Prepacking: A Simple Method for Fast Prefilling and Increased Throughput in Large Language Models

Anonymous Authors[1]

Abstract

During inference for transformer-based LLMs, prefilling computes the key-value (KV) cache for prompt input tokens before autoregressive generation. This work highlights a pitfall of prefilling: for batches containing high-varying prompt lengths, significant computation is wasted by the standard practice of padding sequences to the maximum length. As LLMs support longer context lengths, variations in prompt lengths within a batch become more pronounced. To address this, we propose *prepacking*, a simple yet effective method to optimize prefilling computation. Prepacking combines prompts of varying lengths into a sequence and packs multiple sequences into a compact batch using a bin-packing algorithm, then modifies the attention mask and positional encoding to compute multiple prefilled KV-caches within a single sequence. On standard datasets with varying prompt lengths, our method significantly improves speed and memory efficiency compared to default padding-based prefilling in Huggingface across various model configurations and inference scenarios.

1. Introduction

Transformer-based large language models (LLMs) have emerged as a powerful general purpose tool to service natural language queries (Bai et al., 2022; Touvron et al., 2023; Achiam et al., 2023). As language models continue to grow in scale and their usage proliferates across various domains (Eloundou et al., 2023), the capability to generate tokens with optimal speed and efficiency becomes increasingly paramount.

The conventional approach to LLM inference with varied size inputs is inefficient, and it is exemplified by the Huggingface Transformers library (Wolf et al., 2020). The Huggingface library has seen widespread adoption in the NLP community. Despite its wide use, Huggingface handles prompts of varying lengths by padding all prompts to match the length of the longest sequence and processing the batch through a Transformer model in its entirety. This results in substantial memory utilization and computational inefficiency. While LLMs are compute-bound during prefilling, they are also memory-bound during generation (Kwon et al., 2023), so it is crucial to optimize memory and GPU utilization to enable efficient inference and scalability. As LLMs grow to support longer context lengths (Reid et al., 2024), handling variation in prompt length becomes increasingly important.

In this work, we mitigate wasteful computation with an alternative pre-processing step called *prepacking*. Prepacking is specifically aimed at improving the speed and memory usage of LLM prefilling, which is the initial computation that populates the Key-Value cache (KV cache) preceding generation. Prepacking is conceptually simple; rather than padding every sequence to the same length, we pack multiple prompts together in place of padding tokens using an off-the-shelf bin-packing algorithm. This is made possible by custom attention masking and positional encoding that enable the computation of a batch within a single sequence. The positional encoding restarts its index for each prompt in the sequence and the mask prevents prompts from attending to previous prompts in the packed sequence (Figure 1). A forward pass on the pre-packed batch will populate a KV cache, which we can unpack to get the cache for the original prompts for next token generations.

As compared to optimized LLM serving platforms which write CUDA kernels to mitigate paddings such as vLLM (Kwon et al., 2023), prepacking is entirely implemented in PyTorch. We empirically demonstrate that prepacking leads to a **speedup of up to 6x** in prefilling and time-to-first-token (TTFT) compared to the full batching method used in Huggingface tested on NVIDIA A6000 GPUs. To evaluate prepacking's runtime performance under conditions representative of real-world user traffic, we tested it across six diverse language datasets with language models

[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.

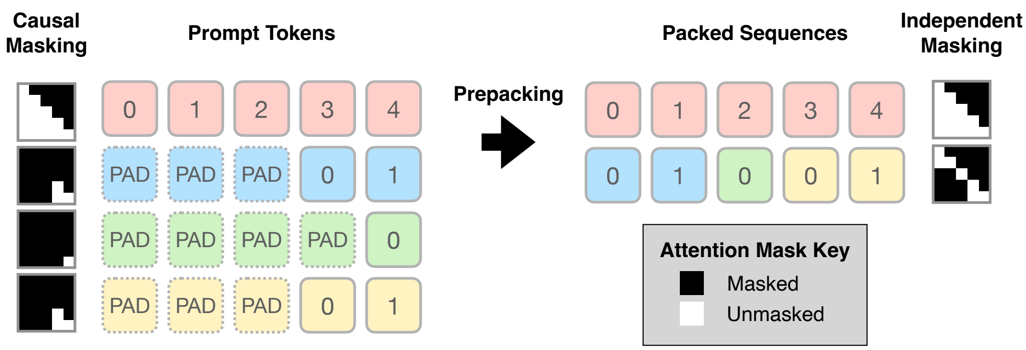

Figure 1: **Left:** The standard full batching approach (e.g., used in HuggingFace) pads shorter prompts to maximum prompt length in the batch. Each prompt has its own causal attention mask. **Right:** Prepacking combines multiple prompts into a single sequence using a bin-packing algorithm, and applies *independent masking* and *restart positional encodings* (numbers inside token boxes) to avoid prompts attending to other prompts. Both strategies are equivalent at decoding time, but prepacking is more compute efficient during prefilling.

ranging from 1B to 13B parameters. Prepacking achieves greater speedup when the sequences within a batch exhibit more diverse length variations and when the batch size is large. Additionally, we demonstrate that prepacking is a simple method for increasing LLM throughput, especially in memory-constrained settings. Specifically, prepacking significantly reduces memory consumption by allowing up to **16x larger batch size** during prefilling.

2. Preliminaries

2.1. Transformer Architecture

The decoder-only Transformer (Vaswani et al., 2017; Radford et al., 2019) is ubiquitous in its use as the deep learning architecture for autoregressive LLMs. The core component of the Transformer is self-attention. 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 where n is the length of the input. To preserve autoregressive dependencies, an $n \times n$ mask M is applied to A such that "past" tokens cannot attend to "future" tokens. Finally, while attention itself is permutation-equivariant, the inputs X typically incorporate positional information through the use of positional embeddings.

2.2. Performance Metrics

Key metrics for evaluating LLM serving (Miao et al., 2023) include latency measures such as Time-to-First-Token (TTFT), the time required for prefilling the KV cache and generating the first token, and Time-per-Output-Token (TPOT), the average time to generate each subsequent token. Throughput measures the number of requests processed per unit time. In this work, we focus on optimizing the prefilling stage by evaluating prefilling time and TTFT metrics.

3. Prepacking

Padding input prompts to the maximum length causes significant computation waste on pad tokens. We propose a simple solution: insert more short prompts where padding was previously located. Because this method "packs" prompts together to speed up prefilling, we refer to this method as prepacking. In formal terms, we have a set of k prompts $p_1, \cdots, p_k$ of lengths $l_1, \cdots, l_k$, and our goal is to create a tensorized batch $B = (p'_1, ..., p'_r)$, where $p'_1, ..., p'_r$ are sequences that contain the original prompts such that $r \leq k$. The full algorithm is shown in Algorithm 1.

3.1. Bin Packing

The problem of packing prompts together can be cast as a bin packing problem, where a bin can contain tokens from several different sequences. The goal of prepacking is to efficiently concatenate prompts together such that original prompts with lengths $l_1, ..., l_k$ are placed into the smallest possible r bins, each of a fixed sized. It is guaranteed that $r \leq k$. We shall select m, where m is the maximum prompt length as previously defined, to be the fixed size of the bins. For sequences that do not completely reach size m after bin-packing, they will be padded to reach m. Note that we choose the smallest possible constant for our bin size because the bin size will incur quadratic running time. In general, bin packing is an NP-hard problem (Garey and

Johnson, 1979), but many heuristic approaches exist obtain approximate solutions (Buljubašić and Vasquez, 2016). We use a First-Fit Decreasing bin packing heuristic as implemented by (Maier, 2021).

3.2. Prompt-wise Independent Masking and Restart Positional Encoding

Prepacking will concatenate multiple smaller prompts under a single bin. Simply using the KV-cache of this packed sequence will be incorrect, because every prompt within the bin will attend causally to previous prompts. As a remedy, we create a custom attention mask to prevent items from attending to each other. We refer to this masking strategy as *independent masking*. We describe our masking strategy below and illustrate it in Figure 1.

Formally, consider a causal (lower triangular) attention mask M , where entry $M_{i,j} = 1$ signifies that token t_i can attend to t_j and $i \geq j$. An independent mask M' is a mask such that for all indices a, b that mark the start and end of a prompt, $M'_{a:b,a:b} = L_{b-a}$, where L_n is an $n \times n$ lower triangular matrix. All other entries will be 0. Creating the attention mask and extracting the resultant KV-cache requires a certain amount of bookkeeping for tracking lengths of sequences and indices, but these operations contribute an insignificant (linear) overhead compared to the Transformer forward pass.

Lastly, we need to modify the positional encodings for the packed sequences. In general, the Transformer architecture is permutation equivariant (Naseer et al., 2021), so the purpose of positional encodings (PE) is to give the model information about the position of a token in a sequence. Thus, in a prepacked sequence, we must edit the PEs for the tokens such that it is the same as it was in the unpacked prompts. This leads to positions that "restart" in the packed sequence at the beginning of any new prompt, hence the name *restart positional encoding*. With packed batches, independent masks, and restart PEs, we can compute and prefill the KV cache for each prompt and use it for autoregressive generation using any decoding algorithm.

3.3. Runtime Analysis

With prepacking, we are guaranteed to compute the *exact* KV caches as a padded, full-batching method. Next, we analyze the gains during the prefilling stage using our approach. Let the sum of prompt lengths over the batch be denoted by $L = \sum_i l_i$. In the best case scenario, our bin packing algorithm is able to pack every prompt into bins with no additional padding. Then we can express the number of bins as $r = L/m$. We can now find the runtime of prefilling a batch with prepacking and compare it to the naive method.

$$\mathcal{O}(rm^2) = \mathcal{O}(Lm) = \mathcal{O}(km(L/k)) \leq \mathcal{O}(km^2) \quad (1)$$

The final inequality holds because the average length must be less than or equal to the maximum length. Also note that the prepacking algorithm itself runs in $\mathcal{O}(k \log k)$ time which is insignificant toward the overall runtime. Thus, we find that prepacking will outperform the naive padding approach in the best case scenario. In the worst case scenario, we cannot reduce the number of bins from the original batch size and $r = k$ will lead to the same runtime. We shall show in our experiments that datasets tend to have enough length variation such that $r < k$ is a comfortable assumption in practice, and the differences between the naive method and prepacking can be stark. Figure 5 illustrates an actual packing done by prepacking which greatly reduces paddings.

4. Experiments

We empirically show the significant throughput improvements and GPU memory savings achieved by prepacking across real-world datasets with diverse length distributions. Our comprehensive evaluation spans language models of varying architectures and scales, ranging from 1.3B to 13B parameters.

4.1. Datasets, models and baselines

We evaluate prepacking's runtime performance on 5 real-world diverse language datasets with a arange of LLMs of varying sizes. We compare prepacking with two baselines: 1) ***Full Batching***: As implemented by Huggingface (Wolf et al., 2020), this method pads shorter prompts to match the longest prompt in a batch, using attention masks to ignore padding. 2) ***Length-Ordered Batching***: This baseline assumes access to all user requests. It sorts inputs by length and samples batches to minimize padding. This is impractical for real-world scenarios with unpredictable request orders. The details of datasets, models and baselines are in Appendix E.

4.2. Prefilling Time and TTFT

We compare the prefilling time and Time-to-First-Token (TTFT) between prepacking and Full Batching across datasets and models in Figure 2. TTFT measures the total time required for prefilling the KV cache and generating the first token. For our method, TTFT additionally includes an overhead which is the unpacking phase, where we unpack the prompts to their original order for generation. This unpacking phase has a linear time complexity in the number of prompts, which is dominated by the quadratic computational complexity of prefilling. Prepacking consistently outperforms Full Batching with less prefilling time and TTFT, enhancing speed ranging from 1.6x to 3.5x. Moreover, Prepacking has lower inference time standard deviations, attributed to reduced padding overhead, enabling more reli-

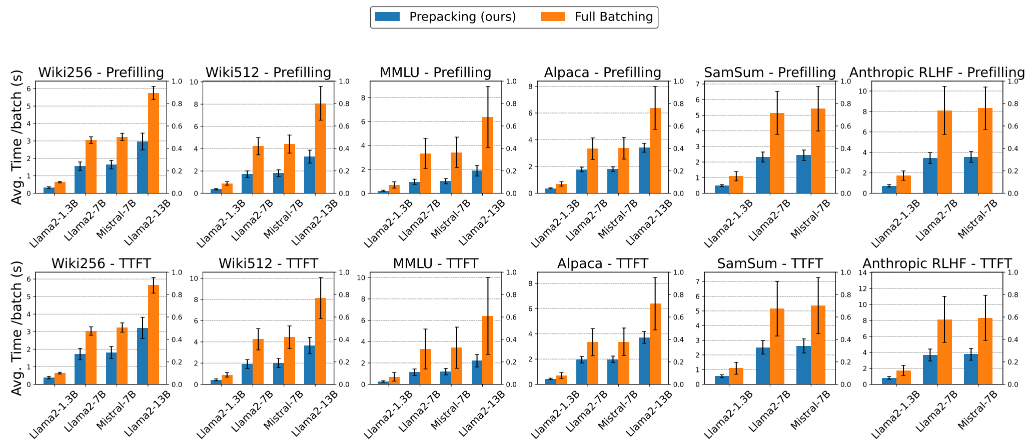

Figure 2: Average inference time per batch for various language models using prepacking and Full Batching, with a batch size of 16. The comparison is conducted across multiple datasets with two metrics, Prefilling Time and TTFT. Error bars represent the standard deviation across batches and seeds. The results show that prepacking consistently leads to reduced inference times compared to Full Batching and exhibits reduced variability, as evidenced by smaller standard deviation errors, indicating more reliable and predictable inference times when adopting prepacking.

able and predictable performance suitable for applications demanding consistent LLM serving.

4.3. GPU Memory Saving and Utilization

We evaluate Prepacking's GPU memory efficiency, stemming from reduced computation on padded tokens, against other baselines in Figure 6. Prepacking consistently shows lower peak memory usage which allows it to process larger batch size without out-of-memory errors. For example, with the Llama2-1.3B model on the MMLU dataset, prepacking can handle batch sizes up to 16 times larger during prefilling compared to Full Batching. More discussion of the results are in Appendix F.

4.4. Enhanced Speedup with Increasing Batch Sizes

In reality, the distribution of batch sizes encountered during language model inference can fluctuate due to non-uniform user requests arrival patterns. To evaluate our method's effectiveness in handling this variability, we conducted experiments across a range of batch sizes for the Llama2-7B and Llama2-1.3B models. The results shown in Figure 3 show substantial speedup gains achieved by our approach over Full Batching. Larger batch sizes exhibit greater performance improvements with our method, up to **4.5x** and **6x** speedup for the 7B and 1.3B Llama2 models, respectively. This trend stems from the increased likelihood of diverse prompt lengths within larger batches, which leads to more padding overhead for Full Batching. In contrast, our method efficiently handles variable-length prompts via bin-packing, mitigating this overhead.

Additionally, in Appendix K, we also show that the speedup can be predicted with characteristics of the lengths within a batch.

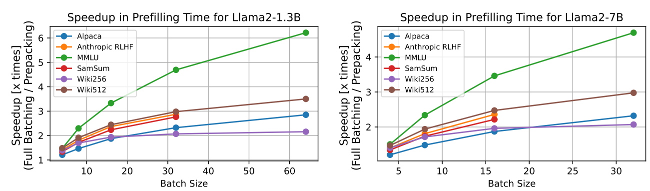

Figure 3: Speed up across various batch sizes. Speed up is calculated as the ratio of the prefilling time with full batching to that of prepacking. Missing data points are due to out-of-memory issues.

5. Conclusion

We proposed prepacking, a simple and effective approach to optimize the prefilling computation for LLMs during inference. Our evaluation on typical datasets with varying prompt lengths demonstrates significant speedups compared to standard prefilling computation in Huggingface's implementation. As language models continue to scale and support longer context lengths, addressing the inefficiencies associated with prefilling computation becomes crucial for optimizing inference speed and computational resource allocation. Prepacking provides a promising solution to this challenge, enabling more efficient inference for prompts with varying lengths. In the future, it would be interesting to explore more complex decoding strategies post-prefilling that also incorporate bin packing for further increase in throughput.

References

J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

A. Agrawal, N. Kedia, A. Panwar, J. Mohan, N. Kwatra, B. S. Gulavani, A. Tumanov, and R. Ramjee. Taming throughput-latency tradeoff in llm inference with sarathi-serve. *arXiv preprint arXiv:2403.02310*, 2024.

R. Y. Aminabadi, S. Rajbhandari, A. A. Awan, C. Li, D. Li, E. Zheng, O. Ruwase, S. Smith, M. Zhang, J. Rasley, et al. Deepspeed-inference: enabling efficient inference of transformer models at unprecedented scale. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15. IEEE, 2022.

Y. Bai, A. Jones, K. Ndousse, A. Askell, A. Chen, N. DasSarma, D. Drain, S. Fort, D. Ganguli, T. Henighan, et al. Training a helpful and harmless assistant with reinforcement learning from human feedback. *arXiv preprint arXiv:2204.05862*, 2022.

M. Buljubašić and M. Vasquez. Consistent neighborhood search for one-dimensional bin packing and two-dimensional vector packing. *Computers & Operations Research*, 76:12–21, 2016.

T. Cai, Y. Li, Z. Geng, H. Peng, J. D. Lee, D. Chen, and T. Dao. Medusa: Simple llm inference acceleration framework with multiple decoding heads, 2024.

T. Eloundou, S. Manning, P. Mishkin, and D. Rock. Gpts are gpts: An early look at the labor market impact potential of large language models. *arXiv preprint arXiv:2303.10130*, 2023.

P. Fegade, T. Chen, P. Gibbons, and T. Mowry. The cora tensor compiler: Compilation for ragged tensors with minimal padding. *Proceedings of Machine Learning and Systems*, 4:721–747, 2022.

M. Garey and D. Johnson. Computers and intractability a guide to the theory of np-completeness new york freeman and co. 1979.

B. Gliwa, I. Mochol, M. Biesek, and A. Wawer. SAMSum corpus: A human-annotated dialogue dataset for abstractive summarization. In *Proceedings of the 2nd Workshop on New Frontiers in Summarization*, pages 70–79, Hong Kong, China, Nov. 2019. Association for Computational Linguistics. doi: 10.18653/v1/D19-5409. URL https://www.aclweb.org/anthology/D19-5409.

D. Hendrycks, C. Burns, S. Basart, A. Critch, J. Li, D. Song, and J. Steinhardt. Aligning ai with shared human values. *Proceedings of the International Conference on Learning Representations (ICLR)*, 2021a.

D. Hendrycks, C. Burns, S. Basart, A. Zou, M. Mazeika, D. Song, and J. Steinhardt. Measuring massive multitask language understanding. *Proceedings of the International Conference on Learning Representations (ICLR)*, 2021b.

A. Q. Jiang, A. Sablayrolles, A. Mensch, C. Bamford, D. S. Chaplot, D. d. l. Casas, F. Bressand, G. Lengyel, G. Lample, L. Saulnier, et al. Mistral 7b. *arXiv preprint arXiv:2310.06825*, 2023.

W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. E. Gonzalez, H. Zhang, and I. Stoica. Efficient memory management for large language model serving with pagedattention, 2023.

Y. Leviathan, M. Kalman, and Y. Matias. Fast inference from transformers via speculative decoding. In *International Conference on Machine Learning*, pages 19274–19286. PMLR, 2023.

Z. Liu, J. Wang, T. Dao, T. Zhou, B. Yuan, Z. Song, A. Shrivastava, C. Zhang, Y. Tian, C. Re, et al. Deja vu: Contextual sparsity for efficient llms at inference time. In *International Conference on Machine Learning*, pages 22137–22176. PMLR, 2023.

B. Maier. GitHub - benmaier/binpacking: Distribution of weighted items to bins (either a fixed number of bins or a fixed number of volume per bin). Data may be in form of list, dictionary, list of tuples or csv-file. — github.com. https://github.com/benmaier/binpacking, 2021. [Accessed 30-03-2024].

S. Merity, C. Xiong, J. Bradbury, and R. Socher. Pointer sentinel mixture models, 2016.

X. Miao, G. Oliaro, Z. Zhang, X. Cheng, H. Jin, T. Chen, and Z. Jia. Towards efficient generative large language model serving: A survey from algorithms to systems. *arXiv preprint arXiv:2312.15234*, 2023.

M. M. Naseer, K. Ranasinghe, S. H. Khan, M. Hayat, F. Shahbaz Khan, and M.-H. Yang. Intriguing properties of vision transformers. *Advances in Neural Information Processing Systems*, 34:23296–23308, 2021.

NVIDIA. GitHub - NVIDIA/FasterTransformer: Transformer related optimization, including BERT, GPT — github.com. https://github.com/NVIDIA/FasterTransformer, 2021. [Accessed 29-03-2024].

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.

M. Reid, N. Savinov, D. Teplyashin, D. Lepikhin, T. Lillicrap, J. baptiste Alayrac, R. Soricut, A. Lazaridou, O. Firat, J. Schrittwieser, I. Antonoglou, R. Anil, S. Borgeaud, A. Dai, K. Millican, E. Dyer, M. Glaese, T. Sottiaux, B. Lee, F. Viola, M. Reynolds, Y. Xu, J. Molloy, J. Chen, M. Isard, P. Barham, T. Hennigan, R. McIlroy, M. Johnson, J. Schalkwyk, E. Collins, E. Rutherford, E. Moreira, K. Ayoub, M. Goel, C. Meyer, G. Thornton, Z. Yang, H. Michalewski, Z. Abbas, N. Schucher, A. Anand, R. Ives, J. Keeling, K. Lenc, S. Haykal, S. Shakeri, P. Shyam, A. Chowdhery, R. Ring, S. Spencer, E. Sezener, L. Vilnis, O. Chang, N. Morioka, G. Tucker, C. Zheng, O. Woodman, N. Attaluri, T. Kocisky, E. Eltyshev, X. Chen, T. Chung, V. Selo, S. Brahma, P. Georgiev, A. Slone, Z. Zhu, J. Lottes, S. Qiao, B. Caine, S. Riedel, A. Tomala, M. Chadwick, J. Love, P. Choy, S. Mittal, N. Houlsby, Y. Tang, M. Lamm, L. Bai, Q. Zhang, L. He, Y. Cheng, P. Humphreys, Y. Li, S. Brin, A. Cassirer, Y. Miao, L. Zilka, T. Tobin, K. Xu, L. Proleev, D. Sohn, A. Magni, L. A. Hendricks, I. Gao, S. Ontañón, O. Bunyan, N. Byrd, A. Sharma, B. Zhang, M. Pinto, R. Sinha, H. Mehta, D. Jia, S. Caelles, A. Webson, A. Morris, B. Roelofs, Y. Ding, R. Strudel, X. Xiong, M. Ritter, M. Dehghani, R. Chaabouni, A. Karmarkar, G. Lai, F. Mentzer, B. Xu, Y. Li, Y. Zhang, T. L. Paine, A. Goldin, B. Neyshabur, K. Baumli, A. Levskaya, M. Laskin, W. Jia, J. W. Rae, K. Xiao, A. He, S. Giordano, L. Yagati, J.-B. Lespiau, P. Natsev, S. Ganapathy, F. Liu, D. Martins, N. Chen, Y. Xu, M. Barnes, R. May, A. Vezer, J. Oh, K. Franko, S. Bridgers, R. Zhao, B. Wu, B. Mustafa, S. Sechrist, E. Parisotto, T. S. Pillai, C. Larkin, C. Gu, C. Sorokin, M. Krikun, A. Guseynov, J. Landon, R. Datta, A. Pritzel, P. Thacker, F. Yang, K. Hui, A. Hauth, C.-K. Yeh, D. Barker, J. Mao-Jones, S. Austin, H. Sheahan, P. Schuh, J. Svensson, R. Jain, V. Ramasesh, A. Briukhov, D.-W. Chung, T. von Glehn, C. Butterfield, P. Jhakra, M. Wiethoff, J. Frye, J. Grimstad, B. Changpinyo, C. L. Lan, A. Bortsova, Y. Wu, P. Voigtlaender, T. Sainath, C. Smith, W. Hawkins, K. Cao, J. Besley, S. Srinivasan, M. Omernick, C. Gaffney, G. Surita, R. Burnell, B. Damoc, J. Ahn, A. Brock, M. Pajarskas, A. Petrushkina, S. Noury, L. Blanco, K. Swersky, A. Ahuja, T. Avrahami, V. Misra, R. de Liedekerke, M. Iinuma, A. Polozov, S. York, G. van den Driessche, P. Michel, J. Chiu, R. Blevins, Z. Gleicher, A. Recasens, A. Rrustemi, E. Gribovskaya, A. Roy, W. Gworek, S. Arnold, L. Lee, J. Lee-Thorp, M. Maggioni, E. Piqueras, K. Badola, S. Vikram, L. Gonzalez, A. Baddepudi, E. Senter, J. Devlin, J. Qin, M. Azzam, M. Trebacz, M. Polacek, K. Krishnakumar, S. yiin Chang, M. Tung, I. Penchev, R. Joshi, K. Olszewska, C. Muir, M. Wirth, A. J. Hartman, J. Newlan, S. Kashem, V. Bolina, E. Dabir, J. van Amersfoort, Z. Ahmed, J. Cobon-Kerr, A. Kamath, A. M. Hrafnkelsson, L. Hou, I. Mackinnon, A. Frechette, E. Noland, X. Si, E. Taropa, D. Li, P. Crone, A. Gulati, S. Cevey, J. Adler, A. Ma, D. Silver, S. Tokumine, R. Powell, S. Lee, M. Chang, S. Hassan, D. Mincu, A. Yang, N. Levine, J. Brennan, M. Wang, S. Hodkinson, J. Zhao, J. Lipschultz, A. Pope, M. B. Chang, C. Li, L. E. Shafey, M. Paganini, S. Douglas, B. Bohnet, F. Pardo, S. Odoom, M. Rosca, C. N. dos Santos, K. Soparkar, A. Guez, T. Hudson, S. Hansen, C. Asawaroengchai, R. Addanki, T. Yu, W. Stokowiec, M. Khan, J. Gilmer, J. Lee, C. G. Bostock, K. Rong, J. Caton, P. Pejman, F. Pavetic, G. Brown, V. Sharma, M. Lučić, R. Samuel, J. Djolonga, A. Mandhane, L. L. Sjösund, E. Buchatskaya, E. White, N. Clay, J. Jiang, H. Lim, R. Hemsley, J. Labanowski, N. D. Cao, D. Steiner, S. H. Hashemi, J. Austin, A. Gergely, T. Blyth, J. Stanton, K. Shivakumar, A. Siddhant, A. Andreassen, C. Araya, N. Sethi, R. Shivanna, S. Hand, A. Bapna, A. Khodaei, A. Miech, G. Tanzer, A. Swing, S. Thakoor, Z. Pan, Z. Nado, S. Winkler, D. Yu, M. Saleh, L. Maggiore, I. Barr, M. Giang, T. Kagohara, I. Danihelka, A. Marathe, V. Feinberg, M. Elhawaty, N. Ghelani, D. Horgan, H. Miller, L. Walker, R. Tanburn, M. Tariq, D. Shrivastava, F. Xia, C.-C. Chiu, Z. Ashwood, K. Baatarsukh, S. Samangooei, F. Alcober, A. Stjerngren, P. Komarek, K. Tsihlas, A. Boral, R. Comanescu, J. Chen, R. Liu, D. Bloxwich, C. Chen, Y. Sun, F. Feng, M. Mauger, X. Dotiwalla, V. Hellendoorn, M. Sharman, I. Zheng, K. Haridasan, G. Barth-Maron, C. Swanson, D. Rogozińska, A. Andreev, P. K. Rubenstein, R. Sang, D. Hurt, G. Elsayed, R. Wang, D. Lacey, A. Ilić, Y. Zhao, L. Aroyo, C. Iwuanyanwu, V. Nikolaev, B. Lakshminarayanan, S. Jazayeri, R. L. Kaufman, M. Varadarajan, C. Tekur, D. Fritz, M. Khalman, D. Reitter, K. Dasgupta, S. Sarcar, T. Ornduff, J. Snaider, F. Huot, J. Jia, R. Kemp, N. Trdin, A. Vijayakumar, L. Kim, C. Angermueller, L. Lao, T. Liu, H. Zhang, D. Engel, S. Greene, A. White, J. Austin, L. Taylor, S. Ashraf, D. Liu, M. Georgaki, I. Cai, Y. Kulizhskaya, S. Goenka, B. Saeta, K. Vodrahalli, C. Frank, D. de Cesare, B. Robenek, H. Richardson, M. Alnahlawi, C. Yew, P. Ponnapalli, M. Tagliasacchi, A. Korchemniy, Y. Kim, D. Li, B. Rosgen, Z. Ashwood, K. Levin, J. Wiesner, P. Banzal, P. Srinivasan, H. Yu, Çağlar Ünlü, D. Reid, Z. Tung, D. Finchelstein, R. Kumar, A. Elisseeff, J. Huang, M. Zhang, R. Zhu, R. Aguilar, M. Giménez, J. Xia, O. Dousse, W. Gierke, S. H. Yeganeh, D. Yates, K. Jalan, L. Li, E. Latorre-Chimoto, D. D. Nguyen, K. Durden, P. Kallakuri, Y. Liu, M. Johnson, T. Tsai, A. Talbert, J. Liu, A. Neitz, C. Elkind, M. Selvi, M. Jasarevic, L. B. Soares, A. Cui, P. Wang, A. W. Wang, X. Ye, K. Kallarackal, L. Loher, H. Lam, J. Broder, D. Holtmann-Rice, N. Martin, B. Ramadhana, D. Toyama, M. Shukla, S. Basu, A. Mohan, N. Fernando, N. Fiedel, K. Paterson, H. Li, A. Garg, J. Park, D. Choi, D. Wu, S. Singh, Z. Zhang, A. Globerson, L. Yu, J. Carpenter, F. de Chaumont Quitry, C. Radebaugh, C.-C. Lin, A. Tudor, P. Shroff, D. Garmon, D. Du, N. Vats, H. Lu, S. Iqbal, A. Yakubovich, N. Tripuraneni, J. Manyika, H. Qureshi, N. Hua, C. Ngani, M. A. Raad, H. Forbes, A. Bulanova, J. Stanway, M. Sundararajan, V. Ungureanu, C. Bishop, Y. Li, B. Venkatraman, B. Li, C. Thornton, S. Scellato, N. Gupta, Y. Wang, I. Tenney, X. Wu, A. Shenoy, G. Carvajal, D. G. Wright, B. Bariach, Z. Xiao, P. Hawkins, S. Dalmia, C. Farabet, P. Valenzuela, Q. Yuan, C. Welty, A. Agarwal, M. Chen, W. Kim, B. Hulse, N. Dukkipati, A. Paszke, A. Bolt, E. Davoodi, K. Choo, J. Beattie, J. Prendki, H. Vashisht, R. Santamaria-Fernandez, L. C. Cobo, J. Wilkiewicz, D. Madras, A. Elqursh, G. Uy, K. Ramirez, M. Harvey, T. Liechty, H. Zen, J. Seibert, C. H. Hu, M. Elhawaty, A. Khorlin, M. Le, A. Aharoni, M. Li, L. Wang, S. Kumar, A. Lince, N. Casagrande, J. Hoover, D. E. Badawy, D. Soergel, D. Vnukov, M. Miecnikowski, J. Simsa, A. Koop, P. Kumar, T. Sellam, D. Vlasic, S. Daruki, N. Shabat, J. Zhang, G. Su, J. Zhang, J. Liu, Y. Sun, E. Palmer, A. Ghaffarkhah, X. Xiong, V. Cotruta, M. Fink, L. Dixon, A. Sreevatsa, A. Goedeckemeyer, A. Dimitriev, M. Jafari, R. Crocker, N. FitzGerald, A. Kumar, S. Ghemawat, I. Philips, F. Liu, Y. Liang, R. Sterneck, A. Repina, M. Wu, L. Knight, M. Georgiev, H. Lee, H. Askham, A. Chakladar, A. Louis, C. Crous, H. Cate, D. Petrova, M. Quinn, D. Owusu-Afriyie, A. Singhal, N. Wei, S. Kim, D. Vincent, M. Nasr, C. A. Choquette-Choo, R. Tojo, S. Lu, D. de Las Casas, Y. Cheng, T. Bolukbasi, K. Lee, S. Fatehi, R. Ananthanarayanan, M. Patel, C. Kaed, J. Li, J. Sygnowski, S. R. Belle, Z. Chen, J. Konzelmann, S. Põder, R. Garg, V. Koverkathu, A. Brown, C. Dyer, R. Liu, A. Nova, J. Xu, S. Petrov, D. Hassabis, K. Kavukcuoglu, J. Dean, and O. Vinyals. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context, 2024.

Y. Sheng, L. Zheng, B. Yuan, Z. Li, M. Ryabinin, B. Chen, P. Liang, C. Ré, I. Stoica, and C. Zhang. Flexgen: High-throughput generative inference of large language models with a single gpu. In *International Conference on Machine Learning*, pages 31094–31116. PMLR, 2023.

R. Taori, I. Gulrajani, T. Zhang, Y. Dubois, X. Li, C. Guestrin, P. Liang, and T. B. Hashimoto. Stanford alpaca: An instruction-following llama model. https://github.com/tatsu-lab/stanford_alpaca, 2023.

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.

T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, J. Davison, S. Shleifer, P. von Platen, C. Ma, Y. Jernite, J. Plu, C. Xu, T. L. Scao, S. Gugger, M. Drame, Q. Lhoest, and A. M. Rush. 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, Online, Oct. 2020. Association for Computational Linguistics. URL https://www.aclweb.org/anthology/2020.emnlp-demos.6.

M. Xia, T. Gao, Z. Zeng, and D. Chen. Sheared llama: Accelerating language model pre-training via structured pruning. *arXiv preprint arXiv:2310.06694*, 2023.

G. Xiao, J. Lin, M. Seznec, H. Wu, J. Demouth, and S. Han. Smoothquant: Accurate and efficient post-training quantization for large language models. In *International Conference on Machine Learning*, pages 38087–38099. PMLR, 2023.

G.-I. Yu, J. S. Jeong, G.-W. Kim, S. Kim, and B.-G. Chun. Orca: A distributed serving system for {Transformer-Based} generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 521–538, 2022.

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.

Y. Zhao, Y. Qu, K. Staniszewski, S. Tworkowski, W. Liu, P. Miłoś, Y. Wu, and P. Minervini. Analysing the impact of sequence composition on language model pre-training. *arXiv preprint arXiv:2402.13991*, 2024.

Z. Zheng, X. Ren, F. Xue, Y. Luo, X. Jiang, and Y. You. Response length perception and sequence scheduling: An llm-empowered llm inference pipeline. *arXiv preprint arXiv:2305.13144*, 2023.

Y. Zhong, S. Liu, J. Chen, J. Hu, Y. Zhu, X. Liu, X. Jin, and H. Zhang. Distserve: Disaggregating prefill and decoding for goodput-optimized large language model serving. *arXiv preprint arXiv:2401.09670*, 2024.

A. Appendix

B. Related Works

Due to space limitations in the main text, we include a detailed discussion on related works in the Appendix.

B.1. Accelerating Inference

Many advancements in accelerating LLM inference make architectural modifications that tradeoff quality with inference latency. These approaches include exploiting contextual sparsity (Liu et al., 2023), multiple decoding heads (Cai et al., 2024), model quantization (Xiao et al., 2023), and improved decoding algorithms such as speculative decoding which augments a base model with an "approximation model" (Leviathan et al., 2023). Packing has been applied to training to speed up training efficiency, while our work focuses on prefilling stage. (Zheng et al., 2023) reduces padding by clustering the prompts with similar response lengths into mini batches for generation. Packing and using independent mask can also be used for pre-training to decrease the distraction between documents (Zhao et al., 2024). Another active area of research is speeding up inference by improving low-level compute scheduling (Aminabadi et al., 2022; Sheng et al., 2023). Our approach for improving LLM throughput differs from the aforementioned techniques because: (1) Prepacking aims to improve prefilling efficiency, not training efficiency; (2) it does not require any architectural changes; (3) it can be fully implemented in PyTorch and is agnostic to the underlying hardware and cloud platforms.

B.2. LLM Serving

A relevant line of work takes a networking perspective on LLMs, in which a model must be "served" to clients that make requests. The core problem LLM serving addresses is the scheduling of inference, creating dynamic schedulers that optimize for throughput and latency. FasterTransformer (NVIDIA, 2021) increases decoding throughput but schedules at the request-level. To address this, Orca (Yu et al., 2022) proposes iteration-level scheduling which processes requests at finer granularity than a batch. PagedAttention in vLLM (Kwon et al., 2023) reduces KV-cache memory fragmentation with techniques inspired by virtual memory with paging. To mitigate wasted computation on padding, vLLM uses custom CUDA kernels that eliminate computation on padding by iterating through the sequences, while our implementation only uses PyTorch. The speed-up methods used in vLLM, mainly pagedAttention, which efficiently manages the vRAM, are orthogonal to our approach, and we believe a direct comparison with vLLM is not feasible. More recent and concurrent works such as Sarathi-Serve (Agrawal et al., 2024) and DistServe (Zhong et al., 2024) optimize a trade-off involving pre-filling and decoding. In our work, we specifically target pre-filling only. As such, our work directly improves TTFT and is complementary to other works that seek to improve decoding efficiency and throughput while minimizing stalling. Our work also has great usability with easy implementation in pytorch, without the need of writting custom CUDA kernel operations such as ragged tensors operations (Fegade et al., 2022).

C. Limitation of GPU parallelization

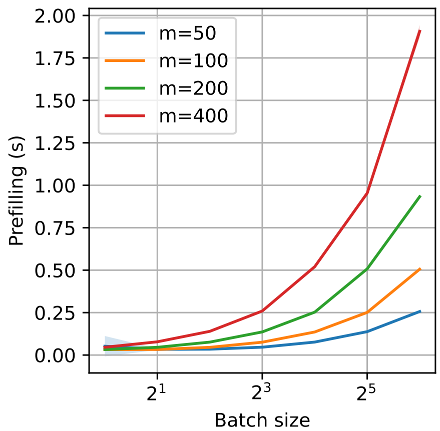

Figure 4: Prefilling latency scaling with batch size k highlights GPU parallelization limits. Results averaged over 100 runs.

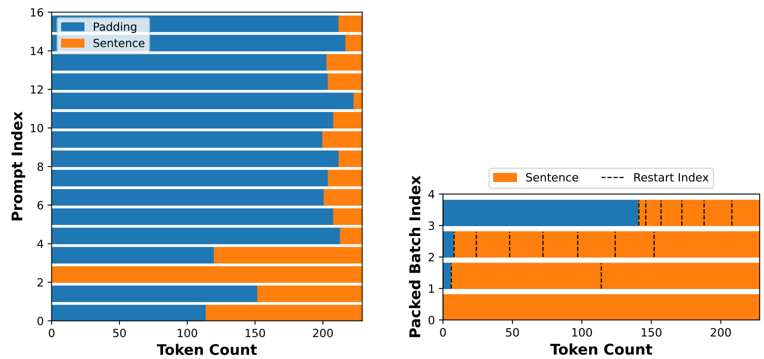

Figure 5: An actual example of one batch sampled from the MMLU dataset shows how a batch of 16 prompts is packed into a compact packed batch. Restart indices denote the point at where independent mask and position encoding are reset to preserve the semantics of each individual prompt.

Limitations of GPU Batch Parallelization Note that the above analysis assumes no parallelization over a batch. With perfect batch parallelization, prepacking will have better memory performance but no time improvement. We show empirically that GPUs cannot parallelize over batches without limitation. To show this, we sample a tensor of dimension (k, m), that is batch size k and prompt length m. In Figure 4, we demonstrate that for a fixed m, increasing k results in a higher latency. As the batch size grows, constraints such as memory bandwidth and synchronization overhead become more pronounced (Yuan et al., 2024). Prepacking exploits this by reducing batch size for a fixed sequence length m. Figure 5 illustrates an actual packing done by prepacking which greatly reduces paddings.

D. Algorithm Box for prepacking

Algorithm 1 The *Prepacking* Algorithm for Efficient Pre-Filling

Procedure Prepacking(Prompts $p_1, \cdots, p_k$, Transformer-based Language Model π)
  Prompt Lengths $l_1, \cdots, l_k \leftarrow len(p_1, \cdots, p_k)$
  Maximum Length $m \leftarrow \max_i l_i$
  Packed sequences $p'_1, \cdots, p'_r$, bins $[idx]_{1:r} \leftarrow$ BINPACK$(l_1, \cdots, l_k, m)$ $\{idx_i$ stores the start indices of the prompt(s) present in the packed sequence $p'_i\}$
  Batch $B \leftarrow$ TENSORIZE$(p'_1, \cdots, p'_r)$
  Independent Masks $[M']_{1:r} \leftarrow$ INDEPENDENT-MASK$([idx]_{1:r})$
  Restart Positional Encodings $[R]_{1:r} \leftarrow$ RESTART-PE$([idx]_{1:r})$
  Caches $KVs \leftarrow$ UNPACK$(\pi(B, [M']_{1:r}, [R]_{1:r}))$ $\{\pi$ will return a KV Cache, which we unpack to obtain prompt-specific caches$\}$
  return KVs
End Procedure

E. Experiment Setup

With constraints on our academic budget, all experiments are conducted on a single NVIDIA 48GB A6000 GPU connected to a Colfax CX41060s-EK9 4U Rackmount Server with AMD EPYC (Genoa) 9124 processors.

E.1. Datasets and Models

To profile prepacking's runtime performance under conditions representative of real-world user traffic, we evaluate on a diverse suite of datasets spanning question answering, summarization, instruction following, language modeling, and human preference modeling. Specifically, we use the MMLU (Hendrycks et al., 2021a), SamSum (Gliwa et al., 2019), Alpaca (Taori et al., 2023), Wikitext (Merity et al., 2016), and Anthropic HH RLHF (Bai et al., 2022) datasets. While not actually evaluating task performance, we leverage the variety of formats and prompt length distributions present in these datasets to simulate the diverse input queries a LLM may encounter from user requests in production environments. Due to computational constraints, we subsample 1000 prompts from each dataset, and the lengths statistics are presented in Table 1. We profile a range of language models to comprehensively assess runtime impacts of scale and architecture choices: the 1.3B Sharded LLAMA (Xia et al., 2023), 7B LLAMA 2 (Touvron et al., 2023) and Mistral (Jiang et al., 2023), and 13B LLAMA 2 (Touvron et al., 2023) spanning 1.3B to 13B parameters with varying configurations shown in Appendix Table 2. We profile them with 4 bit or 8 bit quantization due to computational constraints. Since prepacking aims to reduce wasted computation and memory on padding within batches, for fair evaluation, we do not manually construct batches. Instead, we use actual datasets to randomly sample batches and obtain aggregate metrics with respect to diverse prompt lengths. This also reflects a more realistic setting in which the flow of queries cannot be controlled.

E.2. Baselines

- ***Full Batching*:** As implemented by Huggingface, this method first determines the maximum prompt length across the batch and appends special padding tokens to shorter prompts until they match the maximum length. It then generates corresponding attention masks to ensure that the language model disregards the padded tokens during computation. Huggingface's inference framework (Wolf et al., 2020) employs this approach for handling prompts of variable lengths, serving as the basis for this baseline's profiling.

- ***Length-Ordered Batching*:** This baseline assumes access to the full set of user requests, serving as an oracle baseline that can first sort the inputs according to their lengths and sample batches in order to minimize the padding required when using the Full Batching. This method reduces computational overhead on paddings. However, it is not practical in real-world scenarios where user requests arrive in an unpredictable order, and the entire set of requests is not available upfront. In contrast, prepacking does not rely on this assumption, making it more suitable for handling dynamic and continuous streams of input prompts.

F. GPU Memory Saving and Utilization

We evaluate Prepacking's GPU memory efficiency, stemming from reduced computation on padded tokens, against other baselines in Figure 6. Prepacking consistently exhibits lower peak memory consumption, which directly translates to the ability to process larger batch sizes without encountering out-of-memory errors. For instance, with the Llama2-1.3B model on the MMLU dataset, prepacking can accommodate batch sizes up to 16x larger during prefilling compared to Full Batching before encountering OOM. This has significant implications for deploying models in resource-constrained environments, where maximizing hardware utilization is crucial. Consequently, as shown in Appendix Figure 12, Prepacking also exhibits lower GPU utilization when operating with the same batch size as the baselines, owing to its reduced computational overhead.

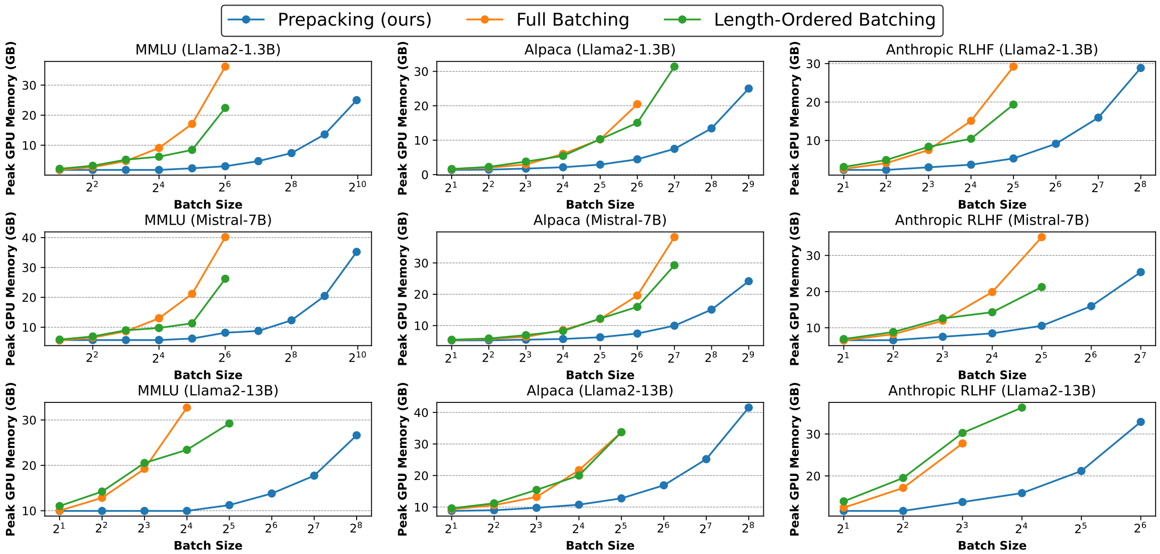

Figure 6: Peak GPU memory usage comparison across models and datasets on a single GPU. Absent data points indicate out-of-memory errors. Prepacking achieves lower peak GPU memory usage and allows for up to **16x larger batch sizes** in prefilling computations than Full Batching and Length-Ordered Batching.

G. Mean GPU utilization comparison

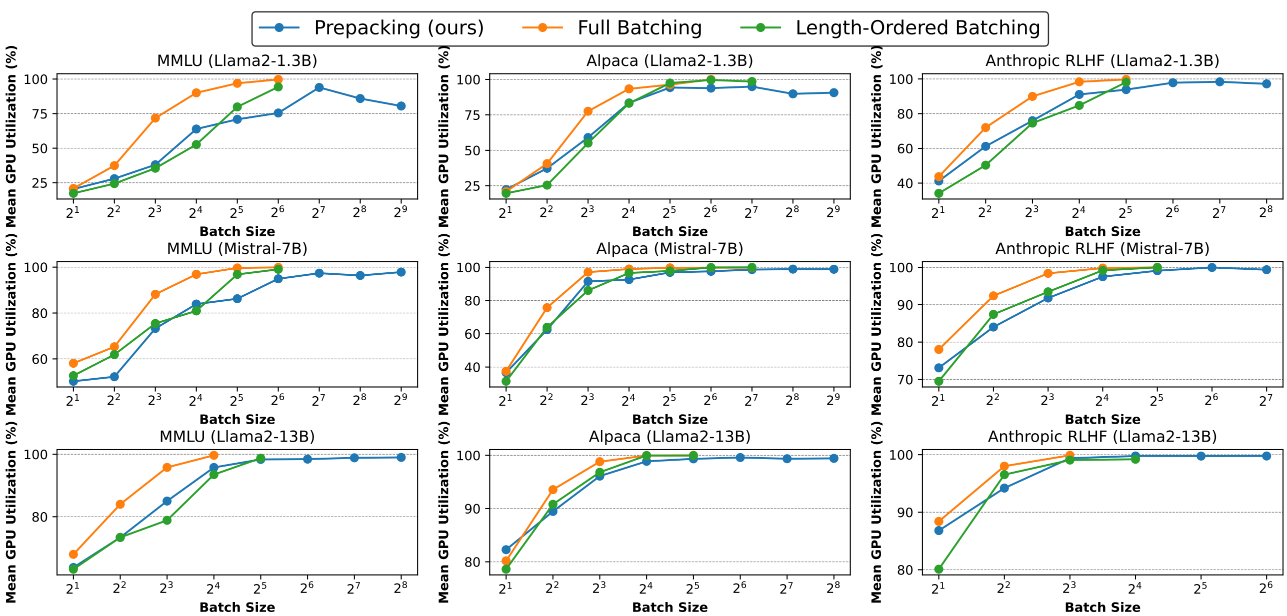

Figure 7: Mean GPU utilization for prefilling the prompts in datasets, sampled with a fixed batch size. Prepacking achieves lightest GPU utilization when the batch size is the same for every method.

H. Dataset Prepacking vs Length-Ordered Batching

In the previous experiments, we apply prepacking on randomly sampled batches from each dataset. However, this assumes the inability to control the contents of each batch. Given the ability to determine batches, a method to padding inefficiency would be to sort the dataset by length and batch accordingly. We refer to this baseline as *Length-Ordered Batching*. Alternatively, we can create batches after performing prepacking on the dataset as a whole and apply prepacking, i.e. *Dataset Prepacking*. We find that even in this scenario, where one might expect length-ordered batching to have a near optimal runtime by reducing the number of pad tokens, we observe prepacking still exhibits improvements as shown in Figure 8, where we compare the prefilling time per prompt.

To ensure a fair comparison with the length-ordered batching baseline, which operates under the assumption of having control over the entire dataset, we also apply prepacking at the dataset level. This entails initially employing a packing algorithm on the dataset and conduct prefilling on the packed requests with customized mask

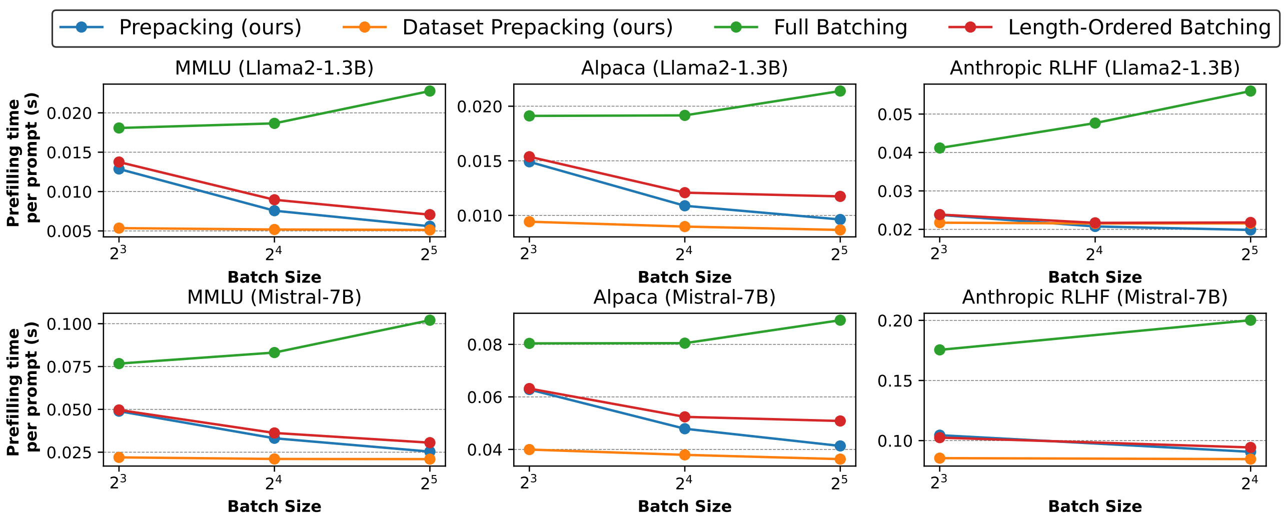

Figure 8: Comparison of prefilling time per prompt. The Dataset prepacking and Length-Ordered Batching benefit from access to the entire dataset, in contrast to Batch prepacking and Full Batching, which operate on a per-batch basis. Dataset prepacking further minimizes prefilling latency through packing when provided with full dataset access. Results are averaged over 5 runs.

I. Dataset length distribution statistics

Table 1: Evaluation Datasets Length Statistics. Due to computational resources constraints, we choose subsets from these datasets for evaluation.

Dataset name	Subset Min. / Mean / Max. SeqLength
Wikitext Max SeqLen 256 (Wiki256) (Merity et al., 2016)	1 / 73 / 256
Wikitext Max SeqLen 512 (Wiki512) (Merity et al., 2016)	6 / 120 / 512
MMLU (Hendrycks et al., 2021b)	4 / 64 / 1102
Anthropic HH RLHF (Bai et al., 2022)	22 / 247 / 1620
Alpaca (Taori et al., 2023)	43 / 126 / 527
SamSum (Gliwa et al., 2019)	21 / 169 / 942

J. Model details

Table 2: Model architecture used in the evaluations

Model	Num Params	Num layers	Hidden dim	Num heads
Sheared LLAMA 1.3B (Xia et al., 2023)	1.3B	24	2048	16
LLAMA 2 7B (Touvron et al., 2023)	7B	32	4096	32
Mistral 7B (Jiang et al., 2023)	7B	32	4096	32
LLAMA 2 13B (Touvron et al., 2023)	13B	40	4096	40

K. How does the performance gain scale with characteristics of lengths within a batch?

Previously in Section 3.3, we find the runtime of full batching is $\mathcal{O}(km^2)$. Prepacking is $\mathcal{O}(rm^2)$, where k is the original batch size, r is the batch size after prepacking, and m is the maximum prompt length. Therefore, we can estimate the speedup as a function of r/k (Batch Size Reduction). Because in practice it is difficult to predict r from the dataset statistics alone, we can also estimate the speedup as a function of $m - L/k$ (Max Absolute Deviation), which is how much the maximum length of a batch deviates from the mean length. We conduct the analysis on 5000 synthetic prompts with lengths uniformly distributed from 1 to 512, using the Llama2 1.3B model with batch size of 32. As can be seen in Figure **??**, these metrics can predict the speedup obtained by using prepacking over full batching. These findings offer insights into prepacking's performance scalability.

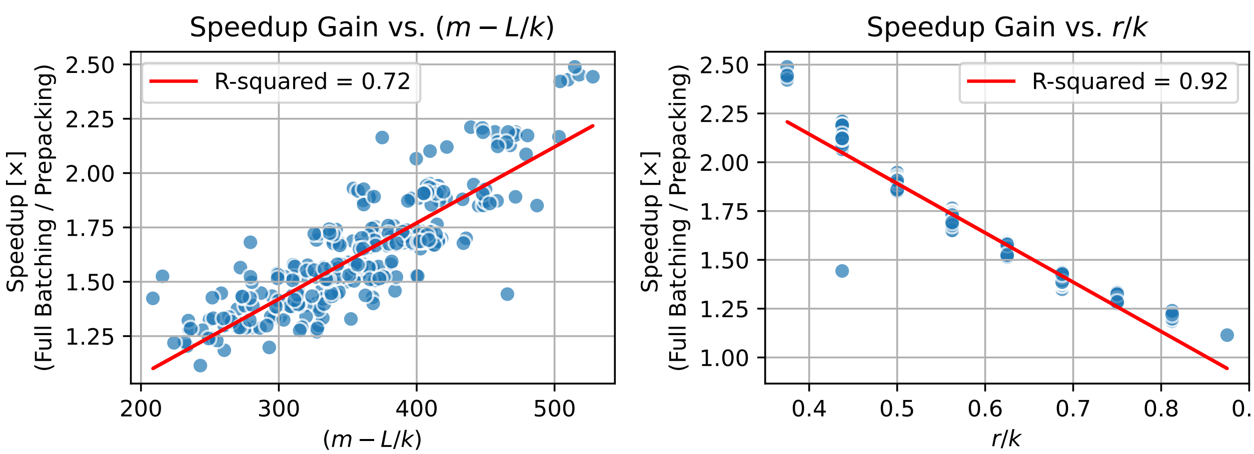

Figure 9: Speedup gains relative to full batching, with respect to Batch Size Reduction (r/k) and Max Absolute Deviation ($m - L/k$), conducted on Llama2 1.3B with batch size 16 and 5000 prompts.

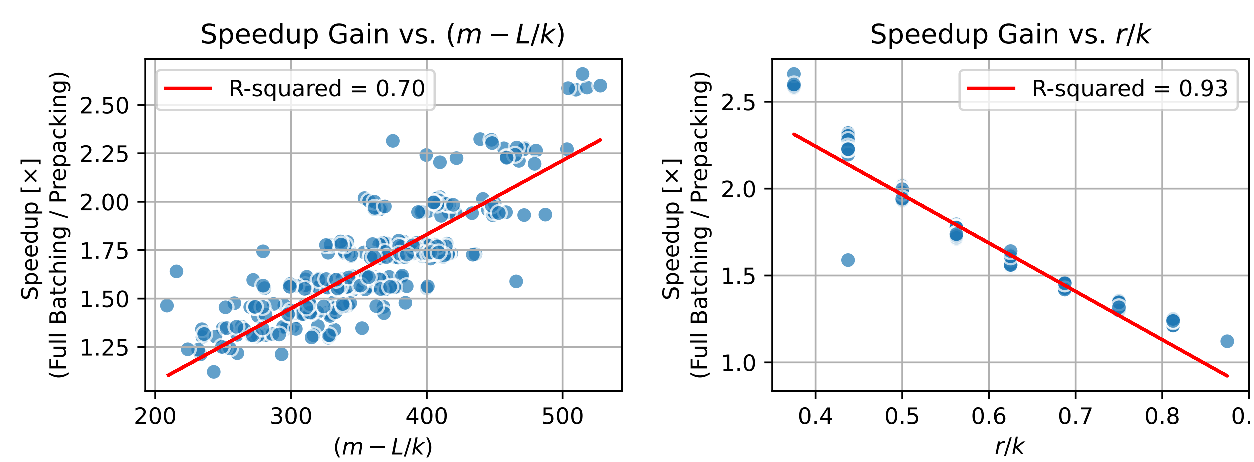

Figure 10: Speedup gains relative to full batching, with respect to Batch Size Reduction (r/k) and Max Absolute Deviation ($m - L/k$), conducted on Llama2 7B with batch size 16 and 2500 prompts.

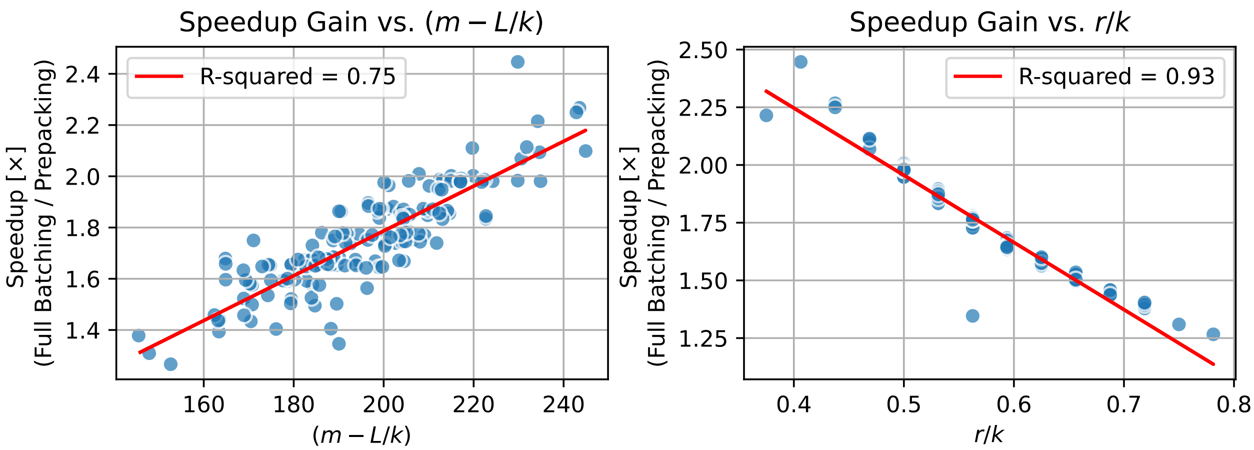

Figure 11: Speedup gains relative to full batching, with respect to Batch Size Reduction (r/k) and Max Absolute Deviation $(m - L/k)$, conducted on Llama2 7B with batch size 32 and 5000 prompts.

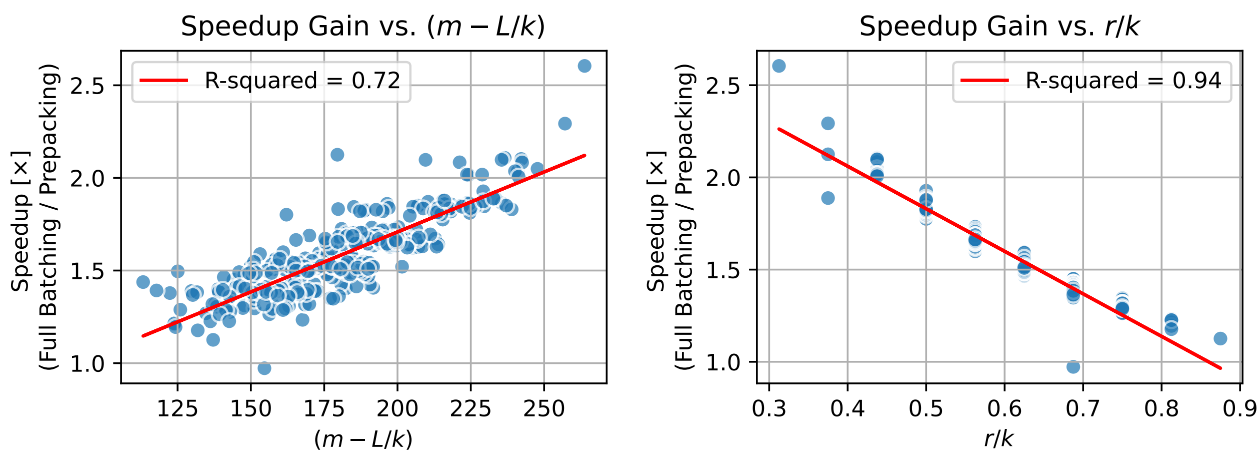

Figure 12: Speedup gains relative to full batching, with respect to Batch Size Reduction (r/k) and Max Absolute Deviation $(m - L/k)$, conducted on Mistral 7B with batch size 16 and 5000 prompts.

L. Limitations

Our prepacking algorithm is currently only used for the prefilling stage. After prepacking, our current approach is to repack and then generate normally. This is suboptimal as it involves extra bookkeeping and rearrangement in the memory space. Extending the prepacking algorithm to the generation stage will be an interesting and efficient future direction.

Additionally, we did not compare our method with hardware-aware approaches or CUDA kernel approaches, which can be more efficient than prepacking. However, we do not consider this a major limitation as we demonstrate the usability of our algorithm, which can be fully implemented in PyTorch.