Kube-Knots: Dynamic Container Orchestration and Resource Management for GPU-based Datacenters

Abstract

Compute heterogeneity in the form of accelerators like GPUs and FPGAs are increasingly gaining prominence in modern datacenters. We observe that datacenter schedulers are agnostic of these emerging accelerators and their resource utilization footprints, and thus, not well equipped to dynamically orchestrate them by providing fine-grained resource guarantees to latency sensitive applications. Specifically for GPUs, this results in resource fragmentation and interference issues leading to overall resource under-utilization. Furthermore, GPUs are not power proportional and hence proactive management of these resources by the cluster-wide resource orchestrator is essential to keep the operational costs low while ensuring the end-to-end Quality of Service (QoS).

In this paper, we identify the challenges in developing a GPU-aware datacenter scheduler and explore the potential opportunities that arise by exposing GPU metrics to the datacenter scheduler. Based on our observations from Alibaba's cluster traces and real hardware GPU cluster experiments, we build *Knots*, a GPU-aware resource orchestration layer and integrate it with *Kubernetes* container orchestrator. We evaluate three GPU-based scheduling techniques to schedule representative GPU workload mixes through *Kube-Knots* on a ten node GPU cluster. Our proposed Peak Prediction (PP) scheduler improves both average and 99^{th} percentile cluster-wide GPU utilization by up to 80%. This leads to 33% cluster-wide power savings on an average for three different workloads compared to state-of-the-art utilization-agnostic schedulers. Further, the proposed PP scheduler ensures end-to-end QoS for latency critical GPU-bound queries by reducing QoS violations up to 53% when compared to GPU-agnostic scheduling policies.

1 Introduction

Modern data centers are being provisioned with compute accelerators such as GPUs and FPGAs to catch up with the workload performance demands and reduce the Total Cost of Ownership (TCO) [18, 26]. By 2021, traffic within hyperscale datacenters is expected to quadruple with 94% of workloads moving into the cloud, according to Cisco's global cloud index [4]. Based on the nature of the application, it can either benefit from the high throughput of GPUs or the low latency present in FPGAs and TPUs [29]. This trend is evident as public cloud service providers like Amazon [3] and Microsoft [10] have started offering GPU and FPGA-based infrastructure services.

In recent years, GPUs have gained prominence due to the increasing computational demands of Deep Learning workloads. This includes both user-facing inference queries and batch-style model training. With the expected increase in such workloads [4], public clouds have provisioned at the scale of thousands of GPU-nodes in datacenters. Since GPUs are relatively new to the cloud stack, support for efficient GPU management lacks as state-of-the-art cluster resource schedulers [8, 17] treat GPUs just as a specific resource constraint ignoring its unique characteristics and capabilities.

In this paper, we address the issues present at the cluster-level resource management layer, which is heavily tuned for CPU-based cloud stacks. Existing production datacenters do not leverage GPU-specific scheduling policies while neglecting both the application-specific characteristics and accelerator utilization metrics. Further, we show how GPU-specific resource management policies improve cluster-wide GPU utilization. Thus, it reduces the datacenter operational costs for service providers.

State-of-the-art resource orchestrators such as Kubernetes [17] and Mesos [8] perform GPU-agnostic uniform scheduling, which statically assigns the GPU resources to the applications. The scheduled containers/pods[1] access the GPUs via PCIe pass-through, which gives guest applications the complete access to GPUs bypassing the hypervisor. Hence, none of the hypervisor-level resource management policies such as fairness, utilization, etc., could be incorporated across multiple tenants on GPUs. Further, Kubernetes supports dynamic orchestration by performing node affinity, pod affinity, and pod preemption in the case of CPUs. However, with respect to GPUs, pods have exclusive access to the device till completion and they are non-preemptible from the hypervisor. Based on this observation, we identify two problems that make GPU-specific resource management challenging.

• **Non-preemptive GPU pods -** The pods scheduled to GPUs cannot be preempted, or context switched, or prioritized and are scheduled based on First Come First Serve (FCFS) order [16]. Therefore, GPU-bound latency sensitive queries when queued behind a batch job will incur severe queuing delays.

[1] We use the terms (Google's) pod and container interchangeably.

• **Utilization agnostic GPU scheduling -** Today's datacenter resource orchestrators treat GPU's as a hardware constraint and are agnostic of GPU utilization metrics. Ensuring the QoS of a latency sensitive query is difficult without knowing the system state due to performance interference in the case of multi-tenancy.

To overcome these limitations, we propose *Kube-Knots*,[2] which enables GPU utilization-aware orchestration along with QoS-aware container scheduling policies. Towards this end, we design *Knots* that aggregates real-time GPU cluster utilization metrics which is leveraged by our proposed Peak Prediction (PP) scheduler. PP scheduler performs safe co-locations through dynamic container resizing along with QoS-aware GPU scheduling. This in contrast to the state-of-the-art GPU scheduling techniques that focus on only one type of application such as DDNN [28, 35, 37, 40, 44, 46]. In addition, *Kube-Knots* performs QoS-aware container colocations on GPUs at the cluster-level without the apriori knowledge of incoming applications when compared to prior node-level techniques [20, 31–33, 42] which require prior profiling of applications.

Using *Kube-Knots*, we make the following contributions:

1. We build three schedulers namely Resource Agnostic **(Res-Ag)**, Correlation Based Prediction **(CBP)**, and Peak Prediction **(PP)** that leverage the cluster-wide GPU utilization time-series data aggregated from *Kube-Knots*.
2. We generate datacenter representative GPU cluster workloads to evaluate our proposed schedulers on a **real-system with multi-node GPU** cluster orchestrated by *Kubernetes* with *Knots*.
3. Our proposed CBP scheduler leverages the real-time GPU usage correlation metrics to perform safe pod co-locations ensuring crash-free **dynamic container resizing** on GPUs.
4. CBP along with PP scheduler can guarantee the end-to-end QoS of latency critical queries by predicting the incoming load through ARIMA [1] based time-series forecasting. This further reduces the QoS violations by up to 53% when compared to the resource agnostic scheduler.
5. *Kube-Knots*[3] improves the **cluster-wide GPU utilization** through performance-aware workload consolidation by up to 80% for both average and 99^{th} percentile utilization when compared to resource agnostic scheduler. Thus reducing the datacenter wide GPU **energy consumption by 33%** on an average when compared against GPU-agnostic orchestration (Res-Ag).

[2]Google's *Kubernetes* means helmsman or captain of the ship managing multiple containers. In *Kubernetes*, *Knots* orchestrates the scheduling speed of the accelerated containers.

[3]We plan to release *Kube-Knots* as an open-source plug-in patch for *Kubernetes*, that could be leveraged by the community to dynamically orchestrate GPU-based containers via Kubernetes.

2 Background

Traditional resource schedulers for CPU-based datacenters improve the overall utilization by virtualization. However, GPUs pose unique challenges in compute virtualization. Hence, public cloud companies like Amazon do not virtualize accelerators like GPUs and FPGAs. But in case of private datacenters, to improve the GPU utilization, time-sharing of the GPU compute cores and space-sharing of the GPU memory is enabled. In this section we discuss the properties unique to GPUs and potential implications of sharing as it leads to QoS violations.

2.1 Utilization vs Performance per Watt

The processing elements of CPUs are designed to operate for an average case load at peak energy efficiency [43] in terms of performance per watt (PPW). As seen in Figure 1 the energy efficiency varies at different load scenarios for CPUs and GPUs. The newer generation CPUs are much more energy proportional when compared to older architectures. For the CPU-bound queries, we observe that the peak energy efficiency at around 60% to 70% core utilization, where the point of peak efficiency is no longer at peak utilization for CPUs.

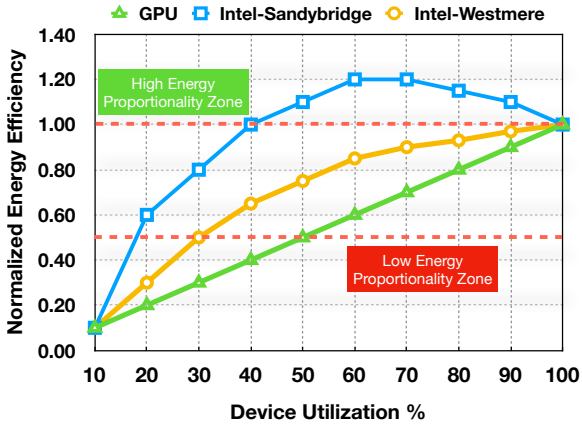
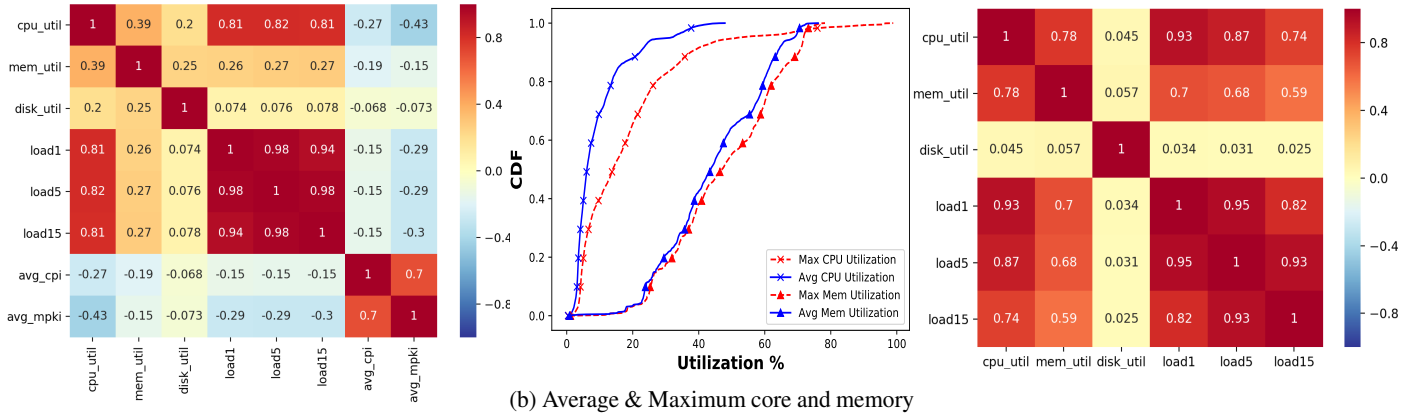


Figure 1: Energy Efficiency (EE) of CPU and GPU at varying utilization %. EE normalized to EE at 100% utilization.

We observe that the GPUs are fundamentally different from CPUs as their energy efficiency has a linear relationship with respect to their utilization. Therefore, a cluster-level resource manager needs to pack the workloads to operate the GPUs at 100% utilization. However, it is not practical to operate all the GPUs in a cluster due to diurnal load interval trends. The GPU cluster scheduler needs to actively consolidate the load among minimum number of GPU nodes while letting other GPU nodes to be in deep sleep power state. Therefore, scheduling policy in terms of utilization should be even more aggressive when compared to CPUs while ensuring the performance of the pods.

Observation 1: *Keeping the GPU utilization high is essential for high energy efficiency.*



(a) Latency-critical task's usage metrics

(b) Average & Maximum core and memory utilization of containerized services.

(c) Long running batch task's usage metrics

Figure 2: Alibaba trace analysis of resource utilization of 1300 machines across 12,951 batch jobs & 11,089 containers (12hr period).

2.2 Cluster-level Trace Analysis

In order to obtain very high utilization in GPUs, we needed to know the user behavior in terms of resource request and resource usage trend. To this end, we analyze the open-sourced Alibaba production datacenter traces [2] to gain insights into the nature of batch and latency critical jobs that are submitted to the datacenter. Through this, We understand the datacenter workload arrival patterns along with their actual resource demands and actual usage patterns.

• **Resource overcommitment -** First, jobs tend to overstate their resource requirements. As observed from Figure 2b, average core and memory utilization is 47% and 76% respectively. Half of the scheduled pods consume less than 45% of the provisioned memory on an average. This is because, over-commitment indirectly helps to ensure the QoS of such queries by provisioning for the worst (peak) case leading to overall underutilization of the cluster. In case of CPUs, virtualization, preemption and fine-grain resource sharing mitigate the problem, while it becomes a critical design point factor as the GPU memory (typically 8-16GB) is limited than server-class CPU memory (64-256GB).

Observation 2: *Due to varying resource needs of an application, instead of provisioning the resources for application's maximum utilization case, it is more resource efficient to provision for the average utilization case. It is ideal if the scheduler could dynamically provision and resize the containers based on the run-time growth of the application instead of static provisioning.*

• **Utilization metrics are tightly correlated -** Figure 2a plots the Spearman's correlation across eight container resource utilization metrics as a heat map. Equation 1 gives the correlation score ρ where, d_i is the difference between the ranks (ordered in descending i.e, highest value gets rank 1 and lowest value gets rank 8) of corresponding utilization metrics in the heatmap and n is the number of observations. Positive relationship between two metrics is represented by a score close to +1 and vice-versa. A score of 0.0 denotes that these metrics do not have any relationship. More positive the relationship, hotter is the color scheme (red). There is a significant relationship between the cores (core_util), average system load recorded every second (load_1) and memory consumption (mem_util) of latency-sensitive service jobs. However there are no clear correlation indicators to predict utilization since these tasks are short-lived in the order of seconds.

$$\rho = 1 - \frac{6\Sigma d_i^2}{n(n^2-1)} \quad (1)$$

Figure 2c plots the correlation between six utilization metrics of batch workloads. It is seen that the memory utilization strongly correlates with the core utilization. Unlike latency-sensitive queries, the batch applications exhibit strong correlation (both +vs & -ve) between its utilization metrics. Thus a container running a batch application on GPUs could be dynamically sized based on the SM_Utilization (GPU core) enabling aggressive bin-packing provisioning for latency-sensitive queries.

Observation 3: *Correlation between different resource utilization metrics provides early markers for proactive admission control. This guarantees the performance of the scheduled application.*

2.3 Node-level GPU Characterization

we identify the popular production workloads which subscribe to GPUs. The workloads include the DNN frameworks like Tensorflow, which are hosted as micro-service deployments inside containers and are shipped via orchestrators like Kubernetes [17], Mesosphere [9], Docker swarm [5], etc. Generally, these deployments are hosted as services such as object detection, feature extraction, etc. Queries to these containers arrive in short bursts, and the tasks are short-lived. For example, the image recognition DNN-based inference query takes 90*ms*

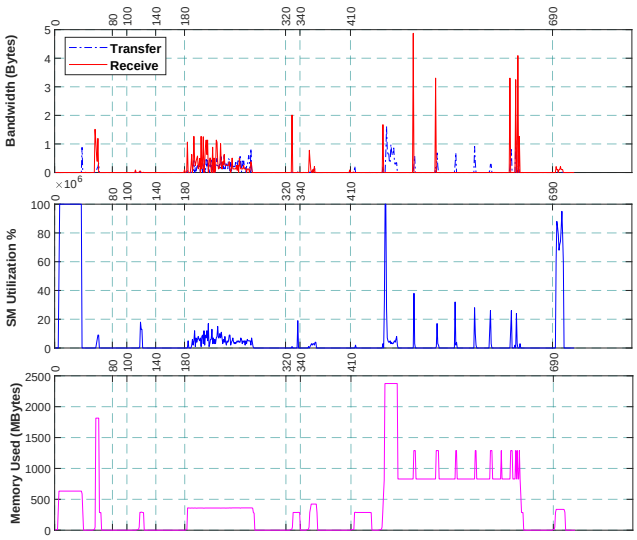


Figure 3: GPU Resource consumption of entire Rodinia suite on Nvidia's P100 GPU. The grid-lines mark the individual benchmark's runtime.(where x-axis is time in milliseconds)

on Nvidia P100 [6] on an average. Whereas, other batch jobs may run on a GPU for hours like an HPC application or even Proof-of-Work (PoW) for crypto-currency transactions.

In order to create a representative workload mix for GPUs, which consists of both batch workloads and user-facing queries: (i) we use applications from Rodinia workload suite [19] for compute-intensive long running workloads and, (ii) we use a mix of DNN based inference queries from Djinn and Tonic suite [26] for user facing queries. We execute the application-mix on a single GPU node to understand the batch-workload's resource demands. The details of the GPU-node is given in Table 2. More details about the workload are explained further in Section 3.

2.3.1 GPU Batch Workload

Figure 3 plots the GPU resource consumption over time for the eight different applications, which were run sequentially. We characterize memory used, Streaming Multiprocessor (SM) utilization and PCIe bandwidth which are three dominant resources for a batch application. It is observed that in general the resource consumption is relatively low with a few exceptions over time. From Figure 3, we also observe that the resource consumption of applications may not always peak at the same time. This temporal nature of the application resource usage can be exploited to avoid resource capacity violations during multi-tenancy. Further, we observe that these applications show a very deterministic pattern of phase changes. For example, typically if an application's input bandwidth activity is high, it is implied that the compute and memory would follow the same trend in the next few milliseconds. These subtle utilization cues could be picked up through real-time feedback and can be exploited during pod scheduling.

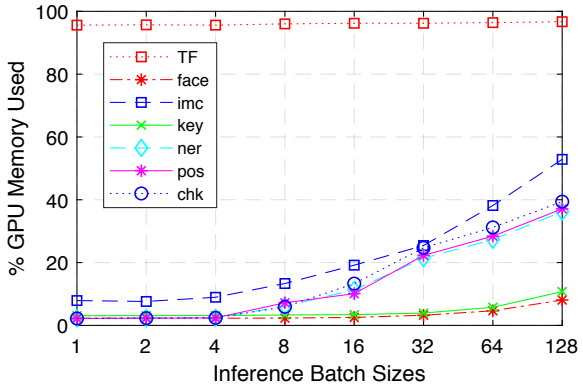


Figure 4: Internal memory fragmentation of DNN inferences of Djinn and Tonic workload suite. TF is the Tensorflow managed memory consumption

Observation 4: *GPU application's utilization footprint is fairly predictable across its lifetime, enabling opportunities for dynamic provisioning.*

2.3.2 GPU User-Facing Queries

The Djinn and Tonic workload uses Tensorflow (TF) as the DNN-framework for executing ML queries. TF also manages the execution flow of the queries that run inside GPU. By default, TF earmarks the entire GPU memory despite actual workload demands. The TF runtime is designed to make this a default choice because GPUs are assumed to run a single context at a given time and are not designed for multi-tenant/application scenarios. However, in public datacenters to keep the costs low, it is ideal to share the GPUs because the individual application/tenant utilization is generally low.

Figure 4 shows the maximum memory consumption of different machine learning inferences. For most of the single inference queries, the memory consumption is less than 10%. We batched these queries up to 128 queries per batch request, however, the majority of the inferences even with batching consume less than 50% of the device memory. Whereas if the same application runs on TF, it consumes 99% of the GPU memory causing severe internal memory fragmentation.

Frameworks like TF would always overcommit the GPU memory by default leading to internal memory fragmentation. This directly affects the scheduler's global policies for fairness and performance. It is ideal for a datacenter scheduler to have two essential properties, namely, sharing incentive and strategy proofness [23]. In this case, neither can be ensured because the cluster-level resource manager has no control over the GPU device. In order to pack more applications to increase the utilization, the applications have to be provisioned by the datacenter scheduler by knowing the real-time GPU utilization metrics.

Observation 5: *To enforce a cluster-level scheduling policy, it is essential to expose the framework APIs to the cluster scheduler to avoid GPU memory fragmentation.*

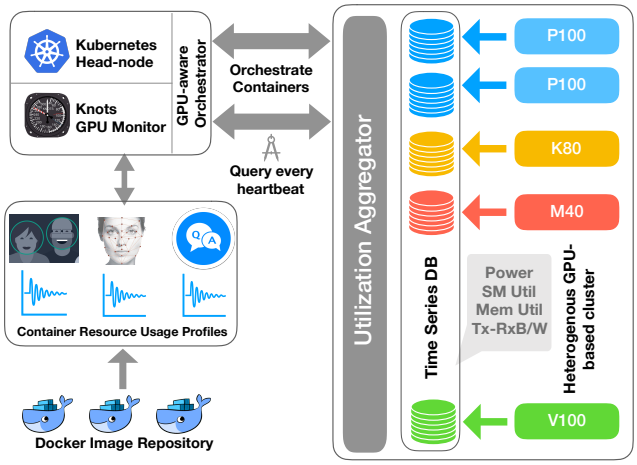


Figure 5: *Kube-Knots* GPU orchestrator design.

Batch workloads from Rodinia suite					Latency critical	Load	COV
App-Mix-1	leukocyte	heartwall	particlefilter	mummergpu	face,key	HIGH	LOW
App-Mix-2	pathfinder	lud	kmeans	streamcluster	chl,ner,pos	MED	MED
App-Mix-3	particlefilter	streamcluster	lud	myocyte	imc,face	LOW	HIGH

Table 1: GPU load and covariance for cluster workload suite mixed with batch jobs and latency critical ML inference queries.

3 Motivation

Accelerators like GPUs in production datacenters are relatively recent, we wanted to faithfully capture the dynamics of applications that would be representative of a production datacenter load. We capture the interarrival pattern of tasks in the Alibaba datacenter traces [2] along with the task resource constraints as discussed in Section 2. Using the arrival rate of incoming jobs to the Alibaba datacenter, we schedule a mix of batch and interactive GPU jobs to our ten node cluster. This mix is determined by a set cut-off threshold based on the pareto principle [38] where 80% jobs consume 20% of the datacenter resources being short-lived tasks and the rest are the long-running batch jobs.

3.1 Representative GPU workload

We choose eight scientific applications from the Rodinia benchmark suite [19] to represent the batch jobs. For short-lived tasks, we used a mix of DNN-based inference queries (speech, text, and image) from Djinn and Tonic workload suite [26] (From Table 1[4]). Based on the resource usage trend in the Alibaba traces, we categorize the application-mix and associate it with three different bins based on the load and covariance as seen in the Table 1.

3.2 Application-mix Analysis

To schedule these workloads onto the GPU nodes, we use the *Kubernetes* default scheduler uses uniform scheduling policy to schedule containers. By default, GPU sharing is disabled as it complicates the scheduler-level performance/QoS guarantees and to ensure fairness across different co-scheduled applications. To set a fair baseline, we enable GPU sharing across multiple applications and we call this as GPU-agnostic scheduler and set it as our baseline scheduler for comparison. Note that the GPU compute is time-shared while the memory is space-shared. Therefore, scheduler can pack multiple containers on GPUs by resizing the containers.

We create an experimental setup as shown in Table 1 and evaluate the baseline Res-Ag/GPU agnostic scheduler. We schedule all three app-mixes and plot the 50^{th}, 90^{th}, 99^{th} percentile GPU utilization along with maximum utilization of individual GPUs as seen in Figure 6. We observe in app-mix-1 (in Figure 6a) the median is much closer to 90^{th}/99^{th} percentile utilization when compared to app-mix-3 (in Figure 6c). This implies that the sustained load in case app-mix-1 is *higher* than app-mix-3, as we classified in Table 1. On the other hand, app-mix-2 in Figure 6b has the 50^{th} to 99^{th} percentile evenly spaced. This also correlates to the classification we make for app-mix-2 in Table 1 (*medium and steady* load).

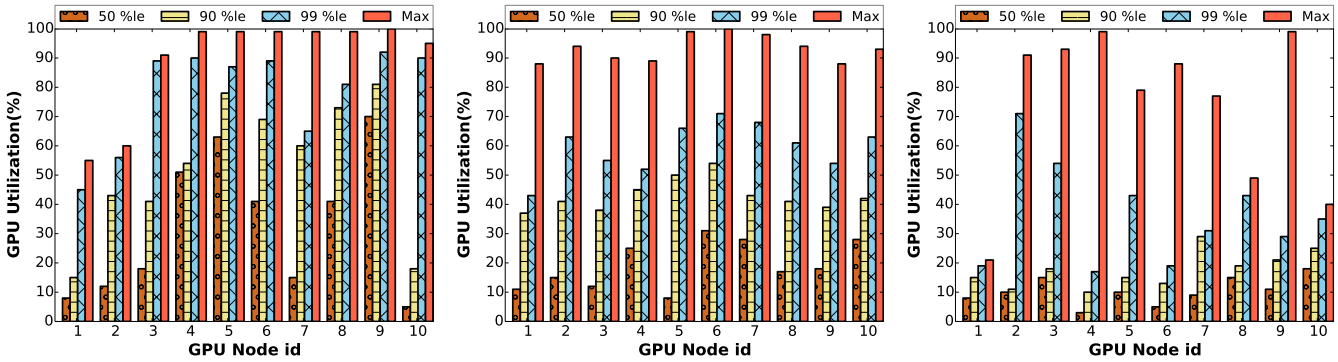
It is evident that, the applications have varying resource needs throughout their lifetime. Also, the applications always overstate their resource requirements by provisioning for peak utilization (also seen in Figure 3). This leads to under-utilization and queuing delays resulting in poor energy efficiency and significant QoS violations. An efficient GPU scheduler would leverage this dynamic workload behavior resulting in improved utilization and energy efficiency.

3.3 Utilization Variance Across GPU Nodes

We look into the variability of GPU node utilization for these three different application mixes through a statistical metric called Coefficient Of Variation (COV). COV is measured by the ratio of standard deviation σ and mean μ. An application-mix with low COV$\leq$1 denotes consistent nature of its load in terms of GPU utilization, making it easier to guarantee the performance of the pods to be scheduled in future and vice-versa. Further, co-locating a pod in a heavy-tailed distribution (COV$\geq$1) case would lead to application interference(noisy-neighbor) and severe capacity violations.

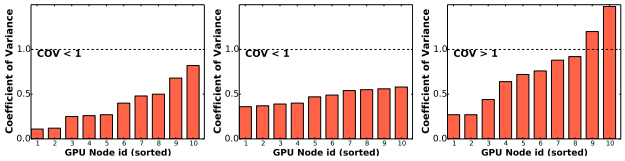
We sort the COV for all GPU nodes within an application mix and show it in Figure 7. We observe that the COV is less than 1 for both app-mix-1 and app-mix-2. Therefore, it is easier to predict and guarantee while scheduling in those scenarios. In case of app-mix-3, the COV is greater than 1, and on top these applications have very low resource consumption. However, if the scheduler is agnostic to the real-time utilization of GPUs, it will lead to co-location that results in heavy-tail distribution (e.g, time quantum 410-690 in Figure 3) resulting in capacity failures. Hence it is important for a GPU orchestrator to provision for an application while considering the real-time GPU utilization metrics.

[4]The abbreviations for application names can be found in [26]



(a) Application-Mix-1 (b) Application-Mix-2 (c) Application-Mix-3

Figure 6: 50/90/99th percentile & maximum GPU utilization for different application mixes scheduled using GPU-agnostic scheduler.



(a) App-Mix-1 (b) App-Mix-2 (c) App-Mix-3

Figure 7: Coefficient of Variance across three Application-Mix.

4 Kube-Knots Design and Integration

In this section, we discuss the *Knots* framework which serves as a real-time distributed data collection layer that aggregates relevant GPU metrics at the headnode. The data collected by Knots is critical to perform GPU-aware scheduling.

Motivated by our key observations from section 3, we design three GPU-aware scheduling policies that leverage the data collected by *Knots*.

4.1 Knots Design

Knots is designed to collect and log the real-time GPU utilization metrics through pyNVML [12]. pyNVML is a python-based API library that exposes utilization metrics to the node-level aggregator. As shown in Figure 5, we log the following five GPU metrics in real-time: (i) streaming multiprocessor (SM) utilization, (ii) memory utilization, (iii) power consumption, (iv) transfer bandwidth and, (v) receive bandwidth. These metrics are pushed to a node-level time-series database IDB (Influx-DB) [7]. *Knots* from the head node can query the GPU nodes for utilization data via the utilization aggregator which is further explained in detail in Section 4.

Based on the desired prediction accuracy, the quality of scheduling is determined by the frequency at which *Knots* queries the IDB. We also cache the recent utilization footprint of all the GPUs at the headnode mitigating communication overheads in case of idle GPU states. From section 2.3.2, we know that the pods could be dynamically resized by predicting the memory consumption using the metrics like I/O bandwidth and SM Utilization, which *Knots* leverages for compaction of pods.

4.2 Resource agnostic sharing

We enable scheduling of multiple containers (pods) on a single GPU by modifying the Nvidia's k8s-device-plugin [11] for Kubernetes. The GPU-compute is time-shared while the memory is space-shared. We used first fit decreasing order bin-packing algorithm to pack the pods on the GPU which greatly improves the average GPU utilization and job turnaround times. The three application mixes shown in Table 1 are scheduled using baseline GPU-agnostic (Res-Ag) scheduler.

Figure 6 shows the utilization of each of the ten GPU nodes. Res-Ag scheduler is actually better in cases where the utilization is already high (for instance application-mix-1 from Figure 6a). *Kubernetes* depends on application frameworks (for instance Spark [13], Tensorflow) to manage its resources internally. From *Observation 5*, we know that it is ideal to expose the application framework's parameters to the cluster scheduler to avoid any internal resource fragmentation as seen in Figure 4 (TF - the top line in red consuming 98% of the memory). Instead, these idle resources could be allocated to the pending pods by sharing the resource. We configured the Tensorflow (TF)'s GPU configuration parameters in such a way that the framework does not subscribe to the entire GPU memory by allowing memory growth for each task. This enables us to pack the GPUs with other pending pods along with the batch applications.

GPU-sharing satisfies the **Observations 1 & 3** which is maximizing the utilization by enabling resource sharing and keeping the utilization high for energy efficiency respectively. However, it fails to consider the actual GPU metrics such as free memory and queue lengths, while scheduling. This leads to severe QoS and capacity violations (discussed in **Observation 2**). When GPU sharing is enabled, it is important that the cluster-wide resource scheduler should take the real-time GPU utilization to safely schedule the containers (pods). At

production scale it becomes an overhead to guarantee the pod safety which led to our next scheme.

4.3 Correlation Based Provisioning

Following the observations made from Alibaba cluster trace, in order to make effective scheduling decisions, the correlation between the utilization metrics must be used for proactive application placement, especially when GPU sharing is enabled. Since *Knots* leverages the utilization aggregator to collect the real-time utilization statistics from each GPU nodes, the scheduler can leverage this information to predict whether the pods can be co-located. For example, two pods positively correlating for using the GPU memory is scheduled to different GPU nodes as the pods have a high probability of memory capacity violations when co-located.

Further, such capacity violations can also lead to container crashing and relaunching. Although the relaunch latency is typically in the order of few seconds, the tasks when relaunched cannot be prioritized over tasks of other pods that are already ahead on the queue. In this case, this particular heavy tailed task affects the overall job completion time.

As observed in Figure 3, all the representative GPU batch workloads have significantly low peak resource usage for most of their execution. For example, SM utilization has a 90x difference between its median and peak. Similarly, bandwidth utilization differs by 400x between the median and peak. The application uses the whole allocated capacity only for 6% of the total execution time, but it is always provisioned for the peak utilization case. This leads to resource fragmentation and underutilization. However, this issue could be fixed by pod resizing. For example, if two uncorrelated applications are containerized as pods and placed individually, they have an X% probability of failure. Whereas if they are co-located, they have a probability of failing together, which is $1-(1-X)^2$. Hence, provisioning based on an average usage and packing uncorrelated applications together can lead to potential savings over Res-Ag scheduling. We call this scheme correlation based provisioning (CBP) which resizes the uncorrelated pods for efficient packing on GPUs. Therefore, the correlation based prediction (CBP) scheduler avoids co-locating two positively correlating pods for any metric such as memory, SM load and bandwidth.

CBP scheduler takes $\boldsymbol{O}(N^2 d)$ to find the optimal placement where N is the number of pending pods and $\boldsymbol{d}$ being the time-series window size which is a crucial configuration parameter that determines the scheduling accuracy. CBP scheduler bin packs the uncorrelated applications together by resizing their respective pods for a common case (80^{th} percentile consumption) than for the maximum case. We provision for 80^{th} percentile based on the first principle observed from Figure 2b, since the maximum memory utilization for almost all the containers do not exceed more than 80% of the provisioned memory. Further, aggressive provisioning (Viz. 50^{th}, 60^{th}, etc.) leads to constant resizing which affects the docker performance at scale. CBP is based on our observations that the probability of all co-located pods reaching their peak resource consumption at the same time is very low. Thus, CBP is aware of both utilization and the pod's correlation metrics.

We notice that the GPUs are still being underutilized due to static provisioning of CBP. This is due to the interarrival pattern of pods, since there are not enough negatively correlating pods to co-locate which results in a skewed schedule order. This indirectly restricts the potential number of GPU nodes to schedule the pods and results in increased queue waiting times for the pending pods which are positively correlating. The average pod waiting time worsens especially in case of heavily loaded application-mix. Pod affinities would further restrict scheduling options for correlating pods. This leads us to our next scheme that attempts to co-locate two positively correlated pods in the same node by predicting the peak resource demand phases of the scheduled pods.

4.4 Peak Prediction Scheduler

Peak Prediction scheduler is built on top of CBP to further exploit the temporal nature of peak resource consumption within an application. This allows PP to schedule two positively correlating pods to the same GPU as they may not contend for the resource at the same time. This is due to the fact that a single application goes through several phase changes during its course of execution. For instance, a DNN instance query first loads the inference weights from the host which results in peak input bandwidth consumption, this is followed by the next phase of the application that is the compute/memory intensive phase. This trend is also very evident in batch workloads as seen in Figure 3. Thus, the peak bandwidth consumption of an application is an early indicator of subsequent compute and memory peaks. Likewise, within an application, the consecutive peak resource demand trends can be predicted using the auto-correlation function given in Equation 2, r_k being the auto-correlation value where Y_i is the utilization measurement at a given time, $\bar{Y}$ is the moving average of Y, and n is the total number of events record at a time window T.

$$r_k = \frac{\Sigma_{i=1}^{n-k}(Y_i - \bar{Y})(Y_{i+k} - \bar{Y})}{\Sigma_{i=1}^{n}(Y_i - \bar{Y})^2} \tag{2}$$

The utilization metric can be further forecasted to anticipate the future GPU utilization using non-seasonal Auto-Regressive Integrated Moving Average(ARIMA). The interval between two consecutive peak resource consumption of a particular resource could be determined by the auto-correlation factor. If the auto-correlation value of a series (i.e., memory utilization) is zero or negative then it shows that (i) the input time-series data is limited or (ii) the trend is not strong enough to predict a positive correlation. In case of correlation value being greater than 0, we use first-order

ARIMA to forecast the utilization of the GPU for the next one second as given in Equation 3 below which is a moving window based linear regression where $\hat{Y}_{pred}$ is a predicted utilization value from previous utilization timeseries Y_{t-1}, where ϕ is the slope of the line and μ is the intercept.

$$\hat{Y}_{pred} = \mu + \phi_1 Y_{t-1} \tag{3}$$

Algorithm 1 CBP + Peak Prediction Scheduler

```
for gpu_node in ∀all_gpu_nodes
    if isActive(QUERY(gpu_node)) then
        All_Active_GPUs.append(gpu_node)
    end if
end for
Node_List ← Sort_by_Free_Memory(All_Active_GPUs)
Pending_Apps ← Sort_Apps_by_Memory_Size(Apps)
Resized_apps ← Docker_Resize(Node_List,Pend_Apps)
Selected_Node ← Head(Node_list)
for App in Pend_Apps
    Schedule(app,Selected_Node)
end for
procedure QUERY(GPU_node)
    node_stats_mem ← query_db(gpu_node.memory)
    node_stats_sm ← query_db(gpu_node.sm)
    node_stats_tx ← query_db(gpu_node.tx_band)
    node_stats_rx ← query_db(gpu_node.rx_band)
    node_stats ← node_stats_mem + node_stats_sm +
node_stats_bw
    return node_stats
procedure SCHEDULE(App, Selected_Node)
    if Can_Co-locate(Covariance,App,Selected_Node)
then
        Ship_Container(App,Selected_Node)
    else

        if AutoCorrelation(node.memory) then ▷ Eqn 2
            pred_free_mem ← ARIMA(node.mem)
            if pred_mem ≥ App.memory then
                Schedule(app)
            else
                Schedule(App,Head→Next(Node_list))
            end if
        end if
    end if
```

Note that, the correlation between consecutive peak resource consumption is a better indicator than the correlation across complete runtime of the application. Also, forecasting the GPU utilization enables the system scheduler to predict the performance better and guarantee the QoS of the application which would be scheduled to that particular GPU node. The peak prediction scheme identifies the set of pods that attains peak consumption of a resource at the same time and schedules them to different GPU nodes.

Algorithm 1 captures the workflow of the scheduler, where for all the active GPU nodes in the datacenter excluding the GPUs which are in deep sleep power state, the utilization for the past five seconds is queried from the respective influxdb databases. This timeseries interval window determines the prediction accuracy. The utilization aggregator in the headnode sorts the nodes by free memory available for the most recent timestamp. The schedule order of pending pods is sorted based first fit decreasing order of pod's requested memory and resized for 80^{th} percentile memory consumption. The `Select_Node` function further analyzes the correlation in case of CBP and schedules to the `Selected_Node` in case of the correlation value is less than 0.5. In case of PP, the auto-correlation function is used on the utilization time-series data for a particular metric(memory_used) of the selected node to look for a trend which predicts a possibility of an impending peak resource consumption to occur through ARIMA, in that case schedule to the next available node in the list by repeating the same admission checks. Once the node is selected, the pod is shipped to the node using *Kubernetes*'s python-based client API calls.

5 Evaluation Methodology

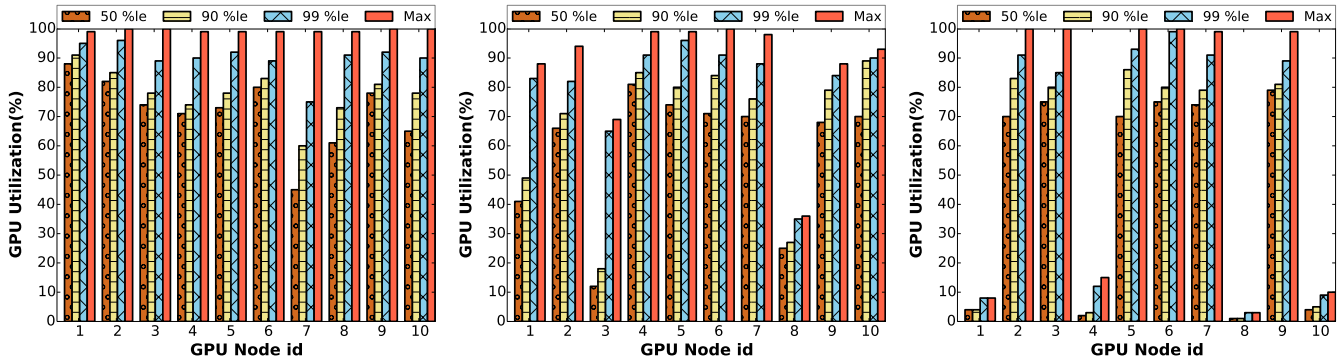
5.1 Hardware

We use Dell PowerEdge R730 eleven node cluster where ten worker nodes have P100 GPUs with Intel Xeon CPU host along with a CPU-only head-node. The details of the single node hardware configuration are listed in the Table 2. We use *Kubernetes* as the resource orchestrator and its default uniform scheduler as our baseline.

- **Worker node -** At every GPU worker node, there is a node-level resource monitor that uses python based API library, pyNVML [12] to query the GPU for every heartbeat interval. The query returns all the five utilization metrics namely, SM, memory, power, transfer and receive bandwidth. These metrics are logged as time-series data into the Influxdb database in their respective nodes.
- **Head node -** The configuration is similar to that in Table 2, with an exception that it does not have a GPU. Next, the utilization aggregator on the head node queries the real-time GPU utilization from the Influxdb. The frequency of querying interval is set to 1ms (justified in Section 6.4). The frequency of data logging (heartbeat) can be varied at the discretion of the utilization aggregator as seen in Figure 5.

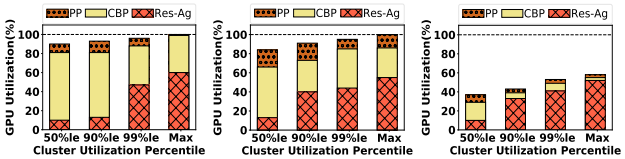
5.2 Container Configuration

Both the batch and user-facing applications are containerized as pods in *Kubernetes*, and their image is made available in all the GPU nodes beforehand, to avoid any startup latencies. The software configuration is given in Table 3 Since these containers are multi-GB the start-up latency depends entirely



(a) Application-Mix-1 (b) Application-Mix-2 (c) Application-Mix-3

Figure 8: 50/90/99th percentile & maximum utilization of GPUs for App-Mixes scheduled using Peak Prediction Scheme.



(a) App-Mix-1 (b) App-Mix-2 (c) App-Mix-3

Figure 9: Cluster-wide GPU utilization improvements.

Hardware	Configuration
CPU	Xeon E5-2670
Cores	12x2(threads)
Clock	2.3 Ghz
DRAM	192 GB
GPU	P100(16GB)

Table 2: Hardware config

Software	Version
Kubernetes	1.9.3
NvidiaDocker	2.0
pyNVML	7.352.0
InFluxDB	1.4.2
CUDA	8.0.61
Tensorflow	1.8

Table 3: Software config

on the network bandwidth to download the container from a docker-hub or launch from the head-node. To avoid unprecedented network delays while shipping these containers, we made sure all the nodes had access to the docker image of the respective workloads in advance. This way, a scheduler can potentially launch the container on any node without cold starts. Typically, the image will be downloaded for the first time and then will continue to persist until the container is killed. PP scheduler uses the Nvidia-docker commands to resize the GPU containers dynamically.

Recall from Section 3 that, for batch applications we used Rodinia [19], HPC and scientific workload suite. Similarly, for latency critical workloads, we used Djinn & Tonic workload suite [26] which has several DNN inference queries using Tensorflow(TF) on GPUs through Keras libraries for execution. We configured TF's GPU configuration knobs to allow incremental memory growth for the DNN inference queries. We capture the inter-arrival pattern of the Alibaba trace which is representative of the datacenter workload. We incorporate this inter-arrival pattern in our launch sequence of jobs. The job sequence comprises of a bin of application-mixes (refer Table 1).

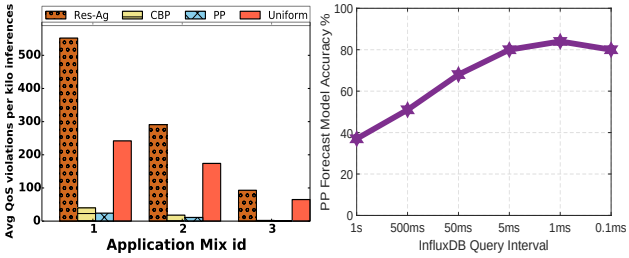
6 Results and Analysis

In this section, we evaluate the different GPU-based scheduling schemes for four major criteria: (i) Cluster-wide utilization, (ii) QoS violations, (iii) Power, and (iv) Accuracy of prediction.

6.1 Cluster-wide GPU Utilization

Building upon the GPU-agnostic scheduling (shown in Figure 6), we propose two scheduling schemes: (i) CBP and (ii) PP. In Figure 8, we plot the utilization improvements of each of the ten GPU nodes for all the three app-mixes using the PP scheduler. The PP scheduler improves the utilization in all the three app-mixes by an average of 62%. Especially in app-mix 2 and app-mix 3, where the utilization is medium and low respectively, the PP scheme does an effective consolidation of workloads using a minimal number of GPUs as possible. As seen in Figure 8c, GPU nodes 1,4,8 and 10 are minimally used as the scheduler effectively consolidated all the low demand workloads to a minimum number of active GPUs as possible. This consolidation shows that, PP leverages the real-time utilization along with utilization forecasting before making any scheduling decisions.

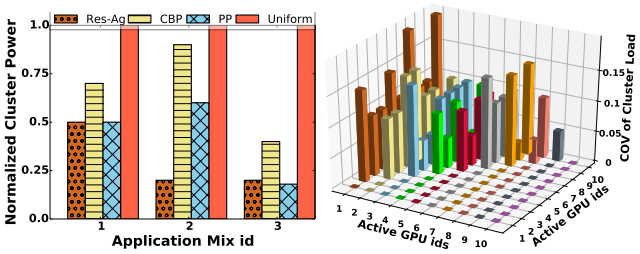
Figure 9 plots the overall resource utilization improvements for all three application mixes for median and tail percentiles. It can be noted that our PP scheme consistently improved the overall GPU utilization in all low, medium and high application-mix cases. For example, PP in app-mix-1 improves by 80% for both median and tail percentiles when compared with Res-Ag scheduler. The improvements are also consistent in the other two app mixes where the improvement over Res-Ag scheduler for median and peak percentiles are 60% and 45% respectively. These benefits reaffirm our prior observation that, it is better to predict the peak resource consumption of the workloads.



(a) Average QoS violations per kilo inferences (queries).

(b) Accuracy of PP across varying time intervals in ms.

Figure 10: QoS guarantees and Accuracy of PP Scheduler.



(a) Normalized power comparisons across four schedulers.

(b) COV of SM-Load using CBP+PP for App-Mix-1.

Figure 11: Cluster-wide Power savings and Load balancing.

6.2 QoS of latency critical workloads

QoS violation threshold for latency-sensitive applications is typically set around 150 milliseconds [22]. When end-to-end latency of such queries exceeds 150ms, they have to be relaunched. We show in Figure 10a that the baseline scheduler violated QoS by 18% on average even though the applications are not co-located together. This is because of queuing delays, where the query is sent to a busy GPU node.

The Res-Ag performs worse than the baseline and violates the threshold from 10% (app-mix-3) to 53% (app-mix-1) of the requests. On the contrary, CBP and PP have almost no violations in all three app-mix (<1%). This is because these schemes were aware of real-time GPU utilization and provisioned the containers for 90 percentile utilization while guaranteeing the QoS via utilization forecasting. Note that, we also co-locate the latency sensitive queries along with the batch jobs without affecting their QoS. PP achieves this by predicting the inter and intra-application correlation for utilization and, at the same time by leveraging Knots real-time data, it can forecast the future node utilization. This avoids QoS violations while keeping the utilization high.

Since *Knots* exposes the GPU utilization through Nvidia's resource monitoring APIs, the schedulers built on top of *Kube-Knots* inherently take advantage of these metrics. Figure 11b plots the pair-wise COV of GPU cluster loads. The lower diagonal values are omitted for the sake of clarity. It can be observed that the COV of loads across different GPU nodes ranges within 0 to 0.2 when compared to the COV in Figure 7a which ranges from 0.1 to 0.7. Hence, PP performs load balancing in high-load scenarios such as app-mix-1 along with consolidation in low-loads such as app-mix-3.

6.3 Power efficiency

Figure 11a plots the overall power consumed by the different schedulers, where Res-Ag consumes the least power on an average (33%) while PP scheme consumes the second least power (43%). Note that, while Res-Ag optimizes for power it also violates QoS for almost 53% of the requests as discussed in section 6.2. On the other hand, the PP scheduler only consumes 10% more power than Res-Ag while guaranteeing minimal QoS violations due to that fact that PP scheduler attempts to pack only active GPUs but it ensures that two pods do not peak for resource consumption at the same time. The CBP policy consumes more power than both Res-Ag and PP schemes by 25% and 15% respectively, although being similar to PP scheme in terms of QoS violations. This emphasizes the importance of predicting the peak resource consumption of pods on top of the correlation metrics. Further, we understand the impact on of PP scheduler's prediction accuracy by varying the frequency of the querying interval.

6.4 Accuracy of the Peak Predictor

In case of PP scheduler, ***d*** is the frequency at which the utilization aggregator at head-node is querying the worker nodes for GPU utilization that forms our time-series data. We vary the frequency at which we query the GPUs to predict the peak resource usage. As seen in Figure 10b, PP improves its accuracy from 40% to 84% when the sampling interval for querying utilization is varied from 1000ms to 1 ms, beyond which the accuracy saturates. Hence for the maximum prediction accuracy, we set the utilization aggregator to query the GPU nodes every 1ms. The overhead for querying the GPU at every 1ms interval does not affect the pod's performance.

6.5 Discussion

An ideal scheduler would keep all 50/90/99th percentile and max utilization high for all the active GPUs, while letting the rest of the GPUs to be in a deep sleep power state. As a consequence, it yields the best performance per watt spent. However, in reality, this scenario is impractical because of the diurnal inter-arrival times of jobs. Therefore, the near-optimal solution would be, in every epoch the scheduler has to ensure performance aware bin-packing while, minimizing the GPU resource fragmentation. Our proposed scheduling schemes can potentially be extended to any non-preemptible accelerators such as FPGAs, TPUs to improve the overall utilization while guaranteeing the QoS of the tasks.

7 Related Work

GPU resource sharing and over-commitment in datacenters is an emerging research problem. We have done a comprehensive analysis of relevant scheduling frameworks, shown in Table 4. Prior works range from node-level techniques such as virtualization to application-based GPU-cluster scheduling. we broadly classified the related work into three categories.

Utilization-aware CPU schedulers: There are several works that focus on guaranteeing QoS and CPU utilization without sacrificing the SLA of applications. Bubble-up [32] and Bubble-flux [45] perform safe co-location to avoid interference while improving chip multiprocessor utilization. However, these techniques cannot be directly extended to GPUs as the global memory and the PCIe bandwidth is different from that of a CPU.

GPU-aware runtime systems: Researchers have proposed runtime changes to the GPU to enable better scheduling of the GPU tasks either by predicting task behavior or reordering queued tasks. Ukidave et al. [42] optimized for workloads which under-utilize the device memory and bandwidth. Chen et al. [21] proposed Baymax, a runtime mechanism which does workload batching and kernel reordering to improve the GPU utilization. Other techniques such as interference driven resource management proposed by Phull et al. [36], predicts the interference on a GPU to do safe co-location of GPU tasks. They do not consider memory bandwidth contention nor over-commitment challenges as they assume sequential execution of the GPU tasks. Most of these approaches aim to increase utilization of a individual GPU node and do not scale at the cluster level as they depend on offline training models for node-level run time prediction [21, 42].

Node-level GPU-aware scheduling: Recent works [24, 30] have proposed docker-level container sharing solutions. In this approach, multiple containers can be made to fit in the same GPU as long as the active working set size of all the containers are within the GPU physical memory capacity. These scheduling techniques over-commit resources for individual containers to ensure crash free execution. This may lead to internal memory fragmentation as we show in Figure 4. We address this by predicting the resource usage and dynamically resizing the containers based on the future resource consumption projections.

Distributed GPU scheduling for DNN applications: Distributed DNN training based applications have started taking advantage of multiple GPUs in a cluster. There are emerging schedulers such as Gandiva [44], Optimus [35], and several other works [15, 40, 46] that focus on prioritizing the GPU tasks that are critical for the DNN model accuracy in case of parameter server-based architecture. These schedulers are solely designed to cut down on the DNN model exploration time and do not scale to other application domains. KubeKnots is designed to cater multi-faceted GPU applications and does not require any application-domain knowledge or apriori profiling data.

Features	Baymax [21]	Gandiva [44]	Optimus [35]	Mystic [42]	Bubble-Flux [45]	KubeKnots
Cluster Level integration	✗	✗	✗	✗	✗	✓
Workload consolidation	✓	✓	✓	✓	✓	✓
Needs apriori profiling	✓	✓	✓	✓	✓	✗
Application memory resizing	✗	✗	✗	✗	✗	✓
Accelerator power aware	✗	✗	✗	✗	✗	✓
Dynamic Orchestration	✗	✗	✗	✗	✓	✓
Application memory resizing	✗	✗	✗	✗	✗	✓
Resource peak prediction	✗	✗	✗	✗	✗	✓

Table 4: GPU-based scheduling features comparison.

Hardware support for virtualizing GPUs: There has been extensive work on providing hardware support for GPU virtualization [25, 27] and preemption [34, 39]. Gupta et al. [25] implemented a task queue in the hypervisor to allow virtualization and preemption of GPU tasks. Tanasic et al. [41] proposed a technique that improves the performance of high priority processes by enabling preemptive scheduling on GPUs. Aguilera et al. [14] proposed a technique to guarantee QoS of high priority tasks by spatially allocating them on more SMs in a GPU. All these techniques require vendors to add extra hardware extensions. Our proposed scheduler does not need any hardware level changes and are readily deployable in Commodity Off-The-Shelf datacenters.

8 Conclusion

In this paper, we observe that the existing resource orchestrators like *Kubernetes* overlook the importance of dynamic orchestration of GPUs. Hence, we identify several challenges in dynamic resource orchestration for GPU-based datacenters. Motivated by our observations, we propose *Knots*, a GPU aware orchestration layer that aids in aggregating real-time GPU cluster utilization. *Knots* along with *Kubernetes* performs GPU utilization aware orchestration. We further evaluate three GPU-based container scheduling techniques on a ten node GPU-cluster, which leverages *Kube-Knots* for orchestration.

Our proposed Peak Prediction (PP) scheduler when compared against the GPU-agnostic scheduler, improve the cluster-wide GPU utilization by up to 80% for both average and 99^{th} percentile, through QoS aware workload consolidation which leads to 33% savings in power consumption across the cluster. Further, we reduced the overall QoS violations of latency sensitive queries by up to 53% when compared to the resource agnostic scheduling with GPU utilization prediction accuracy as high as at 84%.

References

[1] Auto-Regressive Integrated Moving Average. http://people.duke.edu/~rnau/411arim.htm#arima010.

[2] Alibaba Cluster Trace Data description. https://github.com/alibaba/clusterdata.

[3] Amazon EC2 Elastic GPUs. https://aws.amazon.com/ec2/elastic-gpus/.

[4] Cisco's Global Cloud Index. https://www.cisco.com/c/en/us/solutions/collateral/service-provider/global-cloud-index-gci/white-paper-c11-738085.html.

[5] Docker Swarm description. https://docs.docker.com/engine/swarm/swarm-tutorial/.

[6] GPU-Based Deep Learning Inference. https://www.nvidia.com/content/tegra/embedded-systems/pdf/jetson_tx1_whitepaper.pdf.

[7] InfluxDB time series. https://github.com/influxdata/influxdb.

[8] Mesos Scheduler GPU support. http://mesos.apache.org/documentation/latest/gpu-support/.

[9] Mesosphere description. https://mesosphere.com/738085.html.

[10] Microsoft VM with GPU. https://docs.microsoft.com/en-us/azure/virtual-machines/windows/sizes-gpu.

[11] Nvidia k8s-device-plugin. https://github.com/NVIDIA/k8s-device-plugin.

[12] Python Bindings for the NVIDIA Management Library. https://pypi.python.org/pypi/nvidia-ml-py/4.304.04.

[13] Spark resource manager. https://spark.apache.org/docs/latest/cluster-overview.html.

[14] P. Aguilera, K. Morrow, and N. S. Kim. Qos-aware dynamic resource allocation for spatial-multitasking gpus. In *2014 19th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 726–731, Jan 2014.

[15] Marcelo Amaral, Jordà Polo, David Carrera, Seetharami Seelam, and Malgorzata Steinder. Topology-aware gpu scheduling for learning workloads in cloud environments. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 17. ACM, 2017.

[16] Tanya Amert, Nathan Otterness, Ming Yang, James H Anderson, and F Donelson Smith. Gpu scheduling on the nvidia tx2: Hidden details revealed. In *IEEE Real-Time Systems Symposium (RTSS)*, 2017.

[17] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, omega, and kubernetes. *Commun. ACM*, 59(5):50–57, April 2016.

[18] Adrian M Caulfield, Eric S Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, et al. A cloud-scale acceleration architecture. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pages 1–13. IEEE, 2016.

[19] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 44–54. Ieee, 2009.

[20] Quan Chen, Hailong Yang, Minyi Guo, Ram Srivatsa Kannan, Jason Mars, and Lingjia Tang. Prophet: Precise qos prediction on non-preemptive accelerators to improve utilization in warehouse-scale computers. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 17–32. ACM, 2017.

[21] Quan Chen, Hailong Yang, Jason Mars, and Lingjia Tang. Baymax: Qos awareness and increased utilization for non-preemptive accelerators in warehouse scale computers. *ACM SIGARCH Computer Architecture News*, 44(2):681–696, 2016.

[22] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Commun. ACM*, 56(2):74–80, February 2013.

[23] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 323–336, Berkeley, CA, USA, 2011. USENIX Association.

[24] James Gleeson and Eyal De Lara. Heterogeneous gpu reallocation. In *9th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 17)*, 2017.

[25] Vishakha Gupta, Karsten Schwan, Niraj Tolia, Vanish Talwar, and Parthasarathy Ranganathan. Pegasus: Coordinated scheduling for virtualized accelerator-based systems. In *Proceedings of the 2011 USENIX Conference*

on USENIX Annual Technical Conference, USENIX-ATC'11, pages 3–3, Berkeley, CA, USA, 2011. USENIX Association.

[26] Johann Hauswald, Yiping Kang, Michael A Laurenzano, Quan Chen, Cheng Li, Trevor Mudge, Ronald G Dreslinski, Jason Mars, and Lingjia Tang. Djinn and tonic: Dnn as a service and its implications for future warehouse scale computers. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 27–40. ACM, 2015.

[27] Cheol-Ho Hong, Ivor Spence, and Dimitrios S Nikolopoulos. Gpu virtualization and scheduling methods: A comprehensive survey. *ACM Computing Surveys (CSUR)*, 50(3):35, 2017.

[28] Myeongjae Jeon, Shivaram Venkataraman, Junjie Qian, Amar Phanishayee, Wencong Xiao, and Fan Yang. Multi-tenant gpu clusters for deep learning workloads: Analysis and implications. Technical report, MSR-TR-2018, 2018.

[29] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 1–12. ACM, 2017.

[30] Daeyoun Kang, Tae Joon Jun, Dohyeun Kim, Jaewook Kim, and Daeyoung Kim. Convgpu: Gpu management middleware in container based virtualized environment. In *Cluster Computing (CLUSTER), 2017 IEEE International Conference on*, pages 301–309. IEEE, 2017.

[31] Shinpei Kato, Karthik Lakshmanan, Raj Rajkumar, and Yutaka Ishikawa. Timegraph: Gpu scheduling for real-time multi-tasking environments. In *Proc. USENIX ATC*, pages 17–30, 2011.

[32] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, pages 248–259, New York, NY, USA, 2011. ACM.

[33] Jason Jong Kyu Park, Yongjun Park, and Scott Mahlke. Chimera: Collaborative preemption for multitasking on a shared gpu. *ACM SIGARCH Computer Architecture News*, 43(1):593–606, 2015.

[34] Jason Jong Kyu Park, Yongjun Park, and Scott Mahlke. Chimera: Collaborative preemption for multitasking on a shared gpu. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 593–606, New York, NY, USA, 2015. ACM.

[35] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. Optimus: an efficient dynamic resource scheduler for deep learning clusters. In *Proceedings of the Thirteenth EuroSys Conference*, page 3. ACM, 2018.

[36] Rajat Phull, Cheng-Hong Li, Kunal Rao, Hari Cadambi, and Srimat Chakradhar. Interference-driven resource management for gpu-based heterogeneous clusters. In *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '12, pages 109–120, New York, NY, USA, 2012. ACM.

[37] Wei Qiao, Ying Li, and Zhong-Hai Wu. Dltap: A network-efficient scheduling method for distributed deep learning workload in containerized cluster environment. In *ITM Web of Conferences*, volume 12, page 03030. EDP Sciences, 2017.

[38] Charles Reiss, Alexey Tumanov, Gregory R Ganger, Randy H Katz, and Michael A Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of the Third ACM Symposium on Cloud Computing*, page 7. ACM, 2012.

[39] Kittisak Sajjapongse, Xiang Wang, and Michela Becchi. A preemption-based runtime to efficiently schedule multi-process applications on heterogeneous clusters with gpus. In *Proceedings of the 22Nd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '13, pages 179–190, New York, NY, USA, 2013. ACM.

[40] Seetharami R Seelam and Yubo Li. Orchestrating deep learning workloads on distributed infrastructure. In *Proceedings of the 1st Workshop on Distributed Infrastructures for Deep Learning*, pages 9–10. ACM, 2017.

[41] Ivan Tanasic, Isaac Gelado, Javier Cabezas, Alex Ramirez, Nacho Navarro, and Mateo Valero. Enabling preemptive multiprogramming on gpus. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 193–204, Piscataway, NJ, USA, 2014. IEEE Press.

[42] Yash Ukidave, Xiangyu Li, and David Kaeli. Mystic: Predictive scheduling for gpu based cloud servers using machine learning. In *Parallel and Distributed Processing Symposium, 2016 IEEE International*, pages 353–362. IEEE, 2016.

[43] Daniel Wong. Peak efficiency aware scheduling for highly energy proportional servers. In *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*, pages 481–492. IEEE, 2016.

[44] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, et al. Gandiva: introspective cluster scheduling for deep learning. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 595–610, 2018.

[45] Hailong Yang, Alex Breslow, Jason Mars, and Lingjia Tang. Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 607–618, New York, NY, USA, 2013. ACM.

[46] Hao Zhang, Zeyu Zheng, Shizhen Xu, Wei Dai, Qirong Ho, Xiaodan Liang, Zhiting Hu, Jinliang Wei, Pengtao Xie, and Eric P Xing. Poseidon: An efficient communication architecture for distributed deep learning on gpu clusters. *arXiv preprint*, 2017.