Scheduling Algorithm

Iman Tabrizian

July 10, 2020

1 Introduction

With initial results showing the potential performance gain for co-locating ML jobs there are a couple of questions that need to be answered:

1. What is the average GPU utilization for jobs in the data centers?
   (a) Using MSR Philly [1] trace as a reference for low GPU utilization in data center environments.
2. What is a good heuristic to schedule the jobs?
   (a) Add a job to the node if the total GPU utilization doesn't exceed a certain bound?
   (b) Add a job to the node if the total GPU memory utilization doesn't exceed a certain bound?

2 Algorithm

Depending on what algorithm we use and what criteria we are trying to maximize different algorithms can be proposed. The main goal of this body of work is to minimize total job completion time (JCT).

```
   Data: Pods, nodes, queue
   Result: Placement of Pods on the nodes
 1 while queue is not empty do
 2 |   job ← findShortest(queue);
   |   // Find the nodes with available capacity
 3 |   nodes ← filter(nodes, job);
 4 |   if len(nodes) > 0 then
   |   |   // Find the node that maximizes throughput for a
   |   |      given job
 5 |   |   node ← MaximizeThroughput(nodes, job);
 6 |   |   queue.remove(job);
 7 |   else
   |   |   // Can't find a node with enough capacity
 8 |   |   queue.moveBack(job);
 9 |   end
10 end
```

Algorithm 1: Main Scheduling Loop

3 Monitoring and Metrics Collection

For monitoring and collecting metrics we focus on two most important aspects. The first part is collection of systems performance data such as GPU utilization, GPU memory bandwidth utilization, CPU usage, and memory usage.

The second part is collection of application level monitoring data. For the specific problem that we are currently studying, it is important to collect job throughput, job GPU memory usage, job CPU memory usage, and job DRAM usage.

There are some other meta data about each training job that needs to be stored. For example, we need to calculate how many iterations it is required until the end of the training and how long does each iteration take. This will help us identify the job that completes faster and schedule that job first.

4 Modeling the Performance of Co-located Jobs

In order to maximize the throughput of co-located jobs we need to have some kind of performance models to effectively predict how the performance will be.

Each training job needs to perform many different tasks that some of them are running on CPUs (e.g. data loading, logging, benchmarking), some tasks that are running on GPUs (kernel execution), and some other tasks that involve collaboration between GPUs and CPUs (e.g. copying data to GPU memory).

4.1 What Happens When Jobs Run Simultaneously on GPUs?

Sharing a GPU between different CUDA contexts can have the following effects:

1. Time Multiplexing: When the GPU is time multiplexed
2. Space Multiplexing: When the GPU is space multiplexed and they share the GPU space

4.2 How to Model the Performance of Co-Location

Assume that kernel k_i^j, takes t_i^j time for execution. If the kernels are executed sequentially, the total time would be $T_j = \sum_{i=1}^{n} t_i^j$. The assumption here is that kernels will not run concurrently and they will execute in order.

For each k_i^j there are m_i^j thread blocks that need to be scheduled. Assume that our GPU model supports P SMs and each SM can handle c_i^j thread blocks running concurrently for specific kernel k_i^j. By utilizing all the SMs on our GPU, we can run $P * c_i^j$ of kernel k_i^j. The factors that limit the number of thread blocks running concurrently on a GPU are below:

1. Amount of shared memory required by each of the kernels (s_i^k)
2. Total number of threads that each thread block uses (h_i^k)
3. Total number of registers required (r_i^k)

If we denote the total number of threads that each SM supports as H, total number of registers as R, total amount of shared memory as S, and total number thread blocks supported by the SM as B. Also, each kernel requires $\gamma_i^j = \left\lceil \frac{m_i^j}{P.c_j^i} \right\rceil$ iterations to run completely. c_j^i can be calculated using Eq. (1). Each group of thread blocks being scheduled together takes constant time θ_i^j

$$c_i^j = \min(\left\lfloor \frac{S}{s_i^j} \right\rfloor, \left\lfloor \frac{H}{h_i^j} \right\rfloor, \left\lfloor \frac{R}{r_i^j} \right\rfloor, B) \tag{1}$$

Our objective function is depicted in Eq. (2) where $J_j = \sum_{i=1}^{n} t_i^j$.

$$\min \sum_{j=1}^{N} J_j \tag{2}$$

We can (unsafely) assume that $c_i^j \propto \frac{1}{t_i^j}$. This means that if more blocks are being scheduled for a given kernel, the kernel will run faster. This is not completely true in the practical sense. However, this approximation may help us better understand the performance in this specific context.

Then the total execution time of job J_j is equal to:

$$\sum_{i=1}^{n} \theta_i^j . \gamma_i^j \tag{3}$$

Eq. (3) shows that if we are able to run as many thread blocks as possible, we will be able to run the kernel faster since it will decrease γ_i^j without changing θ_i^j.

Eq. (3) should (hopefully!) work when the job is running alone. However, things we'll be more complicated when we are going to share the GPU spatially with other jobs.

4.3 Modeling Co-Location of Two Jobs Spatially

When the jobs are being co-located, c_i^j is no longer constant. The reason is that GPU resources such as S, H, B, and R maybe in use by other CUDA contexts. In this case, $c_i^j(S, H, R, B)$ is a function of remaining S, H, B, and R values. We can add indices to these variables to represent the number currently in use by context j.

$$c_i^j(S_i^j, H_i^j, R_i^j, B_i^j) \tag{4}$$

5 Benchmarking and Verification of Claims

Formulation is not very popular in systems community usually due to the complicated aspects of systems. In this section, we try to run a couple of benchmarks to verify the results of formulation.

5.1 Claim 1: If the Thread Blocks Are Scheduled Simultaneously the Time is Constant

References

Jeon, M., Venkataraman, S., Phanishayee, A., Qian, J., Xiao, W., & Yang, F. (2019). Analysis of large-scale multi-tenant ${$gpu$}$ clusters for ${$dnn$}$ training workloads. In *2019 ${$usenix$}$ annual technical conference (${$usenix$}$${atc}$ 19)* (pp. 947–960).