Summary of "Salus: Fine-Grained GPU Sharing Primitives For Deep Learning Applications"

Jingyuan Zhu (jingyz), Yuze Lou (yuzelou)

Problem and Motivation

With the emerging of Deep Learning. GPU has become a popular hardware for running DL workload due to its ability to achieve high parallelism. However, today's GPUs do not support fine-grained sharing primitives. This weakness makes many cluster scheduling unavailable such as SRTF. In addition, if a DL job cannot fully utilize the GPU (i.e. memory), the GPU will be very under utilized. In fact, experiments shows that models require a very variant memory usage (peak and average), both for different models and even for different time within the same model. This rises a lot of chances to better utilize the GPU memory since GPU requires all the data and model to be in memory in order to do the computation, and the workloads can be mixed and matched within memory if GPU memory sharing is available.

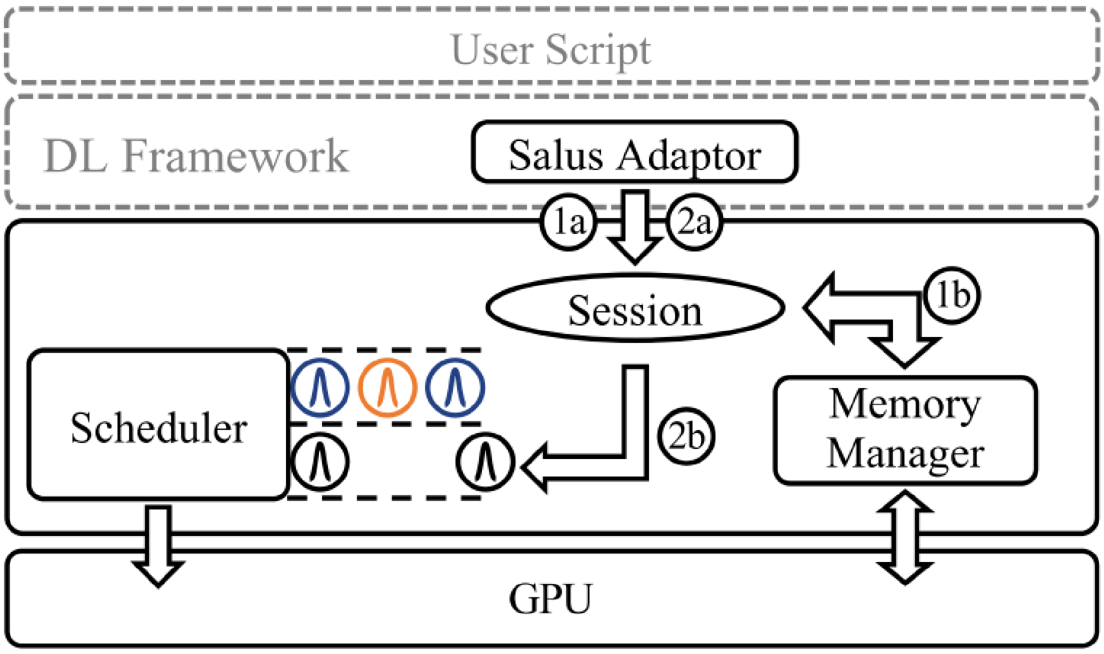
Hypothesis

Salus is a solution proposed for fine-grained GPU sharing. It's design is based on the uniqueness of DL jobs' memory usage characteristics. By enabling 2 GPU sharing primitives: fast job switching and dynamic memory sharing, Salus opens many possibilities for scheduling algorithm on GPU

The most important hypothesis this work is based on is that there is no efficient mechanism for GPU sharing, where a GPU can only be exclusively run on a single DL job at a time. Another significant assumption of Salus is the DL jobs' memory usage characteristics.

Solution Overview

Architecture



Salus serves as a singleton services for all GPUs in the cluster and abstracts away low level details as a virtual device. When the user submit a script as a job, a session is created between DL frameworks and Salus (1a), which is used to maintained the interaction between script (new iteration / inference) (2a) and Salus. Session will then request a lane from memory manager (1b), and each iteration from script will be scheduled by scheduler (2b) at the granularity of iteration.

Job Switching

Job switching is usually achieved by the checkpointing current progress on GPU, but may lead to large data transferring. By observing the DL memory allocations, Salus finds there are 3 types of memory allocations:

- Model for model parameters (large chunks and persistent)
- Ephemeral for intermediate outputs during each iteration (large but temporal)
- Framework internal. (Persistent)

In addition, it notices that persistent memory is much less than ephemeral ones in DL, and it is possible to keep >1 jobs' persistent memory while having either one's ephemeral memory.

Based on the two observations, Salus always keep persistent memory in GPU and use iteration as the granularity of scheduling to naturally save ephemeral memory transferring.

Spatial Sharing

Learned from traditional memory's structure: heap and stack, Salus also divide GPU's memory into persistent and ephemeral. Ephemeral region is further divided into continuous regions (lanes), where each lane is scheduled by itself. Having lanes will not eliminate the memory fragmentation, which still could cause superfluous out-of-mem error. However, since they are ephemeral memory, they are naturally defragmented after iteration.

In terms of assignment of jobs to lane / open a new lane, Salus applies a simple "safety" condition:

$$\sum_{\text{job } i} P_i + \sum_{\text{lane } i} \max_{\text{job } j \text{ in } i} (E_j) \leq C$$

where P_i is the persistent memory usage of job i and E_j is the ephemeral memory usage of job j in lane i (this means the maximum job ephemeral memory usage in lane i), and C is GPU memory's capacity

Scheduling

Salus mainly implements three scheduling techniques, PACK to maximize efficiency, SRTF to enable prioritization and FAIR to equalize.

Limitations and Possible Improvements

- In the paper, Salus only enables sharing among one GPU, what about multiple GPUs on the same machine which have very high bandwidth? Will this bring more space for scheduling?
- How does the job know its persistent/ephemeral memory usage? What happens if the job's ephemeral usage is too large to fit in any of the lane (return not found in the algorithm)? Is it possible to merge different lanes to create a larger continuous space?
- What if the job is executed on the distributed environment. How to schedule them on different GPUs?

Summary of Class Discussion

- In the paper, Salus only enables sharing among one GPU, what about multiple GPUs on the same machine which have very high bandwidth? Will this bring more space for scheduling?

Even in the same machine, transferring data between GPU takes sometime (hundreds of milliseconds), which could be fatal for performance during inference.

- Salus requires a profiling step in ahead, is there any better way to do?

One potential solution is that given the computation graph, it is possible to computer how many parameters and intermediate variables the model may generate. However, it still remains to be a problem that how do we differentiate the persistent memory and ephemeral memory.

- What is the tradeoff between utilization and

We can also schedule at the granularity of operations, which gives more fine-grained operation of But that may cause huge overhead on both the central scheduler and the process of back and forth between GPU and central scheduler to make the decision.

Summary of "SuperNeurons: Dynamic GPU Memory Management for Training Deep Neural Networks"

Jingyuan Zhu (jingyz), Yuze Lou (yuzelou)

Problem and Motivation

One common trend in DL community is to build deeper and wider Deep Learning models in order to achieve better performance, however, deeper and widers models consist of more parameters to be trained, which require a large memory. GPU DRAM capacity soon becomes the bottleneck of training large neural networks.

That's where this paper comes in. SuperNeurons is a dynamic GPU memory scheduling runtime to enable the network training far beyond the GPU DRAM capacity.

Hypothesis

1. The sizes of the DL models the paper look at exceed the GPU DRAM capacity.
2. Assume data parallelism instead of model parallelism.
3. CNN models.

Solution Overview

The main goal in SuperNeurons is to reduce the network-wide peak memory usage. SuperNeurons manages to reduce it down to the maximal memory usage among layers by utilizing three memory optimizations: Liveness Analysis, Unified Tensor Pool, and Cost-Aware Recomputation.

Liveness Analysis

Liveness analysis enables different tensors to reuse the same physical memory at different time partitions. For every layer, the system construct in and out set of tensors, and free the tensors that are not needed by any subsequent layers.

This is one very straight forward optimization and is consistent with the idea of garbage collection. The overall complexity of constructing the in and out set for N layers is O(N^2).

Unified Tensor Pool

Unified Tensor Pool (UTP) further alleviates the GPU DRAM shortage by asynchronously transferring tensors in/out of external memory.

Obviously offloading tensors in GPU memory to external storage introduces overhead. The authors determine that only tensors from CONV layers are worth offloading by analyzing how much each layer contributes to memory and computation usage.

The system also asynchronously brings the offloaded and soon to be reused tensors back to the GPU DRAM.

This is also a bit straight forward optimization and is similar to the idea of page fault / virtual memory in OS design.

Cost-Aware Recomputation

The runtime frees the tensors in cheap-to-compute layers such as POOL, and then recompute those freed layers in back propagation. In general, there are memory-centric and speed-centric strategies for the recomputation for memory.

The speed-centric strategy keeps the recomputed tensors so that other backward layers can directly reuse them, while the memory-centric strategy always recomputes the dependencies for each backward layer.

Limitations and Possible Improvements

I think one of the biggest limitations is that the use case of SuperNeurons is only discussed for CNN models in the paper.

I guess more comprehensive experiments about other DL models could be done.

Summary of Class Discussion

Some people in the discussion mentioned the above limitation. Another interesting discussion is that the strategies proposed in the paper are kind of well-known in the OS domain, but just borrowed to the GPU scheduling area.