Laius: Towards Latency Awareness and Improved Utilization of Spatial Multitasking Accelerators in Datacenters

Wei Zhang
Shanghai Jiao Tong University
zhang-w@sjtu.edu.cn

Weihao Cui
Shanghai Jiao Tong University
weihao@sjtu.edu.cn

Kaihua Fu
Shanghai Jiao Tong University
midway2018@163.com

Quan Chen
Shanghai Jiao Tong University
chen-quan@cs.sjtu.edu.cn

Daniel Edward Mawhirter
Colorado School of Mines
dmawhirt@mymail.mines.edu

Bo Wu
Colorado School of Mines
bwu@mines.edu

Chao Li
Shanghai Jiao Tong University
lichao@cs.sjtu.edu.cn

Minyi Guo
Shanghai Jiao Tong University
guo-my@cs.sjtu.edu.cn

ABSTRACT

Datacenters use accelerators to provide the significant compute 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 accelerator utilization, and prior work has focused on enabling co-location on multicore processors and traditional non-preemptive accelerators. However, current accelerators are evolving towards spatial multitasking and introduce a new set of challenges to eliminate QoS violation. To address this open problem, we explore the underlying causes of QoS violation on spatial multitasking accelerators. In response to these causes, we propose 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. Our evaluation on a Nvidia RTX 2080Ti GPU shows that Laius improves the utilization of spatial multitasking accelerators by 20.8%, while achieving the 99%-ile latency target for user-facing services.

KEYWORDS

Spatial multitasking, QoS, Improved utilization

ACM Reference Format:
Wei Zhang, Weihao Cui, Kaihua Fu, Quan Chen, Daniel Edward Mawhirter, Bo Wu, Chao Li, and Minyi Guo. 2019. Laius: Towards Latency Awareness and Improved Utilization of Spatial Multitasking Accelerators in Datacenters. In *2019 International Conference on Supercomputing (ICS '19), June 26–28, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3330345.3330351

Quan Chen and Minyi Guo are the members of Shanghai Institute for Advanced Communication and Data Science, Shanghai Jiao Tong University. Quan Chen and Minyi Guo are the corresponding authors of this paper.

ICS '19, June 26–28, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6079-1/19/06…$15.00
https://doi.org/10.1145/3330345.3330351

1 INTRODUCTION

Datacenters often host user-facing applications (e.g., web search [8], web service [8], memcached [23]) that have stringent latency requirements. It is crucial to guarantee that the end-to-end latencies of users' queries are shorter than a predefined Quality-of-Service (QoS) target. With the quick advance of machine learning technology, emerging user-facing applications, such as Apple Siri [1], Google Translate [2] and Prisma [4], start to use machine learning technologies (e.g., Deep Neural Network) that are often computational demanding. Datacenters have adopted accelerators (e.g., GPUs, FPGAs and ASICs) to run these services so that they can achieve the required latency target [6, 12, 22]. As prior work states, user-facing applications experience diurnal user access patterns (leaving the accelerator resources under-utilized for most of time except peak hours) [7, 15]. 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 accelerators for higher aggregated throughput for co-location applications [5, 36, 43]. For instance, the latest Nvidia Volta and Turing architectures allow kernels to simultaneously share certain portions of the computational resources. Leveraging the new generation Multi-Process Service (MPS) [31], it is possible to allocate a small percentage of computational resources (active threads) to global memory bandwidth-bound applications and use most computational resource 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 [10, 11, 28, 30, 42, 47]. On CPUs, applications contend for the cores, shared cache, and the main memory bandwidth; On traditional GPUs, kernels are queued up for the processing elements. 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 scalabilities of the kernels**, and **the contention on shared resources**. Therefore, prior work is not applicable for

spatial multitasking accelerators, because the latency of a user-facing query at co-location on these hardwares is impacted by different factors.

For CPU co-location, prior work falls into two categories: profiling-based and feedback-based. The profiling-based method, such as Bubble-Up [30], 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 [28], builds a decision tree 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 [17, 28, 30, 45, 47]. For spatial multitasking GPUs on the other hand, user-facing queries have different workloads and may have to run with various amounts of computational resources. Due to the ignorance of the impact of the computational resources on a query's performance, prior studies on CPU co-location are not applicable for the new generation spatial multitasking accelerators.

For application co-location on traditional accelerators, because kernels from the co-located applications queued up for processing elements, queuing-based methods (e.g., Baymax [11]) are proposed to eliminate QoS violation due to the long queuing time. It predicts the duration of every GPU task, reserves time slots for each user-facing query, and uses the remaining slots to process batch applications. In contrary, on spatial multitasking GPUs, kernels share processing elements spatially. The queuing-based method does not apply for spatial multitasking accelerators.

It is challenging to determine the amount of computational resource 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 **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 so that its QoS is satisfied. When Laius assigns the remaining resource 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. If the query runs slower than expected due to the contention on other shared resources, the progress-aware lag compensator allocates more resource to the unexecuted kernels of the query to enforce its QoS. In this work, we rely on the bandwidth reservation technique proposed in Baymax [11] to ensure that a user-facing query can always transfer data in full speed, thus eliminates 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 accelerators** - 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 accelerators** - We propose 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, Laius allocates it more computational resource to compensate the lag.

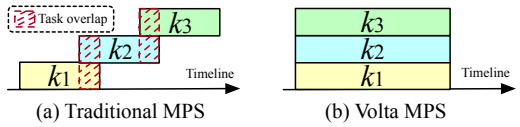


Figure 1: Comparison of the original MPS and Volta MPS.

Our experiment on a Nvidia RTX 2080Ti GPU shows that Laius is able to improve the throughput of batch applications by 20.8% compared with state-of-the-art solution Baymax [11], while guaranteeing the 99%-ile latency of user-facing services.

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-colocation [17, 18, 30, 45, 47]. Bubble-Up [30] and Bubble-Flux [45] identify "safe" co-locations that bound performance degradation while improving chip multiprocessor utilization. SMiTe [47] further extends Bubble-Up and Bubble-Flux to predict performance interference between applications on simultaneous multithreading (SMT) processors. However, all these interference prediction techniques do not consider computational resource allocation, thus do not apply to spatial multitasking accelerators.

For co-location on accelerators, MPS (Multi-Process Service) scheduling [31] enables multiple applications sharing a GPU concurrently. TimeGraph [24] and GPUSync [19] 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 [26] launches different applications to different streaming multiprocessors (SMs) on one GPU. Baymax [11] 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 Laius. KSM [48] predicts the slowdown of co-located applications on spatial multitasking accelerators. However, it relies on a broad spectrum of performance event statistics that are not available on real system GPUs. And KSM is not able to identify the "just-enough" computational resource quota for user-facing queries as Laius does.

3 BACKGROUND AND MOTIVATION

3.1 Spatial Multitasking Accelerators

GPU is one of the most popular accelerators that support spatial multitasking. A GPU often has multiple Streaming Multiprocessors

(SMs) that share the global memory. For instance, Nvidia RTX 2080Ti, a GPU of Turing architecture, has 68 SMs that can run 1024 active threads (i.e., computational resources) concurrently. The SMs share a 12GB global memory.

Since a single kernel may not be able to utilize all the SMs and other on-chip resources all the time [11, 27, 35, 41, 44], starting from Kepler architecture [34], Nvidia proposed Multi-Process Service (MPS) technique [31] 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 [11, 31].

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 is not able to use all computational resources, the remaining threads are allocated to run other kernels. In this case, as shown in Figure 1(a), kernels are very likely to be still executed sequentially with little concurrent execution. In this scenario, the on-chip shared memory and L1 cache etc. are still under-utilized in most cases.

To better utilize the hardware, Volta and Turing architectures introduce Volta MPS with new capabilities that allow multiple applications to share GPU computational resources simultaneously, thereby increasing overall GPU utilization [31]. As shown in Figure 1(b), the new Volta MPS technique enables explicit GPU computational resource allocation, where the full 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 for other spatial multitasking accelerators. 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 [20], as user-facing services, and use benchmarks in Rodinia [9] as batch applications.More details of the experimental hardware and benchmarks are described in Section 8.

3.3 Problem of QoS Violation

Figure 2 shows the QoS violation of user-facing services at co-location adopting the new Volta MPS technique [32]. Adopting Volta MPS, when n applications are co-located, we allocate computational resources in 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 [32]. With priority allocation, the user-facing service is allocated 100% 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 this figure, 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% 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. In this case, the concurrent data access from global memory and shared memory degrades the performance of user-facing queries, which results in QoS violation.

3.4 Factors that Affect the End-to-End Latency

To understand the QoS violation problem at co-location, Figure 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. Analyzed from this figure, three key factors affect the end-to-end latency of a user-facing query q at co-location.

(1) The percentage of computational resources allocated to q. If the percentage is too small, there are not enough active threads to process its tasks, resulting in its long latency. In 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 scalability of a task 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 [32], 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 Accelerators

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 accelerators. 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 percentage of computational resources needed to complete a query within the QoS target varies. There is not a fixed yet the best percentage of computational resources existing 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 between co-located applications. Because tasks have different pressures on the shared memory and global memory

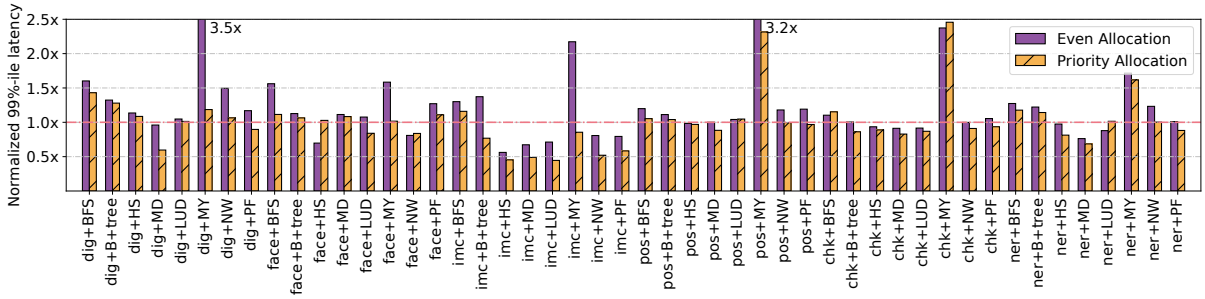


Figure 2: QoS violation of user-facing applications with Volta MPS that adopts even allocation and priority allocation.

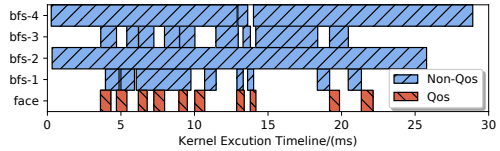


Figure 3: Execution trace of user-facing service *face* and batch applications *BFS* on a spatial multitasking GPU.

bandwidth, a user-facing query may suffer from different performance degradations in two runs even if it is co-located with the same batch applications.

(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 accelerators. 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.

4 LAIUS METHODOLOGY

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

4.1 Design Principals of Laius

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

- Laius should be able to predict the smallest percentage of computational resource needed by a user-facing query to return before the QoS target according to its input data and the scalabilities of its tasks.
- 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.
- Laius should be able to boost the processing of user-facing queries if they cannot complete before the QoS target due to the interference from the co-located batch applications.

4.2 Laius Overview

Figure 4 demonstrates the design overview of Laius, a runtime system consists of a *task performance predictor*, a *contention-aware resource allocator* and a *progress-aware lag compensator*. The performance predictor can precisely estimate the performance of a task[1] with different computational 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 Figure 4, 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 firstly pushed into a *ready task pool*. 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 a spatial multitasking accelerator, Laius processes it in the following steps.

(1) Laius predicts the duration of q with different computational resource quotas, identifies "just-enough" computational resource 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, 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) 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 hard points 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 Laius decides to adjust

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

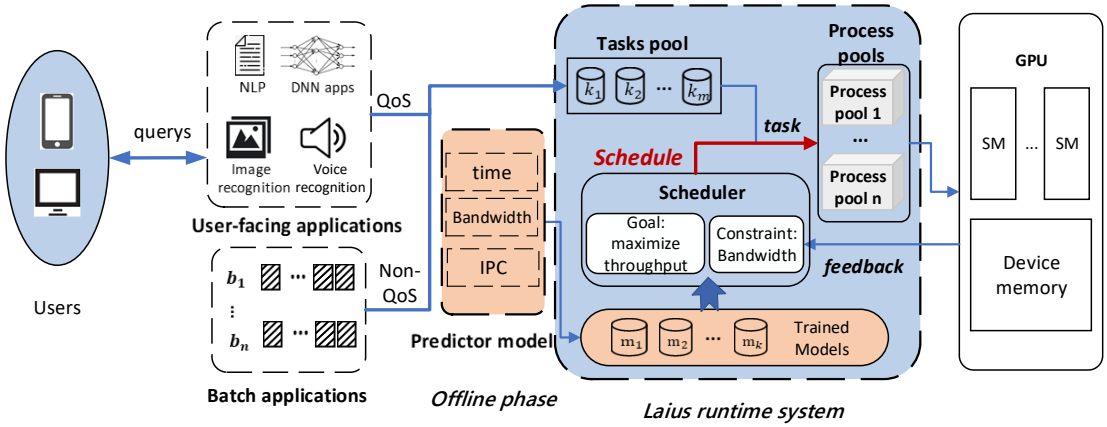


Figure 4: Design overview of Laius.

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 task 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 new 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 computational resources (increasing from 10% to 100% with step 10%). We randomly select 80% 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, Laius needs to understand the duration, global memory bandwidth and IPC of each task. In this way, when 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 (IPC) 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) [3] directly at runtime, $Freq$ 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: hand-written kernel and library call. Hand-written kernels are written by the programmers from scratchpad, library calls are the invoking of highly optimized common libraries (e.g.,cuDNN [13] and cuBLAS [33] on GPU).

It is challenging to predict the performance of the computational tasks 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 [16, 37]. Choosing modeling techniques with low computation complexity and high prediction accuracy is crucial. We evaluated a spectrum of broadly used regression

Table 1: Parameters used in the task 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

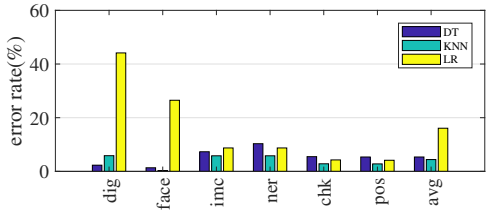
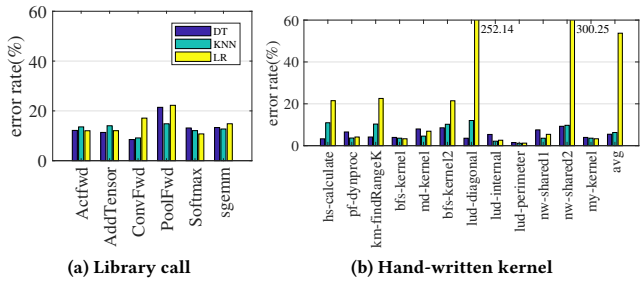


Figure 5: The errors for predicting query duration.



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

Figure 6: Errors of predicting the task duration.

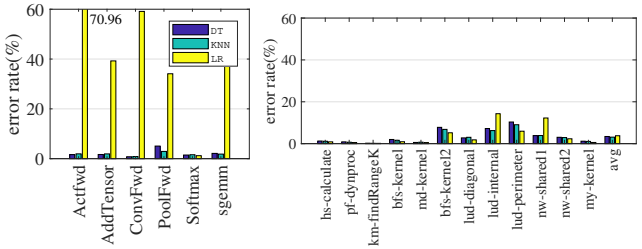
models for the task performance prediction: KNN (k-nearest neighbors) [21], LR (Linear regression) [40] and DT (Decision Tree) [38]. DNN (Deep Neural Network) is famous for its high prediction accuracy. However, the prediction time of existing DNNs [46] is too high to be used at runtime.

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

$$Error = \frac{|Predicted\ value - Measured\ value|}{Measured\ value} \quad (2)$$

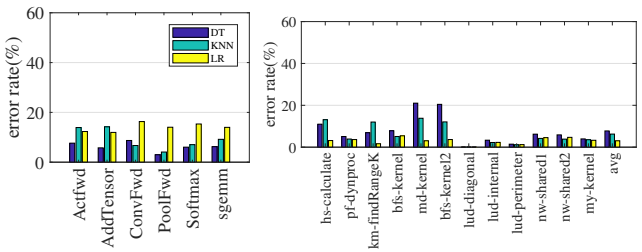
Figure 5 shows the errors of predicting query duration and data transfer duration on the test set with KNN, LR, and DT. Observed from this Figure, DT and KNN are accurate for query duration prediction, with the prediction error lower than 5%. Besides the accuracy, we measure the time of performing a 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. Laius adopts DT to train the query duration model for user-facing services.

Figure 6, Figure 7 and Figure 8 present the errors of predicting the duration, global memory bandwidth usage, and IPC of each task with KNN, LR, and DT. From three Figures, we can find that LR has poor accuracy for predicting the duration of hand-written kernels, global memory bandwidth usage of the library call. In contrary,



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

Figure 7: Errors of predicting the global memory bandwidth usage of tasks.



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

Figure 8: Errors of predicting the IPC of tasks.

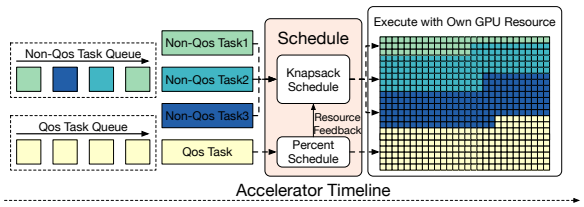


Figure 9: Contention-aware resource allocation in Laius.

KNN and DT are accurate for predicting durations, global memory bandwidth usage, and IPC of both library calls and hand-written kernels. Because DT has a shorter time to perform a prediction than KNN according to our measurement, we use DT as the modeling technique to train the kernel performance model.

6 CONTENTION-AWARE RESOURCE ALLOCATION

In this section, we present 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 the performance interference to user-facing queries due to serious global memory bandwidth contention.

Figure 9 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 and allocates 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, 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. Only if $T_{pcie} + T_p \leq T_{tgt}$, the QoS of q is satisfied. 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 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 computational resource quotas, 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.

6.2 Allocating resource for non-QoS tasks

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

When 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 [14], 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 i-th non-QoS task, $f(x_i)$ and $g(x_i)$ are the predicted IPC and the predicted global memory bandwidth usage of the i-th non-QoS task when it is allocated x_i computational resource quota respectively.

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

Many algorithms can be applied to solve the optimization problem. However, it is time-consuming to resolve a continuous optimization problem [29]. To reduce the time needed to fix the issue, we discretize computational resources that can be allocated to different tasks and turn the continuous optimization problem into a discrete optimization problem, thus significantly reduce the computational complexity to identify the appropriate computational resource allocation. The assumption we made for discretizing computational resources is that: the accelerator 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 are allocated to this task.

The discrete optimization problem can be further modeled to be a complete knapsack problem [39]. Let N_{free} represent the quota of computational resources that is 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. The weight of an item 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 using 0/1 knapsack problem. In the equation, $V[i][j]$ is the maximum value of the items in the backpack when the capacity of the backpack is j and puts i items in the backpack, m is the weight of the i-th item, $IPC_{i,m}$ is the achieved IPC of the i-th non-QoS task when it is allocated m quota of computational resources. We adopt the dynamic programming technique to resolve this complete knapsack problem.

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

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 executed when other tasks release some computational resource quota. 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.

To solve this problem, we propose *process pool* technique in Figure 10. As shown in the figure, Laius launches a pool of processes that are configured to use different computational resource quota.

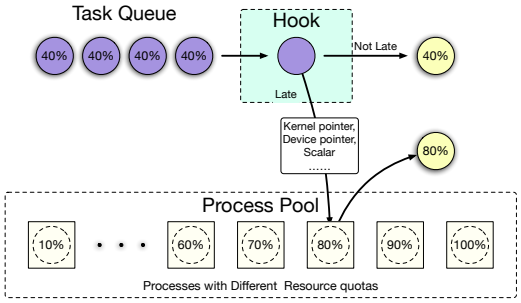


Figure 10: Enabling resource reallocation using process pool.

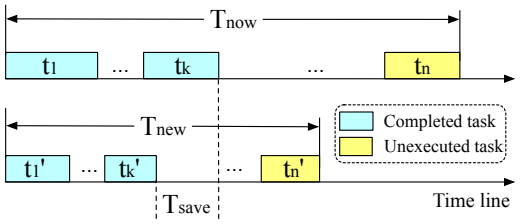


Figure 11: Identifying a new computational resource quota for a user-facing query to compensate the processing lag.

For a process p (executing a user-facing query or a batch application), 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. 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 address space of the remote process, 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.

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, ..., t_k represent the predicted duration, and t_1^r, ..., 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.

Table 2: Benchmarks used in the experiment.

Benchmark Suite	Workloads
Tonic suite [20]	face, dig, imc, ner, pos, chk
Rodinia [9]	BFS, B+tree, PF (pathfinder), NW, LUD HS (hotspot), MD (lavaMD), MY (myocyte)

Figure 11 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 new identified computational 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 i-th completed task with the new resource quota. Observed from the equation, $T_{now} - T_{new}$ calculates the overall reduced duration of query q with the new resource quota. $T_{save} = \sum_{i=1}^{k}(t_i - t_i')$ calculates 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 the 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.

$$T_{now} - T_{new} - \sum_{i=1}^{k}(t_i - t_i') \geq \sum_{i=1}^{k}(t_i^r - t_i) \quad (6)$$

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, Laius adopts 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 is 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, Laius ensures that q completes before the QoS target, and minimizing the resource used by it.

8 EVALUATION OF LAIUS

8.1 Experimental Setup

Benchmarks. We use benchmarks in Tonic suite [20] as user-facing services and use benchmarks in Rodinia benchmark suite [9] as batch applications. Table 2 gives a brief description of the benchmarks. In our experiments, we use 6 user-facing servicesfrom Tonic suite as user-facing services; use 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 [25].

Hardware and software. The experiments are carried out on a machine equipped with one Nvidia GPU RTX 2080Ti. The detailed setups are summarized in Table 3. Note that 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

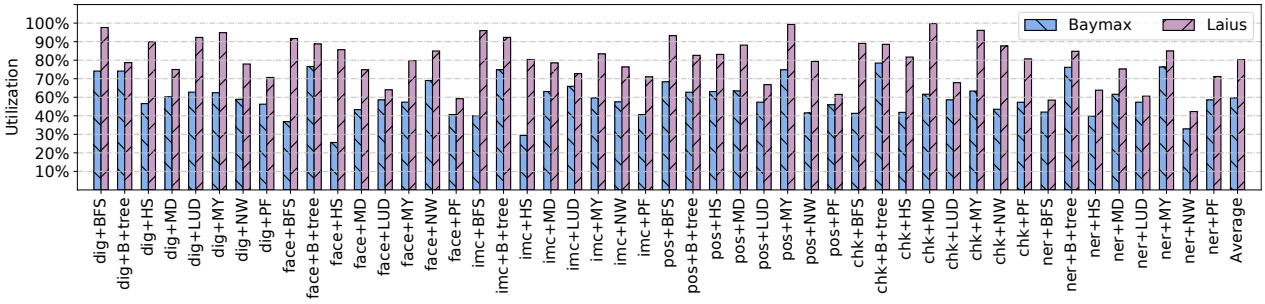


Figure 12: The normalized throughput of batch applications at co-location with Baymax and Laius.

Table 3: Hardware and software specifications.

	Specification
Hardware	Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz GeForce RTX 2080Ti
Software	Ubuntu 16.04.5 LTS with kernel 4.15.0-43-generic CUDA Driver 410.78 CUDA SDK 10.0 CUDNN 7.4.2

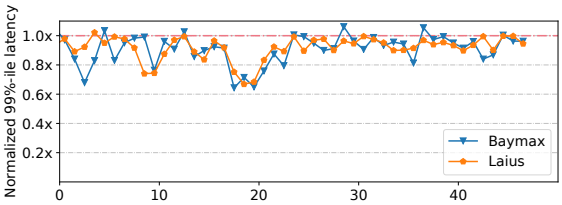


Figure 13: The 99%-ile latency of user-facing services normalized to their QoS targets in the 48 co-locations.

Turing architecture Whitepaper, we can see that the maximum global memory bandwidth provided by 2080Ti is 616GB/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 applications normalized to their throughput when they run alone on the experimental platform.

8.2 QoS and Throughput

In this section, we evaluate the effectiveness of Laius in maximizing the accelerator throughput while ensuring the QoS of requirement emerging user-facing tasks. We compare Laius with Baymax [11], a runtime system that improves accelerator utilization while ensuring the QoS of user-facing services for traditional non-preemptive time-sharing accelerators. Baymax predicts the duration of every task and reserves enough GPU time slices for user-facing queries. If the duration 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 application to use 100% of computational resources for Baymax, to set up an environment that it works.

Figure 13 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 this figure, both Laius and Baymax ensure the QoS of user-facing services. On the contrary, the even allocation and priority allocation in Figure 2 results in QoS violation of user-facing services.

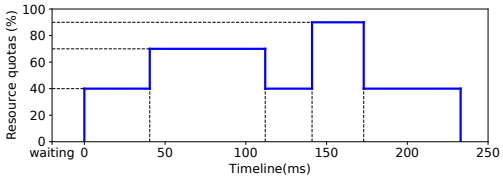


Figure 14: The change of resource quota allocated to a user-facing query *imc* when it is co-located with *BFS*.

Figure 12 shows the normalized throughput of batch applications at co-location with Baymax and Laius. Observed from this figure, batch applications in all the $6 \times 8 = 48$ co-locations achieve higher throughput with Laius than Baymax. Specifically, Laius achieves the throughput of batch applications 70.3%, while Baymax achieves the throughput of batch applications 49.5% on average. Laius improves the throughput of batch applications by 20.8% at co-location compared with Baymax.

Laius can allocate computational resources to run different tasks simultaneously while Baymax runs tasks sequentially. Adopt spatial multitasking, Laius can squeeze more computational resource to execute batch applications. Space sharing often convey better resource utilization than time sharing when a single task cannot fully utilize all the resources [43].

As an example, Figure 14 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% 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 the case, the compensator calculates it a new resource quota 60%, and process the remaining tasks of the query using the new quota. Later, the progress of q is not lag behind expected, and the quota rolls back to the original 40%. In this way, Laius ensures that the query *face* completes before the QoS target, and minimizing the resource used by query q.

8.3 Effectiveness of Constraining the Global Memory Bandwidth Contention

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

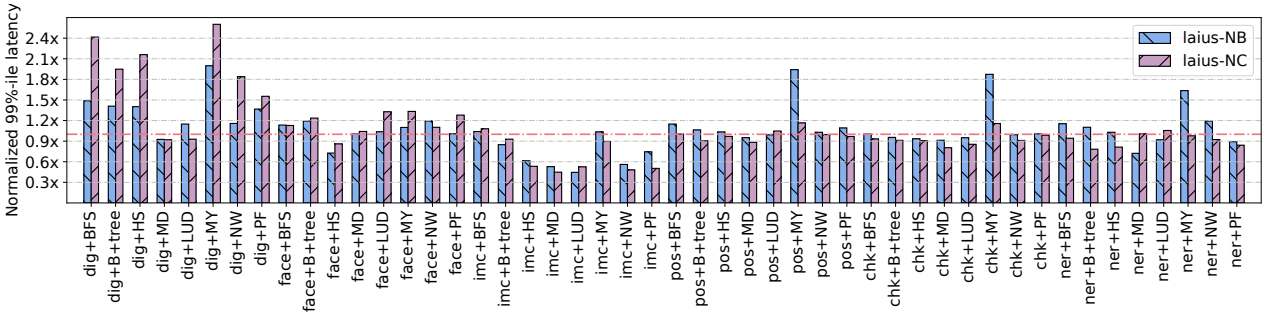


Figure 15: The 99%-ile latency of user-facing services normalized to the QoS target with Laius-NB and Laius-NC.

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 Laius with Laius-NB, a system that disables the global memory bandwidth constraints when allocating computational resources to non-QoS tasks.

Figure 15 presents the 99%-ile latency of user-facing services at co-location in Laius-NB. Observed from this figure, user-facing services in 25 out of the 48 co-locations suffer from QoS violation in Laius-NB. For instance, *dig* suffers from up to 2X QoS violation when it is colocated with *MY*. 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 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, Laius-NB is enough to ensure the QoS of user-facing services if the co-located applications are not global memory bandwidth intensive applications.

8.4 Effectiveness of the Lag Compensator

In this section, we verify the need for the compensation mechanism. We remove the compensation part of laius and test the system. The figure 15 shows the existence of QoS violation in 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.

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, Figure 15 also presents the 99%-ile latency of user-facing services at co-location in Laius-NC, a system that disables the lag compensator in Laius.

Observed from Figure 15, user-facing services in 18 out of the 48 co-locations suffer from QoS violation in 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.

8.5 Overhead of Laius

The main overhead of Laius is from the process pool. Firstly, 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 by accurately scheduling, other than switching between processes with different quota frequently. Secondly, we adopt the CUDA IPC and other techniques to share GPU resources between processes, which consume extra time 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. Overall, switching the execution resource quota of the task through the process pool we designed, the overhead is less than 4%.

9 CONCLUSION

Laius improves the hardware utilization in spatial multitasking accelerators while guaranteeing the QoS requirement of user-facing applications. To achieve this purpose, Laius enables precise task duration prediction, contention-aware resource allocation, and progress-aware lag compensator. Through evaluating Laius with emerging user-facing services, we demonstrate the effectiveness of Laius in eliminating QoS violation due to insufficient computational resource, global memory bandwidth contention, and contentions on other unmanaged shared resources. Laius improves the throughput of batch applications at co-location by 20.8% on average compared with state-of-the-art techniques, without violating the QoS of 99%-ile latency for user-facing services.

10 ACKNOWLEDGEMENT

This work is partially sponsored by the National R&D Program of China(No. 2018YFB1004800), the National Natural Science Foundation of China(NSFC)(61602301,61632017,61702328,61872240), an NSF CAREER Award(CNS-1750760) and CCF-1823005.

REFERENCES

[1] [n. d.]. Apple Siri. https://www.apple.com/siri/.
[2] [n. d.]. Google Translate. https://translate.google.com/.
[3] [n. d.]. Nvidia Night Compute. https://docs.nvidia.com/nsight-compute/NsightCompute/index.html.
[4] [n. d.]. Prisma. https://prisma-ai.com/.
[5] Jacob T Adriaens, Katherine Compton, Nam Sung Kim, and Michael J Schulte. 2012. The case for GPGPU spatial multitasking. In *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on*. IEEE, 1–12.
[6] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. 2013. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis lectures on computer architecture* 8, 3 (2013), 1–154.
[7] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. 2013. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis lectures on computer architecture* 8, 3 (2013), 1–154.
[8] Andrei Broder. 2002. A taxonomy of web search. In *ACM Sigir forum*, Vol. 36. ACM, 3–10.
[9] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*. Ieee, 44–54.
[10] Quan Chen, Hailong Yang, Minyi Guo, Ram Srivatsa Kannan, Jason Mars, and Lingjia Tang. 2017. Prophet: Precise qos prediction on non-preemptive accelerators to improve utilization in warehouse-scale computers. *ACM SIGARCH Computer Architecture News* 45, 1 (2017), 17–32.
[11] Quan Chen, Hailong Yang, Jason Mars, and Lingjia Tang. 2016. Baymax: Qos awareness and increased utilization for non-preemptive accelerators in warehouse scale computers. *ACM SIGARCH Computer Architecture News* 44, 2 (2016), 681–696.
[12] Yunji Chen, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. 2016. DianNao family: energy-efficient hardware accelerators for machine learning. *Commun. ACM* 59, 11 (2016), 105–112.
[13] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759* (2014).
[14] Carlos A Coello Coello. 2000. Treating constraints as objectives for single-objective evolutionary optimization. *Engineering Optimization+ A35* 32, 3 (2000), 275–308.
[15] Jeffrey Dean and Luiz André Barroso. 2013. The tail at scale. *Commun. ACM* 56, 2 (2013), 74–80.
[16] Jeffrey Dean and Luiz André Barroso. 2013. The tail at scale. *Commun. ACM* 56, 2 (2013), 74–80.
[17] Christina Delimitrou and Christos Kozyrakis. 2013. Paragon: QoS-aware scheduling for heterogeneous datacenters. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 77–88.
[18] Christina Delimitrou and Christos Kozyrakis. 2014. Quasar: resource-efficient and QoS-aware cluster management. *ACM SIGPLAN Notices* 49, 4 (2014), 127–144.
[19] Glenn A Elliott, Bryan C Ward, and James H Anderson. 2013. GPUSync: A framework for real-time GPU management. In *2013 IEEE 34th Real-Time Systems Symposium*. IEEE, 33–44.
[20] Johann Hauswald, Yiping Kang, Michael A Laurenzano, Quan Chen, Cheng Li, Trevor Mudge, Ronald G Dreslinski, Jason Mars, and Lingjia Tang. 2015. DjiNN and Tonic: DNN as a service and its implications for future warehouse scale computers. In *Computer Architecture (ISCA), 2015 ACM/IEEE 42nd Annual International Symposium on*. IEEE, 27–40.
[21] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. 2013. *An introduction to statistical learning*. Vol. 112. Springer.
[22] Nicola Jones. 2014. Computer science: The learning machines. *Nature News* 505, 7482 (2014), 146.
[23] Jithin Jose, Hari Subramoni, Miao Luo, Minjia Zhang, Jian Huang, Md Wasi-ur Rahman, Nusrat S Islam, Xiangyong Ouyang, Hao Wang, Sayantan Sur, et al. 2011. Memcached design on high performance rdma capable interconnects. In *2011 International Conference on Parallel Processing*. IEEE, 743–752.
[24] Shinpei Kato, Karthik Lakshmanan, Raj Rajkumar, and Yutaka Ishikawa. 2011. TimeGraph: GPU scheduling for real-time multi-tasking environments. In *Proc. USENIX ATC*. 17–30.
[25] Raphael Landaverde, Tiansheng Zhang, Ayse K Coskun, and Martin Herbordt. 2014. An investigation of unified memory access performance in cuda. In *High Performance Extreme Computing Conference (HPEC), 2014 IEEE*. IEEE, 1–6.
[26] Haeseung Lee, Al Faruque, and Mohammad Abdullah. 2014. GPU-EvR: Run-time event based real-time scheduling framework on GPGPU platform. In *Proceedings of the conference on Design, Automation & Test in Europe*. European Design and Automation Association, 220.
[27] Teng Li, Vikram K Narayana, and Tarek El-Ghazawi. 2015. Reordering GPU kernel launches to enable efficient concurrent execution. *arXiv preprint arXiv:1511.07983* (2015).
[28] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. 2015. Heracles: Improving resource efficiency at scale. In *ACM SIGARCH Computer Architecture News*, Vol. 43. ACM, 450–462.
[29] Szymon Łukasik and Sławomir Żak. 2009. Firefly algorithm for continuous constrained optimization tasks. In *International conference on computational collective intelligence*. Springer, 97–106.
[30] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. 2011. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *Proceedings of the 44th annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 248–259.
[31] NVIDIA. 2012. Sharing a GPU between MPI processes: multi-process service(MPS).
[32] NVIDIA. 2015. Multi-Process Service. https://docs.nvidia.com/deploy/mps/index.htmltopic_6_1_2.
[33] CUDA Nvidia. 2008. Cublas library. *NVIDIA Corporation, Santa Clara, California* 15, 27 (2008), 31.
[34] C Nvidia. 2012. Nvidias next generation cuda compute architecture: Kepler gk110. *Whitepaper (2012)* (2012).
[35] Sreepathi Pai, Matthew J Thazhuthaveetil, and Ramaswamy Govindarajan. 2013. Improving GPGPU concurrency with elastic kernels. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 407–418.
[36] Jason Jong Kyu Park, Yongjun Park, and Scott Mahlke. 2017. Dynamic resource management for efficient utilization of multitasking GPUs. *ACM SIGOPS Operating Systems Review* 51, 2 (2017), 527–540.
[37] Vinicius Petrucci, Michael A Laurenzano, John Doherty, Yunqi Zhang, Daniel Mosse, Jason Mars, and Lingjia Tang. 2015. Octopus-man: Qos-driven task management for heterogeneous multicores in warehouse-scale computers. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 246–258.
[38] S Rasoul Safavian and David Landgrebe. 1991. A survey of decision tree classifier methodology. *IEEE transactions on systems, man, and cybernetics* 21, 3 (1991), 660–674.
[39] Sartaj Sahni. 1975. Approximate algorithms for the 0/1 knapsack problem. *Journal of the ACM (JACM)* 22, 1 (1975), 115–124.
[40] George AF Seber and Alan J Lee. 2012. *Linear regression analysis*. Vol. 329. John Wiley & Sons.
[41] Yusuke Suzuki, Shinpei Kato, Hiroshi Yamada, and Kenji Kono. 2014. GPUvm: Why not virtualizing GPUs at the hypervisor?. In *USENIX Annual Technical Conference*. 109–120.
[42] Lingjia Tang, Jason Mars, and Mary Lou Soffa. 2012. Compiling for niceness: Mitigating contention for qos in warehouse scale computers. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*. ACM, 1–12.
[43] Zhenning Wang, Jun Yang, Rami Melhem, Bruce Childers, Youtao Zhang, and Minyi Guo. 2016. Simultaneous multikernel GPU: Multi-tasking throughput processors via fine-grained sharing. In *High Performance Computer Architecture (HPCA), 2016 IEEE International Symposium on*. IEEE, 358–369.
[44] Bo Wu, Xu Liu, Xiaobo Zhou, and Changjun Jiang. 2017. FLEP: Enabling flexible and efficient preemption on GPUs. *ACM SIGOPS Operating Systems Review* 51, 2 (2017), 483–496.
[45] Hailong Yang, Alex Breslow, Jason Mars, and Lingjia Tang. 2013. Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers. In *ACM SIGARCH Computer Architecture News*, Vol. 41. ACM, 607–618.
[46] Junbo Zhang, Yu Zheng, Dekang Qi, Ruiyuan Li, and Xiuwen Yi. 2016. DNN-based prediction model for spatio-temporal data. In *Proceedings of the 24th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. ACM, 92.
[47] Yunqi Zhang, Michael A Laurenzano, Jason Mars, and Lingjia Tang. 2014. Smite: Precise qos prediction on real-system smt processors to improve utilization in warehouse scale computers. In *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*. IEEE, 406–418.
[48] Wenyi Zhao, Quan Chen, and Minyi Guo. 2018. KSM: Online Application-Level Performance Slowdown Prediction for Spatial Multitasking GPGPU. *IEEE Computer Architecture Letters* 17, 2 (2018), 187–191.