Toward QoS-Awareness and Improved Utilization of Spatial Multitasking GPUs

Wei Zhang, Quan Chen, Ningxin Zheng, Weihao Cui, Kaihua Fu, and Minyi Guo, *Fellow, IEEE*

Abstract—Datacenters use GPUs to provide the significant computing throughput required by emerging user-facing services. The diurnal user access pattern of user-facing services provides a strong incentive to co-located applications for better GPU utilization, and prior work has focused on enabling co-location on multicore processors and traditional non-preemptive GPUs. However, current GPUs are evolving towards spatial multitasking and introduce a new set of challenges to eliminate QoS violations. To address this open problem, we explore the underlying causes of QoS violation on spatial multitasking GPUs. In response to these causes, we propose C-Laius, a runtime system that carefully allocates the computation resource to co-located applications for maximizing the throughput of batch applications while guaranteeing the required QoS of user-facing services. C-Laius not only allows co-locating one user-facing application with multiple batch applications, but also supports the co-location of multiple user-facing applications with batch applications. In the case of a single co-located user-facing application, our evaluation on an Nvidia RTX 2080Ti GPU shows that C-Laius improves the utilization of spatial multitasking GPUs by 20.8 percent, while achieving the 99%-ile latency target for user-facing services. As to the case of multiple co-located user-facing applications, C-Laius ensures no violation of QoS while improving the accelerator utilization by 35.9 percent on average.

Index Terms—Spatial multitasking GPUs, QoS, improved utilization

1 Introduction

Datacenters often host user-facing applications (e.g., web search [1], web service [1], memcached) that have stringent latency requirements. It is crucial to guarantee that the queries' end-to-end latencies are shorter than a predefined Quality-of-Service (QoS) target, which is generally from 100 ms to 300 ms. With the quick advance of machine learning technology, emerging user-facing applications, such as Apple Siri [2] and Google Translate [3], start to use machine learning technologies (e.g., Deep Neural Network) that are often computational demanding. Datacenters have adopted accelerators to run these services so that they can achieve the required latency target [4]. As prior work states, user-facing applications experience diurnal user access patterns (leaving the accelerator resources underutilized for most of the time except peak hours) [5], [6]. The diurnal pattern provides a strong incentive to co-locate user-facing services with batch applications that do not have QoS requirements to improve utilization when the query load is low.

Accelerator manufacturers are now producing spatial multitasking GPUs for higher aggregated throughput for co-location applications [7], [8], [9]. For instance, the latest Nvidia Volta and Turing architectures allow kernels to share certain portions of computational resources simultaneously. Leveraging the new generation Multi-Process Service (MPS) [10], it is possible to allocate a small percentage of computational resources (active threads) to global memory bandwidth-bound applications and use most computational resources to speed up the execution of the co-located compute-intensive applications.

Improving utilization while guaranteeing QoS of user-facing services at low load has been resolved for both CPU servers and traditional GPU-outfitted servers [11], [12], [13], [14], [15]. For CPU co-location, prior works fall into two scenarios: 1. The co-location is limited to at most one user-facing service per physical host, co-scheduled with one or more batch jobs; 2. The co-location supports multiple user-facing jobs with batch jobs on the same physical node.

For the first co-location situation, prior works include two categories: profiling-based and feedback-based. The profiling-based method, such as Bubble-Up [11], profiles user-facing services and batch applications offline to predict their performance degradation at co-location due to shared cache and memory bandwidth contention, and identifies the "safe" co-locations that do not result in QoS violation. The feedback-based method, such as Heracles [13], builds models to determine shared resource allocation in the next time period according to the QoS feedback of user-facing services in the current period. These studies assume that all the user-facing queries have similar workloads and each query is processed by a single thread [11], [12], [13], [16], [17]. For the second co-location situation, PARTIES [18], a recent work, is the only work that aims to co-locate multiple user-facing jobs with batch jobs. PARTIES leverages a set of hardware and software resource partitioning mechanisms to adjust allocations dynamically at runtime.

• *The authors are with the Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai 200240, China. E-mail: {zhang-w, chen-quan, ningxinzheng, weihao, midway}@sjtu.edu.cn, guo-my@cs.sjtu.edu.cn.*

Manuscript received 21 Mar. 2020; revised 7 Feb. 2021; accepted 28 Feb. 2021. Date of publication 8 Mar. 2021; date of current version 11 Mar. 2022. (Corresponding authors: Quan Chen and Minyi Guo.) Recommended for acceptance by J. Shalf. Digital Object Identifier no. 10.1109/TC.2021.3064352

On CPUs, applications contend for the cores, shared cache, and the main memory bandwidth. On the other hand, our investigation shows that the latency of a query on a spatial multitasking GPU is impacted by *the percentage of computational resources allocated to its kernels*, *the scalability of the kernels*, and *the contention on shared resources* (e.g.SMs, global memory bandwidth, and PCIe bandwidth.) Therefore, prior work is not applicable for spatial multitasking GPUs, because the latency of a user-facing query at co-location on this hardware is impacted by different factors.

For application co-location on traditional GPUs, where kernels from the co-located applications queued up for processing elements, queuing-based methods (e.g., Baymax [14]) are proposed to eliminate QoS violation due to the long queuing time. Baymax predicts every GPU task's duration, reserves time slots for each user-facing query, and uses the remaining slots to process batch applications. On the contrary, *on spatial multitasking GPUs*, kernels share processing elements spatially. The queuing-based method does not apply for spatial multitasking GPUs. Meanwhile, these works also have been limited to co-locating at most one user-facing job with one or more background jobs on traditional GPUs.

It is challenging to determine the number of computational resources allocated to each task of a user-facing query so that its QoS can be satisfied while maximizing resource utilization on spatial multitasking GPUs. Since how kernels overlap with each other is only known at runtime, an online methodology is required to eliminate QoS violation caused by contention on shared resources. To this end, we propose *C-Laius*, a runtime system that is comprised of a *task performance predictor*, a *contention-aware resource allocator*, and a *progress-aware lag compensator*. When a user-facing query is submitted, for each of its tasks k, the task performance predictor predicts k's duration and global memory bandwidth usage under various computational resources. Based on the prediction, the resource allocator assigns the query "just-enough" resource to satisfy its QoS. When C-Laius assigns the remaining resources to batch applications, contention-aware resource allocator limits the global memory bandwidth usage of the batch kernels to eliminate QoS violation due to global memory bandwidth contention. Suppose the query runs slower than expected due to the contention on other shared resources. In that case, the progress-aware lag compensator allocates more resources to the unexecuted kernels of the query to enforce its QoS. *C-Laius* that we proposed above is suitable for the fixed and conservative situations where the effort has been limited to co-locating at most one user-facing application with batch applications on spatial multitasking GPUs. In this way, we have expanded the applicable scenarios of C-Laius to multiple QoS services (services with QoS requirements)and proposed a new, priority-based resource allocation strategy. When multiple user-facing queries are submitted, the multiple QoS tasks scheduler identifies their tasks' priority based on prediction and progress monitor. Then C-Laius adjusts the resources quota to each task according to their resource sensitivity. In this work, we rely on the bandwidth reservation technique proposed in Baymax [14] to ensure the data transmission at full speed for each user-facing query, thus eliminating QoS violation which is resulted from PCIe bandwidth contention.

The main contributions of this paper are as follows.

- *Comprehensive analysis of QoS interference on spatial multitasking GPUs* - We identify key factors that impact the end-to-end latency of a user-facing query. The analysis motivates us to design a resource management methodology for ensuring QoS while maximizing utilization.
- *Enabling dynamic resource reallocation for spatial multitasking GPUs* - We propose the process pool to adjust resource allocation between co-located applications at runtime, while the native MPS does not support resource reallocation during the execution of an application.
- *Designing an online progress monitor for identifying potential QoS violation* - If a query runs slower than expected so that it cannot meet the QoS target, C-Laius allocates it with more computational resources to compensate for the lag.
- *New techniques to determine resource allocation among multiple QoS tasks* - C-Laius takes a flexible resource allocation strategy based on priority to explore multiple resource dimensions and tasks simultaneously to (1) effectively co-locate multiple user-facing and background applications, and (2) extract hidden opportunities for improving efficiency by exploiting differences among tasks toward resource sensitivity.

Our experiment on an Nvidia RTX 2080Ti GPU shows that C-Laius can improve the throughput of batch applications by 20.8 percent compared with state-of-the-art solution Baymax [14], while guaranteeing the 99%-ile latency of user-facing services. As to the case of multiple co-located QoS applications, C-Laius ensures no violation of QoS while improving the accelerator utilization by 35.9 percent on average.

2 Related Work

There has been a large amount of prior work aiming to improve resource utilization while guaranteeing QoS of user-facing applications for CPU co-location [11], [12], [16], [19]. Bubble-Up [11] and Bubble-Flux [12] identify "safe" co-locations that bound performance degradation while improving chip multiprocessor utilization. PARTIES [18] proposes a simple solution towards meeting the latency targets of multiple co-located user-facing services. It tries to incrementally increase or decrease one resource (e.g., number of cores, memory capacity, etc.) for one QoS task at a time, and assesses the observed performance. However, they are not applicable for spatial multitasking GPUs because the latency of a user-facing query at co-location on spatial multitasking GPUs is impacted by different factors.

For co-location on accelerators, Multi-Process Service scheduling [10] enables multiple applications to share one GPU concurrently. GPU Maestro [9] aims at increasing GPU utilization on GPGPU-Sim rather than real in-production GPUs. TimeGraph [20] and GPUSync [21] use priority-based scheduling to guarantee the performance of real-time kernels. High priority kernels are executed first if multiple kernels are launched to the same GPU. GPU-EvR [22]

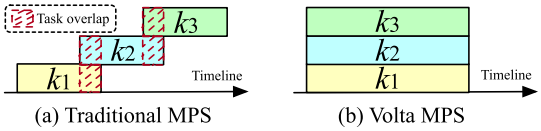


Fig. 1. Comparison of the original MPS and Volta MPS.

launches different applications to different streaming multiprocessors (SMs) on one GPU. Baymax [14] predicts the kernel duration and reorders the kernel based on the QoS headroom of user-facing queries. However, they assume that the accelerator is time-sharing and non-preemptive. The time-sharing assumption results in low resource utilization compared with C-Laius. KSM [23] predicts the slowdown of co-located applications on spatial multitasking GPUs. However, it relies on a broad spectrum of performance event statistics that are not available on real system GPUs. And KSM cannot identify the "just-enough" computational resource quotas for user-facing queries as C-Laius does.

3 Background and Motivation

3.1 Spatial Multitasking GPUs

A GPU often has multiple Streaming Multiprocessors that share the global memory. For instance, Nvidia RTX 2080Ti, a GPU of Turing architecture, has 68 SMs that can run 1,024 active threads (i.e., computational resources) concurrently. The SMs share a 12 GB global memory.

Since a single kernel may not be able to utilize all the SMs and other on-chip resources all the time [14], [24], [25], [26]. Starting from Kepler architecture [27], Nvidia proposed Multi-Process Service technique [10] to enable concurrent execution of kernels from different processes on the same GPU. If a kernel cannot occupy all the SMs, kernels from the co-located applications will use the remaining SMs. Compared with the traditional solution that executes kernels sequentially, MPS improves resource utilization, overall GPU throughput and energy efficiency [10], [14].

The traditional MPS technique does not provide an interface to control how different kernels' thread blocks (TBs) are dispatched into SMs. Only when a kernel cannot use all computational resources, the remaining threads are allocated to run other kernels. In this case, as shown in Fig. 1a, kernels are still executed sequentially with little concurrent execution. In this scenario, the on-chip shared memory and L1 cache are still underutilized in most cases.

To improve GPU utilization, Volta and Turing architectures introduce Volta MPS with new capabilities that allow multiple applications to share GPU resources simultaneously [10]. As shown in Fig. 1b, the new Volta MPS technique enables explicit GPU computational resource allocation, where the spatial multitasking is possible.

3.2 Investigation Setup

We use Nvidia RTX 2080Ti as the experimental platform to perform the investigation. Because our study does not rely on any specific feature of 2080Ti, it applies to other spatial multitasking GPUs. In this experiment, we co-locate user-facing services with batch applications and schedule them with existing GPU resource management techniques. We use applications in a DNN service benchmark suite, Tonic suite [28], as user-facing services, and use benchmarks in Rodinia [29] as batch applications. More details of the experimental hardware and benchmarks are described in Section 9.

3.3 Problem of QoS Violation

Fig. 2 shows the QoS violation of user-facing services at co-location adopting the new Volta MPS technique [30]. Adopting Volta MPS, when n applications are co-located, we allocate computational resources with two policies: *even allocation* and *priority allocation*. With even allocation, each application is configured to use $200\%/n$ of the computational resources following the recommendation of Nvidia [30]. With priority allocation, the user-facing service is allocated 100 percent of the computational resources for QoS requirement while each of the rest $n-1$ applications is allocated $100\%/(n-1)$ of the computational resources.

In Fig. 2, the x-axis shows the co-location pairs while the y-axis presents the 99%-ile latency of the user-facing service normalized to the QoS target. For example, *dig +BFS* presents the normalized 99%-ile latency of the user-facing service *dig* when co-located with the batch application *BFS*. As shown in the figure, different batch applications cause varying amounts of performance degradation to the co-located user-facing services. User-facing services in 28 and 13 out of 48 co-locations pairs suffer from QoS violation with even allocation and priority allocation respectively. The serious QoS violation is mainly due to the limited computational resources allocated to user-facing queries and the shared resource contention. Even if 100 percent of computational resources are allocated to a user-facing service with priority allocation, its tasks may still run concurrently with kernels of batch applications when its tasks do not have enough warps (Warp is the basic execution unit of SM). In this case, the concurrent data accessed from global

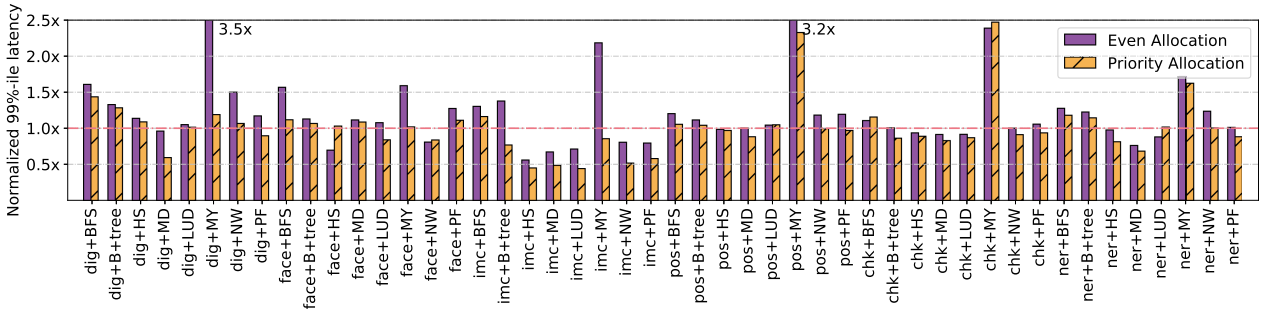


Fig. 2. QoS violation of user-facing applications with Volta MPS that adopts even allocation and priority allocation.

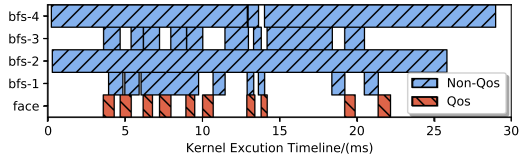


Fig. 3. Execution trace of user-facing service *face* and batch applications *BFS* on a spatial multitasking GPU.

memory and shared memory degrades user-facing queries' performance, which results in QoS violation.

3.4 Factors That Affect the End-to-End Latency

To understand the QoS violation problem at co-location, Fig. 3 presents a real system execution trace of a user-facing service *face* and four batch applications *bfs* on a spatial multitasking GPU. We can see that the tasks from different applications run concurrently. Analyzing from Fig. 3, there are three key factors affecting the end-to-end latency of a user-facing query (q) at co-location.

(1) The Computational Resource Percentage Allocated to q. If the percentage is too small, there are not enough active threads to process its tasks, resulting in its long latency. On the contrary, if too many computational resources are allocated to q, batch applications would suffer from the low throughput.

(2) The Scalability of Every Task in q. A user-facing query often has multiple tasks. The task's scalability determines if it can speed up when more computational resources are allocated to the task. Allocating a large percentage of computational resources to a non-scalable task would not reduce the latency of q.

(3) The Contention on Shared Resources. Because tasks from different applications may run on the same SM [30], concurrent tasks may contend for both shared memory within the SM and global memory bandwidth. The contention may seriously degrade the performance of co-located applications.

3.5 Challenges for Resource Allocation on Spatial Multitasking GPUs

Our real system investigation has shown that three factors affect the latency of user-facing queries at co-location. However, identifying the appropriate percentage of computational resources allocated to a user-facing query is non-trivial due to the complex interference behaviors on spatial multitasking GPUs. Specifically, there are several key challenges to maximize the throughput of batch applications while guaranteeing the QoS of user-facing services.

(1) The Workload of User-Facing Queries Varies - Since users often submit queries with different workloads, the computational resource percentage needed to complete a query within the QoS target varies. There is no optimally static resource percentage for a user-facing service.

(2) The Performance Degradation Varies Due to the Shared Resource Contention - The performance degradation of a user-facing query depends on how the tasks overlap with each other during runtime. Because tasks have different pressures on the shared resources, a user-facing query may suffer from different performance degradations even if it is co-located with the same batch applications.

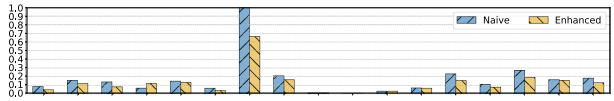


Fig. 4. The throughput of user-facing services normalized with naive and enhanced.

(3) An Approach is Required to Enable Resource Reallocation at Runtime - The percentage of computational resources allocated to an application is not configurable during its lifetime in emerging spatial multitasking GPUs. However, a user-facing query may suffer from QoS violation with the current computational resources due to shared resource contention. Therefore, an approach is required to allocate a larger percentage of computational resources to a query during its execution.

3.6 Co-Locating Multiple User-Facing and Batch Applications Together

For the scenario of multiple user-facing applications, we combine 6 different user-facing applications from Tonic Suite in pairs to form 15 sets of multi-QoS combinations. And we choose one fixed non-QoS application from Rodinia as the batch application.

We first perform a set of baseline experiments without scheduling. As a result, QoS violations existed in all 15 groups in the experiment. In this way, we identify a set of solutions based on static optimization through offline profiling. The static optimization here refers to enumerating all possible percentage situations, and running the program with different percentage configuration situations. Then we identify the least number of QoS violations for all percentage configurations. It can be seen from the Fig. 21 that the optimal set of solutions can guarantee that there are no QoS violations in 7 of the 15 groups. Comparing the GPU throughput of the static optimal solution to the baseline, we can see from the Fig. 4 that the throughput has declined a little in the static optimal case since the percentage of batch applications becomes zero.

4 C-Laius Methodology

In this section, we present C-Laius, which maximizes the throughput of batch applications while guaranteeing the QoS of user-facing services on spatial multitasking GPUs.

4.1 Design Principle of C-Laius

To address the challenges discussed in Section 3.5, we design and implement C-Laius based on three principles.

- C-Laius should be able to predict the smallest percentage of computational resources needed by a user-facing query to return before the QoS target according to its input data and the scalabilities of its tasks.
- C-Laius should be able to allocate the free computational resources to batch applications for maximizing their throughput while minimizing the performance interference to user-facing queries.
- C-Laius should be able to boost the processing of user-facing queries if they cannot complete before

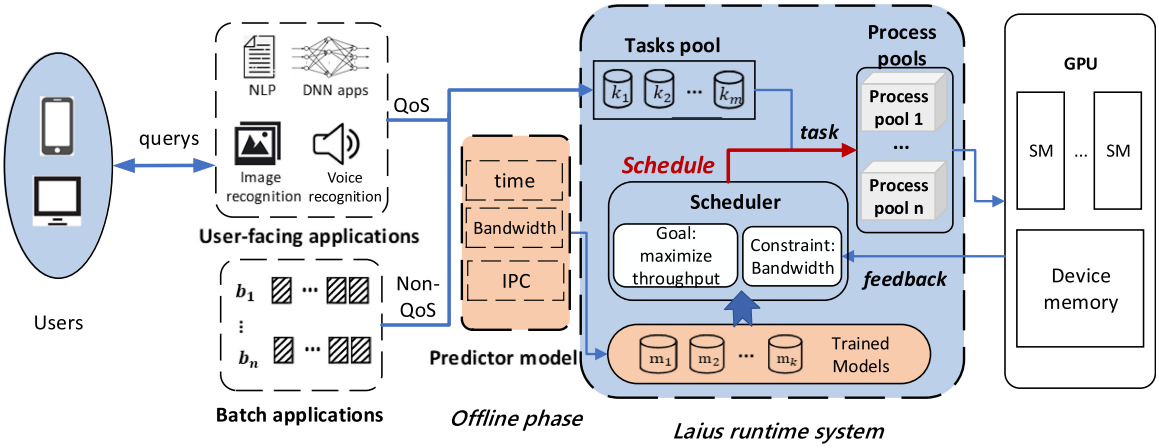


Fig. 5. Design overview of C-Laius.

the QoS target due to the interference from the co-located batch applications.

4.2 C-Laius Overview

Fig. 5 demonstrates the design overview of C-Laius, a runtime system consisting of a *task performance predictor*, a *contention-aware resource allocator* and a *progress-aware lag compensator*. The performance predictor can precisely predict the performance of a task[1] with different resource quotas. The resource allocator maximizes the throughput of batch applications while minimizing the possibility of QoS violation of user-facing queries due to global memory bandwidth contention. Moreover, the lag compensator monitors the progress of user-facing queries and speeds up their execution if they run slower than expected.

As shown in Fig. 5, C-Laius treats tasks of user-facing queries (referred to "*QoS tasks*") and tasks of batch applications (referred to "*non-QoS tasks*") differently. Once a QoS task is submitted, it starts to run directly with "just-enough" resources. And when a non-QoS task is submitted, it is first pushed into a *ready task pool*. C-Laius selects appropriate non-QoS tasks from the ready task pool and executes them only when there are free computational resources.

In more detail, when a user-facing query q is received by GPU, C-Laius processes it in the following steps.

(1) C-Laius predicts the duration of q with different computational resource quotas, identifies "just-enough" quotas so that q can return within the QoS target based on pre-trained duration models. All its tasks run with the same computational resource quota by default (Section 5).

(2) The contention-aware resource allocator allocates the remaining computational resources to execute non-QoS tasks. When performing the allocation, C-Laius aims to maximize the throughput of batch applications while alleviating QoS violation of q due to the contention on global memory bandwidth and shared memory (Section 6). As it is possible that multiple batch applications are co-located with a user-facing service and tasks have divergent characteristics, it is challenging to identify the appropriate allocation.

(3) C-Laius monitors the progress of q. If q runs too slow to meet the QoS, the lag compensator speeds up its execution by allocating more computational resources to q's to-be-executed kernels (Section 7). The difficulties in this step are identifying the new computational resource quota for q's to-be-executed kernels and performing the adjustment because existing accelerators (e.g., Nvidia Volta and Turing GPUs) do not provide an interface to adjust the computational resource allocation during the execution of q.

We propose the *process pool* to enable runtime computational resource reallocation. Specifically, we launch a pool of daemon processes that are configured to run with various accelerator resource percentages (they are idle in most cases). If C-Laius decides to adjust the resource configuration for q during its execution, q's to-be-executed tasks are "hacked" and sent to a daemon process that is configured to run with the corresponding resource configuration. The daemon process then submits these tasks to the accelerator on behalf of q, achieving online resource reallocation.

5 Task Performance Prediction

We train a query duration model for each user-facing service and a performance model for each task. A performance model predicts a task's duration, global memory bandwidth consumption and instructions per cycle (IPC). In a long-running datacenter, it is acceptable to profile a service and build it a model before running it permanently. The profiling is done offline and no runtime overhead is involved.

5.1 Predicting Query Duration

The query duration model is used to identify the "just-enough" computational resources for a user-facing query. We use *input data size* and *percentage of computational resources* as the features to train the query duration model. The input data size reflects the workload of a query, and the percentage of computational resource reflects the computational ability used to process the query.

To build such a duration model for a user-facing service, we submit queries with different inputs to the service, execute them with different computational resource quotas and collect the corresponding duration. During the profiling, queries are executed in solo-run mode to avoid performance interference due to shared resource contention. For a user-facing service, we collect $100 \times 10 = 1000$ samples with 100 different inputs, and 10 different percentages of

1. A kernel or a library call is referred to a task.

TABLE 1
Parameters Used in the Performance Modeling

Task type	Parameters	Dimension
Hand-written kernel	Input data size	1
	Grid size (X*Y*Z)	3
	Block size (X*Y*Z)	3
	Shared memory size	1
	Pct. of computational resource	1
Library call	all parameters	1
	Pct. of computational resource	1

computational resources (increasing from 10 to 100 percent with step 10 percent). We randomly select 80 percent of the samples to train the model and use the rest to evaluate the accuracy of the trained model (Section 5.3).

5.2 Predicting Task Performance

The contention on shared resources, such as global memory bandwidth and shared memory, may result in the QoS violation of query q when it is allocated "just-enough" computational resources. To eliminate the QoS violation and maximize the throughput of batch applications, C-Laius needs to understand the duration, global memory bandwidth and IPC of each task. In this way, when C-Laius assigns the remaining computational resources to non-QoS tasks, it can prioritize the task with higher IPC and lower global memory bandwidth usage (Section 6). By comparing the predicted duration of each task with its actual processing time, the lag compensator can detect potential QoS violation and identify the new resource allocation for q to complete before the QoS target (Section 7).

For a task t, we use instruction-per-cycle to represent its throughput on an accelerator. Let INS and T represent the number of instructions and the processing time of t, respectively. Equation (1) calculates the IPC of t (denoted by IPC_t). In the equation, $Freq$ is the running frequency of the accelerator. Note that, INS and T can be obtained with Night Compute (Nvidia profiling tool) directly, $Freq$ can be found from the design document.

$$IPC_t = \frac{INS}{T \times Freq}. \tag{1}$$

In user-facing services and batch applications, there are generally two types of computational tasks: the hand-written kernel and the library call. Hand-written kernels are written by programmers, while library calls invoke highly optimized common libraries (e.g., cuDNN [31] and cuBLAS [32] on GPUs).

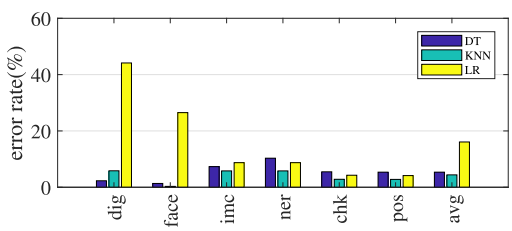
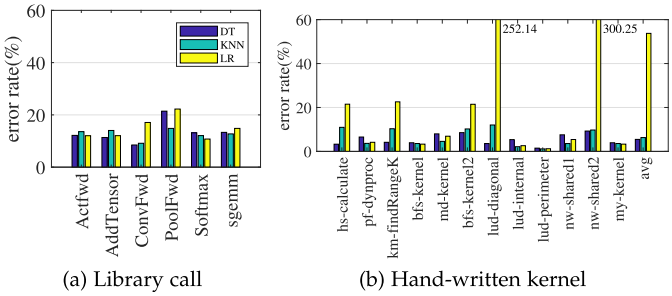


Fig. 6. The errors for predicting query duration.



(a) Library call (b) Hand-written kernel

Fig. 7. Errors of predicting the task duration.

It is challenging to predict the computational tasks' performance owing to the limited information that can be obtained before they are executed. For the two types of tasks, as shown in Table 1, different characters are used to train their performance models. For a hand-written kernel, the parameters we can obtain before it is executed include its configuration (grid size, block size, shared memory size), input data size and compute resource quota. For a library call, because the actual implementation and kernel configurations are hidden behind the API, we need to treat all the kernels in a library call as a whole.

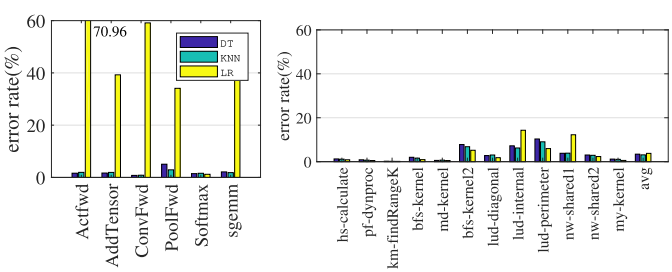
5.3 Determining Low Overhead Models

The QoS target of a user-facing query is hundreds of milliseconds to support smooth user interaction [33]. Choosing modeling techniques with low computation complexity and high prediction accuracy is crucial. We evaluated a spectrum of broadly used regression models for the task performance prediction: k-nearest neighbors (KNN) [34], Linear regression (LR) [35] and Decision Tree (DT) [36]. Besides, some lightweight neural networks are also suitable for performance prediction, such as the Lightweight Augmented Neural Networks (NN+C) proposed in [37].

To construct the training and testing datasets for our prediction model, we have collected a large number of samples while randomly choosing 80 percent of them to train the model and using the rest for testing. The prediction error is measured by Equation (2).

$$Error = \frac{|Predicted\ value - Measured\ value|}{Measured\ value}. \tag{2}$$

Fig. 6 shows the errors of predicting query duration and data transfer duration on the test set with KNN, LR, and DT. As observed from this figure, DT and KNN are accurate for query duration prediction, with the prediction error lower than 5 percent. Figs. 7 and 8 present the errors of predicting



(a) Library call (b) Hand-written kernel

Fig. 8. Errors of predicting the global memory bandwidth usage of tasks.

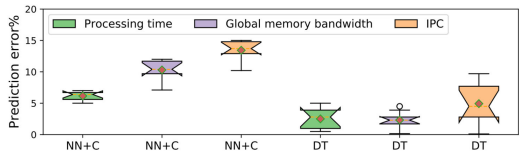


Fig. 9. Errors of predicting models with DT and NN+C.

the duration and global memory bandwidth usage of each task with KNN, LR, and DT. We can find that LR has poor accuracy for prediction, while KNN and DT are accurate for predicting durations and global memory bandwidth usages of both library calls and hand-written kernels. Besides the accuracy, we measure the time of performing a prediction using KNN, LR, and DT. The time of performing a prediction using KNN is longer than 2 milliseconds, while the time of prediction using DT is 0.47 millisecond.

We also evaluate the prediction model NN+C with our training data. Fig. 9 presents the prediction errors of the duration, global memory bandwidth usage, and IPC of kernels/APIs used in C-Laius with DT and NN+C. In general, DT shows higher accuracy for predicting the kernel/API performance than NN+C. Because DT shows higher accuracy with little prediction overhead than other methods according to our measurement, we use DT as the modeling technique to train the performance model.

6 Contention-Aware Resource Allocation

In this section, we present the contention-aware resource allocator that allocates computational resources to co-located applications. The allocator aims to maximize the throughput of co-located batch applications while avoiding performance interference to user-facing queries due to serious global memory bandwidth contention.

Fig. 10 presents the processing flow of the contention-aware resource allocator. As shown in the figure, the resource allocator allocates enough computational resources to QoS tasks directly (Section 6.1). The resource allocator then divides remaining computational resources to non-QoS tasks by modeling the allocation problem as a knapsack problem (Section 6.2).

6.1 Allocating Resource for User-Facing Queries

When a user-facing query q is received, C-Laius obtains its input data size and estimates its duration with various computational resource quotas using the duration model trained in Section 5.1.

The end-to-end latency of q is composed of data transfer time, and task processing time. For query q, let T_{tgt}, T_{pcie}, and T_p represent its QoS target, its data transfer time through PCIe bus, and its actual processing time. T_{pcie} can be collected when q transfers its data to the accelerator. Only when Equation (3) is satisfied, q returns before its QoS target. In Equation (3), T_{tgt}, T_{pcie} are already known when C-Laius allocates computational resources for q.

$$T_p \leq T_{tgt} - T_{pcie}. \tag{3}$$

By comparing $T_{tgt} - T_{pcie}$ with the predicted duration of q using various resource quotas, C-Laius identifies the "just-enough" computational resources for query q. The contention-aware resource allocator then allocates "just-enough" computational resource to q. By default, all QoS tasks in q run with the same computational resource quota.

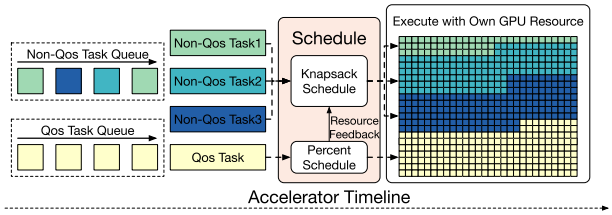


Fig. 10. Contention-aware resource allocation in C-Laius.

6.2 Allocating Resources for Non-QoS Tasks

The remaining computational resources are then allocated to batch applications. Considering a single batch application may not fully utilize computational resources, multiple batch applications can be co-located with a user-facing service. It is non-trivial to allocate resources to non-QoS tasks, because non-QoS tasks may contend for shared resources with query q, resulting in the QoS violation of q.

When C-Laius allocates remaining computational resources to non-QoS tasks, it aims to achieve the best throughput for non-QoS tasks without incurring serious global memory bandwidth contention with QoS tasks. As mentioned in Section 4, the overall throughput of non-QoS tasks is translated to a quantitative IPC goal, which means quotas allocated to non-QoS tasks can be derived from an optimization problem related to its feasible solutions. In more detail, this problem can be formalized to be a single-objective optimization problem [38], where the objective function is maximizing the sum of non-QoS tasks' IPCs and the constraint is global memory bandwidth.

There are two constraints to this optimization problem. First of all, the accumulated global memory bandwidth usage of co-running tasks should be smaller than the available global memory bandwidth of the accelerator to avoid serious bandwidth contention. Second, the computational resource quota allocated to concurrent tasks should not exceed overall available computational resources. Suppose there are n non-QoS tasks waiting in the ready task pool. Let BW, R and x_{QoS} represent the available global memory bandwidth, overall computational resources and the computational resources allocated to the QoS task in query q respectively. Equation 4 expresses the object and the constraints in the optimization problem. In the equation, x_i is the computational resource quota allocated to the ith non-QoS task, $f(x_i)$ and $g(x_i)$ are the predicted IPC and the predicted global memory bandwidth usage of the ith non-QoS task when it is allocated x_i computational resource quota respectively.

$$\begin{aligned} \text{Object:} \quad & MAXIMIZE \quad y = \sum_{i=1}^{n} f(x_i), 0 \leq x_i \leq R \\ \text{Constraint-1:} \quad & \sum_{i=1}^{n} g(x_i) + g(x_{QoS}) \leq BW \\ \text{Constraint-2:} \quad & \sum_{i=1}^{n} x_i + x_{QoS} \leq R. \end{aligned} \tag{4}$$

Many algorithms can be applied to solve the optimization problem. However, it is time-consuming to resolve a continuous optimization problem. To reduce the scheduling

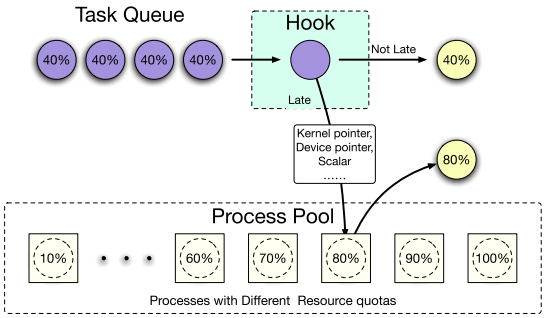


Fig. 11. Enabling resource reallocation using process pool.

overhead, we discretize computational resources allocated to different tasks and transform the continuous optimization problem into a discrete optimization problem, thus significantly reducing the computational complexity to identify the appropriate computational resource allocation. The assumption we made for discretizing computational resources is that: the GPU has computational resources of N quotas, and each quota has $\frac{100}{N}\%$ computational resources. If a task is allocated k quotas of computational resources, $\frac{100\times k}{N}\%$ of computational resources allocates to this task.

The discrete optimization problem can be further modeled to be a complete knapsack problem. Let N_{free} represent the quota of computational resources that are not used by QoS tasks. Suppose there are m non-QoS tasks in the ready task pool. The above discrete optimization problem is the same to find the solution that can maximize the value of items in a backpack of capacity N_{free}, while keeping the weight of the items smaller than N_{free}. In the knapsack problem, there are m items corresponding to the m non-QoS tasks. An item's weight is the computational resource quota of a non-QoS task, and the value of the item is its IPC with the given computational resource quota. As shown in Equation (5), the complete knapsack problem can be further modeled with 0/1 knapsack. In the equation, $V[i][j]$ is the maximum value of the items when the capacity of the backpack is j and the number of items is i in the backpack. m is the weight of the ith item, and $IPC_{i,m}$ is the achieved IPC of the ith non-QoS task when it is allocated m resource quotas. We adopt the dynamic programming technique to resolve this complete knapsack problem.

$$V[i][j] = max(V[i-1][j-m] + IPC_{i,m}); m \in [1, \ldots, j]. \quad (5)$$

It is worth noting that a non-QoS task may get no resource according to the above solution. In this case, the non-QoS task stays in the ready task pool and waits to be launched to GPU when other tasks release some computational resource quotas. Moreover, if the identified resource allocation does not obey the two constraints in Equation (4), the resource allocator invalidates the allocation and searches for another allocation that follows both constraints.

6.3 Enabling Resource Reallocation

The resource allocator needs to update the resource quota allocated to each batch application during its execution. However, emerging Volta MPS does not provide an interface to update the resource quota allocated to a process during its lifetime.

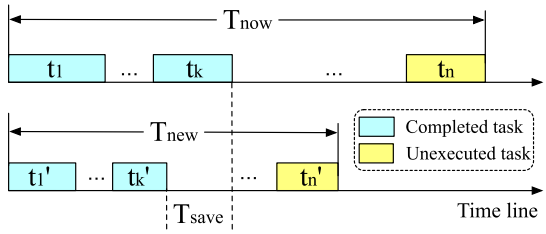


Fig. 12. Identifying a new computational resource quota for a user-facing query to compensate the processing lag.

To solve this problem, we propose *the process pool* in Fig. 11. As shown in the figure, C-Laius launches a pool of processes that are configured to use different computational resource quotas. For a process p (executing a user-facing query or a batch application), C-Laius intercepts all the tasks of p. If the computational resource quota of its tasks is updated, the unexecuted tasks of p are forwarded to run on the process with the expected resource quota. Otherwise, its tasks are executed in p directly.

We intercept an accelerator task through function hooking technique. C-Laius hooks function "cudaLaunchKernel" and APIs in common libraries (e.g., cuDNN), and overrides their function pointers using LD_PRELOAD environment variable. The new implementations of cudaLaunchKernel and the APIs parse and forward the kernel/API pointer, the device (GPU) pointer and parameters to a remote process in the process pool for invocation. Meanwhile, the memory of the executable tasks is also mapped into the remote process's address space, so that the remote process can execute the task using the received kernel/API pointer.

7 Progress-Aware Lag Compensation

The contention-aware resource allocator eliminates the QoS violation of user-facing queries due to global memory bandwidth contention by limiting the global memory bandwidth usage of non-QoS tasks. Besides the contention on global memory bandwidth, concurrent tasks also contend for shared memory and L1 cache so that the contention cannot be explicitly managed. The contention may result in the slow progress of user-facing queries. To this end, we propose a progress-aware lag compensator to monitor the progress of user-facing queries and mitigates the possible QoS violation by adjusting the compute resource quota allocated to each QoS task inspired by the queuing theory [39].

During the execution of a user-facing query q, the compensator periodically checks whether it runs slower than expected due to resource contention. Suppose there are n QoS tasks in query q in total and k of them have completed. Let $t_1, \ldots, t_k$ represent the predicted duration, and $t_1^r, \ldots, t_k^r$ represent the actual duration of the k completed tasks. If $\sum_{i=1}^{k}(t_i^r - t_i)$ is larger than 0, query q runs slower than expected and may suffer from QoS violation. In this scenario, the lag compensator identifies a new computational resource quota for q so that it can return before the QoS target.

Fig. 12 shows the way to identify a new computational resource quota for query q. Let T_{now} and T_{new} represent the predicted durations of q with the current computational resource quota and the newly identified computational

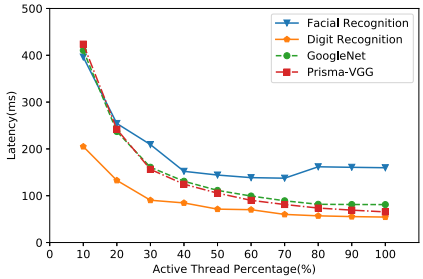


Fig. 13. The latency characteristics of different QoS applications as resource quota changes.

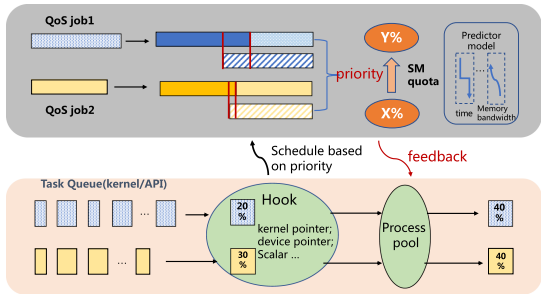


Fig. 14. The multiple QoS tasks scheduler.

resource quota, respectively. If Equation (6) is satisfied, query q can return before the QoS target. In the equation t'_i represents the predicted duration of the ith completed task with the new resource quota. Observed from the equation, $T_{now} - T_{new}$ is the overall reduced duration of query q with the new resource quota. $T_{save} = \sum_{i=1}^{k}(t_i - t'_i)$ is the reduced duration of executing the k already completed tasks. Therefore, $T_{now} - T_{new} - T_{save}$ is the reduced duration of the $n-k$ unexecuted tasks in query q. If the reduced duration of the $n-k$ unexecuted tasks is larger than the lag of the completed tasks $\sum_{i=1}^{k}(t_i^r - t_i)$, query q is able to return before the QoS task.

$$T_{now} - T_{new} - \sum_{i=1}^{k}(t_i - t'_i) \geq \sum_{i=1}^{k}(t_i^r - t_i). \tag{6}$$

Based on Equation (6), the lag compensator is able to identify the new "just-enough" computational resource quota for query q. In the equation, T_{now} and T_{new} can be predicted with the query duration model, t_i and t'_i can be predicted with the task performance model, t_i^r is measured at runtime directly. Once the new quota is identified, C-Laius adopts the process pool proposed in Section 6.3 to run the unexecuted tasks of query q with the new resource quota. Meanwhile, the computational resource quotas allocated to non-QoS tasks are also updated simultaneously.

If the progress of q does not lag behind the expected progress any more with the new quota, the resource quota allocated to q rolls back to its original quota. In this way, C-Laius makes that q completes before the QoS target, and minimizes the resource used by it.

8 Multiple QoS Applications: Design and Implementation

In this section, we present new techniques to determine resource allocation among multiple QoS tasks and non-QoS tasks. C-Laius takes a flexible resource allocation strategy based on priority to explore multiple resource dimensions simultaneously to (1) effectively co-locate multiple user-facing and batch applications, (2) extract hidden opportunities for improving efficiency by exploiting resource sensitivities of tasks. Fig. 14 presents the processing flow of the priority-based resource allocation scheduler.

First, C-Laius needs to determine the initial resources percentage for multiple QoS tasks. The initial determination is particularly important in the scenario of multiple QoS tasks, because both tasks are latency critical. The unscientific initial allocation will directly lead to the final QoS violation. C-Laius uses a method similar to the initial resource allocating strategy for user-facing queries proposed in Section 6.1.

According to the function expression of each QoS task's overall time obtained by prediction and their QoS targets, we determine the percentage of computing resources required for each QoS task $q1, q2$. If the sum of the $q1, q2$ is less than or equal to 100 percent, $q1$ and $q2$ are directly taken as the initial resource percentages of the two QoS tasks respectively. And if the sum of the two is greater than 100 percent, C-Laius will determine which QoS target of two user-facing applications is satisfied first according to the slope of the point where $q1$ and $q2$ are located. In detail, the slopes of $q1$ and $q2$ refer to the tangent slopes of the corresponding coordinate points when running at this percentage in the function of the total execution time of the QoS task obtained from prediction. If the tangent slope of the $q1$ point is greater than which of $q2$, it means that the impact of reducing the resource percentage of the QoS task 1 on its latency is greater than that of the QoS task 2, so the priority of the QoS task 1 is higher. And in this way, C-Laius needs to meet the resource quota requirement of the QoS task 1 preferentially. The percentage of the QoS task 2 is obtained from $100 - q1$.

After describing the initial percentage allocation of two QoS applications that have QoS requirements, the second step needs to consider the dynamic change of the resource allocation during the application execution process. Based on the previous discussion in Section 3, we have proven that pure static optimization cannot fully meet the application's QoS. Due to the process pooling technology we proposed in Section 6.3, we have implemented a resource quota scheduling mechanism at the API and kernel levels. Based on this, we propose the *priority-based resource allocation strategy*.

During the execution of two QoS tasks, the task progress needs to be monitored in real-time. Once in a while, two QoS tasks need to be synchronized to observe the progress of tasks. Each synchronization operation is updated with a priority. Simultaneously, the scheduler determines the resource percentage of the next kernel/API according to the scheduling rule and task priority, and then launches a new percentage process through the enhanced process pool.

The specific priority and scheduling rules are as follows: The first step needs to determine the real-time priority of the two QoS tasks. After a fixed interval, the time spent on completed tasks(APIs/kernels in QoS tasks) obtained by synchronization is $t_{finished}$. The QoS target is t_{QoS}, and the maximum remaining time is t_{rest} as shown in Equation (7) . And if all the subsequent tasks are completed according to the percentage

TABLE 2
Summary of Different Shared Resources on a Spatial Multitasking GPU

Shared resource	Allocation method
the number of SMs	Volta MPS
global memory capacity	capacity limiting
global memory bandwidth	bandwidth limiting

of resources used by the currently hooked process, based on the prediction function, the estimated time is $t_{predict}$. The online compensation time is T. If T is equal to 0, there is no need to modify the current percentage, and the priority of the QoS task is set to 2. Otherwise, the priority of the QoS task is pending. If T is less than 0, the scheduler can appropriately reduce the resource percentage, and the priority of the QoS task is set to 3. The compensation time required for the two tasks are $T1$, $T2$. If $T1 > T2$, the priority of the QoS task 1 is 1, the QoS task 2 is 2, and vice versa.

$$t_{rest} = t_{QoS} - t_{finished} \qquad T = t_{predict} - t_{rest}. \tag{7}$$

Once the priority has been determined, the second step is to determine the detailed resource quotas. Fig. 13 shows the execution time characteristics of some common user-facing applications with the percentage of resources varies. Considering different scalabilities of every task, the latency does not always change linearly as resource percentage increases. For example, increasing the computing resource quota by 10 percent of the QoS task 1 will save 10ms while increasing the computing resource quota by 20 percent of the QoS task 2 will save only 11ms. Therefore, the optimal saving strategy should be included in C-Laius to increase or decrease resource quotas as needed.

Assume that the resource quota of the QoS task1 process being executed is x. To compensate for T, the optionally increased resource quotas are a and b, respectively. And the corresponding shortened latency of each task i in the QoS task1 are x_i and y_i ms.

$$A = \frac{A_a}{A_b}, A_a = a/\sum_{i=1}^{n} x_i, A_b = b/\sum_{i=1}^{n} y_i, \tag{8}$$

where n is the number of remaining tasks (kernels/APIs) in the user-facing application. TH represents the threshold, which we set as 0.9. As shown in Equation (8), A_a and A_b represent the ratio of the increased resources and the total shortened time of the QoS query. Once the ratio A is less than TH the resource quota is a, otherwise b.

During runtime, the scheduler monitors the QoS tasks' priorities from different QoS queries. When the priority is 1, the resource quota required by it is satisfied preferentially.

TABLE 3
Benchmarks Used in the Experiment

Benchmark Suite	Workloads
Tonic suite [28]	face, dig, imc, ner, pos, chk
Rodinia [29]	BFS, B+tree, PF (pathfinder), NW HS (hotspot), MD (lavaMD) MY (myocyte),LUD

TABLE 4
Hardware and Software Specifications

	Specification
Hardware	Intel(R) Xeon(R) CPU E5-2620 v3@ 2.4GHz GeForce RTX 2080Ti
Software	Ubuntu 16.04.5 LTS CUDA Driver 410.78 CUDA SDK 10.0 CUDNN 7.4.2

When the priority is 2, the task will still execute at the current percentage while guaranteeing the QoS of the higher priority. When the priority is 3, the required resources are reduced. If there are remaining resources, batch applications consume them to improve utilization.

In addition to the online resource scheduling of the two QoS applications, C-Laius also considers how to maximize the throughput of batch applications while ensuring QoS requirements of user-facing applications. Similar to Section 6.2, this problem can also be transformed into a single objective optimization implementation. As shown in Table 2, there are three common resource contentions on the spatial multitasking GPU. These three dimensions must be considered as constraints in the optimization problem to avoid the impact of resource contention on QoS tasks, rather than simply allocating the remaining idle resources to batch applications.

There is no doubt that C-Laius's fine-grained scheduling method can allow different kernels of the same task to get the appropriate percentage of resources to utilize GPU resources better. We have modified the original process pool so that it can support two different co-location situations (one or more user-facing applications).

9 Evaluation of C-Laius

9.1 Experimental Setup

We use benchmarks in Tonic suite [28] as user-facing services and use benchmarks in Rodinia benchmark suite [29] as batch applications. Table 3 gives a brief description of the benchmarks. In our experiments, we use 6 user-facing services from Tonic suite as user-facing services and 8 representative batch applications from Rodinia, in which we categorize the first type as computation-intensive works (HS, LUD, MY) due to the possibility of high cache contention and the second type as memory-intensive workloads (BFS, B+ tree, NW, PF, MD) due to the heavy memory traffic.

The experiments are carried out on a machine equipped with one Nvidia GPU RTX 2080Ti. The detailed setups are summarized in Table 4. As to the runtime configuration, there are 40 processes(4 processes to one quota) in the process pool. Note that C-Laius does not rely on any special hardware features of 2080Ti and is easy to be set up on other GPUs with Volta or Turing architecture. From the Turing architecture Whitepaper, we can see that the maximum global memory bandwidth provided by 2080Ti is 616 GB/s.

Throughout our experiments, the QoS target is defined as the 99%-ile latency, and the utilization of the accelerator is calculated as the ratio of the throughput of batch

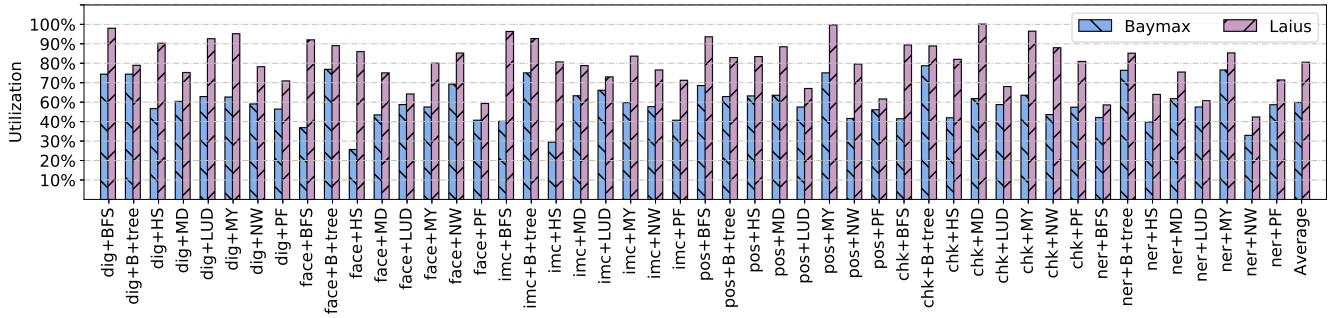


Fig. 15. The normalized throughput of batch applications at co-location with Baymax and C-Laius.

applications normalized to their throughput when they run alone on the experimental platform. As to the resource adjustment granularity, we do the sensitivity analysis using different granularities (3, 5 ,10 and 20 percent). The normalized throughput of batch applications remains the same as granularity increases ($\leq$10%). While the granularity is set to 20 percent, the throughput significantly decreases. So We choose 10 percent resource percentage as the adjustment granularity.

9.2 QoS and Throughput

In this section, we evaluate the effectiveness of C-Laius in maximizing the accelerator throughput while ensuring the QoS of requirement emerging user-facing tasks. We compare C-Laius with Baymax [14]. Baymax predicts the duration of every task and reserves enough GPU time slices for user-facing queries. If the duration of a non-QoS task is shorter than the QoS headrooms of user-facing queries, the non-QoS task is issued. Otherwise, the non-QoS task is blocked. In our experiment, we configure each of the co-located applications to use 100 percent of computational resources for Baymax, to set up an environment that it works.

Fig. 16 presents the 99%-ile latency of user-facing services normalized to their QoS target when they are co-located with batch applications. There are overall $6 \times 8 = 48$ co-location pairs (6 user-facing services and 8 batch applications). Observed from the figure, both C-Laius and Baymax ensure the QoS of user-facing services. On the contrary, the even allocation and priority allocation in Fig. 2 result in QoS violation of user-facing services.

Fig. 15 shows the normalized throughput of batch applications at co-location with Baymax and C-Laius. Observed from the figure, batch applications in all the $6 \times 8 = 48$ co-locations achieve higher throughput with C-Laius than Baymax. Specifically, C-Laius achieves the normalized 70.3 percent utilization of batch applications, while Baymax achieves 49.5 percent utilization on average. In general, C-Laius improves the throughput of batch applications by 20.8 percent at co-location compared with Baymax.

C-Laius allocates computational resources to run different tasks simultaneously while Baymax runs tasks sequentially. C-Laius can squeeze more computational resources to execute batch applications. Space sharing often conveys better resource utilization than time sharing when a single task cannot fully utilize all the resources [7].

As an example, Fig. 17 shows the change of resource quota during the execution of a user-facing query *imc*, when it is co-located with batch applications *BFS*. Observed from the figure, when a query q of *imc* is received, the performance predictor finds that 40 percent of the computational resource is "just-enough" for it. During the execution of q, the lag compensator finds that the query runs slower than expected. In this case, the compensator calculates it as a new resource quota 60 percent, and processes the remaining tasks of the query using the new quota. Later, the progress of q does not lag behind as expected, and the quota rolls back to the original 40 percent. In this way, C-Laius ensures that the query *face* completes before the QoS target, and minimizing the resource used by query q.

9.3 Effectiveness of Constraining the Global Memory Bandwidth Contention

C-Laius predicts the global memory bandwidth requirements of all tasks and makes sure that the overall global memory bandwidth usage of the concurrent tasks is smaller than the peak available global memory bandwidth in the accelerator. To evaluate the effectiveness of this constraint in eliminating QoS violation due to global memory bandwidth contention, we compare C-Laius with C-Laius-NB, a system that disables the global memory bandwidth constraints when allocating computational resources to non-QoS tasks.

Fig. 18 presents the 99%-ile latency of user-facing services at co-location in C-Laius-NB. As observed from the figure, user-facing services in 25 out of the 48 co-locations suffer from QoS violation in C-Laius-NB. For instance, *dig*

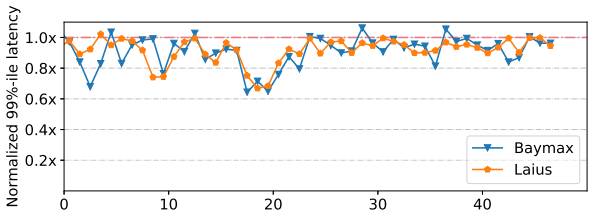


Fig. 16. The 99%-ile latency of user-facing services normalized to their QoS targets in the 48 co-locations.

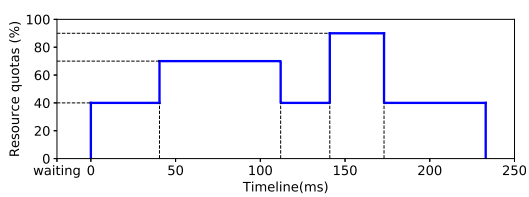


Fig. 17. The change of resource quota allocated to a user-facing query *imc* when it is co-located with *BFS*.

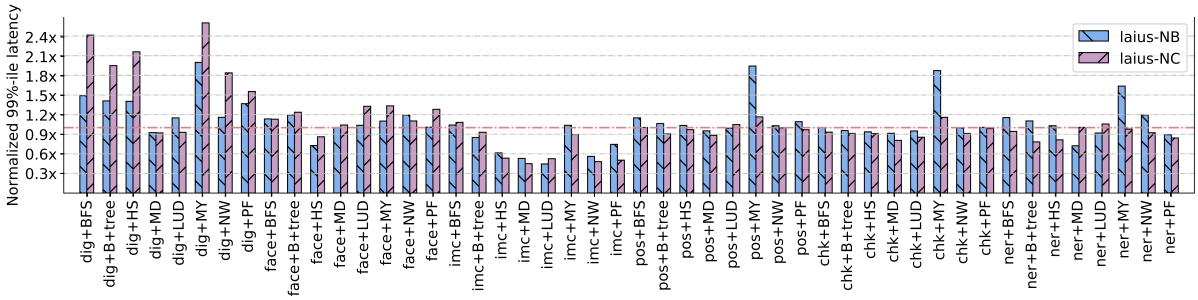


Fig. 18. The 99%-ile latency of user-facing services normalized to the QoS target with C-Laius-NB and C-Laius-NC.

suffers from up to 2X QoS violation when it is colocated with *MY*. Since the only difference between C-Laius and C-Laius-NB is whether to disable global memory bandwidth constraints when scheduling. So the QoS violation is due to the unmanaged global memory bandwidth contention. When the non-QoS tasks in *MY* co-runs with QoS tasks in *dig*, even if *dig* should be able to complete before the QoS target when it runs alone, the global memory bandwidth contention results in serious performance degradation of the QoS tasks. Although the lag compensator can allocate more resources to the unexecuted tasks of *dig*, it is possible that the lag is too long to be compensated. By constraining the global memory bandwidth contention, the lag tends to be short, thus it can be easily compensated if necessary.

For some co-location pairs, the QoS of user-facing services is satisfied using C-Laius-NB. This is because the co-located applications in these pairs do not contend for global memory bandwidth seriously. In this case, with the precise performance predictor and lag compensator, C-Laius-NB is enough to ensure the QoS of user-facing services if the co-located applications are not global memory bandwidth intensive applications.

9.4 Effectiveness of the Lag Compensator

In this section, we verify the need for the compensation mechanism. We remove the compensation part of C-Laius and test the system. Fig. 18 shows the existence of QoS violation in C-Laius without the compensator. It implies that in addition to the contention of bandwidth and computing resources, there is other resource contention, such as shared memory contention.

C-Laius monitors the progress of user-facing queries at co-location and allocates more resources to a slow query to compensate for the processing lag. To evaluate this design choice, Fig. 18 also presents the 99%-ile latency of user-facing services at co-location in C-Laius-NC, a system that disables the lag compensator in C-Laius.

Observed from Fig. 18, user-facing services in 18 out of the 48 co-locations suffer from QoS violation in C-Laius-NC. For instance, *dig* suffers from up to 2X QoS violation when it is colocated with *BFS*. The QoS violation is due to contention on other unmanaged shared resources. As we mentioned before, besides global memory bandwidth, tasks of different applications may run in the same SMs, thus share the shared memory and L1 cache in the SM. In this scenario, even if the performance interference from global memory bandwidth is eliminated, the contention on shared memory and L1 cache may also result in the performance degradation of co-located QoS tasks. Without the lag compensator, the degradation results in the QoS violation of user-facing services.

9.5 Effectiveness of the Priority-Based Resource Allocation Among Multiple QoS Tasks

We also evaluate the effectiveness of C-Laius in maximizing the GPU throughput while ensuring the QoS targets of emerging user-facing applications in multiple QoS situations. And we verify the need for the priority-based scheduling mechanism. We compare C-Laius with the static optimal policies rather than Baymax. In our previous experiments, we proved that Baymax is very unsuitable for scenarios with multiple QoS tasks, leading to strong QoS violation.

In the experiment, we used 6 benchmarks in the tonic suite and combined them to generate 15 different co-location pairs. As shown in Fig. 21, in the static optimal scenario, 8 out of 15 groups of experiments experienced QoS violations. After our priority-based scheduling, all QoS tasks pairs in 15 groups of experiments can meet their QoS requirements as shown in Fig. 20. As for the GPU utilization, Fig. 19 shows the normalizied throughput at co-location with the static optimal baseline and C-Laius. And the last column in the figure shows the average results. Compared with the result of the static optimal baseline, C-Laius has increased the normalized throughput at co-location by 35.9 percent on average. Specifically, C-Laius can significantly improve the utilization rate of the accelerator in the scenario of multiple QoS tasks, and at the same time ensure the QoS goals of multiple tasks. To conclude, the priority-based scheduling mechanism is capable of eliminating QoS violation due to the potential contention on shared resources.

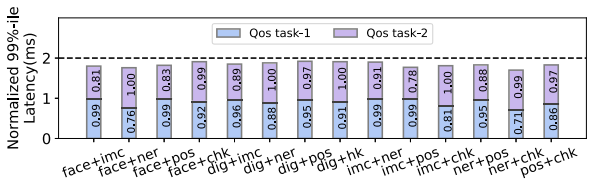


Fig. 19. The normalized overall throughput at co-location with C-Laius and baseline.

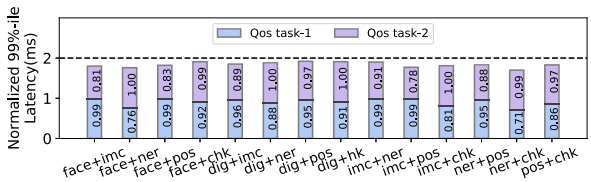


Fig. 20. The 99%-ile latency of user-facing services normalized to the QoS target with two QoS tasks in C-Laius.

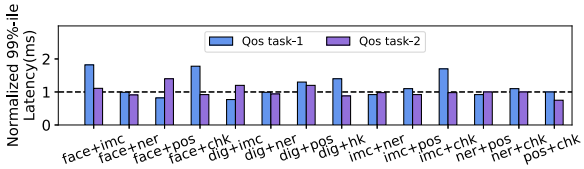
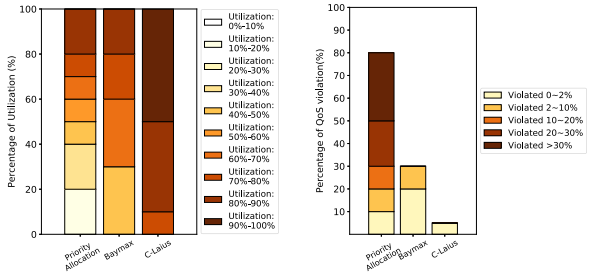


Fig. 21. The 99%-ile latency of user-facing services normalized to the QoS target with two QOS in static optimal baseline.



(a) Throughput at different levels (b) Percentage of QoS violation.

Fig. 22. The scale-out results with Prior,Baymax, and C-Laius.

9.6 Overhead of C-Laius

The main overhead of two cases in C-Laius is from the process pool in contention-aware resource allocation. And the overhead of the process pool originates from three sources: First, to maintain the consistency of data across the origin process and the processes in the process pool, synchronization is essential. We reduce the times of synchronization through accurately scheduling by 80.8 percent, other than switching between processes with different quota frequently. Second, we adopt the CUDA IPC and other techniques to share GPU resources between processes, which consume 3.2 ms in execution. But, these additional operations can be overlapped(e.g., execution for sharing same GPU device pointer only needs to be executed once, the first time hooked). Finally, communications among origin process, scheduler and process pool also have a significant effect on overhead, which finished in 4.8 ms on average. Overall, the overhead of switching the execution resource quota of the task through the process pool we designed is less than 4 percent in one query.

9.7 Large-Scale Cloud Study

In this section, we conduct experiments to evaluate the effectiveness of C-Laius in a GPU-outfitted datacenter scenario. Therefore, we model a datacenter composed of 600 Nvidia RTX 2080Ti GPUs in the same way as Prophet [15], 100 GPUs for each type of user-facing applications in Tonic suite shown in Table 3. The batch workloads are evenly selected from Rodinia shown in Table 3. In the experiment, we use pair-wise co-locations, and randomly select three batch applications to colocate with each user-facing application. As shown in Fig. 22a, the models achieve higher GPU utilization and lower QoS violation with C-Laius compared with priority-based allocation policy and Baymax in large-scale datacenters. On average, C-Laius improves the normalized utilization by 35.2 percent compared with Baymax. As shown in Fig. 22b, 30.6 percent of user-facing applications suffer from severe QoS violations(>30 percent degradation) with Priority Allocation. On the contrary, C-Laius and Baymax can maintain the QoS of user-facing applications, and less than 4.8 percent of user-facing applications suffer from insignificant QoS violations (less than 2 percent degradation) with C-Laius which is better than 24.3 percent with Baymax.

10 Conclusion

C-Laius improves the hardware utilization in spatial multitasking GPUs while guaranteeing the QoS requirement of user-facing applications. To achieve this purpose, C-Laius enables precise task duration prediction, contention-aware resource allocation, and progress-aware lag compensator. Through evaluating C-Laius with emerging user-facing services, we demonstrate the effectiveness of C-Laius in eliminating QoS violation due to insufficient computational resources, global memory bandwidth contention, and contentions on other unmanaged shared resources. C-Laius improves the throughput of batch applications at co-location by 20.8 percent on average compared with state-of-the-art techniques, without violating the QoS of 99%-ile latency for user-facing services. As to the case of multiple user-facing applications, C-Laius ensures no violation of QoS while improving the overall throughput by 35.9 percent on average.

Acknowledgments

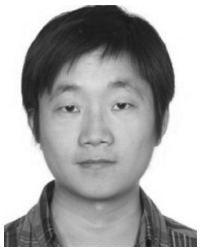
This work was sponsored in part by the National R&D Program of China under Grant 2018YFB1004800, in part by the National Natural Science Foundation of China (NSFC) under Grant 62022057, Grant 61832006, Grant 61632017, and Grant 61872240.

References

[1] A. Broder, "A taxonomy of web search," *ACM SIGIR Forum*, vol. 36, no. 2, pp. 3–10, 2002.
[2] Apple siri. Accessed: Mar. 17, 2021. [Online]. Available: https://www.apple.com/siri/
[3] Google translate. Accessed: Mar. 17, 2021. [Online]. Available: https://translate.google.com/
[4] N. Jones, "Computer science: The learning machines," *Nat. News*, vol. 505, no. 7482, 2014, Art. no. 146.
[5] L. A. Barroso, J. Clidaras, andU. Hölzle,"The datacenter as a computer: An introduction to the design of warehouse-scale machines," *Synthesis Lectures Comput. Archit.*, vol. 8, no. 3, pp. 1–154, 2013.
[6] J. DeanandL. A. Barroso,"The tail at scale," *Commun. ACM*, vol. 56, no. 2, pp. 74–80, 2013.
[7] Z. Wang, J. Yang, R. Melhem, B. Childers, Y. Zhang, and M. Guo, "Simultaneous multikernel GPU: Multi-tasking throughput processors via fine-grained sharing," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2016, pp. 358–369.
[8] W. Zhao *et al.*, "Themis: Predicting and reining in application-level slowdown on spatial multitasking GPUs," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2019, pp. 653–663.
[9] J. J. K. Park, Y. Park, andS. Mahlke,"Dynamic resource management for efficient utilization of multitasking GPUs," *ACM SIGOPS Operating Syst. Rev.*, vol. 51, no. 2, pp. 527–540, 2017.
[10] Multi-process service. Accessed: Mar. 17, 2021. [Online]. Available: https://docs.nvidia.com/deploy/mps/index.html
[11] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa, "Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations," in *Proc. 44th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2011, pp. 248–259.
[12] H. Yang, A. Breslow, J. Mars, andL. Tang,"Bubble-flux: Precise online QoS management for increased utilization in warehouse scale computers," *ACM SIGARCH Comput. Archit. News*, vol. 41, no. 3, pp. 607–618, 2013.
[13] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, andC. Kozyrakis," Heracles: Improving resource efficiency at scale," *ACM SIGARCH Comput. Archit. News*, vol. 43, no. 3, pp. 450–462, 2015.

[14] Q. Chen, H. Yang, J. Mars, andL. Tang,"Baymax: QoS awareness and increased utilization for non-preemptive accelerators in warehouse scale computers," *ACM SIGARCH Comput. Archit. News*, vol. 44, no. 2, pp. 681–696, 2016.
[15] Q. Chen, H. Yang, M. Guo, R. S. Kannan, J. Mars, andL. Tang," Prophet: Precise QoS prediction on non-preemptive accelerators to improve utilization in warehouse-scale computers," *ACM SIGARCH Comput. Archit. News*, vol. 45, no. 1, pp. 17–32, 2017.
[16] C. DelimitrouandC. Kozyrakis,"Paragon: QoS-aware scheduling for heterogeneous datacenters," *ACM SIGPLAN Notices*, vol. 48, no. 4, pp. 77–88, 2013.
[17] W. Zhang *et al.*, "URSA: Precise capacity planning and fair scheduling based on low-level statistics for public clouds," in *Proc. 49th Int. Conf. Parallel Process.*, 2020, pp. 1–11.
[18] S. Chen, C. Delimitrou, and J. F. Martínez, "PARTIES: QoS-aware resource partitioning for multiple interactive services," in *Proc. 24th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2019, pp. 107–120.
[19] Q. Chen, Z. Wang, J. Leng, C. Li, W. Zheng, and M. Guo, "Avalon: Towards QoS awareness and improved utilization through multi-resource management in datacenters," in *Proc. ACM Int. Conf. Supercomputing*, 2019, pp. 272–283.
[20] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa, "TimeGraph: GPU scheduling for real-time multi-tasking environments," in *Proc. USENIX Conf. USENIX Annu. Tech. Conf.*, 2011, pp. 17–30.
[21] G. A. Elliott, B. C. Ward, and J. H. Anderson, "GPUSync: A framework for real-time GPU management," in *Proc. IEEE 34th Real-Time Syst. Symp.*, 2013, pp. 33–44.
[22] H. Lee, A. Faruque, and M. Abdullah, "GPU-EvR: Run-time event based real-time scheduling framework on GPGPU platform," in *Proc. Conf. Des. Autom. Test Eur.*, 2014, Art. no. 220.
[23] W. Zhao, Q. Chen, and M. Guo, "KSM: Online application-level performance slowdown prediction for spatial multitasking GPGPU," *IEEE Comput. Archit. Lett.*, vol. 17, no. 2, pp. 187–191, Jul.–Dec. 2018.
[24] S. Pai, M. J. Thazhuthaveetil, andR. Govindarajan,"Improving GPGPU concurrency with elastic kernels," *ACM SIGPLAN Notices*, vol. 48, no. 4, pp. 407–418, 2013.
[25] Y. Suzuki, S. Kato, H. Yamada, and K. Kono, "GPUvm: Why not virtualizing GPUs at the hypervisor?," in *Proc. USENIX Conf. USENIX Annu. Tech. Conf.*, 2014, pp. 109–120.
[26] B. Wu, X. Liu, X. Zhou, andC. Jiang,"FLEP: Enabling flexible and efficient preemption on GPUs," *ACM SIGOPS Operating Syst. Rev.*, vol. 51, no. 2, pp. 483–496, 2017.
[27] C. Nvidia, "Nvidias next generation CUDA compute architecture: Kepler GK110," WhitePaper, 2012.
[28] J. Hauswald *et al.*, "DjiNN and tonic: DNN as a service and its implications for future warehouse scale computers," in *Proc. ACM/IEEE 42nd Annu. Int. Symp. Comput. Archit.*, 2015, pp. 27–40.
[29] S. Che *et al.*, "Rodinia: A benchmark suite for heterogeneous computing," in *Proc. IEEE Int. Symp. Workload Characterization*, 2009, pp. 44–54.
[30] NVIDIA, "Multi-process service," 2015. [Online]. Available: https://docs.nvidia.com/deploy/mps/index.html
[31] S. Chetlur *et al.*, "cuDNN: Efficient primitives for deep learning," 2014, *arXiv:1410.0759*.
[32] Cublas library. Accessed: Mar. 17, 2021. [Online]. Available: https://docs.nvidia.com/cuda/cublas/index.html
[33] J. DeanandL. A. Barroso,"The tail at scale," *Commun. ACM*, vol. 56, no. 2, pp. 74–80, 2013.
[34] G. James, D. Witten, T. Hastie, and R. Tibshirani, *An Introduction to Statistical Learning*, vol. 112. Berlin, Germany: Springer, 2013.
[35] G. A. Seber and A. J. Lee, *Linear Regression Analysis*, vol. 329. Hoboken, NJ, USA: Wiley, 2012.
[36] S. R. Safavian and D. Landgrebe, "A survey of decision tree classifier methodology," *IEEE Trans. Syst., Man, Cybern.*, vol. 21, no. 3, pp. 660–674, May/Jun. 1991.
[37] A. Srivastava, N. Zhang, R. Kannan, and V. K. Prasanna, "Towards high performance, portability, and productivity: Lightweight augmented neural networks for performance prediction," *CoRR*, vol. abs/2003.07497, 2020.
[38] C. A. C. Coello,"Treating constraints as objectives for single-objective evolutionary optimization," *Eng. Optim.+ A35*, vol. 32, no. 3, pp. 275–308, 2000.
[39] M. Harchol-Balter, *Performance Modeling and Design of Computer Systems: Queueing Theory in Action*. Cambridge, U.K.: Cambridge Univ. Press, 2013.

Wei Zhang received the BSc degree from the Huazhong University of Science and Technology, China. She is currently working toward the PhD degree under supervision of Dr. Minyi Guo with Department of Computer Science and Engineering, Shanghai Jiao Tong University, China. Her research interests include high performance computing and GPU resource management in datacenters.



Quan Chen received the PhD degree from the Department of Computer Science and Engineering, Shanghai Jiao Tong University, China, June 2014. He is a tenure-track associate professor with the Department of Computer Science and Engineering, Shanghai Jiao Tong University, China. His research interests include high performance computing, task scheduling in various architectures, resource management in datacenter, runtime system, and operating system.

Ningxin Zheng received the MSc degree from the Department of Computer Science and Engineering, Shanghai Jiao Tong University, China. He is currently working at Microsoft Research Asia. His research interest includes resource management and job scheduling in datacenters.

Weihao Cui received the BSc degree from Shanghai Jiao Tong University, China. He is currently working toward the PhD degree in the field of computer science under supervision of Dr. Quan Chen with Department of Computer Engineering Faculty of Shanghai Jiao Tong University, China. His research interests include high performance computing and resource management of accelerators in datacenters.

Kaihua Fu recieved the BS degree from the University of Electronic Science and Technology of China, China. He is currently working toward the PhD degree with the Department of Computer Science and Engineering, Shanghai Jiao Tong University, China. His research interests include resource management in various architecture and high performance computing.

Minyi Guo (Fellow, IEEE) received the PhD degree in computer science from the University of Tsukuba, Japan. He is currently Zhiyuan Chair professor, Shanghai Jiao Tong University, China. His present research interests include parallel/distributed computing, compiler optimizations, embedded systems, big data, and cloud computing. He is currently on the editorial board of the *IEEE Transactions on Parallel and Distributed Systems* and the *Journal of Parallel and Distributed Computing*. He is a CCF fellow.