Dynamic GPU Scheduling With Multi-Resource Awareness and Live Migration Support

Xiaoyang Wang, Yongkun Li, Fan Guo, Yinlong Xu, and John C. S. Lui, *Fellow, IEEE*

***Abstract*—In clouds and data centers, GPU servers with multiple GPUs are widely deployed. Current state-of-the-art GPU scheduling policies are "static" in assigning applications to different GPUs. These policies usually ignore the dynamics of the GPU utilization and are often inaccurate in estimating resource demand before assigning/running applications, so there is a large opportunity to further balance the loads and improve GPU utilization. Based on CUDA (Compute Unified Device Architecture), we develop a runtime system called DCUDA which supports *"dynamic"* scheduling of running applications between multiple GPUs. In particular, DCUDA takes into consideration multidimensional resources, including computing cores, memory usage, and energy consumption. It first provides a real-time and lightweight method to accurately monitor the resource demand of applications and GPU utilization. Furthermore, it provides a universal migration facility to migrate *"running applications"* between GPUs with negligible overhead. More importantly, DCUDA transparently supports all CUDA applications without changing their source code. Experiments with our prototype system show that DCUDA can reduce 78.3% of overloaded time of GPUs on average. As a result, for different workloads consisting of a wide range of applications we studied, DCUDA can reduce the average execution time of general applications by up to 42.1%, and even up to 67% for memory-intensive tasks. Besides, DCUDA also reduces 13.3% of energy in light-load scenarios.**

***Index Terms*—Dynamic scheduling, GPU scheduling, live migration, load balance.**

I. Introduction

GRAPHICS Processing Units (or GPUs) are a class of computing devices with massive simple cores and high bandwidth memory. They have been widely used in various systems for efficient parallel computing, such as scientific computing, image processing, data mining, machine learning, and so on [13], [28], [40]. In clusters, there are always many GPUs in each node, to support more powerful computing capacity. However, many past studies demonstrated that the computing resources of GPUs are often under-utilized like when running only a single application on each GPU [20], [31]. To improve GPU utilization, GPU sharing is adopted to run multiple applications concurrently on each GPU, and this scenario is quite common in the current data centers and clouds. To better utilize the computing power of all GPUs in a GPU server, current GPU programming models (e.g., CUDA for NVIDIA GPUs [3]) provide a functionality for applications to explicitly select the GPU device on which they wish to run. However, such user-specified assignments may lead to severe load imbalance between GPUs due to the unawareness of the GPU utilization. To further improve the efficiency of GPU sharing, various scheduling methods are proposed to distribute GPU resources between tenants or applications to balance the load between GPUs. Some well-known GPU scheduling methods are Round-robin scheduling [23], Least-Loaded scheduling [16], [34], [35], Prediction-based scheduling [39] and so on. The main goal of these methods is to assign new applications to proper GPUs, e.g., the Least-loaded scheduling always assigns new applications to the GPU which has the least load. We call these methods *static scheduling*, because they only make the GPU-application assignment before running the application, and once an application is assigned to a GPU, it cannot be migrated to another GPU during its execution.

In this work, we show that the efficiency of GPU sharing is still limited with static scheduling methods. The deficiency of static scheduling not only comes from the difficulty of estimating the exact resource demand of applications before running them, but also comes from the variability of GPU utilization as well as the lack of live migration support for running applications. In particular, our experiments show the case that at least one GPU is overloaded while some other GPUs are underloaded accounting for 41.7% of the whole execution time of all applications. The load imbalance problem not only reduces the GPU utilization but also significantly prolongs the execution time of applications. Furthermore, for static scheduling, it is also difficult to detect real memory usage and memory oversubscription, which also seriously affect GPU execution performance. Moreover, the static scheduling methods do not fully explore the potential of GPU sharing, so the static scheduling solution usually increases energy consumption of GPUs.

To improve the efficiency of GPU sharing, it is important to develop a *dynamic scheduling* method by providing accurate monitor of GPUs and applications as well as live migration support for running applications. We emphasize that it is not an easy task to support "lightweight" monitor and "live" migration, and several challenges exist. First, current monitoring tools (e.g., nvprof [6]) collect the trace data of each function call by replaying APIs, and thus introduce a high performance

Manuscript received 14 December 2022; revised 3 March 2023; accepted 30 March 2023. Date of publication 3 April 2023; date of current version 6 September 2023. Recommended for acceptance by J. Zhai. *(Corresponding author: Yongkun Li.)*

Xiaoyang Wang, Yongkun Li, Fan Guo, and Yinlong Xu are with the School of Computer Science and Technology, University of Science and Technology of China, Hefei 230026, China (e-mail: wxy1999@mail.ustc.edu.cn; ykli@ustc.edu.cn; lps56@mail.ustc.edu.cn; ylxu@ustc.edu.cn).

John C. S. Lui is with the Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong (e-mail: cslui@cse.cuhk.edu.hk).

Digital Object Identifier 10.1109/TCC.2023.3264242

penalty [14]. Therefore, it is challenging to accurately monitor the utilization of each GPU and the resource demand of each application without interrupting the running applications so as to avoid the high overhead. Second, existing programming model like CUDA (Compute Unified Device Architecture) does not support live migration, so it is necessary to develop a new live migration facility to support migrating running CUDA applications between GPUs, while the key challenging issue is how to guarantee the same runtime environment and application data with low overhead. Last but not the least, due to the dynamics of GPU utilization, determining when and which applications should be migrated is also challenging, especially with the goal of balancing both cores and memory utilization, and reducing the number of migration times and the amount of migration data.

In this work, we design and implement DCUDA, a runtime system that supports accurately monitoring GPUs' utilization and applications' resource demands with negligible overhead, provides dynamic scheduling of running applications, and transparently supports live migration for all CUDA applications with a little overhead. Our main contributions are

- We propose a *lightweight* computing core monitoring method by intercepting API calls with wrapper libraries and tracking the utilization information from the function parameters. We also propose a novel way to monitor GPU memory utilization, we check the GPU memory usage on the CPU side to obtain accurate results with low overhead. Compared with current monitoring tools like nvprof, our monitor scheme does not interrupt the running applications, and incurs negligible overhead, while also achieving higher than 90% monitoring accuracy.
- We propose a dynamic scheduling mechanism to guarantee load balance between GPUs by migrating running applications from overloaded GPUs to underloaded GPUs. Besides doing load balance, we also propose two optimization techniques to further reduce energy consumption by compacting lightweight applications and improve fairness with a priority-based time-slicing policy. We also develop a memory-concerned scheduling policy to optimize the performance of memory-intensive applications.
- We develop a live migration facility that is compatible with all CUDA applications without requiring applications to modify their source codes. Besides, the time cost of the live migration task is less than 0.3% of the application execution time due to our optimization techniques, such as handle pooling and data prefetching.
- We implement a prototype and conduct experiments to show the efficiency of DCUDA. Results show that DCUDA reduces 78.3% of overloaded time of GPUs on average. As a result, DCUDA reduces the average execution time of general applications by up to 42.1%, even up to 67% for memory-intensive tasks, and reduces the energy consumption of GPUs by up to 13.3%.

The rest of the paper is organized as follows. In Section II, we introduce background and analyze the limitations of the current static scheduling policy, then motivate the design of DCUDA. In Section III, we present the design details of DCUDA. In Section IV, we describe the experimental setup and present the evaluation results. Section V reviews related work and Section VII concludes the paper.

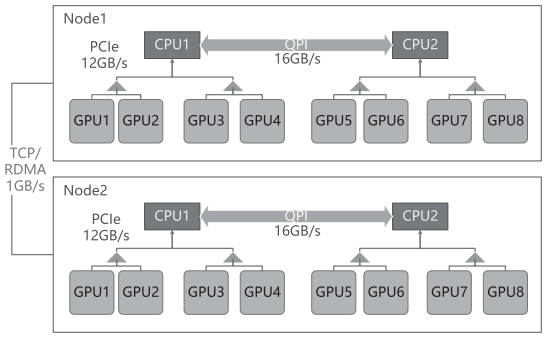


Fig. 1. System architecture in a GPU cluster.

II. Background and Motivation

A. GPU Sharing

GPU sharing is proposed to fully utilize GPU resources. In GPU clusters, there are different ways to share GPU resources. First, we can use a single GPU to run applications concurrently [43]. Second, in a single GPU node, there might be more than one GPU installed, and applications in this node share these GPUs [27], [39]. Finally, clients can manage the whole cluster as a GPU pool, all nodes with multiple GPUs are shared [16], [22]. Fig. 1 illustrates the architecture of a GPU cluster which supports the above three ways of GPU sharing.

In this paper, we focus on the scenario of multiple GPUs on a single node. To share the resources of multiple GPUs more easily, unified memory technology [8] provides a single memory address space to all GPUs and CPUs, with an automatic page migration for data locality. The page migration engine also allows GPU threads to trigger page fault when the accessed data does not reside in GPU memory, and this makes the system efficiently migrates pages from anywhere in the system to the memory of GPUs in an on-demand manner, which enables a GPU to handle larger memory demand than its capacity by memory swap. DCUDA also takes advantage of the unified memory.

B. GPU Scheduling Schemes and Limitations

To improve the fairness and effectiveness in GPU sharing, various GPU scheduling methods are proposed, such as Round-Robin scheduling [23], Least-Loaded scheduling [16], [34], [35], and Prediction-based scheduling [39], [45]. Round-robin scheduling uses a round-robin strategy for choosing GPU to execute a new application. Least-Loaded scheduling assigns the GPU with the lightest load to run new applications. Prediction-based scheduling makes scheduling decisions based on workload pattern predictions and might focus on certain types of applications, like deep learning [29], [44], [45]. We point out that all these methods are static scheduling methods, as they are responsible for only assigning new applications to different GPUs before running, and can not dynamically migrate running applications. Thus, the static scheduling policy still has multiple

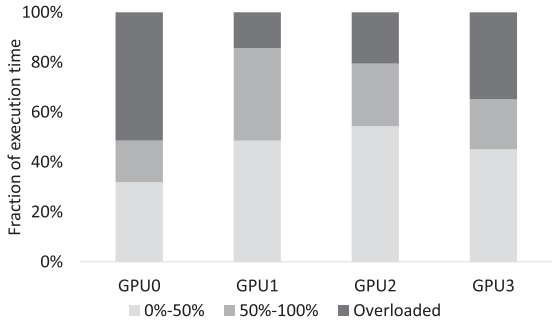


Fig. 2. Load imbalance with Least-Loaded scheduling.

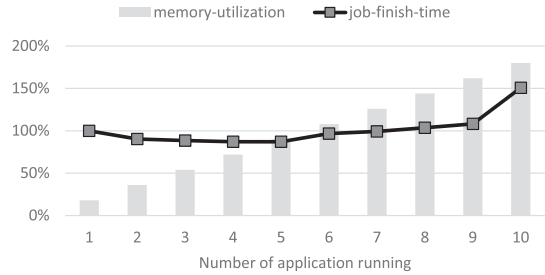


Fig. 3. Normalized Job finish time in a single GPU for running several times of a same application.

limitations on the GPU sharing performance, including core utilization, memory utilization, and energy efficiency.

1) Limitation on Load Imbalance of GPU Cores: We find that load balance could still be greatly improved in GPU sharing with static scheduling. The main reasons are as follows. First, it is hard to obtain the exact resource demand of applications before running them on GPUs, so static scheduling methods could not find out the most appropriate GPU to assign for a newly arrived application. Second, GPU utilization is time-varying due to the dynamic arrivals and departures of applications, but static scheduling methods cannot migrate running applications to re-balance loads of GPUs. Third, static scheduling methods do not distinguish applications with different resource demands, so applications with low resource demands are often blocked by applications with high resource demands. All the above problems may lead to load imbalance between GPUs, which further causes resource contention on overloaded GPUs and energy inefficiency on underloaded GPUs. We emphasize that all these problems may become severe in a multi-tenant GPU-sharing environment, because applications from different tenants usually have different resource demands.

To further validate the load imbalance problem of static scheduling, we conducted experiments to show GPUs' load by deploying the Least-Loaded scheduling. We run a workload consisting of twenty different applications, which arrive with a fixed interval with the length being smaller than the execution time of the application (see Section IV-A for details of the system setup). We use this setup because DCUDA focuses on the scenario of GPU sharing which inherently has multiple applications concurrently running on each GPU.

We classify each GPU into three utilization types, i.e., from 0-50% utilization, 50%-100% utilization, up to "overloaded" which denotes the case in which the total resource demand of all applications exceeds the resource capacity of the GPU. We present the fraction of time of being at each utilization type for each GPU, and the results are shown in Fig. 2. We find that even with the Least-Loaded scheduling method, it is quite common to have GPUs at the overloaded state or underloaded state (i.e., 0-50% utilization). We also find that the case of load imbalance, in which at least one GPU is overloaded and some other GPUs are still underloaded, accounts for 41.7% of the whole execution time. This is because static scheduling can not migrate running applications from the overloaded GPUs to the underloaded GPUs.

2) Limitation on Memory Usage: GPU memory capacity is limited, and it is easy to be over-subscribed when it is shared among multiple tasks. [26] While memory swap is a feasible way to guarantee running applications on memory overloaded GPUs, it has potential performance degradation problems when too many pages are requested and frequently swapped in and swapped out, leading to memory access time even longer than execution time, thus leading to memory thrashing. We emphasize that thrashing may become more severe in a multi-tenant GPU-sharing environment because applications usually have different resource demands and memory access models leading to unpredictable page replacement requirements.

To further validate the thrashing problem in GPU sharing, we conduct experiments to show GPUs memory thrashing by considering the situation when memory is heavy-loaded. As shown in Fig. 3, we run the same application (alexnet) multiple times, and it has 80% core utilization and 16% memory utilization for a single task. We analyze the total job finish time of the applications in both cases of serially and parallelly running the application, respectively. In theory, the parallel execution's job finish time should be never higher than 100% of the serial running time However, we find that after the GPU memory is fully utilized after the memory requirement increases even higher than 150% of the original serial running time. And we also find that when the memory is not over-subscribed, the increase in the job finish time does not happen. The main result of the overhead is caused by memory oversubscribing and thrashing. For the static scheduling policies, the scheduling design considers only the core utilization, and they do not take into consideration the memory usage which may induce the potential performance problem.

3) Limitation on Energy Consumption: Energy consumption is a major cost of data centers, and GPUs are the major electricity consuming devices. To improve the energy efficiency, NVIDIA GPUs support adaptive management of the energy consumption [18]. Precisely, a GPU can be configured to run at multiple levels, and a lower level means lower performance with lower energy consumption. GPUs will adaptively change their power level based on their utilization, e.g., when a GPU becomes idle, it will switch to the lowest level so as to save energy.

To explore the relationship between GPUs' load and their energy consumption, we conduct experiments to evaluate the energy consumption of a GPU by varying the number of applications simultaneously running on it. As shown in Fig. 4, we find that running a single application on a GPU, which we call *single*

Fig. 4. Energy consumption when run a single application or concurrently run two applications on a GPU.

execution, usually increases a lot of energy consumption (note that the GPU power in idle state is around 15W), but running two applications concurrently, which we call *concurrent execution*, only increases the energy consumption a little compared with a single execution. The main reason is that running a single application needs to wake up the GPU from the idle state and increase its clocks, and this consumes a lot of energy. Note that the applications we tested here are very lightweight, and running them concurrently on one GPU only causes a small slowdown in their performance ($< 2\%$). Thus, compacting multiple lightweight applications to run on fewer GPUs in DCUDA can improve the overall energy efficiency. We add this part's management to DCUDA's scheduling model.

C. Challenges of Dynamic Scheduling

To further improve load balance and alleviate resource contention between GPUs, we developed DCUDA, a runtime system that supports the dynamic scheduling of running applications. The main challenges of DCUDA are as follows.

Selection of Key Factors. The GPU sharing performance can be influenced by multi-dimensions, like GPU cores, GPU memory, and GPU transport(I/O), so DCUDA should choose suitable metrics to balance the complexity and performance in monitoring and scheduling.

Monitoring GPUs and Applications. DCUDA needs to monitor both GPUs and applications in real-time, and this task can not be accomplished by current monitor tools, such as nvidia − smi [4] and nvprof [6]. Specifically, nvidia − smi can only monitor GPUs but not applications, while nvprof imposes a large overhead as it collects trace data of each function call by replaying APIs. Thus, how to accurately monitor applications and GPUs with low overhead remains challenging.

Live Migration. CUDA and existing studies do not provide a universal live migration function, and we face three challenges in the design and implementation of the live migration facility. The first challenge is to migrate memory data to the target GPU while keeping the virtual addresses of the data the same as that in the source GPU. The second one is how to construct a consistent runtime environment and how to resume the computing tasks of applications correctly on the target GPU. The third and most important challenge is that from the perspective of performance, all the above tasks may introduce a large overhead, so only a lightweight live migration scheme is practical.

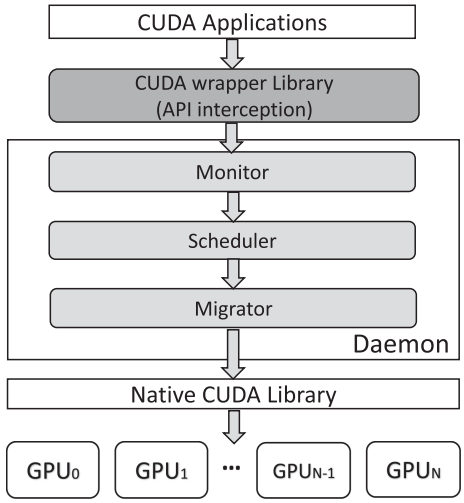


Fig. 5. DCUDA architecture.

Dynamic Scheduling. Dynamic scheduling is more efficient to achieve higher load balance and better GPU utilization. However, three problems need to be addressed. First, we need to be aware of the ping-pong effect to avoid flip-flop migrations. Second, we need to carefully choose candidate applications for migration to reduce the number of migrations. Finally, we need to balance the performance of all applications and avoid lightweight applications being blocked by heavyweight ones during live migration.

III. Design of DCUDA

DCUDA is a CUDA-based GPU-sharing platform that supports dynamic scheduling of running applications between GPUs. In this section, we first introduce the overall design of DCUDA, then present the design details of its key components.

A. Overview of DCUDA

Similar to many other GPU sharing platforms, like DGSF [19], AvA [46], vCUDA [36], and rCUDA [33] , DCUDA adopts a frontend-backend architecture to ease the implementation while providing full compatibility to all CUDA applications. As shown in Fig. 5, the frontend of DCUDA is implemented as a CUDA wrapper library, which dynamically links user applications and intercepts CUDA API calls. The backend is realized as a daemon responsible for receiving GPU requests from the frontend, dispatching CUDA API calls to the corresponding GPUs, and returns error codes and/or output parameters to the frontend. In particular, DCUDA consists of three modules in the backend to realize its key features: *Monitor*, *Scheduler*, and *Migrator*. The Monitor tracks the real-time utilization of GPUs and the resource demand of each application from intercepted API calls. The Scheduler dynamically schedules running applications by taking advantage of the monitored information to balance the load between multiple GPUs. Finally, the Migrator is responsible for migrating CUDA applications between different

GPUs, including cloning runtime, replicating memory data, and computing tasks.

B. Key Factors

To efficiently schedule jobs between GPUs, there are multiple resource requirement factors to be concerned, for example, Liquid [21] uses a resource requirement vector including GPU computing power, GPU memory, and network bandwidth and proposes a network-efficient scheduling solution to improve the execution performance of DL jobs. So before introducing the design of DCUDA, we first discuss about the performance metrics that must be considered in the monitor and scheduling, and we conclude that it is important to simultaneously pay attention to multiple resources of GPUs in DCUDA design, e.g., the computing cores, GPU memory, and energy consumption.

The first key factor is the GPU computing core. In Section II-B1, we have illustrated the GPU load imbalance through core usage, showing that computing core is the central metric for GPU sharing, so it is the key factor being considered in DCUDA scheduling policy design. That is, DCUDA takes into consideration the core usage, and takes it as the most important scheduling metric.

The second factor is GPU memory. For many GPU tasks, especially for deep learning applications, GPU memory requirement in clusters significantly increases under the growth of the computing model and the input data size. So we have to also pay attention to the memory utilization of GPU sharing, by developing memory concerned design and implementation in DCUDA, we can alleviate the memory thrashing effect showing in Section II-B2 and further improve the stability of GPU sharing system.

The third factor is energy consumption. In Section II-B3, we have illustrated the potential energy-efficient optimization on GPU memory scheduling, which is both economical and challenging. DCUDA also considers this challenge, to provide not only high performance but also energy-efficient scheduling design.

We now discuss the NUMA (Non-Uniform Memory Access) effect when scaling up the number of GPUs in a node. In this case, it is inevitable to face the scenario that GPUs are connected with different CPUs (called different NUMA regions), thus causing cross-CPU passing latency. As shown in Fig. 1, GPUs connected with the same CPU have nearly 2× larger transmission bandwidth compared to the GPUs which are connected across CPU-QPI transmission, because the data transmission path includes two CPUs, while the transmission between GPUs within the same NUMA region can be finished by two GPUs using GPU p2p technology and it bypasses CPUs.

DCUDA does not consider the NUMA effect. The main reason is that transmission across NUMA regions is not critical for DCUDA, due to the use of the optimization methods like handle pool and asynchronous migration. We find that the average transmission time cost is less than 100 milliseconds for a 12GB GPU migration, which accounts for only 0.01% - 0.3% of the total execution time. Thus, we do not consider the NUMA effect on the transmission overhead when optimizing GPU scheduling within a node.

C. The Monitor

The Monitor needs to track two kinds of information in realtime, the utilization of each GPU and the resource demand of each application, both on cores and memory.

1) Monitoring the Usage of GPU Cores: We first show how to monitor the core demands of each application. Note that in CUDA applications, most computing tasks of an application are performed by kernel functions. Thus, we can obtain applications' demands of GPU cores by evaluating the occupancy of GPU cores and the execution time of each kernel function, which can be obtained with the timer function. The kernels' occupancy of GPU cores can be estimated by using cuOccupancyMaxActiveBlocksPerMultiprocessor() as well as some parameters of the kernel functions, including the pointer of the kernel function, the number of blocks, the number of threads per block, and so on.

However, evaluating the occupancy of GPU cores each time when a kernel function is called will severely degrade the applications' performance, e.g., it may cause 20%-30% performance slowdown. To handle this problem, DCUDA evaluates a kernel function and records its information when it is called the first time, and uses this recorded information when the kernel is called again with the same system parameters (e.g, the number of threads). The rationale is that GPU applications are usually iteration-based computing, so they may call the same kernel function many times. With this method, DCUDA can still accurately monitor the usage of GPU cores with a small overhead.

On the other hand, to monitor the realtime utilization of each GPU, DCUDA uses a thread to periodically scan the resource demand of each application, and then aggregates them together. We point out that this thread introduces negligible overhead and its accuracy is guaranteed by the accuracy of monitoring each application.

In summary, by tracking only the parameters of API calls with some optimizations, DCUDA can monitor both GPUs and applications with high accuracy and negligible overhead, before the tasks' execution. Different from the high overhead in traditional monitoring tools using API-replying like nvprof [14], our experiment results validate that DCUDA brings very little overhead. Meanwhile, it achieves higher than 90% accuracy.

2) Monitoring the Usage of GPU Memory: A CUDA application allocates most of its needed GPU memory with APIs like cuMemAlloc(), so we can obtain the allocated memory size and the range of virtual addresses from the parameters of the allocation APIs, and futher aggregate them to get the GPU deive memory utilization. However, not all allocated GPU memory would be used by the application and unified memory only maps virtual addresses to physical GPU memory when they are being accessed by the application. For example, in some machine learning platforms implemented based on the CUDA libraries, such as Tensorflow [10] and theano [13], applications usually allocate the whole GPU memory, while they may only use a very small portion of it. To avoid prefetching unused data, we need to detect the actual usage of GPU memory.

To detect the actual usage of GPU memory, we propose a monitoring method by checking whether the memory page is zero-paged, that DCUDA will mark the zero page as a free page

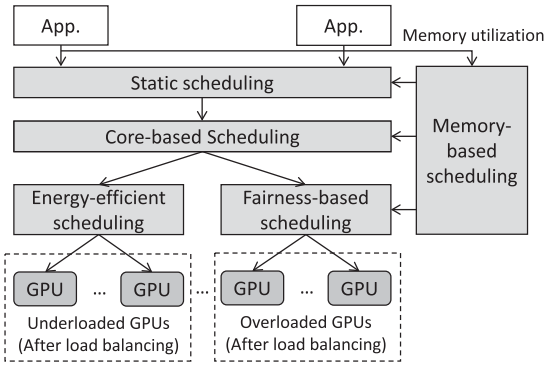


Fig. 6. Scheduling flow of DCUDA.

no matter whether this page is allocated or not. This method mainly has two challenges. The first one is the page-fault allocation and external kernel function cost caused by GPU memory check. The second one is the high memory access cost for the whole memory space. We use a CPU-side monitor approach and sample-based design to solve these problems.

When the monitor is trying to access GPU memory to check zero-page, page-fault will be triggered and the accessed memory page will be truly mapped on the GPU, which may be a serious waste before the application's access. In addition, GPU-side memory access should run a kernel function on the GPU device, leading to frequent switches of kernel execution contexts, and finally, prolonging applications' execution time. To overcome this problem, DCUDA makes use of the unified memory's characteristics. By accessing GPU memory at the CPU side, the monitor can call a memory access function at the CPU side to check GPU memory's status, so it has no effect on GPU kernel execution, and because of remote memory access, no external GPU memory mapping will happen in the monitored process, and there is no GPU memory being wasted.

For the other challenge, i.e., the large overhead caused by checking all memory pages, we use a sampling method to reduce the monitor overhead. Specifically, we sample memory usage information with fixed step size (i.e., 8 MB). For every 8 MB region, we only check one page, and randomly check a page in the 8MB region. We find that this monitor step size is suitable for realizing high accuracy and low latency. This result is also used in migration, if this page is being used, then we migrate the whole 8 MB region data via prefetching.

The memory monitor also supports collection of each GPU's memory utilizatiom, like the core monitor, it first collects memory demand of each application, by recording applications' allocated memory regions, and then aggregates them to each GPU, which can be higher than 100% when swap happening.

D. The Scheduler

DCUDA develops a scheduler to manage running applications on GPUs (see Fig. 6), and it considers core utilization, memory utilization, energy-efficient and fairness.

First, in the core-based scheduling part, DCUDA decides the candidate applications to be migrated from an overloaded GPU to an underloaded GPU so as to achieve dynamical load balance. DCUDA adopts hysteresis control to manage the load balancing operation and uses a greedy policy to determine the applications to be migrated. These methods can efficiently reduce the overloaded time of GPUs, prevent flip-flop migrations (the "*ping-pong*" effect) and reduce the migration times as well as the number of migrated applications.

After doing load balancing, it is still possible for GPUs to be either overloaded with heavyweight applications, or underloaded with multiple lightweight applications. So DCUDA further explore the opportunity of additional optimizations. DCUDA energy-efficient scheduling part leverages energy awareness by compacting several lightweight applications on underloaded GPUs. And DCUDA fairness-based scheduling part ensures the fairness of applications by adopting a priority-based time slicing policy to schedule the applications concurrently running on the same GPU.

Finally, because memory influences the performance only when thrashing happens, we split the memory scheduling as an external module. When potential thrashing is detected, the memory scheduling part will take effect and influence other parts in the scheduling work flow, including static scheduling, load balancing and fairness scheduling. In the following, we introduce the details of each part.

1) GPU Core Awareness: Note that the first key issue in scheduling is to balance the loads of cores between GPUs. So in this part, we just focus on the utilization on computing cores, as shown in Algorithm 1. First, we need to find out a core overloaded GPU and a core underloaded GPU, and then shift some tasks from the overloaded one to the underloaded one. However, to realize this idea, an underloaded GPU may become overloaded right after the migration, and then it may trigger another migration immediately. To alleviate this kind of "*ping-pong*" effect, DCUDA leverages hysteresis control to classify GPUs into three states with two threshold parameters, $Thresh_{over}$ and $Thresh_{under}$, according to their utilization. Specifically, if the utilization of a GPU is greater than $Thresh_{over}$, then we classify this GPU as an *overloaded* GPU, and if the utilization is smaller than $Thresh_{under}$, then we say this GPU is *underloaded*. Otherwise, we consider this GPU to be in a *normal* state. Based on this classification, DCUDA checks every pair of GPUs, and selects a pair as candidate for live migration if one GPU is overloaded and the other is underloaded. By defining a normal state, DCUDA can avoid GPUs changing too frequently between overloaded state and underloaded state, and thus alleviates the "*ping-pong*" effect.

After selecting a pair of candidate GPUs, it is also important to determine which applications on the overloaded GPU should be migrated to the underloaded GPU. The goal is to migrate as few applications as possible so as to reduce the number of migrations which directly affects the migration overhead. DCUDA uses a *greedy* policy to balance the load between the GPUs by always choosing the *most heavyweight and feasible application* for migration, i.e., the application which has the largest resource demand but will not make the underloaded GPU become overloaded if it is migrated. Finally, DCUDA adds all selected applications into a candidate list to wait for real migration. Note

Algorithm 1: GPU Core Awareness Scheduling.

```
candidate = new(List), util = 0
//GPUList is descending order by CoreUtil
//Select over from the start and under from the end
for all i in [1, 2, ..., len(GPUList)/2] do
  over, under = GPUList[i], GPUList[−i]
  if under.CoreUtil < Thresh_under &
  over.CoreUtil > Thresh_over then
    //TaskList is descending order by CoreUtil
    for all task in over.TaskList do
      c = under.CoreUtil + task.CoreUtil + util
      if c < Thresh_over then
        candidate.add(task)
        util += task.CoreUtil
      end if
    end for
    Migrate(under, over, candidate)
  end if
end for
```

that the greedy policy tries to migrate heavyweight applications first and tries to migrate as many applications as possible in one operation so as to reduce the overhead.

DCUDA's load balancing algorithm can scale for a larger number of GPUs with little overhead. Although we scan every pair of GPUs to find candidates for migration, we only perform migration between overloaded and underloaded GPUs based on the scheduling priority and only introduce a little scanning overhead.

2) Energy Awareness: As mentioned before, static scheduling methods usually assign applications to all GPUs to achieve better performance. However, such an assignment makes all GPUs active, even though some of them may be under-utilized, e.g., when the whole system load is low. Even in the heavy load case, the load-balancing operation only reduces the load on overloaded GPUs, but it may still leave some GPUs being underloaded.

On the other hand, the energy consumption of GPUs depends on their load, e.g., a GPU which stays in the idle state consumes only a little energy, but running even a single application on it may increase the energy consumption a lot as it needs to wake up the GPU from the idle state. However, running one more application on active GPUs only increases the energy consumption a little, comparing to the energy consumption caused by waking up from the idle state. Thus, compacting lightweight applications to run on fewer GPUs and letting more GPUs stay idle will save a lot of energy.

To achieve this goal, after calling the load-balancing operation, DCUDA further scans all GPUs and finds the two most under-loaded GPUs. If the applications running on the two GPUs can be compacted together to run on only one GPU, then DCUDA migrates the applications from one GPU to the other and let one GPU stay idle, as shown in Fig. 7. DCUDA repeats the above steps until no GPU pair can be compacted. We like to emphasize that this energy-aware scheduling is performed

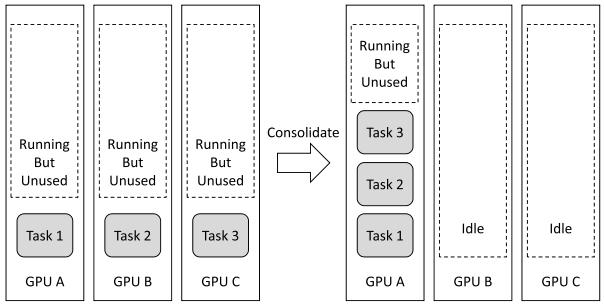


Fig. 7. Energy aware scheduling.

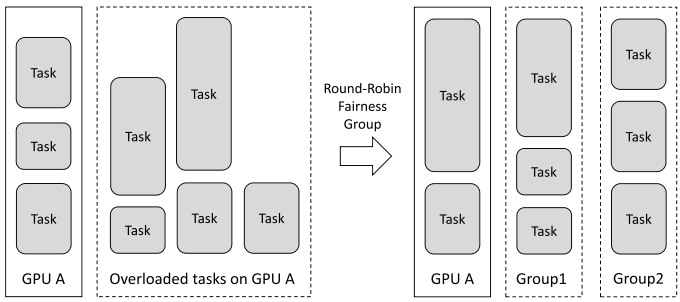


Fig. 8. Fairness aware scheduling.

after load balancing, and more importantly, it does not affect the performance as the compaction is performed only when the computing resource demand of these applications can be satisfied by one GPU.

3) Fairness Awareness: The load balancing operation can only reduce the variance of loads between GPUs, and it is still possible that some GPUs are overloaded after load balancing. When multiple applications with very different resource core demands running on a single overloaded GPU, the applications with low resource demand are harder to compete for a fair share of computing resource and result in a very long execution time. Thus, how to ensure fairness among multiple applications concurrently running on overloaded GPUs is also important. DCUDA uses a priority-based time slicing policy to guarantee the fairness.

First, DCUDA divides time into many slices (i.e., 100ms) and classifies applications on an overloaded GPU into multiple groups. Then DCUDA allocates time slices equally to each group, and allows only one group of applications to run at each time slice, as shown in Fig. 8.

To decide which applications belonging to a group, the key principle is to make sure that the applications in same groups should try to fully use resources and not cause severe competitions on computing resources. So the total resource demand of the applications in a group should be close to the total computing power of the GPU as much as possible. DCUDA also allow the total demand to slightly exceed the computing power for the consideration of GPU utilization, for example, one application with 100% resource requirement and one with 5% can be compact into one group, instead of letting 5% usage application exclusive using resources in one group.

TABLE I
PRIORITY IN DCUDA SCHEDULING PAIR SELECTION (CHOOSE PAIR WITH THE HIGHEST PRIORITY DIFFERENCE AND MIGRATE FROM HIGHER PRIORITY TO LOWER PRIORITY ONE)

Priority	Core Util.	Memory Util.
1 (Highest)	Overloaded	Overloaded
2	Overloaded	Underloaded
3	Underloaded	Overloaded
4 (Lowest)	Underloaded	Underloaded

Note that though allowing group's demand to slightly exceed GPU computing power can improve utilization, it may still lead to the problem of unfair competition between applications within the same group. To handle this problem, DCUDA dynamically adjusts the priority of applications within the same group, set higher priority to applications with low resource demand to increase their share of computing resource so as to guarantee a fair competition. This priority scheme does not cause significant slowdown on the performance as low resource demand tasks only need a little GPU resource to complete their computing tasks.

4) Memory Awareness: Last but not least, as we mentioned before, memory over-subscription is a potential performance bottleneck because of the page thrashing problem. To solve this performance challenge, we propose the memory-aware scheduling. It works only when memory overloading happens and if not, the scheduling policy will return to the previous three parts without external performance influence. The key design is to reduce the proportion of memory overloaded time by restricting the memory usage when designing the scheduling policies.

On the one hand, DCUDA uses a memory-concerned least-loaded static policy before the dynamic scheduling. When a new task arrives, the scheduler will choose candidates from devices with enough free memory space as the first priority, and if there is no suitable device, then it returns all the devices as candidates. After generating candidates, we use the normal least-loaded static policy to select the device to load. DCUDA uses a static watermark to determine whether it has a chance to encounter memory thrashing, e.g., the device with memory usage higher than 90%, and DCUDA will not choose the devices having the risk of thrashing as the candidate.

On the other hand, in the load balance part, both core and memory are considered in the scheduling, so DCUDA provides a dynamic scheduling policy by giving a priority definition as shown in Table I, and uses scheduling progress as shown in Algorithm 2. We think that the device with both core and memory being overloaded is the most critical candidate, then we focus on the device with core overloaded because DCUDA takes the cores as the highest priority for scheduling, and finally we consider the devices with only memory being overloaded. DCUDA selects a candidate GPU pair according to the above defined priority, the GPU pair has the largest difference of priorities will be chosen. When there are multiple pairs with the same difference value, we choose the pair with the highest difference of core utilization. After choosing a candidate GPU pair, DCUDA selects migration tasks by predict the memory usage, according to the memory monitor, and prevent overloaded and "*ping-pong*" effect.

Algorithm 2: Core and Memory Awareness Scheduling.

```
candidate = new(List), utilcore = 0, utilmem = 0
//GPUList is ascending order by Priority
//Select over from the start and under from the end
for all i in [1, 2, ..., len(GPUList)/2] do
  over, under = GPUList[i], GPUList[−i]
    isCorePair=under.CoreUtil < Thresh_under &
     over.CoreUtil > Thresh_over
    isMemPair=under.MemUtil < Thresh_mem &
     over.MemUtil > Thresh_mem
  if isCorePair || isMemPair then
    //TaskList is descending order by CoreUtil
    for all task in over.TaskList do
      c = under.CoreUtil + task.CoreUtil + utilcore
      m =
       under.MemUtil + task.MemUtil + utilmem
      if c < Thresh_over & (m < Thresh_mem ||
      under.MemUtil > Thresh_mem) then
        candidate.add(task)
        utilcore += task.CoreUtil
        utilmem += task.MemUtil
      end if
    end for
    Migrate(under, over, candidate)
  end if
end for
```

Moreover, in the fairness aware part, when applications on overloaded GPU are split into different groups, a group with the total memory usage being larger than the GPU memory is also unsuitable, so we will limit each group's memory requirement to be lower than 100%. The same consideration is used in the energy aware part, DCUDA prevents the total memory cost from being larger than 100% in compaction to a single GPU. This two part is easy to understand, as Figs. 7 and 8 shown, the GPU space in illustrate means both GPU core and memory resource in the memory awareness model.

E. The Migrator

The Migrator is responsible for performing live migration of specified applications from source GPU to target GPU. We note that our live migration method only migrates the unlaunched kernels, i.e., the kernels which have not been launched to GPUs. But we emphasize that it is still necessary to migrate the unlaunched kernels. Because lots of kernels are blocked in the CPU side due to various synchronous operations.

As illustrated in Fig. 9, the key issues and challenges of the live migration are (1) how to efficiently clone a consistent runtime environment on the target GPU? (2) how to reduce the overhead of migrating memory data? and (3) how to guarantee the consistency of the computing tasks contained in a running application after migration?

1) Cloning Consistent Runtime Environment: By analyzing CUDA applications, we find that the runtime environment of an application includes kernel binaries, streams, and relevant

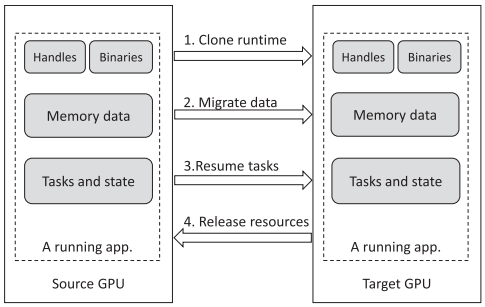


Fig. 9. Process of live migration.

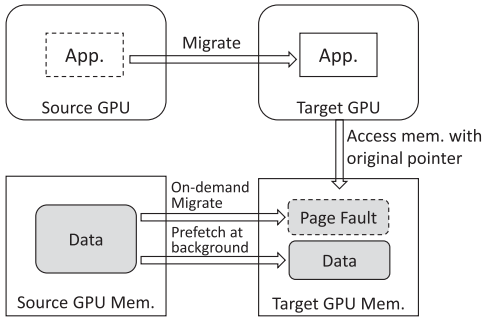


Fig. 10. Data migration with unified memory.

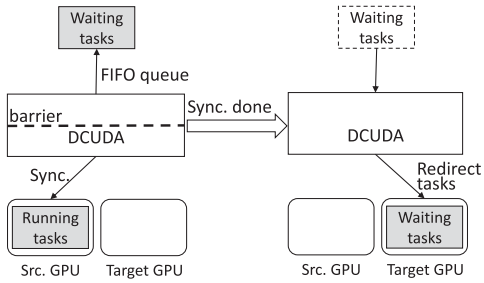


Fig. 11. Procedure of migrating tasks.

libraries' handles, such as cublas handle, cudnn handle, cufft handle and so on. These variables hold all the management data which controls and uses GPUs. Thus, to clone a consistent runtime environment, we first initialize all needed libraries and create new handles of these libraries on target GPU, then copy the configuration information from the corresponding variables on the source GPU. In addition, we need to register the binary of kernel functions so that they could be called by the applications on target GPU.

Note that we also discovered that cloning runtime causes a large overhead, and it mainly comes from libraries initialization, and in particular, the time needed by initializing the necessary libraries of an application may be as high as 200 ms - 400 ms, which accounts for more than 80% of the total cloning time. To reduce this overhead, we employ a handle pooling technique by maintaining a pool of libraries' handles for each GPU which initializes libraries and creates handles at background. During the cloning, DCUDA can immediately fetch the handles of required libraries from the handle pool, instead of creating new handles.

To reduce the overhead caused by registering the binaries of kernel functions, considering that many binaries may not be needed by the remaining tasks of an application after being migrated to the target GPU, so DCUDA uses an on-demand policy to register the binary of kernel functions. Specifically, DCUDA maintains a copy of the binary of kernel functions required by applications in CPU memory and records the relationship between each binary and its corresponding kernel function.

2) Migrating Memory Data: Note that the key challenge of live migration is to migrate memory data, because it requires to keep the virtual memory addresses of the application data on target GPU being exactly the same as those on source GPU. DCUDA addresses the issue of preserving the same virtual memory addresses by leveraging unified memory. Specifically, when an application is triggered to be migrated, we just need to guarantee all the memory of this application is allocated with the unified memory, and we can run tasks of this application on the target GPU immediately without explicitly migrating data first, as shown in Fig. 10. Accessing data not residing on the target GPU causes page fault which triggers data migration.

However, two problems need to be addressed when taking use of unified memory. The first problem is that most applications do not allocate GPU memory with unified memory. To transparently support unified memory in these applications, DCUDA intercepts all GPU memory allocation APIs and replaces them with unified memory allocation APIs, and finally returns unified pointers to applications.

The second problem is that on-demand migration with the support of unified memory may trigger many page faults, which also introduce a large overhead. To mitigate this problem, DCUDA uses a thread at background to asynchronously prefetch data to the target GPU, and the overhead of moving data is hidden in the computation time. Furthermore, this way of prefetching also eliminates data migration overhead across NUMA regions so that DCUDA does not need to take into consideration the transmission latency across NUMA regions when doing migration.

3) Resuming Computing Tasks: An application may consist of multiple computing tasks (i.e., kernels), and these tasks are usually submit to GPUs in batches. As a result, when migrating a running application, it is possible that some of the computing tasks are still running, which we call "*running tasks*", but others are waiting for execution, which we call "*waiting tasks*". That is, the running tasks have been submitted to GPU but have not completed the computation, and the waiting tasks are still waiting to be submitted. Thus, we need to migrate all waiting tasks to the target GPU and wait for the running tasks to finish first so as to preserve the executing order of all computing tasks.

DCUDA uses two techniques to guarantee the executing order of computing tasks as shown in Fig. 11. DCUDA first sends a synchronization command to the source GPU to wait for the completion of all running tasks, then resumes the waiting tasks on the target GPU to preserve the executing order between running tasks and waiting tasks. Besides, DCUDA preserves the executing order among waiting tasks by managing all waiting tasks with a FIFO-queue based on their submitted order during

TABLE II
BENCHMARKS

Suite	Num.	Name of Applications
CUDA Samples	4	[1]MatrixMul, [2]BlackScholes [3]eigenvalues, [4]transpose
SHOC	9	[5]Triad, [6]MaxFlops, [7]MD5Hash [8]Sort, [9]FFT, [10]Scan, [11]S3D [12]BFS, [13]Reduction
Tensorflow Bench.	7	[14]AutoencoderRunner [15]VariationalAutoencoderRunner [16]mnist_cnn, [17]mnist_mlp [18]alexnet, [19]Kmeans, [20]cifar10

the synchronization. Note that DCUDA needs to replace the handles used by the waiting tasks with new handles belonging to the target GPU. After the migration completes, the corresponding resources on source GPU will be released.

IV. Evaluation

To evaluate DCUDA, we implemented a prototype based on CUDA toolkit 8.0. For comparison, we also implemented the Least-Loaded scheduling scheme in our prototype, because it is the most practical and efficient scheme within the class of static scheduling algorithms and has been widely studied in gCloud [16], Rain [34], and Strings [35]. In particular, we evaluate DCUDA to answer the following questions.

- How large are the overheads of CPU cycles and system memory introduced by DCUDA (Section IV-B)?
- How much improvement can DCUDA achieve for load balance between GPUs (Section IV-C1) and reduction of application execution time (Section IV-C2)?
- How much improvement can DCUDA achieve for fairness and QoS between applications (Section IV-C3)?
- What is the impact of different load levels on the performance and the energy saving of DCUDA (Section IV-D1)?
- How is the efficiency of the memory concerned scheduling in DCUDA, and how does it improve the load balance and the speed up the application execution (Section IV-D2)?

A. Setup

We conducted our experiments on a server with two Intel Xeon E5-2620 v4 2.10 GHz processors, 64 GB system memory, and four NVIDIA 1080Ti GPUs, which are based on the PASCAL architecture and interconnected with PCIe. Each GPU has 3,584 computing cores and 12 GB memory.

We select twenty distinct benchmark programs taken from the CUDA Samples [2], SHOC [15], and Tensorflow Benchmarks [7]. Table II lists all of the workloads used in our evaluation. We emphasize that these benchmarks represent a majority of GPU applications, including high performance computing (MatrixMul), data mining (Kmeans), machine learning (Mnist_mlp), graph Algorithm (BFS), and deep learning (Mnist_cnn) [2], [7], [15].

Since we focus on the scenario of GPU sharing which naturally requires multiple applications concurrently run on a GPU server node, we evaluate DCUDA by generating a workload which combines all the twenty benchmark programs together. Precisely, we sequentially submit the twenty benchmark programs to the prototype system with a fixed time interval, and let them compete for the GPU resources. The interval is set as $5s$ by default in our core-intensive experiment on Section IV-C, smaller than the executing time of a benchmark program (around $30s$) so as to simulate a medium-weight workload, and the length of the arrival interval is adjusted to show different load levels impact on DCUDA (see Section IV-D1). We select 50 random application arrival sequences in all 20! ($2.43e^{18}$) possible sequences to evaluate DCUDA.

The threshold settings are chosen by heuristics based on experimental results. We set the threshold $Thresh_{over}$ as 100% default, and for $Thresh_{under}$, considering that as long as the GPU utilization is smaller than 100%, it has a chance to be further improved, so we set default $Thresh_{under}$ as 90% so as to achieve high GPU utilization. These threshold parameters are verified to work well for many applications. And we set the monitoring interval as $100ms$.

Note that in this task setting, the requirements of memory in most times are lower than the memory capacity of the GPUs. Thus, for these workloads, the memory scheduling part will not take effect, we call these tasks core-intensive workloads, so as to differentiate the scenarios of memory heavy loads. We will evaluate the performance under memory heavy workloads in Section IV-D2.

B. Overhead of DCUDA

Overhead of Monitoring and Scheduling. We first evaluate the overhead of CPU cycles and memory usage caused by the monitoring and scheduling processes in DCUDA. Our experiments show that DCUDA uses no more than 0.2% CPU and consumes around 7MB system memory only. This is mainly because our monitoring and scheduling mechanisms are both lightweight, e.g., our monitoring scheme just tracks the usage information from the parameters of API calls and uses CPU-side access to bypass GPU cores, and the scheduling mechanism only needs to run when some GPUs become overloaded, so they only consume a few CPU cycles. In terms of memory overhead, DCUDA only needs to keep the metadata of each application, including handle pointers, kernel binaries and so on, so the metadata size is small compared to the whole memory size. In summary, the memory and CPU overheads of DCUDA are both negligible. We point out that the lightweight monitoring scheme in DCUDA also achieves very high accuracy, due to page limit, we do not show this result, and instead, we show the improvement of DCUDA in load balancing which relies on the accurate monitoring results (see Section IV-C1).

Overhead of Unified Memory. Next, we also evaluate the performance loss caused by the support of unified memory in all scheduling applications. The performance slowdown caused by unified memory is within 1% in most test cases, and the average performance loss is 0.96%. Thus, the overhead of unified

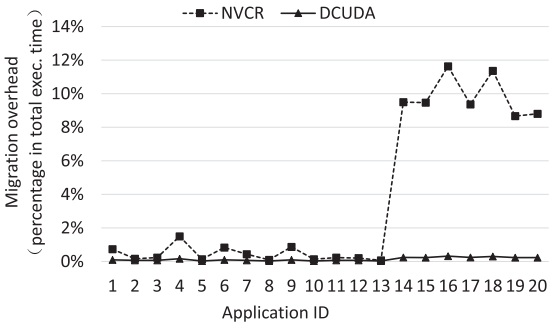


Fig. 12. Reduction of migration overhead with DCUDA.

memory can be negligible, compared with the improvement brought by DCUDA (see Section IV-C2).

Overhead of Live Migration. We further evaluate the time overhead of the live migration process with DCUDA, and compare it with NVCR [30], which is also a live migration approach. The migration overhead is measured as the percentage of the time for one single migration to the total execution time of each application. The results are shown in Fig. 12, we can see that DCUDA uses less than 100 milliseconds to migrate a running application, which accounts for only 0.01% - 0.3% of the total execution time of an application. In general, DCUDA can reduce up to 97.4% migration overhead compared with NVCR. Because the overhead of cloning runtime environment already becomes negligible due to the handle pooling technique used by DCUDA, and the overhead of migrating data can be hidden in the execution of computing tasks with the on-demand data migration technique.

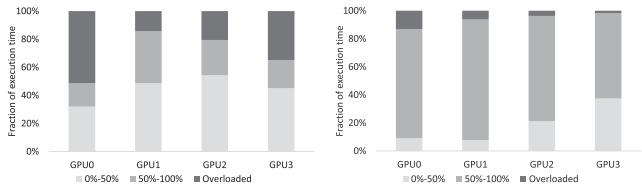
More importantly, NVCR cannot work correctly in the GPU sharing scenario for scheduling applications, because it adopts the replaying technique to keep virtual address unchanged after migration, specifically, it replays memory-related API calls on the target GPU. However, in GPU sharing scenarios, some virtual addresses may be occupied by other applications, so NVCR cannot work correctly. DCUDA is the first work which supports universal live-migration and it can work well in all scenarios, including GPU sharing with multi-tenants.

C. Performance Under Core-Intensive Workloads

The main benefit of DCUDA is to balance the loads on GPU cores via dynamic scheduling, so we first evaluate the performance of DCUDA under core-intensive workloads, and we compare DCUDA with the Least-Loaded scheduling scheme.

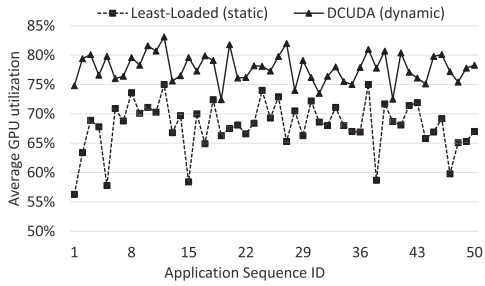
1) Improvement of the Load Balance of GPU Cores: We classify each GPU into three states like in Section II-B, based on its utilization, which is the ratio of the computing resource demand of all applications running on the GPU to its resource capacity, i.e., 0%-50% utilization, 50%-100% utilization, and overloaded state, and show the fraction of time of being at each state for each GPU. We only show the results under one sequence of applications in Fig. 13, and the results are similar for other sequences.

From Fig. 13(a), we find that when using the Least-Loaded scheduling, GPUs are very likely to become overloaded, e.g.,

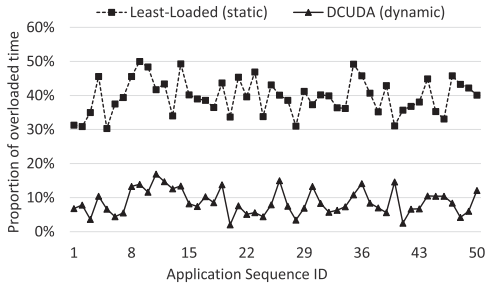


(a) Least-Loaded scheduling (b) DCUDA

Fig. 13. Load of GPU under one application sequence.



(a) Average GPU utilization



(b) Proportion of overloaded time

Fig. 14. GPU utilization under different app. sequences.

the overloaded time of each GPU accounts for 14.3% - 51.4% of the whole running time. And the utilization of other GPUs may be very low ($< 50\%$) for a long time even some GPUs are already overloaded.

As shown in Fig. 13(b), under the same workload, DCUDA significantly improves the load balance between GPUs compared to the Least-Loaded scheduling. Specifically, DCUDA reduces the overloaded time of GPUs by 79.5%, and improves overall GPU utilization by 38.1%. Moreover, the overloaded time of each GPU is always within 6% and the underloaded time of each GPU is also greatly reduced. The main reason of this improvement is that DCUDA can migrate running applications from overloaded GPUs to underloaded GPUs when overload or underload situation occurs.

We further evaluate the performance of DCUDA in load balancing under all 50 application sequences. Fig. 14(a) shows the average GPU utilization and Fig. 14(b) shows the proportion of overloaded time. We can see that DCUDA can achieve a large improvement in load balancing under all workloads. In particular, comparing with the Least-Loaded scheme, DCUDA can improve the GPU utilization by 14.6% and reduce the overloaded time by 78.3% on average.

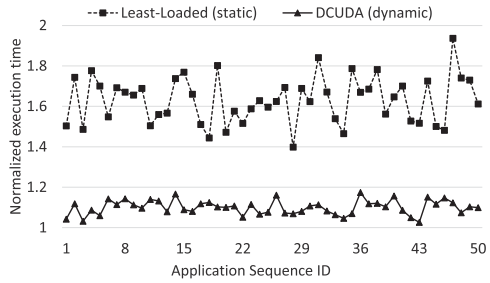


Fig. 15. Average execution time of applications under different application sequences.

2) Execution Time Reduction: In this section, we evaluate the benefit that comes from the improvement of core load balance by comparing the average execution time of applications under different general application sequences. We normalize the execution time of applications to their single execution. The results are shown in Fig. 15. We can see that DCUDA can reduce the average execution time of all applications by up to 42.1% compared to the Least-Loaded scheduling. Moreover, DCUDA achieves a more stable performance across different workloads, e.g., the difference of the average execution time of different workloads is always within 20%. This is because DCUDA achieves better load balance and mitigates resource contentions between applications.

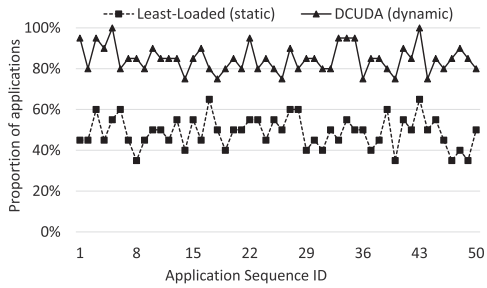
3) Improvement of QoS and Fairness: Besides the execution time, Quality of Service(QoS) and fairness are also two important metrics to measure the efficiency of scheduling methods. First, we analyze the performance degradation of each application in a shared scenario compared to its single execution. If the performance degradation is less than 20%, we see that the QoS requirement is satisfied. We count the proportion of applications which can satisfy the QoS requirement under different application sequences. The results are shown in Fig. 16(a). We observe that across the 50 application sequences, on average, more than 80% of the applications achieve the QoS goal using DCUDA, and for some sequences, the proportion of applications that satisfy the QoS requirement is up to 100%.

Next, we use the Jain's fairness index to measure the fairness between applications with different scheduling methods. Jain's fairness index is a number between zero and one [24], and one indicates perfect fairness (i.e., concurrent executing processes experience equal performance slowdown), while zero indicates no fairness at all. We evaluate the fairness index of applications under different sequences and show the results in Fig. 16(b). From the figure, we find that DCUDA can improve the fairness index by 12.1% on average compared with Least-Loaded scheduling. Furthermore, the fairness index under DCUDA is very close to one, which means that DCUDA almost guarantees the perfect fairness.

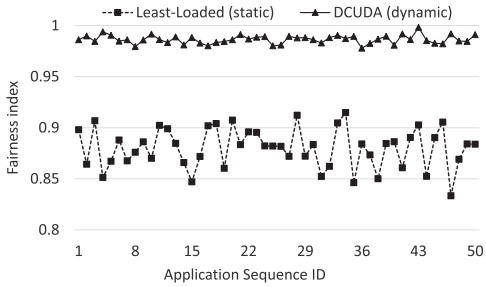
D. Performance Under Different Workloads

In this subsection, we evaluate DCUDA on different core and memory demands to further show the effectiveness of DCUDA.

1) Impact of Heavy Loads on GPU Cores: Note that DCUDA reduces the execution time of applications by balancing the



(a) Proportion of applications satisfying QoS



(b) Fairness between applications

Fig. 16. QoS and fairness under different app. sequences.

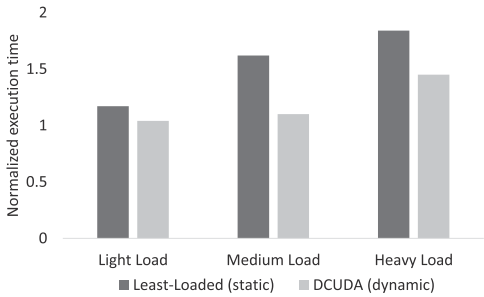


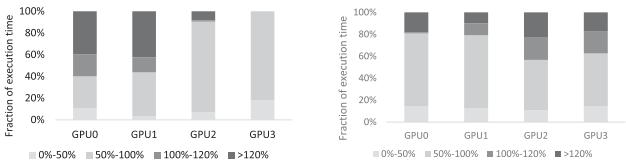
Fig. 17. Reduction of the execution time of applications with DCUDA under different loads.

loads between GPUs to improve GPU utilization, so clearly the improvement of DCUDA may depend on the load levels of applications submitted to GPUs. To show the effectiveness of DCUDA, we consider different load levels by adjusting the time interval of application arrivals. In particular, we vary the length of the arrival interval from $7s$, $5s$ to $3s$ to represent the cases of light load, medium load, and heavy load. The results of execution time are shown in Fig. 17. We can see that the improvement of DCUDA, which is measured by the reduction of average execution time of applications, is the largest under medium load. The reason is that if all GPUs are overloaded or all are underloaded, then there is not much room to further improve GPU utilization by balancing the load, so the benefit of dynamic scheduling should decrease. However, DCUDA still achieves a large improvement in a wide range of load levels, because the scenario of load imbalance is usually very common.

We also show the improvement of DCUDA in energy saving by comparing the energy consumption of all GPUs with DCUDA and Least-Loaded scheduling. The results are shown in Table III. We can see that DCUDA can save more energy if the system load

TABLE III
REDUCTION OF THE ENERGY CONSUMPTION OF APPLICATIONS WITH DCUDA UNDER DIFFERENT LOADS

	Light Load	Medium Load	Heavy Load
Least-Loaded	81201J	74935J	70611J
DCUDA	70449J	70921J	68771J



(a) Without memory scheduling (b) With memory scheduling

Fig. 18. Memory load of each GPU.

is lighter, which is 13.3% in the light load case in our setting. This is because when more GPUs are underloaded in the light load case, DCUDA has more opportunities to compact applications to run on fewer GPUs and make more GPUs stay at idle so as to save more energy. However, with the increase of system load, the opportunity to compact multiple applications to fewer GPUs also decreases, and this is the scenario in which load balancing operation can play an important role.

2) Impact of Heavy Loads on Memory: To further show the performance improvement benefited from the memory scheduling policy in DCUDA, we select a set of workloads which have heavy loads on memory, including 12 times alexnet (each needs 6GB memory), 15 times vgg (each needs 7 GB) and 8 times Mnist_cnn (each needs 5 GB) [7], [9], so it requires 300% more memory than the actual GPU memory. We sequentially submit these applications to the DCUDA prototype system with a fixed time interval, which is set as 4 seconds. We select 20 random arrival sequences to evaluate DCUDA with and without the memory scheduling.

Improvement of Balancing Memory Load. We first evaluate the improvement of DCUDA in memory load balance. We classify each GPU into four states based on its utilization, which is the ratio of the memory demand of all applications running on the GPU to its memory capacity, and show the fraction time of being at each state for each GPU. Like Section IV-C1, we only show the results under one sequence of applications, other sequences have similar results. From Fig. 18(a), when using DCUDA without memory scheduling, GPUs are very likely to become overloaded, e.g., the overloaded time of GPU0 and GPU1 accounts for over than 40% of the whole running time. The utilization of other GPUs like GPU3 is very low ($< 50\%$) for a long time even some GPUs are already overloaded. As shown in Fig. 18(b), under the same workload, DCUDA improves the memory load balance between GPUs compared to the one without memory scheduling, the overloaded time of each GPU reduces to under 20%. The main reason of this improvement is that DCUDA's memory scheduling policy can reduce the time when memory is overloaded by migration.

Reduction of Execution Time. We now evaluate the benefit that comes from the memory load balance improvement by

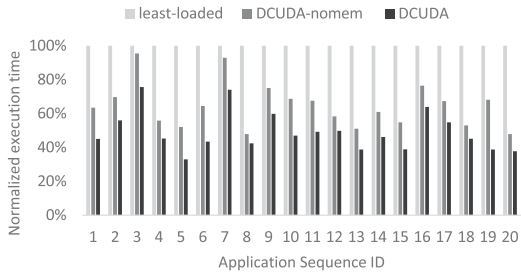


Fig. 19. Reduction of the execution time in the situation when memory is heavy-loaded.

comparing the total execution time of applications under different application sequences. We normalize the execution time of applications to least-loaded scheduling execution time. The results are shown in Fig. 19. DCUDA can reduce the average execution time of all applications by up to 67% compared to that of the Least-Loaded scheduling, and the lower bound of the speedup is higher than 30%. This is because DCUDA reduces the overloaded time and eliminates overhead of memory swap and memory thrash.

V. RELATED WORK

GPU Scheduling. To improve GPU utilization in shared environment, many scheduling schemes are proposed to schedule applications between multiple GPUs, including Round-Robin scheduling [23], Least-Loaded scheduling [16], [34], [35], and Prediction-based scheduling [39]. In gCloud [16], Khaled et al. find that Least-Loaded scheduling can improve the performance of applications by 10.3% than Round-Robin policy. Sengupta et al. [34] also employ a weighted version of the Least-Loaded policy to schedule applications between GPUs by taking into account the different capability of each GPU in a heterogeneous system. Yash et al. [39] develop Mystic, which predicts the resource demand of applications to guide scheduling, while this scheme requires to pre-execute applications for five seconds, and thus brings a large overhead.

GPU Virtualization. For cloud providers, virtual machines should have isolated GPU devices to use, but exclusively using GPU resources will cause serious waste by traditional PCIe passthrough for GPU virtualization [1]. And Full-virtualization [25], [38], mediated pass-through [32], [38], para-virtualization [17] and SR-IOV [5] techniques have limitations prevent them to be actually used in cloud environments, like elasticity and portability, more optimization is needed [47]. API remoting [33], [42], [46] is another choice of virtualization technique that interposes a user-mode API to client-side, and forwards calls to server-side, with more flexibility and controllability. For containerized environments, API remoting calls are used more frequently. DGSF [19] optimize API-remoting specifically for the serverless functions, and Fluid [22] is a novelty cloud-native platform for DL training, with optimized scheduling scheme under the concern of data cache.

Resource Monitoring. Resource monitor is necessary to get the runtime resource utilities and requirements for efficient scheduling in a multi-task system. The traditional way is to

use hypervisor-level monitors like kubectl and cAdvisor, and GPU resource statistics tools like nvidia-smi with negligible overhead[11], [12], [41]. If more precise monitor results are needed, API-replaying monitors like nvprof [6] may cause serious performance influence, while API remoting platforms can collect forwarded API calls and collect information lightly[19], [46]. For certain scenarios like DL, more monitor metrics are needed [42], [44], [45], like Synergy [29] profiles the sensitivity of DNNs to auxiliary resources and allocates them disproportionately among jobs.

Live Migration. Live migration is an important feature in GPU sharing systems, which has been widely used for fault tolerance and load balance. Takizawa et al. first propose CheCUDA [37] which provides checkpoint and restart library for CUDA applications, but CheCUDA requires re-compilation of the application's source code and can not handle software in binary format. NVCR [30] also provides a live migration approach for CUDA application which works transparently without re-compiling source codes. In particular, NVCR can keep virtual address unchanged after migration by replaying memory allocation APIs in order on the target GPU.

VI. Discussion and Future Work

DCUDA also has some limitations. For one thing, the current version of DCUDA only supports APIs in CUDA toolkit 8.0. As most of the common API (e.g., *cuLaunchKernel*, *cuMalloc*, *cuMemAdvise*) and the GPU thread and memory model are not changed from CUDA 8.0 to the latest version, we believe that DCUDA's scheduling design can work well on the new version, and the challenge mainly comes from DCUDA's limited API remoting technical support. As DCUDA adds APIs by hand, keeping the client-side interception library up-to-date is not an easy job with the API number increasing, e.g., the basic CUDA Driver API Manual extending from CUDA 8.0 with 359 pages to CUDA 12.0.1 with 629 pages. In the future, we will research how to automatically collect all APIs supporting different types of parameters and generate the API remoting library without manually writing thousands of lines of code, given the latest CUDA library.

For another, DCUDA only performs migration and scheduling between GPUs intra a server. We will study live migration and scheduling techniques for GPUs across different servers in our future work.

VII. Conclusion

In this paper, we proposed DCUDA which supports dynamic scheduling of running applications between GPUs and is fully compatible to all CUDA applications. In particular, DCUDA accurately and efficiently estimates the resource demand of applications and GPU utilization with a lightweight scheme, and dynamically migrates running applications to achieve load balance between GPUs and improve GPU utilization. With DCUDA, both the execution time of applications and the energy consumption of GPUs can be significantly reduced.

References

[1] Cloud computing is still dangerously underutilized, 2017. [Online]. Available: https://gigenet.com/blog/underutilizing-cloud-computing-resources/

[2] CUDA samples, 2023. [Online]. Available: http://docs.nvidia.com/cuda/cuda-samples/index.html

[3] CUDA toolkit, 2023. [Online]. Available: https://developer.nvidia.com/cuda-toolkit

[4] nvdia-smi, 2023. [Online]. Available: https://developer.nvidia.com/nvidia-system-management-interface

[5] NVIDIA GRID, 2023. [Online]. Available: https://www.nvidia.com/en-us/data-center/virtual-gpu-technology/

[6] nvprof, 2023. [Online]. Available: http://docs.nvidia.com/cuda/profiler-users-guide/index.html

[7] Tensorflow benchmarks, 2023. [Online]. Available: https://github.com/tensorflow/benchmarks

[8] Unified memory on pascal, 2016. [Online]. Available: https://devblogs.nvidia.com/beyond-gpu-memory-limits-unified-memory-pascal/

[9] VGG 16, 2016. [Online]. Available: https://github.com/ry/tensorflow-vgg16

[10] M. Abadi et al., "TensorFlow: A system for large-scale machine learning," in *Proc. 12th USENIX Conf. Operating Syst. Des. Implementation*, 2016, pp. 265–283.

[11] H. Albahar, S. Dongare, Y. Du, N. Zhao, A. K. Paul, and A. R. Butt, "SchedTune: A heterogeneity-aware GPU scheduler for deep learning," in *Proc. IEEE 22nd Int. Symp. Cluster Cloud Internet Comput.*, 2022, pp. 695–705.

[12] T. Allen, X. Feng, and R. Ge, "Slate: Enabling workload-aware efficient multiprocessing for modern GPGPUs," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2019, pp. 252–261.

[13] J. Bergstra et al., "Theano: Deep learning on GPUs with python," in *Proc. NIPS, BigLearning Workshop*, Citeseer, 2011, vol. 3.

[14] L. Braun and H. Fröning, "CUDA flux: A lightweight instruction profiler for CUDA applications," in *Proc. IEEE/ACM Perform. Model. Benchmarking Simul. High Perform. Comput. Syst.*, 2019, pp. 73–81.

[15] A. Danalis et al., "The scalable heterogeneous computing (SHOC) benchmark suite," in *Proc. 3rd ACM Workshop Gen.-Purpose Comput. Graph. Process. Units*, 2010, pp. 63–74.

[16] K. M. Diab, M. M. Rafique, and M. Hefeeda, "Dynamic sharing of GPUs in cloud systems," in *Proc. IEEE 27th Int. Parallel Distrib. Process. Symp. Workshops PhD Forum*, 2013, pp. 947–954.

[17] M. Dowty and J. Sugerman, "GPU virtualization on VMware's hosted I/O architecture," *SIGOPS Oper. Syst. Rev.*, vol. 43, no. 3, pp. 73–82, Jul. 2009.

[18] M. Ferro et al., "Analysis of GPU power consumption using internal sensors," in *Proc. Anais do XVI Workshop em Desempenho de Sistemas Computacionais e de Comunicação*, 2017.

[19] H. Fingler, Z. Zhu, E. Yoon, Z. Jia, E. Witchel, and C. J. Rossbach, "DGSF: Disaggregated GPUs for serverless functions," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2022, pp. 739–750.

[20] C. Gregg, J. Dorn, K. M. Hazelwood, and K. Skadron, "Fine-grained resource sharing for concurrent GPGPU kernels," in *Proc. 4th USENIX Conf. Hot Topics Parallelism*, 2012, Art. no. 10.

[21] R. Gu et al., "Liquid: Intelligent resource estimation and network-efficient scheduling for deep learning jobs on distributed GPU clusters," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 11, pp. 2808–2820, Nov. 2022.

[22] R. Gu et al., "Fluid: Dataset abstraction and elastic acceleration for cloud-native deep learning training jobs," in *Proc. IEEE 38th Int. Conf. Data Eng.*, 2022, pp. 2182–2195.

[23] E. Hermann, B. Raffin, F. Faure, T. Gautier, and J. Allard, "Multi-GPU and multi-CPU parallelization for interactive physics simulations," in *Proc. Eur. Conf. Parallel Process.*, Springer, 2010, pp. 235–246.

[24] R. Jain, D.-M. Chiu, and W. Hawe, "A quantitative measure of fairness and discrimination for resource allocation in shared systems," Digital Equipment Corporation, Maynard, MA, USA, Tech. Rep. DEC-TR-301, 1984.

[25] J. Song, Z. Lv, and K. Tian, "KVMGT: A full GPU virtualization solution," *KVM Forum*, 2014. [Online]. Available: http://www.linux-kvm.org/page/KVM_Forum_2014

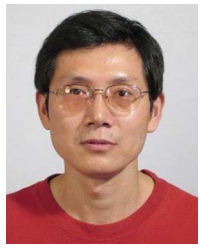
[26] A. K. Kulkarni and B. Annappa, "GPU-aware resource management in heterogeneous cloud data centers," *J. Supercomput.*, vol. 77, no. 11, pp. 12458–12485, Nov. 2021.

[27] C. Li et al., "Priority-based PCIe scheduling for multi-tenant multi-GPU systems," *IEEE Comput. Archit. Lett.*, vol. 18, no. 2, pp. 157–160, Jul.–Dec. 2019.
[28] N. Maruyama, T. Nomura, K. Sato, and S. Matsuoka, "Physis: An implicitly parallel programming model for stencil computations on large-scale GPU-accelerated supercomputers," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2011, pp. 1–12.
[29] J. Mohan, A. Phanishayee, J. Kulkarni, and V. Chidambaram, "Looking beyond GPUs for DNN scheduling on Multi-Tenant clusters," in *Proc. USENIX Conf. Operating Syst. Des. Implementation*, Carlsbad, CA, USENIX Association, 2022, pp. 579–596.
[30] A. Nukada, H. Takizawa, and S. Matsuoka, "NVCR: A transparent checkpoint-restart library for NVIDIA CUDA," in *Proc. IEEE Int. Symp. Parallel Distrib. Process. Workshops Phd Forum*, 2011, pp. 104–113.
[31] J. J. K. Park, Y. Park, and S. Mahlke, "Chimera: Collaborative preemption for multitasking on a shared GPU," *ACM SIGARCH Comput. Architecture News*, vol. 43, no. 1, pp. 593–606, 2015.
[32] B. Peng, H. Zhang, J. Yao, Y. Dong, Y. Xu, and H. Guan, "MDev-NVMe: A NVMe storage virtualization solution with mediated Pass-Through," in *Proc. USENIX Annu. Tech. Conf.*, Boston, MA, USENIX Association, 2018, pp. 665–676.
[33] J. Prades and F. Silla, "GPU-job migration: The rCUDA case," *IEEE Trans. Parallel Distrib. Syst.*, vol. 30, no. 12, pp. 2718–2729, Dec. 2019.
[34] D. Sengupta, R. Belapure, and K. Schwan, "Multi-tenancy on GPGPU-based servers," in *Proc. 7th ACM Int. Workshop Virtualization Technol. Distrib. Comput.*, 2013, pp. 3–10.
[35] D. Sengupta, A. Goswami, K. Schwan, and K. Pallavi, "Scheduling multi-tenant cloud workloads on accelerator-based systems," in *Proc. IEEE Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2014, pp. 513–524.
[36] L. Shi, H. Chen, J. Sun, and K. Li, "vCUDA: GPU-accelerated high-performance computing in virtual machines," *IEEE Trans. Comput.*, vol. 61, no. 6, pp. 804–816, Jun. 2012.
[37] H. Takizawa, K. Sato, K. Komatsu, and H. Kobayashi, "CheCUDA: A checkpoint/restart tool for CUDA applications," in *Proc. IEEE Int. Conf. Parallel Distrib. Comput. Appl. Technol.*, 2009, pp. 408–413.
[38] K. Tian, Y. Dong, and D. Cowperthwaite, "A full GPU virtualization solution with mediated Pass-Through," in *Proc. USENIX Annu. Tech. Conf.*, Philadelphia, PA, USENIX Association, 2014, pp. 121–132.
[39] Y. Ukidave, X. Li, and D. Kaeli, "Mystic: Predictive scheduling for GPU based cloud servers using machine learning," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2016, pp. 353–362.
[40] M. Vogelgesang, S. Chilingaryan, T. dos_Santos Rolo, and A. Kopmann, "UFO: A. scalable GPU-based image processing framework for on-line monitoring," in *Proc. IEEE 9th Int. Conf. Embedded Softw. Syst. IEEE 14th Int. Conf. High Perform. Comput. Commun.*, 2012, pp. 824–829.
[41] W. Xiao et al., "Gandiva: Introspective cluster scheduling for deep learning," in *Proc. 13th USENIX Conf. Operating Syst. Des. Implementation*, USA. USENIX Association, 2018, pp. 595–610.
[42] W. Xiao et al., "AntMan: Dynamic scaling on GPU clusters for deep learning," in *Proc. USENIX Conf. Operating Syst. Des. Implementation*, USA, USENIX Association, 2020, Art. no. 30.
[43] T. T. Yeh, A. Sabne, P. Sakdhnagool, R. Eigenmann, and T. G. Rogers, "Pagoda: A GPU runtime system for narrow tasks," *ACM Trans. Parallel Comput.*, vol. 6, no. 4, Nov. 2019, Art. no. 21.
[44] G. Yeung, D. Borowiec, A. Friday, R. Harper, and P. Garraghan, "Towards GPU utilization prediction for cloud deep learning," in *Proc. 12th USENIX Conf. Hot Topics Cloud Comput.*, USA, USENIX Association, 2020, Art. no. 6.
[45] G. Yeung, D. Borowiec, R. Yang, A. Friday, R. Harper, and P. Garraghan, "Horus: Interference-aware and prediction-based scheduling in deep learning systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 1, pp. 88–100, Jan. 2022.
[46] H. Yu, A. M. Peters, A. Akshintala, and C. J. Rossbach, "AvA: Accelerated virtualization of accelerators," in *Proc. 25th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, New York, NY, USA, Association for Computing Machinery, 2020, pp. 807–825.
[47] H. Yu and C. J. Rossbach, "Full virtualization for GPUs reconsidered," in *Proc. Annu. Workshop Duplicating, Deconstructing, Debunking*, 2017.

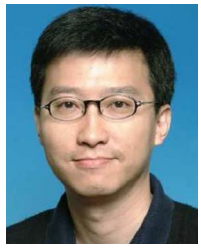
Xiaoyang Wang received the bachelor's degree in computer science and technology from USTC, in 2020. He is now working toward the PhD degree with the School of Computer Science and Technology, University of Science and Technology of China. His research interests include distributed operating system, especially memory management for various systems like GPU computing, RDMA remote memory and so on.

Yongkun Li received the BEng degree in computer science from USTC, in 2008, and the PhD degree in computer science and engineering from The Chinese University of Hong Kong, in 2012. He is currently an associate professor with the School of Computer Science and Technology, University of Science and Technology of China. After that, he worked as a postdoctoral fellow with the Institute of Network Coding, The Chinese University of Hong Kong. His research mainly focuses on memory and file systems, including key-value systems, distributed file systems, as well as memory and I/O support for different applications.

Fan Guo received the PhD degree from the University of Science and Technology of China, in 2019, and now he works with the VirtAI Technology Corporation. His main research interests are algorithm and system optimization for AI accelerator, virtualization, and operating system.



Yinlong Xu received the BS degree in mathematics from Peking University, in 1983, and the MS and PhD degrees in computer science from the University of Science and Technology of China (USTC), in 1989 and 2004, respectively. He is currently a professor with the School of Computer Science and Technology, USTC, and is leading a research group in doing some networking and high performance computing research. His research interests include network coding, storage systems, as well as design and analysis of parallel algorithms, etc.



John C. S. Lui (Fellow, IEEE) received the PhD degree in computer science from the University of California at Los Angeles. He is currently the Choh Ming Li chair professor with the Department of Computer Science and Engineering, The Chinese University of Hong Kong. His current research interests include machine learning, online learning, network science, future Internet architectures and protocols, network economics, network/system security, and large-scale storage systems. He is an elected member of the IFIP WG 7.3, a fellow of the ACM, and a senior research fellow of the Croucher Foundation.