FastDecode: High-Throughput LLM Serving through Disaggregating Attention Computation

Jiaao He [1] **Kezhao Huang** [1] **Jidong Zhai** [1]

Abstract

Cost of serving long-sequence large language models (LLM) is high, but the expensive and scarce GPUs are poorly efficient when generating tokens sequentially, unless the batch of sequences is enlarged. However, the batch size is limited by some constantly reused intermediate results, namely KV-Cache. They occupy too much memory to generate more and longer sequences simultaneously. While they could be offloaded to host memory, the CPU-GPU bandwidth is an inevitable bottleneck.

We find a way to decompose the transformer models into two parts of different characteristics, one of which includes the memory-bound KV-Cache accessing. Our key insight is that the aggregated memory capacity, bandwidth, and computing power of CPUs across multiple nodes is an efficient option to process this part. Performance improvement comes from reduced data transmission overhead and boosted GPU throughput to process the other model part. Evaluation results show that our system achieves $1.88\times - 5.04\times$ the throughput of vLLM when serving modern LLMs with the same GPU.

1. Introduction

The large language models (LLM) are gaining high attention. These transformer-based models are very hardware-friendly when training and evaluating (Narayanan et al., 2021; Ma et al., 2022), because the main computation workload is matrix multiplication, a highly optimized operation to run on accelerators, e.g., GPUs. However, when using the models, the auto-regressive procedure, i.e., decoding, is inefficient. Because tokens in a long sequence are generated one-by-one, one of the operand matrices is in fact a vector. Multiplying a vector with a matrix achieves much lower throughput due to poor utilization of GPUs.

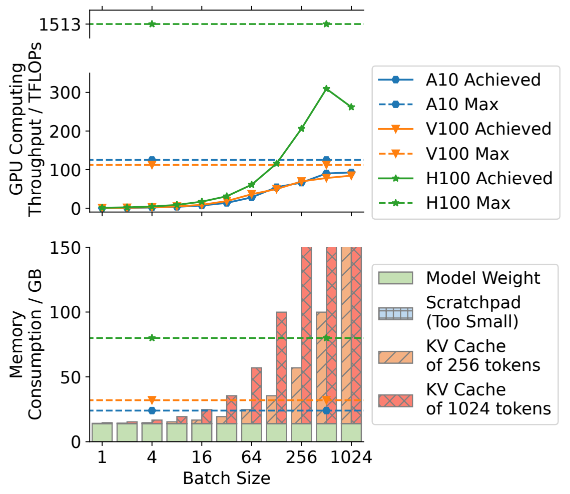

Figure 1. Memory footprint of *KV-cache* stops increasing GPU utilization by enlarging batch size

Enlarging batch size, i.e., generating tokens for multiple requests simultaneously, is the most feasible way to increase GPU utilization. However, generating a new token depends on huge intermediate results of generating the previous tokens, namely *KV-cache* (Pope et al., 2023). Processing batched requests results in a much larger memory footprint, far beyond the capacity of GPU memory. Figure 1 shows the dilemma instantiated on a common 7b model and several different GPUs. Increasing batch size makes the GPUs significantly better utilized, but the memory footprint of the *KV-cache* is much larger than the GPU memory. To make it worse, the *KV-cache* becomes even larger as more tokens are generated and the sequences get longer.

To solve this, host memory has naturally become the place to offload the KV-cache (Kwon et al., 2023), as it is larger and cheaper than GPU memory. However, the *KV-cache* is not cold data: the complete *KV-cache* is loaded into GPU memory to generate every token, leading to a large amount of data movement and overhead.

However, as shown in Figure 2, we find that compared with the huge gap in compute power, GPUs and CPUs have a much closer gap in memory bandwidth. Therefore, instead of just offloading *KV-cache* to CPU memory, we should

[1]Tsinghua University. Correspondence to: Jidong Zhai <zhai-jidong@tsinghua.edu.cn>.

Proceedings of the 1st Workshop on Long-Context Foundation Models, Vienna, Austria. 2024. Copyright 2024 by the author(s).

compute near KV-cache on CPUs. The transmitted tensors then change from *KV-cache* data to activation tensors, which are orders of magnitudes smaller than KV-cache. On the other hand, our approach totally removes intermediate data of sequences, the *KV-cache*, from GPU memory. So the batch size can be greatly increased, and the GPUs can be optimally utilized.

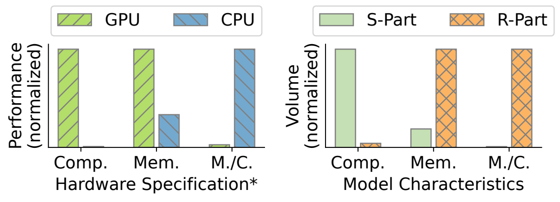

Figure 2. Performance characteristics of typical GPUs and CPUs, matching the need of two parts of the model

2. Challenges and Solutions

To enable KV-cache computation and storage on the CPU side, there are three challenges.

Challenge 1: The CPU is busy but slow. It runs multiple tasks, including batch gathering, tokenization, and coordinating the GPUs. Performing extra computation interferes with these tasks. To add to the difficulty, the memory bandwidth of a CPU is lower than GPU.

Challenge 2: The pattern of workload variation, as the generated sequences get longer, differs between the two parts. In our solution, the CPU and the GPU take turns to perform computation, and pass the results to each other. A basic pipeline of multiple batches of requests is used to utilize both of them. However, computation on the CPU takes longer time as the generated sequence gets longer, while the latency of its counterpart on GPU does not change at all. This makes it hard to always utilize both CPU and GPU.

Challenge 3: Careful orchestration is needed to balance the performance of both types of hardware. Bottleneck may be either of the GPU or CPU, because they are tightly coupled. We need to balance the two considering the heterogeneous hardware and token generation workload. We seek for a minimum CPU requirement that can fully exploit the compute power of the GPU.

Our system, FASTDECODE, is a CPU-GPU heterogeneous pipeline for LLM inference that addresses the challenges by the following innovations.

Solution 1: We employ multiple out-of-chassis remote CPUs for *KV-cache* and the related computation. The aggregated memory capacity and bandwidth of the system are scaled up. The distributed CPUs can achieve sufficient throughput to saturate the GPU, with moderate communication overhead.

Solution 2: We invent a sequence-level load-stabilizing schedule to minimize idling and better utilize both types of hardware. The workload on a CPU is proportional to the total length of sequences it maintains. To keep the latency stable, sequences are fed into the system following a workload control algorithm. Short and long sequences are simultaneously processed by CPU workers, leaving the total length of sequences stable. As a result, the overall latency of CPUs changes more gently, and both types of hardware are better utilized.

Solution 3: We adopt a model-guided approach to orchestrate the GPU with CPUs. It quantitatively characterizes the performance bottleneck considering different aspects of the LLM inference tasks. Aggregated memory bandwidth is identified as the key metric in selecting the CPUs. For a given model and GPU setup, based on profiling result of a micro-benchmark, we can estimate the minimum required aggregated CPU memory bandwidth for different batch sizes.

Overall, the throughput of a single GPU is saturated with a significantly larger batch size. Thanks to the scalability and aggregated power of CPUs across nodes, high overall token generation throughput is achieved with affordable GPU resources. In our evaluation, up to $5\times$ throughput of vLLM is achieved on the same GPU with acceptable latency.

3. System Design

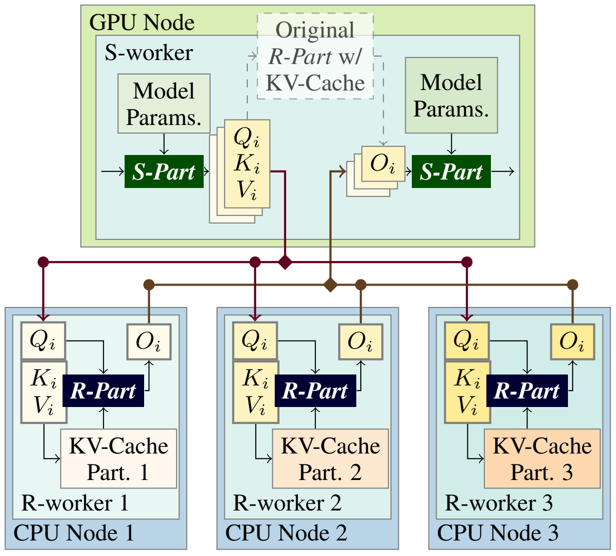

Figure 3. FASTDECODE system design

Figure 3 shows the basic design of FASTDECODE, which consists of two types of workers, related to two parts of the inference process of a transformer model.

- ***R-Part*** denotes the memory-intensive attention operation over the query vector and *KV-cache*.
- ***S-Part*** denotes the rest dense layers, where the throughput can benefit from increasing batch size.

An *S-worker* computes *S-Part* of an LLM. It may use one or multiple GPUs. All weights of the model are on the S-worker, and partitioned by a certain way of model parallelism if using multiple GPUs. It acts as a typical token generation worker simply using GPUs, except for its much larger batch size and the behavior of computing *R-Part*. To generate a new token, it goes through the transformer blocks. After Q_i, K_i, V_i are produced by fully connected layers in *S-Part*, instead of computing *R-Part* locally, the S-worker sends different parts of them related to different sequences to the R-workers, and retrieve the output, O_i, from them. Then, it feeds O_i to succeeding layers in *S-Part* on the GPU.

The *R-workers* may use CPUs on remote nodes to compute *R-Part* with high aggregated throughput. These R-workers are light-weight, because no model parameter is involved in *R-Part* of LLMs. The functionality of a R-worker is simple. It receives Q_i, K_i, V_i of a batch of tokens. K_i and V_i are appended to the existing KV-cache. Q_i is used to in attention computation with the local KV-cache data, and the output is returned. The R-workers may also drop KV-cache of a certain sequence upon its generation ends.

As GPU resource is scarce, the system maximizes the throughput of the S-worker via maximizing the batch size. As *KV-cache* is excluded from the S-worker, there is little tension on GPU memory capacity. Beside the model weight, it only needs memory for a small scratchpad of the current layer. The batch size can be up to millions of sequences. While a large batch leads to high latency, it is now possible to increase the GPU utilization in certain cases.

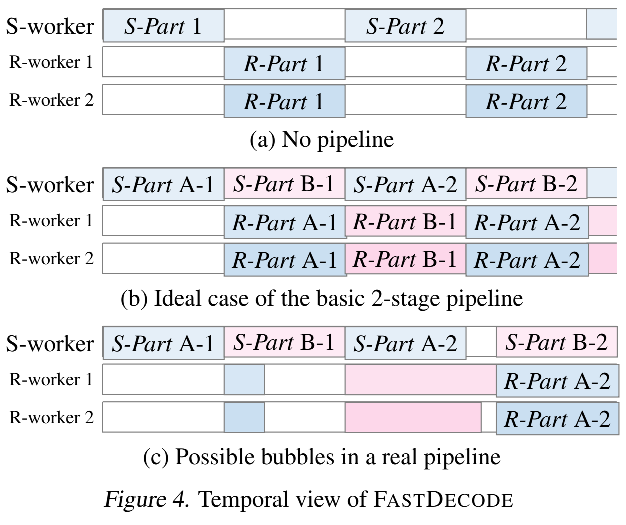

Figure 4. Temporal view of FASTDECODE

In this system, the S-worker and the R-workers work in turns to generate a token. When one type of worker is working, the other idles, as shown in Figure 4(a).

Therefore, a basic two-stage pipeline at token level is applied. As Figure 4(b) shows, the S-worker starts with two separate mini-batches: A and B. After it finishes the first *S-Part* of mini-batch A, it starts working on the first *S-Part* of mini-batch B. At the same time, the R-workers are working on the *R-Part* of mini-batch A. Overall, the two mini-batches are processed by each type of worker in turns.

This token-level pipeline only achieves optimal utilization of all workers if computation latency of *S-Part* and *R-Part* are equal. Otherwise, there are still bubbles in the pipeline, as Figure 4(c) suggests. In fact, the pipeline can barely be free of bubbles, because the workload and the hardware are both of high heterogeneity. Specially, the latency of *S-Part* is related to the number of sequences, while the latency of *R-Part* is related to the total length of the sequences. As the sequences get longer, *R-Part* takes more time, while the latency of *S-Part* remains unchanged.

We apply a load-stabilizing schedule over the decoding task of sequences. The schedule focuses on keeping the total length of the sequences being generated stable. It controls when the system starts to generate tokens for sequences in a batch, targeting on avoiding bursts of workload on the R-workers. It can provide up to 20% throughput gain.

Then, with better predictable load of both parts, we calculate the optimal orchestration of S-worker and R-workers using a fixed formula for each given generation task and hardware configuration.

4. Evaluation

4.1. Setup

Models and tasks We choose two state-of-the-art open-source LLMs, Llama (Touvron et al., 2023b) and OPT (Zhang et al., 2022). We evaluate our system over Llama-7b, and Llama-13b. We reduce the number of layers to reduce evaluation cost, and report the estimated throughput and latency of the original model.

Hardware We use a NVIDIA A10 GPU with 24 GB device memory as the S-worker. The node has 256 GB host memory as swap space of vLLM. Up to 4 additional nodes with dual sockets of AMD Epyc CPUs are used as the R-workers. The cluster is connected via Infiniband network.

Baselines We use vLLM (Kwon et al., 2023), TensorRT-LLM (Ten), FastLLM (fas), and a Vanilla implementation (van) of Llama (Touvron et al., 2023a;b) as our baselines. Different from FASTDECODE, these popular LLM serving systems only use GPUs for computation.

4.2. Performance

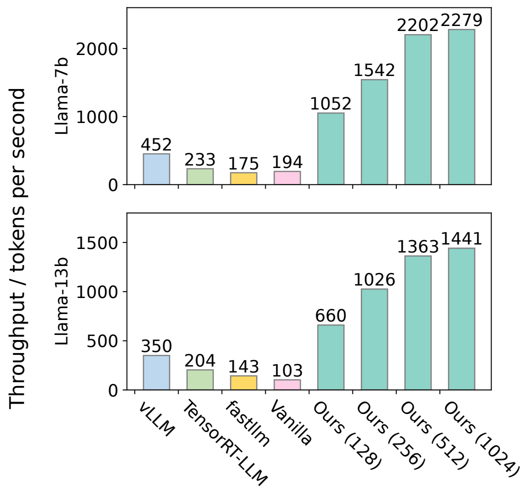

Figure 5. Token generating throughput

Maximum Throughput Figure 5 shows the measured throughput of all the systems. The number in brackets after *ours* indicates the batch size of FASTDECODE. The possible batch size is enormous in our system, because the distributed host memory is large enough for thousands of sequences. Increasing the batch size can increase the utilization of GPUs, and thus the overall throughput. However, as there may be constraint on the latency, the batch size should be set properly. Also, we observe that the performance gain of increasing batch size gets less when the batch size is large enough. When the batch size increases by $8\times$ from 128 to 1024, we only get $2\times$ throughput.

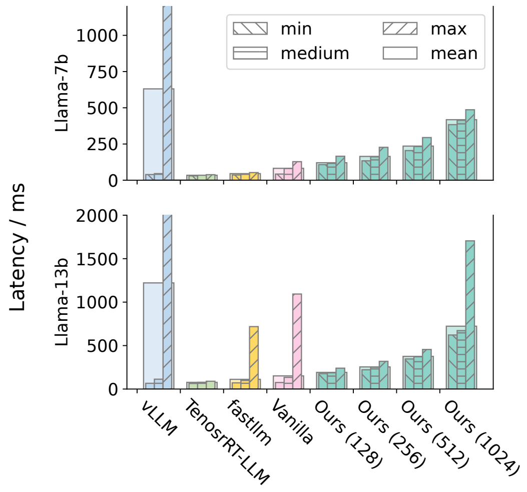

Figure 6. Token generating latency

Token Generating Latency Figure 6 shows the measured latency to generate a new token by all the systems. The wide bar indicates the average latency between generating two adjacent token, and the three narrow bars show $P = 0.01/0.5/0.99$ latency, respectively. When we maximize our batch size to target on highest throughput, the latency is about $3.5\times$ the latency when using $8\times$ smaller batch size. This also implies the GPU utilization improvement of increased batch size.

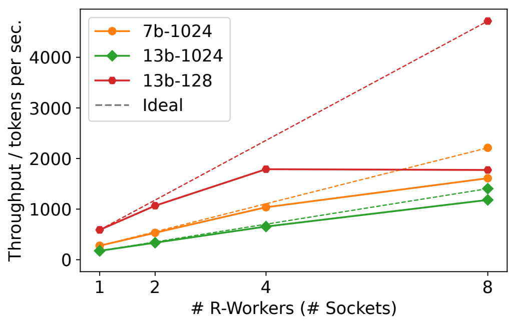

Figure 7. Strong scalability of FASTDECODE

Scalability Figure 7 shows the strong scalability experiment results of FASTDECODE over the 7b and 13b model. When the length of sequences is 1024, FASTDECODE achieves 72.8% and 84.1% efficiency when scaling up from 1 socket to 8 sockets, on the 7b and 13b models, respectively. As the total latency is smaller in the 7b model, overhead of the pipeline is more significant, leading to lower efficiency with 8 sockets. When sequence is as short as 128, the efficiency is 37.6% for the 13b model. Using 8 sockets achieves even lower throughput than using 4 sockets with 75.9% efficiency. This is implied by our performance model. Shorter sequences require less R-workers. Employing more R-workers does not increase the performance when the S-worker is the bottleneck.

5. Conclusion

In this paper, we propose FASTDECODE, a system that achieves high throughput of generating tokens with long-context LLMs using affordable GPU resources. Different from typical solutions that fully use GPUs for computation, we decompose the model into two parts, and move both storage and computation of the memory-bound part to distributed out-of-chassis CPUs, utilizing their aggregated compute power. Performance challenges brought by heterogeneity in both temporally varying workload and hardware are addressed by a sequence-level load-stabilizing schedule and a performance model. Finally, as the GPU is better utilized thanks to greatly enlarged batch size, and the overall throughput is competitive.

References

GitHub - NVIDIA/TensorRT-LLM: TensorRT-LLM provides users with an easy-to-use Python API to define Large Language Models (LLMs) and build TensorRT engines that contain state-of-the-art optimizations to perform inference efficiently on NVIDIA GPUs. TensorRT-LLM also contains components to create Python and C++ runtimes that execute those TensorRT engines. — github.com. `https://github.com/NVIDIA/TensorRT-LLM`. [Accessed 07-06-2024].

GitHub - ztxz16/fastllm. `https://github.com/ztxz16/fastllm`. [Accessed 07-06-2024].

GitHub - meta-llama/llama: Inference code for Llama models — github.com. `https://github.com/meta-llama/llama`. [Accessed 07-06-2024].

Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J., Zhang, H., and Stoica, I. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023*, pp. 611–626. ACM, 2023. doi: 10.1145/3600006.3613165. URL `https://doi.org/10.1145/3600006.3613165`.

Ma, Z., He, J., Qiu, J., Cao, H., Wang, Y., Sun, Z., Zheng, L., Wang, H., Tang, S., Zheng, T., Lin, J., Feng, G., Huang, Z., Gao, J., Zeng, A., Zhang, J., Zhong, R., Shi, T., Liu, S., Zheng, W., Tang, J., Yang, H., Liu, X., Zhai, J., and Chen, W. Bagualu: targeting brain scale pretrained models with over 37 million cores. In *PPoPP '22: 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Seoul, Republic of Korea, April 2 - 6, 2022*, pp. 192–204. ACM, 2022. doi: 10.1145/3503221.3508417. URL `https://doi.org/10.1145/3503221.3508417`.

Narayanan, D., Shoeybi, M., Casper, J., LeGresley, P., Patwary, M., Korthikanti, V., Vainbrand, D., Kashinkunti, P., Bernauer, J., Catanzaro, B., Phanishayee, A., and Zaharia, M. Efficient large-scale language model training on GPU clusters using megatron-lm. In *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2021, St. Louis, Missouri, USA, November 14-19, 2021*, pp. 58. ACM, 2021. doi: 10.1145/3458817.3476209. URL `https://doi.org/10.1145/3458817.3476209`.

Pope, R., Douglas, S., Chowdhery, A., Devlin, J., Bradbury, J., Heek, J., Xiao, K., Agrawal, S., and Dean, J. Efficiently scaling transformer inference. *Proceedings of Machine Learning and Systems*, 5, 2023.

Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., Rodriguez, A., Joulin, A., Grave, E., and Lample, G. Llama: Open and efficient foundation language models. *CoRR*, abs/2302.13971, 2023a. doi: 10.48550/ARXIV.2302.13971. URL `https://doi.org/10.48550/arXiv.2302.13971`.

Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., Bashlykov, N., Batra, S., Bhargava, P., Bhosale, S., Bikel, D., Blecher, L., Canton-Ferrer, C., Chen, M., Cucurull, G., Esiobu, D., Fernandes, J., Fu, J., Fu, W., Fuller, B., Gao, C., Goswami, V., Goyal, N., Hartshorn, A., Hosseini, S., Hou, R., Inan, H., Kardas, M., Kerkez, V., Khabsa, M., Kloumann, I., Korenev, A., Koura, P. S., Lachaux, M., Lavril, T., Lee, J., Liskovich, D., Lu, Y., Mao, Y., Martinet, X., Mihaylov, T., Mishra, P., Molybog, I., Nie, Y., Poulton, A., Reizenstein, J., Rungta, R., Saladi, K., Schelten, A., Silva, R., Smith, E. M., Subramanian, R., Tan, X. E., Tang, B., Taylor, R., Williams, A., Kuan, J. X., Xu, P., Yan, Z., Zarov, I., Zhang, Y., Fan, A., Kambadur, M., Narang, S., Rodriguez, A., Stojnic, R., Edunov, S., and Scialom, T. Llama 2: Open foundation and fine-tuned chat models. *CoRR*, abs/2307.09288, 2023b. doi: 10.48550/ARXIV.2307.09288. URL `https://doi.org/10.48550/arXiv.2307.09288`.

Zhang, S., Roller, S., Goyal, N., Artetxe, M., Chen, M., Chen, S., Dewan, C., Diab, M. T., Li, X., Lin, X. V., Mihaylov, T., Ott, M., Shleifer, S., Shuster, K., Simig, D., Koura, P. S., Sridhar, A., Wang, T., and Zettlemoyer, L. Opt: Open pre-trained transformer language models. *CoRR*, abs/2205.01068, 2022. doi: 10.48550/ARXIV.2205.01068. URL `https://doi.org/10.48550/arXiv.2205.01068`.