Implementing Inference scheduler in Blox

Girish Jonnavithula, Rishideep Reddy Rallabandi, Akhil Perumal Reddy

1 Abstract

The significance of DNN models has increased drastically over the last few decades. The primary objective of this report is to discuss how we implemented Nexus, an inference scheduler on the BLOX toolkit that is used to simulate and deploy ML training/inference schedulers. This is very helpful in analyzing and simulating how schedulers react to different workloads by giving researchers the option to run experiments on several models. Training schedulers like Gandiva have already been implemented on BLOX. Inference schedulers are different from training schedulers as they have an additional constraint of latency, due to which we just can't focus on GPU throughput. Our focus is on implementing the Nexus inference scheduler on BLOX and then running some experiments under different configurations. We have also come up with a few optimizations on top of the Nexus model in this report.

2 Introduction

Deep Neural Networks (DNNs) have facilitated enormous advancements across various applications, including image classification [7], language translation [6], self-driving cars, and video captioning. This widespread adoption of DNNs on various workloads and the pursuit of achieving higher accuracy using larger data-sets has led to extensive research in distributed training systems [9].

DNN applications typically have two phases: *Training* (Learning) which determines the weights of a DNN model by adjusting them repeatedly until the model produces desired results; and *Inference* (prediction) uses the DNN model developed during the training process to render predictions. Today several distributed deep-learning frameworks have been designed to expedite the process of achieving higher accuracy by using techniques such as Data Parallelism [9], Model Parallelism, and Pipeline parallelism [8] for DNN training jobs. In contrast to training, inference jobs do not involve complex iterative algorithms but have an extra restriction on latency.

Organizations today have various machine learning jobs running on their private clusters that share resources using resource schedulers whose primary goal is to decide on how these diverse resources are allocated to several jobs. Huge training data-sets and the iterative nature to achieve high accuracy make training jobs to be long-running and computationally expensive. However, Inference jobs are latency bound making it important to scale the models across multiple resources in a cluster to meet their SLOs'. Scheduler frameworks such as Gandiva [10], Gavel [4], Clipper [3], and Nexus [5] focus on maximizing the utilization of expensive resources by efficiently scheduling resources to expedite either the training time for Training Jobs or to meet Service Level Objectives in case of Inference Jobs (SLOs).

In this paper, we will be working with BLOX- a modular toolkit providing boilerplate implementation of several abstract components (such as Queues, Profilers, and Schedulers) used by most of the Distributed Training Scheduler Frameworks. BLOX helps researchers to directly plugged-in the abstract components provided by the framework to develop, design and test several DNN Training schedulers using it. Our Primary focus would be to study how abstractions provided by BLOX be extended to develop Inference scheduler frameworks such as Nexus. Adding to this if we were successful in implementing the inference schedulers, we would further comparatively analyse how Nexus performs with metrics like GPU Utilization and percentage of requests handled within latency SLOs.

3 Related Work

In present-day scenarios, a single cluster is used for various big data jobs. So, an effective scheduling of these resources is required to obtain high cluster utilization. Over the past decade, we have seen a lot of resource scheduling frameworks that try to handle this scenario. One of the first scheduling policies is the static partitioning of the cluster where each job gets a fixed share of the cluster to use. This policy has a big disadvantage when we have non-uniform workload jobs. The resources will be idle if the workload is less and the resources won't be sufficient if the workload is high. So, to handle this condition we were introduced to a new framework known as Mesos [2] which is a hierarchical scheduling framework. Mesos scheduler decides how many resources of different types can it hand out to a job based on the DRF allotment policy [1]. Now the jobs can decide to take or reject the resources offered. If the job takes the offer, then it can schedule its own tasks using these resources.

The Mesos framework is good for normal big data jobs but it is not efficient for Deep learning models because they have constraints on latency and most of the deep learning jobs are long-running jobs which might result in convoy effect and starvation for other jobs. So, for deep learning models, we need to use different frameworks based on the workload goals.

Deep Neural Networks models have two parts - training and inference. We first discuss about the training jobs. Deep learning training jobs are resource-intensive and time-consuming because we need to iterate over training data and calculate the loss function and update the parameters. So, we need to optimally use accelerators to speed up the training process and should not block the resources till job completion. Scheduler frameworks like Gavel help to maximize cluster utilization by considering the heterogeneous behavior or locality of the resources.

On the other hand, Inference jobs are the processes that evaluate a model to render predictions. The primary goal of inference schedulers is to distribute the large incoming workload onto a cluster of accelerators to obtain high accelerator utilization with low latency and this is important because each job has an SLO requirement to meet. Now, in this paper, we are focusing on implementing the Nexus inference scheduler on the BLOX framework which was developed primarily for deploying, designing, and testing deep learning training schedulers.

4 Design and Implementation

4.1 Nexus

We have scheduling frameworks like Gavel, and Gandiva, which are used to maximize the utilization by efficiently scheduling resources for training but they are not suitable for inference jobs since inference jobs are latency bounded.

One of the primary goals of inference schedulers is to distribute the large incoming workload onto a cluster of accelerators at high accelerator utilization and acceptable latency. There are several challenges in attaining this goal. First, the scheduler should select and schedule jobs on GPUs, so as to maximize their combined throughput while satisfying latency bounds. Second, our scheduler should support running complex queries i.e. jobs that consist of groups of DNNs that feed into each other. In this report, our focus is only on the first point.

Nexus is an inference scheduler that attains high execution throughput under latency Service Level Objectives (SLO). It relies on various policies like the placement of models on GPUs and batching of queries. Additionally, it uses a novel batching-aware scheduler that performs bin packing when the models being packed into bins have variable sizes, depending on the size of the batch they are in. The key policy in Nexus is to batch requests and adapt batch sizes online under a latency SLO to maximize serving throughput under latency constraints.

In this paper, our primary focus was on the squishy-bin algorithm. For instance, consider a scenario where there is a user running three different models like ResNet, LeNet, VGG. Similar to FIFO, we first try to allocate the high requests by running each model in a separate GPU with the maximum batch size which can satisfy their respective latency bound. After this, we are left with residuals from each model. If we run each model's residual in a separate GPU then we will be under-utilizing it. So, we combine the residuals of multiple models in a single GPU such that we satisfy the latency bound and improve GPU utilization. Please refer to Nexus paper [5] for more details on the squishy-bin algorithm.

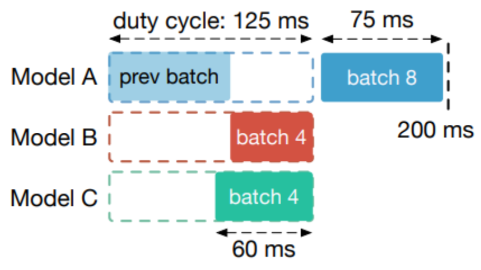

Figure 1: Squishy-bin algorithm

4.2 BLOX

In this paper, we implemented the inference scheduler Nexus on the BLOX framework-a modular toolkit providing boilerplate implementation of several abstract components required to simulate and deploy cluster scheduling frameworks that focus mostly on Machine learning training workloads. Despite having several differences between Machine learning training and inference workloads, the cluster scheduling frameworks generally have common components that are generic to both training and inference workloads. In order to utilize and extend these abstractions to inference schedulers one has to have an overview of the components/abstractions provided by the BLOX framework. This section briefly explains about the in-built components and how these abstractions were extended to develop Inference scheduler frameworks.

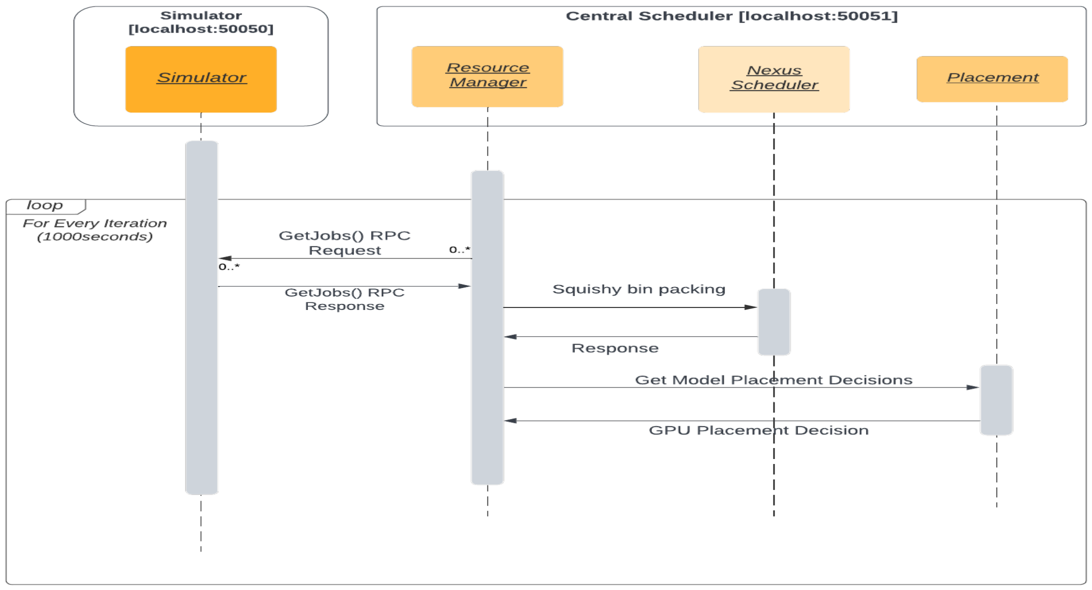

Figure 2: **Implementation of Nexus on BLOX**

4.3 Simulator

The First abstraction as part of the BLOX framework is the Simulator component which is responsible to create and submit jobs to the Central scheduler's Resource manager module using the gRPC channel. As said, Simulator's primary purpose would be to create jobs based on the type of workload with which the user is interested to experiment. Currently, the simulator module supports the following options to generate jobs:

- *Philly Jobs*: Given a set of jobs with resource requirements, the user can configure the arrival of these job being uniform or based on Poisson distribution.
- *Replay Jobs*: Given an input log file, the simulator module creates jobs with the user requirements by parsing the input log file allowing users to simulate real-time cluster workloads by running the simulator with actual cluster logs.

In order to accommodate Nexus, we have introduced *Nexus-Replay* mode which uses the underlying *Replay Jobs* logic to parse log from user-submitted cluster log. Since the Nexus scheduler is batch-specific and non-interruptive, the simulator gets all jobs within a time-frame and forwards these job requests to the Resource Manager. Nexus-Replay uses files that have logs structured as mentioned in figure 3.

$$\left\langle job_{id}, model_{name}, slo, inputRequest_{rate}, arrival_t \right\rangle$$

Figure 3: Log Structure of Nexus-Replay file

4.4 Central Scheduler

Central Scheduler is the most generic component that would be required in any cluster scheduling framework. It is the gateway through which jobs are allocated resources. In analogy, Central Scheduler can be thought of as the Mesos master that keeps track of important information such as:

1. Current state of Jobs which are submitted to Central Scheduler.
2. Current state of resources(.i.e state of GPUs, CPUs) by communicating to Node manager through gRPC channel.
3. It also book-keeps about the jobs that are currently running on a given GPU.

In this section, we will focus on the internal modules present within the Central scheduler and how these modules help the Central scheduler to take intelligent decisions to maximize cluster performance and satisfy user requirements.

4.4.1 Resource Manager

The Resource Manager module can be referred to as a routing module that does the following tasks:

1. Talks to the Simulator application to get the set of jobs that are to be scheduled for the next Iteration.
2. Forwards the set of jobs to the Scheduler module.
3. On receiving the response from Scheduler, it then propagates these results to the placement module to get the best accelerators that should be assigned for a given job.
4. Finally, based on these results it communicates with node managers to start jobs for the current iteration.
5. Adding to this, It also constantly pulls metrics from each node manager to verify if the nodes are active.

4.4.2 Scheduler

The scheduler module helps in building an execution plan based on the required policy by aiming to utilize the cluster efficiently. This execution plan is grabbed by the resource manager and forwarded to the placement holder. In our case, the BLOX framework also has a scheduler abstraction and to that, we have integrated our squishy bin algorithm and performed experiments.

In a manner similar to FIFO, we attempt to allocate high demands first by executing each model on a separate GPU with the largest batch size that can fulfill each model's own latency constraint. We are then left with the residuals from each model. We won't fully utilize the GPU if we run the residuals of each model separately. So, in order to fulfill the latency constraint and increase GPU usage, we integrate the residuals of many models in a single GPU.

On top of the squishy-bin algorithm, we made further enhancements. Firstly, we allocated GPUs to requests based on the first come first serve policy. Requests which are beyond the capacity of GPUs are dropped. Finally, instead of making the scheduler allocate models randomly to GPUs, we placed the models in GPUs optimally by making use of the previous iteration's model locality. Please refer to the pseudo-code for this algorithm from 1.

4.4.3 Placement Module

The Placement Module takes input from the Resource Manager and helps in efficiently assigning the required accelerators for the job based on the user requirement to increase the throughput. In the case of Gavel, it places the accelerators close to the training data and helps in decreasing computing time. But we can't use this technique for Nexus because we are getting the data from the front-end server. So for Nexus, we observed that the average model swap time is time-consuming, and decreasing that time will help in improving the throughput.

Algorithm 1 squishy-bin

```
1: procedure SQUISHYBIN($jobs$)                                  ▷ squishy-bin algorithm
2:     $SaturateOutput, residuals \leftarrow SaturateSched(jobs)$
3:     $ResidualOutput \leftarrow ResidualSched(residuals)$
       return $SaturateOutput, ResidualOutput$
4: procedure SATURATESCHED($jobs$)                               ▷ saturate scheduler algorithm
5:     $SaturateOutput, residuals \leftarrow null, null$
6:     for $job$ in $jobs$ do                                    ▷ Loops through each job
7:         $batch \leftarrow argmax_b(2 * latency(b) < job[SLO]$
8:         $saturatedNodes \leftarrow (job[RequestRate]/throughput(batch))$
9:         $residueRate \leftarrow job[RequestRate] - (saturatedNodes * throughput(batch))$
10:        $SaturateOutput.append(job, saturatedNodes)$
11:        $residuals.append(job, residualRate)$
       return $SaturateOutput, residuals$
12: procedure RESIDUALSCHED($residuals$)                         ▷ residual scheduler algorithm
13:    $SaturateOutput, residuals \leftarrow null, null$
14:    for $residue$ in $residuals$ do                           ▷ Loops through each residue
15:        $batch \leftarrow argmax_b(latency(b) + b/residue[residualRate] < residue[SLO]$
16:        $dutycycle \leftarrow (batch/residue[residualRate])$
17:        $occupancy \leftarrow (latency(bactch)/dutycycle)$
18:    sort residuals by occupancy in descending order
19:    $nodes \leftarrow null$
20:    for $residue$ in $residuals$ do
21:        $maxnode \leftarrow null$
22:        for $node$ in $nodess$ do
23:            if occupancy of newnode is greater than maxnode
24:            replace node with max node
       return $nodes$
```

To decrease the average model swap time we stored the previous iteration's Model to GPU mapping in a table. So for every job in the current iteration, we check that table, and if that Model is used in the previous iteration and is still available, then we allocate that GPU, thus reducing the swap time.

Table 1 : Model to allocated GPUs state

Model	GPUs allocated
Model A	GPU1,GPU2
Model B	GPU3

Table 2 : Input to Placement Module

Job and Model	No of GPUs needed
Job1 - Model A	2
Job2 - Model C	1

Let's look at an example, Table 1 shows the Model to allocated GPUs from the previous iteration. Table 2 shows the input placement module received. Assuming we have only 3 GPUs, for the new input, we just need to swap Model B with Model C from GPU3, and there would be no other swaps required for Model A. Whereas in the case of random placement, we get the average swaps required for this input as 5/3 and in the worst case scenario, we would require 2 model swaps. This technique helped in reducing the average model swap time by nearly 60% when tested on a larger workload which we further discuss in the evaluation section.

4.5 BLOX Interactions

Simulation of the Nexus scheduler on this toolkit majorly uses 2 Components of BLOX which are the Central Scheduler and Simulator. Following is the flow of communications between the components of the Central scheduler and simulator:

- After every T seconds, the Resource Manager talks to Simulator using the gRPC channel to get jobs that are to be scheduled for the next T Seconds.

- Post this, the resource manager uses Nexus scheduler to decide on the optimal number of GPUs required for Deep learning inference models that satisfy the latency bounds for each job.

- Once the Resource manager gets the response about the GPU requirements of each job, it decides on placing these jobs either by communicating it to the Node manager(in case of deployment) or by using the Placement module (in case of simulation). Doing this helps us reduce model swaps on GPUs and thus decreases the launch time of the jobs.

5 Evaluation

After integrating the Nexus scheduler (primarily Squishy-bin algorithm) on BLOX toolkit, we ran experiments to evaluate the GPU utilization, bad rate, and average swap time for the cluster. We have also come up with a few optimizations on top of the Nexus model and have shown the benefits of these optimizations by providing an analysis of the results.

5.1 Unit Tests

We have run the following tests to verify the implementation of our model:

- For the squishy-bin algorithm, we have created a test case using the batch profile details from Nexus paper and verified that the squishy-bin algorithm is behaving as expected.

- We ran the FIFO scheduler on the BLOX toolkit and verified that the job scheduling is done on a first come first serve basis.

5.2 GPU Benefit

In order to see the benefit of the squishy-bin algorithm, we computed the number of GPUs required to run jobs ranging from 1 to 500.

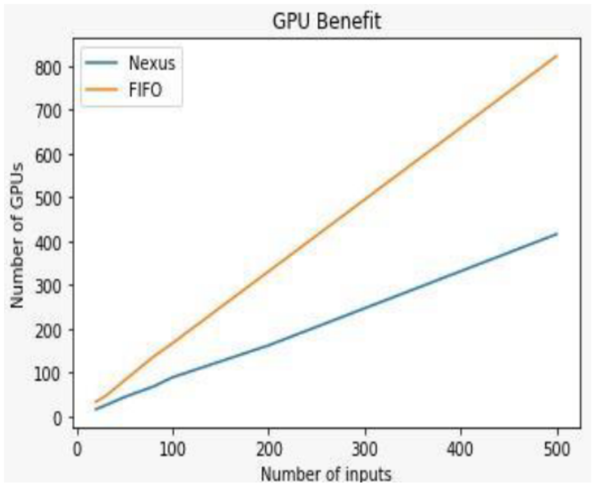

Figure 4: Number of GPUs for various workloads

We generated the jobs randomly and can see from 4 that the number of GPUs required for the Nexus model is always less than that of the FIFO model. This is because in FIFO, we run each residual on a separate GPU, whereas in Nexus, we combine the residuals beforehand, which resulted in less number of GPUs. We also noticed that as the workload increased, the gap between the number of GPUs required for FIFO and Nexus had increased gradually, and for 500 inputs, we observed that the Nexus model requires only 50% of the number of GPUs required for FIFO. Hence, the squishy-bin algorithm is very powerful, which improves GPU utilization and reduces cost.

5.3 Bad Rate

We wanted to see how good the Nexus model is in terms of the number of requests it can satisfy with the given number of GPUs. To benchmark it, we compared the results with the FIFO model. Please refer 5 for more details.

Following are some observations:

- As the number of GPUs is low, we can see that the drop rate is the same for Nexus and FIFO as these few GPUs are utilized in satisfying the saturated jobs.

- When the number of GPUs is very high, both models will be able to handle all the requests. So, the drop rate is zero.

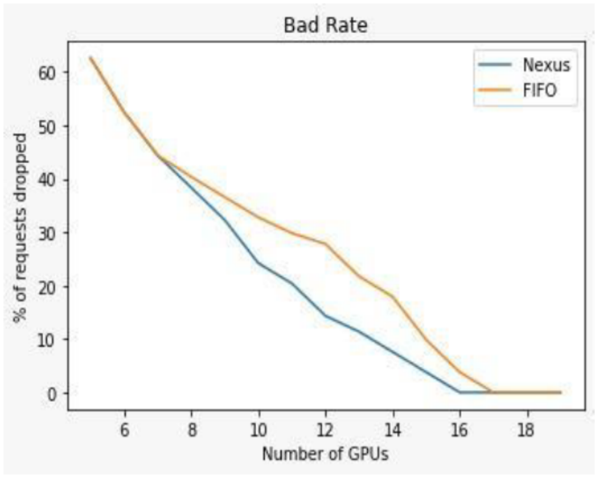

Figure 5: Bad rate vs number of GPUs

- In between, as Nexus uses less number of GPUs, it helps in handling more number of Jobs. Thus resulting in a reduced bad rate.

5.4 GPU Placement

In Nexus, after every iteration, we randomly place models (required for the next iteration) on GPUs. From our observations, we could still optimize the amount of time taken for loading these models on the GPUs, if we make use of the model allotments of the previous iteration. So, instead of making the scheduler allocate models randomly to GPUs, we found that the average swap time into GPUs would drastically reduce if we consider the previous iteration's model locality. We have verified this by running an experiment. The jobs submitted in this experiment used 25 different Deep learning models over the course of 30 scheduling iterations. Making sure that the Central Scheduler took model placement decisions based on global Model-to-GPU mapping helped to achieve an overall reduction in model swaps on GPUs.

From figure 6, we observed that the average hit rate with our Model-GPU placement technique is around 60% and with random placement, it was near 15%. That means our technique gave a 4x better performance when compared with random placement. Along with that, we noticed if we increase the number of unique models to schedule, the overall cache hit rate decreases. So in conclusion, this approach performs best for workloads that have a set of models which are scheduled repetitively.

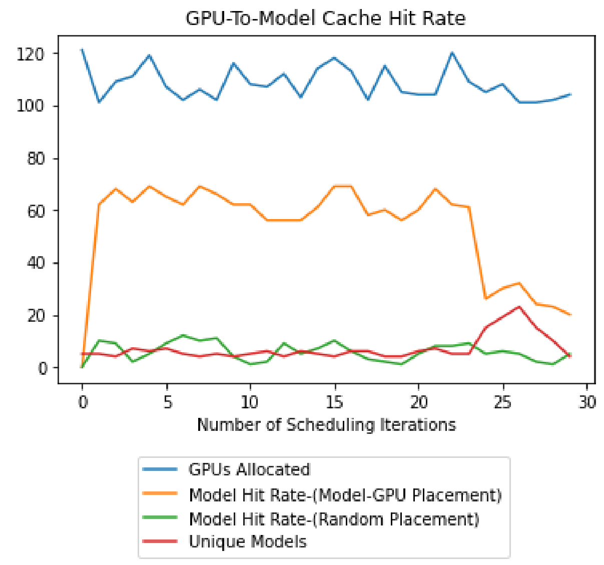

Figure 6: GPU-to-Model cache-hit rate

6 Conclusion

One of the primary objectives of our project is to check if we can integrate an inference scheduler in the BLOX framework which helps researchers to deploy, simulate and compare several DNN cluster scheduling techniques. Previously, training schedulers like Gavel are already implemented in BLOX. Inference schedulers can't just optimize GPU utilization like training schedulers because of latency constraints. We used the Nexus model for this project. First, we deployed the Nexus model (squishy-bin algorithm) on the BLOX toolkit and then ran few experiments on it. We observed the benefit of the squishy-bin algorithm in terms of the number of GPUs and bad rate. We also explored some enhancements to the current Nexus model by optimally placing the models on GPUs. We implemented Squishy-Bin packing on BLOX Framework and made our code available on GitHub (https://github.com/msr-fiddle/bebopblox-private/tree/BDS_744_2022_V2).

7 Future Work

There are a lot of things that can be explored in this project. Following are some possible things to explore in the future:

- We want to verify if clubbing request rates for different jobs with the same model and SLO will have optimal GPU utilization.
- We are currently using batch profile data from other

generic sources. We can implement a cluster-specific batch profiler.

- Bench-marking of Nexus with other inference schedulers like Clipper and Clockwork.
- Integrating both training and inference schedulers.

References

[1] A. ghodsi, m. zaharia, b. hindman, a. konwinski, s. shenker, and i. stoica, "dominant resource fairness: Fair allocation of multiple resource types," in proc. usenix nsdl, (2011).

[2] B. hindman, a. konwinski, m. zaharia, a. ghodsi, a. d. joseph, r. h. katz, s. shenker, and i. stoica. mesos: A platform for fine-grained resource sharing in the data center. in nsdi, 2011.

[3] Daniel crankshaw, xin wang, guilio zhou, michael j franklin, joseph e gonzalez, and ion stoica. clipper: A low-latency online prediction serving system. in proceedings of the 14th usenix symposium on networked systems design and implementation (nsdi), 2017.

[4] Deepak narayanan, keshav santhanam, fiodar kazhamiaka, amar phanishayee, and matei zaharia. 2020. heterogeneity-aware cluster scheduling policies for deep learning workloads. in 14th usenix symposium on operating systems design and implementation (osdi '20).

[5] Haichen shen, lequn chen, yuchen jin, liangyu zhao, bingyu kong, matthai philipose, arvind krishnamurthy, and ravi sundaram. nexus: a gpu cluster engine for accelerating dnn-based video analysis. in proceedings of the 27th acm symposium on operating systems principles (sosp), 2019.

[6] J. devlin, m.-w. chang, k. lee, and k. toutanova. bert: Pre-training of deep bidirectional transformers for language understanding. arxiv preprint arxiv:1810.04805, 2018.

[7] K. he, x. zhang, s. ren, and j. sun. deep residual learning for image recognition. in proceedings of the ieee conference on computer vision and pattern recognition, pages 770–778, 2016.

[8] Narayanan, d., harlap, a., phanishayee, a., seshadri, v., devanur, n. r., ganger, g. r., gibbons, p. b., and zaharia, m. pipedream: Generalized pipeline parallelism for dnn training. in proceedings of the 27th acm symposium on operating systems principles, pp. 1–15, 2019.

[9] Shen li, yanli zhao, rohan varma, omkar salpekar, pieter noordhuis, teng li, adam paszke, jeff smith, brian vaughan, pritam damania, et al. pytorch distributed: Experiences on accelerating data parallel training. arxiv preprint arxiv:2006.15704, 2020.

[10] Wencong xiao, romil bhardwaj, ramachandran ramjee, muthian sivathanu, nipun kwatra, zhenhua han, pratyush patel, xuan peng, hanyu zhao, quanlu zhang, fan yang, and lidong zhou. 2018. gandiva: Introspective cluster scheduling for deep learning. in 13th usenix symposium on operating systems design and implementation (osdi 18). usenix association, carlsbad, ca, 595–610.