FlexGen: High-Throughput Generative Inference of Large Language Models with a Single GPU

Ying Sheng[1] Lianmin Zheng[2] Binhang Yuan[3] Zhuohan Li[2] Max Ryabinin[4,5]
Daniel Y. Fu[1] Zhiqiang Xie[1] Beidi Chen[6,7] Clark Barrett[1]
Joseph E. Gonzalez[2] Percy Liang[1] Christopher Reé[1] Ion Stoica[2] Ce Zhang[3]

[1]Stanford University [2]UC Berkeley [3]ETH Zurich
[4]Yandex [5]HSE University
[6]Meta [7]Carnegie Mellon University

Presenter: Shiwei Zhang

Content

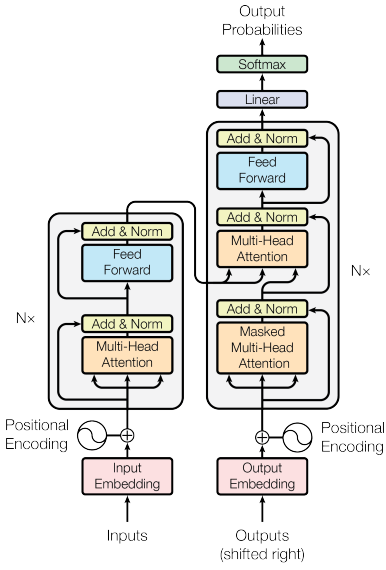
- Introduction
- Background: LLM Inference
- Offloading Strategy
- Approximate Methods
- Evaluation
- Conclusion

Introduction

Large Language Models

Most of the state-of-the-art large language models use the decoder-only Transformer architecture.

They take a sequence of tokens as input and produce the probability distribution of the next token.



Interactive vs. Throughput-oriented Inference

Interactive applications

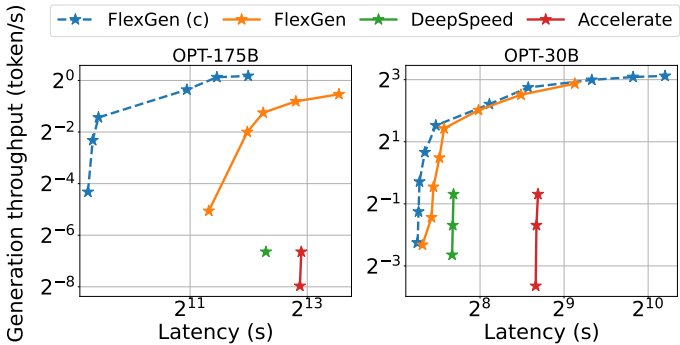
- Chat Bot
- Customer Support
- Search

"Back-of-house" tasks are less sensitive to latency.

- Information Extraction
- Data Wrangling
- Form Processing

It is possible to trade off latency for higher throughput in some workloads, providing opportunities to reduce resource requirements.

Latency-Throughput Trade-off

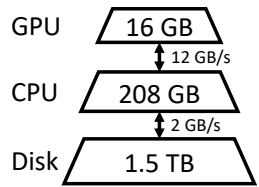


Current State

- FasterTransformer, Orca, LightSeq, PaLM, TurboTransformers, DeepSpeed Inference, and Hugging Face Accelerate focus on **latency-oriented** scenarios with **high-end accelerators**.
- DeepSpeed Inference and Hugging Face Accelerate supports offloading that is inherited from training systems. They fail to exploit the structure of the throughput-oriented LLM inference computation.
- Petals enable LLM inference on accessible hardware by collaborative computing.

FlexGen

The focus of this paper is designing efficient **offloading strategies** for high-throughput generative inference, on **a single commodity GPU**.



Contributions

- A strategy space considering computation schedule, tensor placement, and computation delegation.
- Compression of both the weights and KV cache without retraining or calibration.
- Experiments of OPT-175B on NVIDIA T4 (16GB) showing 100x/40x higher throughput with/without compression.

Background: LLM Inference

Generative Inference

A typical LLM generative inference task consists of two stages, the *prefill stage* and the *decoding stage*.

During the *prefill phase*, the cached key and value vectors for each transformer layer is calculated.

$$\mathbf{x}_K^i = \mathbf{x}^i \cdot \mathbf{w}_K^i; \quad \mathbf{x}_V^i = \mathbf{x}^i \cdot \mathbf{w}_V^i; \quad \mathbf{x}_Q^i = \mathbf{x}^i \cdot \mathbf{w}_Q^i$$

$$\mathbf{x}_{Out}^i = \mathrm{Softmax}(\frac{\mathbf{x}_Q^i {\mathbf{x}_K^i}^T}{\sqrt{h}}) \cdot \mathbf{x}_V^i \cdot \mathbf{w}_O^i + \mathbf{x}^i$$

$$\mathbf{x}^{i+1} = \mathrm{relu}(\mathbf{x}_{Out}^i \cdot \mathbf{w}_1) \cdot \mathbf{w}_2 + \mathbf{x}_{Out}^i$$

Generative Inference (Cont'd)

During the *decoding phase*, given t^i as the embedding of the current generated token in the i-th layer:

$$\mathbf{x}_K^i \leftarrow \text{Concat}(\mathbf{x}_K^i), \mathbf{t}^i \cdot \mathbf{w}_K^i; \quad \mathbf{x}_V^i \leftarrow \text{Concat}(\mathbf{x}_V^i), \mathbf{t}^i \cdot \mathbf{w}_V^i; \quad \mathbf{t}_Q^i = \mathbf{t}^i \cdot \mathbf{w}_Q^i$$

$$\mathbf{t}_{Out}^i = \text{Softmax}(\frac{\mathbf{t}_Q^i {\mathbf{x}_K^i}^T}{\sqrt{h}}) \cdot \mathbf{x}_V^i \cdot \mathbf{w}_O^i + \mathbf{t}^i$$

$$\mathbf{t}^{i+1} = \text{relu}(\mathbf{t}_{Out}^i \cdot \mathbf{w}_1) \cdot \mathbf{w}_2 + \mathbf{t}_{Out}^i$$

Memory Analysis

Denote the batch size by b, the input sequence length by s, the output sequence length by n, the hidden dimensions of the Transformer and MLP layers by h_1 and h_2, and the number of layers by l.

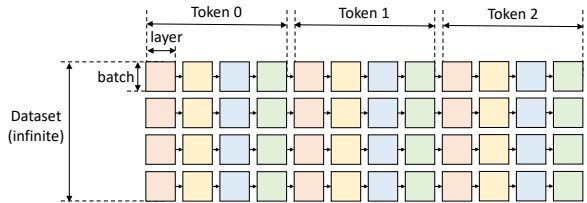
The model weights is roughly (ignoring the embedding layer) $l(8h_1^2 + 4h_1h_2)$ bytes and the peak KV cache is $4blh_1(s+n)$.

The OPT-175B ($l = 96$, $h_1 = 12288$, $h_2 = 49152$) model takes 325 GB. With $b = 512$, $s = 512$, and $n = 32$, the KV cache is 1.2 TB, which is 3.8$\times$ the model weights.

Offloading Strategy

Problem Formulation

The generative inference with offloading is formulated as a graph traversal problem. In the figure, a square means the computation of a GPU batch for a layer. The squares with the same color share the same layer weights.



Problem Formulation (Cont'd)

A valid path is a path that traverses all squares under the following constraints:

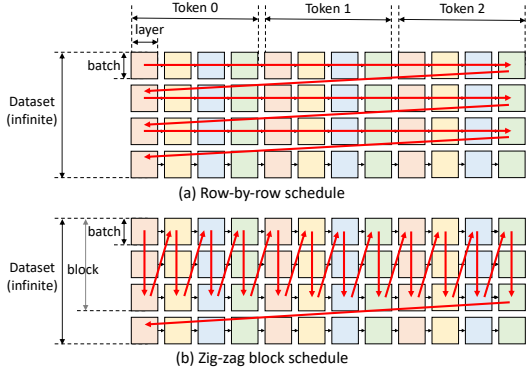
- A square can only be computed if all squares to its left on the same row were computed.
- To compute a square on a device, all its inputs must be loaded to the same device.
- After being computed, a square produces two outputs: activations and KV cache. The activations should be stored until its right sibling is computed. The KV cache should be stored until the rightmost square on the same row is computed.
- At any time, the total size of tensors stored on a device cannot exceed its memory capacity.

Objective

The goal is to find a valid path that minimizes the total execution time, which includes the compute cost and I/O cost when moving tensors between devices.

Compute Schedule

All existing systems traverse the graph row-by-row to reduce latency.



The zig-zag block schedule reduces I/O costs by reusing the weights and KV cache for multiple batches. It introduces two parameters into the search space: the GPU batch size and the number of GPU batches in a block.

Compute Schedule with Overlapping

Another typical optimization is overlapping the weights load of the next layer, cache/activation load of the next batch, cache/activation store of the previous batch, and the computation of the current batch.

Algorithm 1 Block Schedule with Overlapping

for $i = 1$ **to** $generation\_length$ **do**
  for $j = 1$ **to** num_layers **do**
    // Compute a block with multiple GPU batches
    for $k = 1$ **to** $num\_GPU\_batches$ **do**
      // Load the weight of the next layer
      `load_weight`$(i, j + 1, k)$
      // Store the cache and activation of the prev batch
      `store_activation`$(i, j, k - 1)$
      `store_cache`$(i, j, k - 1)$
      // Load the cache and activation of the next batch
      `load_cache`$(i, j, k + 1)$
      `load_activation`$(i, j, k + 1)$
      // Compute this batch
      `compute`(i, j, k)
      // Synchronize all devices
      `synchronize`$()$
    end for
  end for
end for

Tensor Placement

FlexGen uses variables wg, wc, and wd to define the percentages of **weights** stored on GPU, CPU, and disk, hg, hc, and hd to define the percentages of **activations**, and cg, cc, cd for the **KV cache**.

FlexGen uses layer granularity (e.g., assign 50% of the tensors in a layer to the GPU) for weights, and tensor granularity (e.g., assign 50% of the elements in a tensor to the GPU) for activations and the KV cache.

Computation Delegation

Using CPU for computation can be beneficial when the computation is I/O-bounded. Take the computation of attention scores $\mathrm{Softmax}(\frac{\mathbf{t}_Q^i {\mathbf{x}_K^i}^T}{\sqrt{h}})$ as example: the size of the moved KV cache is $b \times s \times h_1 \times 4$ bytes, and the size of the moved activation is $b \times h_1 \times 4$. For long sequences (e.g., $s \geq 512$), it is better to compute the attention scores on the CPU if the associated KV cache is not stored on the GPU.

Cost Model

The total latency for computing a block can be estimated as

$$T = T_{\text{pre}} \cdot l + T_{\text{gen}} \cdot (n-1) \cdot l$$

where T_{pre} and T_{gen} are the estimated latencies of the prefill stage and the decoding stage for one layer.

Cost Model (Cont'd)

Assuming perfect overlapping, T_{pre} can be esitmated as

$$T_{\text{pre}} = \max(ctog^p, gtoc^p, dtoc^p, ctod^p, comp^p)$$

where $ctog^p$, $gtoc^p$, $dtoc^p$, $ctod^p$, and $comp^p$ denote the latency of read from CPU to GPU, write from GPU to CPU, read from disk to CPU, write from CPU to disk, and computation, respectively, during prefill for one layer.

Similarly, T_{gen} can be esitmated as

$$T_{\text{gen}} = \max(ctog^g, gtoc^g, dtoc^g, ctod^g, comp^g)$$

Cost Model Example

I/O terms like $dtoc^g$ are estimated by summing up the I/O events, which contain weights, activations, and cache reads.

The size of FP16 weights for one Transformer layer is $8h_1^2 + 4h_1 \cdot h_2$ bytes. Let bls denote the block size and s be the prompt length. The size of the activation for one layer is $2 \cdot bls \cdot h_1$ and the size of KV cache for one layer on average is $4 \cdot bls \cdot (s + \frac{n}{2}) \cdot h_1$.

Since wd, hd, and cd percent of weights, activations, and KV cache are load from disk, the total latency of disk read is

$$dtoc^g = \frac{1}{\text{bandwidth}}((8h_1^2 + 4h_1 \cdot h_2) \cdot wd + 2 \cdot bls \cdot h_1 \cdot hd + 4 \cdot bls \cdot (s + \frac{n}{2}) \cdot h_1 \cdot cd)$$

Policy Search

A policy includes 11 variables: block size bls, GPU batch size gbs, and 9 percentages for tensor placement.

FlexGen first enumerate a few choices of (bls, gbs) tuple. With fixed bls, gbs, the best placement becomes a linear programming problem.

$$
\begin{aligned}
\min_{p} \quad & T/bls \\
\text{s.t.} \quad & gpu\ peak\ memory < gpu\ mem\ capacity \\
& cpu\ peak\ memory < cpu\ mem\ capacity \\
& disk\ peak\ memory < disk\ mem\ capacity \\
& wg + wc + wd = 1 \\
& cg + cc + cd = 1 \\
& hg + hc + hd = 1
\end{aligned}
$$

Extension to Multiple GPUs

Tensor parallelism can reduce the single-query latency but pipeline parallelism can achieve good scaling on throughput due to its low communication costs.

FlexGen implements pipeline parallelism by equally partitioning an l-layer LLM on m GPUs.

Approximate Methods

Group-wise Quantization

FlexGen direclty quantize both the weights and KV cache into 4-bit integers without any retraining or calibration.

Given a tensor, FlexGen choose g continous elements along a certain dimension as a group. For each group, the min and max are calculated and each element x is quantized as

$$x_{\text{quant}} = \text{round}(\frac{x - min}{max - min} \times (2^b - 1))$$

Group-wise Quantization (Cont'd)

FlexGen uses 4 bits quantization with a group size of 64. The weights are grouped along the output channel dimension and the KV cache are grouped along the hidden dimension.

Fine-grained group-wise quantization in FlexGen causes some overhead in compression and decompression. Such an overhead could be very significant if run on a CPU which makes the CPU delegation useless, so FlexGen turns off the CPU delegation when enabling quantization.

Sparse Attention

After computing the attention matrices, for each query, FlexGen calculates the indices of the Top-K tokens from the K cache, then simply drops other tokens and only loads the subset of the V cache according to the indices.

Experiments

Experimental Setup

Hardware:

Device	Model	Memory
GPU	NVIDIA T4	16 GB
CPU	Intel Xeon @ 2.00GHz	208 GB
Disk	Cloud default SSD (NVMe)	1.5 TB

Models: OPT models with 6.7B to 175B parameters.

Workloads: Synthetic datasets with all prompts padded to the 512/1024 tokens. The system is required to generate 32 tokens for each prompt.

Implementation: FlexGen is implemented on top of PyTorch. FlexGen manages multiple CUDA streams and CPU threads to overlap I/O with compute. FlexGen creates files for tensors stored on the disk and maps them as virtual memory to access them.

Baselines

DeepSpeed Zero-Inference supports offloading the whole weights to CPU or disk. It uses ZeRO data parallelism when given multiple GPUs.

Hugging Face Accelerate supports offloading a fraction of the weights.

Petals lowers the resource requirements for LLM inference with decentralized collaborative inference.

Maximum Throughput Benchmark

For OPT-175B, baseline systems can only use a GPU batch size of 2, but FlexGen can use a GPU batch size of 32 and a block size of 32 $\times$ 8, achieving a 69$\times$ higher throughput

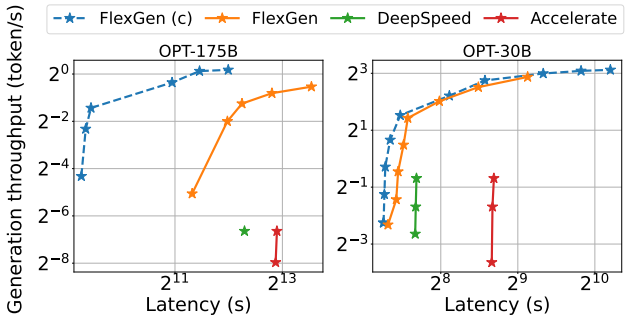
Seq. length	512			1024		
Model size	6.7B	30B	175B	6.7B	30B	175B
Accelerate	25.12	0.62	0.01	13.01	0.31	0.01
DeepSpeed	9.28	0.60	0.01	4.59	0.29	OOM
Petals	8.25	2.84	0.08	6.56	1.51	0.06
FlexGen	25.26	7.32	0.69	13.72	3.50	0.35
FlexGen (c)	29.12	8.70	1.12	13.18	3.98	0.42

Maximum Throughput Benchmark with Multiple GPUs

FlexGen achieves super-linear scaling on decoding throughput with pipeline parallelism.

Metric	Generation Throughput			Decoding Throughput		
Model size	6.7B	30B	175B	6.7B	30B	175B
FlexGen (1)	25.26	7.32	0.69	38.28	11.52	0.83
FlexGen (4)	201.12	23.61	2.33	764.65	48.94	3.86
DeepSpeed (4)	50.00	6.40	0.05	50.20	6.40	0.05

Latency-Throughput Trade-off



Runtime Breakdown

Stage	Total	Compute	Weight (R)	Cache (R)	Cache (W)
Prefill	2711	2220	768	0	261
Decoding	11315	1498	3047	7046	124

Ablation Study

The numbers are generation throughput on 1 GPU with prompt length 512. The gray tuple denotes a policy (GPU batch size $\times$ #GPU-batch, wg, wc).

Model size	30B	175B
All optimizations	7.32 (48×3, 20, 80)	0.69 (32×8, 0, 50)
No policy search	7.26 (48×3, 0, 100)	0.27 (32×1, 0, 50)
No overlapping	5.86	0.59
No CPU compute	4.03	0.62
No disk	7.32	OOM
w/ DeepSpeed policy	1.57	0.01

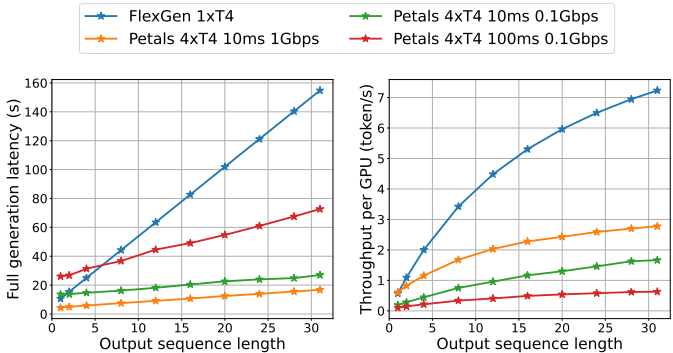
Approximations

4-bit means using group-wise quantization to compress both weights and KV cache into 4-bit integers. 4-bit-S means combining the quantization and sparse attention with a 10% sparsity on the value cache. Both methods show negligible accuracy loss compared to FP16.

Dataset	Lambada (acc)			WikiText (ppl)		
Config	FP16	4-bit	4-bit-S	FP16	4-bit	4-bit-S
OPT-30B	0.725	0.724	0.718	12.72	12.90	12.90
OPT-175B	0.758	0.756	0.756	10.82	10.94	10.94

Offloading vs. Collaborative Inference

The throughput of FlexGen with a single T4 outperforms the per-GPU throughput of the Petals cluster (4 nodes on GCP with one T4 GPU per node) under all tested network conditions.



Summary

Strength

- A new and important problem.
- In-depth analysis of the problem and locating the bottleneck with experiments.
- A lot of experiments (6 pages in the appendix).

Limitation

- The approximate methods are not novel and are not strongly related to other designs.
- The linear cost model does not reflect the fact that larger batch sizes bring better GPU utilization.
- The strategy space is not very complete and many of the decisions (e.g., CPU delegation) are manual.

Takeaways

- New scenarios (throughput-oriented LLM inference) brings new challenges to well-studied problems (offloading).
- It is easier to run experiments for resource-constraint systems.
- Search for parameters of well-designed heurstics instead of every possible solutions.
  - It better illustrates the benefits instead of being pure "black-box"
  - It may help maintain good performance with inaccurate profile data and imperfect cost models.

Thank you!