Summary of "AntMan: Dynamic Scaling on GPU Clusters for Deep Learning"

Muhammed Ugur (meugur)

Problem and Motivation

Generally, supporting deep learning (DL) workloads on GPUs have become a critical component of production pipelines. Typically, companies will have large multi-tenant GPU clusters for these workloads. In addition, DL workloads are becoming more complex. Thus, it has become crucial to design scheduling systems that can efficiently utilize cluster resources for a variety of DL workloads.

Specifically, a key motivation of this paper is that at Alibaba, the researchers noticed that there is low utilization of GPUs (memory and computation) in these shared multi-tenant clusters. From their observations, this is due to two main reasons: DL workloads do not typically utilize all GPU resources during execution and the common reservation-based method for scheduling in these clusters results in GPU idling. Another problem is that the memory caching design in DL frameworks make it difficult to share GPU memory. Thus, a system is needed to address these issues while also dealing with fairness and strategic resource allocation.

Hypothesis

By exploiting the unique characteristics (such as memory used or training time) DL workloads, jobs can be better scheduled on shared GPUs. Specifically, small model sizes are common, so a large section of GPU memory could be scheduled for these jobs. Also, mini-batch cycles are generally small which allows for fine-grained GPU memory and compute scheduling. In addition, mini-batch computation can be used to profile the job performance and provide a metric for overall job interference. Designing a scheduler that uses this information can lead to better overall GPU utilization across the cluster. This scheduler will also need to work closely with DL frameworks to effectively share GPU memory.

Solution Overview

Key Insights:

- GPU resource sharing needs to achieve performance isolation for important jobs and prevent failure of jobs due to GPU memory contention.
- Scheduling needs to work closely with DL frameworks to better handle GPU memory sharing. This can be done by taking into account various statistics provided by these DL frameworks.

- Ideas in operating systems such as paging and cgroups can be implemented for properly managing GPU memory and compute.

AntMan is the proposed system to handle the aforementioned problems related to low GPU utilization in multi-tenant clusters. This system co-designs cluster schedulers and DL frameworks to address GPU sharing challenges. Below are the key components of Antman:

- Introduction of dynamic scaling mechanisms to enhance DL frameworks to allow fine-grained control of GPU memory and compute. This is done with a memory management technique similar to that of paging in operating systems. Specifically, universal memory is allocated to DL application tensors for page-like granularity.
- AntMan uses monitoring to dynamically shrink and grow GPU memory for better memory management. AntMan utilizes host memory for allocating tensors to prevent potential job failure and for dynamically adjusting cached memory. This method avoids tensor unnecessary tensor copies and has low overhead due to the unique characteristics of DL workloads.
- GPU computation is managed with AntMan's GpuOpManager which controls operation launching frequency, profiles GPU execution time, and distributes idle time slots to other operations. This allows better control over job interference during compute as it takes ideas from cgroups in operating systems.
- AntMan implements collaborative scheduling by implementing a global scheduler and local coordinator model. Local coordinators report metrics for dynamic resource scaling based on information from DL frameworks. The global scheduler makes high-level decisions across all jobs and sends out commands to local coordinators. Jobs are also classified into resource-guarantee and opportunistic (fits into unused cycles) to make better use of resources.
- AntMan focuses on multi-tenant fairness as a first priority over improving cluster efficiency. Fairness is achieved by policies in the global scheduler and local coordinators. Local coordinators manage opportunistic jobs by limiting GPU resource usage for them, optimizing aggregated performance for many jobs, and upgrading jobs when resources are available.

The above DL framework monitoring and fine-grained allocation are implemented in PyTorch and TensorFlow. The high-level cluster management is implemented on Kubernetes. AntMan has been fully integrated into Alibaba's internal systems. Based on their evaluation, AntMan improved the overall GPU memory and compute utilization in Alibaba's clusters by 42% and 34% respectively without taking away from job fairness.

Limitations and Possible Improvements

- The authors evaluate memory allocation overhead, but based on the model, it appears that this leads to different overheads. Thus, there may be some limitation here as DL workloads get more complex and larger over time.

- It is not clear how much overhead is associated with the GpuOpManager as it can wait on executing GPU kernels based on job interference. In addition, it also is not clear how adaptively adjusting computation scales with different model sizes.
- Another limitation is that different models appear to have different utilization during scheduling opportunistic jobs. Specifically, it is not clear why the local scheduling policy affects models differently.
- AntMan was specifically evaluated on data-parallel jobs with synchronous training. The authors state that other distributed training methods are supported, but it would be interesting to see how these methods perform using AntMan.

Summary of Class Discussion

For memory management, do they implement universal memory or is it based on NVIDIA's built-in unified memory?

- NVIDIA's unified memory has a similar mechanism that can swap out memory and swap it back in as necessary. This paper uses a unique approach that takes into account the fact that memory is typically allocated and deallocated within the same mini-batch. This saves copying from CPU to GPU, reducing overhead.

By giving resources to resource-guaranteed jobs, doesn't that still lead to idle time?

- Based on the scheduling policy, opportunistic jobs can be filled in to these idle cycles and improve utilization.

How does the system address how much computation is given to jobs?

- This is done through the GpuOpManager which does not starve specific operations for jobs. Also, the system uses the mini-batch computation time as a metric to adjust resources over time.

Why did they not compare against Salus (fine-grained GPU sharing)?

- Salus allows fast job switching only if the models are all loaded in memory which is slightly different from this paper.

Why not optimize inter-GPU and intra-GPU at the same time instead of splitting up the system design for each case separately?

- This is mainly for simplicity and abstraction.

When will AntMan perform worse compared to the baseline scheduler?

- Trade off is based on how much overhead is associated with going back and forth and scheduling many things at once.

Summary of "PipeSwitch: Fast Pipelined Context Switching for Deep Learning Applications"

Muhammed Ugur (meugur)

Problem and Motivation

Deep learning (DL) workloads consist of throughput-intensive training tasks and latency-sensitive inference tasks. The common practice for provisioning resources for these tasks is to dedicate GPU clusters for training and inference separately. This leads to a problem where idle resources for training cannot be used by inference tasks and vice versa (which is largely due to different usage patterns for inference and training). Moreover, inference clusters tend to be over-provisioned to meet their service level objectives (SLOs) and systems tend to allocate resources for inference tasks based on a per-GPU granularity. This leads to inefficient resource utilization within clusters.

Hypothesis

Similar to how operating systems achieve high CPU utilization via task scheduling and context switching, multiple DL workloads should be able to achieve high GPU utilization via time-sharing. CPU time-sharing has worked for cluster-level scheduling as well, so if this can translate well to GPUs, then DL workloads can achieve similar benefits. Moreover, fine-grained time-sharing should provide better utilization than provisioning dedicated resources.

This fine-grained approach is primarily limited by the switching overhead in GPUs. By minimizing the switching overhead, this time-sharing approach should achieve very high GPU utilization across different DL workloads. Due to the well-defined structure of DL workloads, switching should be highly optimizable through a computation and switching pipeline.

Solution Overview

Key Insights:

- Context switching and task scheduling in operating systems can be ported over to DL workloads to improve GPU utilization in a cluster.
- GPU memory is limited and DL models are continuing to get larger. This significantly impacts switching overhead on GPUs.

- Minimizing GPU switch overhead can be done by using a pipeline structure to overlap computation and swapping. This is primarily motivated by the well-defined structure of DL workloads in terms of their computation.

PipeSwitch is the proposed system that implements time-sharing for DL workloads across GPU clusters using a pipelined approach to multiplex applications. This approach enables the applications to still achieve their SLOs even with the overhead of switching.

- PipeSwitch uses a centralized controller to control memory daemons and workers for different tasks. The memory daemons manage GPU memory and models. Workers execute tasks actively or they standby and initialize/clean up tasks.
- The authors introduce a section to determine switching bottlenecks across four components: task cleaning, task initialization, memory allocation, and model transmission. The main bottleneck is task initialization, but due to SLOs, all components need to be optimized.
- PipeSwitch introduces a pipelined model transmission mechanism that takes advantage of the well-defined layered structure of DL workloads. Specifically, this pipelines model transmission to the GPU and executing the subsets of the entire task (layers). This mechanism tackles model transmission switching overhead.
- This pipeline is done on a per-group granularity where a group consists of multiple layers. This is different from the naive per-layer granularity which introduces large pipeline overhead. Small groups have large pipelining overhead while big groups have low pipelining efficiency. The optimal grouping is found using the proposed Optimal Model-Aware Grouping algorithm.
- For minimizing memory allocation overhead, PipeSwitch uses a unified memory management system that uses the insight that model memory is typically fixed and results in memory change in a simple and deterministic way. The memory daemon handles most of this management.
- For minimizing task initialization and cleaning, PipeSwitch uses active-standby worker switching. Specifically, initialization and cleaning can occur concurrently due to how memory is being managed. At a high-level, the controller will send messages to standby workers to initialize tasks and preempt active workers to clean tasks.

PipeSwitch was implemented and integrated into PyTorch. The changes are related to memory management in GPUs. PipeSwitch only incurs a task startup overhead of 3.6-6.6 ms and a total overhead of 5.4-34.6 ms (10-50x better than NVIDIA MPS) and has close to 100% GPU utilization.

Limitations and Possible Improvements

- PipeSwitch was implemented with the focus on single-GPU tasks for training and inference. Specifically, synchronous multi-GPU task support is not in the system, but this workload is predicted to increase in the future. One way to easily use PipeSwitch in this case is to use elastic synchronous training, but this is not very mature in DL frameworks. So, this is a limitation that needs to be explored in future work.

Summary of Class Discussion

Would GPU with unified memory with CPU reduce the overhead of switching?

- The paper makes the assumption that memory is not unified. An issue with this unified memory as well is that it does not really know what should be moved in and out when considering the entire system. So, unified memory can be too general.

What happens when you do not have a perfect group alignment when finding the optimal grouping strategy?

- Assumption that the execution does not necessarily have gaps during pruning. The actual algorithm will take into time of executing when grouping.

What type of resources are needed for launching new processes and initializing CUDA contexts? This is a question related to active-standby worker switching.

- Initializing a new CUDA context is around a couple hundred megabytes for GPU memory. This is characterized as not being an issue in the paper. Initialization is more time-intensive.

What is GPU utilization measured as for PipeSwitch?

- This is mainly for GPU memory since compute is not mentioned much

AntMan and PipeSwitch make different assumptions on model size. Which assumptions make sense?

- AntMan does not work well if you cannot fit all the models into GPU memory. PipeSwitch focuses mainly on larger models (otherwise use Salus).
- Both assumptions make sense since there are different DL workloads and individuals have different requirements.