

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 Re' 1 Ion Stoica 2 Ce
Zhang

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

Motivation 1: Making Large Models Accessible



To



Motivation 2: Throughput-Oriented LLM Use Cases



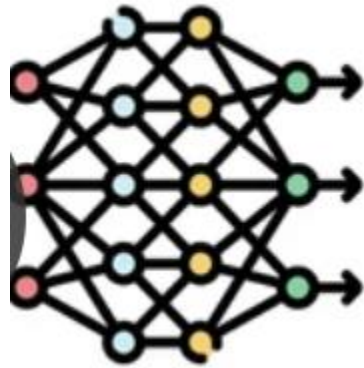
Latency-sensitive Tasks
(e.g., Chatbot)

To



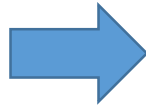
Throughput-Oriented Tasks
(e.g., benchmarking,
information extraction, data
wrangling, Text/forms)

How to efficiently run LLMs with limited resources



GPT-175B

Parameter size: 325 GB



NVIDIA T4

Memory capacity: 16 GB

Results: Latency-throughput Trade-off

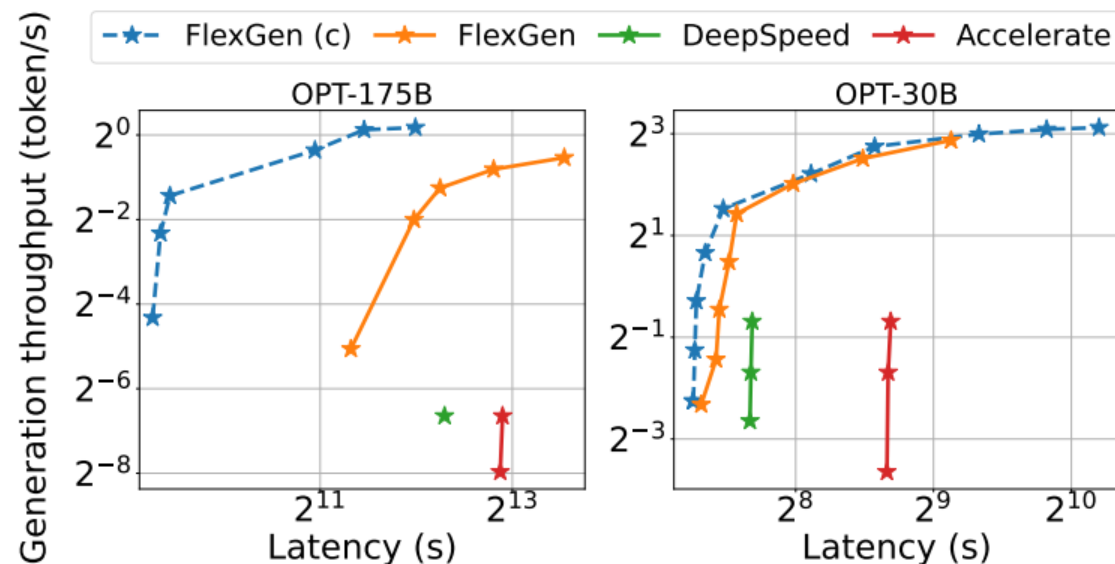


Figure 1. The total latency for a block and throughput trade-offs of three offloading-based systems for OPT-175B (left) and OPT-30B (right) on a single NVIDIA T4 (16 GB) GPU with 208 GB CPU DRAM and 1.5TB SSD. FlexGen achieves a new Pareto-optimal frontier with $100\times$ higher maximum throughput for OPT-175B. Other systems cannot further increase throughput due to out-of-memory issues. “(c)” denotes compression.

Workload:

- Running OPT-175B on a T4 GPU

Existing systems:

- Throughput of 0.01 token/s
- GPU utilization < 1%
- Bound by a very small batch size (≤ 2)

FlexGen:

- 69x higher throughput
- 128x larger batch size
- Also achieving lower latency

OPT: Open Pre-trained Transformer Language Model

Related work

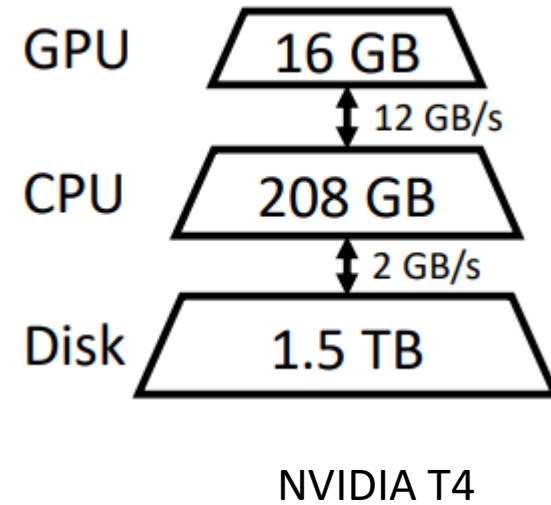
- Prior efforts to lower resource requirements of LLM inference
 - Model compression to decrease total memory footprint
 - Collaborative inference to amortize inference cost via decentralization
 - Offloading to utilize memory from CPU and disk
- Shortcomings:
 - Research in the first two directions often assume that the model fits into the GPU memory
 - The third category do not achieve acceptable throughput on a single GPU due to inefficient I/O scheduling and tensor placement

Goal

Designing efficient offloading strategies for high-throughput generative inference, on a single commodity GPU

Method

Offload LLM to secondary storage and perform computation part-by-part by partially loading it.



Challenge 1: efficient offloading strategy

- There are three kinds of tensors: weights, activations, and key-value (KV) cache.
- Strategy specifies
 - what tensors to offload
 - where to offload them within the three-level memory hierarchy
 - when to offload them during inference
- The batch-by-batch, token-by-token, and layer-by-layer structure of the computation forms a complex dependency graph

Challenge 2: effective compression strategies

- When combining compression with offloading for high-throughput inference, the I/O costs and memory reduction of the weights and KV cache become more important, motivating alternative compression schemes

FlexGen: Proposed offloading framework

- FlexGen aggregates memory from the GPU, CPU, and disk, and efficiently schedules I/O operations, along with possible compression methods and distributed pipeline parallelism
 - Contribution 1:
 - Formally define a search space of possible offloading strategies by considering computation schedule, tensor placement, and computation delegation
 - Search space captures a computation order with I/O complexity within 2× of optimality
 - A linear programming-based search algorithm
 - Contribution 2:
 - Fine-grained groupwise quantization (Shen et al., 2020), which is suitable for reducing I/O costs and memory usage during offloading
 - Contribution 3:
 - Demonstrate the efficiency of FlexGen by running OPT-175B on NVIDIA T4 (16GB) GPUs

FlexGen: Proposed offloading framework (cont.)

- Two stages:
 - i) the prefill stage which takes a prompt sequence to generate the key-value cache (KV cache) for each transformer layer of the LLM;
 - ii) the decoding stage which utilizes and updates the KV cache to generate tokens step-by-step
- Considering FP16, the total number of bytes to store the parameters can be roughly 1 calculated by $l(8h_1^2 + 4h_1h_2)$. The total number of bytes to store the KV cache in peak is $4 \times blh_1(s + n)$.
- Notations:
 - b: batch size by b
 - s: the input sequence length
 - n: the output sequence length
 - h1: the hidden dimension of the transformer
 - h2: the hidden dimension of the second MLP layer
 - l: the total number of transformer layers

FlexGen: Proposed offloading framework (cont.)

- OPT-175B model ($l = 96$, $h_1 = 12288$, $h_2 = 49152$) takes 325 GB. With a batch size of $b = 512$, an input sequence length $s = 512$, and an output sequence length of $n = 32$, the total memory required to store the KV cache is 1.2 TB, which is $3.8\times$ the model weights
- Considering an effective batch size b , an input sequence length s , and an output sequence length of n ,
 - Latency t is defined as the total number of seconds spent to process the prompts and generate all the bn tokens
 - The generation throughput is defined as bn/t

FlexGen: Proposed offloading framework (cont.)

- The model has 4 layers and the system generate 3 tokens per prompt
- A square means the computation of a GPU batch for a layer
- The squares with the same color share the same layer weights

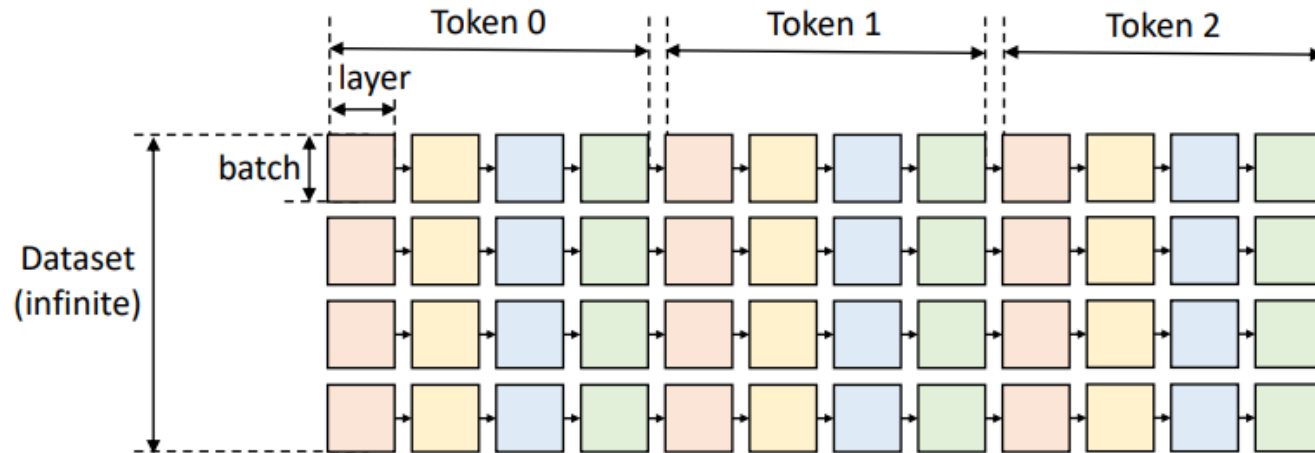


Figure 2. Computational graph of LLM inference.

FlexGen: Proposed offloading framework (cont.)

- Define a valid path as a path that traverses (i.e., computes) all squares, while subject to 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 (weights, activations, cache) 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.
- 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.

FlexGen: Proposed offloading framework (cont.)

- Search space
 - Compute schedule
 - Tensor placement
 - Computation delegation
- Cost Model and Policy Search
- Extension to Multiple GPUs

FlexGen: Proposed offloading framework (cont.)

- All existing systems traverse the graph row-by-row
- Because every two contiguous squares do not share weights, this schedule has to repeatedly load the weights and incurs huge I/O costs

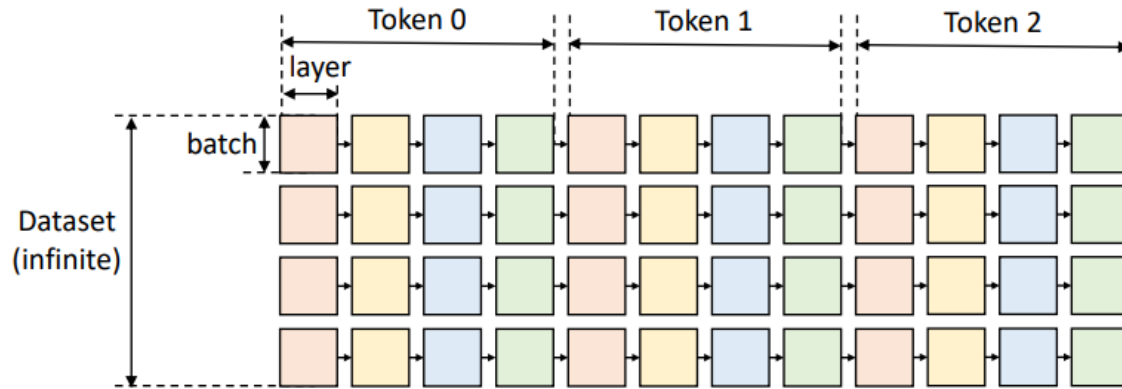
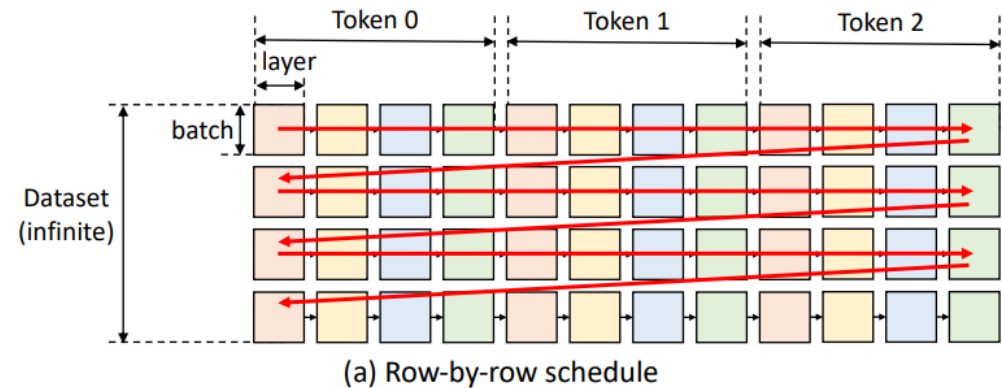


Figure 2. Computational graph of LLM inference.



FlexGen: Proposed offloading framework (cont.) -- Search space

- To reduce the I/O costs of the weights, we can traverse the graph column-by-column.
- All squares in a column share weights, so we can let the weights stay on GPU for reusing and only load/unload the activations and KV cache.
- Stop when they fill the CPU and disk memory

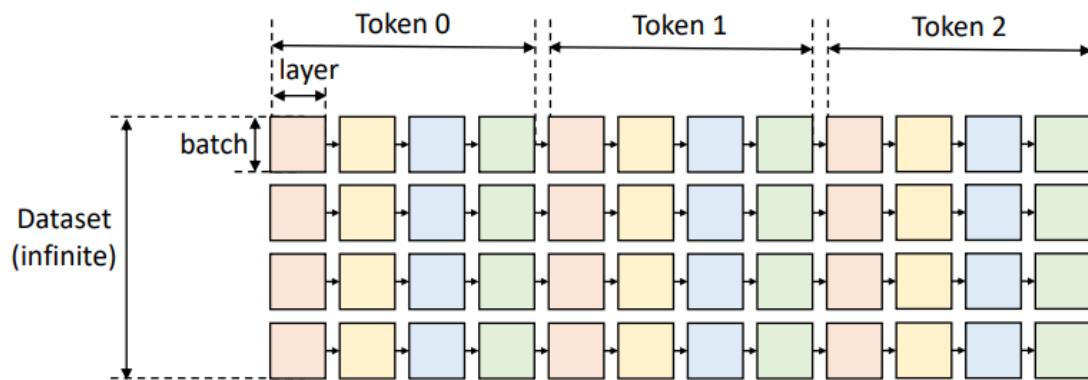
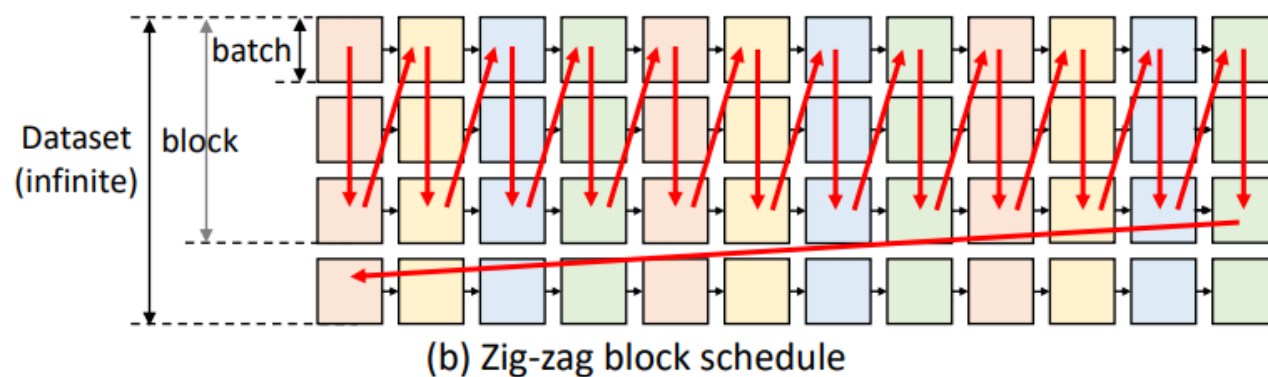


Figure 2. Computational graph of LLM inference.



FlexGen: Proposed offloading framework (cont.) -- Search space

Theorem 4.1. The I/O complexity of the zig-zag block schedule is within $2\times$ of the optimal solution.

FlexGen: Proposed offloading framework (cont.) -- Search space

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
```

- 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
 - The computation of the current batch.

FlexGen: Proposed offloading framework (cont.) – Tensor placement

- How to store these tensors within the memory hierarchy
 - Three variables wg, wc, and wd to define the percentages of weights stored on GPU, CPU, and disk respectively.
 - Three variables hg, hc, hd to define the percentages of activations
 - Three variables cg, cc, cd for the KV cache
- Partition the weights
 - At the model granularity (e.g., assign 50% of the layers in a model to the GPU)
 - At the layer granularity (e.g., assign 50% of the tensors in a layer to the GPU)
 - At the tensor granularity (e.g., assign 50% of the elements in a tensor to the GPU)
- Coarser granularity leads to lower runtime overhead but it is less flexible and its cost is difficult to analyze
- Use layer granularity for weights, and tensor granularity for activations and the KV cache.

FlexGen: Proposed offloading framework (cont.) – Computation delegation

- Computing the attention scores on the GPU requires moving the entire KV cache to the GPU, which incurs a substantial I/O cost as the KV cache is huge
- Computing the attention score on the CPU does not require moving the KV cache. It only requires moving the activations from the GPU to the CPU.
- 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.

FlexGen: Proposed offloading framework (cont.) – Cost Model and Policy Search

- The cost model predicts the latency during prefill for one layer denoted as T_{pre} , and the averaged latency during decoding for one layer denoted as T_{gen} in one block
- The total latency for computing a block can then be estimated as:
 - $T = T_{pre} \cdot l + T_{gen} \cdot (n - 1)$
- T_{pre} can be estimated as $T_{pre} = \max(ctogp, gtocp, dtocp, ctodp, comp_p)$, where $ctogp$, $gtocp$, $dtocp$, $ctodp$, $comp_p$ denote the latency of read from CPU to GPU, write from GPU to CPU, read from disk to CPU, write from CPU to disk, computation, respectively, during prefill for one layer
- T_{gen} can be estimated as $T_{gen} = \max(ctogg, gtocg, dtocg, ctodg, comp_g)$, with $ctogg$, $gtocg$, $dtocg$, $ctodg$, $comp_g$ denoting the latency of read from CPU to GPU, write from GPU to CPU, read from disk to CPU, write from CPU to disk, computation, respectively, during decoding for one layer

FlexGen: Proposed offloading framework (cont.) – Cost Model and Policy Search

- A policy includes 11 variables: block size bls , GPU batch size gbs , weight placement wg , wc , wd , activation placement hg , hc , hd , and KV cache placement cg , cc , cd .
- Enumerate a few choices of (bls, gbs) tuple. Typically, gbs is a multiple of 4, and bls is less than 20 so there are not too many choices.
- Then with the fixed bls, gbs , finding the best placement $p = (wg, wc, wd, cg, cc, cd, hg, hc, hd)$ becomes a linear programming problem shown in Eq. (1).

FlexGen: Proposed offloading framework (cont.) – Cost Model and Policy Search

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

FlexGen: Proposed offloading framework (cont.) – Extension to Multiple GPUs

- If we are given more GPUs and more CPUs, model parallelism can be utilized to reduce the memory pressure of each GPU
- Two kinds of model parallelisms:
 - Tensor: reduce the single-query latency
 - Pipeline parallelism: achieve good scaling on throughput due to low communication cost
- FlexGen implements pipeline parallelism
- Use pipeline parallelism by equally partitioning an l -layer LLM on m GPUs, and then the execution of all GPUs follows the same pattern

FlexGen: Proposed offloading framework (cont.) – Approximate Methods

- Group-wise Quantization
 - Both the weights and KV cache can be directly quantized into 4-bit integers without any retraining or calibration on OPT-175B, all while preserving similar accuracy
 - Previous quantization methods are for accelerated computation, the goal of quantization in this case is primarily for compression and reducing I/O costs
 - Choose a fine-grained quantization format in favor of a high compression ratio and dequantize the tensors back to FP16 before computation
 - Given a tensor, choose g contiguous elements along a certain dimension as a group. For each group, we compute the min and max of the group elements and quantize each element x into b -bit integers by
$$x_{\text{quant}} = \text{round} \{ (x - \min) / (\max - \min) \times (2^b - 1) \}$$

FlexGen: Proposed offloading framework (cont.) – Sparse Attention

- Sparse attention
 - We demonstrate that the sparsity of self-attention can be exploited by only loading the top 10% attention value cache on OPT-175B
 - Top-K sparse approximation
 - After computing the attention matrices, for each query, calculate the indices of its Top-K tokens from the K cache
 - Simply drop the other tokens and only load a subset of the V cache according to the indices

Evaluation

- Hardware
 - NVIDIA T4 GPU instances from Google Cloud

Table 1. Hardware Specs

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

- The read bandwidth of SSD is about 2GB/s and the write bandwidth is about 1GB/s
- Model
 - OPT models with 6.7B to 175B parameters
 - GPT-3 (Brown et al., 2020), PaLM (Chowdhery et al., 2022), and BLOOM (Scao et al., 2022) share a similar structure

Evaluation

- Workload
 - Synthetic datasets where all prompts are padded to the same length: 512 and 1024
 - The system is required to generate 32 tokens for each prompt
- Baseline
 - DeepSpeed ZeRO-Inference (Aminabadi et al., 2022)
 - Hugging Face Accelerate (HuggingFace, 2022)
 - Both of them use the row-by-row schedule and can only put cache/activations on GPU
 - Petals (Borzunov et al., 2022; Ryabinin et al., 2023)
- Implementation
 - FlexGen is implemented on top of PyTorch (Paszke et al., 2019)
 - 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

Evaluation

- Offloading
 - Maximum throughput benchmark

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

1 GPU

Table 3. The scaling performance on 4 GPUs. The prompt sequence length is 512. The number of GPUs is denoted in the parenthesis. Generation throughput (token/s) counts the time cost of both prefill and decoding while decoding throughput only counts the time cost of decoding assuming prefill is done.

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

4 GPUs

Evaluation

- Offloading
 - Latency-throughput trade-off

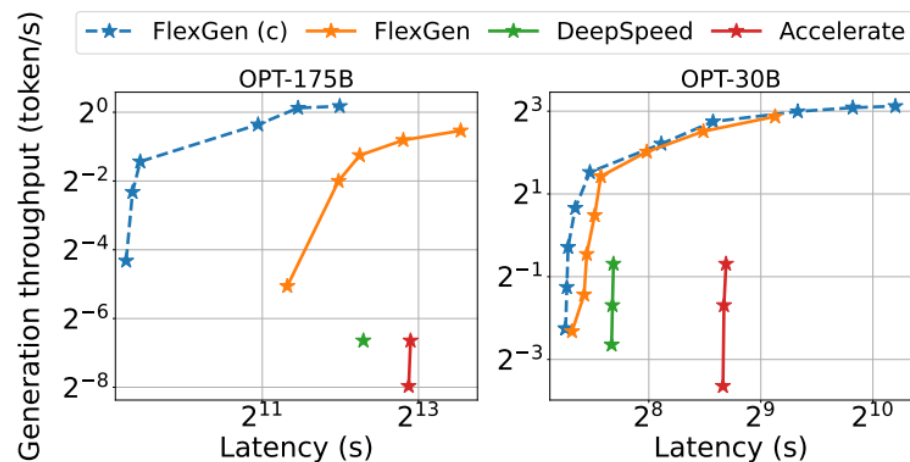


Figure 1. The total latency for a block and throughput trade-offs of three offloading-based systems for OPT-175B (left) and OPT-30B (right) on a single NVIDIA T4 (16 GB) GPU with 208 GB CPU DRAM and 1.5TB SSD. FlexGen achieves a new Pareto-optimal frontier with $100\times$ higher maximum throughput for OPT-175B. Other systems cannot further increase throughput due to out-of-memory issues. “(c)” denotes compression.

Evaluation

- Offloading
 - Runtime breakdown

Table 8. Execution time breakdown (seconds) for OPT-175B. The prompt length is 512. (R) denotes read and (W) denotes write.

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

The GPU compute utilization is 82% and 13% for prefill and decoding, respectively

Evaluation

Table 4. Ablation study of proposed techniques. 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). More see Appendix A.4.

Model size	30B	175B
All optimizations	7.32 (48 \times 3, 20, 80)	0.69 (32 \times 8, 0, 50)
No policy search	7.26 (48 \times 3, 0, 100)	0.27 (32 \times 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

Evaluation

- Approximations
 - Next-word prediction on Lambada (Paperno et al., 2016) and language modeling on WikiText (Merity et al., 2016).
 - “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

Table 5. The accuracy (higher is better) and perplexity (lower is better) with approximate methods.

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

Evaluation

- Offloading vs. Collaborative Inference
 - FlexGen and Petals under different network conditions

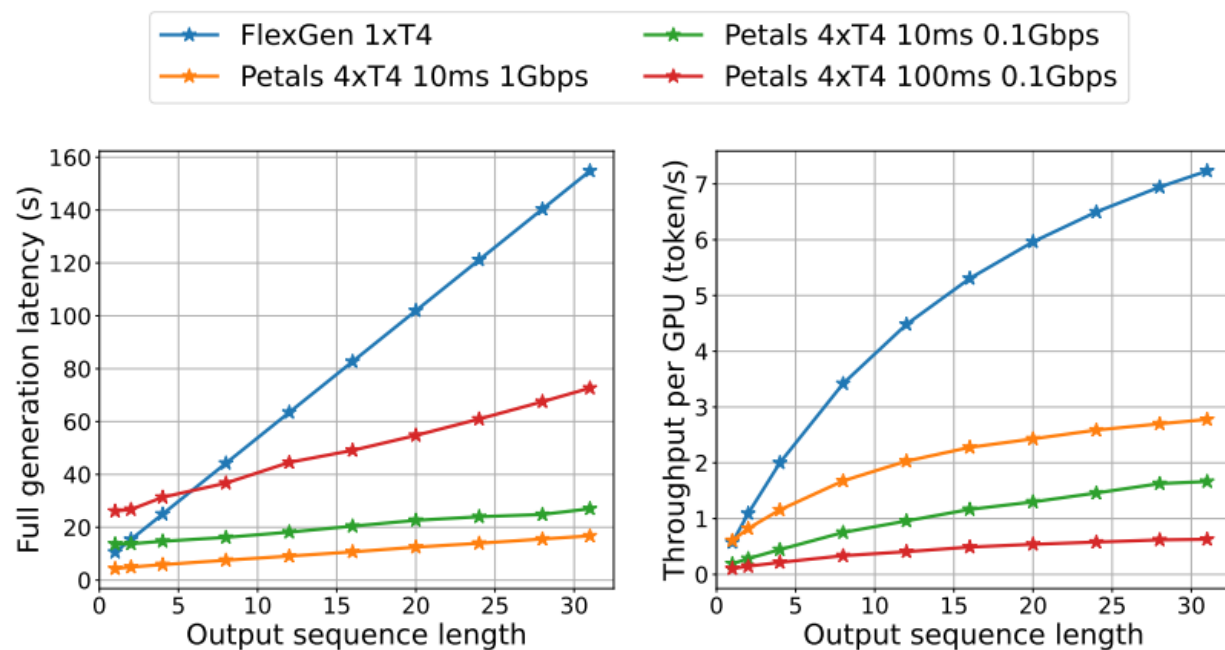


Figure 4. Full latency and per-GPU throughput of FlexGen and Petals in different network delay and bandwidth.

Contributions

- Efficient offloading strategies
 - Formulate the search space for offloading policies
 - Computation order, tensor placement, computation delegation
 - A proof of 2x optimality
 - A policy optimization algorithm
- 4-bit quantization on weights & attention (KV) cache
- Extend to distributed GPUs with pipeline parallelism

- What is MLP in transformer?
- An MLP-Attention model employs a multi-layer perceptron (MLP) to compute attention weights for sequences of word embeddings, optionally with positional encodings. The MLP is designed to perform multiple layers of nonlinear transformations on the input, and its output serves as the attention weights.

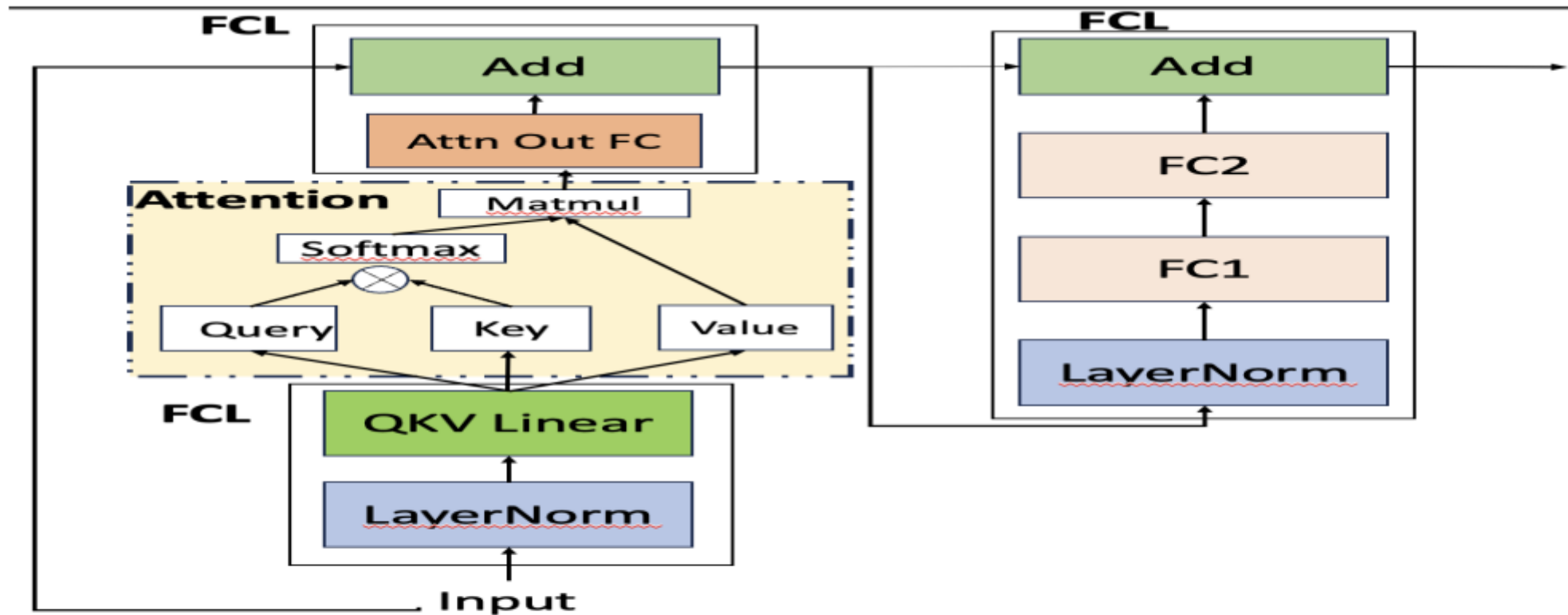


Figure 22: Transformer layer of GPT.