

# Big Brain: Efficient Cluster Management for Deep Learning Jobs

Ajay Uppili Arasanipalia, Chirag Chandrahass Shetty  
University of Illinois at Urbana-Champaign  
{aia2, 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 Beachy - 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 [?] [?]. In addition, to manage costs, companies also build and manage their own local cluster[?].

In spite 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 [?] notes that the GPU utilization for over a third of their users is less than 15%, in spite of their users being experienced DL practitioners. There are two primary causes:

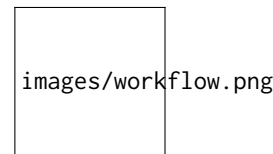



Figure 1. A Typical Workflow

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 [?].
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 [?]

The resulting inefficiencies is illustrated in Figure 1. This over-provisioning coupled with low utilization of the provisioned resource leads to, as [?] 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 [?])

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 [?] exploits the periodic nature of resource consumption in model training to improve cluster utilisation. Clockwork [?] 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 [?] ‘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,



images/pic.jpeg

**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 [?] released a feature to use fractional GPU resources directly through Kubernetes. A highlight of Nvidia's Volta architecture and Multiprocess-Services [?] 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 granularity and allots GPU resources. Further, drawing from our learnings designing Baechi [?] - 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 fit in one single GPU or when using large batch sizes (often data parallelism is preferred for the later case).

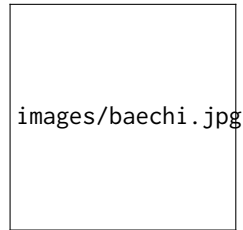
Specifically, we highlight the following:

- 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 [?]
- 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. [?]. In the talk, we will look at how these different components contribute towards building an ideal cluster management system as described in Figure 1.



images/baechi.jpg

**Figure 3.** High Level Design

## 2 Design

### 2.1 System 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 or 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.

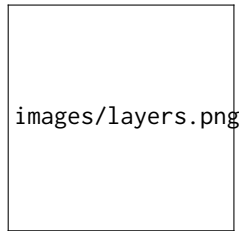


Figure 4. Job Model & the model split

## 3 Implementation

## 4 Experimental Results

### 4.1 Preliminary Experiment

As a first step, we build a toy setup emulating Figure 3. The purpose of the preliminary experiment is to verify the feasibility of job packing into a GPU. Also with the eventual objective of building an easy to use tool, we try to not deviate much from the tools used in the typical workflow used by developers as in Figure 1. Accordingly the system is built as a wrapper over Kubernetes.

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 has been studied previously as in [?]. 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.

The main objective of this experiment is to measure the process-time degradation as more and more process share a

GPU. We ensure that a single job does not saturate a GPU's compute resources (as measured by GPU utilization and power consumption using the linux command *nvidia-smi*).

### 4.2 Experiment Setup

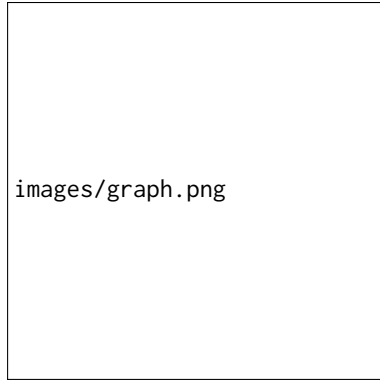
We assume incoming jobs are all training jobs for the model shown in Figure 4. Jobs arrive as a Poisson process. The model has only linear layers. We assume a simply model splitting routine which either places the entire model on an available GPU or splits along the red-line and placed across two GPUs if required. 'train.py' scripts with these two possibilities hard coded are already created. So when a job arrives, the based on resource available, one of the images is submitted to the cluster manager. Eventually this block be replaced with Baechi which can create a model split on the fly for any arbitrary model in the incoming model.

In summary, we built a toy emulator that can handle one type of job with two possible placements. The job queue and scheduler are implemented in Python. Kubernetes is used to manage the cluster and Docker is used to package the job (one for each kind of `train.py`). We note that in Kubernetes if no GPU resources are mentioned in the configuration YAML of the job, then job has access to all the GPUs on the machine the job is scheduled. Thus we specify no GPU resources and this allows us to pack jobs in GPU without involving Kubernetes to allot GPUs to jobs (which would be in integers). But this also means that the scheduler takes care not to cause Out-Of-Memory error on the GPU by allotting more jobs than it can handle. Our setup consists of only one node with two RTX 3090 Founders Edition GPUs.

### 4.3 Results

'Concurrency Factor' is number of entire jobs that are allowed to be packed in to a GPU. Thus a factor of 2 means upto 2 full jobs can run on a GPU. Thus with 2 GPUs, at most four jobs and one job split across the GPUs can run at a time. For a given factor, we run for the experiment for 30 sec with a job arrival rate enough to keep the GPUs always occupied. We measure the mean and max process time among all the jobs that ran during the 30 sec interval. The result is plotted in Figure 5.

One job uses about 25% of a GPU's compute resource. One would thus expect upto 4 process to run on a GPU without affecting each others process-times. However this is clearly not true as shown in Figure 5. Process time increases linearly with the concurrency factor. Further, the variance of the process times also increases as in clear from the max plot (orange). Thus packing of the GPU has diminishing returns and we should find a sweet spot that balances the process time degradation with job queue waiting time.



**Figure 5.** Results

## 5 Future Work

Based on the observations above, the immediate next step is to incorporate better process sharing in a GPU. We note that Nvidia's Multi-Process Service (MPS) provides for this. We will be integrating Baechi into the setup discussed above.

This would allow us to run the same experiment with a realistic and diverse workload. The observations would guide us in adding the intelligence layer in the scheduler that can free the developer from having to deal with the hardware. For example, given a model, is it better to split it across available GPUs or wait until an entire GPU is free.

## 6 Related Work

The related work of your project [? ].

## 7 Conclusion

This project is awesome.

## 8 Metadata

The presentation of the project can be found at:

<https://www.loom.com/share/bc3d2cd2195c4eac8ffc8b565e068cfb>

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

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