HyperGen: Optimizing Generative Inference with Long Prompts for Resource-Constrained Systems

Lingwen Gong
Huazhong University of Science and Technology
Wuhan, China

Kaixin Liu
Huazhong University of Science and Technology
Wuhan, China

Xiaolu Li
Huazhong University of Science and Technology
Wuhan, China

Shujie Han
Northwestern Polytechnical University
Xi'an, China

Patrick P. C. Lee
The Chinese University of Hong Kong
Hong Kong, China

Yuchong Hu
Huazhong University of Science and Technology
Wuhan, China

Dan Feng
Huazhong University of Science and Technology
Wuhan, China

Abstract

Generative inference with long prompts often exceeds GPU memory limits in resource-constrained systems (e.g., workstations and edge devices), thereby causing inference failures. We present HyperGen, a lightweight generative inference framework that optimizes the prefill stage (i.e., when input prompts are processed) via two fine-grained partitioning techniques: (i) partitioning and loading model parameters with size awareness into GPU memory; and (ii) partitioning computations of different inference steps to fit into GPU memory and offloading concatenation of partial results to CPU memory. Evaluation shows that HyperGen supports a maximum prompt length of up to 3.8× longer than an existing GPU-based inference approach and reduces the time-to-first-token from hours to seconds compared to the CPU-based prefill approach.

CCS Concepts

• **Computer systems organization → Pipeline computing**; • **Information systems → Hierarchical storage management**.

Keywords

Generative inference; Resource-constrained system; Memory management

APSys '25, Seoul, Republic of Korea

ACM ISBN 979-8-4007-1572-3/25/10
https://doi.org/10.1145/3725783.3764409

ACM Reference Format:
Lingwen Gong, Kaixin Liu, Xiaolu Li, Shujie Han, Patrick P. C. Lee, Yuchong Hu, and Dan Feng. 2025. HyperGen: Optimizing Generative Inference with Long Prompts for Resource-Constrained Systems. In *16th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys '25), October 12–13, 2025, Seoul, Republic of Korea.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3725783.3764409

1 Introduction

Generative inference is critical for applications such as code generation [1, 15, 17] and healthcare [7, 14, 25], where input prompts often exceed thousands of tokens to produce contextually relevant outputs. Meanwhile, there is a growing demand for deploying these applications in resource-constrained systems, including workstations for enhanced data security [21, 27] and edge devices for personalized recommendation [5, 27].

Enabling efficient generative inference with long prompts is challenging with limited resources. First, modern large-scale models require substantial GPU memory space that surpasses the capacities of commodity workstations and edge devices. For example, GPT-3, with 175 billion parameters, requires approximately 350 GiB of memory [16, 22]; the smaller OPT-30B, with 57 GiB of parameters (quantized to 2 bytes per parameter), still exceeds the memory capacity of the consumer-grade NVIDIA RTX 4090 GPU with only 24 GiB of memory. Caching all model parameters in GPU memory is thus infeasible. Second, processing long prompts significantly increases GPU memory demands (§2.1). Our evaluation (§4) shows that FlexGen [18], a generative inference system designed for consumer-grade GPUs, returns inference failures when processing 4096-token prompts on an NVIDIA RTX 4090 GPU.

We propose HyperGen, a lightweight generative inference system aiming to optimize long prompts in resource-constrained systems. HyperGen focuses on optimizing the prefill stage, while FlexGen targets the decode stage (see §2.1

for background details on generative inference). It applies *fine-grained partitioning* to GPU memory management in two aspects. First, it partitions model parameters into smaller variable-size groups, which are loaded into GPU memory with size awareness. Second, it partitions operations with high memory demands into sub-operations that fit within GPU memory, and offloads partial results to the CPU for concatenation to utilize both GPU and CPU resources.

We implement HyperGen atop FlexGen [18] and evaluate it in a resource-constrained environment. Compared with FlexGen, HyperGen increases the maximum allowed prompt length by up to 3.8×. Compared to the CPU-based prefill approach, HyperGen reduces the time-to-first-token from hours to seconds. Its prefill-stage optimization also complements FlexGen by increasing its throughput by up to 25.2%. Our HyperGen prototype is now open-sourced at **https://github.com/ukulililixl/hypergen**.

2 Background and Motivation

2.1 Challenges of Generative Inference

Generative inference comprises the *prefill* and the *decode* stages. In the prefill stage, input prompts are tokenized and processed via multiple *transformer blocks* [20] to initialize transient states for the decode stage. Each transformer block generates *query*, *key*, and *value* (QKV) matrices, computes self-attention, and processes the results through a *multi-layer perceptron (MLP)*. The decode stage operates iteratively, generating one output token per iteration by updating QKV matrices, computing self-attention, and processing through the MLP in each transformer block. To operate within GPU memory constraints, generative inference in resource-constrained environments typically offloads transient states to CPU memory or persistent storage, thereby allowing the prefill stage and each iteration of the decode stage to proceed within GPU memory [18, 19].

However, modern large-scale models include billions of parameters and consume substantial GPU memory (§1). In addition, both prefill and decode stages process prompts through multiple transformer blocks, each requiring distinct model weights and generating unique transient states (e.g., QKV matrices and other temporary results within each transformer block), and hence lead to significant GPU memory usage as the prompt length increases. Our analysis reveals that GPU memory remains the major limiting factor for generative inference, even when transient states are offloaded. Figure 1 shows the peak GPU memory usage in FlexGen with the OPT-30B model for various prompt lengths (see §4 for testbed details) with an output length of 32 tokens by default. As the prompt length increases from 1,000 to 4,000 tokens, the peak GPU memory usage in the prefill stage increases significantly from 2.3 GiB to 15.9 GiB, while that in the decode stage remains stable at 2.1-2.6 GiB. This disparity occurs because transformer blocks in the prefill stage process the entire prompts, while those in the decode stage only process a single output token per iteration. Thus, inference failures can occur during the prefill stage due to insufficient GPU memory before generating the first output token.

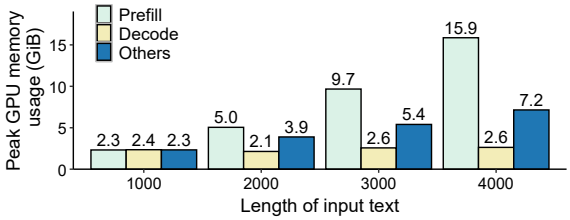


Figure 1: Peak GPU memory usage in the prefill and decode stages in FlexGen with the OPT-30B model; *Others* refers to the GPU memory reserved by PyTorch [10].

2.2 Opportunities for Optimization

We explore partitioning to solve generative inference as multiple sub-problems and optimize GPU memory usage for resource-constrained systems. Here, we identify two partitioning opportunities for our HyperGen design.

Opportunities for model parameters. Partitioning model parameters into smaller, more manageable units offers significant potential for GPU memory optimization. We find that FlexGen [18] partitions model parameters based on their usage in the prefill and decode stages prior to generative inference execution. For example, in the OPT-30B model, FlexGen partitions model parameters into 773 groups. Such parameter groups also exhibit substantial size variation, with 482 groups smaller than 60 KiB each (totaling 8.6 MiB), 290 groups exceeding 90 MiB each (totaling 56.5 GiB), and one group of size 28.0 MiB (storing position embeddings). This variation suggests that caching all model parameters in GPU memory is impractical, while selectively caching small groups, which contribute negligibly to memory overhead, can optimize GPU memory utilization.

Opportunities for transient states. The prefill stage can be decomposed into fine-grained steps of processing transient states with predictable memory requirements. Each transformer block in the prefill stage comprises five steps: QKV calculation, attention score calculation, softmax, attention output generation, and MLP processing. Figure 2 shows the peak GPU memory usage across these steps for prompt lengths from 1,000 to 4,000 tokens. We observe significant variation in memory usage for different steps. Notably, the peak memory usage for each step is predictable for fine-grained computation scheduling (§3.2).

Furthermore, memory-intensive operations, including attention score calculation, softmax, and attention output

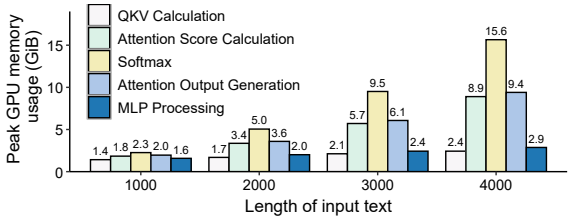


Figure 2: Peak GPU memory usage for different steps in the prefill stage under different prompt lengths.

generation, can be decomposed into sub-operations that fit within GPU memory. For instance, matrix multiplications in attention score calculation and attention output generation are partitionable into smaller sub-matrix multiplications.

2.3 Related Work

The most related work to ours includes FlexGen [18] and PowerInfer [19], both of which are designed for high-throughput generative inference on consumer-grade GPUs and target decode-stage optimization. FlexGen searches for efficient strategies to offload transient states to CPU memory and disk, subject to hardware constraints, and further compresses transient states to reduce I/O costs. PowerInfer partitions the working set based on operation frequency, and executes frequent operations on GPUs and less frequent ones on CPUs. However, for long prompts, FlexGen may fail in inference during the GPU-based prefill stage, while PowerInfer suffers from inefficient CPU-based prefill processing. In contrast, HyperGen focuses on prefill-stage optimization for long prompts and is compatible with FlexGen for further performance enhancements (§4).

Our work focuses on generative inference with long prompts and addresses a critical challenge in long-context generative inference. Many research efforts address long-context processing by carefully managing memory footprints for transient states. InfiniGen [11] speculates on and retains only critical entries. IMPRESS [3] exploits cross-head similarity to identify crucial tokens. RetroInfer [4] leverages attention sparsity to reduce memory footprints. MagicPig [6] uses locality-sensitive hashing for selection to preserve accuracy. ARKVALE [2] introduces a page-based offloading technique to adapt to importance shifts across decoding steps. HEADINFER [12] employs a head-wise offloading technique to reduce GPU memory footprints. SpecOffload [28] utilizes speculative decoding to exploit idle GPU resources during long-context processing. However, in resource-constrained environments, the effectiveness of these solutions may still be limited when GPU memory is the primary bottleneck. HyperGen offers a complementary solution to specifically address resource-constrained environments.

There are other efficient generative inference approaches. TwinPilots [24] optimizes the decode stage and reduces data transfers between GPU and CPU memories. OmniKV [9] and H_2O [26] compress transient states to improve prefill-stage performance. LazyLLM [8] selectively computes transient states for only the most relevant tokens. However, even with compression, high memory demands may persist for very long prompts (§2.1). HyperGen explores fine-grained partitioning techniques to relieve memory demands.

3 HyperGen Design

HyperGen addresses GPU memory limitations in generative inference with long prompts in resource-constrained systems. It employs two fine-grained partitioning strategies: (i) size-aware parameter partitioning and (ii) step-aware computation partitioning.

3.1 Size-aware Parameter Partitioning

HyperGen performs an offline pre-processing phase on model parameters prior to the execution of generative inference. It first divides model parameters into independent groups based on transformer blocks, where parameters within the same group are used by the same transformer block. Each group is further divided into independent *fragments*, where each fragment represents a weight matrix used in a specific computation (e.g., each of the model weight matrices for query, key, and value calculations is assigned to a single dedicated fragment). This fine-grained partitioning enables efficient access of model parameters in resource-constrained systems.

HyperGen further employs a *size-aware caching* for fragment management to optimize GPU memory usage. It classifies fragments based on their memory footprints into small and large fragments. It caches only small fragments in GPU memory within a predefined *GPU allocation limit*, which is configurable to accommodate different models. For example, we set a default limit of 10 MiB in total for the OPT-30B model. HyperGen then sorts all fragments in ascending order by size and applies a smallest-first caching policy (i.e., smaller fragments have a higher priority to be stored in GPU memory) until the GPU allocation limit is reached.

Unlike FlexGen [18], which repeatedly loads small fragments from disk to GPU memory for each transformer block, HyperGen's size-aware caching always keeps small fragments in GPU memory as they can be accessed by different transformer blocks during generative inference, while they contribute to limited memory overhead (§2.2). Our evaluation shows that size-aware caching significantly enhances generative inference performance (§4).

3.2 Step-aware Computation Partitioning

Memory usage estimation. To accurately partition computations and allocate them into GPU memory, HyperGen

Steps	Peak memory	Retained memory
QKV calculation	$4bsh_1 + 3h_1^2$	$4bsh_1$
Attention score calculation	$4bsh_1 + nbs^2$	$2bsh_1 + nbs^2$
Softmax	$2bsh_1 + 2nbs^2$	$2bsh_1 + nbs^2$
Attention output generation	$3bsh_1 + h_1^2 + nbs^2$	bsh_1
MLP processing	$2bsh_1 + 2h_1h_2 + bsh_2$	bsh_1

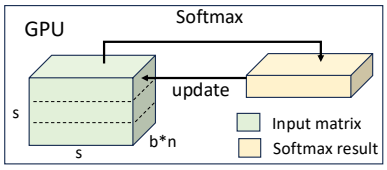
Table 1: Estimation of peak and retained memory usage before and after each step in the prefill stage.

first divides the prefill stage into five distinct steps (§2.2) and estimates memory requirements for each step before executing generative inference. Such estimations can be feasibly done based on key model parameters, including input prompts length (s), batch size (b), hidden layer dimensions (h_1 for attention and h_2 for MLP processing), and the number of attention heads (n).

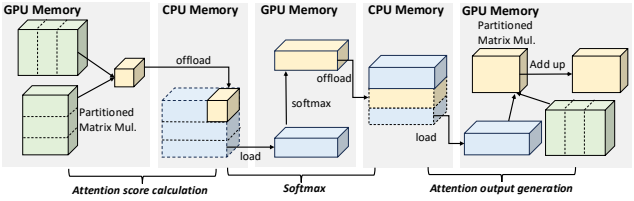
The peak memory usage for each step comprises three components (see Table 1), including (i) memory retained from the previous step, (ii) memory used for model parameter fragments, and (iii) memory for generating transient states. For example, in the QKV calculation step, HyperGen estimates the peak memory usage starting with the input token matrix $\mathbf{X}$, with dimensions $[b \times s \times h_1]$. To compute the query, key, and value matrices ($\mathbf{Q}$, $\mathbf{K}$, and $\mathbf{V}$, respectively), HyperGen loads model parameter groups $\mathbf{W}_Q$, $\mathbf{W}_K$, $\mathbf{W}_V$ into GPU memory, each with dimensions $[h_1 \times h_1]$. The resulting transient states $\mathbf{Q}$, $\mathbf{K}$, and $\mathbf{V}$ each have dimensions $[b \times s \times h_1]$. Thus, the peak memory usage is $4bsh_1 + 3h_1^2$. After this step, HyperGen retains only $\mathbf{X}$, $\mathbf{Q}$, $\mathbf{K}$, and $\mathbf{V}$ for subsequent steps and releases $\mathbf{W}_Q$, $\mathbf{W}_K$, and $\mathbf{W}_V$, resulting in a memory footprint of $4bsh_1$. This retained memory serves as the starting point for estimating the peak memory usage in the next step (i.e., attention score calculation).

Computation scheduling. HyperGen implements fine-grained computation scheduling for the prefill stage. It focuses on the three memory-intensive steps identified in §2.2: attention score calculation, softmax, and attention output generation. Based on memory usage estimation, in addition to the default case where GPU memory is sufficient to accommodate all steps in the prefill stage, HyperGen classifies the deployment into one of the following memory-pressure scenarios and schedules computations accordingly: (i) *moderate pressure*, where only softmax exceeds GPU memory limits, and (ii) *severe pressure*, where all three memory-intensive steps exceed GPU memory limits.

(i) Moderate pressure. Under moderate pressure, all steps, except softmax, fit within GPU memory. Based on our memory usage estimation, the softmax input fits in memory, but the output may not. HyperGen addresses this by partitioning the softmax input matrix into independent groups, such that the output of each group fits within GPU memory. As shown in Figure 3(a), HyperGen processes each group, updates the corresponding portion of the input matrix with the softmax results, and continues until all groups are processed. This ensures that softmax operates within GPU memory constraints.



(a) Partitioning for moderate pressure



(b) Partitioning for severe pressure

Figure 3: Partitioning for computations.

(ii) Severe pressure. Under severe pressure, QKV calculation fits within GPU memory, but attention score calculation, softmax, and attention output generation exceed memory limits. Since MLP processing has low memory requirements, HyperGen focuses on partitioning the three middle steps and leverages CPU memory for result concatenation, as shown in Figure 3(b).

- *Attention score calculation:* With input matrices residing in GPU memory, HyperGen partitions matrix multiplication into sub-matrix multiplications. Each input matrix is divided into an equal number of sub-matrices that are sized to ensure that the resulting sub-matrix fits within available GPU memory. Once computed, the result is offloaded to CPU memory for concatenation, so that HyperGen proceeds to process the next sub-matrix.
- *Softmax:* As the softmax input now resides in CPU memory, HyperGen partitions it into groups based on available GPU memory, such that both the input and output of each group fit in GPU memory. The output of each group is offloaded to CPU memory for further processing.
- *Attention output generation:* Similar to attention score calculation, HyperGen also partitions matrix multiplication into sub-matrix multiplications. However, unlike the earlier step, the input sub-matrices reside in CPU memory. Thus, HyperGen needs to allocate GPU memory for both the input sub-matrices and the result during each sub-matrix multiplication.

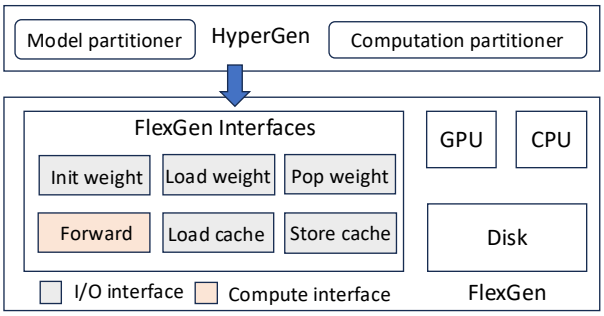
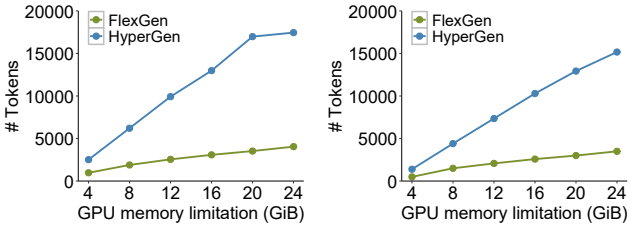


Figure 4: HyperGen's architecture.



(a) OPT-30B (b) OPT-66B

Figure 5: (Exp#1) Maximum prompt length.

3.3 Implementation

HyperGen is implemented as an extension to FlexGen [18] with 325 LoC of changes. It modifies FlexGen's prefill-stage implementation, while retaining FlexGen's decode-stage workflow. Figure 4 shows HyperGen's architecture, which comprises two key modules: a *model partitioner* and a *computation partitioner*.

Model partitioner. The model partitioner implements size-aware parameter partitioning (§3.1) based on FlexGen's model partitioning strategy. We modify the following interfaces in FlexGen: (i) *init_weight*, modified to distinguish between small and large model parameter fragments during model weight initialization; (ii) *load_weight*, modified to load fragments into GPU memory based on size-aware caching (§3.1); (iii) *pop*, modified to free large fragments of a transformer block from GPU memory.

Computation partitioner. The computation partitioner implements step-aware partitioning for the prefill stage (§3.2). We modify the *forward* interface to execute either the prefill or decode stage within a transformer block. For fine-grained operation partitioning, HyperGen leverages PyTorch's I/O interface to manage transient states: (i) *load_cache*, which transfers key and value transient states from CPU or disk to GPU memory; and (ii) *store_cache*, which offloads transient states to disk.

4 Evaluation

We evaluate HyperGen on a commodity machine with an NVIDIA GeForce RTX 4090 GPU (24 GiB GDDR6X), a 48-core Intel Xeon Silver 4310 CPU (2.1 GHz), and 256 GiB DDR4 memory. The machine runs Linux kernel version 5.15.0 with CUDA 11.8 and PyTorch 2.4.1. We compare HyperGen against two variants of FlexGen [18]: the default FlexGen, which performs the prefill stage in GPU, and FlexGen-CPU, which performs the prefill stage in CPU.

By default, we set the batch size $b = 4$, the number of output tokens as 32 (consistent with FlexGen), and the input prompt length $s = 512$ tokens. We set the available GPU memory as 4 GiB to simulate a resource-constrained scenario. We consider two models: (i) OPT-30B, with the hidden dimensions $h_1 = 7168$ and $h_2 = 28672$, and the number of attention heads $n = 56$; (ii) OPT-66B, with $h_1 = 9216$, $h_2 = 36864$, and $n = 72$. Our results are averaged over five runs.

Exp#1 (Maximum prompt length). We examine HyperGen's ability to handle long prompts by measuring the maximum prompt length supported by HyperGen and FlexGen across various GPU memory constraints, varied from 4 GiB to 24 GiB. We exclude FlexGen-CPU from this evaluation as it has extremely long runtime for long prompts. Figure 5 shows the results. HyperGen consistently supports significantly longer prompts than FlexGen. It achieves up to 3.8× and 3.3× improvements in maximum input prompt length for OPT-30B and OPT-66B, respectively. In most cases, the maximum prompt length scales linearly with GPU memory capacity, aligning with our memory usage estimates (Table 1). An exception is that HyperGen's maximum prompt length shows only a sub-linear increase at 24 GiB for OPT-30B, where excessively long prompts cause CPU concatenation of sub-operation results to become inefficient.

Exp#2 (Latency analysis). We measure time-to-first-token (TTFT) for HyperGen, FlexGen, and FlexGen-CPU across various prompt lengths of 256, 512, and 1,024 tokens. Due to FlexGen-CPU's prohibitively long runtime for 1,024 tokens, we exclude it from evaluation at this length. Table 2 shows the results. As the prompt length grows, FlexGen encounters out-of-memory errors, while FlexGen-CPU, although capable of processing long prompts, suffers from extremely high TTFT (e.g., 6.67 hours for OPT-30B with 256 tokens). HyperGen supports long-prompt inference and achieves the shortest TTFT. For example, HyperGen reduces TTFT by 40.1% for OPT-66B with 256 input tokens compared to FlexGen, mainly because HyperGen's size-aware caching strategy enables efficient memory access during transformer block execution.

We further evaluate TTFT for HyperGen and FlexGen with long prompts (1,000 to 4,000 tokens) under a 24 GiB GPU memory constraint. We exclude FlexGen-CPU due to its prohibitively long runtime. Table 3 shows the results. HyperGen demonstrates two key advantages: (i) when the prefill stage

#Tokens	FlexGen	FlexGen-CPU	HyperGen
256	7.80 s	6.67 h	7.06 s
512	7.59 s	13.53 h	7.09 s
1024	OOM	-	32.44 s

(a) OPT-30B

#Tokens	FlexGen	FlexGen-CPU	HyperGen
256	25.03 s	14.70 h	14.60 s
512	OOM	30.08 h	66.49 s
1024	OOM	-	107.29 s

(b) OPT-66B

Table 2: (Exp#2) TTFT under 4 GiB GPU memory (OOM = out-of-memory).

#Tokens	OPT-30B		OPT-66B	
	FlexGen	HyperGen	FlexGen	HyperGen
1000	9.71 s	7.39 s	23.43 s	15.50 s
2000	9.77 s	7.57 s	23.85 s	15.09 s
3000	9.87 s	7.93 s	24.14 s	16.24 s
4000	10.64 s	9.21 s	OOM	68.37 s

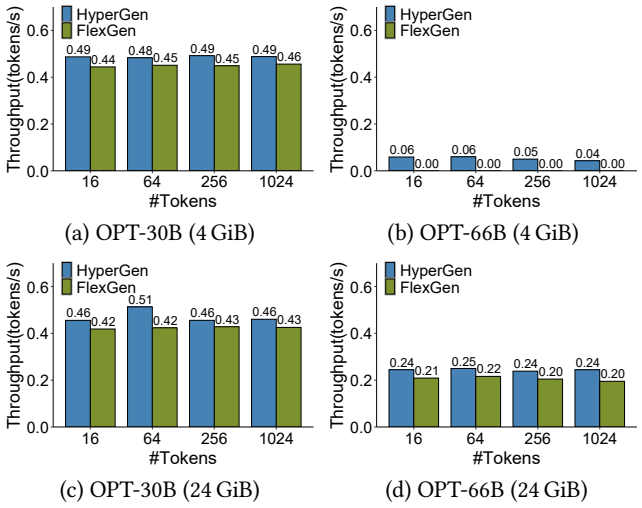
Table 3: (Exp#2) TTFT under 24 GiB GPU memory (OOM = out-of-memory).

fits within GPU memory, HyperGen achieves up to 36.7% lower TTFT than FlexGen for OPT-66B due to its size-aware caching optimization; and (ii) for prompts exceeding GPU memory capacity (e.g., 4,000 tokens), HyperGen successfully generates the first token in 68.37 s, while FlexGen fails to complete inference.

Exp#3 (Throughput analysis). We measure inference throughput for HyperGen and FlexGen across output lengths, varied from 16 to 1,024 tokens (FlexGen-CPU is excluded due to its long runtime). We configure the input prompts with 512 tokens for 4 GiB GPU memory and 1,024 tokens for 24 GiB GPU. For 4 GiB GPU memory, HyperGen achieves up to 9.7% higher throughput than FlexGen for OPT-30B with 16 output tokens using size-aware caching (Figure 6(a)). FlexGen fails to complete inference for OPT-66B (Figure 6(b)), while HyperGen enables inference. For 24 GiB GPU memory, HyperGen increases throughput by 21.1% for OPT-30B with 64 output tokens (Figure 6(c)), while for OPT-66B, its throughput gain increases by up to 25.2% with 1,024 output tokens (Figure 6(d)).

5 Conclusion and Future Work

HyperGen enhances generative inference for long prompts in resource-constrained systems. It optimizes the prefill stage with fine-grained partitioning techniques, including size-aware parameter partitioning and step-aware computation partitioning. Experiments show that HyperGen supports long input prompts, reduces TTFT, and increases inference throughput in a resource-constrained setting.



(a) OPT-30B (4 GiB) (b) OPT-66B (4 GiB)

(c) OPT-30B (24 GiB) (d) OPT-66B (24 GiB)

Figure 6: (Exp#3) Throughput versus number of output tokens.

We explore the following issues in future work. First, HyperGen is currently designed for a single consumer-grade GPU. One research direction is to extend HyperGen for multiple GPUs through new partitioning techniques. Second, real-world edge environments often feature heterogeneous hardware configurations over a networked environment. We plan to extend HyperGen to support long-prompt generative inference in edge-based networked scenarios. Third, the current implementation of HyperGen is based on full attention. We plan to explore HyperGen's compatibility with sliding window and KV-Cache selection approaches. Finally, we plan to extend our evaluation for various model architectures, such as Llama [13] and QWen [23].

Acknowledgments

This work was supported by the National Natural Science Foundation of China (No. 62302175). The corresponding author is Xiaolu Li.

References

[1] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[2] Renze Chen, Zhuofeng Wang, Beiquan Cao, Tong Wu, Size Zheng, Xiuhong Li, Xuechao Wei, Shengen Yan, Meng Li, and Yun Liang. ArkVale: Efficient Generative LLM Inference with Recallable Key-Value Eviction. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang, editors, *Advances in Neural Information Processing Systems*, volume 37, pages 113134–113155. Curran Associates, Inc., 2024.

[3] Weijian Chen, Shuibing He, Haoyang Qu, Ruidong Zhang, Siling Yang, Ping Chen, Yi Zheng, Baoxing Huai, and Gang Chen. IMPRESS: An Importance-Informed Multi-Tier Prefix KV Storage System for Large Language Model Inference. In *23rd USENIX Conference on File and*

Storage Technologies (FAST 25), pages 187–201, 2025.

[4] Yaoqi Chen, Jinkai Zhang, Baotong Lu, Qianxi Zhang, Chengruidong Zhang, Jingjia Luo, Di Liu, Huiqiang Jiang, Qi Chen, Jing Liu, et al. RetroInfer: A Vector-Storage Approach for Scalable Long-Context LLM Inference. *arXiv preprint arXiv:2505.02922*, 2025.

[5] Yuxuan Chen, Rongpeng Li, Zhifeng Zhao, Chenghui Peng, Jianjun Wu, Ekram Hossain, and Honggang Zhang. NetGPT: A native-AI network architecture beyond provisioning personalized generative services. *arXiv preprint arXiv:2307.06148*, 2023.

[6] Zhuoming Chen, Ranajoy Sadhukhan, Zihao Ye, Yang Zhou, Jianyu Zhang, Niklas Nolte, Yuandong Tian, Matthijs Douze, Leon Bottou, Zhihao Jia, and Beidi Chen. Magic PIG: LSH Sampling for Efficient LLM Generation. In *The Thirteenth International Conference on Learning Representations*, 2025.

[7] Yinglong Dai and Guojun Wang. A deep inference learning framework for healthcare. *Pattern Recognition Letters*, 139:17–25, 2020.

[8] Qichen Fu, Minsik Cho, Thomas Merth, Sachin Mehta, Mohammad Rastegari, and Mahyar Najibi. LazyLLM: Dynamic token pruning for efficient long context LLM inference. In *Workshop on Efficient Systems for Foundation Models II @ ICML2024*, 2024.

[9] Jitai Hao, Yuke Zhu, Tian Wang, Jun Yu, Xin Xin, Bo Zheng, Zhaochun Ren, and Sheng Guo. OmniKV: Dynamic context selection for efficient long-context LLMs. In *The Thirteenth International Conference on Learning Representations*, 2025.

[10] Sagar Imambi, Kolla Bhanu Prakash, and G. R. Kanagachidambaresan. *PyTorch*, pages 87–104. Springer International Publishing, Cham, 2021.

[11] Wonbeom Lee, Jungi Lee, Junghwan Seo, and Jaewoong Sim. InfiniGen: Efficient Generative Inference of Large Language Models with Dynamic KV Cache Management. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 155–172, Santa Clara, CA, July 2024. USENIX Association.

[12] Cheng Luo, Zefan Cai, Hanshi Sun, Jinqi Xiao, Bo Yuan, Wen Xiao, Junjie Hu, Jiawei Zhao, Beidi Chen, and Anima Anandkumar. HeadInfer: Memory-Efficient LLM Inference by Head-wise Offloading. In *ICML 2025 Workshop on Long-Context Foundation Models*, 2025.

[13] Meta. Llama. https://www.llama.com/, 2025.

[14] Khadijeh Moulaei, Atiye Yadegari, Mahdi Baharestani, Shayan Farzanbakhsh, Babak Sabet, and Mohammad Reza Afrash. Generative artificial intelligence in healthcare: A scoping review on benefits, challenges and applications. *International Journal of Medical Informatics*, 188:105474, 2024.

[15] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. CodeGen: An open large language model for code with multi-turn program synthesis. In *The Eleventh International Conference on Learning Representations*, 2023.

[16] Tiernan Ray. OpenAI's gigantic GPT-3 hints at the limits of language models for AI. =https://www.zdnet.com/article/openais-gigantic-gpt-3-hints-at-the-limits-of-language-models-for-ai/, 2020. Accessed: 2025-05-23.

[17] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. Code Llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

[18] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Re, Ion Stoica, and Ce Zhang. FlexGen: High-Throughput generative inference of large language models with a single GPU. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett, editors, *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 31094–31116. PMLR, 23–29 Jul 2023.

[19] Yixin Song, Zeyu Mi, Haotong Xie, and Haibo Chen. PowerInfer: Fast large language model serving with a consumer-grade gpu. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, SOSP '24, pages 590–606, New York, NY, USA, 2024. Association for Computing Machinery.

[20] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[21] Yiming Wang, Yu Lin, Xiaodong Zeng, and Guannan Zhang. Privatelora for efficient privacy preserving LLM. *arXiv preprint arXiv:2311.14030*, 2023.

[22] Guangxuan Xiao, Ji Lin, and Song Han. Offsite-tuning: Transfer learning without full model. *arXiv preprint arXiv:2302.04870*, 2023.

[23] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

[24] Chengye Yu, Tianyu Wang, Zili Shao, Linjie Zhu, Xu Zhou, and Song Jiang. TwinPilots: A new computing paradigm for GPU-CPU parallel LLM inference. In *Proceedings of the 17th ACM International Systems and Storage Conference*, SYSTOR '24, pages 91–103, New York, NY, USA, 2024. Association for Computing Machinery.

[25] Philipp Zagar, Vishnu Ravi, Lauren Aalami, Stephan Krusche, Oliver Aalami, and Paul Schmiedmayer. Dynamic fog computing for enhanced LLM execution in medical applications. *Smart Health*, 36:100577, 2025.

[26] Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett, Zhangyang "Atlas" Wang, and Beidi Chen. H2o: Heavy-Hitter oracle for efficient generative inference of large language models. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, editors, *Advances in Neural Information Processing Systems*, volume 36, pages 34661–34710. Curran Associates, Inc., 2023.

[27] Zhi Zhou, Xu Chen, En Li, Liekang Zeng, Ke Luo, and Junshan Zhang. Edge intelligence: Paving the last mile of artificial intelligence with edge computing. *Proceedings of the IEEE*, 107(8):1738–1762, 2019.

[28] Xiangwen Zhuge, Xu Shen, Zeyu Wang, Fan Dang, Xuan Ding, Danyang Li, Yahui Han, Tianxiang Hao, and Zheng Yang. SpecOffload: Unlocking Latent GPU Capacity for LLM Inference on Resource-Constrained Devices. *arXiv preprint arXiv:2505.10259*, 2025.