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]
Beidi Chen [6 7] **Percy Liang** [1] **Christopher Ré** [1] **Ion Stoica** [2] **Ce Zhang** [3]

Abstract

The high computational and memory requirements of large language model (LLM) inference make it feasible only with multiple high-end accelerators. Motivated by the emerging demand for latency-insensitive tasks with batched processing, this paper initiates the study of high-throughput LLM inference using limited resources, such as a single commodity GPU. We present FlexGen, a high-throughput generation engine for running LLMs with limited GPU memory. FlexGen can be flexibly configured under various hardware resource constraints by aggregating memory and computation from the GPU, CPU, and disk. By solving a linear programming problem, it searches for efficient patterns to store and access tensors. FlexGen further compresses the weights and the attention cache to 4 bits with negligible accuracy loss. These techniques enable FlexGen to have a larger space of batch size choices and thus significantly increase maximum throughput. As a result, when running OPT-175B on a single 16GB GPU, FlexGen achieves significantly higher throughput compared to state-of-the-art offloading systems, reaching a generation throughput of 1 token/s for the first time with an effective batch size of 144. On the HELM benchmark, FlexGen can benchmark a 30B model with a 16GB GPU on 7 representative sub-scenarios in 21 hours. The code is available at https://github.com/FMInference/FlexGen.

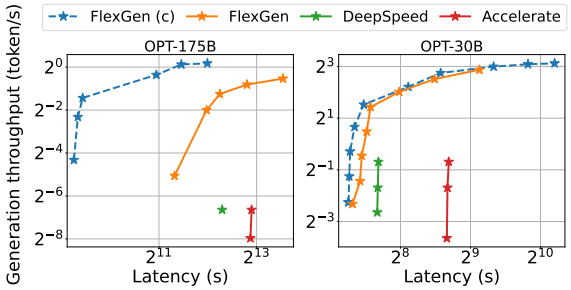


Figure 1. The total latency for a block and throughput trade-offs of three offloading-based systems for OPT-175B (left) and OPT-30B (right) on a single NVIDIA T4 (16 GB) GPU with 208 GB CPU DRAM and 1.5TB SSD. FlexGen achieves a new Pareto-optimal frontier with 100× higher maximum throughput for OPT-175B.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

1. Introduction

In recent years, large language models (LLMs) have demonstrated strong performance across a wide range of tasks (Brown et al., 2020; Bommasani et al., 2021; Zhang et al., 2022; Chowdhery et al., 2022). Along with these unprecedented capabilities, generative LLM inference comes with unique challenges. These models can have billions, if not trillions of parameters (Chowdhery et al., 2022; Fedus et al., 2022), which leads to extremely high computational and memory requirements to run. For example, GPT-175B requires 325GB of GPU memory simply to load its model weights. Fitting this model onto GPUs would require at least five A100 (80GB) GPUs and complex parallelism strategies (Pope et al., 2022; Aminabadi et al., 2022). Thus, lowering LLM inference resource requirements has recently attracted intense interest.

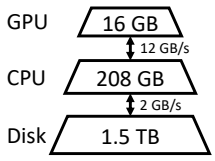
In this paper, we focus on a setting that we call *throughput-oriented generative inference*. In addition to interactive use cases such as chatbots, LLMs are also applied to many "back-of-house" tasks such as benchmarking (Liang et al., 2022), information extraction (Narayan et al., 2018), data wrangling (Narayan et al., 2022), and form processing (Chen et al., 2021). One key characteristic of these tasks is that they often require running LLM inference in batches over a large number of tokens (e.g., all the documents in a company's

[1]Stanford University [2]UC Berkeley [3]ETH Zurich [4]Yandex [5]HSE University [6]Meta [7]Carnegie Mellon University. Correspondence to: Ying Sheng <ying1123@stanford.edu>.

Proceedings of the 40^{th} International Conference on Machine Learning, Honolulu, Hawaii, USA. PMLR 202, 2023. Copyright 2023 by the author(s).

corpus), and are less sensitive to latency. As a result, it is possible to trade off latency for higher throughput in these workloads, providing opportunities to reduce resource requirements.

Prior efforts to lower resource requirements of LLM inference correspond to three directions: (1) *model compression* to decrease total memory footprint (Dettmers et al., 2022; Yao et al., 2022; Frantar et al., 2022; Xiao et al., 2022); (2) *collaborative inference* to amortize inference cost via decentralization (Borzunov et al., 2022); and (3) *offloading* to utilize memory from CPU and disk (Aminabadi et al., 2022; HuggingFace, 2022). These techniques have significantly lowered the resource requirements for using LLMs, but there are distinct limitations. Research in the first two directions often assume that the model fits into the GPU memory and thereby struggle to run 175B-scale models with a single commodity GPU. On the other hand, state-of-the-art offloading-based systems in the third category do not achieve acceptable throughput on a single GPU due to inefficient I/O scheduling and tensor placement. For example, these systems can be bottlenecked by small batch sizes (e.g., batch sizes of only one or two for OPT-175B in some cases).



Our focus is designing efficient *offloading* strategies for high-throughput generative inference, *on a single commodity GPU*. To run an LLM with limited GPU memory, we can offload it to secondary storage and perform computation part-by-part by partially loading it. On a typical machine, there are three levels of the memory hierarchy, as illustrated in the figure to the right. Higher levels are faster but scarce, while lower levels are slower but abundant. In throughput-oriented scenarios, we can sacrifice latency by using a large batch size, and amortize the expensive I/O operations among different memory hierarchies over a large batch of inputs, overlapped with computation. Fig. 1 shows the latency-throughput trade-off of three inference systems with offloading on a single NVIDIA T4 (16 GB) GPU. Note that the performance in terms of latency and throughput on limited resources is significantly inferior to that of the cases with sufficient resources.

Achieving high-throughput generative inference with limited GPU memory is challenging even if we can sacrifice the latency. The first challenge is to design an *efficient offloading strategy*. During generative inference, there are three kinds of tensors: weights, activations, and key-value (KV) cache. The strategy should specify what tensors to offload, where to offload them within the three-level memory hierarchy, and 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 where there are multiple ways to conduct computation. Together, these choices form a complex design space. Existing offloading-based inference systems (Aminabadi et al., 2022; HuggingFace, 2022) inherit strategies from training, which turn out to be some suboptimal points for inference, performing excessive I/O and achieving throughput far below theoretical hardware limits.

The second challenge is to develop *effective compression strategies*. Previous works have demonstrated promising results in compressing the weights and activations of LLMs. However, 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.

To address these challenges, we present FlexGen, an offloading framework for high-throughput LLM inference. 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) We formally define a search space of possible offloading strategies by considering computation schedule, tensor placement, and computation delegation. We prove that our search space captures a computation order with I/O complexity within $2\times$ of optimality. We then develop a linear programming-based search algorithm to optimize the throughput within the search space. This algorithm can be configured for various hardware specifications and can be easily extended to incorporate latency and throughput constraints, thus helping to navigate the trade-off space smoothly. Compared with existing strategies, our solution unifies the placement of weights, activations, and the KV cache, enabling a dramatically higher batch size upper bound, which is key to achieving high throughput.

(Contribution 2) We show that it is possible to compress both the weights and KV cache for LLMs like OPT-175B to 4 bits without retraining or calibration, all with negligible accuracy loss. This is achieved through fine-grained group-wise quantization (Shen et al., 2020), which is suitable for reducing I/O costs and memory usage during offloading.

(Contribution 3) We demonstrate the efficiency of FlexGen by running OPT-175B on NVIDIA T4 (16GB) GPUs. Compared to DeepSpeed Zero-Inference (Aminabadi et al., 2022) and Hugging Face Accelerate (HuggingFace, 2022), two state-of-the-art offloading-based inference systems, FlexGen often allows a batch size that is orders of magnitude larger. As a result, FlexGen can achieve much higher throughputs. On a single T4 GPU with 208 GB CPU DRAM and 1.5 TB SSD, input sequence length 512, and output sequence length 32:

- With the same latency of 5000 seconds, FlexGen (effective batch size 64, or 2048 tokens in total) can achieve

more than $40\times$ higher throughput than DeepSpeed Zero-Inference (batch size 1, or 32 tokens in total), while Hugging Face Accelerate cannot complete a single batch.
- By allowing a higher latency of 12000 seconds, FlexGen achieves $69\times$ higher maximum throughput compared to baselines because it can enlarge the effective batch size to 256 (8192 tokens generated in total), while DeepSpeed Zero-Inference and Hugging Face Accelerate cannot use a batch size larger than 2 due to out-of-memory issues.
- If allowing 4-bit compression, FlexGen can reach $100\times$ higher maximum throughput with effective batch size 144 (4608 tokens generated in total) with latency 4000 seconds by holding all weights in CPU and getting rid of disk offloading.

We also compare offloading and decentralized collective inference based on FlexGen and Petals (Borzunov et al., 2022) as two representative systems. We conduct comparisons between the two systems from the aspects of delay and bandwidth of the decentralized network and output sequence length. The results show that FlexGen outperforms a decentralized Petals cluster in terms of per-GPU throughput and can even achieve lower latency in certain cases.

2. Related Work

Given the recent advances of LLMs, LLM inference has become an important workload, encouraging active research from both the **system** side and the **algorithm** side.

Recent years have witnessed the emergence of systems specialized for LLM inference, such as FasterTransformer (NVIDIA, 2022), Orca (Yu et al., 2022), LightSeq (Wang et al., 2021), PaLM inference (Pope et al., 2022), TurboTransformers (Fang et al., 2021), DeepSpeed Inference (Aminabadi et al., 2022), and Hugging Face Accelerate (HuggingFace, 2022). Unfortunately, most of these systems focus on latency-oriented scenarios with high-end accelerators, limiting their deployment for throughput-oriented inference on easily accessible hardware. To enable LLM inference on such commodity hardware, offloading is an essential technique — as far as we know, among current systems, only DeepSpeed Zero-Inference and Hugging Face Accelerate support offloading. These inference systems typically inherit the offloading techniques from training systems (Rajbhandari et al., 2021; Ren et al., 2021; Li et al., 2022; Huang et al., 2020; Wang et al., 2018) but ignore the special computational property of generative inference. They fail to exploit the structure of the throughput-oriented LLM inference computation and miss great opportunities for efficient scheduling of I/O traffic. Another attempt to enable LLM inference on accessible hardware is collaborative computing proposed by Petals (Borzunov et al., 2022).

There are also many algorithm-oriented works that relax certain aspects of computation in LLM inference to accelerate the computation or reduce the memory footprint. Both sparsification (Hoefler et al., 2021; Frantar & Alistarh, 2023) and quantization (Kwon et al., 2022; Yao et al., 2022; Park et al., 2022; Xiao et al., 2022; Frantar et al., 2022; Dettmers et al., 2022) have been adopted for LLM inference. On the quantization side, prior works have shown weights can be compressed down to 3 bits without compressing activations (Frantar et al., 2022), or both weights and activations can be compressed to 8 bits (Yao et al., 2022; Dettmers et al., 2022; Xiao et al., 2022). In FlexGen, we compress both the weights and KV cache to 4 bits and show how to combine the compression with offloading to make further improvements.

Within broader domains, memory optimizations and offloading have been studied for training (Huang et al., 2020; Ren et al., 2021; Steiner et al., 2022) and linear algebra (Jia-Wei & Kung, 1981; Demmel, 2013).

3. Background: LLM Inference

In this section, we describe the LLM inference workflow and its memory footprint.

Generative Inference. A typical LLM generative inference task consists of 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; and ii) the *decoding* stage which utilizes and updates the KV cache to generate tokens step-by-step, where the current token generation depends on previously generated tokens.

For a particular inference computation, denote the batch size by b, the input sequence length by s, the output sequence length by n, the hidden dimension of the transformer by h_1, the hidden dimension of the second MLP layer by h_2, and the total number of transformer layers by l. Given the weight matrices of a transformer layer specified by $\mathbf{w}_K^i$, $\mathbf{w}_Q^i$, $\mathbf{w}_V^i$, $\mathbf{w}_O^i$, $\mathbf{w}_1^i$, $\mathbf{w}_2^i$, where $\mathbf{w}_K^i, \mathbf{w}_Q^i, \mathbf{w}_V^i, \mathbf{w}_O^i \in \mathcal{R}^{h_1 \times h_1}$, $\mathbf{w}_1 \in \mathcal{R}^{h_1 \times h_2}$, and $\mathbf{w}_2 \in \mathcal{R}^{h_2 \times h_1}$.

During the ***prefill phase***, the input of the i-th layer is specified by $\mathbf{x}^i$, and `key`, `value`, `query`, and `output` of the attention layer is specified by $\mathbf{x}_K^i$, $\mathbf{x}_V^i$, $\mathbf{x}_Q^i$, $\mathbf{x}_{\text{Out}}^i$, where $\mathbf{x}^i, \mathbf{x}_K^i, \mathbf{x}_V^i, \mathbf{x}_Q^i, \mathbf{x}_{\text{Out}}^i \in \mathcal{R}^{b \times s \times h_1}$. Then, the cached `key`, `value` can be computed by:

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

The rest of the computation in the i-th layer is:

$$\mathbf{x}_Q^i = \mathbf{x}^i \cdot \mathbf{w}_Q^i$$
$$\mathbf{x}_{\text{Out}}^i = f_{\text{Softmax}}\left(\frac{\mathbf{x}_Q^i {\mathbf{x}_K^i}^T}{\sqrt{h}}\right) \cdot \mathbf{x}_V^i \cdot \mathbf{w}_O^i + \mathbf{x}^i$$
$$\mathbf{x}^{i+1} = f_{\text{relu}}\left(\mathbf{x}_{\text{Out}}^i \cdot \mathbf{w}_1\right) \cdot \mathbf{w}_2 + \mathbf{x}_{\text{Out}}^i$$

During the ***decode phase***, given $\mathbf{t}^i \in \mathcal{R}^{b \times 1 \times h_1}$ as the embedding of the current generated token in the i-th layer, the inference computation needs to i) update the KV cache:

$$\mathbf{x}_K^i \leftarrow \mathsf{Concat}\left(\mathbf{x}_K^i, \mathbf{t}^i \cdot \mathbf{w}_K^i\right)$$
$$\mathbf{x}_V^i \leftarrow \mathsf{Concat}\left(\mathbf{x}_V^i, \mathbf{t}^i \cdot \mathbf{w}_V^i\right)$$

and ii) compute the output of the current layer:

$$\mathbf{t}_Q^i = \mathbf{t}^i \cdot \mathbf{w}_Q^i$$
$$\mathbf{t}_{\mathsf{Out}}^i = f_{\mathsf{Softmax}}\left(\frac{\mathbf{t}_Q^i \mathbf{x}_K^{i\,T}}{\sqrt{h}}\right) \cdot \mathbf{x}_V^i \cdot \mathbf{w}_O^i + \mathbf{t}^i$$
$$\mathbf{t}^{i+1} = f_{\mathsf{relu}}\left(\mathbf{t}_{\mathsf{Out}}^i \cdot \mathbf{w}_1\right) \cdot \mathbf{w}_2 + \mathbf{t}_{\mathsf{Out}}^i$$

Memory Analysis. The memory footprint of LLM inference mainly comes from the model weights and the KV cache. Considering the OPT-175B model in `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 $4 \times blh_1(s+n)$.

In a realistic setting with a sufficient number of GPUs, the 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, making the KV cache a new bottleneck of large-batch high-throughput inference. In FlexGen, for OPT-175B, we enlarge the effective batch size to 256 to achieve the throughput at 0.69 token/s.

Throughput and Latency. Considering an effective batch size b, an input sequence length s, and an output sequence length of n, the 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.

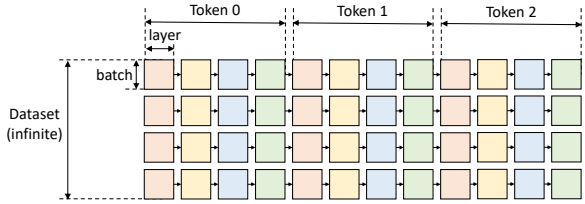


Figure 2. Computational graph of LLM inference.

4. Offloading Strategy

In this section, we do not relax any computation of LLM inference and illustrate how to formalize the offloading procedure under the GPU, CPU, and disk memory hierarchy. We first formulate the problem and then construct the search space of the possible offloading strategies in FlexGen. To find an efficient strategy, FlexGen builds an analytical cost model and searches for configurations with an optimizer based on linear programming.

[1] We ignore the embedding layer(s), which is relatively small.

4.1. Problem Formulation

Consider a machine with three devices: a GPU, a CPU, and a disk. The GPU and CPU can perform computation while the disk cannot. The three devices form a three-level memory hierarchy where the GPU has the smallest but fastest memory and the disk has the largest but slowest memory. When an LLM cannot fit entirely within the GPU, we need to offload it to secondary storage and perform computation part-by-part by partially loading the LLM.

We formulate the generative inference with offloading as a graph traversal problem. Fig. 2 shows an example computational graph, where the model has 4 layers and we generate 3 tokens per prompt. As our focus is throughput-oriented scenarios, we assume a given dataset with an infinite number of prompts that need to be processed. 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. We 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.

4.2. Search Space

Given the formulation above, we construct a search space for possible valid strategies in FlexGen.

Compute schedule. Intuitively, there are two orders to traverse the graph in Fig. 2: row-by-row and column-by-column. All existing systems (Aminabadi et al., 2022; HuggingFace, 2022) traverse the graph row-by-row, as shown in Fig. 3(a). This is reasonable because it is the fastest way to finish the generation for one batch and the KV cache can be freed immediately after a row. However, because every two contiguous squares do not share weights, this schedule has to repeatedly load the weights and incurs huge I/O costs.

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. How-

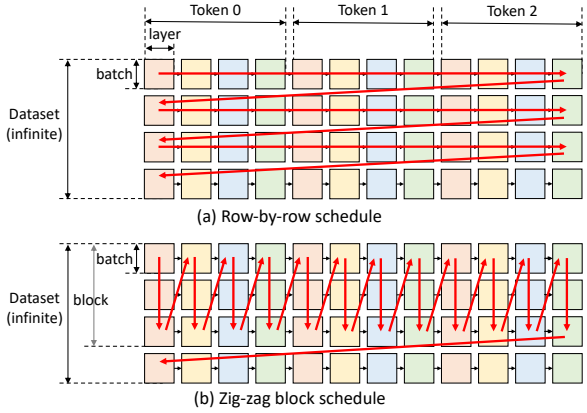


Figure 3. Two different schedules. The red arrows denote the computation order.

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
```

ever, we cannot traverse a column all the way to the end because the activations and KV cache still need to be stored. Hence, we have to stop when they fill the CPU and disk memory. Taking all this into consideration, we converge to a zig-zag block schedule, as shown in Fig. 3(b). Besides, we propose another more advanced and I/O-optimal schedule, but only implement the simpler block schedule due to the practical implementation difficulty of the optimal one. However, we prove that the block schedule is at most twice worse than the optimal schedule in Appendix A.2.

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

Another typical optimization is overlapping. We can overlap 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. Adding overlapping to the block schedule results in Algorithm 1. The first six functions in the innermost loop can be seen as launched in parallel with six logical threads because there are no dependencies. The last function then synchronizes these six logical threads. We rely on operating systems and CUDA drivers to resolve the schedule of the underlying hardware resources. As a conclusion, the algorithm introduces two parameters into our search space: the GPU batch size and the number of GPU batches in a block. The product of the GPU batch size and the number of GPU batches is called block size (or **effective batch size**).

Tensor placement. Besides compute schedule, a strategy should specify how to store these tensors within the memory hierarchy. We use three variables wg, wc, and wd to define the percentages of weights stored on GPU, CPU, and disk respectively. Similarly, we use three variables hg, hc, hd to define the percentages of activations and use cg, cc, cd for the KV cache. Given the percentages, there are still multiple ways to partition the tensors. Taking weight tensors as an example, from coarse grain to fine grain, we can 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), or 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. Considering both the runtime overhead and desired flexibility, we use layer granularity for weights, and tensor granularity for activations and the KV cache.

Computation delegation. While CPUs are much slower than GPUs, we find using CPU compute can still be beneficial in some cases. This is because the computation of attention scores during decoding is I/O-bounded. Consider a case where the KV cache is stored on the CPU. 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. In contrast, 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. Quantitatively, let b be the GPU batch size, s be the sequence length, and h_1 be the hidden size. 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 $b \times h_1 \times 4$ bytes, so computing attention score on CPU reduces I/O by $s\times$. 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.

4.3. Cost Model and Policy Search

The schedule and placement in Section 4.2 constructs a search space with several parameters. Now we develop an analytical cost model to estimate the execution time given these algorithm parameters and hardware specifications.

Cost Model. The cost model predicts the latency during prefill for one layer denoted as T_{pre}, and the averaged la-

tency during decoding for one layer denoted as T_{gen} in one block. The total latency for computing a block can then be estimated as $T = T_{pre} \cdot l + T_{gen} \cdot (n-1) \cdot l$, where l is the number of layers and n is the number of tokens to generate.

Assuming perfect overlapping, T_{pre} can be estimated as $T_{pre} = \max(ctog^p, gtoc^p, dtoc^p, ctod^p, comp^p)$, where $ctog^p$, $gtoc^p$, $dtoc^p$, $ctod^p$, $comp^p$ 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.

Similarly, T_{gen} can be estimated as $T_{gen} = \max(ctog^g, gtoc^g, dtoc^g, ctod^g, comp^g)$, with $ctog^g$, $gtoc^g$, $dtoc^g$, $ctod^g$, $comp^g$ 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.

For I/O terms like $dtoc^g$, it is 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, with h_1 denoting the hidden size, and h_2 denoting the hidden size of the second MLP layer. Let bls be the block size and s be the prompt length; then the size of activations for one layer is $2 \cdot bls \cdot h_1$. The size of the KV cache for one layer on average is $4 \cdot bls \cdot (s + \frac{n}{2}) \cdot h_1$. We have to load wd, hd, cd percent of weights, activations, and the KV cache from the disk respectively so that the total latency of disk read is $dtoc^g = \frac{1}{\text{disk_to_cpu_bandwidth}}((8h_1^2 + 4h_1 \cdot h_2) \cdot wd + 4 \cdot bls \cdot (s + \frac{n}{2}) \cdot h_1 \cdot cd + 2 \cdot bls \cdot h_1 \cdot hd)$.

Similarly for computation terms, we sum up all computation events, including matrix multiplications and batched matrix multiplications on the CPU and the GPU.

Besides latency estimation, we also estimate the peak memory usage of the GPU, CPU, and disk, and then we add memory constraints. The full cost model is in Appendix A.3.

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. In practice, the percentage cannot be an arbitrary real number between 0 and 1, because the tensor cannot be split arbitrarily. However, we relax the percentage variables in the cost model to be any real number between 0 and 1 since it is changing gradually. We solve the problem as a two-level optimization problem. We first 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). The linear programming problem can be solved very quickly because there are only 9 variables. This formulation can also be flexibly extended to include latency constraints and model approximate methods such as compression.

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

To use the cost model, we run profiling on the hardware to sample some data points and fit the hardware parameters. We then call the optimizer to get an offloading policy. Due to our relaxation and the hardness of accurately modeling peak memory usage (e.g., fragmentation), sometimes a strategy from the policy search can run out of memory. In this case, we manually adjust the policy slightly. The cost model can usually return a good policy, but it is common that a better policy can be obtained by tuning manually.

4.4. Extension to Multiple GPUs

We discuss how to extend the offloading strategy in FlexGen if there are multiple GPUs. Although we can find a nearly optimal strategy for one GPU, the strategy is still heavily limited by I/O and has a low GPU utilization. If we are given more GPUs and more CPUs, model parallelism can be utilized to reduce the memory pressure of each GPU, which can potentially lead to a super-linear scaling in decoding.

There are two kinds of model parallelisms: tensor and pipeline parallelism (Narayanan et al., 2021; Zheng et al., 2022). Tensor parallelism can reduce the single-query latency but pipeline parallelism can achieve good scaling on throughput due to its low communication costs. Since we target throughput, FlexGen implements pipeline parallelism.

We use pipeline parallelism by equally partitioning an l-layer LLM on m GPUs, and then the execution of all GPUs follows the same pattern. The problem is reduced to running an n/m-layer transformer on one GPU. We can directly reuse the policy search developed for one GPU. To achieve micro-batch pipelining, a new for-loop is added to Algorithm 1 to combine the iteration-level pipeline parallel execution schedule (Huang et al., 2019; Yu et al., 2022) with our single-device offloading runtime.

5. Approximate Methods

The previous section focuses on the exact computation. However, the inference throughput can be greatly boosted with negligible accuracy loss by allowing some approximations, because LLMs are typically robust to careful approximations. This section introduces two such approximations: group-wise quantization and sparse attention.

Group-wise Quantization. We show that 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 (Section 6.2). When compared to some related works (Yao et al., 2022; Dettmers et al., 2022; Xiao et al., 2022) that try to use integer matrix multiplication mainly for accelerated computation, the goal of quantization in our case is primarily for compression and reducing I/O costs. Therefore, we can choose a fine-grained quantization format in favor of a high compression ratio and dequantize the tensors back to FP16 before computation. We use a fine-grained group-wise asymmetric quantization method (Shen et al., 2020). Given a tensor, we 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 $x_{quant} = round\left(\frac{x-min}{max-min} \times (2^b - 1)\right)$.

The tensors are stored in the quantized format and converted back to FP16 before computation. Since both the weights and KV cache consume a significant amount of memory, we compress both to 4 bits with a group size of 64. There are multiple ways to choose which dimension to group on. We find that grouping the weights along the output channel dimension and the KV cache along the hidden dimension preserves the accuracy while being runtime-efficient in practice. One thing to mention is that such a 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 we turn off the CPU delegation when enabling quantization. A concurrent work (Dettmers & Zettlemoyer, 2022) also finds that 4-bit precision is almost optimal for total model bits and zero-shot accuracy on OPT models. Compared to this previous work, we first propose to compress the KV cache and present the results on OPT-175B.

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, all while maintaining the model quality. We present one simple Top-K sparse approximation. After computing the attention matrices, for each query, we calculate the indices of its Top-K tokens from the K cache. We then simply drop the other tokens and only load a subset of the V cache according to the indices.

The application of these approximations is straightforward. We present these preliminary but interesting results and intend to emphasize that FlexGen is a general framework that can seamlessly plug in many approximation methods.

6. Evaluation

Hardware. We run experiments on the NVIDIA T4 GPU instances from Google Cloud. The hardware specifications are listed in Table 1. The read bandwidth of SSD is about 2GB/s and the write bandwidth is about 1GB/s. Our methods and implementations do not depend on specific hardware architectures. Some architecture (e.g. unified memory) could be more friendly to our method. See Appendix A.4 for discussions and experiments on different hardware setups.

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

Model. OPT models (Zhang et al., 2022) with 6.7B to 175B parameters are used in the evaluation. Although we do not evaluate other models, the offloading in FlexGen can be applied to other transformer LLMs, e.g., GPT-3 (Brown et al., 2020), PaLM (Chowdhery et al., 2022), and BLOOM (Scao et al., 2022) because they all share a similar structure.

Workload. Our focus is high-throughput generation on a given dataset. We use synthetic datasets where all prompts are padded to the same length. The system is required to generate 32 tokens for each prompt. We test two prompt lengths: 512 and 1024 (for experiments in more settings, see Appendix A.4). The evaluation metric is generation throughput, defined as the number of generated tokens / (prefill time + decoding time). Sometimes running a full batch takes too long for certain systems — in this cases, we generate fewer tokens and project the final throughput. We use dummy model weights in throughput benchmarks for all systems and real weights for accuracy evaluations.

Baseline. We use DeepSpeed ZeRO-Inference (Aminabadi et al., 2022) and Hugging Face Accelerate (HuggingFace, 2022) as baselines. They are the only systems that can run LLMs with offloading when there is not enough GPU memory. DeepSpeed supports offloading the whole weights to the CPU or disk. It uses ZeRO data parallelism if there are multiple GPUs. Accelerate supports offloading a fraction of the weights. It does not support distributed GPUs on different machines. Both of them use the row-by-row schedule and can only put cache/activations on GPU. These systems support different quantization methods. However, the quantization in Accelerate is not compatible with offloading, and the quantization in DeepSpeed cannot preserve accuracy up to 175B, so we do not enable quantization on these systems. In addition to offloading, decentralized collaborative inference is another option to lower the resource requirement for LLM inference. Thus, we also include Petals (Borzunov et al., 2022; Ryabinin et al., 2023) as an additional baseline.

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.

6.1. Offloading

Maximum throughput benchmark. We first evaluate the maximum generation throughput the systems can achieve with one GPU on two prompt lengths. As shown in Table 2, FlexGen outperforms all baselines in all cases. On OPT-6.7B, Accelerate and FlexGen can successfully fit the whole model into a single GPU, so they choose to only use the GPU. DeepSpeed has a higher memory overhead and cannot fit OPT-6.7B into the GPU, so it uses slower CPU offloading. On OPT-30B, all systems switch to CPU offloading. DeepSpeed and Accelerate store the KV cache on the GPU, so they cannot use a very large batch size, while FlexGen offloads most weights and all KV cache to the CPU and enables a larger GPU batch size. In addition, FlexGen reuses the weights by block scheduling. On OPT-175B, all systems start to offload the weights to the disk. Baseline systems can only use a maximum batch size of 2, but FlexGen can use a GPU batch size of 32 and a block size of 32×8, achieving a $69\times$ higher throughput. With compression enabled, FlexGen achieves a $112\times$ higher generation throughput on a single GPU for prompt sequence length 512. This huge improvement is because FlexGen uses an effective batch size of 144 and compresses the weights and KV cache to fit into CPU memory to avoid slow disk swapping. More details on the policy setups and effective batch sizes can be found in Appendix A.4. More experiments on how disk specification affects the throughput see Appendix A.4.

Table 3 shows the results on 4 machines, with one GPU on each machine. OPT-30B or OPT-175B still cannot fit into 4 GPUs. Naively, we can run 4 independent FlexGen in a data-parallel fashion to get a linear scaling on throughput. But here we show that pipeline parallelism can achieve super-linear scaling on decoding throughput. With pipeline parallelism, the memory pressure of each machine is reduced so we can switch from small batch sizes to larger batch sizes, or switch from disk offloading to CPU-only offloading. In Table 3, FlexGen does not achieve linear scaling on generation throughput (which counts both prefill and decoding time costs). This is because there are pipeline bubbles during the prefill stage and our workload settings only generate 32 tokens. However, FlexGen achieves super-linear scaling on decoding throughput (which only counts decoding time costs assuming the prefill is done). This means if we generate more tokens, pipeline parallelism will show its benefits as decoding time will dominate.

Latency-throughput trade-off. We configure these systems to achieve maximum throughput under various latency constraints and draw their latency-throughput trade-off curves in Fig. 1. FlexGen sets a new Pareto-optimal frontier that significantly outperforms baselines. On the low-latency side, FlexGen supports partial offloading and uses more space for weights. On the high-throughput side,

Table 2. Generation throughput (token/s) of different systems. Accelerate, DeepSpeed, and FlexGen use 1 GPU. Petals uses 1 GPU for OPT-6.7B, 4 GPUs for OPT-30B, and 24 GPUs for OPT-175B, but reports per-GPU throughput. We benchmark Petals under a good network assumption with a delay of less than 10ms and bandwidth of 1 Gbps. The models are run in INT8 as the default for Petals. See Section 6.3 for more details about Petals. FlexGen is our system without compression; FlexGen (c) uses 4-bit compression. "OOM" means out-of-memory.

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

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

FlexGen aggressively offloads all things out of the GPU to achieve a large GPU batch size and block size. Given the same latency requirement of 5000 seconds, FlexGen without compression can achieve a $40\times$ higher throughput compared to DeepSpeed and Accelerate. If allowing a higher latency and compression, FlexGen can further boost throughput and reach a $100\times$ improvement by using an effective batch size of 144. In this case, compression enables FlexGen to fit all things in the CPU memory and avoid disk I/O. The detailed latency, throughput, and policy setup can be found in Appendix A.4.

Runtime breakdown. We shows the runtime breakdown of OPT-175B on FlexGen in Table 8 in Appendix A.4. We disable overlapping and profile the time used for major components. The GPU compute utilization is 82% and 13% for prefill and decoding, respectively.

Ablation study. We then isolate the improvement brought by each individual technique. Table 4 lists the throughput FlexGen can achieve if disabling one technique at a time. On OPT-30B, with all optimizations enabled, we put 20% weights on GPU, 80% weights on CPU, and all activations and KV cache to CPU. We also choose a GPU batch size of 48 and a block size of 48×3. "No policy search" illustrates the performance of worse strategies, showing the importance of a good policy. On both models, using CPU compute and overlapping brings non-trivial improvement. We also

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 × #GPU-batch, wg, wc). More see Appendix A.4.

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

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

port the policy used in DeepSpeed/Accelerate into FlexGen runtime, showing the suboptimality of their policy. A more detailed ablation study can be found in Appendix A.4.

HELM and Data wrangling. We tested the interaction of FlexGen and HELM (Liang et al., 2022) by evaluating a new model OPT-IML-30B (Iyer et al., 2022), which has not been included in the official release of HELM. FlexGen finishes the benchmark of 7 representative sub-scenarios in 21 hours , with all system overhead included, under the hardware setup described in Table 1. Table 9 in Appendix A.4 shows the details of the tasks and the corresponding running time. We also use FlexGen to run the data wrangling tasks (Narayan et al., 2022) with OPT models. The detailed task configurations and running time are in Appendix A.4.

6.2. Approximations

We use two tasks to show that our approximation methods exhibit negligible accuracy loss: next-word prediction on Lambada (Paperno et al., 2016) and language modeling on WikiText (Merity et al., 2016). As shown in Table 5, "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`. The results reveal the robustness of LLMs against these approximations. We also tried 3-bit compression but it cannot preserve accuracy.

6.3. Offloading vs. Collaborative Inference

We compare FlexGen and Petals under different network conditions by setting a private Petals cluster on GCP with 4 nodes having one T4 GPU per node. We use Linux traffic control to constrain the connections between instances to simulate a realistic decentralized network and benchmark the performance of an OPT-30B model (input sequence length: 512, output sequence length: 32). We tune the batch size of each request to be 2 and issue requests by 6 parallel client processes to achieve the maximum throughput[2]. In addition, we normalize the throughput of Petals by the number of used GPUs. As shown in Fig. 4, we find that the throughput of FlexGen with a single T4 outperforms the per-GPU throughput of the Petals cluster under all tested network conditions. Petals does not utilize offloading, so it cannot use a very large batch size, which limits its scaling on throughput. Thus, we believe offloading could be a more efficient solution for throughput than communicating a large volume of activations in a long decentralized pipeline; on the other hand, collaborative inference can be a more viable option in more latency-sensitive scenarios.

Interestingly, we find that FlexGen can achieve lower latency than Petals in slow networks with short generation. We speculate this is because the network bandwidth becomes the bottleneck for activation transfer, and a large delay incurs a significant overhead on each communication step in the pipeline. For the curve of a 100ms delay network, we can observe a cross point between FlexGen and Petals. This is because the activations during prefill are larger than the activations during decoding by a factor of the input sequence length. Thus, the communication overhead is proportionally larger, which significantly slows down Petals during prefill.

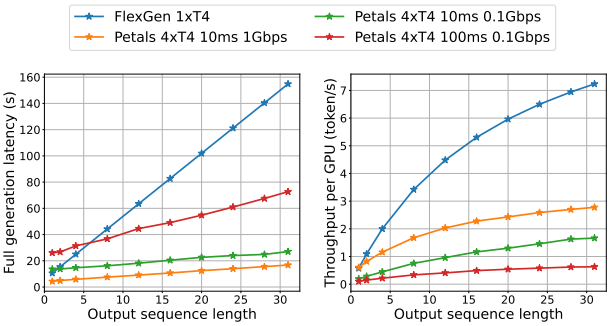


Figure 4. Full latency and per-GPU throughput of FlexGen and Petals in different network delay and bandwidth.

7. Conclusion

We introduce FlexGen, a high-throughput generation engine for LLM inference, which focuses on latency-insensitive batch-processing tasks for resource-constrained scenarios.

Acknowledgements

We would like to thank Clark Barrett and Joseph E. Gonzalez for funding support, and Zhiqiang Xie, Daniel Y. Fu, Hao Zhang, Nick Chow, Benjamin Spector, Guangxuan Xiao, Jue Wang, Arjun Desai, Yao Fu, Anjiang Wei, and Zihao Ye for their insightful review and discussions.

[2]The batch size of 1 did not result in a noticeably better latency.

References

Aminabadi, R. Y., Rajbhandari, S., Awan, A. A., Li, C., Li, D., Zheng, E., Ruwase, O., Smith, S., Zhang, M., Rasley, J., et al. Deepspeed-inference: Enabling efficient inference of transformer models at unprecedented scale. In *2022 SC22: International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pp. 646–660. IEEE Computer Society, 2022.

Bommasani, R., Hudson, D. A., Adeli, E., Altman, R., Arora, S., von Arx, S., Bernstein, M. S., Bohg, J., Bosselut, A., Brunskill, E., et al. On the opportunities and risks of foundation models. *arXiv preprint arXiv:2108.07258*, 2021.

Borzunov, A., Baranchuk, D., Dettmers, T., Ryabinin, M., Belkada, Y., Chumachenko, A., Samygin, P., and Raffel, C. Petals: Collaborative inference and fine-tuning of large models. *arXiv preprint arXiv:2209.01188*, 2022.

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33: 1877–1901, 2020.

Chen, X., Maniatis, P., Singh, R., Sutton, C., Dai, H., Lin, M., and Zhou, D. Spreadsheetcoder: Formula prediction from semi-structured context. In *International Conference on Machine Learning*, pp. 1661–1672. PMLR, 2021.

Chowdhery, A., Narang, S., Devlin, J., Bosma, M., Mishra, G., Roberts, A., Barham, P., Chung, H. W., Sutton, C., Gehrmann, S., et al. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*, 2022.

Demmel, J. Communication-avoiding algorithms for linear algebra and beyond. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pp. 585–585. IEEE, 2013.

Dettmers, T. and Zettlemoyer, L. The case for 4-bit precision: k-bit inference scaling laws. *arXiv preprint arXiv:2212.09720*, 2022.

Dettmers, T., Lewis, M., Belkada, Y., and Zettlemoyer, L. Llm.int8(): 8-bit matrix multiplication for transformers at scale. In Oh, A. H., Agarwal, A., Belgrave, D., and Cho, K. (eds.), *Advances in Neural Information Processing Systems*, 2022.

Fang, J., Yu, Y., Zhao, C., and Zhou, J. Turbotransformers: an efficient gpu serving system for transformer models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 389–402, 2021.

Fedus, W., Zoph, B., and Shazeer, N. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *Journal of Machine Learning Research*, 23(120):1–39, 2022.

Frantar, E. and Alistarh, D. Massive language models can be accurately pruned in one-shot. *arXiv preprint arXiv:2301.00774*, 2023.

Frantar, E., Ashkboos, S., Hoefler, T., and Alistarh, D. Gptq: Accurate post-training quantization for generative pre-trained transformers. *arXiv preprint arXiv:2210.17323*, 2022.

Hoefler, T., Alistarh, D., Ben-Nun, T., Dryden, N., and Peste, A. Sparsity in deep learning: Pruning and growth for efficient inference and training in neural networks. *J. Mach. Learn. Res.*, 22(241):1–124, 2021.

Huang, C.-C., Jin, G., and Li, J. Swapadvisor: Pushing deep learning beyond the gpu memory limit via smart swapping. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 1341–1355, 2020.

Huang, Y., Cheng, Y., Bapna, A., Firat, O., Chen, D., Chen, M., Lee, H., Ngiam, J., Le, Q. V., Wu, Y., et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems*, 32, 2019.

HuggingFace. Hugging face accelerate. https://huggingface.co/docs/accelerate/index, 2022.

Iyer, S., Lin, X. V., Pasunuru, R., Mihaylov, T., Simig, D., Yu, P., Shuster, K., Wang, T., Liu, Q., Koura, P. S., et al. Opt-iml: Scaling language model instruction meta learning through the lens of generalization. *arXiv preprint arXiv:2212.12017*, 2022.

Jia-Wei, H. and Kung, H.-T. I/o complexity: The red-blue pebble game. In *Proceedings of the thirteenth annual ACM symposium on Theory of computing*, pp. 326–333, 1981.

Kwon, S. J., Kim, J., Bae, J., Yoo, K. M., Kim, J.-H., Park, B., Kim, B., Ha, J.-W., Sung, N., and Lee, D. Alphatuning: Quantization-aware parameter-efficient adaptation of large-scale pre-trained language models. *arXiv preprint arXiv:2210.03858*, 2022.

Li, Y., Phanishayee, A., Murray, D., Tarnawski, J., and Kim, N. S. Harmony: Overcoming the hurdles of gpu memory capacity to train massive dnn models on commodity servers. *arXiv preprint arXiv:2202.01306*, 2022.

Liang, P., Bommasani, R., Lee, T., Tsipras, D., Soylu, D., Yasunaga, M., Zhang, Y., Narayanan, D., Wu, Y., Kumar, A., et al. Holistic evaluation of language models. *arXiv preprint arXiv:2211.09110*, 2022.

Merity, S., Xiong, C., Bradbury, J., and Socher, R. Pointer sentinel mixture models. *arXiv preprint arXiv:1609.07843*, 2016.

Morton, A. Pagecachemangagement. https://code.google.com/archive/p/pagecache-mangagement/source/default/source, 2008.

Narayan, A., Chami, I., Orr, L., and Ré, C. Can foundation models wrangle your data? *arXiv preprint arXiv:2205.09911*, 2022.

Narayan, S., Cohen, S. B., and Lapata, M. Don't give me the details, just the summary! topic-aware convolutional neural networks for extreme summarization. *arXiv preprint arXiv:1808.08745*, 2018.

Narayanan, D., Shoeybi, M., Casper, J., LeGresley, P., Patwary, M., Korthikanti, V., Vainbrand, D., Kashinkunti, P., Bernauer, J., Catanzaro, B., et al. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–15, 2021.

NVIDIA. Fastertransformer. https://github.com/NVIDIA/FasterTransformer, 2022.

Paperno, D., Kruszewski, G., Lazaridou, A., Pham, N.-Q., Bernardi, R., Pezzelle, S., Baroni, M., Boleda, G., and Fernández, R. The lambada dataset: Word prediction requiring a broad discourse context. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 1525–1534, 2016.

Park, G., Park, B., Kwon, S. J., Kim, B., Lee, Y., and Lee, D. nuqmm: Quantized matmul for efficient inference of large-scale generative language models. *arXiv preprint arXiv:2206.09557*, 2022.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

Pope, R., Douglas, S., Chowdhery, A., Devlin, J., Bradbury, J., Levskaya, A., Heek, J., Xiao, K., Agrawal, S., and Dean, J. Efficiently scaling transformer inference. *arXiv preprint arXiv:2211.05102*, 2022.

Rajbhandari, S., Ruwase, O., Rasley, J., Smith, S., and He, Y. Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–14, 2021.

Ren, J., Rajbhandari, S., Aminabadi, R. Y., Ruwase, O., Yang, S., Zhang, M., Li, D., and He, Y. Zero-offload: Democratizing billion-scale model training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pp. 551–564, 2021.

Ryabinin, M., Dettmers, T., Diskin, M., and Borzunov, A. Swarm parallelism: Training large models can be surprisingly communication-efficient. *arXiv preprint arXiv:2301.11913*, 2023.

Scao, T. L., Fan, A., Akiki, C., Pavlick, E., Ilić, S., Hesslow, D., Castagné, R., Luccioni, A. S., Yvon, F., Gallé, M., et al. Bloom: A 176b-parameter open-access multilingual language model. *arXiv preprint arXiv:2211.05100*, 2022.

Shen, S., Dong, Z., Ye, J., Ma, L., Yao, Z., Gholami, A., Mahoney, M. W., and Keutzer, K. Q-bert: Hessian based ultra low precision quantization of bert. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pp. 8815–8821, 2020.

Steiner, B., Elhoushi, M., Kahn, J., and Hegarty, J. Olla: Optimizing the lifetime and location of arrays to reduce the memory usage of neural networks. 2022. doi: 10.48550/arXiv.2210.12924.

Wang, L., Ye, J., Zhao, Y., Wu, W., Li, A., Song, S. L., Xu, Z., and Kraska, T. Superneurons: Dynamic gpu memory management for training deep neural networks. In *Proceedings of the 23rd ACM SIGPLAN symposium on principles and practice of parallel programming*, pp. 41–53, 2018.

Wang, X., Xiong, Y., Wei, Y., Wang, M., and Li, L. Lightseq: A high performance inference library for transformers. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies: Industry Papers*, pp. 113–120, 2021.

Xiao, G., Lin, J., Seznec, M., Demouth, J., and Han, S. Smoothquant: Accurate and efficient post-training quantization for large language models. *arXiv preprint arXiv:2211.10438*, 2022.

Yao, Z., Aminabadi, R. Y., Zhang, M., Wu, X., Li, C., and He, Y. Zeroquant: Efficient and affordable post-training quantization for large-scale transformers. In Oh, A. H., Agarwal, A., Belgrave, D., and Cho, K. (eds.), *Advances in Neural Information Processing Systems*, 2022.

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

Zhang, S., Roller, S., Goyal, N., Artetxe, M., Chen, M., Chen, S., Dewan, C., Diab, M., Li, X., Lin, X. V., et al. Opt: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068*, 2022.

Zheng, L., Li, Z., Zhang, H., Zhuang, Y., Chen, Z., Huang, Y., Wang, Y., Xu, Y., Zhuo, D., Gonzalez, J. E., et al. Alpa: Automating inter-and intra-operator parallelism for distributed deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 2022.

A. Appendix

A.1. Notations

We use notations in Table 6 in this appendix.

Var	Meaning
l	number of layers in the model
s	prompt sequence length
n	output sequence length
bls	block size
h_1	hidden size
h_2	hidden size of the second MLP layer
nh	number of head in the model

Table 6. Notations

A.2. Compute Schedule Optimality

This subsection discusses the graph traversal problem described in Section 4.1 and only considers the case that the model cannot fit in a single GPU. We assume no application of CPU computation. To compute a square, the GPU loads the tensors it needs and offloads the cache and activations when finished. We will analyze two schedules: the zig-zag block schedule used in Section 4.2 and an I/O-optimal diagonal block schedule introduced in this section. Note that our analysis only considers the theoretical I/O complexity. In the real system, the latency and memory consumption cannot be the same as in the theoretical calculations.

There are three things that need to be stored during the generation process: weights, activations, and the KV cache. From the computational graph, we have three observations. (1) Suppose we need to swap the weights in and out of the GPU. Whatever the portion is, to finish the generation for one prompt, we need to swap n times for n tokens. Therefore, it would be preferable to reuse the loaded weights for a batch of prompts, amortizing the weights I/O time. (2) Each square will output activations which will be fed into the next layer. Each row in the computational graph only needs to hold activations for one square at the same time. (3) For each square besides the last l squares in a row, the KV cache dumped by the square cannot be released until generating the last token (the last l columns in the computational graph). It is not shared across rows or columns, which will be the major factor in limiting the batch size.

A.2.1. Zig-zag block schedule and Diagonal block schedule

Zig-zag block schedule. Inspired by the three observations introduced in Section 4.2, we compute the first column in the computational graph for bls samples, save the dumped caches and activations, then compute the second column for bls samples, until the last column for bls samples. We call bls as the block size as introduced in Section 4.2. The computed $bls \cdot n \cdot l$ squares are called a block.

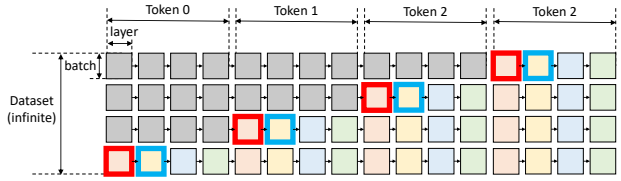


Figure 5. diagonal block schedule

Assume `FP16` precision, to generate $n \cdot bls$ tokens during one block computation, we have to load n times the whole model weights, do I/O operations on activations with $2(2h_1 \cdot s \cdot bls \cdot l + 2h_1 \cdot bls \cdot l \cdot (n-1))$ bytes in total, and do I/O on the KV cache with $4h_1 \cdot bls \cdot l \cdot (s \cdot n + n(n-1)/2)$ bytes in total.

Let w denote the size of one-layer weights. The peak memory used to store the weights, activations, and KV caches can be estimated as

$$\text{peak_mem} = w + 2h_1 \cdot bls + 4h_1 \cdot bls \cdot l \cdot (s+n)$$

If we only swap with CPU, then there is the constraint that peak_mem $<$ CPU memory - some overhead. Let $cmem$ denote the right hand, there is

$$bls \le \frac{cmem - w}{2h_1 + 4h_1 \cdot l \cdot (s+n)} = bls_1$$

Now we show that there is a better schedule that gives the same I/O efficiency but can enlarge the bls by around 2 in some cases.

Diagonal block schedule Figure 5 is an illustration of our diagonal block schedule. We have a block containing 4 GPU batches, and we are going to generate 4 tokens with a model that has 4 layers. There will be a one-time warm-up phase (gray area) to compute the area above the diagonal. Then for each iteration, the system will compute a diagonal that contains 4 sub-diagonals (4 squares enclosed by red outlines as the first sub-diagonal, then 4 squares enclosed by blue outlines as the second sub-diagonal). After finishing the 4 sub-diagonals, it will repeat the same computation in the next row.

For simplicity, consider the good case that the memory capacity is large enough that the diagonal can cover all n generation iterations for n tokens. The block size bls now is defined as the number of samples touched by the diagonal.

In total, to compute one diagonal, the weights of each layer will be loaded once, and the I/O of the activations and KV

cache will be in size roughly as $1/n$ as the value in the zig-zag block schedule. There will be bls tokens generated. So the I/O per token is the same with the zig-zag block schedule after the one-time warm-up if for the same bls.

The peak memory needed to hold the necessary weights, activations, and KV cache is estimated as

$$\text{peak_mem} = w + 2h_1 \cdot bls + \frac{4h_1 \cdot bls \cdot l(2s+n)(n-1)}{2n}$$

from peak_mem $\leq cmem$, we have

$$bls \leq \frac{n(cmem - w)}{2h_1 \cdot n + 2h_1 \cdot l \cdot (2s+n)(n-1)} = bls_2$$

Despite a one-time warm-up at the beginning. The diagonal block schedule can accommodate a larger block size than zig-zag block schedule at the ratio of

$$\frac{bls_2}{bls_1} = \frac{2s+2n}{2s+n} + O\left(\frac{1}{n}\right)$$

which is close to 2 when $n \gg s$, and close to 1 when $s \gg n$.

A larger bls does not change the activations and KV caches I/O per token, but can reduce the weights I/O per token proportionally, while weights I/O can normally occupy a large portion.

Discussions. In offloading setting, I/O is a significant bottleneck in latency and throughput, so the diagonal block schedule should be able to give considerable gain when n is relatively large compared to s and the memory is sufficiently large to fit n samples.

When the compute resources are sufficient to avoid offloading, the diagonal block schedule can still help to reduce the peak memory and enlarge the batch size, which increases GPU utilization.

Another benefit compared to the zig-zag block schedule is that with the same throughput, the generation latency for each prompt is reduced. For example, suppose in the zig-zag block schedule the bls samples finish the generation at the same time with latency T. In the diagonal block schedule, the first bls/n samples finish the generation with latency T/n, the second bls/n samples finish with latency $2T/n$, and so on. The average latency of completion is reduced by half.

Despite its advantages, there are some difficulties in implementing the diagonal block schedule. The major implementation difficulty is the dynamic update of the KV cache buffer. To improve runtime efficiency, FlexGen now pre-allocates continuous buffers for all KV cache at the beginning of a block. This works well for the zig-zag block schedule. However, for the diagonal block schedule, pre-allocating continuous buffers make it impossible to save memory anymore. To utilize the memory-saving property of the diagonal block schedule, one needs to implement efficient attention computation on non-contiguous memory.

A.2.2. Proof of Theorem 4.1

Note that in any case when we move from computing a square to another square, we need to offload and load the corresponding KV cache. So that the total I/O incurred by KV cache is constant. The total I/O incurred by activations could vary, but despite the prefill phase, its size for each square is much smaller than the KV cache for the same square. In total, the size of activations is around $1/(2s+n)$ of the size of KV cache. We will ignore the I/O incurred by activations for simplicity, which can cause a multiplicative error of $1/(2s+n)$ at most. Then the only thing left is the weights I/O. Starting from now, the I/O complexity in the context refers to the I/O complexity incurred by weights.

Definition A.1. We define the working state at any time when the GPU is computing a square as follows. Suppose there are k GPU batches working in progress. The column indices of the last squares that have been computed (including the current one) are $a_1, a_2, ..., a_k$, and $1 \leq a_i \leq n \times l$. Different batches are identically independent, so w.l.o.g., suppose $a_1 \geq a_2 \geq ... \geq a_k$. Then the working state is a tuple $(a_1, a_2, ..., a_k)$. A move that does a computation on a square is a pair of states $s^{(1)}, s^{(2)}$ that means transit from state $s^{(1)}$ to $s^{(2)}$.

Consider an optimal order denoted as an infinite sequence $m_1, m_2, ..., m_\infty$, where m_i is the ith move. For each i, let s_i be the current working state.

Lemma A.2. *If there is a list of moves that start from state s, and back to state s at the end, the number of computed squares for every column (one layer for one token) is the same.*

Proof. Suppose the start state $s = (a_1, a_2, ..., a_k)$. For computations that occupy the whole row, the number of computed squares for every column is the same. So we only need to consider the rows that have not been fully traversed (captured by the end state). For each a_i, if the underlying row has not been finished at the end, and ends with the index b_i, then we pair a_i with b_i. If the underlying row has been finished, we pair it with a newly opened but not finished row, still, let b_i denote the new index.

Thus we have transited from state $S_a = (a_1, a_2, ..., a_k)$ to another state $S_b = (b_1, b_2, ..., b_k)$. The indices in S_a are sorted by $a_1 \geq a_2 \geq ... \geq a_k$. The indices in S_b are not sorted, but b_i is paired to a_i according to the above

paragraph. For each i, if $b_i > a_i$, we need to count the squares in $(a_i, b_i]$ by 1. If $b_i < a_i$, we need to count the squares in $(b_i, a_i]$ by -1. Now we argue that for each column index j and $1 \leq j \leq n \times l$, the count over it is summed to 0. Suppose not, that there are p positive count and q negative count and $p \neq q$. Then there are p values lower than j in state a and q values lower than j in state b. This contradicts the fact that S_a and S_b are the same state with different orders. Therefore, the number of computed squares for every column is the same. □

Theorem A.3. *The diagonal block schedule is I/O-optimal asymptotically.*

Proof. Notice that since the memory capacity is finite, the length of the state is finite, thus the number of the possible state is finite. If each state appears finite times in the sequence, then the sequence cannot be infinite. Therefore, there exists a state s that appears in the sequence infinite times.

Let $j_1, j_2, ..., j_\infty$ be the indices in the sequence that have state s. The moves between each two neighboring s states correspond to a throughput. The moves between j_1 and j_2 should create the highest possible throughput that pushes from state s to s. Otherwise, we can replace it to get a higher total throughput, which contradicts to that it is an optimal order. So that we can repeat such a strategy between each neighboring j_i, j_{i+1} to get an optimal compute order.

Now the problem is reduced to finding an optimal compute order between j_1 and j_2. With infinite loops, the highest throughput from j_1 to j_2 gives the highest throughput among the whole sequence.

Assume an optimal compute order between j_1 and j_2. From Lemma A.2, there is the same number of squares to be computed for every column denoted as c. With such fixed c, the throughput is determined by the I/O time between j_1 and j_2. The number of times we load weights for each color in Figure 2 determines the total I/O time. Each time we load weights, for example, the weights for computing the yellow squares, we cannot compute two yellow squares in the same row without other weights swaps, because the squares between them have not been computed and require other weights.

Therefore, for one load, we can only compute squares from different rows, which means all the caches and activations corresponding to those squares need to be held (either on the CPU or on the disk). Every square corresponds to some memory consumption, for example, the squares in the range of the i-th token cost caches for $s + i - 1$ tokens. The sum of the memory consumption of all squares is a constant denoted as M. Let M' denote the memory capacity. The number of weights loading times is at least $\lceil M/M' \rceil$. Let t_w denote the I/O time for loading weights for one color, the optimal throughput is at most $c/\lceil M/M' \rceil/t_w$.

In the diagonal block schedule, after warm-up, each time with the loaded weights, the peak memory is the sum of the memory consumption of each computed square, which is the same each time we load weights. We can set it to hit M'[3]. Take c number of diagonals as the repeated list of moves denoted as $\vec{q}$. Set the starting state to be s mentioned before, $\vec{q}$ will restore the state to s by construction. The number of weights loading times during $\vec{q}$ is $\lceil M/M' \rceil$, which meets the lower bound, and achieves the throughput upper bound $c/\lceil M/M' \rceil/t_w$. The warm-up phase can be ignored in the setting of an infinite sequence. In summary, the diagonal block schedule is I/O optimal asymptotically. □

The zig-zag block schedule is not optimal, as the peak memory consumption is not the same each time loading the weights. When computing the layers for the last token, the peak memory is scaled with $s + n - 1$, while for the first token, it is scaled with s. In order to let the former fit in M', the latter must be smaller than M'. But the memory consumption change is linear when generating the tokens, thus the average memory consumption for each weights loading can be pushed to at least $M'/2$. From this, the zig-zag block schedule can achieve the throughput at least $c/\lceil M/(M'/2) \rceil/t_w$ which is $1/2$ of the throughput upper bound. In the infinite sequence setting, this means the zig-zag block schedule can achieve an I/O complexity that is at most $2\times$ optimal. Therefore, we have:

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

A.3. Cost Model

In this section, we present the full cost model. Note that we use a single variable to represent constants like bandwidth and TFLOPS to simplify the formulation below. In real systems, these constants vary according to the total load. We handle such dynamics by using piece-wise functions and adding regularization terms. We carefully model the dynamics by depending only on other constants (e.g., hidden size), so the optimization problem remains a linear programming problem with respect to policy variables.

Table 6 and Table 7 give the meaning of constants used in the cost model.

The object is to maximize throughput (token/s), which is equivalent to minimizing the reciprocal (s/token). Free variables are colored blue.

[3]The size value is discrete, we cannot exactly hit M', but with large enough parameters, such a gap could be set to only affect the total value by less than 1%. For example, the layer could be at the tensor level to make squares extremely fine-grained.

Var	Meaning
$ctog_bdw$	CPU to GPU bandwidth
$gtoc_bdw$	GPU to CPU bandwidth
$dtoc_bdw$	disk to CPU bandwidth
$ctod_bdw$	CPU to disk bandwidth
mm_flops	GPU flops per second for matrix multiplication
bmm_flops	GPU flops per second for batched matrix multiplication
cpu_flops	CPU flops per second
wg	percentage of weights on GPU
wc	percentage of weights on CPU
wd	percentage of weights on disk
cg	percentage of KV cache on GPU
cc	percentage of KV cache on CPU
cd	percentage of KV cache on disk
hg	percentage of activations on GPU
hc	percentage of activations on CPU
hd	percentage of activations on disk

Table 7. Notation Variables

Objective

$$\text{Minimize} \quad T/bls$$

Then the following constraints describe the calculation of total latency:

$$T = Tpre \cdot l + Tgen \cdot (n-1) \cdot l$$

$$Tpre = \max(ctog^p, gtoc^p, dtoc^p, ctod^p, comp^p)$$

$$\begin{aligned} ctog^p &= \frac{weights_ctog^p + act_ctog^p}{ctog_bdw} \\ &= \frac{1}{ctog_bdw}((wc+wd)(8h_1^2+4h_1\cdot h_2) \\ &\quad + 2(hc+hd)s\cdot h_1 \cdot bls) \end{aligned}$$

$$\begin{aligned} gtoc^p &= \frac{cache_gtoc^p + act_gtoc^p}{gtoc_bdw} \\ &= \frac{1}{gtoc_bdw}(4(cc+cd)(s+1)h_1\cdot bls \\ &\quad + 2(hc+hd)s\cdot h_1\cdot bls) \end{aligned}$$

$$\begin{aligned} dtoc^p &= \frac{weights_dtoc^p + act_dtoc^p}{dtoc_bdw} \\ &= \frac{1}{dtoc_bdw}(wd(8h_1^2+4h_1\cdot h_2) \\ &\quad + 2hd\cdot s\cdot h_1\cdot bls) \end{aligned}$$

$$\begin{aligned} ctod^p &= \frac{cache_ctod^p + act_ctod^p}{ctod_bdw} \\ &= \frac{1}{ctod_bdw}(4cd\cdot bls\cdot(s+1)\cdot h_1 \\ &\quad + 2hd\cdot s\cdot h_1\cdot bls) \end{aligned}$$

$$\begin{aligned} comp^p &= \frac{linear_layer^p}{mm_flops} + \frac{att^p}{bmm_flops} \\ &= \frac{bls(8s\cdot h_1^2 + 4s\cdot h_1\cdot h_2)}{mm_flops} \\ &\quad + \frac{4bls\cdot s^2\cdot h_1}{bmm_flops} \end{aligned}$$

$$Tgen = \max(ctog^g, gtoc^g, dtoc^g, ctod^g, comp^g)$$

$$\begin{aligned} ctog^g &= \frac{weights_ctog^g + act_ctog^g}{ctog_bdw} \\ &= \frac{1}{ctog_bdw}((wc+wd)(8h_1^2+4h_1\cdot h_2) \\ &\quad + 2(hc+hd)h_1\cdot bls) \end{aligned}$$

$$\begin{aligned} gtoc^g &= \frac{act_gtoc^g}{gtoc_bdw} \\ &= \frac{1}{gtoc_bdw}(2(hc+hd)\cdot h_1\cdot bls) \end{aligned}$$

$$\begin{aligned} dtoc^g &= \frac{cache_dtoc^g + weights_dtoc^g + act_dtoc^g}{dtoc_bdw} \\ &= \frac{1}{dtoc_bdw}(4cd\cdot bls\cdot(s+n/2)\cdot h_1 \\ &\quad + wd(8h_1^2+4h_1\cdot h_2) \\ &\quad + 2hd\cdot h_1\cdot bls) \end{aligned}$$

$$\begin{aligned} ctod^g &= \frac{cache_ctod^g + act_ctod^g}{ctod_bdw} \\ &= \frac{1}{ctod_bdw}(4cd\cdot bls\cdot h_1 + 2hd\cdot h_1\cdot bls) \end{aligned}$$

$$comp^g = gpu_comp^g + cpu_comp^g$$

$$\begin{aligned} gpu_comp^g &= \frac{linear_layer^g}{mm_flops} + \frac{att^g}{bmm_flops} \\ &= \frac{bls(8h_1^2+4h_1\cdot h_2)}{mm_flops} \\ &\quad + \frac{4cg\cdot bls\cdot(s+n/2)\cdot h_1}{bmm_flops} \end{aligned}$$

$$\begin{aligned} cpu_comp^g &= \frac{att^g}{cpu_flops} \\ &= \frac{4(cc+cd)bls\cdot(s+n/2)\cdot h_1}{cpu_flops} \end{aligned}$$

Peak Memory Constraints

- GPU peak memory constraints during prefill:
  GPU memory used to hold a fixed percentage of weights, activations, and cache is

$$\begin{aligned} gpu_home^p = {} & wg \cdot (8h_1^2 + 4h_1 \cdot h_2) \cdot l \\ & + hg \cdot 2s \cdot h_1 \cdot bls \\ & + 4(s+n)h_1 \cdot cg \cdot bls \cdot l. \end{aligned}$$

  GPU working memory (omit mask):

$$\begin{aligned} qkv^p &= gbs \cdot (2s \cdot h_1 + 3(2s \cdot h_1)) \\ &= gbs \cdot 8s \cdot h_1 \\ att_1^p &= cg \cdot gbs \cdot (2s \cdot h_1 + 2s \cdot h_1 + 2nh \cdot s^2) \\ att_2^p &= cg \cdot gbs \cdot (2nh \cdot s^2 + 2s \cdot h_1 + 2s \cdot h_1) \\ embed^p &= gbs \cdot (2s \cdot h_1 + 2s \cdot h_1) \\ &= gbs \cdot 4s \cdot h_1 \\ mlp_1^p &= gbs \cdot 2(s \cdot h_1 + s \cdot h_2) \\ &= 2gbs \cdot s(h_1 + h_2) \\ mlp_2^p &= gbs \cdot 2(s \cdot h_2 + s \cdot h_1) \\ &= 2gbs \cdot s(h_1 + h_2) \end{aligned}$$

$$\begin{aligned} gpu_w^p = {} & 2(1 - wg)(8h_1^2 + 4h_1 \cdot h_2) \\ & + (1 - hg) \cdot 2s \cdot h_1 \cdot gbs \\ & + \max(qkv, att_1, att_2, embed, mlp_1, mlp_2) \end{aligned}$$

$$gpu_peak^p = gpu_home^p + gpu_w^p < gmem$$

- GPU peak memory constraints after prefill:
  GPU memory used to hold a fixed percentage of weights, activations, and cache is

$$\begin{aligned} gpu_home^g = {} & wg \cdot (8h_1^2 + 4h_1 \cdot h_2) \cdot l \\ & + hg \cdot 2h_1 \cdot bls \\ & + 4(s+n)h_1 \cdot cg \cdot bls \cdot l. \end{aligned}$$

  GPU working memory (omit mask):

$$\begin{aligned} qkv^g &= gbs \cdot (2h_1 + 3(2h_1)) = 8gbs \cdot h_1 \\ att_1^g &= cg \cdot gbs \cdot (2h_1 + 2(s+n)h_1 \\ &\qquad + 2nh(s+n)) \\ att_2^g &= cg \cdot gbs \cdot (2nh(s+n) + 2(s+n)h_1 \\ &\qquad + 2h_1) \\ embed^g &= gbs \cdot (2h_1 + 2h_1) = 4gbs \cdot h_1 \\ mlp_1^g &= 2gbs \cdot (h_1 + h_2) \\ mlp_2^g &= 2gbs \cdot (h_2 + h_1) \end{aligned}$$

$$\begin{aligned} gpu_w^g = {} & 2(1 - wg)(8h_1^2 + 4h_1 \cdot h_2) \\ & + (1 - hg) \cdot 2s \cdot h_1 \cdot gbs \\ & + \max(qkv^g, att_1^g, att_2^g, embed^g, mlp_1^g, mlp_2^g) \end{aligned}$$

$$gpu_peak^g = gpu_home^g + gpu_w^g < gmem$$

- CPU peak memory constraints during prefill:
  CPU memory used to hold a fixed percentage of weights, activations, and cache is

$$\begin{aligned} cpu_home^p = {} & wc \cdot (8h_1^2 + 4h_1 \cdot h_2) \cdot l \\ & + hc \cdot 2s \cdot h_1 \cdot bls \\ & + 4(s+n)h_1 \cdot cc \cdot bls \cdot l. \end{aligned}$$

  CPU working memory:

$$\begin{aligned} cpu_w^p = {} & (1 - wg)(8h_1^2 + 4h_1 \cdot h_2) \\ & + (1 - hg) \cdot 2s \cdot h_1 \cdot gbs. \end{aligned}$$

$$cpu_peak^p = cpu_home^p + cpu_w^p < cmem$$

- CPU peak memory constraints after prefill:
  CPU memory used to hold fixed percentage of weights, activations, and cache is

$$\begin{aligned} cpu_home^g = {} & wc \cdot (8h_1^2 + 4h_1 \cdot h_2) \cdot l \\ & + hc \cdot 2h_1 \cdot bls \\ & + 4(s+n)h_1 \cdot cc \cdot bls \cdot l. \end{aligned}$$

  CPU working memory:

$$\begin{aligned} cpu_w^g = {} & wd(8h_1^2 + 4h_1 \cdot h_2) \\ & + 2hd \cdot 2 \cdot h_1 \cdot gbs \\ & + 2cd \cdot 4(s+n)h_1 \cdot gbs \\ & + 2nh \cdot (s+n) \cdot gbs \\ & + 2h_1 \cdot gbs. \end{aligned}$$

$$cpu_peak^g = cpu_home^g + cpu_w^g < cmem$$

- NVMe peak memory constraints:

$$\begin{aligned} nvme_peak = {} & (8h_1^2 + 4h_1 \cdot h_2) \cdot wd \cdot l \\ & + hd \cdot 2s \cdot h_1 \cdot bls \\ & + cd \cdot 4(s+n)h_1 \cdot bls \cdot l \\ & < nmem \end{aligned}$$

A.4. Tables and Additional Experimental Results

Execution Breakdown Table 8 shows the execution time breakdown for OPT-175B running on FlexGen with the setup in Table 1.

HELM and Data Wrangling Table 9 lists the details of HELM integration experiments. Table 10 and Table 11 shows additional results for the data wrangling task.

Complementary Tables for Policy Details Table 15 and Table 16 list the concrete policy setups for the results in Table 2 for prompt length 512 and 1024, from end-to-end throughput experiments. Table 19 and Table 20 list the latency and throughput for the data points in Fig. 1 which demonstrate latency-throughput tradeoff.

Abalation Study Table 23 list the concrete policy setups for the main ablation study result in Table 4. Table 21 and Table 22 shows some additional ablation study on policies. In Table 23, DeepSpeed chooses to store the KV cache and activations on GPU. For OPT-30B, the weights will be stored on the CPU entirely because it cannot fit in GPU. The corresponding percentage is $(0, 100, 100, 0, 100, 0)$. The computation order of DeepSpeed is row-by-row, so the number of GPU batches in a block is 1. The GPU batch size is set to be as large as possible, which is set to 8. For OPT-175B, the weights will be stored on disk entirely according to DeepSpeed's strategy, since it cannot be stored on CPU. The corresponding percentage is $(0, 0, 100, 0, 100, 0)$. The number of GPU batches in a block is 1, and the GPU batch size is 2. For "No policy search", we use different policy changes for OPT-30B and OPT-175B to demonstrate the impact of different policy dimensions. For OPT-30B, we change the percentage for weights from (20, 80) to (0, 100), and show that the throughput does not change much. For OPT-175B, we change the number of GPU batches in a block from 8 to 1 and show that the throughput degrades significantly. For "No CPU compute", it degrades OPT-30B more than OPT-175B because the bottleneck for OPT-175B is on disk offloading. Therefore, the gain for CPU computation is small for OPT-175B. While for OPT-30B, the disk has not been used, so the gain for CPU computation is more significant.

Different SSD Speed To highlight the limitation and requirements of SSD speed. We tested two kinds of disk on GCP and report the generation throughput (token/s) in Table 24 (input sequence length = 512 and output sequence length = 32).

Additional Hardware and Sequence Length Our methods and implementations do not depend on specific hardware architectures. It can work well on different CPU architectures (e.g., Intel, AMD) and different GPU architectures (e.g., NVIDIA Ampere, NVIDIA Turing) as long as the architectures are supported by PyTorch. Some architecture (e.g. unified memory) could be more friendly to our approach. To tune the system for different architectures, we need to fit a cost model and run policy search to generate offloading policies, which can be different according to the compute capabilities, memory capacities, and memory bandwidth of different architectures. The final absolute performance will vary, but FlexGen can be easily adapted to different architectures. We did additional experiments on a different hardware setup of 24GB RTX 3090 with 125GB CPU Memory and 1TB SSD, in addition to our previous setting of 16GB T4 with 208GB CPU Memory and 1.5TB SSD, shown in Table 12. The input sequence length is set to 512 and the output sequence length is set to 32. We can see the results follow similar trends to the setup in the main paper. FlexGen outperforms other baselines significantly. Comparing this 3090 setting with the T4 setting in the main paper, the performance under the 3090 setting is worse than the T4 setting for 30B and 175B. This is because CPU memory also plays a critical role when offloading is needed, making our T4 setting with larger CPU memory better.

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

Table 14 and Table 13 show the results for an additional prompt length 256. As all of our benchmarks in the main paper are done with output sequence length 32, so we add two additional fixed sequence lengths in Table 17 and Table 18. The numbers are generally higher in the former one because the input sequence length is smaller and the output sequence length is larger. As the throughput is defined as (number of generated tokens) / (prefill time + generation time), such a setting makes the fraction of prefill time smaller. The numbers are generally lower in the latter one because the output sequence length is smaller.

In summary, FlexGen outperforms baselines in all newly added settings. The Compression techniques used in FlexGen are helpful only for large models that need offloading. CPU memory capacity is essential for large models that need offloading.

Batches with Various Sequence Length We also add experiments of one realistic use case with a mixture of prompt and output lengths (HELM benchmark) in Table 25. To batch sequences of variable lengths, FlexGen simply pads all inputs to the maximum prompt length, which is a common method used in many systems. Depending on the distribution of the prompt length, the efficiency of this simple padding method varies. For example, if most sequences have similar lengths, then the baching efficiency should be very high. if some sequences are very long and some sequences are short, then FlexGen will spend a lot of time on the useless computation of padding tokens. We use two metrics: padded throughput = (number of tokens in padded prompts + number of tokens in padded outputs) / latency and actual throughput = (number of non-padding tokens in prompts + number of non-padding tokens in outputs) / latency. To better handle prompts with various lengths, one can utilize some complementary techniques from Orca(Yu et al., 2022).

Table 9. The setup and running time of 7 representative sub-scenarios in the HELM integration. The running time consists of dataset downloading, model initialization, generation, and metric computation. "Prompt len" denotes the input sequence length, and "Gen len" denotes the output sequence length. "Num seq" denotes the number of sequences (prompts). "time" denotes the running time in minutes.

Scenario description	Prompt len	Gen len	Num seq	time
wikifact: k=5, subject=plaintiff	256	8	288	10
wikifact: k=5, subject=instance_of	256	8	2592	55
mmlu: subject=abstract_algebra	512	1	864	31
mmlu: subject=us_foreign_policy	512	1	1008	33
synthetic_reasoning: mode=pattern_match	256	50	1584	118
synthetic_reasoning_natural: difficulty=easy	512	20	1584	100
summarization_xsum: temperature=0.3	1984	64	1568	902

Table 10. The setup and running time of 6 representative data wrangling tasks with OPT-30B. Because the output seq. length is short for this task, we use a new metric total throughput = (number of tokens in the prompt + number of generated tokens) / total latency.

Task	Number of seq.	Input seq. length	Output seq. length	Running time (s)	Total throughput (token/s)
EM: Fodors-Zagats	189	744	3	541.550	248.287
EM: Beer	91	592	3	238.58	224.450
EM: iTunes-Amazon	109	529	3	267.639	198.775
DI: Restaurant	86	123	5	60.310	169.790
DI: Buy	65	488	10	185.882	160.747
ED: Hospital	200	200	3	158.329	256.429

Table 11. The setup and running time of 6 representative data wrangling tasks with OPT-175B. Because the output seq. length is short for this task, we use a new metric total throughput = (number of tokens in the prompt + number of generated tokens) / total latency.

Task	Number of seq.	Input seq. length	Output seq. length	Running time (s)	Total throughput (token/s)
EM: Fodors-Zagats	189	744	3	3928.310	34.228
EM: Beer	91	592	3	1356.786	35.083
EM: iTunes-Amazon	109	529	3	1569.062	33.906
DI: Restaurant	86	123	5	648.762	16.968
DI: Buy	65	488	10	2086.961	14.317
ED: Hospital	200	200	3	1154.133	35.178

Table 12. Generation throughput (token/s) on 1 GPU (RTX 3090) with 125 GB CPU memory and 1TB SSD, run with **input sequence length 512 and output sequence length 32**. FlexGen is our system without compression; FlexGen (c) uses 4-bit compression. The gray tuple denotes a policy (GPU batch size $\times$ #GPU-batch, wg, wc, cg, cc, hg, hc).

Seq. length	512 + 32		
Model size	6.7B	30B	175B
Accelerate	183.177 (16×1, 100, 0, 100, 0, 100, 0)	2.077 (13×1, 0, 100, 100, 0, 100, 0)	0.026 (4×1, 0, 0, 100, 0, 100, 0)
DeepSpeed	38.027 (32×1, 0, 100, 100, 0, 100, 0)	3.889 (12×1, 0, 100, 100, 0, 100, 0)	0.019 (3×1, 0, 0, 100, 0, 100, 0)
FlexGen	233.756 (28×1, 100, 0, 100, 0, 100, 0)	5.726 (4×15, 25, 75, 40, 60, 100, 0)	0.384 (64×4, 0, 25, 0, 0, 100, 0)
FlexGen (c)	120.178 (144×1, 100, 0, 100, 0, 100, 0)	16.547 (96×2, 25, 75, 0, 100, 100, 0)	1.114 (24×1, 0, 100, 0, 100, 100, 0)

Table 13. Generation throughput (token/s) on 1 GPU with different systems. Accelerate, DeepSpeed, and FlexGen use 1 GPU. Petals uses 1 GPU for OPT-6.7B, 4 GPUs for OPT-30B, and 24 GPUs for OPT-175B, but reports per-GPU throughput. Petals is benchmarked under different network delay and bandwidth. The models are run in INT8 as the default for Petals. We tune the batch size of each request to be 2 and issue requests by 6 parallel client processes to achieve the maximum throughput. FlexGen is our system without compression; FlexGen (c) uses 4-bit compression. "OOM" means out-of-memory.

Seq. length	256			512			1024		
Model size	6.7B	30B	175B	6.7B	30B	175B	6.7B	30B	175B
Accelerate	50.66	1.34	0.02	25.12	0.62	0.01	13.01	0.31	0.01
DeepSpeed	14.52	1.30	0.01	9.28	0.60	0.01	4.59	0.29	OOM
Petals (<5ms, 1Gb/s)	9.03	3.55	0.09	8.25	2.84	0.08	6.56	1.51	0.06
Petals (<5ms, 100Mb/s)	9.15	2.53	0.06	8.18	1.67	0.05	6.52	0.87	0.03
Petals (100ms, 100Mb/s)	8.64	0.75	0.01	7.82	0.64	0.01	5.89	0.37	0.01
FlexGen	53.29	16.01	1.36	25.26	7.32	0.69	13.72	3.50	0.35
FlexGen (c)	56.72	16.86	2.26	29.12	8.70	1.12	13.18	3.98	0.42

Table 14. Generation throughput (token/s) on 1 GPU with **input sequence length 256 and output sequence length 32**. FlexGen is our system without compression; FlexGen (c) uses 4-bit compression. "OOM" means out-of-memory. The gray tuple denotes a policy (GPU batch size $\times$ #GPU-batch, wg, wc, cg, cc, hg, hc).

Seq. length	256		
Model size	6.7B	30B	175B
Accelerate	50.66 (4×1, 100, 0, 100, 0, 100, 0)	1.34 (16×1, 0, 100, 100, 0, 100, 0)	0.02 (4×1, 0, 0, 100, 0, 100, 0)
DeepSpeed	14.52 (32×1, 0, 100, 100, 0, 100, 0)	1.30 (12×1, 0, 100, 100, 0, 100, 0)	0.01 (2×1, 0, 0, 100, 0, 100, 0)
FlexGen	53.29 (4×1, 100, 0, 100, 0, 100, 0)	16.01 (160×2, 10, 90, 0, 100, 0, 100)	1.36 (64×8, 0, 50, 0, 0, 0, 100)
FlexGen (c)	56.72 (128×1, 100, 0, 100, 0, 100, 0)	16.86 (128×8, 0, 100, 0, 100, 0, 100)	2.26 (96×3, 0, 100, 0, 100, 0, 100)

Table 15. Generation throughput (token/s) on 1 T4 GPU with **input sequence length 512 and output sequence length 32**. FlexGen is our system without compression; FlexGen (c) uses 4-bit compression. "OOM" means out-of-memory. The gray tuple denotes a policy (GPU batch size $\times$ #GPU-batch, wg, wc, cg, cc, hg, hc).

Seq. length	512		
Model size	6.7B	30B	175B
Accelerate	25.12 (2×1, 100, 0, 100, 0, 100, 0)	0.62 (8×1, 0, 100, 100, 0, 100, 0)	0.01 (2×1, 0, 0, 100, 0, 100, 0)
DeepSpeed	9.28 (16×1, 0, 100, 100, 0, 100, 0)	0.60 (4×1, 0, 100, 100, 0, 100, 0)	0.01 (1×1, 0, 0, 100, 0, 100, 0)
FlexGen	25.26 (2×1, 100, 0, 100, 0, 100, 0)	7.32 (48×3, 20, 80, 0, 100, 0, 100)	0.69 (32×8, 0, 50, 0, 0, 0, 100)
FlexGen (c)	29.12 (72×1, 100, 0, 100, 0, 100, 0)	8.70 (16×20, 20, 80, 0, 100, 100, 0)	1.12 (48×3, 0, 100, 0, 100, 0, 100)

Table 16. Generation throughput (token/s) on 1 T4 GPU with **input sequence length 1024 and output sequence length 32**. FlexGen is our system without compression; FlexGen (c) uses 4-bit compression. "OOM" means out-of-memory. The gray tuple denotes a policy (GPU batch size $\times$ #GPU-batch, wg, wc, cg, cc, hg, hc).

Seq. length	1024		
Model size	6.7B	30B	175B
Accelerate	13.01 (1×1, 100, 0, 100, 0, 100, 0)	0.31 (4×1, 0, 100, 100, 0, 100, 0)	0.01 (1×1, 0, 0, 100, 0, 100, 0)
DeepSpeed	4.59 (8×1, 0, 100, 100, 0, 100, 0)	0.29 (2×1, 0, 100, 100, 0, 100, 0)	OOM
FlexGen	13.72 (1×1, 100, 0, 100, 0, 100, 0)	3.50 (20×4, 4, 96, 0, 100, 0, 100)	0.35 (12×12, 0, 50, 0, 0, 0, 100)
FlexGen (c)	13.18 (28×1, 100, 0, 100, 0, 100, 0)	3.98 (20×12, 0, 100, 0, 100, 0, 100)	0.42 (12×4, 0, 100, 0, 100, 0, 100)

Table 17. Generation throughput (token/s) on 1 T4 GPU with **input sequence length 128 and output sequence length 128**. FlexGen is our system without compression; FlexGen (c) uses 4-bit compression. "OOM" means out-of-memory. The gray tuple denotes a policy (GPU batch size × #GPU-batch, wg, wc, cg, cc, hg, hc).

Seq. length	128 + 128		
Model size	6.7B	30B	175B
Accelerate	73.411 (5×1, 100, 0, 100, 0, 100, 0)	1.547 (16×1, 0, 100, 100, 0, 100, 0)	0.021 (4×1, 0, 0, 100, 0, 100, 0)
DeepSpeed	19.193 (36×1, 0, 100, 100, 0, 100, 0)	1.717 (12×1, 0, 100, 100, 0, 100, 0)	0.024 (3×1, 0, 0, 100, 0, 100, 0)
FlexGen	106.404 (7×1, 100, 0, 100, 0, 100, 0)	24.634 (32×10, 25, 75, 0, 100, 100, 0)	2.409 (64×8, 0, 50, 0, 0, 0, 100)
FlexGen (c)	92.568 (196×1, 100, 0, 100, 0, 100, 0)	39.141 (128×8, 25, 75, 0, 100, 0, 100)	4.264 (80×3, 0, 100, 0, 100, 100, 0)

Table 18. Generation throughput (token/s) on 1 T4 GPU with **input sequence length 512 and output sequence length 8**. FlexGen is our system without compression; FlexGen (c) uses 4-bit compression. "OOM" means out-of-memory. The gray tuple denotes a policy (GPU batch size × #GPU-batch, wg, wc, cg, cc, hg, hc).

Seq. length	512 + 8		
Model size	6.7B	30B	175B
Accelerate	17.290 (2×1, 100, 0, 100, 0, 100, 0)	0.628 (7×1, 0, 100, 100, 0, 100, 0)	0.009 (2×1, 0, 0, 100, 0, 100, 0)
DeepSpeed	9.055 (18×1, 0, 100, 100, 0, 100, 0)	0.872 (6×1, 0, 100, 100, 0, 100, 0)	0.007 (1×1, 0, 0, 100, 0, 100, 0)
FlexGen	16.425 (2×1, 100, 0, 100, 0, 100, 0)	3.938 (512×8, 20, 80, 0, 100, 0, 100)	0.451 (32×8, 0, 50, 0, 0, 0, 100)
FlexGen (c)	14.244 (76×1, 100, 0, 100, 0, 100, 0)	4.019 (16×36, 25, 75, 0, 100, 0, 100)	0.559 (48×3, 0, 100, 0, 100, 0, 100)

Table 19. The Pareto frontier of the latency-throughput trade-off of OPT-175B. The numbers are generation throughput (token/s) and effective batch latency (s) on 1 GPU with **input sequence length 512 and output sequence length 32**. The numbers in the parentheses are corresponding effective batch sizes. The numbers in bold are the best throughput and latency for each model. We organize the table so that the latency numbers of different methods in each row are similar for each model. The top value of each column corresponds to the setting of effective batch size 1. (To reach the lowest latency, FlexGen uses an effective batch size of 2 rather than 1 because the latency difference between batch sizes 1 and 2 is negligible in this case. So, a run with batch size 2 dominates the one with batch size 1 with higher throughput and similar latency.)

175B (generation throughput / latency)			
Accelerate	DeepSpeed	FlexGen	FlexGen (c)
-	-	-	0.052 / **612** (1)
-	-	-	0.198 / 647 (4)
-	-	-	0.369 / 693 (8)
-	-	-	0.779 / 1973 (48)
-	-	0.025 / 2555 (2)	1.092 / 2813 (96)
-	-	0.254 / 4028 (32)	**1.122** / 4072 (144)
-	0.006 / 5024 (1)	0.421 / 4864 (64)	-
-	-	0.572 / 7159 (128)	-
0.004 / 7508 (1)	-	-	-
0.008 / 7633 (2)	-	-	-
-	-	0.687 / 11916 (256)	-

Table 20. The Pareto frontier of the latency-throughput trade-off of OPT-30B. The numbers are generation throughput (token/s) and effective batch latency (s) on 1 GPU with **input sequence length 512 and output sequence length 32**. The numbers in the parentheses are corresponding effective batch sizes. The numbers in bold are the best throughput and latency for each model. We organize the table so that the latency numbers of different methods in each row are similar for each model. The top value of each column corresponds to the setting of effective batch size 1.

30B (generation throughput / latency)			
Accelerate	DeepSpeed	FlexGen	FlexGen (c)
-	-	-	0.21 / **153** (1)
-	-	-	0.42 / 154 (2)
-	-	0.20 / 159 (1)	0.82 / 155 (4)
-	-	0.37 / 172 (2)	1.58 / 162 (8)
-	-	0.73 / 174 (4)	2.88 / 178 (16)
-	0.16 / 203 (1)	1.40 / 183 (8)	-
-	0.31 / 204 (2)	2.70 / 190 (16)	-
-	0.62 / 206 (4)	4.05 / 253 (32)	4.63 / 277 (40)
0.08 / 405 (1)	-	5.71 / 359 (64)	6.72 / 381 (80)
0.31 / 408 (4)	-	-	-
0.62 / 413 (8)	-	-	-
-	-	7.32 / 559 (144)	-
-	-	-	7.96 / 644 (160)
-	-	-	8.49 / 904 (240)
-	-	-	**8.70** / 1177 (320)

Table 21. Ablation study of policies. The numbers correspond to generation **throughput** on 1 GPU with **input sequence length 512 and output sequence length 32**. All policies have CPU computation turned on. The numbers for OPT-175B show some inconsistency with the end-to-end evaluation in Table 2 and Table 15 (0.49 vs 0.69) because we turn on the pagecache-mangagement (Morton, 2008) tool to prevent the automatic disk cache in operating systems, which makes the ablation results more accurate but brings some overheads. This added some overhead and misses the advantage of using CPU cache. A real run should be expected to have a better throughput. (gbs denotes the GPU batch size, $\#gb$ denotes the number of GPU batches in a block.)

gbs	$\#gb$	wg	wc	cg	cc	hg	hc	30B (token/s)	175B (token/s)
48	3	20	80	0	100	0	100	**7.32**	OOM
48	3	0	100	0	100	0	100	7.26	OOM
48	1	20	80	0	100	0	100	5.40	OOM
32	8	0	50	0	0	0	100	1.66	**0.49**
32	8	0	0	0	0	0	100	1.55	0.44
32	1	0	50	0	0	0	100	0.88	0.23
1	1	20	80	100	0	100	0	0.20	OOM
1	1	0	50	100	0	100	0	0.04	0.01
8	1	0	100	100	0	100	0	1.57	OOM
2	1	0	0	100	0	100	0	0.05	0.01

Table 22. Ablation study of policies. The numbers are full generation **latency** on 1 GPU with **input sequence length 512 and output sequence length 32**. All policies have CPU computation turned on. We turn on the pagecache-mangagement (Morton, 2008) tool to prevent the automatic disk cache in operating systems, which makes the ablation results more accurate but brings some overheads. This added some overhead and misses the advantage of using CPU cache. A real run should be expected to have a better latency. (gbs denotes the GPU batch size, $\#gb$ denotes the number of GPU batches in a block.)

gbs	$\#gb$	wg	wc	cg	cc	hg	hc	30B (s)	175B (s)
48	3	20	80	0	100	0	100	559	OOM
48	3	0	100	0	100	0	100	635	OOM
48	1	20	80	0	100	0	100	284	OOM
32	8	0	50	0	0	0	100	4930	16611
32	8	0	0	0	0	0	100	5287	18704
32	1	0	50	0	0	0	100	1164	4476
1	1	20	80	100	0	100	0	**160**	OOM
1	1	0	50	100	0	100	0	737	**3107**
8	1	0	100	100	0	100	0	170	OOM
2	1	0	0	100	0	100	0	1215	6072

Table 23. Ablation study of proposed techniques. The numbers are generation throughput on 1 T4 GPU with prompt length 512 and generating length 32. The gray tuple denotes a policy (GPU batch size $\times$ #GPU-batch, wg, wc, cg, cc, hg, hc).

Model size	30B	175B
All optimizations	7.32 (48×3, 20, 80, 0, 100, 0, 100)	0.69 (32×8, 0, 50, 0, 0, 0, 100)
No policy search	7.26 (48×3, 0, 100, 0, 100, 0, 100)	0.27 (32×1, 0, 50, 0, 0, 0, 100)
No overlapping	5.86 (48×3, 20, 80, 0, 100, 0, 100)	0.59 (32×8, 0, 50, 0, 0, 0, 100)
No CPU compute	4.03 (48×3, 20, 80, 0, 100, 0, 100)	0.62 (32×8, 0, 50, 0, 0, 0, 100)
No disk	7.32 (48×3, 20, 80, 0, 100, 0, 100)	OOM
w/ DeepSpeed policy	1.57 (8×1, 0, 100, 100, 0, 100, 0)	0.01 (2×1, 0, 0, 100, 0, 100, 0)

Table 24. Generation throughput (token/s) on hardware specifed in Table 1 with **input sequence length 512 and output sequence length 32**. The performance of OPT-30B is not affected because OPT-30B does not use SSD. The disk speed is measured using the Linux command dd with a block size (bs) of 1MB and the number of blocks (count) of 16000. The PageCacheManagement tool is used to disable disk cache in the operating system during measurement.

Disk Specification	30B	175B
1.6GB/s read, 1.3GB/s write (local SSD, the one used in the main paper)	7.32	0.69
0.5GB/s read, 0.5GB/s write (persistent SSD, a new setting)	7.32	0.30
1.6GB/s read, 1.3GB/s write (local SSD, use PageCacheManagement)	7.32	0.49
0.5GB/s read, 0.5GB/s write (persistent SSD, use PageCacheManagement)	7.32	0.292

Table 25. Selected example of FlexGen on real-world tasks from the HELM benchmark, which consists of prompts of various lengths with different output lengths. We use two metrics: padded throughput = (number of tokens in padded prompts + number of tokens in padded outputs) / latency, actual throughput = (number of non-padding tokens in prompts + number of non-padding tokens in outputs) / latency. The throughput are measured in token/s. To batch sequences of variable lengths, FlexGen simply pads all inputs to the maximum prompt length, which is a common method used in many systems. Depending on the distribution of the prompt length, the efficiency of this simple padding method varies. For example, if most sequences have similar lengths, then the batching efficiency should be very high. if some sequences are very long and some sequences are short, then FlexGen will spend a lot of time on the useless computation of padding tokens.

Task	Padded input seq. length	Padded output seq. length	Padded throughput	Actual throughput	Efficiency
MMLU (abstract_algebra)	512	1	251.5	188.6	75.0%
xsum	1984	64	60.5	47.6	78.7%