Optimizing the Deep Learning Cluster

Deep learning brings break throughs in AI and is rapidly gaining popularity in various domains, such as computer vision, and natural language processing etc. Research institutes and enterprises are setting up their deep learning teams and infrastructures to expedite their AI transformations.

However, deep learning is computing intensive and requires powerful and expansive AI accelerators like GPUs. A GPU VM in the cloud costs nearly 10x that of a regular VM. Enterprises and cloud operators manage clusters of thousands of GPUs via cluster managers. The common practice today is to use cluster managers such as Yarn or Kubernetes which are designed mainly for big data jobs and non-AI application deployment. Due to many unique characteristics of the deep learning workloads, those technologies cannot efficiently leverage the power of GPUs, causing high cost and low ROI of AI investment.

This article will go in-depth to understand the uniqueness of DL workloads, their challenges to AI cluster management, the state-of-the-art optimization methodologies and outlook the future works in this domain.

# Dig into the problems

To better understand the current challenges of DL Cluster management, we start from the personas and scenarios involved in the large-scale deep learning development, as shown in Figure 1 below.

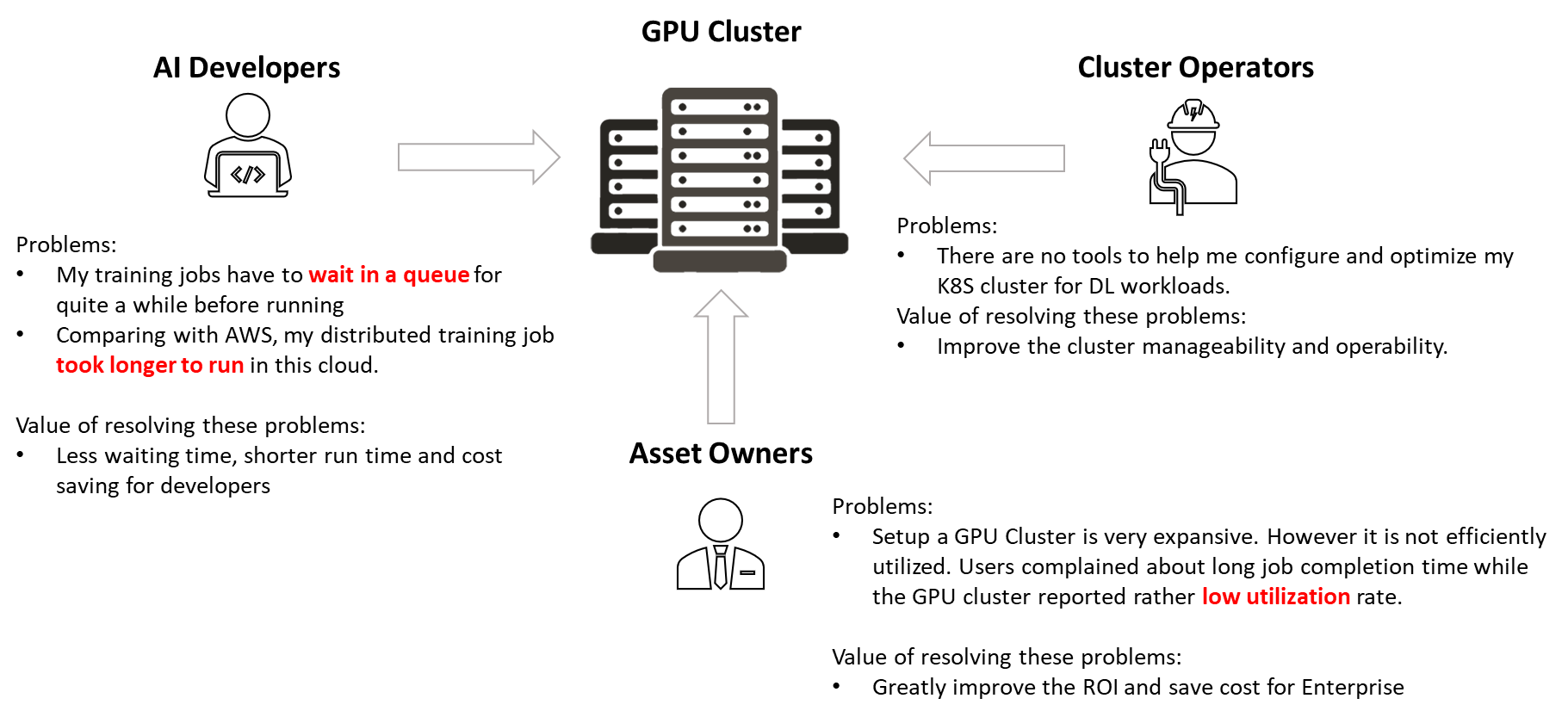


Figure 1: Persona of DL cluster management lifecycle

## Deep Learning developers

For deep learning developers, they will use GPUs throughout the AI development lifecycle. The typical usage scenarios are shown below.

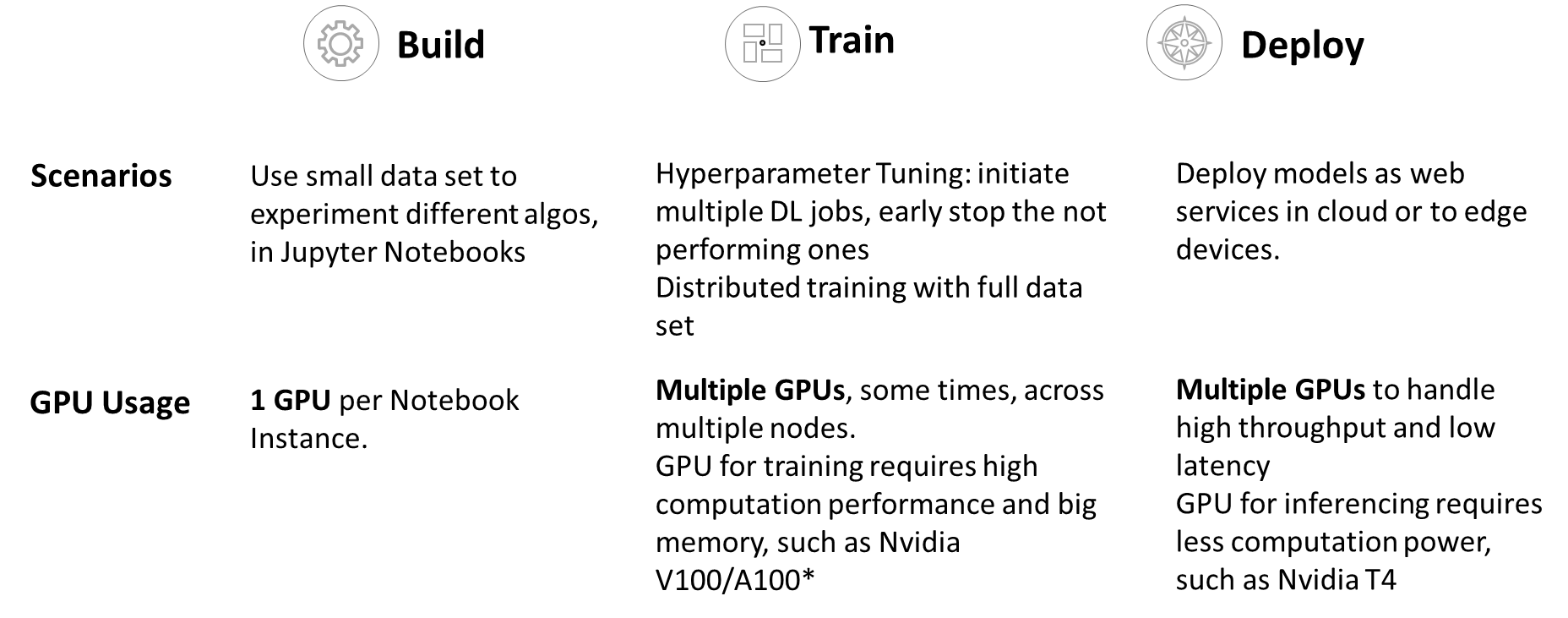


Table 1: AI Application Development lifecycle

Nowadays, a GPU server with 8 Nvidia V100 GPUs could easily cost $100K. A deep learning training job typically runs hours, days, and sometimes even weeks, on multiple GPUs. Therefore, to best leverage the GPU investment, the common practice for companies that fund AI teams is to set up their own GPU clusters for multiple users or purchase public cloud services. Either way, for developers, they will have a “shared pool” experience during AI development. The most common complains we heard from them with the “shared pool” experience are:

* Slow: Comparing with XX cloud, my training jobs took longer to run in this cloud.
* Starving: Some of my training jobs have to wait in a queue for quite a while before running. I can not do anything about it.

## Asset owners

The asset owners are those who own the GPU resources. They could be the company IT teams that set up their on-premise GPU clusters, or the public cloud vendors who manages tens of thousands of GPUs.

For assets owners, with the increasing of the DL workloads every day, they will surely expand their GPU assets to serve users’ needs and mitigate the users’ pain mentioned above. However, a key question from them should be addressed is: were the current GPU resources best used? because a few percent of GPU unitization performance improvement will save millions of dollars from purchasing new GPUs.

The ugly fact is that the utilization rate in production clusters is fairly low. We have seen many scenarios making the GPU utilization low in real production clusters. For example,

* For a given AI training workload, how much GPU resources it needs is mostly a guess work, as a result, AI developers request more GPU resources than the workload needs in most cases, which causes other AI workload waiting in the queue.
* Developers got dedicated notebook services with GPU attached, but most of the time, the developers were coding while the GPUs were occupied, but not in use.
* Developers asked for multiple GPUs for training, but in their distributed training code, they didn’t use all of them unintentionally. Some of the allocated GPUs were not in use.
* Some of the deep learning training jobs didn’t need that much of memory or cores offered by a powerful GPU. There are still GPU power left but cannot be used by other jobs.
* Some of the big jobs (need many GPUs) have to wait for a quite long time, even though there are scattered idle GPUs across the cluster (fragmentation).

The main reason for the low GPU utilization is because the current mainstream technologies used in deep learning job scheduling and computing resource management are not optimized for AI jobs, such as Kubernetes and YARN. In these systems, a fixed set of GPUs is assigned exclusively for the lifetime of a deep learning job. Those technologies didn’t design for the unique characteristics of deep learning workloads and thus we can see the big potential to improve

## Cluster Operators

Cluster operators are those who manage and operate the DL clusters every day. They ensure the smooth running of the clusters, monitor their usage and performance, make proper configurations and help users troubleshooting their job running issues. They hope to have a system and a set of ease-of-use tools to handle the job management automatically, elastically, and reliably.

# The uniqueness of Deep Learning workload

Different from generic computing workloads, deep learning workloads have many unique characteristics, which make the current cluster manager not efficient. In this section, we will dig into those unique characteristics to understand the challenges they bring to the cluster manager and pave the road to understand the solutions to address those challenges.

## Massive data communication

Deep learning training typically involves huge amount of input data and intensive computing. Thus, distributed computing is the go-to technology. There are several ways to handle the parallelization, such as model parallel and data parallel. The data parallel is the most efficient way for most cases. We use it to explain why there are massive data communication during distributed deep learning training

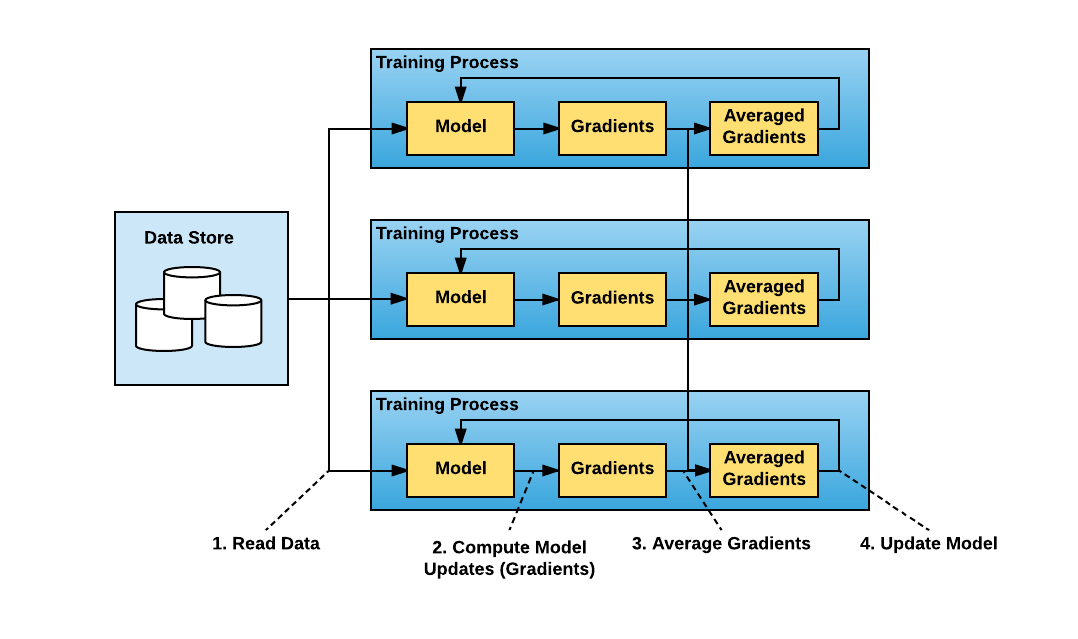


Figure 2: Data Parallel Distributed Training (from Horovod documentation)

For “data parallel”, the steps to train a model are shown as below:

|  |  |  |
| --- | --- | --- |
| Steps | Where it happens | CPU/GPU |
| 1. load the training scripts and a batch of training data | In each node | CPU |
| 1. Run those data through the model neural networks | In each node | GPU |
| 1. Compute model updates, a.k.a., the “gradients” | In each node | GPU |
| 1. Share those gradients for model aggregation | **Cross the nodes** | N/A |
| 1. Average gradients from all pods | Various approaches\* | GPU |
| 1. Update the model | In each node | GPU |
| 1. Repeat Step 1 until reach the desired accuracy |  |  |

*Note*:

* *“node” here stands for a computing unit. It could be as a container, or a virtual machine.*
* *“Various approaches” for Step 5 means that there are different approaches to do the gradients average. In traditional “parameter server” approach, the average will happen in dedicated “Parameter servers” which will cause additional data transmissions in the network to send the average data back to each worker node. The more efficient way is MPI based “ring all-reduce” approach (such as Horovod). The model aggregation happened in the worker node themselves.*

From the distributed training steps, we can see that in step 4, we need to pass the “gradients” across the nodes, eating up the network bandwidth. Those “gradients” data transfer among the nodes could reach hundreds of Mbytes (depending on the model architecture) for each iteration of the remodel training. We can expect thousands of iterations (or even more) to complete a thorough training. **Such kind of high-volume network communication will cause the training performance bottleneck and waste GPU cycles**. That’s why we take the massive data communication in distributed deep learning training as a unique challenge to the DL cluster optimization effort.

## Cyclic Memory Usage Pattern

In the training steps we described above, the step 1 through step 6 is defined as an iteration. For each epoch (gliding through all training dataset), there could be thousands of iterations. And it takes many epochs to compete a full training.

From the figure shown below, we can see that each iteration of a DL training job is very similar with a well-defined peak memory usage and a trough in between iterations. This is because DL jobs go through the same sequence of operations and memory allocations in each iteration. The trough is during model updates which needs to wait for the model synchronization among nodes. There is not much computing happening.

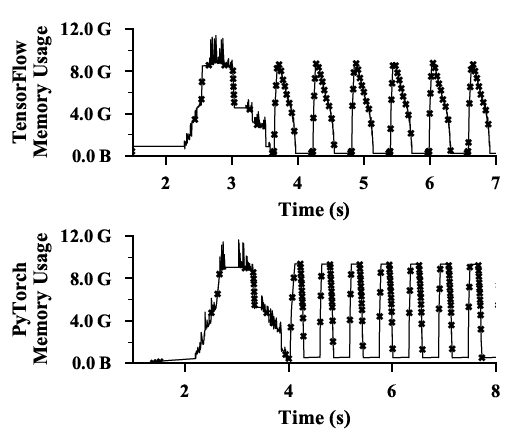


Figure 3: Cyclic Resource usage pattern training Resnet\_101 model with batch size 75 with P100 16G Memory (From Salus)

For different models, even though their peak memory consumption and cycle time are different, this cyclic pattern stay the same. This pattern indicates that the GPU are not fully utilized (in the trough), and there are substantial room to improve, by leveraging its predictable cyclic usage pattern.

## Time overhead of job stop and resume

For a deep learning job, it can stop and resume through “checkpoints”, where the intermediate training results will be saved. The checkpoint mechanism is supported by mainstream DL frameworks such as TensorFlow and PyTorch. By using checkpoint, a job can achieve job migration to other nodes.

However, the save and resume process of a deep learning job is cumbersome and time consuming. The chart below shows the steps involved, and the time overhead. We can see that:

* It takes substantial time (seconds to minutes) for DL jobs to suspend and resume
* The overhead is more obvious for multi-GPU jobs

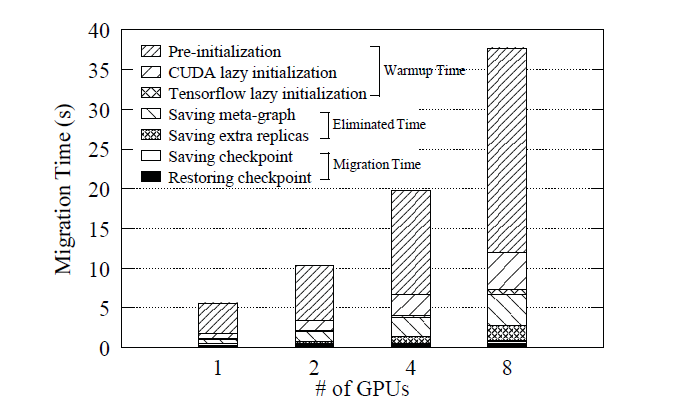


Figure 4: The breakdown of TensorFlow Migration Overhead for ResNet 50 model (from Gandiva)

The time overhead of migration will also make it challenge for the cluster manager to do the preemptive scheduling for DL jobs.

## All-or-nothing execution mode

To achieve best model accuracy and computing efficiency, the most common parallelization approach is Synchronized training. It requires all workers to be simultaneously active. The synchronization of model updates happens in each iteration. That means if any one of the worker nodes is impacted (network issue or node down), the whole training progress will get impacted.

Furthermore, the mainstream distributed deep learning training (DDL) frameworks support only fixed number GPUs for a job. To increase the number of GPUs, the job has to be fully stopped first, and then saves the checkpoint, rebalances the batch size, sets up a new communication ring, and resumes with new GPUs. An elastic distributed learning mechanism is required to make the whole process easy and efficient.

## Unpredictable training time

From job scheduling point of view, it is much easier if a job’s completion time is predictable. The algorithms like shortest-job-first (SJF) and shortest-remaining time-first (SRTF) are known effective to minimize the average Job Completion Time.

It is hard, however, to predict a DL job’s completion time. Even though some methods, such as [Optimus](https://github.com/kzhang28/Optimus), predict a DL training job’s remaining execution time by relying on its repetitive execution pattern and assume that its loss curve will converge. However, such proposals oversimplified the cases in production environment where

* Many models don’t converge well.
* Many models have very rough loss curves, not smooth at all.
* Many jobs are early terminated due to the failing of some criteria during hyperparameter tuning.

Therefore, it is still a challenge to schedule DL jobs with its unpredictable completion time characteristics.

# State-of-the-art DL cluster optimization technologies

Because of the substantial value to improve the DL cluster efficiency, there were many research efforts to resolve the unique challenges of managing DL workloads. This section will introduce some state-of-the-art work in this area and how they address those challenges.

## Profiling based Scheduling

We described the unique characteristics of DL workloads in previous section. The best way to understand the characteristics of a job is to profile it (such as the network traffic, memory usage, and iteration timing etc.), so that the Cluster manager can schedule it and place it based on its uniqueness, into the right resource at the right time. [**Tiresias**: A GPU Cluster Manager for Distributed Deep Learning](https://www.usenix.org/system/files/nsdi19-gu.pdf) profiles network traffic to prevent over-aggress consolidation during placement; [**Gandiva**: Introspective Cluster Scheduling for Deep Learning](https://www.usenix.org/system/files/osdi18-xiao.pdf) profiles job memory usage to pack several jobs into 1 GPU.

### Prevent over-aggressive consolidation during placement

Trying to minimize network communication during model aggregation is a common optimization in distributed training because the network can be a performance bottleneck and waste GPU cycles. Therefore, many existing GPU cluster managers blindly follow a consolidation constraint when placing DDL jobs. They assign all job workers to the same or the minimum number of servers. This method often causes job starving when it cannot be consolidated, even if there are enough spare resources elsewhere in the cluster.

According to [Tiresias](https://www.usenix.org/system/files/nsdi19-gu.pdf), the level of sensitivity to locality is very different for different type of models. The chart below shows the experimentation for 8-worker jobs with different placement strategies. “Consolidation” strategy is to place the job to minimal number of servers while “Random” strategy is to place the job randomly across several servers. We can see the VGG family and Alex model are very sensitive to locality, while other models are not.

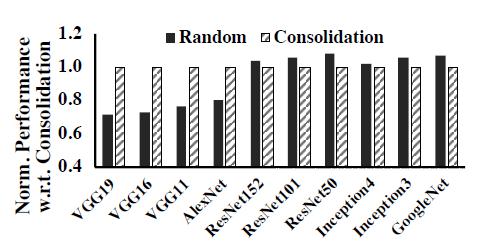


Figure 5: The sensitivity to locality are different for different models (from Tiresias)

With these findings, [Tiresias](https://www.usenix.org/system/files/nsdi19-gu.pdf) implements the profiler as a loadable library that intercepts RDMA ibverbs APIs (the profiler can easily be extended to support TCP/IP networks by intercepting socket APIs). Therefore, it can record all the network traffic info on each server, such as building connections, sending and receiving data. With the profiling information, Tiresias can decide which placement strategy (consolidation or random) to apply for a job.

### Pack multiple jobs into one GPU

[Gandiva](https://www.usenix.org/system/files/osdi18-xiao.pdf) enables multiple DLT jobs run on a GPU simultaneously and let the GPU time-share the jobs. It was called Packing. However, Packing in GPU is efficient only when the packed jobs do not exceed the GPU resources (cores, memory) and do not adversely impact each other. If jobs interfere, packing can be significantly worse (refer to the chart below).

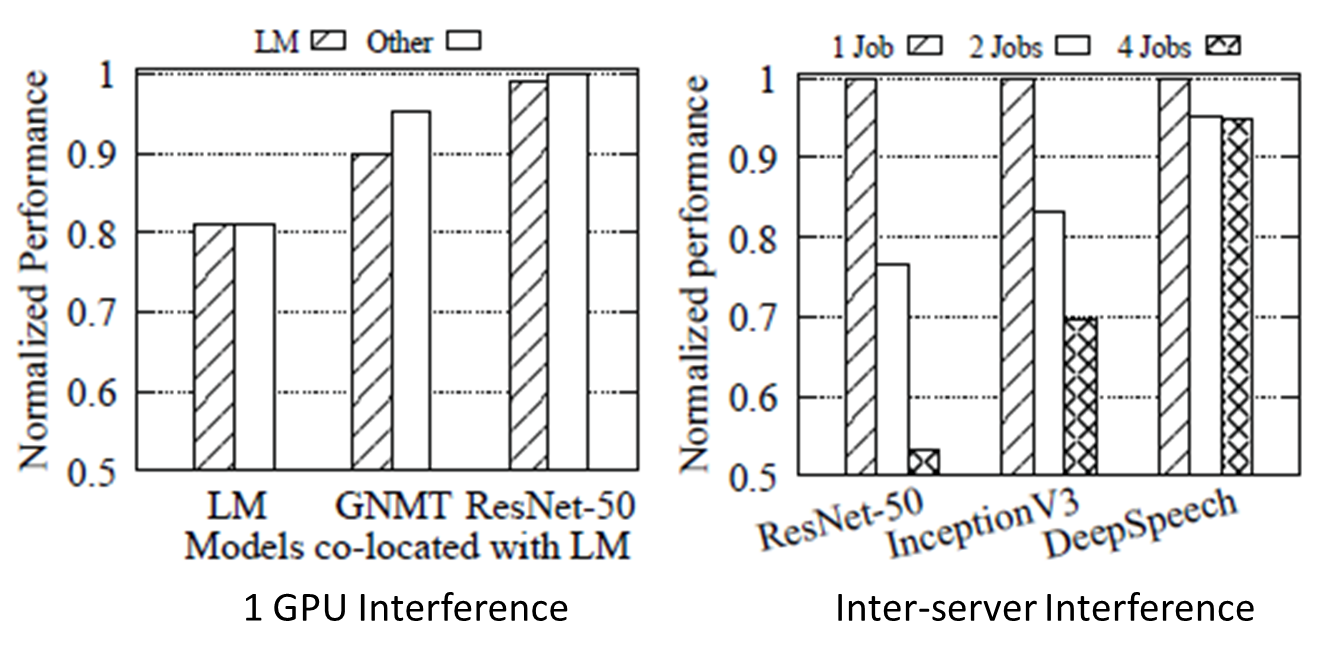


Figure 6: Interference of running multiple Jobs in same GPU (from Gandiva)

[Gandiva](https://www.usenix.org/system/files/osdi18-xiao.pdf) uses profiling to monitor the resource and progress of DLT jobs when they have exclusive access. If two jobs are identified as candidates for packing, it packs them together and continue monitoring them. If a given packing results in adverse impact on jobs’ performance, it unpacks those jobs.

Gandiva scheduler provides early feedback to jobs. It supports over-subscription by allocating GPUs to a new job immediately and using the suspend resume mechanism to provide early results. And then continuously optimizes the scheduling via continuous profiling.

## Advanced Preemptive Scheduling

Current DL cluster resource managers in production are rather naive. Many of them are just extended from Apache YARN’s Capacity Scheduler which was originally built for big data jobs. they only perform non-preemptive scheduling of jobs. Therefore, users often experience long queuing delays when the clusters are over-subscribed.

[Tiresias](https://www.usenix.org/system/files/nsdi19-gu.pdf) provides a preemptive scheduling framework to minimize the average Job Completion Time with no or partial prior knowledge on jobs. It does not rely on any intermediate DL algorithm states (e.g., training loss values) or framework specifics (e.g., tensors-to-parameter server mapping). The key idea in Tiresias preemptive scheduling is the **2DAS scheduling framework** that has two scheduling algorithms: Discretized 2DLAS and Discretized 2D-Gittins index.

The 2DAS ((2-Dimensional Attained Service) scheduling framework schedules DL jobs without relying on their exact durations while taking their GPU requirements into consideration. 2DAS generalizes the classic least-attained service (LAS) scheduling discipline as well as the Gittins index policy to DL job scheduling by considering both the spatial and temporal aspects of such jobs as well as their all-or-nothing characteristic.

At a high-level, 2DAS assigns each job a priority based on its attained service. The attained service of a job is calculated based on the number of GPUs it uses and the amount of time it has been running so far. The former becomes known upon the job arrival, while the latter continuously increases. On top of the 2DAS algorithms, Tiresias adopts the priority discretization frame to use different priority queues to manage jobs.

The benefits of adopting Tiresias is that it applied to all DL workloads, with no changes to existing DL Frameworks (TensorFlow or PyTorch). It can work with other optimization technologies mentioned in this article seamlessly.

## Elastic Distributed Deep Learning Training

As we discussed in the All-or-nothing execution mode section, current distributed deep learning frameworks (TensorFlow Parameter Server, Horovod etc.) don’t support Elastic training well. There are several efforts focusing on this area to support elastic distributed training, such as [Elastic PyTorch](https://github.com/pytorch/elastic), [Kungfu](https://github.com/lsds/KungFu), and [EDL: a Horovod like distributed deep learning framework](https://arxiv.org/abs/1909.11985). In this section, we will dive into **EDL** to understand the benefit of Elastic training, its challenges and approaches to address those challenges.

According to [EDL](https://arxiv.org/abs/1909.11985), there are several benefits from adopting elastic training:

* **Adjusting the trade-off between throughput and efficiency.** It is well known that the throughput of a model increases with the parallelism (# of GPUs) but the GPU efficiency decreases with the parallelism, sometimes even diminished. This is due to the higher communication cost under a larger parallelism. Elasticity can be used to dynamically adjust the parallelism of training jobs with profiling to get the best balance of throughput and efficiency.
* **Improving cluster utilization and Job Completion Time.** When the cluster load is high, scaling in large jobs to improve GPU efficiency, while the GPUs freed from the scale-in can be used to run small jobs that are queuing as to reduce their JCT; when the cluster load is low, scaling out jobs to make fuller utilization of the cluster and improve throughput and JCT.
* **Utilizing transient idle resources.** Some production workload data ([Microsoft Polly open data](https://github.com/msr-fiddle/philly-traces)) tells that during cluster peak time, around 40% of the idle intervals are less than 4 minutes, which takes up 41.5% of the idle resources. Elasticity can help utilize transient idle resources by scaling out a job when some GPUs become idle and scaling in the job when other jobs need to use these GPUs later.

There are several challenges, however, to implement an efficient elasticity.

* Existing parallelism adjustment causes **high scaling overheads**. Adding new workers to a running job takes many steps sequentially (execution context preparation, communication topology construction, and model preparation). Sometimes the time wasted on those overheads even longer than the gains of adding more GPUs.
* Existing Static **data partition is too complicate** for elasticity. For example, to maintain the same batch size after scaling out, a training job must rebalance the data partition for each worker node. Sometimes it has to wait until the completion of the current epoch to do the new data partition. It is a substantial waste for those new assigned GPUs.

[EDL](https://arxiv.org/abs/1909.11985) proposes a design to enables low-overhead elastic deep learning by using **stop-free scaling and dynamic data pipeline**.

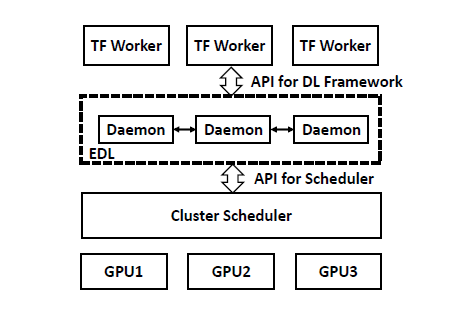


Figure 7: EDL Architecture

For stop-free scaling, the key idea is that the training on existing workers does not need to be stopped when the new workers conduct execution context preparation (which consumes majority of the scaling overhead). Each new worker launches two separate threads, a main thread and a background thread. The main thread conducts execution context preparation while at the same time the background thread performs leader discovery and sends a registration request to the leader. By using this approach, the existing workers can continue the training without being affected, and thus achieve stop-free scaling.

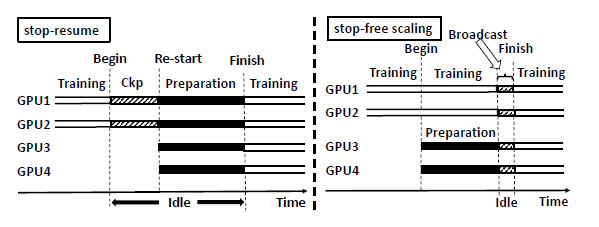


Figure 8: Stop-free scaling (by EDL)

For Dynamic data pipeline, EDL assigns data partitions to workers dynamically in an on-demand fashion. The dataset is logically divided into partitions. The partitioning is only conducted at the meta-data level, e.g., recording file names and offsets, and the dataset is not physically partitioned. The leader generates a random permutation of the indexes of the partitions and uses it for dynamic data assignment. When a worker needs a new partition, it sends a data-read request to the leader. The leader replies the request with the meta-data (e.g., File path, offset and length) of the next unassigned partition.

EDL also develops a new scheduler by extending Tiresias to fully accomplish the benefits of elastic training.

## GPU Sharing

The most straightforward way to improve the GPU utilization rate in a DL cluster is to enable multiple jobs running on 1 GPU, a.k.a., GPU sharing. However, for a long time, there is no easy way to share a GPU like sharing a CPU (vCPU), until the very recent release of Nvidia MIG (Multi-Instance GPU). This section will describe some state-of-the-art technologies in GPU sharing.

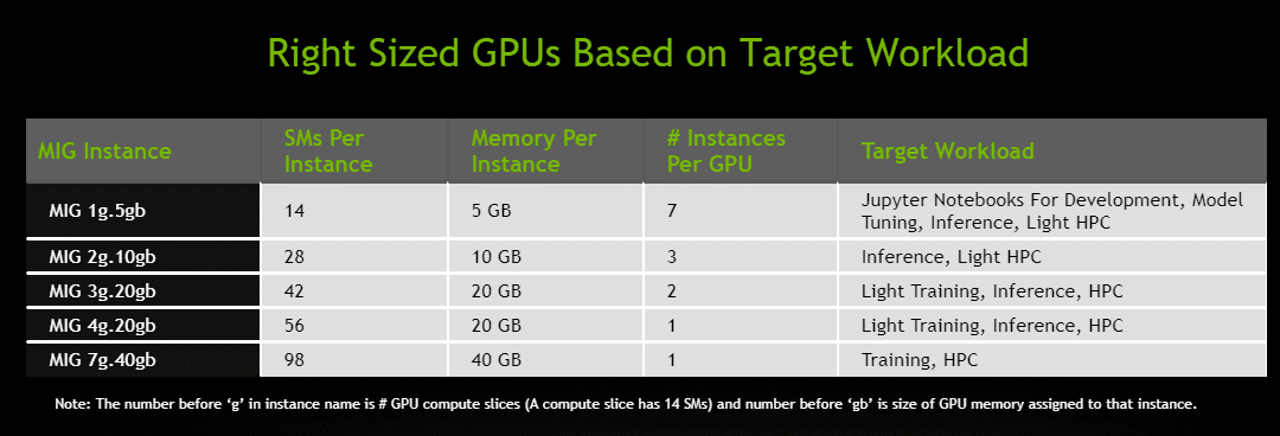
### GPU Sharing support from Nvidia: MPS and MIG

Nvidia released [MPS (Multi-Process Service)](https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf) a few years ago to allow multiple CUDA processes sharing a single GPU context. Each process receives some subset of the available connections to that GPU. However, users still have to use static partitioning of the GPU memory for each concurrently running job. The job isolation support of MPS is weak.

In May 2020, Nvidia announced its latest data center GPU A100, based on Ampere Architecture. Along with A100, Nvidia also announced the new GPU sharing technology: [MIG (Multi-Instance GPU).](https://www.nvidia.com/en-us/technologies/multi-instance-gpu/) MIG is a large leap over MPS.

MIG partitions a single NVIDIA A100 GPU into as many as seven independent GPU instances. They run simultaneously, each with its own memory, cache and streaming multiprocessors. That enables the A100 GPU to deliver guaranteed quality-of-service (QoS) at up to 7x higher utilization compared to prior GPUs.

An NVIDIA A100 GPU can be partitioned into different-sized MIG instances. For example, users can create two MIG instances with 20 gigabytes (GB) of memory each, three instances with 10 GB or seven with 5 GB. Users create the mix that’s right for their workloads.



MIG provides fault isolation: a problem in one instance won’t affect others running on the same physical GPU. Each instance delivers guaranteed QoS, assuring users their workloads get the latency and throughput they expect. It works smoothly with existing Linux operating systems as well as Kubernetes and containers.

All these features of MIG make it achieve a true GPU sharing technology. We can anticipate the future DL clusters equipped with A100 will have much better GPU utilization rate than current ones.

### Fine Grained GPU Sharing

Besides MPS and MIG, there are research works on software level to support fine grained GPU sharing, such as [**Salus**: Fine-Grained GPU Sharing Primitives for Deep Learning Applications](https://arxiv.org/abs/1902.04610) .

Salus addresses both temporal and spatial aspects of the memory management problem by enabling two GPU sharing primitives:

* Fine-grained time sharing via efficient job switching among ongoing DL jobs. It realizes the fast job switching (instead of full stop and resume) by keeping the models in memory and adding on demand checkpoint support in DL frameworks (TensorFlow). Salus achieves iteration-based fine-grained sharing.
* Dynamic memory sharing via the GPU lane abstraction. It is similar to Gandiva’s packing multiple jobs in one GPU, based on their profiling info.

With the introduction of MIG, this kind of fine-grained GPU sharing approach may be “obsolete”, because the gains are trivial (if not none) comparing with MIG, while complicated implementation will make it error prone.

# Outlook of future works

After understanding the challenges brought by the unique characteristics of deep learning workloads, and studying the state-of-the-art technologies that address those challenges, please think of a few questions: If you would oversee a large-scale DL cluster in a company, what you would do to improve its efficiency and GPU utilization rate? With all these state-of-the-art technologies, where to start with and what are the priorities?

Here are my thoughts pending for Proof of Concepts experimentation.

* I would start from setting up a benchmark first, which includes a testing environment, a DL workload simulator, a production dataset with latest DL algorithms as workloads and a benchmark metrics to evaluate all the future optimization work. It would be great if such kind of DL Cluster Optimization benchmark is open to and adopt by everyone so that the future optimization work can be compared apple to apple.
* Among all the optimization technologies, I would adopt GPU Sharing first, because it will have the most direct effect to increase GPU utilization. If I have the luxury to equip with A100 GPUs, I would use MIG, both for training and inferencing. If I have P100/V100 GPUs, I will leverage MPS to create fixed size virtual GPU instances (copycat MIG) for my clusters. They are not as robust as MIG instances, but could still be a solid improve for GPU utilization.
* My next pick is to implement profiling-based scheduling. A comprehensive and full lifecycle profiler will identify the uniqueness of a DL job, such as its network traffic load, GPU memory usage, and iteration pattern. It is the key to enable the cluster to make the right scheduling decision for DL workload.
* My next move is to invest on implementing the advanced preemptive scheduling, because it won’t change any of the DL framework and can seamlessly work with other optimization technologies.
* Last, but not least, I would adopt an elastic deep learning framework, such as Kungfu and EDL. In fact, if we can have a mature elastic training framework, I would do it right after the GPU sharing, because elastic training will bring more benefits than the rest technologies. However, at this very moment, we don’t have a well-proved one yet. I would keep a close eye on Elastic PyTorch and Elastic Horovod to see if they are advanced enough to take care of elastic training efficiently. At this stage, if I have to make a decision for team to start a trial of elastic framework in my cluster, I would go for Kungfu because besides its under developing elastic training supports, Kungfu itself is a more efficient distributed deep learning framework than Horovod.

With the thinking above and the research we have done so far, we are working on an open source project focusing on addressing the challenges of DL workload cluster management. Please stay tuned for the progress of this project.