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

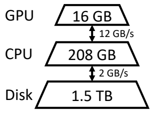
The high computational and memory requirements of large language model (LLM) inference traditionally make it feasible only with multiple high-end accelerators. In this paper, we study how to lower the requirements of LLM inference down to one commodity GPU. We present FlexGen, a high-throughput generation 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. Through a linear programming optimizer, it searches for the best pattern to store and access the tensors, including weights, activations, and attention key/value (KV) cache. FlexGen further compresses both weights and KV cache to 4 bits with negligible accuracy loss. These techniques enable FlexGen to use a much larger batch size and significantly increase throughput by trading off latency. As a result, when running OPT-175B on a single 16GB GPU, FlexGen achieves up to 100× higher throughput compared to state-of-the-art offloading systems and reaches a generation speed of 1 token/s for the first time. If more distributed GPUs are given, FlexGen also comes with a pipeline parallelism runtime to allow super-linear scaling on decoding. The code is available at `https://github.com/FMInference/FlexGen`.

1. Introduction

In recent years, large language models (LLMs) have shown superior 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 the unprecedented capabilities of LLMs, come their unique challenges for inference. 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 for running them. For example, GPT-175B requires 325GB of memory just for storing the model weights. To fit this model into accelerators, one would need at least five A100 (80GB) GPUs and complex parallelism strategies (Pope et al., 2022; Aminabadi et al., 2022).

Lowering the resource requirements of LLM inference has attracted intensive interest recently. These efforts fit into three directions: (1) *model compression* to decrease the total memory footprint (Dettmers et al., 2022; Yao et al., 2022; Frantar et al., 2022; Xiao et al., 2022); (2) *collaborative inference* to amortize the cost via decentralization (Borzunov et al., 2022); and (3) *offloading* to utilize the memory from CPUs and disks (Aminabadi et al., 2022; HuggingFace, 2022). These techniques have significantly lowered the resource requirements for using LLMs. However, they often assume that the model fits into the GPU memory while existing offloading-based systems still struggle to run 175B-scale models at acceptable throughput with a single GPU.

In this paper, we focus on generative inference with limited GPU memory in *throughput-oriented* scenarios. Instead of pursuing low latency, we target high throughput, which is useful for applications such as benchmarking (Liang et al., 2022), information extraction, and data wrangling (Narayan et al., 2022). Another reason for targeting high throughput is that we can significantly increase throughput by trading off latency, especially in memory-limited cases. To run an LLM with limited GPU memory, we need to offload it to secondary storage and perform computation part by part by partially loading it. On a typical machine, there are three levels of memory hierarchies, as shown in the right figure. The higher-level memory is faster but scarce, while the lower-level memory is slower but abundant. In throughput-oriented scenarios, we can sacrifice the latency by using a large batch size, and then the expensive I/O operations among different memory hierarchies can be amortized by a large batch of inputs and overlapped with computation. Fig. 1 illustrates the latency-throughput trade-off of three inference systems with offloading.



[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>.

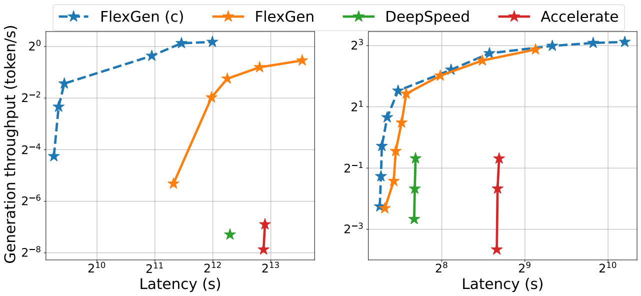


Figure 1. The latency and throughput trade-off of three offloading-based systems on OPT-175B (left) and OPT-30B (right). FlexGen achieves a new Pareto-optimal frontier with 100× higher maximum throughput for OPT-175B. Other systems cannot further increase throughput due to out-of-memory. "(c)" denotes FlexGen with compression.

Additionally, we show that a single throughput-optimized T4 GPU at home can be 4× more efficient than latency-optimized 8 A100 GPUs on the cloud in terms of tokens per dollar in Section 6.3.

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 KV cache. The strategy should specify what tensors to offload, where, and when to offload on the three-level hierarchy. Besides, the batch-by-batch, token-by-token, and layer-by-layer structure of the computation forms a complex dependency graph, where multiple ways to conduct computation are possible. These choices together form a complex design space. Exiting offloading-based inference systems (Aminabadi et al., 2022; HuggingFace, 2022) inherent strategies from training systems, which are only some suboptimal points in the whole space, hence they perform excessive I/O and only achieve throughput far below the hardware capabilities. The second challenge is to develop *effective compression strategies*. Previous works have shown promising results in compressing the weights and activations of LLMs. However, when combining compression with offloading for high-throughput generative inference, the I/O costs and memory reduction of weights and KV cache become more important, motivating the utilization of other compression schemes.

To that end, we present FlexGen, an offloading framework for 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. In particular, we prove that our search space captures an almost I/O-optimal computation order whose I/O complexity is within 2× of the optimal one. We then develop a linear programming based search algorithm that can be configured for various hardware specifications and latency/throughput constraints, providing a way to smoothly navigate the trade-off space. Compared with existing strategies, our solution unifies the placement of weights, activations, and KV cache, and enables a much larger batch size, which is the 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/calibration with negligible accuracy loss. This is achieved by fine-grained group-wise quantization, 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 can often allow a batch size that is orders of magnitude larger. On a single GPU, given the same latency requirement, FlexGen without compression can achieve 65× higher throughput. If allowing a higher latency and compression, FlexGen can further boost throughput and reach a 100× improvement. If given multiple distributed GPUs, FlexGen with pipeline parallelism achieves super-linear scaling on decoding.

We also compare offloading and decentralized collective inference based on FlexGen and Petals (Borzunov et al., 2022) as two representative systems. We conduct detailed comparisons between the two systems from the aspects of bandwidth and latency of the decentralized network. The results show that FlexGen with a single T4 GPU outperforms a decentralized Petals cluster with 12 T4 GPUs in terms of throughput and can even achieve lower latency in some cases.

2. Related Work

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

Recent years have witnessed the emergence of systems that are 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), Huggingface 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 hardware, offloading is an essential technique component — as far as we know, among current systems, only Deepspeed-Inference and Huggingface Accelerate include such functionality. These inference systems typically inherit the offloading techniques from training (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 the I/O traffic. Another attempt to enable LLM inference on accessible hardware is collaborative computing such as Petals (Borzunov et al., 2022).

Besides these advances on the system side, there are also many works oriented on the algorithm side 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 that they can compress weights down to 3 bits without compressing activations (Frantar et al., 2022), or compress both weights and activations 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.

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. Formally, the computation of a transformer layer can be summarized below. 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 $\mathbf{w}^i_K$, $\mathbf{w}^i_Q$, $\mathbf{w}^i_V$, $\mathbf{w}^i_O$, $\mathbf{w}^i_1$, $\mathbf{w}^i_2$, where $\mathbf{w}^i_K, \mathbf{w}^i_Q, \mathbf{w}^i_V, \mathbf{w}^i_O \in \mathcal{R}^{h_1 \times h_1}$, $\mathbf{w}_1 \in \mathcal{R}^{h_1 \times h_2}$ and $\mathbf{w}_2 \in \mathcal{R}^{h_2 \times h_1}$.

During ***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 $\mathbf{x}^i_K$, $\mathbf{x}^i_V$, $\mathbf{x}^i_Q$, $\mathbf{x}^i_{\text{Out}}$, where $\mathbf{x}^i, \mathbf{x}^i_K, \mathbf{x}^i_V, \mathbf{x}^i_Q, \mathbf{x}^i_{\text{Out}} \in \mathcal{R}^{b \times s \times h_1}$. Then, the cached `key`, `value` can be computed by:

$$\mathbf{x}^i_K = \mathbf{x}^i \cdot \mathbf{w}^i_K; \quad \mathbf{x}^i_V = \mathbf{x}^i \cdot \mathbf{w}^i_V$$

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

$$\begin{gathered}
\mathbf{x}^i_Q = \mathbf{x}^i \cdot \mathbf{w}^i_Q \\
\mathbf{x}^i_{\text{Out}} = f_{\text{Softmax}}\left(\frac{\mathbf{x}^i_Q {\mathbf{x}^i_K}^T}{\sqrt{h}}\right) \cdot \mathbf{x}^i_V \cdot \mathbf{w}^i_O + \mathbf{x}^i \\
\mathbf{x}^{i+1} = f_{\text{relu}}\left(\mathbf{x}^i_{\text{Out}} \cdot \mathbf{w}_1\right) \cdot \mathbf{w}_2 + \mathbf{x}^i_{\text{Out}}
\end{gathered}$$

During ***decoding phase***, given the embedding of the current generated token in the i-th layer noted by $\mathbf{t}^i \in \mathcal{R}^{b \times 1 \times h_1}$, the inference computation needs to i) update the KV cache; and ii) compute the output of the current layer. The update of the KV cache can be formalized as:

$$\begin{gathered}
\mathbf{x}^i_K \leftarrow \text{Concat}\left(\mathbf{x}^i_K, \mathbf{t}^i \cdot \mathbf{w}^i_K\right) \\
\mathbf{x}^i_V \leftarrow \text{Concat}\left(\mathbf{x}^i_V, \mathbf{t}^i \cdot \mathbf{w}^i_V\right)
\end{gathered}$$

The rest of the computation for the current layer is:

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

Memory Analysis. The memory footprint of LLM inference mainly comes from two components: the model weights and the KV cache. Consider 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, 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× the model weights, making KV cache a new bottleneck of large-batch high-throughput inference.

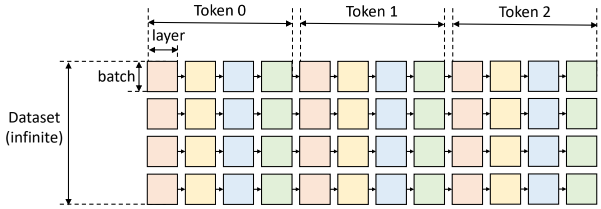


Figure 2. Computational graph of LLM inference.

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 memory hierarchy of GPU, CPU, and disk. We first formulate the problem and then construct the search space of the possible offload strategies in FlexGen. To find the best strategy, FlexGen builds an analytical cost model and searches for the optimal configurations with an optimizer based on linear programming. We also show how to extend FlexGen to support multi-GPU settings.

4.1. Problem Formulation

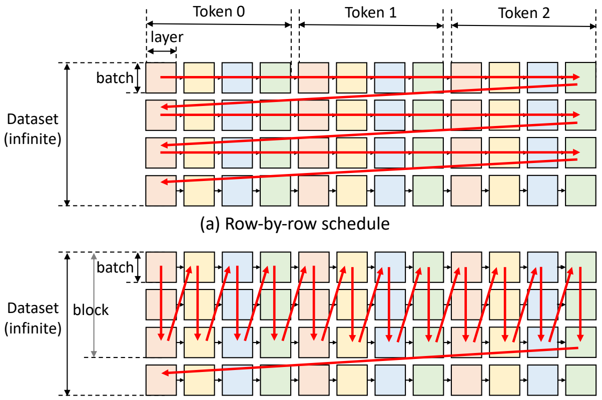
Consider a machine with three devices: a GPU, a CPU, and a disk. The GPU and CPU can perform computation while the disk cannot. The three devices form a three-level memory hierarchy where the GPU has the smallest but fastest memory and the disk has the largest but slowest memory. When an LLM cannot fit into the GPU as a whole, 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 batch on a layer on GPU. 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 meeting the following constraints:

- A square can only be computed if all squares left to it 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.

[1] We ignore the word embedding layer(s) which is relatively small compared with all the transformer layers.



(a) Row-by-row schedule

(b) Zig-zag block schedule

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

```
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
```

Figure 4. Block schedule with overlapping

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 weights I/O, we can traverse the graph column by column. All squares in a column share the weights so we can let the weights stay on GPU for reusing and only load/unload the activations and KV cache. However, 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 the zig-zag block schedule, there does exist another more advanced and IO-optimal schedule. We 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 weight 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 Fig. 4. 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 the block size.

Tensor placement. Besides compute schedule, a strategy should specify how to store these tensors on the three-level 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 KV cache. Given the percentages, there are still multiple possible ways to partition the tensors. Take weight tensors for 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 GPU), at the layer granularity (e.g., assign 50% of the tensors in a layer to GPU), or at the tensor granularity (e.g., assign 50% of the elements in a tensor to 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 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 GPU requires moving all KV cache to GPU, which incurs a substantial I/O cost as the KV cache is huge. On the contrary, computing the attention score on the CPU does not need to move the KV cache. It only requires moving the activations from GPU to 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 stored on the CPU/disk.

4.3. Cost Model and Policy Search

The schedule and placement in Section 4.2 construct 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 latency during decoding for one layer denoted as T_{gen} in one block. The total latency for computing a block can then be estimated as $T = T_{pre} \cdot l + T_{gen} \cdot (n-1) \cdot l$, 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, the 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, the 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 read. 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, s be the prompt length, 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 KV cache from the disk respectively so that the total latency of disk read is $dtoc^g = \frac{1}{\text{disk\_to\_cpu\_bandwidth}}((8h_1^2 + 4h_1 \cdot h_2) \cdot gd + 4 \cdot bls \cdot (s + \frac{n}{2}) \cdot h_1 \cdot cd + 2 \cdot bls \cdot h_1 \cdot hd)$

For computation terms, similarly, we sum up all computation events, including matrix multiplications and batched matrix multiplications on CPU and GPU.

Besides latency estimation, we also estimate the peak memory usage of GPU, CPU, and disk and 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. But since it is changing gradually, we relax the percentage variables in the cost model to be any real number between 0 and 1. 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 $p = (wg, wc, wn, cg, cc, cn, hg, hc, hn)$ becomes a linear programming problem shown in Eq. (1). The linear programming problem can be solved very fast because there are only 9 variables. This formulation can also be flexibly extended to include latency constraints and model approximate methods such as compression.

$$\begin{array}{rrcl} \min\limits_{p} & & T/bls & \\ \text{s.t.} & gpu\ peak\ memory & < & gpu\ mem\ capacity \\ & cpu\ peak\ memory & < & cpu\ mem\ capacity \\ & disk\ peak\ memory & < & disk\ mem\ capacity \\ & wg + wc + wn & = & 1 \\ & cg + cc + cn & = & 1 \\ & hg + hc + hn & = & 1 \end{array} \tag{1}$$

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 batch size slightly.

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

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 an almost optimal strategy for one GPU, the strategy is still heavily limited by I/O and has a low GPU utilization. If given more GPUs and associated more CPUs, model parallelism can be utilized to reduce the memory pressure of each GPU, which can potentially lead to a super-linear scaling.

There are two kinds of model parallelism: tensor parallelism 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. As we target throughput, FlexGen implements pipeline parallelism.

We use pipeline parallelism by equally partitioning a l-layer LLM on m GPUs, then the execution of all GPUs follows the same pattern. The problem is reduced to running a 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 Fig. 4 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 if allowing some approximations, because LLMs are typically robust to approximations if done carefully. This section introduces two 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, while preserving similar accuracy (Section 6.2). Different from 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 mainly for compression and reducing the I/O. 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 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 doing computation. Since both 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 possible ways to choose which dimension to group on. We find grouping the weights along the output channel dimension and grouping the KV cache along the hidden dimension preserve the accuracy while being runtime-efficient in practice. Interestingly, 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. Differently from this paper, 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 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 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.

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), BLOOM (Scao et al., 2022), because they 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. The evaluation metric is generation throughput, defined as the number of generated tokens / (prefill time + decoding time). Sometimes even running a full batch takes too long for some systems. We generate fewer tokens and project the final throughput in these cases. We use dummy model weights in throughput benchmarks for FlexGen and all baselines but use 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 them.

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 outperform 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 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. We also include Petals (Borzunov et al., 2022) as an additional baseline. We normalize the throughput reported in the Petals paper[2] by the number of used GPUs and get an effective per-GPU throughput of $0.68/14 \approx 0.05$ token/s. For a more comprehensive comparison with Petals, see Section 6.4.

Table 3 shows the results on 4 machines with one GPU on each machine. Even with 4 GPUs, OPT-30B or OPT-175B still cannot fit into 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 size to larger batch size, 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, FlexGen aggressively offloads all things out of the GPU to achieve a large GPU batch size and block size. Given the same latency requirement, FlexGen without compression can achieve a $65\times$ 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.

[2] From the case of 14 real servers in Europe and North America.

Table 2. Generation throughput (token/s) on 1 GPU. 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*	-	-	0.05	-	-	0.05
FlexGen	25.26	7.32	0.69	13.72	3.50	0.35
FlexGen (c)	29.12	8.38	1.12	13.18	3.98	0.42

Table 3. The scaling performance on 4 GPUs. The prompt sequence length is 512. 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 GPU	25.26	7.32	0.69	38.28	11.52	0.83
FlexGen 4 GPU	201.12	23.61	2.33	764.65	48.94	3.86
DeepSpeed 4 GPU	50.00	6.40	0.05	50.20	6.40	0.05

Table 4. 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

Runtime breakdown. Table 4 shows the runtime breakdown of OPT-175B on FlexGen. 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 5 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 64 and a block size of 64×2. "No policy search" illustrates the performance of worse strategies, showing the importance of a good policy. On both models, using CPU compute and overlapping bring non-trivial improvement. We also port the policy used in DeepSpeed/Accelerate into FlexGen runtime, showing the suboptimality of their policy.

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 6, "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. Token per Dollar Efficiency

We compare the token per dollar efficiency between FlexGen and FasterTransformers (NVIDIA, 2022), an industry-standard inference system. Assume running each framework for generation for one hour in `FP16` for OPT-175B. With FlexGen and a T4 at home, it generates $3600\,\text{s} \times 0.69\,\text{token/s} = 2484$ tokens and costs 250 W (the GPU/CPU/SSD power consumption) $\times$ $\$0.2/\text{kWh}$ (electricity price) $\times$ 1 h = \$0.05. The efficiency is 49680 token/\$. With FasterTransformers and 8×A100 (tensor parallelism) on AWS, it generates 497355 tokens but costs \$40.96. The efficiency is 12142 token/\$. FlexGen with a single T4 at home is $4.2\times$ more efficient than the latency-optimized 8×A100 on the cloud.

Table 5. 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)

Model size	30B	175B
All optimizations	7.32 (64×2, 20, 80)	0.69 (32×8, 0, 50)
No policy search	6.93 (64×2, 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 6. The accuracy/perplexity 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

6.4. Offloading vs. Collaborative Inference

In addition to offloading, decentralized collaborative inference is another option to lower the resource requirement for LLM utilization. Collaborative inference demands decentralized communication to accommodate the LLM inference workflow distributively. For example, Petals (Borzunov et al., 2022) transfers activations between decentralized devices following the design of Swarm Parallelism (Ryabinin et al., 2023). We carefully compare FlexGen and Petals under different network conditions by setting a private Petals cluster on AWS with 12 `g4dn.2xlarge` instances each equipped with a T4 GPU under the help of an author of the Petals paper. We use Linux traffic control to constrain the connections between instances to simulate a realistic decentralized network (delay: 10ms/100ms and bandwidth: 1Gbps/0.1Gbps) and benchmark the performance of an OPT-30B model (`FP16`) (input sequence length: 512, output sequence length: 32). We find that in terms of throughput, FlexGen runs on a single T4 GPU outperforms the whole Petals cluster with 12 T4 GPUs under all the decentralized network conditions, see Figure 5, where we believe offloading could be a more efficient solution than communicating a large volume of activations in a long decentralized pipeline with memory-limited GPUs. Petals cannot utilize offloading so it cannot use a very large batch size, which limits its scaling on throughput.

Interestingly, we find that FlexGen could even outperform Petals in terms of inference latency in slow connections, we speculate that this is because of the fact that the pipeline has to include more GPU due to limited per-GPU memory capacity, which amplifies the impact of high-delay and low-bandwidth networks. For a relatively not-that-slow network (10 ms latency), 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 larger proportionally, which significantly slows down the Petals during prefill. The decoding latency of Petals is lower than FlexGen, so it has a lower slope.

7. Conclusion

We introduce FlexGen, a high-throughput generation engine for LLMs. FlexGen lowers the resource requirements of running 175B-scale models down to a single 16GB GPU and achieves a practical performance for the first time. FlexGen provides a viable option for deploying LLMs for resource-constrained users and throughput-oriented scenarios.

Acknowledgement

We would like to thank Hao Zhang, Guangxuan Xiao, and Jue Wang for their insightful discussions.

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.

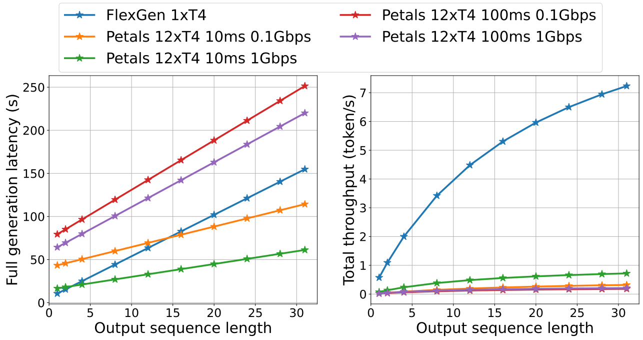


Figure 5. The comparison of inference latency and throughput between FlexGen and Petals under different network conditions.

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.

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.

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.

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., Newman, B., Yuan, B., Yan, B., Zhang, C., Cosgrove, C., Manning, C. D., Ré, C., Acosta-Navas, D., Hudson, D. A., Zelikman, E., Durmus, E., Ladhak, F., Rong, F., Ren, H., Yao, H., Wang, J., Santhanam, K., Orr, L., Zheng, L., Yuksekgonul, M., Suzgun, M., Kim, N., Guha, N., Chatterji, N., Khattab, O., Henderson, P., Huang, Q., Chi, R., Xie, S. M., Santurkar, S., Ganguli, S., Hashimoto, T., Icard, T., Zhang, T., Chaudhary, V., Wang, W., Li, X., Mai, Y., Zhang, Y., and Koreeda, Y. Holistic evaluation of language models, 2022. URL https://arxiv.org/abs/2211.09110.

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

Narayan, A., Chami, I., Orr, L., Arora, S., and Ré, C. Can foundation models wrangle your data?, 2022. URL https://arxiv.org/abs/2205.09911.

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*, abs/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.

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 7 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 7. 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 the CPU computation. To compute a square, 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 KV cache. From the computational graph, we can 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. So it would be good if we can reuse the loaded weights for a batch of prompts. Then the weights I/O time will be amortized. (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 but not 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 3 observations, as 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.

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 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

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

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.

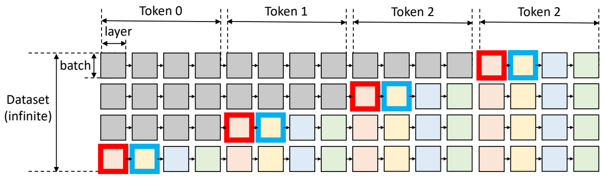


Figure 6. diagonal block schedule

Diagonal block schedule Figure 6 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 one row next.

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 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 and so forth. 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 IO-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 7 and Table 8 give the meaning of constants used in the cost model.

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
wn	percentage of weights on disk
cg	percentage of KV cache on GPU
cc	percentage of KV cache on CPU
cn	percentage of KV cache on disk
hg	percentage of activations on GPU
hc	percentage of activations on CPU
hn	percentage of activations on disk

Table 8. Notation Variables

[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 really fine-grained.

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

Object

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

Then the following constraints describe the calculation of total latency:

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

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

$$\begin{aligned} ctog^p &= \frac{weights\_ctog^p + hidden\_ctog^p}{ctog\_bdw} \\ &= \frac{1}{ctog\_bdw}((wc + wn)(8h_1^2 + 4h_1 \cdot h_2) \\ &\qquad + 2(hc + hn)s \cdot h_1 \cdot bls) \end{aligned}$$

$$\begin{aligned} gtoc^p &= \frac{cache\_gtoc^p + hidden\_gtoc^p}{gtoc\_bdw} \\ &= \frac{1}{gtoc\_bdw}(4(cc + cn)(s+1)h_1 \cdot bls \\ &\qquad + 2(hc + hn)s \cdot h_1 \cdot bls) \end{aligned}$$

$$\begin{aligned} dtoc^p &= \frac{weights\_dtoc^p + hidden\_dtoc^p}{dtoc\_bdw} \\ &= \frac{1}{dtoc\_bdw}(wn(8h_1^2 + 4h_1 \cdot h_2) \\ &\qquad + 2hn \cdot s \cdot h_1 \cdot bls) \end{aligned}$$

$$\begin{aligned} ctod^p &= \frac{cache\_ctod^p + hidden\_ctod^p}{ctod\_bdw} \\ &= \frac{1}{ctod\_bdw}(4cn \cdot bls \cdot (s+1) \cdot h_1 \\ &\qquad + 2hn \cdot s \cdot h_1 \cdot bls) \end{aligned}$$

$$\begin{aligned} comp^p &= \frac{linear\_layer^p}{mm\_flops} + \frac{att^p}{bmm\_flops} \\ &= \frac{bls(8s \cdot h_1^2 + 4s \cdot h_1 \cdot h_2)}{mm\_flops} \\ &\qquad + \frac{4bls \cdot s^2 \cdot h_1}{bmm\_flops} \end{aligned}$$

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

$$\begin{aligned} ctog^g &= \frac{weights\_ctog^g + hidden\_ctog^g}{ctog\_bdw} \\ &= \frac{1}{ctog\_bdw}((wc + wn)(8h_1^2 + 4h_1 \cdot h_2) \\ &\qquad + 2(hc + hn)h_1 \cdot bls) \end{aligned}$$

$$\begin{aligned}
gtoc^g &= \frac{hidden\_gtoc^g}{gtoc\_bdw} \\
&= \frac{1}{gtoc\_bdw}(2(hc + hn) \cdot h_1 \cdot bls)
\end{aligned}$$

$$\begin{aligned}
dtoc^g &= \frac{cache\_dtoc^g + weights\_dtoc^g + hidden\_dtoc^g}{dtoc\_bdw} \\
&= \frac{1}{dtoc\_bdw}(4cn \cdot bls \cdot (s + n/2) \cdot h_1 \\
&\qquad + wn(8h_1^2 + 4h_1 \cdot h_2) \\
&\qquad + 2hn \cdot h_1 \cdot bls)
\end{aligned}$$

$$\begin{aligned}
ctod^g &= \frac{cache\_ctod^g + hidden\_ctod^g}{ctod\_bdw} \\
&= \frac{1}{ctod\_bdw}(4cn \cdot bls \cdot h_1 + 2hn \cdot h_1 \cdot bls) \\
comp^g &= gpu\_comp^g + cpu\_comp^g
\end{aligned}$$

$$\begin{aligned}
gpu\_comp^g &= \frac{linear\_layer^g}{mm\_flops} + \frac{att^g}{bmm\_flops} \\
&= \frac{bls(8h_1^2 + 4h_1 \cdot h_2)}{mm\_flops} \\
&\quad + \frac{4cg \cdot bls \cdot (s + n/2) \cdot h_1}{bmm\_flops} \\
cpu\_comp^g &= \frac{att^g}{cpu\_flops} = \frac{4(cc + cn)bls \cdot (s + n/2) \cdot h_1}{cpu\_flops}
\end{aligned}$$

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

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

  GPU working memory (omit mask):

$$\begin{aligned} \mathit{qkv}^g &= \mathit{gbs} \cdot (2h_1 + 3(2h_1)) = 8\mathit{gbs} \cdot h_1 \\ \mathit{att}_1^g &= \mathit{cg} \cdot \mathit{gbs} \cdot (2h_1 + 2(s+n) \cdot h_1 + 2nh \cdot (s+n)) \\ \mathit{att}_2^g &= \mathit{cg} \cdot \mathit{gbs} \cdot (2nh \cdot (s+n) + 2(s+n) \cdot h_1 + 2h_1) \\ \mathit{embed}^g &= \mathit{gbs} \cdot (2h_1 + 2h_1) = 4\mathit{gbs} \cdot h_1 \\ \mathit{mlp}_1^g &= 2\mathit{gbs} \cdot (h_1 + h_2) \\ \mathit{mlp}_2^g &= 2\mathit{gbs} \cdot (h_2 + h_1) \end{aligned}$$

$$\begin{aligned} \mathit{gpu\_w}^g = {} & 2(1 - \mathit{wg})(8h_1^2 + 4h_1 \cdot h_2) \\ & + (1 - \mathit{hg}) \cdot 2s \cdot h_1 \cdot \mathit{gbs} \\ & + \max(\mathit{qkv}^g, \mathit{att}_1^g, \mathit{att}_2^g, \mathit{embed}^g, \mathit{mlp}_1^g, \mathit{mlp}_2^g) \end{aligned}$$

$$\mathit{gpu\_peak}^g = \mathit{gpu\_home}^g + \mathit{gpu\_w}^g < \mathit{gmem}$$

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

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

  CPU working memory:

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

$$\mathit{cpu\_peak}^p = \mathit{cpu\_home}^p + \mathit{cpu\_w}^p < \mathit{cmem}$$

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

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

  CPU working memory:

$$\begin{aligned} \mathit{cpu\_w}^g = {} & \mathit{wn}(8h_1^2 + 4h_1 \cdot h_2) \\ & + 2\mathit{hn} \cdot 2 \cdot h_1 \cdot \mathit{gbs} \\ & + 2\mathit{cn} \cdot 4(s+n)h_1 \cdot \mathit{gbs} \\ & + 2nh \cdot (s+n) \cdot \mathit{gbs} \\ & + 2h_1 \cdot \mathit{gbs}. \end{aligned}$$

$$\mathit{cpu\_peak}^g = \mathit{cpu\_home}^g + \mathit{cpu\_w}^g < \mathit{cmem}$$

- NVMe peak memory constraints:

$$\begin{aligned}nvme\_peak = {} & (8h_1^2 + 4h_1 \cdot h_2) \cdot wn \cdot l \\ & + hn \cdot 2s \cdot h_1 \cdot bls \\ & + cn \cdot 4(s+n)h_1 \cdot bls \cdot l \\ < {} & nmem\end{aligned}$$