Big Brain: Efficient Cluster Management for Deep Learning Jobs

Ajay Uppili Arasanipalia, Chirag Chandrahasa Shetty
University of Illinois at Urbana-Champaign
{aua2,cshetty2}@illinois.edu

Abstract

With the rising popularity of large scale Deep Learning (DL) workload, GPU clusters have become a necessity. Yet, the tools to manage them are far from ideal. On one hand the deep learning engineers are burdened with task of estimating resource management usage of their models and routinely face Out-of-memory (OOM) errors. On the other hand, typical GPU cluster utilization are low, often below 30%. But with recent developments in GPU virtualization and drawing from our learnings building Beachi - a fast model splitting system - we explore ways to improve GPU cluster management for DL. Model Parallelism (MP) techniques have so far been seen as a need, when the models are too big. However, we (plan to) demonstrate MP as an essential tool in building efficient DL workflows.

1 Introduction

Today, Deep Learning jobs are a big part of workloads run by organizations, both big and small. Often, DL training and inference tasks require GPUs and other specialized hardware to run in reasonable times. Hence it is not uncommon to have a cluster of GPU-enabled machines in production pipelines. Accordingly the cloud infrastructure providers like AWS, GCP, Azure etc offer this choice to their customers [1] [2]. In addition, to manage costs, companies also build and manage their own local cluster[3].

Inspite of the importance of DL jobs and large investments in the hardware infrastructure, the software to manage them are mostly same as those used for traditional big data jobs (like MapReduce). A typical workflow involves packaging the job into a container like Docker and submitting it, along with resource requirements, to a general purpose orchestration tool like Kubernetes or YARN. Running DL jobs using this process leads to significant interactions between the developer (a data scientist) and the underlying infrastructure as we detail in Section []. On the other hand, GPUs remain under-utilised. For instance, a startup that helps companies setup ML pipelines [4] notes that the GPU utilization for over a third of their users is less than 15%, inspite of their users being experienced DL practitioners. There are two primary causes:

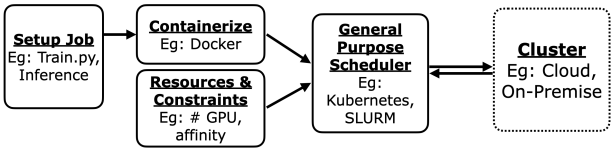

Figure 1. A Typical Workflow

1. Schedulers do not yet support fine-grained abstraction over GPUs like they do with CPUs. Thus a job requiring GPU resources can acquire them only in integer units, resulting in wastage [5].
2. Secondly, the responsibility of estimating the resource requirements (memory/compute) is on the developer. Not surprisingly, demanding more resource than strictly necessary is common practice. In addition, it takes significant effort to design models and training mechanisms so as to make most of the allotted GPU. This is tangential to expertise and objectives of the data scientists [15]

The resulting inefficiencies is illustrated in Figure 1. This over-provisioning coupled with low utilization of the provisioned resource leads to, as [10] puts it, “mostly-allocated, but lightly-utilized systems". This obviously has serious financial as well as developer productivity implications, especially when most data science teams are small (>60% have less than 10 members [6])

This by no means is a new problem. A large body of recent works have suggested improvements to cluster management tools by exploiting the special characteristics of DL jobs. For instance AntMan [11] exploits the periodic nature of resource consumption in model training to improve cluster utilisation. Clockwork [13] exploits the predictable time duration of DL tasks to design a scheduler. It is also worth noting that many of the works are proprietary systems that address the problems in large companies like Microsoft and Alibaba. The systems either build custom tools or accept the integer-level granularity and instead focus on other aspects like fairness of allocation or generality of the the jobs supported (eg: both Pytorch and Tensorflow jobs). Even then, in systems like Gandiva [12] ‘job packing’ on a single GPU is one of the possible actions taken by the job scheduler.

Recently there have been attempts to do allocation at sub-GPU level i.e sharing jobs on the same GPU. For instance,

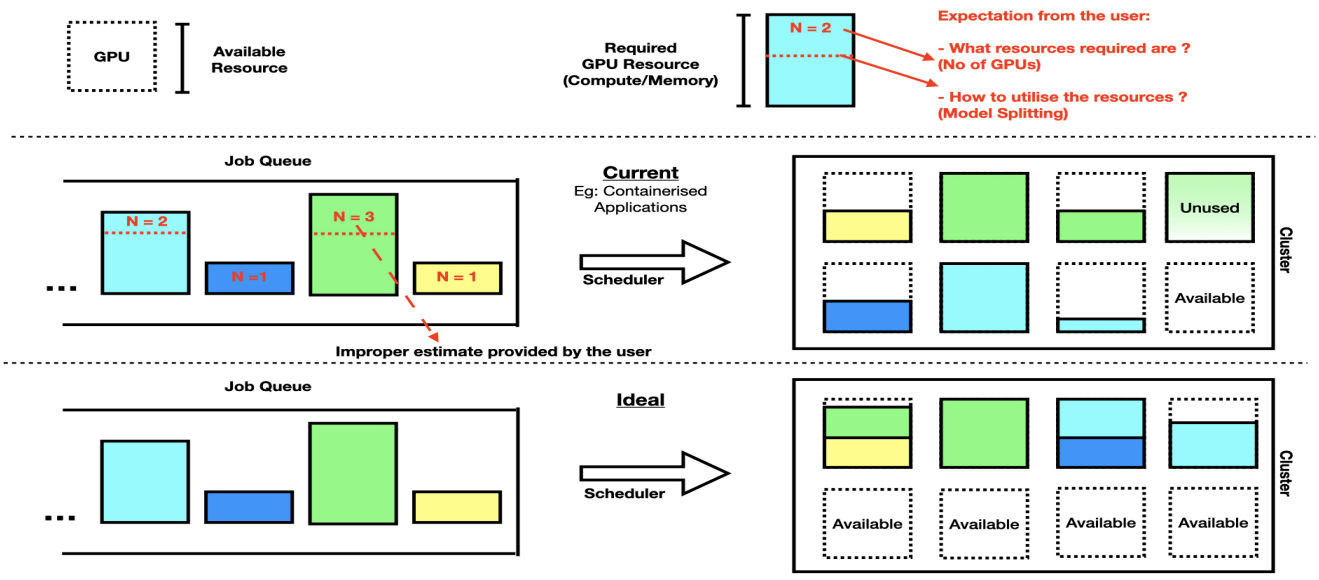

Figure 2. Low cluster utilization by coarse-grained resource allocation and incorrect estimates. An Ideal system would require no such estimation and optimally pack jobs to exploit inherent parallelism in GPUs

startup run.ai [7] released a feature to use fractional GPU resources directly through Kubernetes. A highlight of Nvidia's Volta architecture and Multiprocess-Services [8] is the ability for multiple processes to seamlessly share GPU memory and compute resources. Memory leakage protection is also builtin. While these improvements provide the substrate to utilize GPUs better, the systems still depend heavily on the developers expertise. Nevertheless, fast evolving efforts in virtualization of GPU only further validates the need for fine-grained allocation of GPU resources. We ask, even if GPU virtualization is accessible to a team, how can they make most of their hardware and do so transparently, without requiring intervention of the developer. Thus, GPU virtualization will make our work more reliable by making it easy to manage process boundaries.

We argue that Model Parallelism - the practice of dividing a model across multiple GPU's - naturally arises in designing an efficient cluster management system that granularly allots GPU resources. Further, drawing from our learnings designing Baechi [14] - a model splitting mechanism - we build a scheduler that with properties of an ideal scheduler show in Figure 2.

Model Parallelism so far has been exclusively seen as a method to be used when training a model that does not to fit in one single GPU or when using large batch sizes (often data parallelism is preferred for the later case).

Specifically, we highlight the following:

- Consider an incoming job to be scheduled on a cluster already running some jobs. With the ability to do fine-grained allocation of GPU resources, the problem of scheduling is equivalent to model parallelism of the model over the residual resources in the cluster
- Model parallelism splits the computation graph of a DL job across multiple GPUs with the objective of minimizing the training or inference step time. However the same algorithms can be used to split computations within a GPU to exploit the inherent parallelism of GPU, thus improving utilization. This however requires careful design using streams [15]
- A resource estimation routine is an essential component of a model splitting system like Baechi. This in turn frees the developer from the burden of estimating the resource requirements.

Figure 2 shows the high level description of a scheduler inspired by this approach. The grey highlight shows the blocks that constitutes current design of Baechi. [14]. In the report, we will look at how these different components contribute towards building an ideal cluster management system as described in Figure 1.

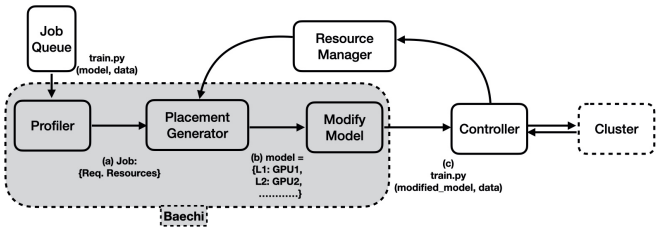

Figure 3. High Level Design

2 Design

Current workflow used by developers to run DL jobs is as shown in Figure 1. However when multiple users try to access the cluster, we need a scheduler to manage the job queue. A naive approach, treating GPUs in integer granularity, would simply keep a count of free GPUs (a Resource Manager), and allot as many GPUs as the incoming job requires. The number is updated every time a job enters of leaves the cluster.

However, to make fine grained use of the GPUs, we must have the ability to split an incoming job among the residual resources left in the cluster. Accordingly, the model splitting routine sits between the job queue and the cluster controller. Figure 3 shows the high level design of the system.

2.1 Big Brain Job Operator

The interface for our system is managed almost entirely through Kubernetes and other popular, well-established cluster management systems. Specifically, we use the Kubernetes Operators pattern to implement our job management pipeline. This architecure is illustrated in 2.1.

We define `BBJob` as a Kubernetes Custom Resource Definition (CRD) that represents a Big Brain job. The `BBJob` specification contains the bare minimum required to build a container that can launch a training loop for a machine learning model.

Once the operator receives a `BBJob`, it performs a quick profiling and placement step to determine the best node and GPUs to run the job on. The operator then generates a pod and pushes a new conatiner to the cluster registry for future reference. While we do not perform any intellgent caching of models, datasets, etc. yet, this is a future enhancement.

Once the pod is running, the operator will monitor it and update the state of the corersponding `BBJob`. When the job is run to complete or is killed due to an error, the operator will invoke a `BBJob` delete callback that can be handled by the developer or client library.

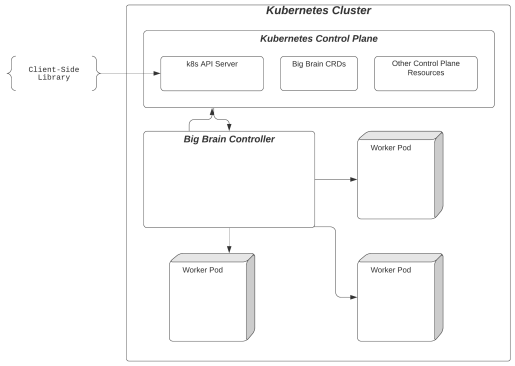

Figure 4. Kubernetes system diagram

2.2 Resource Management

Two resources that matter for us are GPU compute and GPU memory. The safety property we aim for is that GPU should never have an OOM error. So the scheduler must be aware of available GPU memory at all times. A global resource manager maintains the record of memory changes at all GPUs as jobs enter and leave. On the other hand, it is difficult to keep track of the compute resource, since it is handled by the GPUs internal scheduler and context handler. Allotting more jobs than available compute does not lead to errors, but may cause severe slowdown. Thus we had to control how many jobs are allotted to a GPU, even when memory availability may allow it. This was simply done by having a tunable parameter 'packing factor' that scaled up the memory requirement of the job. Thus greater the parameter, less jobs will be packed together.

2.3 Baechi

Once a job is received, it is passed through a profiler. The profiler runs some iterations of the training loop and constructs a graph. Nodes in the graph are layers of the model. Nodes are tagged with the net memory requirement of that layer and forward computation time. Edges show data dependency among the layers and are tagged with the computation time (proportional to the size of data being transferred). The graph and resource manager are then input into an LP solver to get a good placement of layers across the GPUs [Details in Baechi paper]. The model is then distributed across GPUs as per the placement and resource manager is updated.

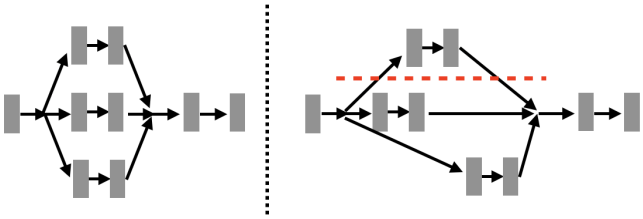

Figure 5. Job Model & the model split

3 Experiments

3.1 Setup

The Scheduler is written in python, with PyTorch for defining the ML jobs.

Sharing a GPU among multiple processes is not without cost. Ideally, if the compute units (cuda cores) of GPU are not saturated then two processes launched together must execute in parallel. However, this is not true in practice because each process has its own context and the GPU can operate within only one context at a time. Therefore, execution of two processes can be serial in nature (though entire detail is not in public domain). Thus, sharing GPU increases the

execution time for all the processes involved. This is has been studied previously as in [12]. On the other hand, if the GPU were not shared, the jobs would be waiting in the queue. There is a tradeoff between waiting time and processing time.

We assume incoming jobs are all training jobs for the model shown in Figure 4. Jobs arrive as a Poisson process. The model has only linear layers. When a job arrives, based on the resource available, Baechi creates a model split for the incoming model.

We had two experimental setups. One of them had 4 GeForce RTX 2080 with 8GB memory each. Other had two RTX 3090 Founders Edition GPUs with 40GB memory each.

3.2 Key Questions and Results

The key question we had to answer for this project was: *Is it ever beneficial to share GPU among multiple jobs?*

We found that under no scenario, sharing jobs on a GPU is beneficial, except when the underlying hardware itself supports granular compute resource allocation (eg: Multi-instance GPUs, Multi-process services). In particular we considered these scenarios:

3.2.1 Simple Packing

In simple packing, we pack entire training jobs, without splitting the model across GPUs. The result is for a 100 sec run with 3 available GPUs. Packing number indicates how many jobs can are allowed to be run on one GPU. So with packing number of 2 and with three GPUs, atmax 6 jobs can run simultaneously. For each case we measure, GPU utilization as shown by nvidia-smi, 'Waiting Time' - the time between arrival of a job and when it is placed on a GPU, 'Process Time' - the time from when job is placed on GPU to its completion. Jobs completed is the total number of jobs processed by the 3 GPUs in 100sec. The process time degrades almost proportionally to packing number, such that it cancels any advantage from simultaneous execution of jobs. Clearly, there is no advantage to packing more. Further, as shown in Figure 5, on packing more, the variation in process time increases thus making the system more unpredictable.

3.2.2 Parallel Models

One scenario in which simultaneous execution might be expected to help is in case of highly parallel deep learning models. Consider the model shown in Figure 4. Splitting the model across two devices along the red line executes faster than on a single device. Thus if there are two such jobs, then we might expect that dividing each job across both the GPUs might be faster than allotting one job per GPU. However this is not true either. Process time degradation eliminates any advantage. Since all jobs are launched from the same process (as different threads), there should not be any context switching costs. However, all tasks are placed in to a default stream by the GPU and is thus serialized on the GPU.

3.2.3 Use of streams

Streams are the basic primitive for parallel task execution on a GPU. So we might expect that launching each job on a new stream would be advantageous. However, this is not observed, because streams only interleave execution of two jobs. It does ensure their concurrent execution

Thus unless there is hardware support for simultaneous execution of tasks, sharing GPU's across job is not useful.

Experiment Duration = 100 sec			
Packing Number	1	2	3
Utilization	17-20%	18-22%	19-22%
Waiting Time(s)	12.44	13.27	11.58
Process Time(s)	4.58	8.97	13.82
Net Time(s)	17.02	22.24	25.41
Jobs Completed	59	61	59

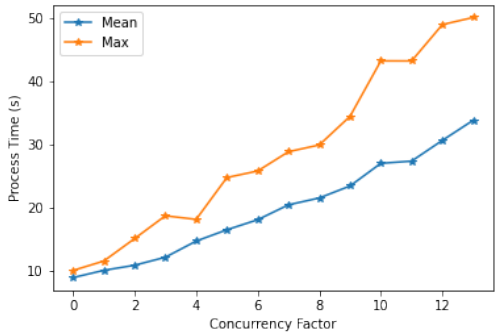

Figure 6. Results

In light of these observations, we are curious about benfits of project like [9] (from Alicloud). They explicitly say they don't solve for isolation among jobs sharing a GPU. In which case, we have observed no speed or utilization advantage.

3.2.4 Multi-process Service (MPS)

One such hardware support is MPS. We used the two-GPU node to run jobs split across the two GPUs.Having the MPS on gave about 12% in the training speed, which is much less the expected ideal 100% speedup. We have not explored the reasons for the same in depth yet.

4 Conclusion

In this project we attempted to combine fast and automated Model Parallelism with deep learning job scheduler. The key objective was to improve GPU utilization by granular GPU allocation while also preventing OOM errors without developer having to explicitly worry about it. The project succeeded in automating the model splitting without developer intervention. However such model splitting is beneficial

only when the model is large enough not to fit on one GPU. We could not demonstrate that model parallelism can be a useful tool in enhancing GPU utilization due to having no support from hardware for task parallelism.

We also built a BBJob abstraction to submit deep learning jobs to such a scheduler. More information about the same is in the video.

5 Metadata

The presentation of the project can be found at:

https://www.loom.com/share/69e060342e6740939578587f7474e930

The code/data of the project can be found at:

https://github.com/iyaja/big-brain

References

[1] AWS GPU instances : https://docs.aws.amazon.com/dlami/latest/devguide/gpu.html.

[2] GCP GPU instances : https://cloud.google.com/compute/docs/gpus.

[3] On-premise solutions : https://www.nvidia.com/en-us/deep-learning-ai/solutions/on-premises/.

[4] Weights and Biases - Monitor and Improve GPU Usage for Training Deep Learning Models: https://towardsdatascience.com/measuring-actual-gpu-usage-for-deep-learning-training-e2bf3654bcfd.

[5] Kubernetes GPU restrictions : https://kubernetes.io/docs/tasks/manage-gpus/scheduling-gpus/.

[6] Kaggle: State of Machine Learning And Data Science 2021: https://www.kaggle.com/kaggle-survey-2021.

[7] Run.ai - Optimal Cluster Utilization: https://www.run.ai/blog/reduce-cost-by-75-with-fractional-gpu-for-deep-learning-inference/.

[8] Nvidia MPS overview: https://docs.nvidia.com/deploy/mps/index.html.

[9] Presented here- https://www.youtube.com/watch?v=zDddDTbEYpE .

[10] Delimitrou, C. Improving resource efficiency in cloud computing.

[11] et al., W. X. Antman: Dynamic scaling on GPU clusters for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)* (Nov. 2020), USENIX Association, pp. 533–548.

[12] et al., X. Gandiva: Introspective cluster scheduling for deep learning. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation* (USA, 2018), OSDI'18, USENIX Association, p. 595âĂŞ610.

[13] Gujarati, A. e. a. *Serving DNNs like Clockwork: Performance Predictability from the Bottom Up.* USENIX Association, USA, 2020.

[14] Jeon, B. e. a. Baechi: Fast device placement of machine learning graphs. In *Proceedings of the 11th ACM Symposium on Cloud Computing* (New York, NY, USA, 2020), SoCC '20, Association for Computing Machinery, p. 416âĂŞ430.

[15] Kwon, W., Yu, G.-I., Jeong, E., and Chun, B.-G. Nimble: Lightweight and parallel gpu task scheduling for deep learning. *NIPS abs/2012.02732* (2020).