Summary of FlexGen and ZeRO-Infinity

Shiqi He (shiqihe), Aakash Puntambekar (apuntamb), Yong Seung Lee (leeyongs)

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

Problem and Motivation

Running inference on large language models (LLMs) demands significant computational power and memory. However, some LLM tasks do not require immediate responses and are not latency-sensitive. Specifically, in addition to interactive applications like chatbots, LLMs are employed in various “back-of-house” tasks, including benchmarking, information extraction, and data wrangling. Therefore, in these scenarios, it is possible to trade off latency for higher throughput, providing opportunities to reduce resource requirements.

The paper focuses on designing efficient offloading strategies for high-throughput generative inference for low-resource scenarios (e.g., a single commodity GPU). This is challenging for two reasons. Firstly, designing an effective offloading strategy itself is difficult. During generative inference, tensors have three types: weights, activations, and key-value (KV) cache. The strategy must detail which tensors to offload, where to place them within the three-level memory structure, and when to offload these tensors during the inference process. These form a complex design space. Secondly, developing effective compression methods that are compatible with the offloading strategies is non-trivial. When mixing model compression with offloading strategies for efficient high-throughput LLM inference, the focus shifts towards minimizing input/output costs and reducing the memory footprint of both the weights and the KV cache. This underlines the need to explore new compression methods.

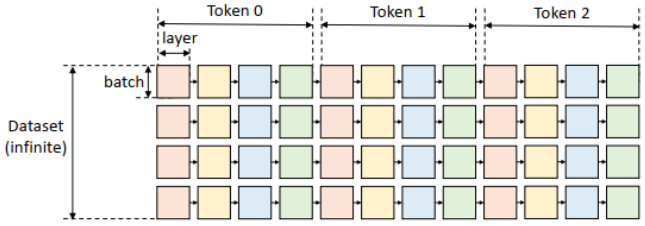
Related Works

Prior efforts to lower resource requirements of LLM inference correspond to the following three directions: (1) Model compression. There are many algorithm-oriented methods (e.g., Zeroquant) to decrease the total memory footprint. These methods aim to speed up the inference process or reduce the amount of memory needed, utilizing techniques like sparsification and quantization. In the paper, FlexGen compresses both the weights and the KV cache down to 4 bits. The authors then demonstrate how integrating this compression with offloading strategy can lead to further improvements; (2) Offloading. To enable LLM inference on such commodity hardware, employing offloading to utilize memory from CPU and disk is crucial. Among existing systems, only DeepSpeed Zero-Inference and Hugging Face Accelerate offer offloading capabilities. These systems usually adopt offloading methods from those used during training. However, they often overlook the unique computational demands of generative inference, which sets it apart from other processes; (3) Collaborative inference. Collaborative

computing, such as Petals, aims to amortize inference costs via decentralization. This enables LLM inference on accessible hardware.

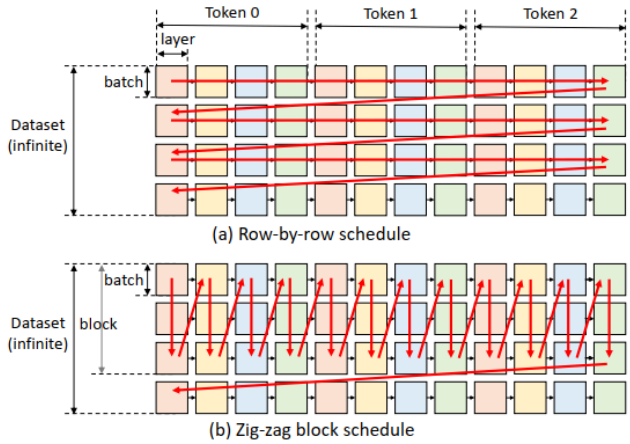
Solution Overview

The paper proposes FlexGen, an offloading strategy designed for executing high-throughput LLM inference tasks using limited hardware resources. The paper considers a machine with three devices: a GPU, a CPU, and a disk. These 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.



Problem Formulation

A typical LLM generative inference task involves two primary phases: (1) Prefill stage, which takes a prompt input to generate the KV cache; and (2) Decoding stage, which utilizes and updates the KV cache to generate tokens step-by-step. With the computational patterns generative inference, the paper formulates offloading strategies as a graph traversal problem. The above figure shows an example computational graph. The model has 4 layers and one request generates 3 tokens. A square represents the process of computing a batch of data for a particular layer. The squares with the same color share the same layer weights. The paper defines a valid path as a sequence that successfully processes (or computes) all the squares.



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
```

Search Space

As shown in the figure, there are two ways to traverse the graph: row-by-row and column-by-column. Existing systems traverse the graph row-by-row, which is the most efficient

method for completing the generation of one batch, allowing for the immediate release of the key-value (KV) cache after finishing a row. However, since two contiguous squares do not share the same weights, this schedule has to repeatedly load the weights, resulting in huge I/O costs.

FlexGen traverses the graph column-by-column to reduce the I/O costs of the weights. By traversing column-by-column, the weights stay on the GPU for reusing and only activations and KV cache are load/unload. FlexGen further applies some optimizations to this strategy. The first optimization is overlapping. Specifically, FlexGen manages to overlap the loading of weights for the next layer, the loading of cache/activations for the upcoming batch, the storing of cache/activations from the preceding batch, and the computation activities for the current batch. Besides, for long sequences, the paper suggests computing attention scores on the CPU if the associated KV cache is not stored on the GPU.

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

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

Cost Model

With the column-by-column approach, the paper defines the latency during prefill and decoding for one layer as T_pre and T_gen, respectively. The total latency for computing a block can be computed as the above equations, where l is the number of layers and n is the number of tokens to generate. In equation (1), bls is the block size and s is the prompt length. Furthermore, “w h c” represents the percent of weights, activations, and the KV cache, while “g c d” represents the percent of GPU memory, CPU memory, and disk. Besides latency estimation, FlexGen also estimates the peak memory usage of the GPU, CPU, and disk, and then adds memory constraints correspondingly. These form the cost model and lead to a linear programming problem that can be solved quickly.

Extensions

For multiple GPUs, the paper uses pipeline parallelism by equally partitioning an l-layer LLM on m GPUs. Since the execution follows the same pattern, FlexGen can reuse the policy search developed for one GPU. For quantization, the paper finds that grouping the weights along the output channel dimension and the KV cache along the hidden dimension preserves accuracy while being runtime-efficient. For sparsification, the paper demonstrates that loading the top 10% attention value cache on OPT-175B is sufficient to maintain the model quality.

Limitations

While the paper doesn't explicitly discuss the limitations of FlexGen, there are design choices that could potentially make it less than optimal in certain scenarios. First, the granularity for partitioning the weights is layer granularity. This impacts the cost model's analysis and might lack the flexibility needed for optimal performance across different computational environments. Furthermore, the paper acknowledges that the fine-grained, group-wise quantization method employed by FlexGen introduces some overhead during the compression process.
Besides, for computation delegation, relying on the CPU to do the attention score computation may not be feasible in practice when the CPU is occupied by other tasks.

Future Research Directions

The paper focuses on doing LLM inference on a single commodity GPU. However, exploring the application of FlexGen in multi-GPU setups offers an attractive expansion of its utility. While FlexGen only considers pipeline parallelism due to scaling concerns, it would be interesting to implement it with tensor parallelism to reduce the single-query latency.

Class Discussion

1. Is the idea that you don't have to keep evicting and reloading your weight matrix?
    - Yes - one batch, one activation, one KV cache
    - The idea is to finish computation and offload memory - this offloading can be at the same time as computing on a new batch.
2. This is fairly general, not specific to LLM, works for any model inference, right?
    - Yes, it can generalize, but if you don't have the memory bandwidth limitation you might not need this
    - Transformer has a memory challenge
    - This is for where you have so many parameters that you can't fit them on a single GPU

Zero-Infinity: Breaking the GPU Memory Wall for Extreme Scale Deep Learning

Problem and motivation

The problem the paper is trying to solve is the lack of memory when training large transformer-based models. The advancement of deep learning has been backed by the increasing size of models. Training huge models has been a trend and lots of studies have shown that this trend will continue. However, the size of the GPU couldn't follow the rapid growth of the model size. In the last three years, the largest trained dense model has grown over 1000x but the single GPU memory has increased by only 5x (16GB to 80GB). This limited memory capacity of the GPU urges the need to solve the memory issue in a systematic way. As a result, this paper presents "Zero-infinity," a heterogeneous system technology that leverages GPU, CPU, and NVMe memory to allow training large models on limited resources without requiring model code refactoring.

Related works

There have been lots of efforts to train the large models by leveraging multiple GPUs. 3D parallelism is one of them. It partitions the model or tensors into pieces and puts them into multiple GPUs. The problem with this approach is the necessity of significant model code refactoring, the difficulty of building load-balanced pipeline stages of complex dependencies, and still being limited by the total available GPU memory. In terms of using heterogeneous memory, ZeRO-offload stores the gradients and the optimizer states in CPU memory, but it still requires the parameters to be stored in GPU memory and replicated across all devices, so the model scale is limited to the total number of parameters that can be stored in a single GPU. Another effort to reduce memory usage of training was focused on reducing activation memory. Activations are the intermediate results produced during the forward propagation that need to be retained to compute the gradients during backward propagation. Researchers utilized compression, checkpointing, or live analysis to reduce this memory. Finally, ZeRO removes the memory redundancies across data-parallel princesses by partitioning the three model states: optimizer, gradients, and parameters. As a result, there has been extensive research in reducing memory usage while training, but they are all limited by the single GPU memory capacity and do not utilize different kinds of memory.

Solution overview

Zero-infinity employs lots of optimizations to satisfy the memory bandwidth requirements since there are more data movements in heterogeneous memory systems than in a traditional system that only uses GPU. To come up with the specific bandwidth requirements, the paper defined efficiency in terms of arithmetic intensity and bandwidth and found out the bandwidth requirements to achieve a certain efficiency. While quantifying the arithmetic intensity(AIT), they also found out that AIT varies a lot based on whether its parameters and gradients, optimizer states, or activation checkpoints. Since AIT varies, bandwidth requirements to achieve a certain

efficiency also vary. The result showed that the required bandwidth is the highest for optimizer states, then parameters and gradients, and activation memory required the least amount of bandwidth.

Based on these bandwidth requirements, Zero-infinity proposed six core designs. First is the infinity offload engine. It allows offloading all of the partitioned model states to CPU or NVMe memory by utilizing aggressive parallelization and asynchrony. The second is a memory-centric tiling. It reduces the working memory requirements by breaking down a large operator into smaller tiles that can be executed sequentially. Third is bandwidth-centric partitioning. It partitions individual parameters across all the data parallel processes and uses an all-gather instead of a broadcast. All-gather shines when it comes to moving data across CPU or NVMe since it allows us to use all PCIe links in parallel and use a high bandwidth Infini band to perform all gather. Thus, the bandwidth increases linearly with the number of nodes. Fourth is overlap-centric design. It overlaps lots of operations so that it can hide the costs. For example, it overlaps GPU-GPU communication with GPU computations, NVMe-CPU communication, and CPU-GPU communication. Also, it utilizes a dynamic prefetcher to load parameters that are needed in the future to minimize the communication overheads. Fifth is ease-inspired implementation, which eliminates the need for refactoring the model code. It provides automated data movement, parameter partitioning, and model partitioning using forward/backward hooks. As a result, the key contribution of this paper's solution is identifying and quantifying the bandwidth requirements based on the type of parameters and using various optimization techniques, such as tiling, partitioning, and overlapping to hide the communication overheads.

Limitations

One limitation of this paper is a hardware requirement. Since the paper is targeting training a large model, it involves lots of hardware requirements, which makes it difficult for individuals or researchers to try out. In Particular, the experiment was using hundreds of GPUs which is impossible to have unless you are working in a big corporation with a lot of resources. Another limitation is the paper was mainly focused on solving memory requirements and capacity issues. When training large models, performance is usually bounded by computation. Even though they explored the efficiency with respect to arithmetic intensity and memory bandwidth requirements, the focus was on meeting bandwidth requirements not increasing the computation performance.

Future research direction

Future research can focus on increasing GPU-GPU bandwidth and compute performance. According to the Deepspeed blog, training the 30 trillion parameter model will "demand 100x improvements in compute performance and the interconnect bandwidth between GPUs compared to what is available on current NVIDIA DGX V100 clusters." Moreover, we can try applying some techniques to serve large models. Techniques, such as overlapping communication and computation, would be useful for increasing the efficiency of inference and making inferences use fewer GPUs.

Class Discussion

1. What is the primary contribution? There are lots of competing works. Is it just the overlap centric design?
    - The “Infinity” part of ZeRO-Infinity is cuts one model into pieces to send to different GPUs then reconstruct, that's the major contribution, not so much overlap in computation
2. One optimization might be prefetching, i.e. operator i fetches i+1, etc. Are there heuristics for step? Is it i+2, i+4, what?
    - TMA is a new chip feature that is a memory copy engine capable of pipeline computation. The logic is, given a certain amount of memory, cut it into parts, fetch one until memory is full and batch the next one while working on it.
3. In evaluation - did they actually run 20 trillion parameters or is this simulation?
    - They actually ran it
    - Follow-up question(s): So what kind of model is that? No one has that big of a model, 1.8T is about as big as we've seen, this is an order of magnitude bigger. How did they build that big of a model? More layers? More nodes per layer? What?
        - Table 1 shows they increased both nodes and number of layers
        - There have to be some boundaries of where you can operate, this would be interesting to investigate.
4. 3D parallelism is probably Megatron - why wouldn't that work when the number of parameters is high?
    - 3D parallelism doesn’t have the ability to use disk
    - With this system vs. Megatron, in theory you could have infinite parameters and they would still go down the storage hierarchy.
5. Each layer is on one of the GPU nodes? Isn't it slicing each one of the layers?
    - The process is to offload some layers to CPU, then cuts each layer to run on multiple GPUs
6. Offloading strategy - is there a way to formulate this like the first paper did, where we find the optimal way to offload? Are there parameters behind the offloading e.g. how much to offload to CPU vs. GPU
    - It's purely discretized as the layers. You could redefine a layer as multiple transformer layers
    - PlaceTO - paper if you're interested in this

General

1. Maybe we should have a dedicated fetch engine for CPU to go between GPU and CPU
2. People will still be bounded by PCIe for a while