DCUDA: Dynamic GPU Scheduling with Live Migration Support

Fan Guo
University of Science and Technology of China
lps56@mail.ustc.edu.cn

Yongkun Li
University of Science and Technology of China
ykli@ustc.edu.cn

John C. S. Lui
The Chinese University of Hong Kong
cslui@cse.cuhk.edu.hk

Yinlong Xu
University of Science and Technology of China
ylxu@ustc.edu.cn

ABSTRACT

In clouds and data centers, GPU servers which consist of multiple GPUs are widely deployed. Current state-of-the-art GPU scheduling algorithm are "static" in assigning applications to different GPUs. These algorithms 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 load balance and to 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 provides a realtime 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 codes. 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 applications we studied, DCUDA can reduce the average execution time of applications by up to 42.1%. Furthermore, DCUDA also reduces 13.3% energy in the light load scenario.

CCS CONCEPTS

• **Computer systems organization** → **Availability**.

KEYWORDS

GPU scheduling, live migration

ACM Reference Format:
Fan Guo, Yongkun Li, John C. S. Lui, and Yinlong Xu. 2019. DCUDA: Dynamic GPU Scheduling with Live Migration Support. In *ACM Symposium on Cloud Computing (SoCC '19), November 20–23, 2019, Santa Cruz, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3357223.3362714

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SoCC '19, November 20–23, 2019, Santa Cruz, CA, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6973-2/19/11…$15.00
https://doi.org/10.1145/3357223.3362714

1 INTRODUCTION

Graphics processing units (or GPUs) are a class of computing device 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 [8, 14, 22, 31]. With the rapid growth of GPUs' computing capacity, a GPU server usually contains many GPUs, each of which further consists of thousands of computing cores. However, many past studies demonstrated that the computing resources of GPUs are often under-utilized when running only a single application on each GPU [16, 24, 25]. 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 [2]) 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 GPU utilization. To further improve the efficiency of GPU sharing, various scheduling methods are proposed to distribute GPU resources between tenants or applications so as to balance the load between GPUs. Some well-known GPU scheduling methods are Round-robin scheduling [19], Least-Loaded scheduling [12, 26, 27], Prediction-based scheduling [30] 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 accounts 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. Moreover, the static scheduling methods cannot explore the potential of GPU sharing, which 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 [4]) collect the trace data of each function call by replaying APIs, and thus introduce a high performance penalty. 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 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, how to determine when and which applications should be migrated is also a challenge, especially how to reduce the number of migration times and the amount of migration data.

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

- We propose a *lightweight* monitoring method by intercepting API calls with wrapper libraries and tracking the utilization information from the function parameters. Comparing with current monitoring tools like nvprof, our monitoring scheme does not interrupt running applications, and incurs negligible overhead while also achieves higher than 90% of monitoring accuracy.
- We develop a live migration facility which is compatible to 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 propose a dynamic scheduling mechanism to guarantee load balance between GPUs by migrating some running applications from overloaded GPUs to underloaded GPUs. After doing load balancing, we also propose two optimization techniques to further reduce the energy consumption by compacting lightweight applications and improve the fairness with a priority-based time slicing policy.
- 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 all applications by up to 42.1%, and reduces the energy consumption of GPUs by up to 13.3%.

The rest of the paper is organized as follows. In Section 2, we introduce necessary background and analyze the limitations of current scheduling policy, then motivate the design of DCUDA. In Section 3, we present the design details of DCUDA. In Section 4, we describe the experiment setup and present the evaluation results. Section 5 reviews related work and Section 7 concludes the paper.

2 BACKGROUND AND MOTIVATION

2.1 GPU with Unified Memory

Traditionally, GPUs and CPUs have their own memory spaces, and applications running on one particular GPU cannot access the data directly from the memory of other GPUs or CPUs. To improve memory utilization, the latest NVIDIA PASCAL GPU released in 2016 supports unified memory [6], i.e., each GPU can access the whole memory space of both GPUs and CPUs via uniform memory addresses. In particular, the unified memory provides to all GPUs and CPUs a single memory address space, 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 migrate pages from anywhere in the system to the memory of GPUs in an on-demand manner.

The benefits of unified memory are twofold. First, it enables a GPU to handle dataset which is larger than its own memory size, because the unified memory can migrate data from CPU memory to GPU memory in an on-demand fashion. Second, using the unified memory can simplify the programming model. In particular, programmers can simply use a pointer to access data pages no matter where they reside, instead of explicitly calling data migration.

We point out that the implementation of DCUDA takes advantage of the unified memory architecture, that is, migrating data from source GPU to target GPU is performed in an *on-demand fashion*, i.e., data movement is triggered by page faults caused by accessing data on the target GPU with unified memory pointers.

2.2 Energy Management of GPUs

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 [15]. 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. 1, we find that running a single application on a GPU, which we call *single execution*, usually increases a lot of energy consumption (note that the GPU energy consumption in idle state is 441J), but running two applications concurrently, which we call *concurrent execution*, only increases the energy consumption a little compared with single execution. The main reason is that running a single application needs to wake up the GPU from the idle state and increases 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 on their performance (< 2%). Thus, compacting multiple lightweight applications to run on fewer GPUs in DCUDA can improve the overall energy efficiency.

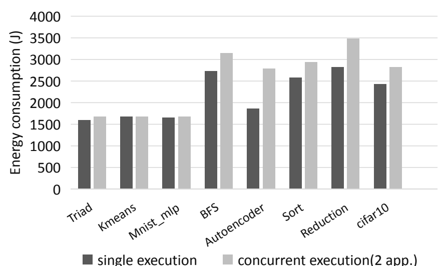


Figure 1: Energy consumption when run one application (~30sec) or concurrently run two applications on a GPU

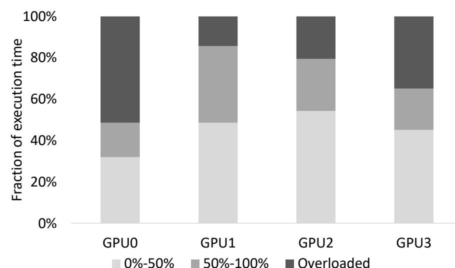


Figure 2: Load imbalance with Least-Loaded scheduling

2.3 Static Scheduling

In this paper, we focus on the scheduling in GPU sharing scenarios. In these scenarios, kernels from different applications can run concurrently on a GPU by performing spatial multiplexing on thousand of GPU cores.

To improve the fairness and effectiveness of GPU sharing, various GPU scheduling methods are proposed, such as Round-robin scheduling [19] and Least-Loaded scheduling [12, 26]. Round-robin scheduling uses round-robin strategy to choose GPUs to execute new applications. Least-Loaded scheduling assigns the GPU with the lightest load to run new applications. We point out that both methods are static scheduling methods, as they are responsible for only assigning new applications to different GPUs before running them, but they can not dynamically migrate running applications.

We find that load balance and energy efficiency could still be greatly improved in GPU sharing which uses 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 the 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 resources 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 length being smaller than execution time of the application (see Section 4.1 for details of the 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 being at overloaded state or underloaded state (i.e., 0-50% utilization). In fact, 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.

2.4 DCUDA with Dynamic Scheduling

To further improve load balance and alleviate resource contention between GPUs, we developed DCUDA, a runtime system which supports dynamic scheduling and live migration of running applications. We claim that our live migration method migrates applications between kernel invocations, and it cannot migrate a running kernel. Because current techniques cannot interrupt a running kernel and save its state during execution. The main challenges of DCUDA are as follows.

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 [3] and nvprof [4]. 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 still 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 virtual addresses of the data exactly the same with that in the source GPU. This requirement of preserving identical addresses is challenging as modern GPUs allocate virtual addresses in a stochastic manner. 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, because runtime environment and running state of applications are determined by many factors, which can not be obtained directly from CUDA. The third and the 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.

Dynamic scheduling. Dynamic scheduling is more efficient to achieve higher load balance and better GPU utilization. However,

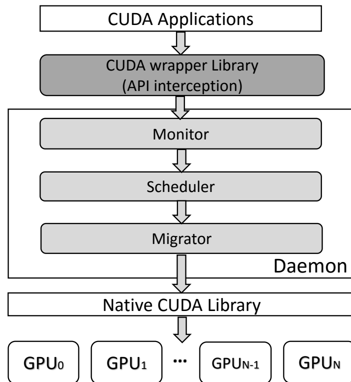


Figure 3: DCUDA Architecture

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.

3 DESIGN OF DCUDA

DCUDA is a CUDA-based GPU sharing platform which 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.

3.1 Overview of DCUDA

Similar to many other GPU sharing platforms, like vCUDA [28], rCUDA [13], GVim [17], and Pegasus [18], DCUDA adopts a frontend-backend architecture to ease the implementation while providing full compatibility to all CUDA applications. As shown in Fig. 3, 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 returning 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 realtime 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 so as 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. We will introduce these three modules in details in the following subsections.

3.2 The Monitor

The Monitor needs to track two kinds of information in real time, the utilization of each GPU and the resource demand of each application. Current monitoring tools (i.e., nvidia-smi and nvprof) introduce a large overhead as they replay API calls. To achieve both accuracy and low overhead, we provide a *lightweight* monitoring method by tracking information only from the parameters of intercepted APIs.

Monitoring applications' usage of GPU cores. Note that in CUDA applications, most computing tasks of an application are performed by kernel functions. Thus, we can obtain applications' demand 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 at the first time, and uses these 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 for many times. With this method, DCUDA can still accurately monitor the usage of GPU cores with a small overhead.

Monitoring applications' usage of GPU memory. A CUDA application allocates most of its needed GPU memory with APIs like cuMemAlloc(), so we obtain the allocated memory size and the range of virtual addresses from the parameters of the allocation APIs. 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 [7] and theano [8], 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 pointers really point to GPU memory or not with cuPointerGetAttribute(). However, checking all pointers introduces large overhead. We use a sampling method to reduce the monitor overhead. Specifically, we sample memory usage information with fixed step size (i.e., 64MB). For every 64MB region, we only check one page. If this page is being used, then we migrate the whole 64MB data via prefetching. Here we note that using a sampling method may cause false negative, i.e., some really used pages can not be identified and migrated using the prefetching operation. However, this false negative will not affect the correctness of running applications, because these omissive pages can still be migrated to the destination GPU via on-demand paging when page fault happens (see Sec 3.4).

Monitoring GPU utilization. Finally, to monitor the 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. Our experiment results validate that DCUDA brings very little overhead compared with traditional monitoring tool like nvprof, meanwhile, it achieves higher than 90% accuracy.

3.3 The Scheduler

Our first goal in designing DCUDA is to achieve better load balance so as to improve application performance, so the first task of the Scheduler is to determine when to perform load balancing and which applications to be migrated, and two key problems need to be addressed: (1) How to alleviate the "*ping-pong*" effect to avoid flip-flop migrations? (2) How to reduce the number of migrations when selecting candidate applications for migration? To address these problems, 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 and reduce the migration times as well as the number of migrated applications.

After doing load balancing, we further explore the opportunity of additional optimizations. We find that after load balancing, it is still possible for GPUs to be either overloaded with heavyweight applications, or underloaded with multiple lightweight applications. Thus, we try to address two problems: (1) How to achieve a good fairness and reduce resource contention, especially for the slightly overloaded GPUs? (2) How to reduce the energy consumption without sacrificing performance, especially for the underloaded GPUs? To address these problems, DCUDA proposes some techniques to take into consideration the energy awareness and fairness awareness after doing load balancing.

In summary, DCUDA employs a three-step process to schedule running applications on GPUs (see Fig. 4). First, in the load balancing step, DCUDA decides the candidate applications to be migrated from an overloaded GPU to an underloaded GPU so as to achieve dynamical load balance. After that, DCUDA leverages energy awareness by compacting several lightweight applications on underloaded GPUs to make them run on as few GPUs as possible. This can let more GPUs stay idle and can save a lot of energy without sacrificing the application performance. Finally, DCUDA also ensures the fairness of applications by adopting a priority-based time slicing policy to schedule the applications concurrently running on the same GPU. In the following, we introduce the details of each step.

Step 1: Load balancing operation. Note that the first key issue in scheduling is to balance the load between GPUs. Thus, we need to find out an overloaded GPU and an underloaded GPU, and then shift some workload 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 which causes frequent flip-flop migrations, DCUDA leverages hysteresis control to classify GPUs into three states with two threshold parameters, $Thresh_{over}$ and $Thresh_{under}$, according to

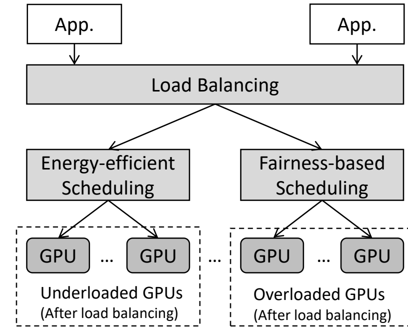


Figure 4: Scheduling flow of DCUDA

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. Furthermore, we also need to ensure that the migration will not make the underloaded GPU become overloaded immediately. To achieve the above goals, 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. We point out that DCUDA may migrate more than one application at one time by repeating the selection until no application satisfies the above condition or the source GPU becomes not overloaded any more. Finally, DCUDA adds all selected applications into a candidate list to wait for real migration. Note 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.

Step 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 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 doing load balancing, 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. DCUDA repeats the above steps until no GPU pair can be compacted. We like to emphasize that this energy-aware scheduling is performed 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.

Step 3: Fairness awareness. We point out that the load balancing operation can only reduce the variance of loads between GPUs, and it is still possible that some GPUs are slightly overloaded after load balancing. For an overloaded GPU, it may contain multiple applications which have very different resource demands on GPU cores. This may make the applications with low resource demand hard 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. To handle this issue, DCUDA uses a priority-based time slicing policy to guarantee the fairness.

First, DCUDA divides time into many slices (i.e., 100ms). It also 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. DCUDA also records the usage history in the past scheduling epoch. If any group overshoots its allocated time, it is penalized in subsequent epochs.

To decide which applications belonging to a group, the key principle is to make sure that the applications belonging to the same group should not cause a severe competition for the computing resource. In addition, they should also utilize the computing resource as much as possible as they are allowed to run concurrently on a GPU. To be more specific, the total resource demand of the applications belonging to the same group should be close to the total computing power of the GPU as much as possible. In DCUDA, we also allow the total demand to slightly exceed the computing power for the consideration of GPU utilization. For example, if one application requires 100% of computing resource and another application requires only 5%, then we will compact them into one group instead of two. Otherwise, the GPU utilization will be very low during the time slices in which the application with very low resource demand is running.

Note that allowing the resource demand of applications within a group to slightly exceed the computing power of GPU can greatly improve the GPU utilization, but it may make the GPU become overloaded when this group is scheduled to run, and thus it may still lead to the problem of unfair competition between applications within the same group. To handle this problem, we propose a priority-based reordering scheme which dynamically adjusts the priority of applications within the same group. In particular, we set higher priority to applications with low resource demand to

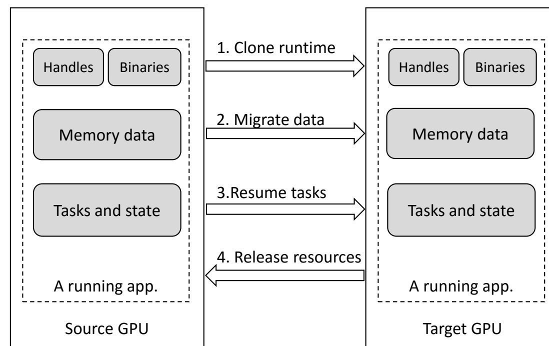


Figure 5: Process of live migration

increase their share of computing resource so as to guarantee a fair competition. We point out the above priority scheme does not cause significant slowdown on the performance of applications with high resource demand, the main reason is that even though applications with low resource demand has higher priority, they only need a little GPU resource to complete their computing tasks.

In the implementation, when an application's priority needs to be changed, DCUDA calls cuStreamCreateWithPriority() to create some new stream handles with the specified priority, and then uses these new stream handles to replace the old stream handles of the application.

3.4 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. 5, 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?

Cloning consistent runtime environment. By analyzing CUDA applications, we find that the runtime environment of an application includes kernel binaries, streams, and relevant 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

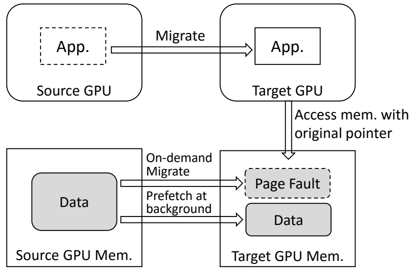


Figure 6: Data migration with unified memory

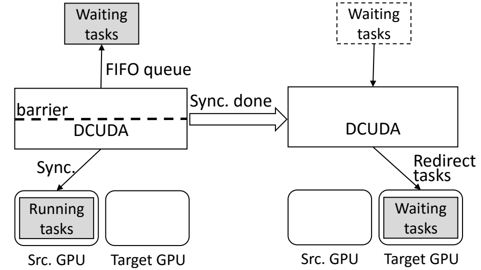


Figure 7: Procedure of migrating tasks

an application may be as high as 200ms - 400ms, 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. DCUDA triggers the registration of kernel binaries in an on-demand way when the kernel functions are really called.

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. 6. 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. We emphasize that by intercepting APIs to support unified memory, DCUDA can also significantly reduce the memory consumption of most applications, mainly because many applications may allocate a lot of GPU memory but only use a very small portion of it.

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. By using the on-demand migration policy with data prefetching, data migration and executing kernels can work well in pipeline, so the overhead of moving data is hidden in the computation time.

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 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. 7. 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 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.

4 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 [12], Rain [26], and Strings [27]. 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 4.2)?
- How much improvement can DCUDA achieve for load balance between GPUs (Section 4.3) and reduction of application execution time (Section 4.4)?
- How much improvement can DCUDA achieve for fairness and QoS between applications (Section 4.5)?
- What is the impact of different load levels on the performance and the energy saving of DCUDA (Section 4.6)?

4.1 Setup

We conducted our experiments on a server with two Intel Xeon E5-2620 v4 2.10GHz processors, 64GB system memory, and four NVIDIA 1080Ti GPUs, which are based on the PASCAL architecture and interconnected with PCIe. Each GPU has 3584 computing cores and 12GB memory.

We select twenty distinct benchmark programs taken from the CUDA Samples [1], SHOC [11], and Tensorflow Benchmarks [5]. Table 1 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) [1, 5, 11].

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

Table 1: Benchmarks

Since we focus on the scenario of GPU sharing which naturally requires multiple applications concurrently run on a GPU server, 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 5*s* by default in our experiment. Note that the interval is smaller than the executing time of a benchmark program (around 30*s*) so as to simulate a medium-weight workload. We also study the impact of different load levels by adjusting the length of the arrival interval (see Section 4.6). Note that with the 20 benchmark applications, we can have 20! ($2.43e^{18}$) application sequences by adjusting the arrival order of each application, and each sequence can represent a particular workload. We select 50 random arrival sequences to evaluate DCUDA. We set the threshold $Thresh_{over}$ as 100%, 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 $Thresh_{under}$ as 90% so as to achieve high GPU utilization. At last, we set the monitoring interval as 100*ms*.

4.2 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 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 4.3).

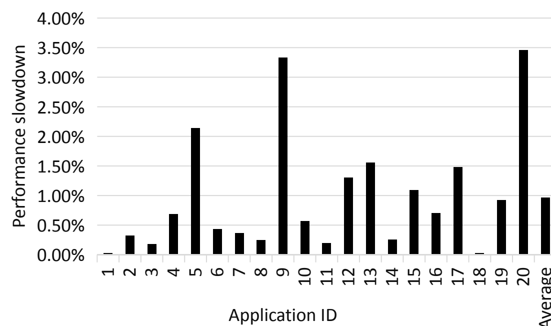


Figure 8: Performance loss caused by unified memory

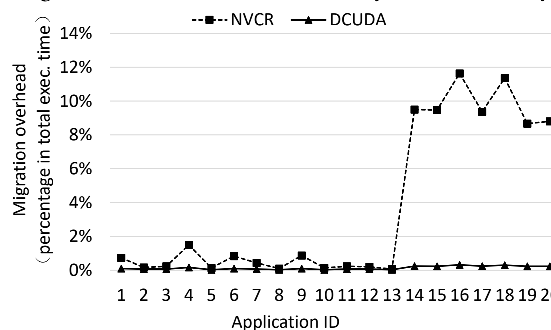
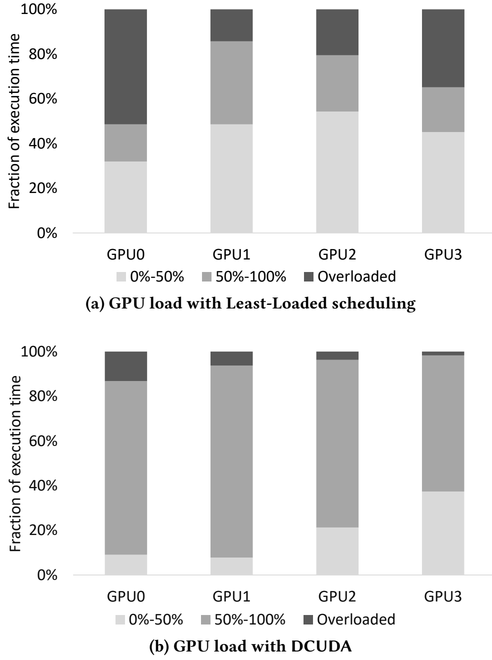


Figure 9: Reduction of migration overhead with DCUDA

Overhead of unified memory. Next, we also evaluate the performance loss caused by the support of unified memory in all scheduling applications. As shown in Fig. 8, 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 memory can be negligible, compared with the improvement brought by DCUDA (see Section 4.4).

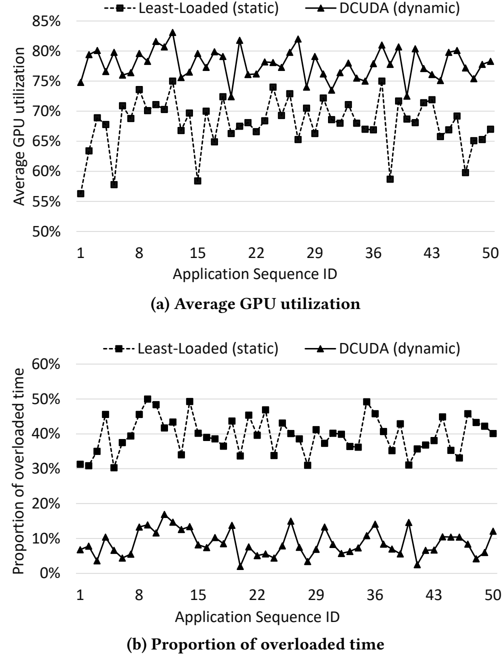
Overhead of live migration. We further evaluate the time overhead of the live migration process with DCUDA, and compare it with NVCR [23], which is the state-of-the-art live migration approach. The migration overhead is measured as the percentage of the time for one single migration to the total execution time of each application. The results are shown in Fig. 9, we can see that the migration overhead of NVCR is *very high*, which accounts for up to 11.6% of total execution time. In contrast, the overhead of the live migration in DCUDA is very small, and precisely, it takes 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



(a) GPU load with Least-Loaded scheduling

(b) GPU load with DCUDA

Figure 10: Load of each GPU under one app. sequence



(a) Average GPU utilization

(b) Proportion of overloaded time

Figure 11: GPU utilization under different app. sequences

compared with NVCR. This is because that 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.

4.3 Improvement of Load Balance

The main benefit of DCUDA is to balance the load between GPUs via dynamic scheduling, so we first evaluate the improvement of DCUDA in load balancing, and compare it with the Least-Loaded scheduling scheme. We classify each GPU into three states 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. 10, and the results are similar for other sequences.

From Fig. 10(a), we find that when using the Least-Loaded scheduling, GPUs are very likely to become overloaded, e.g., the overloaded time of each GPU accounts for 14.3% - 51.4% of the whole running time. The main reason is that when assigning a new application with high resource demand, there is often no GPU which has enough resources to run this application as all GPUs may have been assigned some applications before. Even though the Least-Loaded scheduling chooses the most underloaded GPU, it makes the GPU overloaded immediately after assigning the new heavy-weight application to it because of the lack of rescheduling running applications. Moreover, the utilization of other GPUs may be very low (< 50%) for a long time even some GPUs are already overloaded. The reason is that even some applications on these GPUs have completed, applications running on other overloaded GPUs could not be migrated to the underloaded GPUs to realize realtime load balance.

As shown in Fig. 10(b), even 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 situation occurs. This live migration not only reduces the overloaded time duration, but also reduces the underloaded time duration by taking advantage of the computing resources of underloaded GPUs.

We further evaluate the performance of DCUDA in load balancing under all 50 application sequences. Fig. 11(a) shows the average GPU utilization and Fig. 11(b) shows the proportion of overloaded time. We can see that DCUDA can achieve a large improvement in

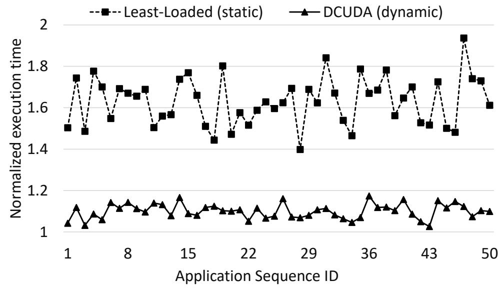


Figure 12: Average execution time of applications under different application sequences

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.

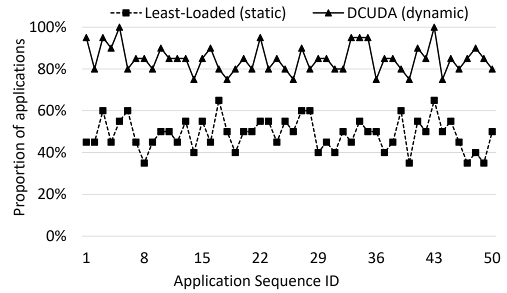
4.4 Reduction of Execution Time

In this section, we evaluate the benefit that comes from the improvement of load balance by comparing the average execution time of applications under different application sequences. We normalize the execution time of applications to their single execution. The results are shown in Fig. 12. DCUDA reduces 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. Furthermore, DCUDA can also guarantee the performance of lightweight applications due to the use of priority-based policy.

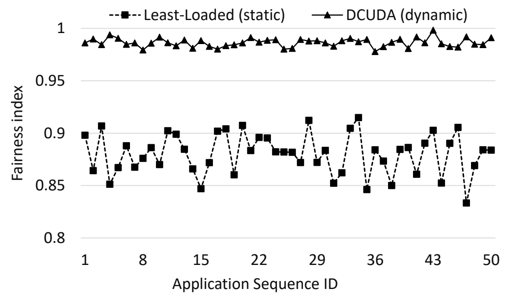
4.5 Improvement of QoS and Fairness

Besides the execution time, 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. 13(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%. In contrast, the Least-Loaded scheduling only has less than 60% of the applications satisfying the QoS requirement on average, and the largest proportion is 65%. We also run more experiments by varying the allowable degradation threshold instead of using 20%, and we also observe significant improvement with DCUDA.

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 [20], 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



(a) Proportion of applications which satisfy QoS requirement



(b) Fairness between applications with/without DCUDA

Figure 13: QoS and fairness

sequences and show the results in Fig. 13(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.

4.6 Performance under Different Loads

Note that DCUDA reduces the execution time of applications by balancing the 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. 14. 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, we emphasize that DCUDA still achieves a large improvement in a wide range of load levels, and in practical systems, the scenario of load imbalance is usually very common because of the static feature of existing scheduling schemes, so DCUDA could achieve a large improvement.

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 2. We

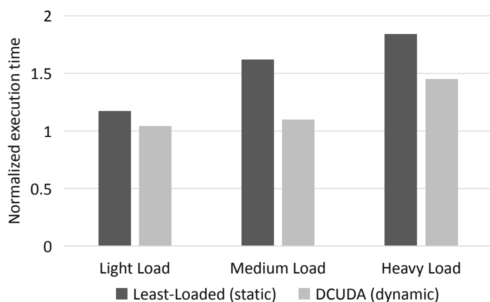


Figure 14: Reduction of the execution time of applications with DCUDA under different loads

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

Table 2: Reduction of the energy consumption of applications with DCUDA under different loads

can see that DCUDA can save more energy if the system load 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.

5 RELATED WORK

GPU scheduling. To improve GPU utilization in shared environment, multiple scheduling schemes are proposed. Some of them focus on the scheduling of applications on a single GPU, such as Baymax [10], FLEP [32], EffiSha [9] and TimeGraph [21]. But these works do not consider the scheduling between multiple GPUs. Additionally, many others scheduling schemes are proposed to schedule applications between multiple GPUs, including Round-Robin scheduling [19], Least-Loaded scheduling [12, 26, 27], and Prediction-based scheduling [30]. In particular, Least-Loaded scheduling, which always assigns new applications to the GPU with the least load, has been widely used in practical systems due to its superiority. For example, in gCloud [12], Khaled et al. find that Least-Loaded scheduling can improve the performance of applications by 10.3% than Round-Robin policy. Sengupta et al. [26] 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. Recently, Yash et al. [30] 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. Different from existing schemes, which can be classified as static scheduling as they consider only the assignment of new applications, DCUDA provides dynamic scheduling of running applications via live migration.

Live migration. Live migration is an important feature in GPU sharing systems, which has been widely used for fault tolerance and load balance. In particular, Xiao et al. [33] propose a live migration approach for OpenCL applications, while it is not applicable to CUDA applications. Takizawa et al. first propose CheCUDA [29] 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 [23] 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. But the replaying method provided by NVCR cannot work in a GPU sharing environment. Because in these scenarios, some virtual address range may be occupied by other applications. DCUDA differs from the above works by providing a universal live migration approach to transparently support all CUDA applications. Besides, DCUDA greatly improves the load balance between GPUs with negligible migration overhead, with the help of migrating data in an on-demand fashion, as well as the optimization techniques like handle pooling and data prefetching.

6 DISCUSSION AND FUTURE WORK

DCUDA also has some limitations. First, 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. Second, DCUDA only considers the contention of GPU cores in the design of scheduling policy. We will consider multiple shared resources in our future work, including memory bandwidth and shared caches within the GPU. Last but not least, current version of DCUDA only supports APIs in CUDA toolkit 8.0. We will add the supports of more APIs in the newest CUDA library. We emphasize that this work will take very little effort due to the low update frequency of CUDA libraries.

7 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.

ACKNOWLEDGMENTS

We thank the anonymous reviewers and our shepherd, Jayneel Gandhi, for their valuable comments and suggestions. Yongkun Li is USTC Tang Scholar, and he is the corresponding author. This work was supported in part by National Key R&D Program of China 2018YFB1003204, NSFC 61772484, and Youth Innovation Promotion Association CAS. The work of John C.S. Lui was supported in part by GRF 14200117.

REFERENCES

[1] 2019. CUDA samples. http://docs.nvidia.com/cuda/cuda-samples/index.html.
[2] 2019. CUDA Toolkit. https://developer.nvidia.com/cuda-toolkit.
[3] 2019. nvdia-smi. https://developer.nvidia.com/nvidia-system-management-interface.
[4] 2019. nvprof. http://docs.nvidia.com/cuda/profiler-users-guide/index.html.
[5] 2019. Tensorflow benchmarks. https://github.com/tensorflow/benchmarks.
[6] 2019. Unified Memory on Pascal. https://devblogs.nvidia.com/beyond-gpu-memory-limits-unified-memory-pascal/.
[7] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. TensorFlow: A System for Large-Scale Machine Learning.. In *OSDI*, Vol. 16. 265–283.
[8] James Bergstra, Frédéric Bastien, Olivier Breuleux, Lamblin, et al. 2011. Theano: Deep learning on gpus with python. In *NIPS 2011, BigLearning Workshop*. Citeseer.
[9] Guoyang Chen, Yue Zhao, Xipeng Shen, and Huiyang Zhou. 2017. EffiSha: A Software Framework for Enabling Effficient Preemptive Scheduling of GPU. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 3–16.
[10] Quan Chen, Hailong Yang, Jason Mars, and Lingjia Tang. 2016. Baymax: QoS Awareness and Increased Utilization for Non-Preemptive Accelerators in Warehouse Scale Computers. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 681–696.
[11] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S Meredith, Philip C Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S Vetter. 2010. The scalable heterogeneous computing (SHOC) benchmark suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*. ACM, 63–74.
[12] Khaled M Diab, M Mustafa Rafique, and Mohamed Hefeeda. 2013. Dynamic sharing of GPUs in cloud systems. In *IEEE 27th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW)*. IEEE, 947–954.
[13] José Duato, Antonio J Pena, Federico Silla, Rafael Mayo, and Enrique S Quintana-Ortí. 2010. rCUDA: Reducing the number of GPU-based accelerators in high performance clusters. In *2010 International Conference on High Performance Computing and Simulation (HPCS)*. IEEE, 224–231.
[14] Wenbin Fang, Ka Keung Lau, Mian Lu, Xiangye Xiao, Chi K Lam, Philip Yang Yang, Bingsheng He, Qiong Luo, Pedro V Sander, and Ke Yang. 2008. Parallel data mining on graphics processors. *Hong Kong Univ. Sci. and Technology, Hong Kong, China, Tech. Rep. HKUST-CS08-07* (2008).
[15] Mariza Ferro, André Yokoyama, Vinicius Klôh, Gabrieli Silva, Rodrigo Gandra, Ricardo Bragança, Andre Bulcao, Bruno Schulze, and Petróleo Brasileiro SA-PETROBRAS. 2017. Analysis of gpu power consumption using internal sensors. In *Anais do XVI Workshop em Desempenho de Sistemas Computacionais e de Comunicaç ao, Sao Paulo-SP. Sociedade Brasileira de Computaçao (SBC)*.
[16] Chris Gregg, Jonathan Dorn, Kim M Hazelwood, and Kevin Skadron. 2012. Fine-Grained Resource Sharing for Concurrent GPGPU Kernels.. In *HotPar*.
[17] Vishakha Gupta, Ada Gavrilovska, Karsten Schwan, Harshvardhan Kharche, Niraj Tolia, Vanish Talwar, and Parthasarathy Ranganathan. 2009. GViM: GPU-accelerated virtual machines. In *Proceedings of the 3rd ACM Workshop on System-level Virtualization for High Performance Computing*. ACM, 17–24.
[18] Vishakha Gupta, Karsten Schwan, Niraj Tolia, Vanish Talwar, and Parthasarathy Ranganathan. 2011. Pegasus: Coordinated scheduling for virtualized accelerator-based systems. In *2011 USENIX Annual Technical Conference (USENIX ATC'11)*. 31.
[19] Everton Hermann, Bruno Raffin, François Faure, Thierry Gautier, and Jérémie Allard. 2010. Multi-GPU and multi-CPU parallelization for interactive physics simulations. In *European Conference on Parallel Processing*. Springer.
[20] Raj Jain, Dah-Ming Chiu, and William Hawe. 1984. A Quantitative Measure of Fairness and Discrimination for Resource Allocation in Shared Systems, Digital Equipment Corporation. *Technical Report DEC-TR-301* (1984).
[21] Shinpei Kato, Karthik Lakshmanan, Ragunathan Raj Rajkumar, and Yutaka Ishikawa. 2011. TimeGraph: GPU Scheduling for Real-Time Multi-Tasking Environments. In *2011 USENIX Annual Technical Conference (USENIX ATC)*. Citeseer, 17.
[22] Naoya Maruyama, Tatsuo Nomura, Kento Sato, and Satoshi Matsuoka. 2011. Physis: an implicitly parallel programming model for stencil computations on large-scale GPU-accelerated supercomputers. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM.
[23] Akira Nukada, Hiroyuki Takizawa, and Satoshi Matsuoka. 2011. NVCR: A transparent checkpoint-restart library for NVIDIA CUDA. In *IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW)*. IEEE, 104–113.
[24] Sreepathi Pai, Matthew J Thazhuthaveetil, and Ramaswamy Govindarajan. 2013. Improving GPGPU concurrency with elastic kernels. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 407–418.
[25] Jason Jong Kyu Park, Yongjun Park, and Scott Mahlke. 2015. Chimera: Collaborative preemption for multitasking on a shared GPU. *ACM SIGARCH Computer Architecture News* 43, 1 (2015), 593–606.
[26] Dipanjan Sengupta, Raghavendra Belapure, and Karsten Schwan. 2013. Multi-tenancy on GPGPU-based servers. In *Proceedings of the 7th international workshop on virtualization technologies in distributed computing*. ACM, 3–10.
[27] Dipanjan Sengupta, Anshuman Goswami, Karsten Schwan, and Krishna Pallavi. 2014. Scheduling multi-tenant cloud workloads on accelerator-based systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 513–524.
[28] Lin Shi, Hao Chen, Jianhua Sun, and Kenli Li. 2012. vCUDA: GPU-accelerated high-performance computing in virtual machines. *IEEE Trans. Comput.* 61, 6 (2012), 804–816.
[29] Hiroyuki Takizawa, Katsuto Sato, Kazuhiko Komatsu, and Hiroaki Kobayashi. 2009. CheCUDA: A checkpoint/restart tool for CUDA applications. In *International Conference on Parallel and Distributed Computing, Applications and Technologies*. IEEE, 408–413.
[30] Yash Ukidave, Xiangyu Li, and David Kaeli. 2016. Mystic: Predictive scheduling for gpu based cloud servers using machine learning. In *2016 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 353–362.
[31] Matthias Vogelgesang, Suren Chilingaryan, Tomy dos_Santos Rolo, and Andreas Kopmann. 2012. UFO: A scalable GPU-based image processing framework for on-line monitoring. In *IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS) & IEEE 14th International Conference on High Performance Computing and Communication*. IEEE.
[32] Bo Wu, Xu Liu, Xiaobo Zhou, and Changjun Jiang. 2017. FLEP: Enabling Flexible and Efficient Preemption on GPUs. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 483–496.
[33] Shucai Xiao, Pavan Balaji, James Dinan, Qian Zhu, Rajeev Thakur, Susan Coghlan, Heshan Lin, Gaojin Wen, Jue Hong, and Wu-chun Feng. 2012. Transparent accelerator migration in a virtualized GPU environment. In *Proceedings of the 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*. IEEE Computer Society, 124–131.