Deadline-Aware Offloading for High-Throughput Accelerators

Tsung Tai Yeh[*,^] Matthew D. Sinclair[+,||] Bradford M. Beckmann[||] Timothy G. Rogers[*]

|| AMD Research * Purdue University ^ National Chiao Tung University + University of Wisconsin
ttyeh@cs.nctu.edu.tw sinclair@cs.wisc.edu Brad.Beckmann@amd.com timrogers@purdue.edu

Abstract

Contemporary GPUs are widely used for throughput-oriented data-parallel workloads and increasingly are being considered for latency-sensitive applications in datacenters. Examples include recurrent neural network (RNN) inference, network packet processing, and intelligent personal assistants. These data parallel applications have both high throughput demands and real-time deadlines (40µs-7ms). Moreover, the kernels in these applications have relatively few threads that do not fully utilize the device unless a large batch size is used. However, batching forces jobs to wait, which increases their latency, especially when realistic job arrival times are considered.

Previously, programmers have managed the tradeoffs associated with concurrent, latency-sensitive jobs by using a combination of GPU streams and advanced scheduling algorithms running on the CPU host. Although GPU streams allow the accelerator to execute multiple jobs concurrently, prior state-of-the-art solutions use the relatively distant CPU host to prioritize the latency-sensitive GPU tasks. Thus, these approaches are forced to operate at a coarse granularity and cannot quickly adapt to rapidly changing program behavior.

We observe that fine-grain, device-integrated kernel schedulers efficiently meet the deadlines of concurrent, latency-sensitive GPU jobs. To overcome the limitations of software-only, CPU-side approaches, we extend the GPU queue scheduler to manage real-time deadlines. We propose a novel laxity-aware scheduler (LAX) that uses information collected within the GPU to dynamically vary job priority based on how much laxity jobs have before their deadline. Compared to contemporary GPUs, 3 state-of-the-art CPU-side schedulers and 6 other advanced GPU-side schedulers, LAX meets the deadlines of 1.7X – 5.0X more jobs and provides better energy-efficiency, throughput, and 99-percentile tail latency.

***Keywords** – GPGPU, job scheduling, laxity*

1 Introduction

GPUs are the programmable accelerator of choice for massively data-parallel applications that do not have strict latency requirements. However, there is a growing class of latency-sensitive, data-parallel workloads that can benefit from the GPU's throughput. Examples include machine learning (ML) inference for RNNs [24]-[28][51], network packet processing [61]-[63], and natural language processing (NLP) in intelligent personal assistants (IPAs) [65][70]. These latency-sensitive applications have become a staple of contemporary datacenters, which increasing include GPUs and other high-throughput accelerators. Given the availability of GPUs in the datacenter, and the data-parallel nature of the applications, there is significant potential to offload work from overburdened CPUs to an accelerator. However, contemporary GPUs are deadline-blind and have no mechanism to predict which work can be offloaded and completed in time.

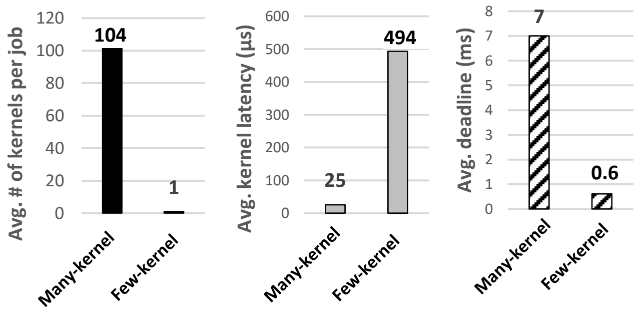


Figure 1: Characteristics of *many-kernel* latency-sensitive jobs versus *few-kernel* latency-sensitive jobs, listed in Table 4.

Many deadline-driven applications exhibit a middling amount of data-parallelism [43]. Enough to justify GPU acceleration, but not enough to fully utilize the GPU's resources [69][70]. As a result, executing one job on the GPU at a time causes severe underutilization. To alleviate this issue, programmers batch similar jobs together [28], greatly improving throughput and utilization at the expense of additional latency. This increase in latency is usually unacceptable for tasks with tight deadlines [3], especially when realistic job arrival rates are considered. GPU programs can avoid batching, while still executing multiple jobs at once with streams. Streams allow kernels from independent jobs to be scheduled concurrently on multiple command queues located between the CPU and GPU [45][49][52]. However, as we discuss further in Section 2, software cannot efficiently manage the relative priority of these queues at short time scales, which makes it difficult to efficiently re-prioritize jobs with different deadlines as contention in the GPU changes.

State-of-the-art GPU solutions for managing latency-sensitive tasks are restricted to varying priorities at a coarse granularity on the host CPU [53]-[55], and thus do not fully utilize the GPU's integrated queue scheduling logic. Consequently, the precision of information available to these CPU-side mechanisms is limited. Dynamic, microsecond-scale information about GPU-side contention, which some latency-sensitive applications require, is difficult to track from host-side software. As a result, these software-only techniques are less effective when scheduling many latency-sensitive jobs and primarily focus on mixing latency-insensitive and latency-sensitive work. We discuss related work further in Section 7. In contrast, we target a common situation where datacenters execute homogenous, latency-sensitive jobs in parallel [80].

Figure 1 demonstrates how quickly scheduling decisions must be made when executing concurrent latency-sensitive jobs. To better understand their demands, we subdivide our latency-sensitive applications into two categories: *many-kernel* and *few-kernel*. The *many-kernel* applications we study, which come from ML inference, are composed of several relatively small, short kernels, and typically have deadlines on the order of milliseconds. The f*ew-kernel* applications, which come from network packet processing and IPAs, execute a single, much longer kernel, but have more aggressive deadlines (usually < 1 ms). To efficiently manage both *many-kernel* and *few-kernel* applications, per-kernel scheduling decisions must be made at the microsecond timescale.

We argue that dynamic, integrated stream scheduling is necessary to meet the low-latency scheduling demands of these workloads. An analogy can be made to the memory hierarchy in modern CPUs. At the lower-levels of the CPU memory hierarchy, the operating system is responsible for managing the replacement of relatively large pages in physical memory from the relatively high-latency disk. However, smaller cache blocks, which require nanosecond-scale response times, are managed by hardware. In throughput-oriented GPUs, scheduling relatively few, millisecond- or second-scale kernels in software is acceptable. However, managing many short-running kernels to meet sub-millisecond or millisecond-scale deadlines can be enhanced with improved scheduling within the GPU, which to our knowledge no prior work evaluating compute workloads has proposed.

Integrated GPU stream scheduling in contemporary compute-oriented GPUs operates in a deadline-blind manner. Typically, the GPU driver statically assigns priority levels to each command queue [76], although some APIs allow priorities to be set by the application [16] on stream allocation. In contrast, an effective deadline-aware scheduler must: (1) be aware of each job's deadline, (2) estimate each job's remaining execution time, and (3) frequently adjust job priority as time progresses and the contention level in the GPU changes. We propose an integrated *laxity-aware* stream scheduler (LAX) that achieves all three of these requirements.

LAX leverages the idea that stream-based GPU applications enqueue all their kernels in quick succession. In *many-kernel* jobs, although each kernel launch is dependent on the data output by the previous kernel, all kernels associated with a particular job are known before the GPU begins execution. Thus, LAX uses the GPU's queue scheduler [or command processor (CP)], to perform a novel *stream inspection* technique that estimates the amount of work in each job. LAX's scheduling algorithm then combines this information, the job's deadline, and fine-grain information about current per-kernel work completion rates to accurately estimate how much *laxity* the job has. A job's laxity is an estimate of how much earlier than its deadline it will finish given current conditions [46]. Based on each job's estimated laxity, LAX re-prioritizes jobs to complete as many as possible by their respective deadlines. With the rich, fine-grained information available to GPU stream schedulers, LAX also prevents job oversubscription with a Little's Law-based queuing delay estimate [30][50] to reject work predicted to miss its deadline.

Contemporary GPUs perform device-side stream scheduling in a round-robin fashion, which ignores deadlines and the amount of remaining work. To our knowledge, no prior work has considered both job deadline and remaining work for real-time prioritization techniques on GPU compute applications. We compare LAX against 6 other advanced schedulers in the command processor (described in Table 3). LAX outperforms all other advanced schedulers by leveraging stream inspection and the work completion rate to judiciously reject jobs and prioritize critical work, demonstrating that deadline-aware scheduling is possible and practical in GPUs.

Prior work on real-time systems in the CPU space has used laxity to schedule jobs (discussed further in Section 7). However, the GPU's task-based (a.k.a, kernel-based) programming model presents a unique set of challenges and opportunities compared to applying laxity to OS-managed CPU threads. GPUs use a hierarchical execution model, where jobs contain one or more executed kernels that are themselves composed of workgroups. To leverage laxity scheduling within the GPU, we propose a novel job estimation mechanism based on workgroup completion times that naturally adjusts as both workgroups and kernels scale (discussed further in Section 4). Another unique challenge in applying laxity to GPUs is quickly and appropriately adapting to the extreme contention in massively parallel workloads. Our workgroup-centric estimation mechanism adapts to contention by monitoring the fine-grained workgroup completion rate.

Prior work on GPU kernel preemption or re-execution [56]-[59][79] are alternative mechanisms that can be used in combination with better stream scheduling. However, for latency-sensitive workloads, the overheads associated with preempting GPU kernel contexts, whose aggregate registers and scratchpad size can be 100s of KBs (Table 1), may be prohibitive. Additionally, the benefits of preemption are muted for short running kernels that finish long before the cost of preemption and rescheduling can be amortized. Specifically,, Table 1 indicates that the vast majority of kernels in our evaluated latency-sensitive workloads complete within 10 µs. Recently proposed preemption-based techniques, such as PREMA, are effective at intelligently preempting and scheduling relatively coarse-grained tasks [79]. However, LAX is able to outperform PREMA by 2.0X geomean on fine-grain

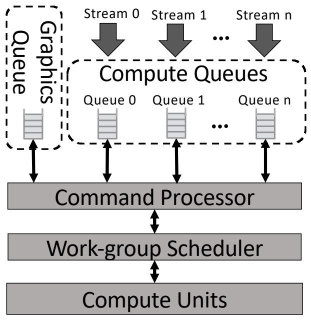


Figure 2: GPU Queue Scheduler Architecture.

tasks, by making intelligent fine-grained scheduling decisions without preemption (Section 6). We compare against additional preemption-based techniques in Section 7.

Overall, this paper makes the following contributions:

1. We observe that emerging, latency-sensitive applications use a *many-kernel* execution pattern and *few-kernel* jobs have very tight deadlines, both of which require microsecond-level scheduling decisions.
2. We propose a novel stream inspection mechanism, which is used in combination with a dynamic, per-kernel work completion rate to generate accurate estimates of work and time remaining in each job, given current contention conditions.
3. We propose a laxity-aware algorithm (LAX) and compare it to a continuum of solutions that range from doing all scheduling in host-side software to entirely within the GPU's CP. Given a per-job deadline provided by the programmer, LAX dynamically varies job priorities to improve throughput while attempting to meet real-time latency requirements.
4. LAX's combination of access to fine-grained information, more accurate queuing delay model, tight CP integration, and ability to rapidly adapt to contention completes more jobs by their deadlines and significantly improves GPU job throughput over a variety of contemporary and advanced schedulers by 1.7X-5.0X. LAX also provides a better combination of energy and performance, as well as throughput and 99-percentile latency, making latency-sensitive RNN inference [12][13], networking [61]-[63][66], and IPA [65][70] applications more practical on GPUs.

2 GPU Stream Scheduler Background

Unlike CPUs, GPUs contain multiple levels of hardware scheduling to manage the large number of in-flight threads. Contemporary GPUs contain multiple queues to manage independent work submitted asynchronously with streams [45][49][52]. This independent work can be executed concurrently when GPU resources are available. We next describe current stream scheduling architecture and operation.

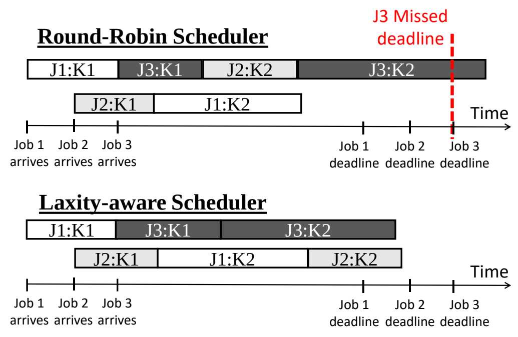


Figure 3: Comparison of Round Robin and Laxity-aware Schedulers for a GPU that can simultaneously execute 2 jobs.

2.1 GPU Command Processor

The CP is an integrated microprocessor within a GPU, which parses the kernel contexts and schedules streams. In Figure 2, each stream is mapped to a queue and each queue holds multiple kernels from a single stream. Inter-kernel dependencies between kernels in the same stream are maintained, but GPUs can asynchronously execute kernels from different streams simultaneously. Each queue entry describes a separate kernel and includes details such as thread dimensions, register usage, and local data store (LDS) size. The work-group (WG) scheduler reads these fields to dispatch work groups to compute units (CUs). Generally, GPU WG schedulers issue all WGs from one kernel before switching to WGs from another kernel. Despite this restriction, WGs from kernels in different queues often interleave execution.

Normally, the CP in modern GPUs schedules kernels within these queues in a round robin (RR) manner [48]. This deadline-blind scheduling policy improves throughput, but makes it harder to complete jobs by their real-time deadlines. The top half of Figure 3 illustrates the problem with RR. In this example, the GPU is running three jobs with varying arrival times such that the deadline of each job varies. Each job contains two kernels with different execution times. For simplicity, we assume that at most two kernels can be concurrently executed. RR will schedule kernel 1 from job 1 (J1:K1) and kernel 1 from job 2 (J2:K1) first because they arrive before job 3. When job 3 arrives, its first kernel is scheduled after J1:K1, and then J3 is not scheduled again until both J1:K2 and J2:K2 have executed. Since J3 is the longest job, if it had been prioritized over J1 and J2, all the jobs could have made their deadlines. However, since RR is unaware of this, J3 misses its deadline.

2.2 Priority-based GPU Programming

At the application level, programmers can specify a limited number of priorities (e.g., high and low) typically immediately after allocating the stream [16]. Contemporary drivers and CPs are not designed to dynamically vary the priority of streams, which limits their ability to adapt to tight deadlines. First, the priority level submitted by programmers simply indicates the kernel's relative importance and does not indicate *when* the kernel must be completed. Second, priorities assigned to individual streams do not provide the GPU a global view of when to complete a chain of dependent kernels. Programmers conservatively set a job's priority to ensure that its deadline is met. Finally, jobs can have different amounts of work despite potentially having the same static priority level.

We propose to dynamically adjust the priorities of each job (and its associated queue) based on the job's estimated execution time. By adjusting the priorities, kernel launches are re-ordered to increase the number of jobs completed by their deadlines. The bottom half of Figure 3 demonstrates that with reasonably accurate execution time estimates, a deadline-aware scheduler can optimize the scheduling of deadline-sensitive jobs (similar to prior work for CPUs [74][75]). The bottom example begins like the top example, with the GPU scheduling J1 and J2 first, because they arrive earlier than J3. However, the LAX scheduler is aware of the deadlines and durations of all 3 jobs, so it prioritizes J3 since it will miss its deadline if not immediately scheduled (i.e., it has zero laxity). As a result, all jobs completed by the deadlines.

3 Latency-sensitive GPU Applications

This section characterizes important, latency-sensitive applications by their response time, level of parallelism and kernel composition. It then examines the tradeoff of increasing batch size versus the number of streams, and the impact of realistic arrival times on latency-sensitive GPU applications.

3.1 Applications

We study a wide group of latency-sensitive GPU applications that represent different use cases and access patterns to understand how they perform on contemporary GPUs.

3.1.1 Recurrent Neural Networks

RNNs are well suited for domains such as language translation [8][9] and speech recognition [10][11] where prior events persist and influence subsequent ones. RNNs contain loops that allow this information to persist across multiple iterations (or time steps). The number of times the loop is unrolled represents the RNN's sequence length, which varies across jobs and determines the length of the recurrent step. As a result, RNNs behave very differently than convolutional neural networks (CNNs) [4][5][7][44]. The hidden state is calculated by looking at the previous hidden state and the input at the current step. RNN models such as long-short-term-memory (LSTM) [32] and gated recurrent unit (GRU) [42] add memory cells to improve accuracy.

Each RNN time step contains multiple kernels with varying degrees of parallelism and execution time. As shown in Table 1, a single-batched LSTM job with a sequence length of 13 consists of 6 unique kernels and each kernel is called multiple times (we only show LSTM due to space constraints, Vanilla and GRU are similar). Unlike the training phase where latency is less critical [1][71]-[73], RNN inference jobs have real-time constraints [2][3][28][31][69]. It is challenging to fully utilize the GPU while minimizing the end-to-end latency of RNN inference applications.

Table 1: Summary of kernels in latency-sensitive benchmarks.

Applications	Kernel name	# of calls	Exec time	Threads	Context size
LSTM [12][13]	TensorKernel 1	3	3.96 µs	16384	397 KB
	TensorKernel 2	5	1.79 µs	128	3.1 KB
	TensorKernel 3	2	4.45 µs	2048	106.8 KB
	TensorKernel 4	40	4.74 µs	64	9.1 KB
	ActivationKernel 5	39	8.87 µs	128	11.1 KB
	rocBLASGEMMKernel 1	13	127.48 µs	1024	562.4 KB
IPV6 [66]	IPV6Kernel	1	25 µs	8192	329 KB
CUCKOO [66]	cuckooKernel	1	300 µs	8192	566 KB
GMM [65]	GMMKernel	1	1.5 ms	2048	195.5 KB
STEM [65]	STEMKernel	1	150 µs	4096	317 KB

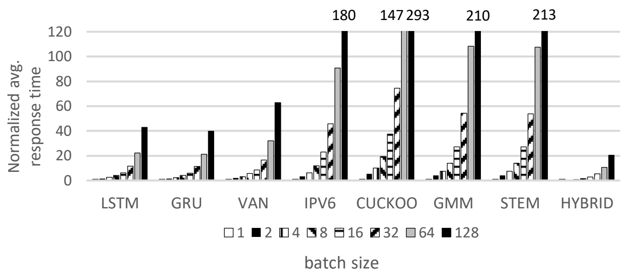


Figure 4: Comparing response times with varying job arrival rates, normalized to batch size 1.

3.1.2 Network Packet Processing

Network packet processing increasingly utilizes GPUs to take advantage of their massive parallelism. For example, IPV6 performs a Longest Prefix Matching computation used in IPV6 network packet table lookups and has a stringent 40 µs deadline [61][66]. Similarly, Cuckoo must complete cuckoo hash table lookups to map MAC address to output ports within 600 µs [61][66]. Unlike RNNs, these networking applications are composed of a single kernel, and their input sizes are determined by the speed of the network. In Table 1, the input size of 8K represents the number of network packets that arrived per 100 µs in 40 Gbps networks.

3.1.3 Intelligent Personal Assistants

IPAs also have significant real-time constraints. Although prior work explores a series of algorithms used in an automatic speech recognition (ASR) pipeline by IPAs, we focus on Gaussian mixture model (GMM) and Stemmer (STEM), two single kernel pieces that consume the most time in IPAs and thus present the biggest challenge [70]. GMM maps input feature vectors to multi-dimensional space and consumes 85% of ASR's computational time [65][70]. STEM reduces inflected words to a certain word stem and takes up to 85% of the remaining time in the ASR pipeline [65][70].

3.2 Small Data-Parallel Kernels

Table 1 characterizes each kernel in a single HIP [17][18] RNN LSTM inference job where its batch size is 1 and its hidden layer is 128. Both LSTM and GRU use 5 unique MIOpen [14] kernels and one rocBLAS [19] GEMM kernel that are called multiple times in an RNN forward pass. The MIOpen kernels perform tensor and activation operations. Each kernel has a varying number of threads. However, most kernels have few threads, and do not occupy the entire GPU.

The number of threads, registers, and LDS size of kernels determine the GPU utilization. In an AMD Radeon RX 580 GPU with 36 CUs based on the GCN architecture [22], each CU can concurrently execute 2560 threads, has 256 KB 32-bit vector registers, and has 64 KB of LDS. However, LSTM's GEMM kernel only uses 1.11% of thread contexts, 1.26% of registers, and 2.78% of the LDS space. The other LSTM kernels similarly use relatively few resources. Hence, a single RNN job significantly under-utilizes the GPU, as prior work has also shown for other sequence lengths, hidden sizes, and batch size combinations [27][69]. Moreover, although IPV6, Cuckoo, GMM, and STEM are single kernel applications, they also complete very quickly and have narrow kernels with few threads that also under-utilize the GPU.

3.3 Impact of Job Arrival Rate

In a real system, the GPU receives job requests from different users or processes with varying arrival rates. Batching improves GPU utilization and throughput when requests arrive at the same time. However, it will delay individual jobs when requests arrive at varying rates. Streams alleviate this aspect of batching by allowing work to begin as soon as it arrives.

Figure 4 measures our application's response time on an AMD Radeon RX 580 GPU. We use streams to launch 32K jobs for the networking and IPA benchmarks and 512 jobs for the RNN benchmarks based on our GPU's maximum memory space. For the RNNs, we also show data for Hybrid RNNs (described in Section 5.2). In this experiment, all streams use the same static priority. We issue 10000 short execution time jobs per second for IPV6, CUCKOO, and STEM and 1000 jobs per second for RNNs and GMM with an exponential arrival rate. Each RNN job may have a different sequence length (see Section 5.2). We add padding and additional waiting time for the arrival of all jobs in a batch when the batch size is greater than 1 as needed.

In general, the high degree of parallelism within large batches increases resource contention and job execution time. For example, the response time of applications with a batch size of 128 can be 20-293X slower than the single-batched job due to the overhead of waiting for additional jobs to arrive. Thus, larger batch sizes may improve utilization for these applications, but this often comes at the cost of not meeting its deadline. In contrast, using multiple streams reduces normalized runtime and allows the GPU to process multiple jobs simultaneously. However, closer inspection of these results reveals that individual job execution times vary tremendously. For example, RNN jobs with long sequence lengths complete much slower than RNN jobs with shorter sequence lengths. The observation exposes an opportunity for a more advanced GPU scheduler to prioritize longer running jobs and allow more overall jobs to meet a given deadline.

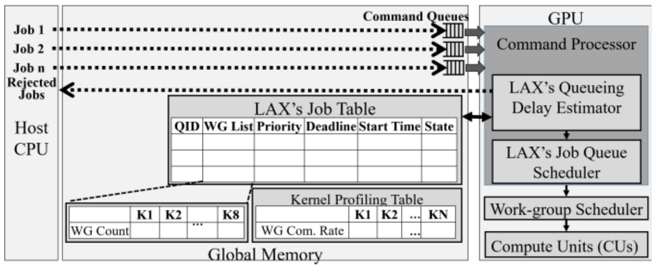


Figure 5: LAX procedure and system overview.

$$LaxityTime = Deadline - (TimeRemaining + DurationTime)$$

Equation 1: Laxity Time Calculation.

4 LAX: Laxity-Aware GPU Job Scheduling

4.1 LAX System Overview

Figure 5 presents an overview of our LAX framework. In multi-job GPU applications, all kernels associated with a single job are enqueued on the same stream or underlying GPU compute queue. Before running a job, as discussed in Section 1, LAX performs *stream inspection* to look ahead, parsing all the kernels in a queue to determine their names and associated number of WGs. For example, for the RNNs the sequence length determines the number of kernels in the job. To store this information, LAX introduces a *Job Table* that stores information about the work remaining in each compute queue. After parsing the WG information for every kernel in a queue, this information is added to the corresponding *WGList* for the queue. *Stream inspection* can be performed entirely in the CP, by reading the contents of the command queues, or by having the CPU driver insert a custom packet at the head of the queue indicating the work behind it.

LAX computes the estimated time remaining in each job using the number of WGs from the *WGList* and a per-kernel work completion rate stored in a *Kernel Profiling Table*. Given the estimated time remaining and a programmer-specified deadline (passed when initializing a job on a stream), LAX computes the laxity of each job using Equation 1.[1] The *LaxityTime* tells us how close to its deadline a job is predicted to finish. Jobs with less *LaxityTime* have higher priority.

Similar to prior work, LAX uses a pull-based model for offloading work from a CPU server [53][54][80]. As jobs arrive at the server, LAX successfully offloads as many jobs as possible. However, unlike prior work, LAX uses its per-job completion time estimates to generate a queuing delay estimate for new jobs entering the system. Based on current contention, if LAX estimates that the new job will not meet its deadline, it will not attempt to offload the job to the GPU.

4.1.1 *Implementing LAX*

Although modern GPUs have CPs, their software is not easily modified, and GPU vendors have not publicly released an API to reprogram them. Moreover, LAX requires fast access

[1] If additional work is later enqueued to the job's stream, LAX will update its prediction.

Algorithm 1: Steady State Queuing Delay Calculation.

```
1. totRemTime = 0
2. // new jobs are pushed to the end of the queue
3. For i = JobQ.begin() to JobQ.end()
4.   holdJobTime = 0
5.   durTime = curTick() - JobQ[i].startTime
6.   For (j = 0; j < JobQ[i].WGList.size; j++)
7.     kernelID = JobQ[i].WGList[j].kernelID
8.     If (JobQ[i].state != init)
          /* sum the total remaining time of jobs */
9.       totRemTime += JobQ[i].WGList[j].numWG
10.               / kernelTable[kernelID].WGCompRate
11.    Else /* initialize new job's estimate */
12.      holdJobTime += (JobQ[i].WGList[j].numWG /
            kernelTable[kernelID].WGCompRate)
13.    End /*end if*/
14.  End /*end for*/
15.  If (totRemTime + (holdJobTime + durTime) <
         JobQ[i].Deadline)
16.    /* New-invoked job's priority is the highest */
17.    If (JobQ[i].state == init) JobQ[i].prior = 0
18.    JobQ[i].state = readyState
19.    totRemTime += holdJobTime
20.  Else /* Cannot complete job in time, tell CPU */
21.    rejectJob()
22.  End /*end if*/
23. End /*end for*/
```

to custom performance counters since LAX frequently recalculates job priorities based on dynamic information (Sections 4.2 – 4.4). LAX also utilizes a customized per-kernel WG completion rate as part of its calculations (Section 4.2). Accordingly, we extend the CP as necessary. First, we extend the CP software to store the Job Table that tracks important information per kernel and make scheduling decisions based on information passed from the hardware. Next, we modify the GPU to add a new counter that tracks the WG completion rate and extend the existing counters to allow them to be frequently read by the CP, which stores and uses these counter values when making its scheduling decisions.

4.2 Job Remaining and Laxity Time Estimates

To estimate the laxity of currently executing jobs and the queuing delay for incoming jobs, LAX generates an estimate of the time remaining in each job with the help of an in-memory Job Table. As shown in Figure 5, each table entry has six fields: 1) QueueID (QID); 2) Priority (used by the CP to make scheduling decisions); 3) WGList (the list of the job's kernels and the number of WGs in each kernel); 4) Deadline (provided by the programmer); 5) StartTime; and 6) State (either init, ready, or running). In addition to the Job Table, LAX stores the per-kernel WG completion rates in a Kernel Profiling Table which is periodically updated (empirically set at 100 µs) to reflect the GPU's contention conditions. Overall, LAX requires only 4240 bytes of memory to store this information for a 128-compute queue system.

To predict a job's remaining time, LAX scans the WGList to generate an estimate for how long each kernel in the job will take. For each entry in the list, LAX looks up the current WG completion rate for this particular kernel in the Kernel Profiling Table. By dividing the number of WGs in each kernel by the current WG completion rate for that kernel type, LAX generates a time estimate for the kernel. The kernels in our evaluated jobs have sequential dependencies and thus must be executed sequentially, thus LAX simply sums the estimated execution time of each kernel to generate the per-job estimate. As WGs complete, the WGCount entry in the Job Table is decremented to reflect the fact that the job has less work remaining. LAX combines the job's remaining time estimate with the user-specified Deadline and the job's StartTime to generate the estimated LaxityTime. We describe the State and Priority fields in more detail in Sections 4.3 and 4.4.

4.3 Preventing Oversubscription with Queuing Delay Estimation

When designing a GPU to accept multiple jobs, each of which has a tight deadline, preventing oversubscription is critical. As discussed in Section 4.1, LAX only accepts jobs predicted to complete before their deadlines. To make this prediction, LAX must: (1) estimate how long a job *J* will take on the GPU under current conditions and (2) estimate how long *J* may be delayed behind other jobs already sent to the GPU, i.e. its queuing delay. Using each job's time remaining estimate from Section 4.2, LAX first computes how long *J* should take, given current completion rates. This allows LAX's estimates to adapt quickly and effectively to changing contention levels. If no estimate exists yet for a given kernel, LAX optimistically assumes it takes no time, to avoid rejecting work it could potentially complete.

Estimating *J*'s queuing delay is more challenging because the deadlines and arrival rates of latency-sensitive jobs vary significantly. However, Little's Law works well independent of arrival rate [30][50]. Thus, LAX uses Little's Law to model the queuing delay of the jobs running on the GPU. Accordingly, Algorithm 1 uses Little's Law to sum up the predicted remaining time of all jobs currently execution in the system, including jobs that are *ready* but not *running*. Combining this estimate with the runtime estimate, if LAX predicts *J* will complete by its deadline, it accepts *J* and changes its state from *init* to *ready*, informing the CP that *J*'s first kernel is ready to be executed on the GPU. Algorithm 1 shows the steady-state behavior; before enough WGs complete (line 12, Algorithm 1), we use the programmer-provided deadline.

4.4 Laxity-Aware Job Scheduling Algorithm

Next LAX needs to determine which job(s) should be run next. A job's laxity determines its priority in the laxity-aware job scheduler. The scheduler assigns each queue (job) a priority level and may adjust it over time. The job with the smallest current laxity is assigned the highest priority.

Algorithm 2 describes LAX's priority update mechanism, where priority zero is the highest priority level. Every 100 µs, the priorities are updated, as we empirically found this improved performance. Since we want to prioritize jobs with the least laxity, any job that is predicted to complete by its deadline is assigned its laxity value as its priority (Line 12, Algorithm 2). LAX decreases a job's priority when it predicts the job will not reach the deadline (Line 18, Algorithm 2). When

Algorithm 2: Laxity-aware Scheduling.

```
1. For i = JobQ.begin() to JobQ.end()
2.   JobQ[i].RemTime = 0
3.   For j = 0; j < JobQ[i].WGList.size; j++
4.     kernelID = JobQ[i].WGList[j].kernelID
5.     JobQ[i].RemTime += JobQ[i].WGList[j].numWG
6.       / kernelTable[kernelID].WGCompRate
7.   End /*end for*/
8.   JobQ[i].durTime = curTick() - JobQ[i].startTime
9.   ComplTime = JobQ[i].RemTime + JobQ[i].durTime
10.  If (JobQ[i].deadline > ComplTime)
11.    /*laxityTime = deadline - ComplTime*/
12.    JobQ[i].prior = JobQ[i].deadline - ComplTime
13.  Else
14.    JobQ[i].prior = ComplTime
15.  End /*end if*/
16.  /*deprioritize job if LAX cannot make deadline */
17.  If (JobQ[i].durTime > JobQ[i].deadline)
18.    JobQueue[i].prior = INF
19.  End /*end if*/
20. End /*end for*/
```

a job is predicted to miss a deadline, its *completionTime* (*remainingTime* + *durationTime*, where durationTime is the time since this job was enqueued) is greater than the deadline. To de-prioritize the job, LAX sets its priority to be equal to the *completionTime* (Line 14, Algorithm 2), because it is greater than the deadline, guaranteeing that the job has a lower priority than any other job that still has positive laxity.

After adjusting all job's priority, LAX issues all WGs from the highest priority job. If additional WG slots are available, it moves on to the next highest priority ready job, and so on until all WG slots are filled. After issuing the WGs from a kernel, LAX updates the associated job's status to *running*.

5 Methodology

We use the gem5 simulator [20][29], which offers native GPU ISA support [20]-[23] and a high fidelity, cycle-level GPU microarchitecture model [29] to evaluate the latency-driven applications (Table 2). The simulated system assumes the CPU and GPU share a single unified cache coherent address space and do not require explicit copies [77]. Since the original benchmark's codes assume discrete memory spaces between the CPU and GPU and use device copies, we modified the benchmarks to remove the device copies wherever possible. We analyze energy consumption with per-instruction energies [6][81]. Finally, we assume the CP can parse four streams in parallel every 2 µs [29][48], including the latency of any memory accesses required by the scheduler.

5.1 Evaluated Compute Queue Scheduling Policies

We compare our laxity-based scheduler against ten other queue scheduling policies, which are detailed in Table 3 and implemented in gem5. These schedulers leverage various policies with static and dynamic information to schedule kernels and can be broken into three groups: state-of-the-art CPU-side schedulers [28][53][54], GPU approaches that extend the CP, and variants of our laxity-based scheduler.

CPU-side scheduling mechanisms such as BAT [28], BAY [54], and PRO [53] (see Table 3 for details) improve throughput without requiring hardware changes. However, BAT,

Table 2: Key simulated system parameters.

GPU Clock	1500 MHz
The number of CUs	8
Number of SIMD units per CU	4
Max wavefronts per SIMD unit	10
Vector register size per CU	256KB
The number of compute queues	128
CPU Clock	4000MHz
# CPUs	2
GPU L1-D$ per CU	16 KB, 64B line
GPU L1-I$ per 2 CUs	32KB, 64B line, 16 way
GPU L2 cache per 64 CUs	4MB, 64B line
Main Memory	16 GB DDR4 [92], 16 channels, 16 banks/channel, 1000 MHz

Table 3: Scheduling Policies.

Scheduler	Description
Prior CPU-Side Scheduling	
BatchMaker (BAT) [28]	A dynamic batching technique where each stream can have a different batch size.
Baymax (BAY) [54]	Uses pre-trained models to predict a jobs execution time and re-orders the priorities of jobs based on their QoS headroom.
Prophet (PRO) [53]	Uses offline profiling to choose which concurrent jobs to issue in order to fully utilize the GPU and improve QoS.
Contemporary GPU Command Processor Scheduling	
Round-Robin (RR)	The baseline scheduler that processes compute queues in a cyclic manner.
Advanced GPU Command Processor Scheduling	
Multi-Level Feedback Queue (MLFQ) [64]	Moves jobs between two priority queues based on their runtime and uses RR to schedule jobs in the high priority queue.
Earliest Deadline First (EDF) [91]	A dynamic scheduling policy that schedules kernels from the job with the earliest deadline first.
Shortest-Job First (SJF)	A static scheduling policy that schedules kernels with the shortest job first.
Shortest Remaining Time Job First (SRF)	A dynamic policy that uses LAX's remaining execution time estimator to assign job priorities. It then assigns the job with the shortest estimated remaining time the highest priority.
Longest-Job First (LJF)	A static scheduling policy that schedules kernels from the longest jobs first.
PREMA [79]	A multi-task scheduler for heterogeneous systems that predicts job priorities and preempts lower priority jobs.
Proposed Laxity-Aware Scheduling Variants	
LAX	Our laxity-aware scheduling policy described in Section 4.
LAX-SW	A variant of LAX that uses CPU-side scheduling.
LAX-CPU	A variant of LAX that does CPU-side scheduling but changes the API to allow rapid changing of the priority of the jobs.

BAY, and PRO incur overheads for communicating between the CPU and GPU. For a tightly coupled GPU like the one in our system, this adds 4 µs of host-device communication overhead per kernel in a job. Similarly, we add 50 µs of overhead to BAY for calls to its regression model, based on reported data [54].

Modern GPUs perform deadline-blind RR scheduling (Section 2.1), but since the CP is programmable (although GPU vendors have not disclosed an API), it is possible to extend the CP for other widely used schedulers like LJF, MLFQ, SJF, and SRF. Like LAX, SJF, SRF, and LJF utilize predicted runtime information to decide what to schedule; however, they do not model queuing delay or laxity. MLFQ performed better with two priority levels, demoting jobs to lower priority level [64] when runtime exceeds 1/3 of the jobs' deadline, and promoting back to the higher priority when its runtime exceeded 2/3 of its deadline. We also compare against EDF [91], which prioritizes the job with the earliest deadline. Although some EDF implementations strictly ensure that the job(s) with the earliest deadline is always executing, this requires preemption. For jobs with longer deadlines such as those studied in prior work [91], this context switching overhead can be amortized. However, given the short deadlines in our workloads and the fine-grained scheduling granularity we use, strict EDF with preemption would perform poorly. For example, prior work assumes preemption incurs around 1 ms of overhead [91], which exceeds some of our workload's deadlines (CUCKOO, IPV6, and STEM) and consumes a significant portion of the deadline for the remainder (GMM and the RNNs). Thus, we instead implement EDF by prioritizing jobs with the earliest deadlines first, without preemption.

Finally, we compare against PREMA, which utilizes user-defined prioritizes and slowdown calculations to preempt lower priority jobs [79]. Like the authors, we use a 250 µs preemption interval. Although PREMA was designed for TPUs running a single, large job, we extended it to run multiple jobs since our workloads do not fully utilize the GPU. We also extended PREMA to use LAX's frequent updates for PREMA's calculations.

Finally, since LAX changes multiple components (Section 4), we also design the three variants to identify if laxity-aware scheduling could provide the same benefits without extending the CP: LAX, which extends the CP as discussed in Section 4; LAX-SW, which performs CPU-side scheduling (and incurs overheads for host-device communication); and LAX-CPU, which also does CPU-side scheduling, but changes the API to allow dynamic job priority updates from user-level software. To do this, we update the API to write updated priorities to memory-mapped registers that control each queue's priorities [29]. Finally, for all LAX variants we initialize the job priority to the highest priority, as this empirically gave the best results.[2]

5.2 Benchmarks

To evaluate the schedulers, we use the eight latency-sensitive benchmarks discussed in Section 3. Table 4 details their input size, deadline, and arrival rates. Where available, we use deadlines from recent work: 7 ms for RNNs [2][3][28][31], 40 µs for IPV6 [61][63], and 600 µs for Cuckoo [61]. For the many-kernel RNNs, deadlines are set for the entire multi-layer computation. For the IPA benchmarks, we used the same methodology as the authors: we ran each benchmark in isolation, then doubled the worst case latency [53][54]. LAX does not affect latency-insensitive applications because the programmer does not provide a deadline for them.

To demonstrate how GPUs can simultaneously execute kernels with different degrees of parallelism, we also include a Hybrid RNN benchmark that includes the two most popular RNN variants, LSTM and GRU, with a mixed hidden layer size of 128 and 256, respectively. The input for all RNNs is based on the WMT '15 language translation trace [47], which has an average sequence length of 16. Furthermore, we share weight data across RNN inference jobs with the same hidden size [6][28]. Although our technique is applicable to any data

Table 4: LAX Benchmarks.

Benchmark	Deadline	Input / hidden layer size	High Job Arrival Rate (jobs/s)	Medium Job Arrival Rate (jobs/s)	Low Job Arrival Rate (jobs/s)
Many Kernels per Job					
LSTM [12][13]	7 ms	128	8000	5000	3000
GRU [12][13]	7 ms	128	8000	5000	3000
VAN [12][13]	7 ms	256	8000	5000	3000
HYBRID [12][13]	7 ms	128/256	8000	5000	3000
Few Kernels per Job					
IPV6 [61]-[63]	40 µs	8192	64000	32000	16000
CUCKOO [61]-[63]	600 µs	8192	8000	5000	3000
GMM [65][70]	3 ms	2048	32000	16000	8000
STEM [65][70]	300 µs	4096	64000	32000	16000

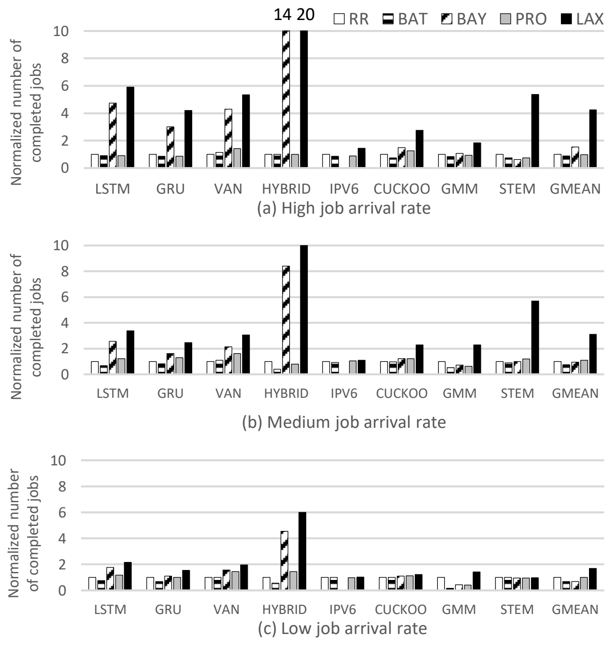


Figure 6: Jobs completed by their deadlines for CPU-side schedulers, RR, and LAX, normalized to RR.

width, we use DeepBench's provided precision for the RNNs. Additionally, our schedulers do not affect the RNN inference accuracy since they do not change the underlying algorithms, just how they are scheduled (as described in Section 4).

5.3 Job Arrival Rate

We simulate 128 jobs per benchmark with different arrival times and map one job to one GPU stream. Our simulated server only processes one type of job at a time, similar to modern datacenters [78]. Real world systems continually receive requests with varying arrival rates. As with determining the deadlines, wherever possible we used the same arrival rates as previous work. For CUCKOO, GMM, and STEM, we modified these rates to account for the difference in system size. Moreover, we sweep multiple levels of contention (high, medium, and low arrival rates) for each benchmark to evaluate how contention affects the scheduler, where jobs with fewer kernels have faster arrival rates due to their shorter

[2] Initializing each job with the lowest priority or running an initial laxity estimate upon each job's arrival degraded performance by 10% and 1% on average, respectively, compared to initializing with the highest priority.

deadlines. For each arrival rate, we randomly generate specific job arrival times based on an exponential distribution.

6 Experimental Results

Overall, LAX successfully offloads more jobs than prior approaches. At the highest arrival rate LAX completes a geometric mean (geomean) of 2.8X – 4.8X and 1.7X – 5.0X more jobs by their deadlines than CPU-side schedulers and schedulers that extend the CP, respectively. Moreover, CPU-side laxity-aware scheduling outperforms other CPU-side schedulers but requires CP extensions to obtain laxity's full benefits. Finally, LAX wastes less work, accurately predicts job laxity, provides a better combination of energy consumption and performance, and provides a better combination of throughput and 99-percentile latency.

6.1 Completing Jobs by Their Deadlines

6.1.1 CPU-Side Schedulers

For the CPU-side schedulers, RR, and LAX, Figure 6 plots the number of jobs successfully offloaded to the GPU for each arrival rate, normalized to RR. In general, most schedulers do well for the lower arrival rates, where contention is low. At the high job arrival rate, contention increases, and all schedulers start missing more deadlines.

RR: As expected, RR does not do very well because it schedules jobs in deadline-blind fashion. However, for few kernel benchmarks (IPV6, CUCKOO, GMM, and STEM), which also have equal job sizes, RR does better, especially at higher arrival rates, because a new job will sometimes be chosen to run soon if RR is near the end of the queue when the job is added, reducing queuing delay. Although this also occurs for the jobs with many kernels, since these jobs may have inter-kernel dependency chains, the benefit is smaller.

BAT: BAT dynamically combines kernels in a batch. When jobs arrive simultaneously, and are executing the same kernel, this significantly improves efficiency. However, BAT executes these kernels in a lock-step manner and is not aware of the job's deadlines. As a result, BAT performs poorly for many of these latency-sensitive workloads, especially as contention increases. Overall, BAT completes a geomean of 23% fewer jobs than RR by their deadlines.

BAY: BAY generally outperforms deadline-blind schedulers like RR and BAT by effectively predicting the execution time of jobs and using its QoS headroom calculations to control the number of concurrent jobs. However, BAY's 50 µs prediction overhead (Section 5.1) prevents it from completing any IPV6 jobs by their 40 µs deadlines – which significantly decreases BAY's overall performance such that RR and BAY complete the same geomean number of jobs by deadline. Otherwise, BAY is the top performing CPU-side scheduler for latency-sensitive workloads. Compared to LAX, the host-device and prediction overheads hamper BAY's ability to dynamically respond, especially at the high arrival rate for applications with many kernels, where LAX's accurate queuing

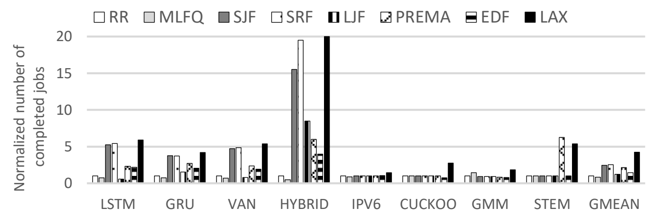


Figure 7: Jobs completed by their deadlines at the high job arrival rate, for schedulers that extend the CP, normalized to RR.

delay estimate and increased responsiveness help it complete a geomean 3.1X more jobs than BAY by their deadlines.

PRO: PRO leverages offline profiling to infer the QoS of kernels, which reduces prediction overhead compared to BAY. However, since PRO focuses on co-scheduling memory- and compute-intensive workloads, it suffers with the purely latency-sensitive workloads we are studying. As a result, it only completes a geomean of 1.02X more jobs by their deadlines than RR. As contention increases, PRO especially suffers for LSTM, GRU, and GMM, where the increased contention exacerbates its focus on co-scheduling.

LAX: LAX completes a geomean of 1.7X, 3.1X, and 4.2X more jobs by their deadlines compared to RR, respectively, for the low, medium, and high arrival rates. Unlike other schedulers, LAX utilizes the laxity of jobs, which increases the number of medium and large size jobs it can complete by their deadlines, especially as contention increases. Additionally, extending the CP helps LAX adjust more quickly and accurately to dynamically changing conditions. Finally, LAX's accurate queuing delay model helps it avoid oversubscription. Thus, the combination of accurate queuing delay modeling, rapid, accurate responsiveness, and laxity allow LAX to significantly outperform the CPU-side schedulers.

Overall, LAX significantly outperforms state-of-the-art CPU-side schedulers for both many- and few-kernel workloads. Although some of these schedulers also model job runtime or utilize QoS calculations to avoid oversubscription, LAX's combination of laxity, rapid responsiveness, and accurate queuing delay modeling help it successfully offload more jobs, especially for jobs with fewer kernels and deadlines < 1ms. We focus on the high arrival rate since it magnifies the differences between the schedulers.

6.1.2 Extending the Command Processor Schedulers

For each scheduler that extends the CP (Section 5), Figure 7 compares the number of jobs completed by their deadlines.

SJF and SRF: SJF and SRF greedily schedule kernels from the shortest jobs (e.g., RNN jobs with the shortest sequence lengths). As a result, SJF and SRF complete 2.46X and 2.54X more jobs by geomean, respectively, over RR at the highest job arrival rate. However, SJF and SRF perform poorly for benchmarks with fewer kernels per job because all jobs have the same input size. This causes SJF and SRF to default to first-come-first-serve (FCFS) order, so queuing delay dominates for these applications. Nevertheless, exploiting runtime information allows SJF and SRF to complete more jobs than any other schedulers beside LAX. Moreover, compared to the CPU-side schedulers, extending the CP allows SJF and SRF's to improve performance over BAY, the top performing CPU-side scheduler, by 1.6X at the highest arrival rate.

MLFQ: MLFQ performs poorly – only geomean 0.85X jobs complete by their deadlines compared to RR. For both many- and few-kernel jobs like RNNs, CUCKOO, and IPV6, MLFQ completes relatively few jobs because once long-running jobs get promoted back to the higher priority queue, they take up high priority resources even after their deadline [67]. However, in GMM and STEM, deprioritizing jobs long running jobs (e.g., from queuing delay), schedules newer jobs sooner.

EDF: By greedily scheduling the job with the next deadline, EDF completes geomean 1.5X more jobs than RR. However, EDF performs poorly for jobs with uniform deadlines and varying lengths (e.g., the RNNs). LAX uses the work remaining in jobs to dynamically adjust job priorities and complete 2.9X more jobs by their deadlines. Thus, by considering both remaining work and job deadline, LAX outperforms EDF, which only considers job deadline.

LJF: Compared to RR, LJF completes 1.24X more jobs by their deadlines because it reorders jobs and schedules the longest jobs (e.g., RNN jobs with many kernels and long sequence lengths) first. Although this allows some longer jobs to complete by the deadline, in general LJF does not perform well because it sacrifices the smaller jobs to complete longer ones (for jobs like the RNNs with different sized jobs).

PREMA: PREMA's user-defined priorities and slowdown calculations help it complete geomean 2.2X more jobs than RR. PREMA performs particularly well for the low-latency (250 µs) jobs, like STEM. However, overall LAX completes a geomean 2.0X more jobs than PREMA because LAX predictively uses WG completion and queuing delay estimates to make more accurate predictions, while PREMA reactively predicts based on feedback from running jobs.

Overall, extending the CP can significantly improve the number of jobs that meet their real-time deadlines versus CPU-side schedulers, especially for CP schedulers that are able to predict the remaining runtime or amount of work. However, these advantages alone are insufficient: LAX completes a geomean of 1.7X more jobs by their deadlines than SJF and SRF (the next highest performing CP schedulers) because it also utilizes laxity and an accurate queuing delay model to better schedule the jobs. LAX outperforms all other schedulers except on STEM, indicating that a hybrid solution which combines elements of LAX and PREMA could be interesting future work. However, this may complicate the design for relatively small gain, since LAX also outperforms PREMA in terms of energy (Section 6.4), throughput (Section 6.5) and tail latency (Section 6.5).

6.1.3 *Is CPU-Side LAX Scheduling Sufficient?*

Figure 8 compares the number of jobs completed by their deadlines for the three laxity-aware schedulers. Although

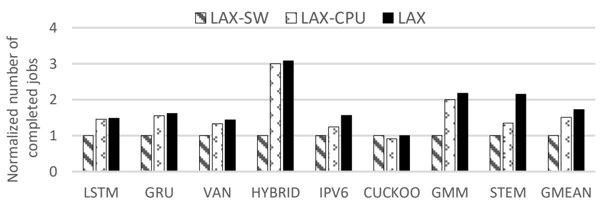
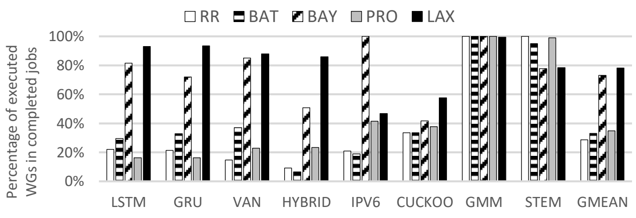
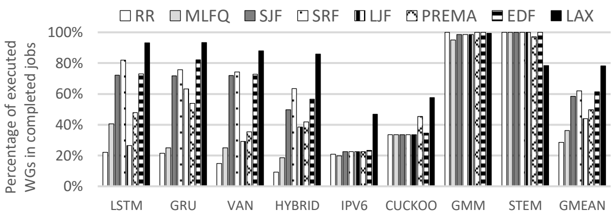


Figure 8: Jobs completed by their deadlines over different laxity-aware implementations, normalized to LAX-SW.



(a) CPU-side schedulers, RR, and LAX



(b) Schedulers that extend CP

Figure 9: Percentage of completed WGs from jobs that meet their deadlines at the high job arrival rate.

LAX-SW suffers from host-device overheads like BAT, BAY, and PRO, and is neither obtains nor rapidly responds to GPU information as quickly as the CP schedulers, it still performs well. BAY, the top performing CPU-side scheduler (Figure 6), outperforms LAX-SW for jobs with many kernels and deadlines > 1 ms (GMM and the RNNs) by 26%. However, for the jobs with fewer kernels and deadlines < 1 ms (IPV6, CUCKOO, and STEM), LAX-SW successfully offloads significantly more jobs due to its more accurate queuing delay model. Overall, LAX-SW completes geomean 1.8X more jobs by their deadlines than BAY. Thus, LAX-SW improves on the state-of-the-art even without hardware support.

LAX-CPU and LAX successfully offload 1.5X and 1.7X more jobs, respectively, than LAX-SW. Interestingly, LAX-CPU, where applications use a user-level API to dynamically adjust job priorities, provides most of LAX's benefits. Overall, LAX completes a geomean 1.1X more jobs than LAX-CPU, because it responds more rapidly and has access to higher fidelity information. Thus, to obtain all the benefits of laxity-aware scheduling, extending the CP is necessary, although API changes can provide most of the benefits.

6.2 Scheduling Effectiveness

To measure how efficiently the schedulers utilized GPU resources, Figure 9 plots the percentage of the WGs completed that are part of jobs that meet the deadline. This metric shows how effective the schedulers were at identifying and perform-

Table 5: The job throughput, latency, and energy.

	RR	MLFQ	BAT	BAY	PRO	LJF	SJF	SRF	PREMA	EDF	LAX
LSTM	511	419	458	2651	465	372	2883	3069	1302	1209	3317
GRU	912	700	775	2828	775	1551	3466	3558	2463	1870	3859
VAN	729	515	750	2574	987	472	2832	2960	1416	1158	3226
HYBRID	85	43	85	1147	85	766	1277	1702	511	340	1757
IPV6	13158	13816	11842	0	13816	13158	13158	13158	12500	13157	23953
CUCKOO	289	289	276	651	295	289	289	289	289	289	831
GMM	2242	2841	2242	2446	2242	2242	2242	2242	1921	2038	4646
STEM	3937	3937	2624	1969	2624	3937	3937	3937	23622	3937	20954

(a) Successful job throughput (# of successful jobs per second)

	RR	MLFQ	BAT	BAY	PRO	LJF	SJF	SRF	PREMA	EDF	LAX
LSTM	47.7	38.2	51.9	21.4	6.7	50.1	46.4	46.3	43.2	37.8	6.0
GRU	35.1	25.6	37.9	20.4	6.5	36.9	33.7	33.4	27.6	25.7	6.5
VAN	43.9	34.2	38.7	9.4	7.0	47.0	43.6	42.9	38.7	34.9	6.6
HYBRID	84.5	75.7	88.4	20.9	2.4	85.7	81.9	83.9	83.7	75.6	7.2
IPV6	0.2	0.2	0.2	0.0	0.4	0.2	0.2	0.2	0.2	0.2	0.04
CUCKOO	9.7	9.0	9.2	1.0	1.3	9.2	9.2	9.2	9.4	9.2	4.5
GMM	41.5	42.3	42.2	3.3	1.8	42.2	42.2	42.2	40.2	42.3	2.8
STEM	3.1	3.1	3.2	0.3	0.3	3.1	3.1	3.1	4.8	3.1	0.5

(b) 99-percentile job latency (ms)

	RR	MLFQ	BAT	BAY	PRO	LJF	SJF	SRF	PREMA	EDF	LAX
LSTM	1.35	1.80	1.47	0.08	0.08	2.32	0.26	0.25	0.58	0.62	0.08
GRU	0.58	0.78	0.69	0.07	0.06	1.30	0.21	0.21	0.43	0.53	0.08
VAN	0.72	0.96	0.90	0.07	0.08	1.30	0.21	0.21	0.43	0.53	0.08
HYBRID	15.4	31.19	15.39	0.21	0.36	1.65	0.89	0.74	2.53	3.94	0.15
IPV6	0.014	0.016	0.014	0.00	0.014	0.014	0.014	0.014	0.014	0.014	0.007
CUCKOO	0.78	0.78	1.04	0.05	0.05	0.79	0.79	0.79	0.79	1.05	0.12
GMM	2.35	1.62	2.78	0.14	0.20	2.55	2.55	2.52	2.75	3.13	0.21
STEM	0.12	0.12	0.16	0.011	0.009	0.08	0.08	0.08	0.21	0.12	0.008

(c) Energy rate (consumed energy over # of successful jobs) (mJ)

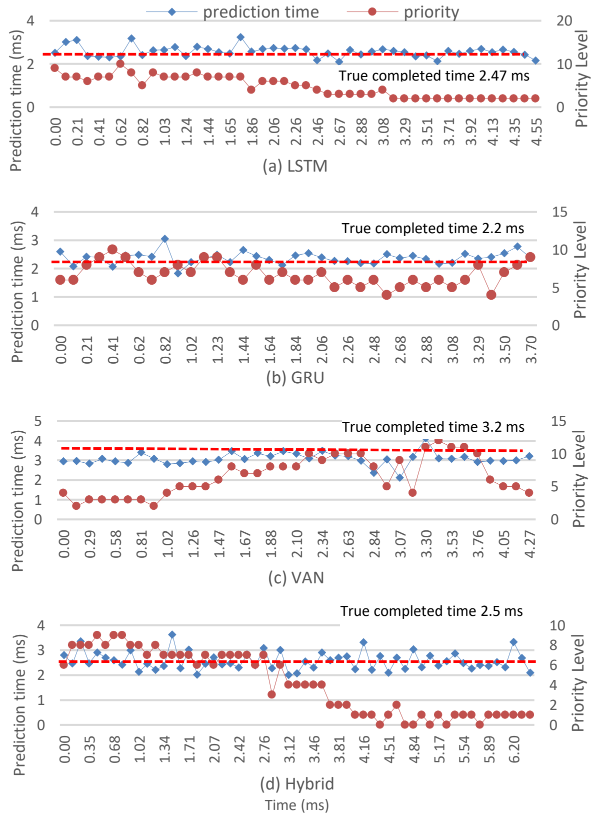


Figure 10: LAX's Job Time and Priority Prediction. P0 is the highest priority. Prediction has a mean absolute error of 8%.

ing useful work. Unsurprisingly, the deadline-blind schedulers (RR, BAT) waste a geomean of 67% - 71% of their resources on jobs that will not make the deadline, reinforcing their poor performance in Section 6.1.1. BAY's QoS prediction model reduces contentions and wastes fewer compute resources (27% geomean). Finally, PRO wastes geomean 65% of its effort on jobs that cannot make their deadlines. In particular, PRO struggles with the RNNs. PRO has conservative QoS estimates that do not consider overlapping kernels.

Since SJF and SRF issue small jobs first, they waste less work than deadline-blind schedulers (only 41% and 38%, respectively). Intuitively, since LJF schedules large jobs first, which are less likely to be completed, LJF wastes more work (56% at the highest job arrival rate). LAX's queuing delay model helps it waste the least work of all schedulers – a geomean 22% of compute resources.

6.3 Execution Time Prediction & Priority Over Time

To examine how well LAX's execution time predictions track over time, Figure 10 plots the the predicted job execution time and the priority of a sample job versus time for each of the 4 RNN workloads. The dashed red line is the job's actual execution time (i.e., time the job is in the *running* state, where its WGs were actively being executed). The x-axis indicates the duration time of the job and the endpoint of the x-axis shows the job's actual completion time (i.e., time in *ready* and *running*). Initially, the LSTM job's priority stays relatively steady until its laxity starts to decrease. For the Hybrid RNN (d), which is more computationally intense, the job's priority starts off very low, then increases toward P0 as the job gets closer to the deadline, finishing the job just before its 7ms deadline. This shows that LAX successfully deprioritizes jobs when they still have plenty of slack, then correctly prioritizes them once their slack is small. Additionally, LAX's execution time prediction tracks very closely to its actual time in the running state. To varying degrees, this trend holds for the other RNNs in (b) and (c) as well. Overall, these results show that LAX effectively varies the dynamic priority of these workloads and tracks the slack effectively, even when contention is high.

6.4 Energy Consumption

Table 5 compares the schedulers normalized energy consumption per successful job. In general, LAX provides comparable or better energy consumption relative to most CPU-side schemes (0.9X – 13.0X geomean less energy) and schedulers that extend the CP (4.3X – 13.2X geomean less energy). LAX outperforms all schedulers in this regard except for BAY (10% less energy per job than LAX, respectively). However, BAY and PRO are overly conservative and do not accept larger jobs that consume more energy, whereas LAX completes many more small and large jobs (Section 6.1.1).

6.5 Throughput and 99-percentile Tail Latency

Table 5 also shows the scheduler's throughput and 99-percentile tail latency. Overall, LAX provides a better blend of throughput and tail latency. LAX has better or comparable tail latency than CPU-side schemes (0.8X-7.2X geomean faster) and has geomean 1.25X-7.2X better throughput. Moreover, LAX's throughput is 1.1X-8.9X better than the CP schedulers and has 5.6X–7.3X better tail latency. BAY and PRO provide better throughput than LAX – their queueing models avoid offloading jobs that are unlikely to be completed by their deadlines. However, PRO and BAY complete far fewer jobs by their deadlines than LAX (Section 6.1.1).

7 Related Work

Improving Application Latency on Accelerators: Table 6 compares LAX to related work across several key metrics. Recent work optimized GPUs and accelerators for latency-sensitive applications like ML algorithms. At the architecture level, these optimizations include distributing and pipelining RNNs across FPGAs [2], compressing weights [6], increasing batch size and adding special purpose functional units [3], designing custom accelerators from domain-specific languages to improve memory access latency [68] [69][78], and moving shared weights on-chip [25]-[27]. At the software and system levels, prior work preemptively schedules kernels [56]-[59], increases data reuse [31], dynamically combines same-sized RNN cells [28], or uses persistence [27][55]. Although these solutions provide some of LAX's features, they focus on different problems.

QoS or Priority-Aware Scheduling Policies: Recent work applied QoS and prioritization to GPUs. The most relevant related work is Baymax [54], Prophet [53], and PREMA [79]. Baymax pre-trains regression models to predict job execution time, then uses its predictions to adjust job priorities to prevent latency-sensitive jobs from missing QoS targets. Prophet

Table 6: Comparing LAX with other prior work.

	Baymax [54]	Prophet [53]	BatchMaker [28]	TimeGraph [40]	GPUSync [41]	BrainWave [2]	EIE [6]	TPU [3]	GPU Preemption/ Reexecution [56]-[59]	PREMA [79]	LAX
Deadline awareness	✓	✓		✓	✓					✓	✓
Estimates job's execution time	✓	✓								✓	✓
Frequent job priority updates										✓	✓
Improve utilization	✓	✓	✓					✓	✓		✓
Improve latency	✓	✓	✓	✓		✓	✓	✓	✓	✓	✓
Concurrent execution	✓	✓	✓						✓		✓
Avoid host communication						✓	✓	✓			✓
Avoid preemption overhead	✓	✓	✓	✓	✓	✓	✓	✓			✓
Integrated GPU scheduler											✓

[53] uses offline profiling and prediction models to co-locate kernels and improve GPU utilization and QoS. Wang et al. measure the GPU's IPC to provision GPU resources and meet QoS targets [60]. Although this work provides some of LAX's features, it relies on software-only, CPU-side schedulers, whereas LAX extends the GPU's CP to better respond to dynamic changes in behavior and avoid host-device overheads. PREMA [79] uses user priorities and slowdown calculations to estimate execution time, but focuses on single jobs and suffers from preemption overhead. Other work adds QoS support at the memory controller [67][68], OS- or hypervisor-level scheduling [87]-[90], or uses similar profiling and prediction mechanisms to BAY, PRO, or PREMA [82]-[86][93]. Thus, LAX's provides similar benefits over them.

Real-time Scheduling: Embedded and real-time systems have also utilized laxity [46], and prior solutions use laxity on CPUs [33]-[36][74][75]. Others use prioritization on GPUs [37]-[41]. EVDZL applies laxity to mobile GPUs, but assumes offline profiling and oracular knowledge, unlike LAX which uses dynamic, online information to determine what jobs to schedule [94]. Other work preempts lower priority kernels in order to execute higher priority kernels [56][59]. However, preemption schemes are usually guided by the operating system and have high overhead on GPUs due to their amount of context state [56]-[58]. Furthermore, communication latency between the OS and GPU makes fine-grained updates difficult. In comparison, as shown in Table 6, LAX dynamically adjusts job priorities. Prior CPU-side work such as backfilling also exploits similar ideas [75], including predicting job runtime based different job's runtimes [74]. Although these CPU-side ideas utilize similar underlying concepts, they suffer from the same inefficiencies as other CPU-centric solutions.

Modern GPUs allow programmers to provide limited priority information for jobs in different queues [15][16]. However, this information is static and associated with an individual kernel, thus the scheduler cannot determine how its priority relates to the global situation. LAX mitigates these issues by transparently enhancing the queue scheduler to dynamically change job priorities based on deadlines.

8 Conclusion

To address the inefficiency of executing latency-sensitive workloads on GPU, we propose a new kernel scheduler, LAX. By tracking the WG completion rates and monitoring the queuing delay, LAX accurately estimates the overall execution of individual latency-sensitive jobs. Our results show that LAX completes a geomean of 1.7X-5.0X more jobs by their deadlines compared to ten GPU queue schedulers, while also having the better combination of both energy and performance, as well as throughput and 99-percentile tail latency.

Acknowledgment

The authors would like to thank Sooraj Puthoor and Michael LeBeane for their contributions to gem5 command processor model used in our evaluation. This work was supported, in part, by NSF CCF #1910924 and MOST 109-2222-E-009-009-MY2.

AMD, the AMD Arrow logo, and combinations thereof are trademarks of Advanced Micro Devices, Inc. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

References

[1] A. Jain, et al. Gist: Efficient Data Encoding for Deep Neural Network Training. In *ISCA*, pp. 776-789, 2018.

[2] J. Fowers, et al. A Configurable Cloud-Scale DNN Processor for Real-Time AI. In *ISCA*, pp. 1- 14, 2018.

[3] N. P. Jouppi, et al. In-Datacenter Performance Analysis of a Tensor Processing Unit, in *ISCA*, pp. 1-12, 2017.

[4] J. Yu, et al. Scalpel: Customizing DNN Pruning to the Underlying Hardware Parallelism. In ISCA, pp. 548-560, 2017.

[5] J. Albericio, et al. Cnvlutin: ineffectual-neuron-free deep neural network computing. In *ISCA*, pp. 1-13, 2016.

[6] S. Han, et al. EIE: efficient inference engine on compressed deep neural network. In *ISCA*, pp. 243-254, 2016.

[7] K. Hazelwood, et al. Applied Machine Learning at Facebook: A Datacenter Infrastructure Perspective. In *HPCA*, pp. 620-629, 2018,.

[8] D. Britz, et al. Massive Exploration of Neural Machine Translation Architectures, in *EMNLP*, pp. 1442-1451, 2017.

[9] Y. Wu, et al. "Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation," arXiv preprint arXiv: 1609.08144, 2016.

[10] D. Amodei, et al. "Deep speech 2: end-to-end speech recognition in English and mandarin." In *ICML*, pp. 173-182, 2016.

[11] A. Hannun, et al. "Deep Speech: Scaling up end-to-end speech recognition," arXiv preprint arXiv: 1412:5567, 2014.

[12] S. Narang. DeepBench. https://svail.github.io/DeepBench/. 2016.

[13] S. Narang and G. Diamos. An update to DeepBench with a focus on deep learning inference. https://svail.github.io/DeepBench-update/. 2017.

[14] J. Khan, et al. "MIOpen: An Open Source Library For Deep Learning Primitives", arXiv preprint arXiv: 1910.00078, 2019.

[15] NVIDIA, CUDA Stream Management. http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/online/group__CUDART__STREAM.html, 2018.

[16] J. Luitjens. CUDA Streams: Best Practices and Common Pitfalls. In *GTC*, 2014.

[17] HIP: Heterogeneous-computing Interface for Portability. https://github.com/ROCm-Developer-Tools/HIP/.

[18] AMD. It's HIP to be Open. https://www.amd.com/Documents/HIP-Datasheet.pdf.

[19] AMD. rocBLAS library document. https://rocm-documentation.readthedocs.io/en/latest/ROCm_Tools/rocblas.html.

[20] N. L. Binkert, et al: The gem5 simulator. SIGARCH Computer Architecture News 39(2): pp. 1-7, 2011.

[21] AMD Radeon Technology Group. Radeon's next-generation Vega architecture. https://www.techpowerup.com/gpu-specs/docs/amd-vega-architecture.pdf, November 2017.

[22] AMD. AMD Graphics Core Next (GCN) Architecture. https://www.techpowerup.com/gpu-specs/docs/amd-gcn1-architecture.pdf, June 2012.

[23] AMD. Graphics Core Next Architecture, Generation 3. http://developer.amd.com/wordpress/media/2013/12/AMD_GCN3_Instruction_Set_Architecture_rev1.1.pdf. August, 2016.

[24] J. Appleyard, et al. "Optimizing Performance of Recurrent Neural Networks on GPUs," arXiv preprint arXiv: 1604.01946, 2016.

[25] S. Narang, et al. Exploiting Sparsity in Recurrent Neural Networks, in *ICLR*, 2017.

[26] F. Zhu, et al. Sparse Persistent RNNs: Squeezeing Large Recurrent Neural Networks On-Chip, in *ICLR*, 2018.

[27] G. Diamos, et al. Persistent RNNs: Stashing Recurrent Weights On-Chip. In *ICML*, pp. 2024-2033, 2016.

[28] P. Gao, et al. Low latency RNN inference with cellular batching, in *EuroSys*, pp. 1-15, 2018.

[29] A. Gutierrez, et al. Lost in Abstraction: Pitfalls of Analyzing GPUs at the Intermediate Language Level, in *HPCA*, pp. 608-619, 2018.

[30] J. D. C. Little and S. C. Graves, (2008) Little's Law. In: Building Intuition. International Series in Operations Research & Management Science, vol 115, pp 81-100.

[31] M. Zhang, et al. DeepCPU: serving RNN-based deep learning models 10x faster, in *USENIX ATC*, pp. 951-965, 2018.

[32] S. Hochreiter and J. Schmidhuber. "Long short-term memory." Neural computation 9, no. 8 (1997): 1735-1780.

[33] C. Laung Liu and J. W. Layland. "Scheduling algorithms for multiprogramming in a hard-real-time environment." *JACM*, vol. 20, no. 1, pp. 46-61, 1973.

[34] S. K. Baruah and J. R. Haritsa. "Scheduling for overload in real-time systems." *IEEE TOCS*, vol. 46, no. 9, pp. 1034-1039, 1997.

[35] K .S. Hong and J.T. Leung, On-line scheduling of real-time tasks. *IEEE TOCS*, vol. 41, no. 10, pp.1326-1331, 1992.

[36] A. K. Mok and D. Chen. "A multiframe model for real-time tasks." *IEEE TOSE*, vol. 23, no. 10, pp. 635-645, 1997.

[37] U. Verner, et al, Processing data streams with hard real-time constraints on heterogeneous systems, in *ICS*, pp. 120-129, 2011.

[38] K. Sajjapongse, et al. A preemption-based runtime to efficiently schedule multi-process applications on heterogeneous clusters with GPUs, in *HDPC*, pp. 179-190, 2013.

[39] H. Lee, et al. GPU-EvR: Run-time event based real-time scheduling framework on GPGPU platform, in *DATE*, pp. 1-6, 2014.

[40] S. Kato, et al. TimeGraph: GPU scheduling for real-time multi-tasking environments, in *USENIX ATC*, pp. 17-30. 2011.

[41] G. A. Elliott, et al. GPUSync: A framework for real-time GPU management, in *RTSS*, pp. 33-44, 2013.

[42] K. Cho, et al. On the properties of neural machine translation: Encoder-decoder approaches. In *SSST-8*, 2014.

[43] T. Tai Yeh, et al, Pagoda: Fine-grained GPU Resource Virtualization for Narrow Tasks, in *PPoPP*, pp 221-234, 2017.

[44] J. Hestness, et al. Beyond human-level accuracy: computational challenges in deep learning, in *PPoPP*, pp. 1-14, 2019.

[45] AMD's Asynchronous Shaders White Paper. https://developer.amd.com/wordpress/media/2012/10/Asynchronous-Shaders-White-Paper-FINAL.pdf

[46] A. Ka-Lau Mok. "Fundamental design problems of distributed systems for the hard-real-time environment." PhD diss., Massachusetts Institute of Technology, 1983.

[47] M-T Luong and C. D. Manning, Achieving Open Vocabulary Neural Machine Translation with Hybrid Word-Character Models, in *ACL*, pp. 1054-1063, 2016.

[48] S. Puthoor, et al. Oversubscribed command queues in GPUs, in *GPGPU-11*, pp. 50-60, 2018.

[49] NVIDIA, CUDA HyperQ Example. http://developer.download.nvidia.com/compute/DevZone/C/html_x64/6_Advanced/simpleHyperQ/doc/HyperQ.pdf, 2013

[50] J. D. C. Little, Introduction to Little's Law as Viewed on Its 50th Anniversary, Operations Research, vol. 59, no. 3, pp. 535-535, 2011.

[51] X. Zhang, et al. Towards Memory Friendly Long-Short Term Memory Networks (LSTMs) on Mobile GPUs, in *MICRO*, pp. 162-174, 2018.

[52] S. Puthoor, et al. Implementing directed acyclic graphs with the heterogeneous system architecture, in *GPGPU-9*, pp. 53-62, 2018.

[53] Q. Chen, et al. Prophet: Precise QoS prediction on non-preemptive accelerators to improve utilization in warehouse-scale computers, in *ASPLOS*, pp. 17-32, 2017.

[54] Q. Chen, et al. Baymax: QoS awareness and increased utilization for non-preemptive accelerators in warehouse scale computers, in *ASPLOS*, pp. 681-696, 2016.

[55] C. Holmes, et al. GRNN: Low-Latency and Scalable RNN Inference on GPUs, in *EuroSys*, pp. 1-16, 2019.

[56] B. Wu, et al. Flep: Enabling flexible and efficient preemption on GPUs, in *ASPLOS*, pp 483-496, 2017 .

[57] I. Tanasic, et al. Enabling preemptive multiprogramming on GPUs, in *ISCA*, pp. 193-204, 2014.

[58] J. Jong Kyu Park, et al. Chimera: Collaborative preemption for multitasking on a shared GPU, in *ASPLOS*, pp. 593-606, 2015.

[59] G. Chen, et al. Effisha: A software framework for enabling effficient preemptive scheduling of GPU, in *PPoPP*, pp 3-16, 2017.

[60] Z. Wang, et al. Quality of service support for fine-grained sharing on GPUs, In *ISCA*, pp. 269-281, 2017.

[61] A. Kalia, et al. Raising the bar for using GPUs in software packet processing, in *NSDI*, pp. 409-423. 2015.

[62] Y. Go, et al. APUNet: Revitalizing GPU as Packet Processing Accelerator, in *NSDI*, pp. 83-96, 2017.

[63] D. Zhou, et al. Scalable, high performance ethernet forwarding with cuckooswitch, in *CoNEXT*, pp. 97-108, 2013.

[64] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau. Operating Systems: Three easy pieces. Chapter 8, Multilevel Feedback Queue, 2018.

[65] Lucida Benchmark Suite: https://github.com/jhauswald/lucida, 2016.

[66] G-Opt Benchmark Suite: https://github.com/efficient/gopt, 2017.

[67] M. Kyu Jeong, et al. A QoS-aware memory controller for dynamically balancing GPU and CPU bandwidth use in an MPSoC, in *DAC*, pp. 850-855, 2012.

[68] H. Usui, et al. DASH: Deadline-aware high-performance memory scheduler for heterogeneous systems with hardware accelerators, in *TACO*, vol. 12, issue 4, no. 65, pp. 1-28, 2016.

[69] T. Zhao, et al. Serving Recurrent Neural Networks Efficiently with a Spatial Accelerator, in *SysML*, 2019.

[70] J. Hauswald, et al. Sirius: An Open End-to-End Voice and Vision Personal Assistant and Its Implications for Future Warehouse Scale Computers, in *ASPLOS*, pp. 1-32, 2015.

[71] S. Zhang, et al. Asynchronous stochastic gradient descent for DNN training, in *ICASPP*, pp. 6660-6663, 2013.

[72] R. Kaleem, et al. Stochastic gradient descent on GPUs, in *GPGPU-8*, pp. 81-89, 2015.

[73] C. Noel and S. Osindero. "Dogwild! - Distributed Hogwild for CPU & GPU." In *NeurIPS Workshop on Distributed Machine Learning and Matrix Computations*, pp. 693-701. 2014.

[74] D. Tsafrir, et al. Backfilling Using System-Generated Predictions Rather than User Runtime Estimates, in *IEEE TPDS, vol*. 18, no. 6, pp. 789-803, 2007.

[75] W. Smith, et al. Using Run-Time Predictions to Estimate Queue Wait Times and Improve Scheduler Performance, in *IPPS/SPDP/JSSPP*, pp. 202-219, 1999.

[76] AMD Polaris GPU Architecture White Paper, https://www.amd.com/system/files/documents/polaris-whitepaper.pdf, 2017.

[77] J. Alsop, et al. Optimizing GPU Cache Policies for MI Workloads, short paper at *IISWC*, November 2019.

[78] L. Ke, et al. RecNMP: Accelerating Personalized Recommendation with Near-Memory Processing. In *ISCA*, pp. 790-803, 2020.

[79] Y. Choi and M. Rhu, PREMA: A Predictive Multi-task Scheduling Algorithm For Preemptible Neural Processing Units, in *HPCA*, 2020.

[80] A. F. Cuervo and S. Chavis, "Site Reliability Engineering: How Google Runs Production Systems – Chapter 21: Handling Overload", https://landing.google.com/sre/sre-book/chapters/handling-overload/, (2016).

[81] W. J. Dally, Hardware for Deep Learning, keynote at *SysML*, February 2018.

[82] W. Zhang, et al. Laius: Towards latency awareness and improved utilization of spatial multitasking accelerators in datacenters, in *ICS*, pp. 58–68, 2019.

[83] P. Aguilera, et al. QoS-aware dynamic resource allocation for spatial-multitasking GPUs, in *ASP-DAC*, pp. 726-731, 2014

[84] Z. Wang, et al. Simultaneous Multikernel GPU: Multi-tasking throughput processors via fine-grained sharing, in *HPCA*, pp. 358-369, 2016.

[85] X. Long, et al. "Towards OS-level and Device-level Cooperative Scheduling for Multitasking GPUs," in *IEEE Access*.

[86] Y. Ukidave, et al. Mystic: Predictive Scheduling for GPU Based Cloud Servers Using Machine Learning, In *IPDPS*, pp. 353-362, 2016.

[87] V. Gupta, et al. Pegasus: Coordinated Scheduling for Virtualized Accelerator-based Systems. In *USENIX ATC*, 2011.

[88] S. Panneerselvam and M. M. Swift. Rinnegan: Efficient Resource Use. In *PACT*, pp. 373–386, 2016.

[89] K. Menychtas, et al. Disengaged Scheduling for Fair, Protected Access to Fast Computational Accelerators. In *ASPLOS*, pp. 301–316, 2014.

[90] C. J. Rossbach, et al. 2011. PTask: Operating System Abstractions to Manage GPUs as Compute Devices. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (*SOSP*). 233–248.

[91] N. Capodieci, et al. "Deadline-Based Scheduling for GPU with Preemption Support," *2018 IEEE Real-Time Systems Symposium (RTSS)*, 2018, pp. 119-130.

[92] K. T. Malladi, et al. "Towards energy-proportional datacenter memory with mobile DRAM." In *ISCA*, pp. 37-48, 2012.

[93] U. Ahmed, et al. "A load balance multi-scheduling model for OpenCL kernel tasks in an integrated cluster." In *Soft Computing* (2020).

[94] S. Choi, et al., Earliest Virtual Deadline Zero Laxity Scheduling for Improved Responsiveness of Mobile GPUs. *JSTS*, Vol.17, No.1, February, 2017.