sacrifice in time.

all to the same value of 50. We then vary the free input parameter so as to maximize GPU utilization such that we can empirical the asymptotic behavior of our method. We run each method on random input tokens and average over 5 runs, clearing the cache to avoid cross contamination.

Plotting generation time over these values, we can make the following observations in Figure 2. First, we observe a superior runtime for MultiGen compared to Huggingface baseline. This is an encouraging result because in terms of FLOPs, our method theoretically only purports to be equal in speed to the baseline. In practice, because our method operates with a much smaller batch size, it is able to run faster because larger batches on GPUs often cause issues relating to synchronization overhead and memory bandwidth (Yuan et al., 2024). As predicted by our analysis, the naive approach to multi-sequence generation is penalized in runtime by growing the sequence length faster than the other two approached. While it is inefficient in time, it is dramatically more efficient in memory.

4.2. Memory

Another explanation for the improvement in runtime for our method compared to Huggingface is due to the nature of decoding. Decoding is known to be a memory-bound operation, so improvements in memory will affect the overall runtime than compute efficiency. As we can see from Figures 2, MultiGen dramatically improves the memory efficiency of decoding. We make these measurements in the same manner as before, varying a parameter and holding two constants. Again, we average over 5 runs over random input and cleared memory cache for each run. We confirm our analysis of memory that that the naive approach to multi-sequence generation by a factor k. This is confirmed by the plot that shows as we vary the number of sequences sampled, i.e. k, the maximum memory usage of methods diverges. Thus, we confirm that in our experiments, MultiGen enjoys a dramatic reduction in memory usage without any increase in time, a decrease in fact. In these experiments, we show that a free lunch is possible, in which we show better memory efficiency *and* compute time.

5. Conclusion

In conclusion, MultiGen presents a significant advancement in the field of multi-sequence generation with large language models. By addressing the memory inefficiencies of existing approaches, MultiGen offers a more efficient and scalable solution. Through a combination of structured masking and a novel attention computation algorithm, MultiGen maintains exactness while reducing both generation time and memory consumption. The empirical results across various generation scenarios further validate the effectiveness of MultiGen. With its open-source implementation and minimal code requirements, MultiGen is a straightforward-to-implement solution for applications that rely on multi-sequence generation.

References

Z. Lin, S. Trivedi, and J. Sun. Generating with confidence: Uncertainty quantification for black-box large language models. *arXiv preprint arXiv:2305.19187*, 2023.

A. Malinin and M. Gales. Uncertainty estimation in autoregressive structured prediction. *arXiv preprint arXiv:2002.07650*, 2020.

P. Manakul, A. Liusie, and M. J. Gales. Selfcheckgpt: Zero-resource black-box hallucination detection for generative large language models. *arXiv preprint arXiv:2303.08896*, 2023.

A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

X. Wang, J. Wei, D. Schuurmans, Q. Le, E. Chi, S. Narang, A. Chowdhery, and D. Zhou. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*, 2022.

T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, et al. Huggingface's transformers: State-of-the-art natural language processing. *arXiv preprint arXiv:1910.03771*, 2019.

Z. Yuan, Y. Shang, Y. Zhou, Z. Dong, Z. Zhou, C. Xue, B. Wu, Z. Li, Q. Gu, Y. J. Lee, Y. Yan, B. Chen, G. Sun, and K. Keutzer. Llm inference unveiled: Survey and roofline model insights, 2024.