Baymax: QoS Awareness and Increased Utilization for Non-Preemptive Accelerators in Warehouse Scale Computers

Quan Chen*†1 Hailong Yang*‡1 Jason Mars* Lingjia Tang*

*Clarity Lab, University of Michigan - Ann Arbor, MI, USA
†Shanghai Jiao Tong University, Shanghai, China
‡Beihang University, Beijing, China
{quanchen, hailong, profmars, lingjia}@umich.edu

Abstract

Modern warehouse-scale computers (WSCs) are being outfitted with accelerators to provide the significant compute required by emerging intelligent personal assistant (IPA) workloads such as voice recognition, image classification, and natural language processing. It is well known that the diurnal user access pattern of user-facing services provides a strong incentive to co-locate applications for better accelerator utilization and efficiency, and prior work has focused on enabling co-location on multicore processors. However, interference when co-locating applications on non-preemptive accelerators is fundamentally different than contention on multi-core CPUs and introduces a new set of challenges to reduce QoS violation.

To address this open problem, we first identify the underlying causes for QoS violation in accelerator-outfitted servers. Our experiments show that queuing delay for the compute resources and PCI-e bandwidth contention for data transfer are the main two factors that contribute to the long tails of user-facing applications. We then present Baymax, a runtime system that orchestrates the execution of compute tasks from different applications and mitigates PCI-e bandwidth contention to deliver the required QoS for user-facing applications and increase the accelerator utilization. Using DjiNN, a deep neural network service, Sirius, an end-to-end IPA workload, and traditional applications on a Nvidia K40 GPU, our evaluation shows that Baymax improves the accelerator utilization by 91.3% while achieving the desired 99%-ile latency target for for user-facing applications. In fact, Baymax reduces the 99%-ile latency of user-facing applications by up to 195x over default execution.

Keywords scheduling; quality of service; warehouse scale computers; non-preemptive accelerators

ASPLOS '16, April 02-06, 2016, Atlanta, GA, USA
Copyright © 2016 ACM 978-1-4503-4091-5/16/04...$15.00
DOI: http://dx.doi.org/10.1145/2872362.2872368

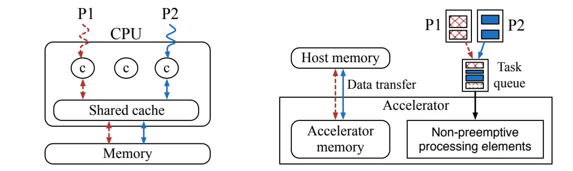
1. Introduction

Emerging intelligent personal assistant (IPA) workloads including speech recognition [1, 2], image classification [3], face recognition [4] and natural language processing [5, 6] have recently gained tremendous momentum. Several major Internet-service companies including Google [7], Microsoft [8], Apple [9] and Baidu [10] have all released their IPA services providing a wide range of features. Compared to traditional warehouse scale computer (WSC) applications such as web-search, IPA applications are significantly more computationally demanding [11].

Accelerators, such as GPUs, ASICs and FPGAs, have been shown to be particularly suitable for these IPA applications from both performance and total cost of ownership (TCO) perspectives [11]. Therefore, to satisfy the ever-growing user demand at a low cost, datacenters have recently adopted accelerator-outfitted servers for these applications [12, 13]. Meanwhile, since these IPA services generally experience diurnal pattern [14, 15] (leaving the accelerator resources under-utilized for most of the time except peak hours), it is more cost efficient to co-locate user-facing applications and throughput-oriented applications on accelerators. However, accelerator sharing introduces varying amount of performance interference between co-located applications, and thus poses critical challenges for guaranteeing that user-facing applications can meet their quality of service targets.

There has been a significant amount of prior work recognizing the importance, and addressing the problem, of contention due to co-locations to enforce quality of service

[1] Work was conducted as a postdoc fellow of Clarity Lab at the University of Michigan.



(a) Interference on traditional server (b) Interference on accelerator-outfitted server

Figure 1: Interference between co-located applications on a CMP server and an accelerator-outfitted server. Interference on a traditional server is mainly due to cache and memory bandwidth contention. Interference on an accelerator-outfitted server is caused by both queuing delay for processing elements and PCI-e bandwidth contention.

(QoS) and maximize utilization. However, because traditional datacenter servers use only commodity general purpose processors, these researches have focused exclusively on techniques to predict the QoS interference among co-located applications on multi-core processors [16–22] and simultaneous multi-threading (SMT) processors [23]. These solutions are not adequate for the emerging generation of datacenter architectures that have introduced accelerators as a key element of their design.

Figure 1 compares the performance interference between co-located applications on traditional multicore servers and accelerator-outfitted servers. While performance interference on traditional servers is mainly due to cache and memory bandwidth contention [18, 20, 23], we discover that the performance interference on accelerator-outfitted servers is often caused by queuing delay and PCI-e bandwidth contention. Together they can cause as much as 195x slowdown in terms of the 99%-ile latency for user-facing applications. As shown in Figure 1(b), when a compute task is running on a non-preemptive accelerator, all the following tasks have to wait for its completion before they can get executed. This mechanism introduces severe queuing delay to user-facing services. In addition, the co-located applications contend for the PCI-e bandwidth to transfer data between the host memory and the accelerator memory. The prior techniques based on shared resource contention (e.g., contention on shared cache and memory bandwidth) cannot be applied to this class of non-preemptive hardware that has distinctive data transfer phases.

This paper aims to improve the utilization of accelerator hardware while guaranteeing the required QoS target of user-facing time-sensitive applications in WSCs. We find that four main factors affect the queuing delay and data transfer latency and thus the end-to-end latency of user-facing applications. These factors include the number of tasks in a user-facing query that indicates how many tasks could be delayed, the task execution order that decides which tasks may cause the delay for each user-facing task, the duration and occupancy of throughput-oriented tasks that impact the queuing delay of each task in a user-facing query, as well as the PCI-e bandwidth contention that affects data transfer rate between host memory and accelerator memory.

Because key factors such as the task execution order and PCI-e bandwidth contention may change during runtime, an offline solution is not adequate. A runtime system that can dynamically monitor the accelerator and PCI-e bus, and schedule tasks accordingly is needed to maximize accelerator utilization while satisfying QoS of user-facing applications. To this end, we propose **Baymax**, a runtime system composed of two parts: a *task duration predictor* and a *task re-ordering engine*. The task duration predictor leverages novel models to predict the duration of tasks across different inputs. The task re-ordering engine then intercepts and analyzes task launching function calls before passing control to the accelerator. Based on the precisely predicted task duration, Baymax re-orders compute tasks issued to the accelerator. Meanwhile, Baymax limits the number of concurrent active data transfer tasks to mitigate PCI-e bandwidth contention. By reordering tasks and managing PCI-e bandwidth, Baymax guarantees that QoS of user-facing applications is always satisfied regardless of the order of tasks issued by applications.

To the best of our knowledge, Baymax is the first work that improves the utilization of non-preemptive accelerators while guaranteeing the QoS of user-facing applications on real systems. Specifically, this paper makes the following contributions:

1. **Comprehensive analysis of QoS violation on non-preemptive accelerators -** We identify four key factors that significantly affect the end-to-end latency of user-facing applications when they are co-located with other applications. The analysis motivates the design of a task re-ordering system based on precisely predicted task duration for accelerator co-locations.
2. **Design of online task duration prediction models -** We establish accurate and low-overhead models to estimate the duration of tasks on accelerators.
3. **Design of a task re-ordering mechanism to manage accelerator tasks -** We design a task re-ordering mechanism that intercepts and re-orders task invocations from both user-facing and throughput-oriented applications. The mechanism trades off QoS headroom of user-facing applications for increased accelerator utilization while guaranteeing the satisfactory QoS.
4. **Design of an online mechanism to mitigate PCI-e bandwidth contention -** We design a mechanism that monitors the realtime data transfer pressure on PCI-e bus and mitigates PCI-e bandwidth contention to eliminate QoS violation.

We implement Baymax runtime system combining all the above techniques. Our evaluation using Nvidia K40 GPU demonstrates that Baymax can greatly increase the utilization of accelerators by 91.3% while guaranteeing the 99%-ile latency of user-facing applications within the QoS target.

Compared with the default scheduling, Baymax reduces the tail latency of user-facing applications by up to 195x when co-located with throughput-oriented applications.

2. Understanding Performance Interference on Non-Preemptive Accelerators

We refer to accelerators that do not support context switching during kernel execution (such as ASICs, FPGAs and GPUs) as *non-preemtive*. In this section, we seek to answer the following research questions.

- Is there serious performance interference for user-facing applications when co-located with throughput-oriented applications on non-preemptive accelerators?
- What are the root causes of long tail latency when a user-facing application is co-located with other throughput-oriented applications?
- What can we do to improve the accelerator utilization while guaranteeing that user-facing applications achieve the desired QoS for tail latency?

2.1 Real System Setup

We use the GPU as our non-preemptive accelerator platform throughout this work. Our real system study uses both user-facing applications and throughput-oriented applications. User-facing applications, such as emerging IPA application Sirius [11] and deep neural network service DjiNN [24], run as permanent services on the accelerator, accepting user queries and returning the results with stringent QoS requirement. Throughput-oriented applications on the other hand do not have QoS requirement but only require high throughput. Both user-facing applications and throughput-oriented applications consist of various number of tasks (*kernels* and *memcpy tasks*[1]), and the duration of each task also varies across applications. In this experiment, multiple user-facing applications and throughput-oriented applications submit kernels and memcpy requests to GPU simultaneously. The details on the platform and benchmarks can be found in Section 7.1.

2.2 Long Tail Latency and Low Utilization

Interference between co-located applications often incurs long tail latency for user-facing applications. Figure 2 shows the QoS violation when a user-facing application is co-located with throughput-oriented applications on a Nvidia K40 GPU. In the figure, the x-axis indicates the combination of user-facing application and throughput-oriented application, and the y-axis shows the 99%-ile latency of the user-facing applications normalized to its QoS target (150 milliseconds [15, 25]). The left part of the figure and the right part (shadowed part) of the figure show the results when a user-facing application is co-located with compute intensive throughput-oriented applications and PCI-e intensive throughput-oriented applications, respectively.

[1] A task that runs on processing elements is refer as a *kernel* and a task that transfers data through PCI-e bus is refer as a *memcpy task*.

MPS (Multi-Process Service) scheduling [26] enables concurrent sharing of a GPU among multiple applications. As shown in Figure 2(a), the 99%-ile latency of user-facing queries in 40 out of the 88 co-locations is much larger than the expected QoS target with default MPS scheduling. The 99%-ile latency of user-facing applications is 10.8x of the QoS target on average and up to 195.9x in the worst case.

Priority-based scheduling [27] used in TimeGraph [28] and GPUSync [29] for improving performance of realtime kernels on accelerators executes high priority kernels first if multiple kernels are ready to run. Adopting priority-based scheduling, as shown in Figure 2(b), user-facing applications in 33 out of the 88 co-locations still suffer from QoS violation by 1.6x on average (up to 5.2x in the worst case).

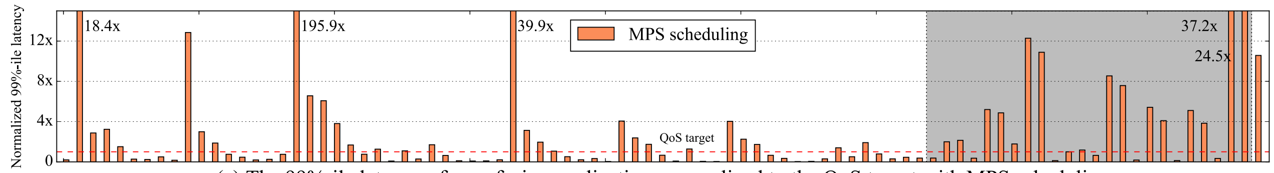
The reason priority-based scheduling polices are not capable to guarantee the QoS of user-facing applications (high priority) is that they are not aware of the duration of tasks. Whenever a user-facing application is not submitting kernels to GPU due to stalls such as CPU synchronization, kernels of throughput-oriented applications may take over the GPU resource with long duration and high occupancy. Because emerging accelerators (e.g., GPU) are non-preemptive, even if a user-facing kernel becomes ready right after the submission of the long throughput-oriented kernel, the user-facing kernel would not be executed until the previous kernel completes. In this case, long queuing delay is added to the user-facing kernel, risking QoS violations.

Meanwhile, as shown in Figure 2(c), the end-to-end latency of user-facing queries in some other co-locations is much smaller than the acceptable QoS target while the GPU utilization is low. Always prioritizing user-facing kernels even if the latency is much smaller than the QoS target wastes the opportunity to improve the utilization. If the kernels are scheduled properly, the QoS headroom can be leveraged for higher utilization.

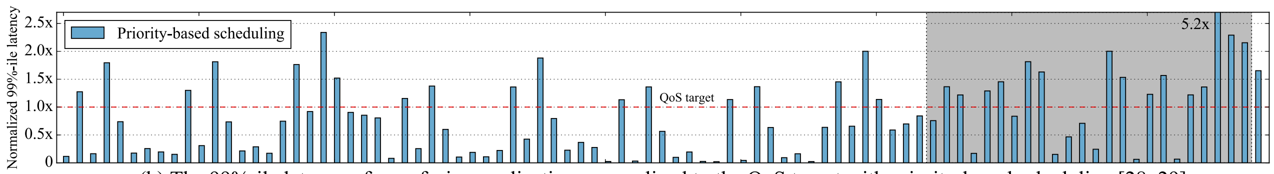
2.3 Root Causes of Long Tail Latency

In order to show the root causes of long tail latency on non-preemptive accelerators, Figure 3 presents two task execution timelines captured with *nvprof* [30] when co-locating *face* (user-facing) and four compute intensive application *hw*, *stemmer* (user-facing) and four PCI-e intensive application *pf* (details on benchmarks are shown in Table 3). Note that the overlapping of green bars in Figure 3(a) is not kernel preemption, but concurrent kernel execution when using MPS. From the figure we observe that four factors may impact the tail latency of a user-facing application when it is co-located with other applications.

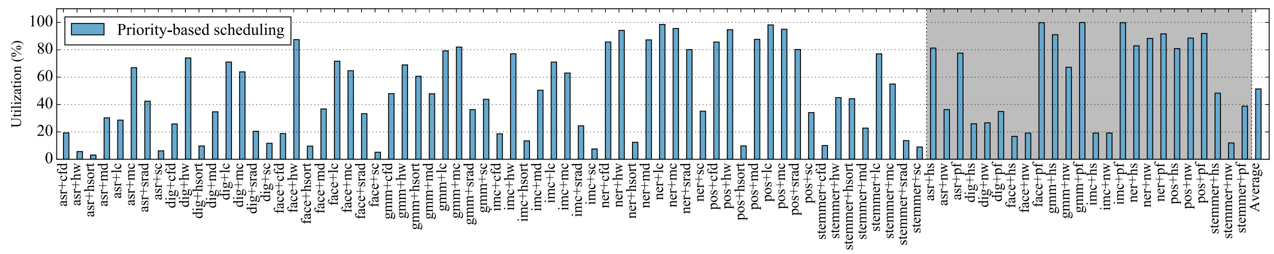
The duration and occupancy of kernels - If the occupancy of a kernel is high, MPS is not able to overlap the kernel with its neighbor kernels to boost concurrent kernel execution. In this case, if the duration of throughput-oriented kernels is long, the execution of user-facing kernels will be delayed significantly.



(a) The 99%-ile latency of user-facing applications normalized to the QoS target with MPS scheduling

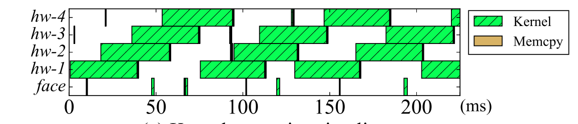


(b) The 99%-ile latency of user-facing applications normalized to the QoS target with priority-based scheduling [28, 29]

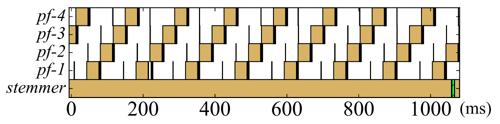


(c) Percentage of GPU time used by throughput-oriented applications with priority-based scheduling

Figure 2: QoS violation of user-facing applications and low GPU utilization at co-locations with default MPS scheduling policy and priority-based scheduling policy.



(a) Kernel execution timeline



(b) Data transfer timeline

Figure 3: Kernel execution timeline on GPU and data transfer timeline between host memory and accelerator memory.

The kernel scheduling order - Accelerators, such as GPUs, schedule kernels in the same order as they arrived (even if neighbor kernels can run concurrently when the kernel occupancy is small). If the co-located throughput-oriented applications submit kernels frequently, the user-facing application will be delayed by a large amount of throughput-oriented kernels.

The number of kernels in a user-facing query - The more kernels a user-facing query has, the longer its tail latency could be, because every kernel in the user-facing query can be delayed by throughput-oriented kernels. For example, as shown in Figure 3(a), every kernel of *face* is delayed by at least two kernels of *hw*.

The contention on PCI-e bandwidth - If throughput-oriented applications consume high PCI-e bandwidth, user-facing applications may suffer from slow data transfer due to the contention on PCI-e bandwidth. For example, as shown in Figure 3(b), the memcpy task of *stemmer* is severely slowed down to more than 1000 milliseconds from only 15 milliseconds when it is running alone. This slow down in turn results in long tail latency.

2.4 Design Guidelines of Baymax

Based on the identified root causes of long tail latency, to improve the utilization of non-preemptive accelerator while guaranteeing the QoS of user-facing applications, we design and implement Baymax following four guidelines.

- Baymax should be able to predict the duration of each kernel and memcpy task. In this case, Baymax can quantify the impact of each task on the end-to-end latency of user-facing applications.
- Baymax should be able to re-order all the kernels issued to the same accelerator, no matter how they are submitted by the co-located applications.
- For a user-facing query, Baymax should be able to limit the overall time delayed by the co-located applications regardless of the number of kernels in the query.
- Baymax should be able to monitor realtime data transfer pressure on PCI-e bus and mitigate PCI-e bandwidth contention.

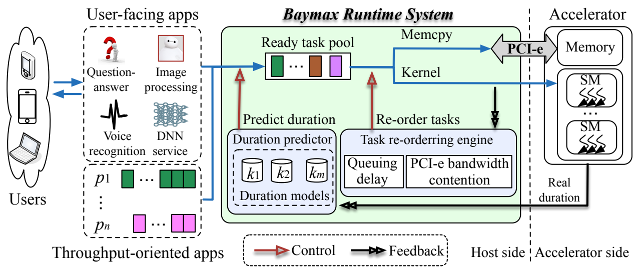


Figure 4: Design of Baymax.

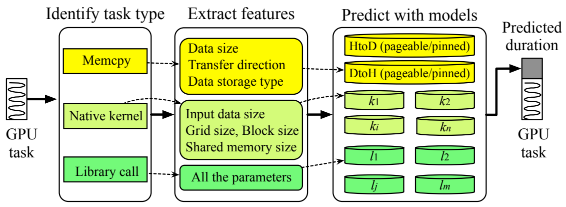


Figure 5: Predict the duration of GPU tasks (memcpy, native kernel, and library call).

3. Baymax Methodology

Figure 4 presents the design overview of Baymax. Limited by the existing GPU design, there is no open interface to schedule tasks that are already launched to the GPU. We therefore design a mechanism to re-order tasks on the CPU side. If such interface is provided in the future, Baymax can be implemented directly on accelerators. The re-ordering decision is based on the QoS target of user-facing applications and the predicted duration of each task.

In Baymax, all the tasks submitted to the accelerator are first pushed into a ready task pool managed by Baymax on the CPU side. This is achieved by simple automatic instrumentation of the original task submission code. The task submission rerouting APIs can be provided to programmers to submit tasks through Baymax. When a task is pushed into the ready task pool, the task duration predictor first predicts its duration leveraging regression models (Section 4).

The *task re-ordering engine* periodically iterates over all the tasks in the ready task pool and decides whether each task can be launched to GPU. If the task is a kernel and its predicted duration is larger than the realtime QoS headroom of any active user-facing query, the kernel will stay in the ready task pool. Otherwise, the kernel is launched to GPU (Section 5). On the other hand, if the task is a memcpy task, the engine decides whether to launch the task based on realtime data transfer pressure on PCI-e bus (Section 6).

Baymax incorporates a feedback mechanism to update the duration models used by the task duration predictor. Once a task completes, the actual duration of the task is fed back to the duration predictor to update its duration model.

4. Task Duration Modeling

In this section, we present the modeling methodology used to predict the duration of GPU tasks.

4.1 Task Duration Predictor

Baymax builds duration models for three types of GPU tasks: *memcpy*, *native kernel*, and *library call*. Native kernels are the kernels defined by programmers. Besides writing their own native kernels, an application can also call APIs defined in highly-optimized GPU libraries (e.g., cuDNN [31] and cuBLAS [32]). For the three types of GPU tasks, Figure 5 shows the methodology to predict their duration.

When a task is submitted to Baymax, the duration predictor identifies the type of the task, extracts the representative features, and selects pre-trained duration model according to the name of the GPU task (function name for native kernels and API name for library calls). Once the duration model is found, Baymax predicts the duration of the task using the extracted features and the model, and attaches the predicted duration to the task. After that, the task is pushed into the ready task pool waiting for launching to GPU.

4.2 Selecting Representative Features

It is challenging to predict the duration of GPU tasks because there is very limited information we can obtain at runtime. Although *nvprof* [30] provides comprehensive performance metrics after measuring the entire task execution, no performance information can be accessed before the task is executed. It is not applicable to rely on these metrics to predict the duration of GPU tasks.

Table 1: Features selected for different types of tasks.

Task Type	Features	Dimension
Memcpy	Data size	1
	Data transfer direction	1
	Data storage (pageable/pinned [33])	1
Native kernel	Input data size	1
	Grid size ($X \times Y \times Z$)	3
	Block size ($X \times Y \times Z$)	3
	Size of required shared memory	1
Library call	All the parameters	/

The only information we can obtain before a GPU task is executed includes its configurations (e.g., grid size, block size etc.) and the parameters passed to the task. We further select the information that strongly impact a task's duration on GPU (e.g., input scale and task configurations) as representative features. Empirically, to capture the correlation between features of a GPU task and its duration, as listed in Table 1, we select different features for different types of GPU tasks.

For a memcpy task, we select data size, data transfer direction and data storage type as its representative features. Data to be transferred from/to GPU can be stored in either *pageable memory* or *pinned memory* [33]. It is much faster (around 4x faster) to transfer data from pinned memory compared with pageable memory while more time is needed to initialize pinned memory when it is allocated.

For a native kernel, we use kernel configuration and input data size as the features. The grid size and the block size determine the scale of thread level parallelism on GPU and the GPU occupancy of the kernel, which significantly affect its duration; the size of required shared memory (both *static shared memory* and *dynamic shared memory*) reflects the efficiency of the kernel leveraging the memory hierarchy on GPU. We train different duration models for native kernels executing different functions, because they often have totally different characteristics.

A library call may consist of multiple kernels, while the actual kernels and their configurations are hidden behind the API. Therefore, we treat all the kernels in a library call as a whole, and use all the parameters of the API as its representative features. For several widely used libraries (i.e., cuBLAS and cuDNN), we only need to train models for them once and use the models in all applications.

Besides fine-grained GPU tasks, the duration predictor also predicts the solo-run duration of each user-facing query when the query is first launched. For a user-facing query, we select its input data size as its representative feature.

4.3 Low Overhead Prediction Models

The QoS target of a user-facing query is in the granularity of hundreds of milliseconds to support smooth user interaction [15, 25]. Therefore, choosing the modeling techniques with low computation complexity and high prediction accuracy for the online duration predictor becomes critical.

We evaluated a spectrum of widely used prediction models (e.g., *Linear Regression* (LR) [34], *Approximate Nearest Neighbor* (ANN) [35], *K-Nearest Neighbor* (KNN) [36] and *Support Vector Machines* (SVM) [37]) to predict task duration and eventually selected LR and KNN for their high accuracy and low overhead. While LR assumes the linear relationship between input and output variables, KNN regression holds no such assumption. Therefore using both LR and KNN allows us to achieve accurate prediction for both linear and non-linear relations. Other evaluated models either require longer calculation time with no accuracy improvement (e.g., SVM), or cannot provide satisfactory accuracy (e.g., ANN). Both KNN and LR achieve low prediction overhead. According to our measurement on real hardware, the duration prediction overhead with KNN model and LR model in Baymax is under 0.05 millisecond.

Suppose a task has p representative features. Let X_i represent an input sample with p features $(x_1, x_2, ..., x_p)$, and n represent the total number of input samples $(i = 1, 2, ..., n)$. The linear regression model is defined as Equation 1, and the Euclidean distance for KNN model between sample X_i and X_l $(l = 1, 2, ..., n)$ is defined as Equation 2. In our case, the input is the task features and the output is the predicted task duration. The primary computation of KNN is to calculate the Euclidean distance between the predicting and training samples, which can be accelerated with different tree searching algorithms such as K-D tree and ball tree. We pick the most efficient KNN searching algorithm when training prediction model according to the number of samples and the number of features in every sample.

$$y_i = \beta_1 x_{i1} + ... + \beta_p x_{ip} + \varepsilon_i, i = 1, ..., n \tag{1}$$

$$d(X_i, X_l) = \sqrt{(x_{i1} - x_{l1})^2 + ... + (x_{ip} - x_{lp})^2} \tag{2}$$

4.4 Minimizing Prediction Error

To achieve high prediction accuracy, we apply both KNN and LR to each task in both user-facing and throughput-oriented applications, and choose the model that fits the data most to predict the task duration at runtime. As shown in Section 7.2, LR model and KNN model achieve different prediction accuracy for user-facing applications and throughput-oriented applications respectively. Since the duration models are trained offline with the profiled performance samples from the workloads, more sample data is usually effective to improve the accuracy of the duration models. Especially, in WSCs, the workloads become stable after certain time scale and the models become more accurate with periodical updates. Moreover, the duration predictor detects the prediction deviation at runtime. If the deviation exceeds a certain threshold, incremental update [38] and parallel update [39] can be applied during runtime with low overhead to refine the duration models, which continuously improves the accuracy of the duration prediction.

5. Task Re-ordering Mechanism

In this section, we describe the mechanism used to re-order native kernels and library calls in Baymax. For ease of description, a kernel can be either a native kernel or a library call in this section.

5.1 Breaking Down the End-to-End Latency

It is important to understand the end-to-end latency breakdown of a user-facing query when it is co-located with other applications before diving into the task re-ordering policy. We first assume the co-located applications do not contend for PCI-e bandwidth (to be mitigated in Section 6).

Figure 6 presents the end-to-end latency breakdown of a user-facing query Q when it is co-located with other applications. The end-to-end latency of a query is the time from the first kernel of the query is issued to the last kernel of the query is returned. As shown in the figure, Q's end-to-end latency is composed of three parts. The first part is the processing time of the queued kernels (black kernels in Figure 6) that are issued before k_1 gets executed (denoted by T_q). The second part is the processing time of Q's own kernels (denoted T_{self}). The last part is the processing time of the kernels (line-filled and white kernels) from the co-located applications between k_1 and k_n (denoted by T_{other}).

5.2 Re-ordering Native Kernels and Library Calls

Let T_{tgt} represent the QoS target of query Q. Only if $T_{self} + T_q + T_{other} \leq T_{tgt}$, Q's QoS is satisfied. In the equation,

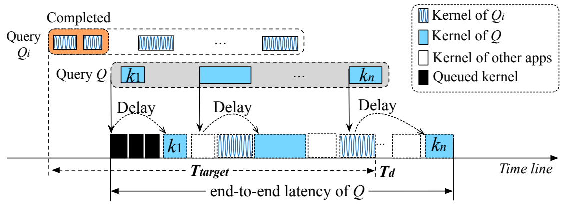


Figure 6: Calculating QoS headroom of Q when its first kernel is launched.

T_{self} is predicted according to the prediction model proposed in Section 4. In this case, to guarantee Q's QoS, the task re-ordering engine in Baymax monitors T_q and reduces T_{other} as follows.

5.2.1 Monitoring Queued Time

To estimate the queuing delay a user-facing query will experience, T_q, Baymax sums up the predicted duration of all the kernels that are already issued to GPU by our re-ordering engine but are not yet executed (still waiting in the GPU queue). Specifically, once a kernel is issued to GPU, we add its predicted duration to T_q, the duration of all the un-executed kernels on GPU. Once a kernel completes, we subtract its predicted duration from T_q.

To eliminate the situation that a user-facing query is significantly delayed by the queued-up kernels on GPU, even if no active user-facing query is running on the GPU, Baymax makes sure that T_q is smaller than the QoS target T_{tgt}. If $T_q > T_{tgt}$, Baymax would not issue any kernel to GPU until some kernels complete. This method would not reduce the GPU utilization because the kernel will be queued up on GPU even if it is issued to GPU.

5.2.2 Calculating QoS Headroom

As discussed above, T_{self} and T_q are known and cannot be reduced when Q is launched. In this case, to guarantee Q's QoS, Baymax makes sure that $T_{other} \leq T_{tgt} - T_{self} - T_q$. We use T_{hr} to represent the free GPU time left for kernels from the co-located applications during the execution of Q (referred as *QoS headroom*). When the first kernel of Q is launched, $T_{hr} = T_{tgt} - T_{self} - T_q$.

Based on T_{hr}, the task re-ordering engine periodically iterates over the ready task pool to check whether each kernel can be safely issued to GPU without causing any QoS violation. Suppose the predicted duration of a kernel is t. If t is larger than T_{hr}, the kernel is delayed until Q completes. On the other hand, if t is smaller than T_{hr}, the kernel is launched to GPU, and at the same time, T_{hr} is reduced by t.

5.2.3 Dealing with Multiple Active User-facing Queries

When multiple user-facing queries are active, more complexity is introduced when calculating the headroom of each user-facing query. Figure 6 describes the method to calculate T_{hr} of query Q when multiple user-facing queries are active. As shown in the figure, if query Q_i is still active when the first kernel of query Q is launched, the un-executed kernels of Q_i have to be completed before T_d so that the QoS of Q_i is satisfied. In this case, when we calculate T_{hr} for Q, the GPU time reserved by the un-executed kernels of Q_i need to be subtracted from T_{tgt} as well. Therefore, we monitor the GPU time each active query still needs to complete the whole query. For Q_i in Figure 6, we estimate Q_i's remaining GPU time by subtracting the time of its completed kernels from its estimated overall GPU time (T_{self} of Q_i).

Suppose there are n active user-facing queries when Q is launched. Let t_1, ..., t_n represent the remaining GPU time required by the n active user-facing queries respectively. Equation 3 calculates Q's QoS headroom when it is issued.

$$T_{hr} = T_{tgt} - T_q - T_{self} - \sum_{i=1}^{n} t_i \tag{3}$$

When multiple queries are active, if the predicted duration of a kernel (denoted by t) is larger than the QoS headroom of any active query, the kernel will be delayed. Otherwise, the kernel is launched and the QoS headroom of each user-facing query is reduced by t.

It is worth noting that Baymax would not result in starvation of any user-facing query even if multiple queries are active concurrently. User-facing kernels are issued in an FIFO order and a throughput-oriented kernel can be issued only when it will not result in QoS violation of any active user-facing query.

5.3 Utilizing Concurrent Kernel Execution

In Section 5.2, we assume that a GPU is not able to concurrently execute multiple kernels. Actually, leveraging emerging MPS technique [26], a GPU is able to execute multiple independent kernels that have low occupancy concurrently.

When concurrent kernel execution happens, T_{hr} calculated in Equation 3 is smaller than the real GPU time available for the co-located applications. In this case, the GPU utilization is not maximized because there is actually more GPU time can be used to process throughput-oriented applications while guaranteeing the QoS of all the active user-facing queries.

To further increase GPU utilization when MPS is enabled, as shown in Figure 7, when kernel k_i of Q is submitted to the ready task pool, Baymax updates the QoS headroom of Q. In this way, the time saved from previous concurrent kernel execution can be refilled to the QoS headroom for executing throughput-oriented applications. Based on Equation 3, the QoS headroom of Q when it submits k_i can be calculated in Equation 4.

$$T_{hr} = (T_{tgt} - T_{used}) - T_q - (T_{self} - \sum_{j=1}^{i} T_j) - \sum_{i=1}^{n} t_i \tag{4}$$

In the equation, T_j is the processing time of kernel k_j, $T_{self} - \sum_{j=1}^{i} T_j$ is the remain GPU time reserved by Q itself, T_{used} is the time from the beginning of Q to k_i is

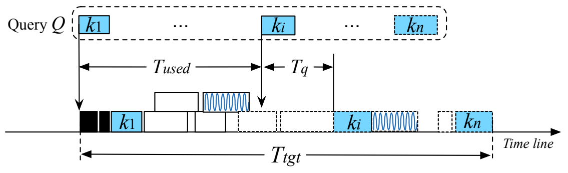


Figure 7: Updating QoS headroom of Q when it submits k_i, if concurrent kernel execution is enabled.

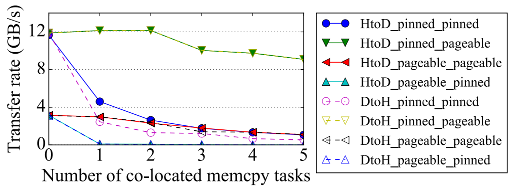


Figure 8: Data transfer rate of *stemmer* when it is co-located with applications that transfer data in the same direction.

submitted, T_q is the realtime queuing time, t_i is the remaining GPU time required by the active user-facing queries launched before Q as calculated and defined in Section 5.2.3.

In summary, the QoS headroom of a user-facing query will be updated when a kernel of the co-located applications is launched to GPU and when a new kernel of the query is submitted to the ready task pool.

6. Mitigating PCI-e Bandwidth Contention

Even if the native kernels/library calls are re-ordered as presented in Section 5, without considering PCI-e bandwidth contention, user-facing applications may still suffer from severe QoS violation. In this section, we analyze the impact of PCI-e bandwidth contention on CPU-accelerator data transfer rate per memcpy task and mitigate the contention for achieving QoS of user-facing applications.

6.1 Characterizing PCI-e Bandwidth Contention

Figure 8 reports the data transfer rate of a user-facing application *stemmer* when it is co-located with several applications that transfer data in the same direction. Data transfers in different directions do not interfere with each other, because PCI-e bus supports full-duplex communication. In the figure, the legends show the data transfer direction. For example, "HtoD_pageable_pinned" means *stemmer* transfers data from pageable memory to GPU, while the co-located applications transfer data from pinned memory to GPU. From the experiment, we have two main observations.

Observation 1: Transferring data from and to pageable memory degrades the performance of its co-located memcpy tasks only when more than three memcpy tasks are running concurrently ("*_*_pageable" in Figure 8). As shown in the figure, when *stemmer* uses pageable memory and transfers data through PCI-e bus alone, the achieved data transfer rate is 3,150MB/s. Because the theoretical peak bandwidth of 16x PCI-e 3.0 bus used in our platform is 15,800MB/s and the effective bandwidth is 12,160MB/s [40], the bus can only support $\lfloor \frac{12160}{3150} \rfloor = 3$ memcpy tasks to transfer data in their full speeds in the same direction. We generalize this observation in Section 6.2.

Observation 2: A single memcpy task that transfers data from/to pinned memory would severely degrade the performance of its co-located memcpy tasks ("*_*_pinned" in Figure 8). As shown in the figure, transferring data from/to pinned memory requires up to 11,883MB/s PCI-e bandwidth, which saturates the whole PCI-e bus. In this case, all the other memcpy tasks will be queued up and have to wait for its completion.

6.2 Managing Memcpy Tasks

Baymax mitigates QoS violations due to PCI-e bandwidth contention by reducing the number of concurrent memcpy tasks and considering data transferring delay when calculating QoS headroom for active user-facing queries.

Let BW_{peak} represent the effective PCI-e bandwidth, BW_{memcpy} represent the peak data transfer rate from/to pageable memory per memcpy task. According to our observation 1, to make sure that memcpy tasks of user-facing applications can always transfer data in full speed, Equation 5 calculates the number of active throughput-oriented memcpy tasks N_{tr} that Baymax should allow in each direction. For our platform N_{tr} is two.

$$N_{tr} = \lfloor BW_{peak}/BW_{memcpy} \rfloor - 1 \quad (5)$$

Baymax periodically iterates over the ready task pool to check whether each memcpy task can safely start to transfer data. If the memcpy task is from a throughput-oriented application and there are already N_{tr} active memcpy tasks, the task is delayed until one memcpy task completes. If the memcpy task is from a user-facing query, it is directly issued to GPU to minimize queuing delay.

According to the second observation, if a memcpy task *mc* uses pinned memory, it may severely delay the data transfer of user-facing queries. Let t represent the predicted duration of *mc*. If t is larger than the QoS headroom of any active user-facing query, *mc* will not be launched. Otherwise, *mc* can start to transfer data, but to avoid QoS violation due to the possible queuing delay caused by *mc*, the QoS headroom of every active user-facing query is reduced by t. This method would not degrade the accelerator utilization. If *mc* does not cause severe queuing delay, the QoS headroom of each active user-facing query will be refilled when a new task is launched as described in Section 5.3.

7. Evaluation

7.1 Experimental Setup

We evaluate Baymax using Nvidia GPU K40. Note that Baymax does not rely on any special hardware features or characteristics of K40 and treats it as a generic non-preemptive accelerator. The detailed setups are summarized in Table 2. MPS [26] is enabled to allow concurrent kernel execution on GPU. As listed in Table 3, We use *Tonic suite* [24] in DjiNN and *Sirius suite* [11] in Sirius as the user-facing applications; use eight most compute intensive and three most PCI-e intensive applications from *Rodinia* [41] as throughput-oriented applications. In order to evaluate the impact of memcpy tasks using both pageable memory and pinned memory, we configure *hs* to use pageable memory, *pf* and *nw* to use pinned memory. To construct the training and testing data sets for our prediction models, we collect a large amount of samples, and randomly choose 90% of the samples to train the model and use the rest to test. For KNN model, we choose the number of nearest neighbors to be 5 ($K = 5$).

Table 2: Hardware and software specifications.

	Specifications
Hardware	CPU Intel Xeon E5-2620 @ 2.10GHz Nvidia GPU Tesla K40
Software	CentOS 6.6 x86_64 with kernel 2.6.32-504 CUDA Driver 340.29, CUDA SDK 6.5, CUDA MPS

Table 3: Benchmarks used in the experiment.

Benchmark Suite	Workloads
Sirius suite in Sirius [11]	asr, gmm, stemmer
Tonic suite in DjiNN [24]	dig, face, imc, ner, pos
Rodinia [41]	heartwall (hw), lavaMD (md), cfd hybridsort (hsort), streamcluster (sc), srad, leukocyte (lc), myocyte (mc), hotspot (hs), nw, pathfinder (pf)

Throughout this section, the QoS is defined as the 99%-ile latency, and the accelerator utilization is measured as the ratio of throughput-oriented application execution time to the whole co-location execution time. The prediction error for the duration of task t (memcpy, native kernel or library call) is calculated in Equation 6.

$$Err_t = \frac{\left|Duration_t^{predicted} - Duration_t^{measured}\right|}{Duration_t^{measured}} \quad (6)$$

7.2 Task Duration Prediction

In this section, we first evaluate the accuracy of the task duration predictor in Baymax. The representative features for different types of tasks are listed in Table 1.

7.2.1 Prediction for Memcpy

In order to build duration models for memcpy tasks, we create a micro kernel to transfer data between main memory and GPU global memory with arbitrary input sizes. The range of data transfer size in our experiment reflects the actual size of memcpy tasks cross all the benchmarks. As shown in Figure 9(a), with the tested data size profiled from all the benchmarks, LR model is able to accurately predict the duration of memcpy across all workloads, which also in accordance with existing literature. The average prediction error is smaller than 3.2%, when the duration is longer than two milliseconds. Thus, Baymax uses LR to predict the duration of memcpy tasks.

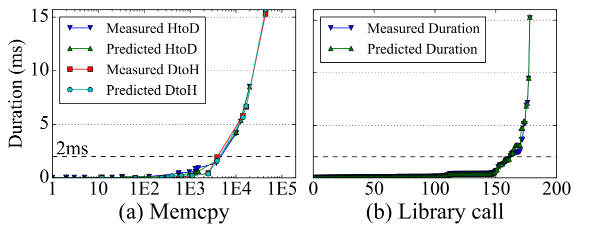


Figure 9: Prediction error for the duration of memcpy tasks and library calls. In (a), the x-axis is the size of data to be transferred (KB); In (b), the x-axis is the library calls. Baymax achieves 3.2% and 6.2% prediction errors on average for memcpy and library call respectively.

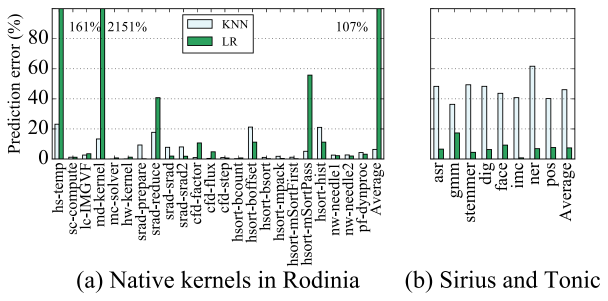


Figure 10: Prediction error for the duration of Sirius, Tonic, and native kernels in Rodinia. KNN model achieves 7.2% prediction error for Rodinia; LR model achieves 5.8% prediction error for Sirius suite and Tonic suite.

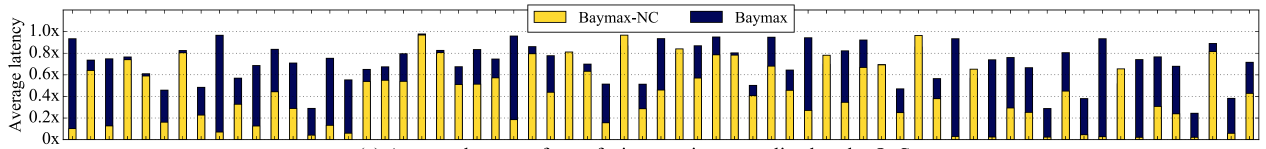
7.2.2 Prediction for Library Call

Library calls take a large portion of GPU execution time across emerging user-facing applications. All the library APIs used in the benchmarks are listed in Table 4. These library calls control which kernel to launch as well as the launch configuration with detailed information hidden behind the APIs.

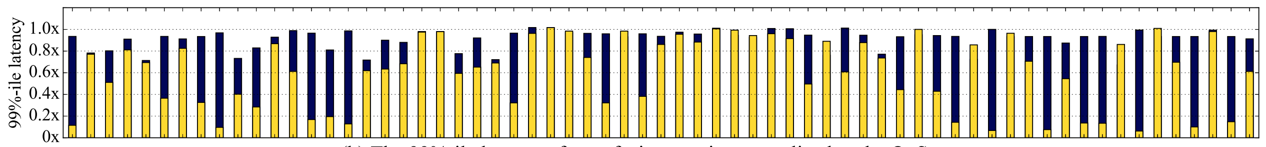
Table 4: Frequently used library APIs.

Library	API Name
cuBLAS [32]	sgemm/dgemm
cuDNN [31]	convolutionForward, addtensor4d poolingForward, activationForward, softmaxForward

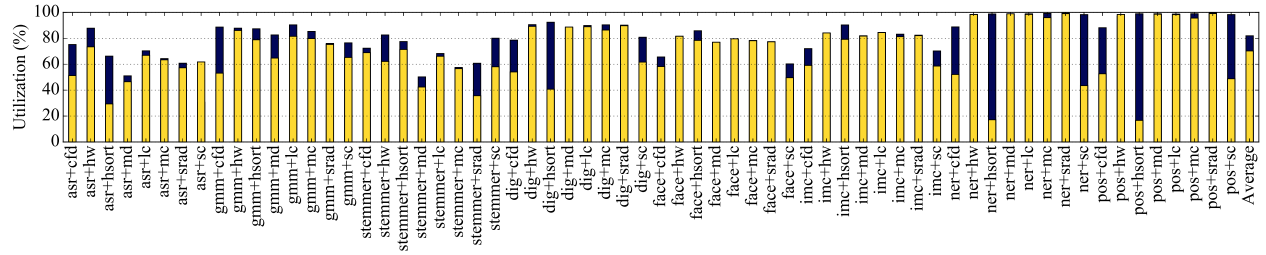
To build duration model for a library call, we analyze every parameter to the library call according to its API definition and extrapolates the size of the input based on the number as well as the data type of the input parameters. Using the input size as the representative feature available at runtime, the prediction fits well into linear regression model



(a) Average latency of user-facing queries normalized to the QoS target



(b) The 99%-ile latency of user-facing queries normalized to the QoS target



(c) Percentage of GPU time used by throughput-oriented applications

Figure 11: Normalized average latency, 99%-ile latency of user-facing queries, and accelerator utilization when user-facing applications are co-located with compute-intensive throughput-oriented applications. Different from Baymax, Baymax-NC does not consider concurrent kernel execution.

as shown in Figure 9(b), which is consistent with the findings in prior work [31, 42]. Across all the 180 calls of the library APIs in all the benchmarks, our models can precisely predict the duration of library calls with the prediction error smaller than 6.2%, when the duration is longer than two milliseconds.

If the duration of a library call or a memcpy task is shorter than two milliseconds, even if its duration is not predicted precisely, it will not affect the latency of the co-located applications seriously.

7.2.3 Prediction for Native Kernel

The behaviors of native kernels are quite diverse across benchmark suites. While Rodinia is composed of classic HPC workloads that exhibit high thread level divergence on GPU, workloads in Sirius and Tonic are speech recognition, nature language processing and DNN computation that rely on large matrix multiplication with almost no divergence. To build duration models for native kernels, we collect performance samples, including features and duration, using *nvprof* [30]. Note that most of the workloads in Rodinia contain iterative kernel invocations in their implementations and we treat each kernel invocation as an individual sample. To provide rigid validation, we use different samples to train model and to evaluate prediction accuracy.

As shown in Figure 10, no single regression model fits both user-facing and throughput-oriented applications perfectly. In general, KNN works better than LR for Rodinia since in some cases (e.g., *hs* and *md*) the prediction of LR goes extremely wrong. This observation reveals that the duration of a kernel and its inputs do not always have a linear relationship. Whereas for Tonic suite and Sirius suite, the computation is more regular and predictable, LR has more advantage over KNN with a constrained sample dataset. The average prediction error of KNN for the kernels in Rodinia is 7.2% on average, and the prediction error of LR for Sirius suite and Tonic suite is 5.8% on average.

7.3 QoS and Throughput

In this section, we evaluate the effectiveness of Baymax in increasing the accelerator utilization while satisfying the QoS requirement of emerging user-facing applications.

Figure 11 presents the average latency, 99%-ile latency of user-facing queries, and the improved accelerator utilization when user-facing applications are co-located with throughput-oriented applications. In the figure, "Baymax" updates the QoS headroom of each user-facing query when a new kernel is issued to squeeze the extra QoS headroom benefited from concurrent kernel execution as presented in Section 5.3. "Baymax-NC", on the contrary, does not squeeze the extra QoS headroom.

Figure 11(a) and Figure 11(b) show that both Baymax-NC and Baymax are able to effectively satisfy the QoS for user-facing applications under different pair-wise co-locations. On the contrary, default MPS scheduling [26] and priority-based scheduling [28, 29] cannot satisfy the QoS for user-facing applications as presented in Figure 2 in Section 2. With MPS scheduling and priority-based scheduling, the 99%-ile latency of user-facing queries is up to 195.9x and 5.2x of the QoS target, respectively.

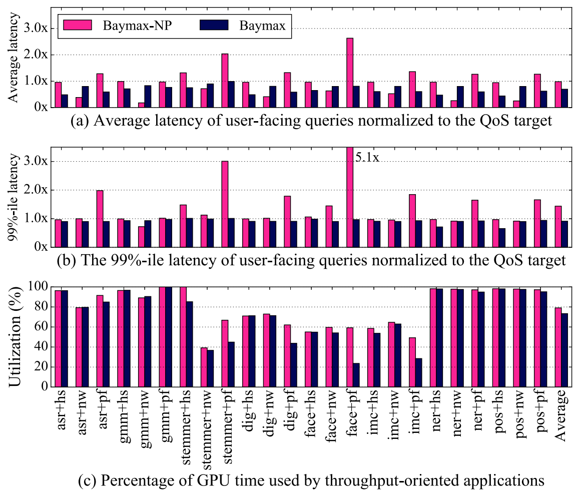


Figure 12: Normalized average latency, 99%-ile latency of user-facing queries, and accelerator utilization when user-facing applications are co-located with PCI-e intensive throughput-oriented applications. Different from Baymax, Baymax-NP does not mitigate PCI-e bandwidth contention.

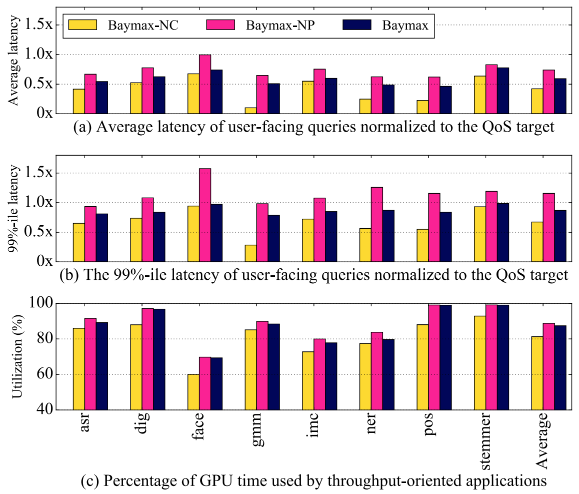


Figure 13: Normalized average latency, 99%-ile latency of user-facing queries, and accelerator utilization when each user-facing application is co-located with all the throughput-oriented applications.

Figure 11(a) and (b) also show that the average latency and 99%-ile latency of user-facing queries in Baymax is higher than in Baymax-NC. This is because Baymax squeezes more QoS headroom to trade off higher GPU utilization. As shown in the Figure 11(c), Baymax-NC increases the accelerator utilization by 70.8% on average, and Baymax further increases the average accelerator utilization by 11.4%. The reason of utilization increasing is that Baymax can utilize the saved GPU time from concurrent kernel execution to execute more throughput-oriented kernels.

Observed from Figure 11, for some co-location pairs (e.g., *dig+hsort* and *ner+hw*), the accelerator utilization is not increased using Baymax compared to Baymax-NC. This is because the kernels of these throughput-oriented applications have large GPU occupancy. In this case, MPS does not have chance to execute multiple kernels concurrently and Baymax cannot squeeze extra GPU time for throughput-oriented applications.

7.4 Mitigating PCI-e Bandwidth Contention

As presented in Section 6, Baymax also mitigates PCI-e bandwidth contention for achieving QoS of user-facing applications. Figure 12 shows the average latency and 99%-ile latency of user-facing queries when they are co-located with PCI-e intensive throughput-oriented applications. As shown in the figure, the QoS requirement of user-facing queries cannot be satisfied if PCI-e bandwidth contention is not mitigated (shown as "Baymax-NP" in Figure 12). As shown in the figure, user-facing queries still suffer from up to 5.1x QoS violation in Baymax-NP.

Even if a user-facing application is not PCI-e intensive, its occasional data transfer can be severely delayed by memcpy tasks from throughput-oriented applications. For example, while less than 10% of GPU time is spent on PCI-e data transfer for *imc* and *face*, they still suffer from severe QoS violation due to the unmanaged and unpredicted PCI-e bandwidth contention in Baymax-NP.

Figure 12(c) shows that the accelerator utilization in Baymax and Baymax-NP are similar for most of the co-locations. This is mainly because existing emerging user-facing applications do not transfer data between CPU and GPU frequently, and the duration of their memcpy tasks is often less than 10 milliseconds (Figure 9). In this case, the memcpy tasks in throughput-oriented applications will not be delayed seriously and the accelerator utilization is not reduced seriously in Baymax compared with in Baymax-NP.

7.5 Beyond Pair-wise Co-locations

To evaluate the robustness of Baymax in dealing with more complex co-location scenarios, we pick all the Rodinia benchmarks in Table 3 to form a mixture of throughput-oriented applications, and co-locate them all with the user-facing applications from both Sirius suite and Tonic suite.

We report the normalized average latency and 99%-ile latency of user-facing queries, and accelerator utilization in this scenario in Figure 13. As shown in the figure, Baymax is robust enough to increase the accelerator utilization while guaranteeing the QoS of user-facing applications. The average latency and 99%-ile latency of user-facing applications with Baymax and Baymax-NC are always within the QoS target as shown in Figure 13(a) and Figure 13(b). On the contrary, Baymax-NP cannot satisfy the QoS of user-facing application (up to 1.6x QoS violation in terms of 99%-ile latency) due to the unawareness of PCI-e bandwidth contention. Compared with Baymax-NP, Baymax can achieve similar utilization improvement while satisfying the QoS of all the user-facing applications. Compared with Baymax-NC, Baymax can further increase the average accelerator utilization from 81.2% to 87.4% as shown in Figure 13(c).

7.6 Applying Baymax in a WSC

In this section, instead of evaluating Baymax on a single GPU, we conduct experiments to evaluate the effectiveness of Baymax in a GPU-outfitted datacenter scenario. In the experiment, we model a datacenter composed of 800 Nvidia K40 GPUs, 100 GPUs for each type of user-facing applications in Sirius suite and Tonic suite. The throughput-oriented workloads are composed of 8000 instances (10 instances assigned to each GPU) evenly selected from Rodinia shown in Table 3. In the experiment, we use pair-wise co-locations, and randomly select throughput-oriented applications to co-locate with each user-facing application.

Figure 14 shows the percentage of co-locations that suffer from QoS violation under different scheduling policies. The first three bars present QoS violation when co-located applications on each GPU are scheduled using the default MPS scheduling policy, priority-based scheduling policy and Baymax, respectively. As shown here, 55% of the user-facing applications suffer from severe QoS violations($>$40% degradation) with MPS scheduling, and 37.5% with priority-based scheduling. On the contrary, Baymax is able to maintain the QoS of user-facing applications for most co-locations. Less than 5% of user-facing applications suffer from insignificant QoS violations (less than 2% degradation) with Baymax. In addition to randomly mapping jobs at the cluster level to each GPU (the first three bars), we also present data for Baymax when the cluster-level job mapping is done using *Hungarian algorithm* [43]. When the accelerator utilization for each co-location pair with Baymax is known through profiling, *Hungarian algorithm*, a combinatorial optimization algorithm can be used at the cluster level to select the best mapping of applications to the GPUs that achieve the highest utilization (Denoted by "Baymax+Hungarian"). In other words, this presents the best case utilization Baymax can achieve. As shown here, Baymax+Hungarian also only incur negligible QoS violation.

Figure 15 presents the accelerator utilization of GPUs when the co-located applications are scheduled with MPS scheduling, priority-based scheduling, and Baymax. As shown in the figure, Baymax significantly improves accelerator utilization by selecting appropriate co-locations and scheduling tasks appropriately. On average, Baymax is able to achieve 79.9% accelerator utilization improvement at the WSC level. If Hungarian algorithm is applied to choose co-location pairs at WSC level and Baymax is applied to schedule tasks on the same GPU, the average accelerator utilization is further increased to 91.3%.

Figures 14 and 15 show that Baymax is effective at significantly improving accelerator utilization while guaranteeing the QoS of user-facing applications at the WSC level. On the contrary, default MPS scheduling policy and priority-based scheduling policy result in severe QoS violation and low utilization.

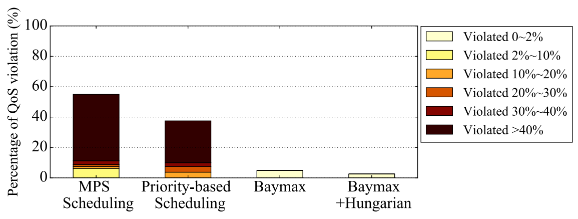


Figure 14: Percentage of QoS violation in all scheduled co-locations when using MPS scheduling policy, priority-based scheduling policy, and Baymax.

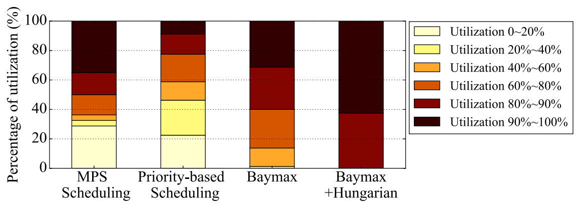


Figure 15: Percentage of accelerator utilization of each GPU at different levels when using MPS scheduling policy, priority-based scheduling policy, and Baymax.

8. Related Work and Limitations

In this section, we discuss the state-of-the-art techniques and their limitations.

8.1 Improving CPU Utilization

There has been a large amount of prior work focusing on improving application QoS and hardware utilization [18, 20, 21, 23, 46]. Recently, techniques have been proposed to improve CPU utilization while guaranteeing the QoS requirement of high priority user-facing applications. Bubble-Up [18] and Bubble-Flux [20] identify "safe" co-locations that bound performance degradation while improving chip multiprocessor utilization. SMiTe [23] 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 apply to non-preemptive accelerators.

Some other prior scheduling infrastructures, such as Loadleveler [44] and Maui [47], attempt to increase the hardware utilization by allocating jobs to servers using backfilling scheduling algorithm [48]. These infrastructures require users to provide resource requirement of every job. Moreover, as these techniques overlook the interference between co-located jobs, for example, PCI-e bandwidth contention is not considered, they are not able to guarantee the QoS of user-facing applications on accelerators.

Table 5: Comparison between Baymax and prior work

	Loadleveler [44]	GPUSync [29]	Bubble-Flux [20]	GPU-EvR [45]	SMiTe [23]	Quasar [21]	TimeGraph [28]	Baymax
QoS awareness	✓		✓		✓	✓		✓
Improved utilization	✓		✓		✓	✓		✓
Work on accelerator		✓		✓			✓	✓
Commodity HW/driver	✓	✓	✓		✓	✓		✓
Mitigated PCI-e contention								✓
No user-provided info.			✓		✓	✓		✓
Concurrent kernel exec.				✓				✓
Adaptive	✓	✓	✓		✓	✓	✓	✓

In addition to backfilling scheduling policy, rate monotonic scheduling algorithm [49, 50] and its variations [51, 52] are proposed to schedule periodical tasks with different priorities in embedded systems. In rate monotonic scheduling, shorter tasks are given higher priorities to be scheduled earlier. These scheduling algorithms assume that the inter-arrival rate of every task is fixed and the duration of every task is known before scheduling. However, in real world datacenters, the duration of user-facing queries and the inter-arrival rate of queries may vary significantly at runtime, therefore these scheduling algorithms do not apply to those real-world datacenter scenarios.

8.2 Scheduling on Accelerator

Realtime scheduling on accelerator is another research direction related to Baymax. Prior work [28, 29, 53, 54] has proposed techniques to improve the performance of traditional realtime GPU tasks (e.g., frames per second for video processing) when they are co-located with other GPU tasks. TimeGraph [28] and GPUSync [29] use priority-based policies to manage kernel execution on GPUs. High priority kernels are executed first if multiple kernels are launched to the same GPU. GPU-EvR [45] maps concurrent applications to different streaming multiprocessors (SMs) on the same GPU. These techniques assign a fix proportion of GPU time to high priority tasks but cannot guarantee that the realtime tasks do not violate the QoS requirement [45]. In addition, these techniques rely on users to provide task arrival rate, length of time window and the expected GPU time for each type of GPU tasks. Such information is often unavailable in real datacenter environment. In addition, these techniques focus on increasing throughput for high priority tasks, overlooking the long tail latency problem, which is more critical for user-facing applications.

At the hardware level, GPU thread preemption [55, 56] is also proposed to intelligently schedule threads for improved hardware utilization. Tanasic et al. [57] proposed a technique that improves performance of high priority processes by enabling preemptive scheduling on GPUs. The proposed technique requires vendors to add extra hardware extensions and does not work on commodity accelerators. Aguilera et al. [54] proposed a technique to guarantee QoS of high priority tasks by spatially allocating them more SMs on a GPU. This work assumes that programmers can decide how to allocate SMs to the co-located applications. However, commodity GPUs do not support allocating a set of SMs to a specific application.

Other techniques improve application performance on GPUs through addressing the problems of data transfer [58, 59], thread divergence [60], data placement [61], synchronization overhead [62] and configuration tuning [63, 64]. GPU resource sharing has been studied at both system [65, 66] and architecture levels [67, 68] to address the resource contention and performance interference. Table 5 compares our proposed technique, Baymax, with prior utilization improving techniques and GPU scheduling techniques.

9. Conclusion

Baymax improves the hardware utilization in WSCs while guaranteeing the QoS requirement of user-facing applications on the non-preemptive accelerator. To achieve the above purpose, Baymax enables precise kernel duration prediction, QoS aware kernel re-ordering, and PCI-e bandwidth contention aware data transfer management. Through evaluating Baymax with emerging user-facing workloads, we demonstrate the effectiveness of Baymax in eliminating QoS violation due to kernel interference and PCI-e bandwidth contention. We achieve up to 91.3% utilization improvement on average for pair-wise co-locations at the WSC level. Beyond the pair-wise co-locations, Baymax can improve the accelerator utilization by 87.4% without violating the QoS of 99%-ile latency for user-facing applications.

10. Acknowledgements

We thank our anonymous reviewers for their feedback and suggestions. This work was partially sponsored by Google and by the National Science Foundation (NSF) under grant CNS-CSR-1419243. We also would like to acknowledge Xayden, born 5 weeks before the deadline, for being an awesome and easy baby.

References

[1] David Huggins-Daines, Mohit Kumar, Arthur Chan, Alan W Black, Mosur Ravishankar, and Alex I Rudnicky. Pocketsphinx: A Free, Real-time Continuous Speech Recognition System for Hand-Held Devices. In *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, volume 1, pages 185–188. IEEE, 2006.

[2] Daniel Povey, Arnab Ghoshal, Gilles Boulianne, Lukáš Burget, Ondřej Glembek, Nagendra Goel, Mirko Hannemann,

Petr Motlíček, Yanmin Qian, Petr Schwarz, et al. The Kaldi Speech Recognition Toolkit. 2011.

[3] Herbert Bay, Tinne Tuytelaars, and Luc Van Gool. Surf: Speeded Up Robust Features. In *Computer Vision–ECCV 2006*, pages 404–417. Springer, 2006.

[4] Qualcomm Acquires Kooaba Visual Recognition Company. http://mobilemarketingmagazine.com/qualcomm-acquires-kooaba-visual-recognition-company.

[5] Erik F Tjong Kim Sang and Sabine Buchholz. Introduction to the CoNLL-2000 Shared Task: Chunking. In *the 2nd Workshop on Learning Language in Logic and the 4th Conference on Computational Natural Language Learning-Volume 7*, pages 127–132. Association for Computational Linguistics, 2000.

[6] Marti A Hearst. 'Natural' Search User Interfaces. *Communications of the ACM*, 54(11):60–67, 2011.

[7] Google's Google Now. `http://www.google.com/landing/now/`.

[8] Microsoft's Cortana. `http://www.windowsphone.com/en-us/features-8-1`.

[9] Apple Siri. `https://www.apple.com/ios/siri/`.

[10] Baidu YuYin. `http://yuyin.baidu.com/`.

[11] Johann Hauswald, Michael A. Laurenzano, Yunqi Zhang, Cheng Li, Austin Rovinski, Arjun Khurana, Ron Dreslinski, Trevor Mudge, Vinicius Petrucci, Lingjia Tang, and Jason Mars. Sirius: An Open End-to-End Voice and Vision Personal Assistant and Its Implications for Future Warehouse Scale Computers. In *the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2015.

[12] Andrew Putnam, Adrian M Caulfield, Eric S Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jordan Gray, et al. A Reconfigurable Fabric for Accelerating Large-scale Datacenter Services. In *the 41st International Symposium on Computer Architecture (ISCA)*, pages 13–24. ACM/IEEE, 2014.

[13] Nicola Jones. The Learning Machines, 2014.

[14] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines. *Synthesis Lectures on Computer Architecture*, 8(3):1–154, 2013.

[15] Jeffrey Dean and Luiz André Barroso. The Tail at Scale. *Communications of the ACM*, 56(2):74–80, 2013.

[16] Lingjia Tang, Jason Mars, and Mary Lou Soffa. Compiling for niceness: Mitigating contention for qos in warehouse scale computers. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization (CGO)*, CGO '12, pages 1–12, New York, NY, USA, 2012. ACM.

[17] Jason Mars and Lingjia Tang. Whare-map: Heterogeneity in "homogeneous" warehouse-scale computers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*, ISCA '13, pages 619–630, New York, NY, USA, 2013. ACM.

[18] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. Bubble-Up: Increasing Utilization in Modern Warehouse Scale Computers via Sensible Co-locations. In *the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 248–259. IEEE/ACM, 2011.

[19] Wei Wang, Tanima Dey, Jason Mars, Lingjia Tang, Jack W. Davidson, and Mary Lou Soffa. Performance analysis of thread mappings with a holistic view of the hardware resources. In *Proceedings of the 2012 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*, ISPASS '12, pages 156–167, Washington, DC, USA, 2012. IEEE Computer Society.

[20] Hailong Yang, Alex Breslow, Jason Mars, and Lingjia Tang. Bubble-flux: Precise Online QoS Management for Increased Utilization in Warehouse Scale Computers. In *the 40th Annual International Symposium on Computer Architecture (ISCA)*, pages 607–618. ACM/IEEE, 2013.

[21] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-efficient and QoS-aware Cluster Management. In *the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 127–144. ACM, 2014.

[22] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: Improving Resource Efficiency at Scale. In *the 42nd International Symposium on Computer Architecture (ISCA)*, pages 450–462. ACM/IEEE, 2015.

[23] Yunqi Zhang, Michael Laurenzano, Jason Mars, and Lingjia Tang. SMiTe: Precise QoS Prediction on Real System SMT Processors to Improve Utilization in Warehouse Scale Computers. In *the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 406–418. IEEE/ACM, 2014.

[24] Johann Hauswald, Yiping Kang, Michael A. Laurenzano, Quan Chen, Cheng Li, Ronald Dreslinski, Trevor Mudge, Jason Mars, and Lingjia Tang. DjiNN and Tonic: DNN as a Service and Its Implications for Future Warehouse Scale Computers. In *the 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 27–40. ACM/IEEE, 2015.

[25] Vinicius Petrucci, Michael Laurenzano, John Doherty, Yunqi Zhang, Daniel Mosse, Jason Mars, and Lingjia Tang. Octopus-Man: QoS-driven Task Management for Heterogeneous Multicores in Warehouse-Scale Computers. In *the 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 246–258. IEEE, 2015.

[26] Nvidia Multi-Process Service. `https://docs.nvidia.com/deploy/pdf/CUDA\_Multi\_Process\_Service\_Overview.pdf`.

[27] Carlos Boneti, Francisco J. Cazorla, Roberto Gioiosa, Alper Buyuktosunoglu, Chen-Yong Cher, and Mateo Valero. Software-Controlled Priority Characterization of POWER5 Processor. In *the 35th International Symposium on Computer Architecture (ISCA)*, pages 415–426. ACM/IEEE, 2008.

[28] Shinpei Kato, Karthik Lakshmanan, Raj Rajkumar, and Yutaka Ishikawa. TimeGraph: GPU Scheduling for Real-time Multi-tasking Environments. In *USENIX Annual Technical Conference (ATC)*, pages 17–30. USENIX, 2011.

[29] Glenn Elliott, Bryan C Ward, and James H Anderson. GPUSync: A Framework for Real-time GPU Management. In *the 34th Real-Time Systems Symposium (RTSS)*, pages 33–44. IEEE, 2013.

[30] Profiler User's Guide. `http://docs.nvidia.com/cuda/profiler-users-guide`.

[31] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cuDNN: Efficient Primitives for Deep Learning. *arXiv preprint arXiv:1410.0759*, 2014.

[32] CUDA Nvidia. cuBLAS library. *Nvidia Corporation, Santa Clara, California*, 15, 2008.

[33] David Kirk et al. Nvidia CUDA Software and GPU Parallel Computing Architecture. In *the 6th International Symposium on Memory Management (ISMM)*, volume 7, pages 103–104. ACM, 2007.

[34] George AF Seber and Alan J Lee. *Linear Regression Analysis*, volume 936. John Wiley & Sons, 2012.

[35] Sunil Arya, David M Mount, Nathan S Netanyahu, Ruth Silverman, and Angela Y Wu. An Optimal Algorithm for Approximate Nearest Neighbor Searching Fixed Dimensions. *Journal of the ACM*, 45(6):891–923, 1998.

[36] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An Introduction to Statistical Learning*. Springer, 2013.

[37] Chih-Chung Chang and Chih-Jen Lin. LIBSVM: A Library for Support Vector Machines. *ACM Transactions on Intelligent Systems and Technology*, 2(3):27, 2011.

[38] Cui Yu, Rui Zhang, Yaochun Huang, and Hui Xiong. High-Dimensional KNN Joins with Incremental Updates. *Geoinformatica*, 14(1):55–82, 2010.

[39] Vincent Garcia, Eric Debreuve, and Michel Barlaud. Fast K Nearest Neighbor Search using GPU. In *Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, pages 1–6. IEEE, 2008.

[40] Alex Goldhammer and John Ayer Jr. Understanding Performance of PCI Express Systems. *Xilinx WP350, Sept*, 4, 2008.

[41] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *IEEE International Symposium on Workload Characterization (IISWC)*, pages 44–54. IEEE, 2009.

[42] Sergio Barrachina, Maribel Castillo, Francisco D Igual, Rafael Mayo, and Enrique S Quintana-Orti. Evaluation and Tuning of the Level 3 cuBLAS for Graphics Processors. In *International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–8. IEEE, 2008.

[43] Harold W Kuhn. The Hungarian Method for the Assignment Problem. *Naval research logistics quarterly*, 2(1-2):83–97, 1955.

[44] Subramanian Kannan, Mark Roberts, Peter Mayes, Dave Brelsford, and Joseph F Skovira. Workload Management with Loadleveler. *IBM Redbooks*, 2:2, 2001.

[45] Haeseung Lee, Al Faruque, and Mohammad Abdullah. GPU-EvR: Run-time Event based Real-time Scheduling Framework on GPGPU Platform. In *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 1–6. IEEE, 2014.

[46] Michael A Laurenzano, Yunqi Zhang, Lingjia Tang, and Jason Mars. Protean code: Achieving Near-Free Online Code Transformations for Warehouse Scale Computers. In *the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 558–570. IEEE/ACM, 2014.

[47] David Jackson, Quinn Snell, and Mark Clement. Core Algorithms of the Maui Scheduler. In *Job Scheduling Strategies for Parallel Processing*, pages 87–102. Springer, 2001.

[48] Ahuva W Mu Alem and Dror G Feitelson. Utilization, Predictability, Workloads, and User Runtime Estimates in Scheduling the IBM SP2 with Backfilling. *IEEE Transactions on Parallel and Distributed Systems*, 12(6):529–543, 2001.

[49] Chung Laung Liu and James W Layland. Scheduling Algorithms for Multiprogramming in a Hard-real-time Environment. *Journal of the ACM*, 20(1):46–61, 1973.

[50] Lui Sha, Ragunathan Rajkumar, and Shirish S Sathaye. Generalized Rate-Monotonic Scheduling Theory: A Framework for Developing Real-time Systems. *Proceedings of the IEEE*, 82(1):68–82, 1994.

[51] Neil C Audsley, Alan Burns, MF Richardson, and AJ Wellings. *Deadline Monotonic Scheduling*. Citeseer, 1990.

[52] Alan Bertossi, Luigi V Mancini, and Federico Rossini. Fault-tolerant Rate-Monotonic First-fit Scheduling in Hard-real-time Systems. *IEEE Transactions on Parallel and Distributed Systems*, 10(9):934–945, 1999.

[53] Glenn A Elliott and James H Anderson. Globally Scheduled Real-time Multiprocessor Systems with GPUs. *Real-Time Systems*, 48(1):34–74, 2012.

[54] Pedro Aguilera, Katherine Morrow, and Nam Sung Kim. QoS-aware Dynamic Resource Allocation for Spatial-multitasking GPUs. In *the 19th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 726–731. IEEE, 2014.

[55] Jason Jong Kyu Park, Yongjun Park, and Scott Mahlke. Chimera: Collaborative Preemption for Multitasking on a Shared GPU. In *the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 593–606. ACM, 2015.

[56] Kittisak Sajjapongse, Xiang Wang, and Michela Becchi. A Preemption-based Runtime to Efficiently Schedule Multi-process Applications on Heterogeneous Clusters with GPUs. In *the 22nd International Symposium on High-performance Parallel and Distributed Computing (HPDC)*, pages 179–190. ACM, 2013.

[57] Ivan Tanasic, Isaac Gelado, Javier Cabezas, Alex Ramirez, Nacho Navarro, and Mateo Valero. Enabling Preemptive Multiprogramming on GPUs. In *the 41st International Symposium on Computer Architecuture (ISCA)*, pages 193–204. ACM/IEEE, 2014.

[58] Neha Agarwal, David Nellans, Mike O'Connor, Stephen W Keckler, and Thomas F Wenisch. Unlocking Bandwidth for GPUs in CC-NUMA Systems. In *the 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 354–365. IEEE, 2015.

[59] Jens Breitbart. Analysis of a Memory Bandwidth Limited Scenario for NUMA and GPU systems. In *the 25th International Symposium on Parallel and Distributed Processing Workshops(IPDPSW)*, pages 693–699. IEEE, 2011.

[60] Ping Xiang, Yi Yang, and Huiyang Zhou. Warp-level Divergence in GPUs: Characterization, Impact, and Mitigation. In

the 20th International Symposium on High Performance Computer Architecture (HPCA), pages 284–295. IEEE, 2014.

[61] Guoyang Chen, Bo Wu, Dong Li, and Xipeng Shen. Enabling Portable Optimizations of Data Placement on GPU. *Micro*, 35(4):16–24, July 2015.

[62] Daniel Lustig and Margaret Martonosi. Reducing GPU Offload Latency via Fine-grained CPU-GPU Synchronization. In *the 19th International Symposium on High Performance Computer Architecture (HPCA)*, pages 354–365. IEEE, 2013.

[63] Ankit Sethia and Scott Mahlke. Equalizer: Dynamic Tuning of GPU Resources for Efficient Execution. In *the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 647–658. IEEE/ACM, 2014.

[64] J-F Dollinger and Vincent Loechner. Adaptive Runtime Selection for GPU. In *the 42nd International Conference on Parallel Processing (ICPP)*, pages 70–79. IEEE, 2013.

[65] Vignesh T Ravi, Michela Becchi, Gagan Agrawal, and Srimat Chakradhar. Supporting GPU Sharing in Cloud Environments with a Transparent Runtime Consolidation Framework. In *the 20th International Symposium on High Performance Distributed Computing (HPDC)*, pages 217–228. ACM, 2011.

[66] Khaled M Diab, M Mustafa Rafique, and Mohamed Hefeeda. Dynamic Sharing of GPUs in Cloud Systems. In *the 27th International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 947–954. IEEE, 2013.

[67] Onur Kayiran, Nachiappan Chidambaram Nachiappan, Adwait Jog, Rachata Ausavarungnirun, Mahmut T Kandemir, Gabriel H Loh, Onur Mutlu, and Chita R Das. Managing GPU Concurrency in Heterogeneous Architectures. In *the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 114–126. IEEE/ACM, 2014.

[68] Rashid Kaleem, Rajkishore Barik, Tatiana Shpeisman, Brian T Lewis, Chunling Hu, and Keshav Pingali. Adaptive Heterogeneous Scheduling for Integrated GPUs. In *the 23rd International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 151–162. ACM, 2014.