TReM: A Task Revocation Mechanism for GPUs

Manos Pavlidakis[1,2], Stelios Mavridis[1,2], Nikos Chrysos[1], and Angelos Bilas[1,2]
[1]*Institute of Computer Science, Foundation for Research and Technology - Hellas, Greece*
[2]*Computer Science Department, University of Crete, Greece*

***Abstract*—GPUs in datacenters and cloud environments are mainly offered in a dedicated manner to applications, which leads to GPU under-utilization. Previous work has focused on increasing utilization by sharing GPUs across batch and user-facing tasks. With the presence of long-running tasks, scheduling approaches without GPU preemption fail to meet the SLA of user-facing tasks. Existing GPU preemption mechanisms introduce variable delays up to several seconds, which is intolerable, or require kernel source code, which is not always available.**

In this paper, we design TReM, a GPU revocation mechanism that stops a task at any point in its execution. TReM has a constant latency, of about 5ms to stop the currently executing kernel and about 17ms to start a new task. TReM does not store the state of the revoked kernel to obviate transfer latencies. We design and implement two scheduling policies, Priority and Elastic, that prioritize user-facing over batch tasks and utilize TReM to improve SLAs for user-facing tasks. To evaluate TReM, we use a workload generator that creates workloads with different characteristics, based on real traces. TReM reduces SLA violations by up to 10% compared to baseline policies that do not use a revocation mechanism. TReM incurs negligible overhead for non-revoked tasks and wastes only 3% of computation due to revocations for the workloads we examine.

***Index Terms*—GPGPUs, Scheduling, Multi-GPU Servers, Elasticity, Task Revocation, Task Preemption**

I. Introduction

Heterogeneous datacenters deploy accelerators, typically GPUs and FPGAs, to process more data within the same power budget. Although resources in cloud environments are shared, most cloud providers offer accelerators in a dedicated manner. This exclusive assignment improves the Quality of Service (QoS) at the cost of accelerator under-utilization because a single task is not always capable of fully utilizing the accelerator [1].

Recently, there has been work [2]–[5] towards increasing GPU utilization by sharing GPUs across user-facing and batch processing tasks [6], [7]. User-facing tasks require tail latency guarantees based on a Service Level Agreement (SLA) [2]. The execution time of user-facing tasks, e.g. inference in machine learning models, ranges from several microseconds to a few hundreds of milliseconds. Batch applications do not have strict response-time requirements per task [6].

Previous scheduling policies for GPUs [2]–[4] ensure that more than 95% of user-facing tasks meet their SLA (200ms) in the presence of batch tasks *with execution time shorter than SLA*. Under this assumption, one can afford to wait for such, short-batch tasks to finish before launching a new, user-facing task. However, batch tasks commonly have execution times orders of magnitude longer than the tail latency for user-facing tasks (i.e., ranging up to several minutes) [8], [9].

To guarantee that user-facing tasks will meet their SLA in the presence of long running batch tasks, previous work provides mechanisms that preempt the running batch task. However, previous work on GPU preemption [8], [10]–[12] does not provide bounded latency. Therefore, user-facing tasks still incur high tail latencies. Additionally, approaches as FLEP [10] and GPES [11] require the source code of the kernel, while Pascal Compute Preemption [8] works only with Pascal architecture NVIDIA GPUs.

A preemption mechanism [8], [10], [11] consists of three parts; (1) stop the currently executing task, (2) save its state, and (3) replay the task later. GPES [11], Sajjapongse et al. [12], and FLEP [10] rely on existing CUDA thread blocks or slice tasks to provide preemption points. This approach introduce variable latency: Their overhead depends on thread block size or sub-task granularity. Regarding the preempted task state, this can be saved in the GPU or the host memory. Saving the state in the GPU, as in Pascal's preemption or FLEP, can lead to memory monopolization for tasks with large memory footprint. On the other hand, transferring the task state to host memory introduces variable latency.

In this work, we design and implement TReM, a revocation mechanism that overcomes the problems of existing preemption approaches. TReM stops a task by aborting its currently executing kernel, without saving any state, and replays it later. The first challenge is to stop the executing kernel at any point of its execution, providing bounded latency. To achieve this, TReM stops the kernel from inside the GPU. In particular, we start the actual kernel from a wrapper kernel using CUDA dynamic parallelism [13]. After issuing the actual kernel, the wrapper kernel polls a revocation flag on CUDA unified memory and calls `asm(trap)` to abort the execution of the actual kernel when the host sets the revocation flag.

The second challenge is to eliminate the variable latency of moving task state from GPU to host. TReM avoids saving any state of the currently running task to host memory to reduce overhead and latency. Instead, TReM replays revoked tasks. To reduce the amount of replayed work in task granularity, TReM selects to revoke tasks with the latest start time.

We design and implement a runtime scheduler that prioritizes user-facing over batch tasks, instructs TReM when to revoke batch tasks, and is able to manage multiple GPUs in a single node. We design and develop two scheduling policies, Priority and Elastic. Priority tries to allocate a GPU for every user-facing task. As a result, Priority+TReM may revoke as many batch tasks as the number of newly arrived user-facing tasks. Elastic dynamically computes a *minimum* number of accelerators needed to sustain the SLA and devotes the remaining accelerators to batch tasks. Effectively, Elastic+TReM results in fewer revocations i.e., less work to be re-executed. If there is a need to limit the wasted work for long-running applications, e.g. executing for days, TReM can also be coupled with existing checkpointing mechanisms [14], [15], as discussed in Section VII.

Our evaluation shows that TReM revokes an executing kernel in 5ms, while the next kernel requires another 17ms to start (22ms in total). As we explain in Section VI-A, the 22ms depend on the CUDA runtime. TReM adds negligible overhead to non-revoked tasks and consumes minimal resources in modern GPUs, as discussed in Section VI-A. Our experimental results from a real testbed and realistic workloads show that using TReM allows us to meet the SLA for 98% of user-facing tasks in the presence of long-running batch tasks under 89% GPU utilization. Wasted work due to revocations is only 3% of the total user-facing and batch execution time.

Our main contributions in this work are the following:

1) We design TReM, a task revocation mechanism that i) exhibits constant and low overhead, independent of task size and memory footprint, ii) avoids kernel recompilation, iii) incurs zero overhead to non-revoked tasks, and iv) can be deployed to all NVIDIA state-of-the-art GPU architectures.
2) We evaluate TReM using two scheduling policies, Elastic and Priority, in a *real system* with four (4) GPUs and show the benefits of revoking batch tasks.
3) We use simulation to examine how our approach scales with an increasing number of GPUs and how it behaves under different revocation latencies.

The remainder of this paper is organized as follows. In Section II, we discuss previous work and the problem statement that motivate this work. In Section III we present our revocation mechanism, and in Section IV our scheduling policies that prioritize user-facing tasks. Then, Section V presents the evaluation testbed and workloads, and Section VI presents our results, including a detailed latency breakdown of TReM. We conclude with discussions in Section VII and summarize our findings in Section VIII.

II. Related work & Motivation

Sharing a GPU across applications introduces difficulties in providing latency guarantees to user-facing tasks. We categorize previous approaches in scheduling and preemption.

A. SLA-based scheduling

Timegraph [3], Baymax [2], gVirt [16], and VGRIS [4] implement sophisticated SLA-aware task scheduling policies. These approaches consider only batch tasks with execution time comparable to the SLA, in the order of tens or a few hundreds milliseconds. Thus, they alleviate the *priority inversion problem*, when a critical task waits for a batch task to finish, only with short batch tasks.

With modern workloads, long-running batch tasks are becoming more common, as the complexity of the algorithms and the amount of data they use increases. Long-running tasks can monopolize the GPU [8], hence requiring a GPU preemption or revocation mechanism. TReM is a task revocation mechanism that can be coupled with suitable scheduling policies. A scheduling policy will determine when a task has to be killed and instruct TReM to kill it. In this paper, we show that TReM coupled with two scheduling policies, Elastic and Priority, can ensure the SLA for user-facing tasks in the presence of long-running batch tasks.

B. State-saving preemption mechanisms

Operating systems can preempt a running process and give the CPU to another process within a few microseconds. Such low overhead preemption mechanisms are not available in modern GPUs.

Chimera [17] is an effective preemption approach that provides block context switching, draining, and flushing. However, Chimera is only implemented in a simulation environment and is not supported by existing GPUs. PEP [18] and GPU snapshot [14] add incremental checkpoints to decrease the overhead of saving the full context of a revoked or failed kernel. These approaches are orthogonal to ours. TReM can be integrated with Kyushick's [14] approach to bound the amount of wasted work to a single checkpoint interval at the cost of additional memory usage at the GPU.

Sajjapongse et al. [12] rely on existing synchronization points to preempt kernels. Their approach cannot provide low response time to user-facing tasks when synchronization points are rare. Moreover, they transfer both the state and data of the preempted kernel to host memory, resulting in long preemption latency for tasks with a large memory footprint. GPES [11] splits the kernel statically into multiple subkernels. Consequently, it avoids the problems implied from synchronization points. However, GPES has to determine the granularity of a slice. If the slices are too small, they introduce runtime overhead, whereas if they are too large, they can not provide low preemption latency. In contrast, TReM provides low revocation latency because it can revoke a kernel at any arbitrary point of its execution without saving any task state.

FLEP [10] uses `asm(exit)` to preempt kernels, as discussed in detail in Section III. To reduce the preemption latency, FLEP splits the initial kernel into multiple smaller kernels using persistent threads [19]. Thus it limits the number of launched thread blocks in each kernel to the maximum number of thread blocks that the GPU can simultaneously execute [19]. However, the preemption latency of FLEP depends on the execution time of thread blocks. Additionally, FLEP does not discuss how to free the GPU DRAM from preempted task data, thus potentially inducing memory monopolization. Finally, FLEP requires kernel source code to make it preemptible. TReM is based on CUDA dynamic parallelism [13], so it does not rely on kernel slicing and does not require kernel source code. Furthermore, it can kill a kernel at any point of its execution, with constant delay. TReM does not save the state of the killed kernel in GPU memory; hence, it does not prevent other high priority kernels from starting due to memory shortage.

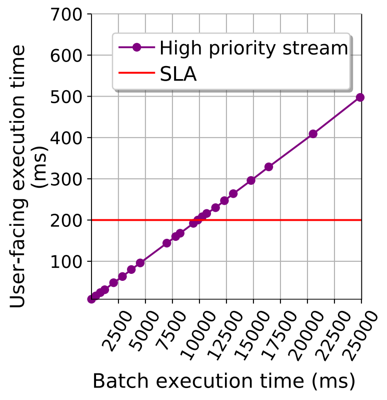


Fig. 1: Evaluating NVIDIA compute preemption. Increase in the execution time of a user-facing task (y-axis), assigned to a high priority stream, in the presence of a collocated batch task, assigned to a low priority stream, with varying execution time (x-axis).

NVIDIA Pascal GPUs, similarly to FLEP, provide a compute preemption mechanism that allows CUDA kernels to be interrupted at block-level granularity. According to CUDA [5] *the higher priority stream will preempt blocks already executing in the low priority stream*. We evaluate the compute preemption of Quadro P1000 (Pascal architecture), using streams with priorities [5]. In our experiment, we use a benchmark with two kernels, one assigned to a low priority stream (i.e. batch) and the other to a high priority stream (i.e. user-facing). The two kernels consist of 1024 thread-blocks with 512 threads per block to ensure that they will compete for stream multiprocessors (SMs). These two kernels are identical, iteratively copying one array to another. We first start the low priority kernel, and we measure the latency of the high priority kernel. Figure 1 shows that the execution time of the high-priority kernel *increases linearly* with the execution time of the low priority kernel, even though the high priority kernel size stays fixed. Therefore, the preemption latency is variable and depends on the low priority kernel's thread block execution time.

TABLE I: TReM and prior state-of-the-art approaches.

	FLEP [10]	GPES [11]	Pascal Preemption [8]	Chimera [17]	Baymax [2]	TReM + Elastic
Revocation/Preemption	+	+	+	+	-	+
Provides Low & Constant preemption latency	-	-	*Not Known*	+	-	+
Handles tasks with large memory footprint	-	-	+	+	+	+
Does not need kernel source code	-	-	-	-	-	+
Supports all NVIDIA GPUs (*CC≥3)	+	+	-	+	+	+
Provides SLA aware scheduling policy	-	-	-	-	+	+

**Compute Capability*

Table I compares our work with previous state-of-the-art approaches. To the best of our knowledge, Elastic+TReM is the only SLA-aware scheduling solution that deals with long-running batch tasks, using a revocation mechanism that: (i) provides low and constant latency; (ii) handles tasks with large memory footprint avoiding to store or transfer large amounts of state and merely restarting the task; (iii) does not require GPU kernel source code; and (iv) works with all NVIDIA GPUs (Compute Capability (C.C.) $\geq$ 3) that exist in today and future datacenters.

III. TReM Revocation Mechanism

TReM revokes the currently executing task by killing its kernel and replays it later using the information maintained at the host. A task consists of a set of kernels with their input and output data.

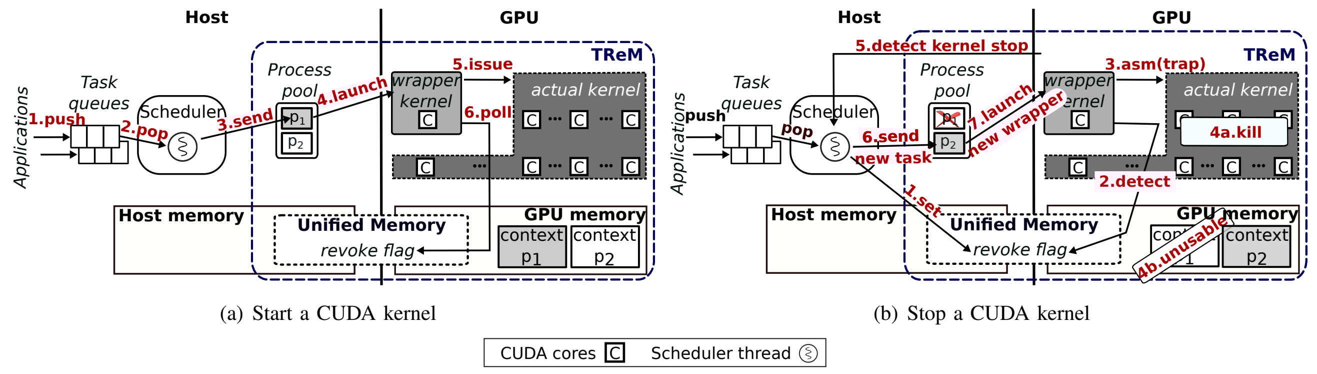
TABLE II: Latency of different methods to revoke/preempt a kernel running on a GPU.

Kernel dimensions	Total Threads	Latency (ms)		
		Process kill	asm(exit)	TReM
kernel <16, 16>	256	3000	130	22
kernel <32, 32>	1024	3000	195	22
kernel <64, 64>	4096	3000	600	22
kernel <128, 128>	16384	3000	1430	22

There are three ways to stop the execution of a running kernel; (1) Kill the issuing process on the host, (2) use `asm(exit)`, and (3) use `asm(trap)`. Each host process that performs a CUDA call is associated with a CUDA context. When killing the host process, option (1), the NVIDIA driver clears the killed kernel context and prepares the CUDA environment to be functional. We evaluate this approach while designing TReM, and we find that killing a process introduces a delay of up to 3s to the next kernel (Table II).

Option (2) is to use *asm(exit)*, which is normally called when a kernel uses `return`, terminating the execution of the current thread. We measure the performance of `asm(exit)` for a varying number of kernel dimensions, i.e., threads and blocks. Table II shows that when the number of threads increases, the latency of `asm(exit)` increases significantly. This is because the GPU must wait for all threads of the kernel to exit. Typical kernels launch thousands of thread blocks with hundreds of threads each [10], which can incur a delay of several seconds. For instance, Table II shows that for a kernel dimension of <128, 128> the latency of this mechanism is almost 1.5s. Consequently, `asm(exit)` cannot be used to provide low revocation latency. For Table II, we run each experiment 20 times with the same setup as in Section V-A. To measure the revocation/preemption latency, we start a timer when a kernel is issued and stop it when the next kernel (the kernel after the revoked/preempted kernel) starts its execution in the GPU.

TReM uses `asm(trap)`, option (3), to abort the kernel execution. Commonly, `asm(trap)` is called when a kernel-code assertion fails, and the CUDA context of the issuing process is unusable, thus it cannot be used for any subsequent CUDA calls. Calling `asm(trap)` results in immediate termination of the kernel, with constant latency as we discuss next.



(a) Start a CUDA kernel

(b) Stop a CUDA kernel

Fig. 2: TReM overview.

A. Revoking a kernel with TReM

Figure 2 shows an overview of TReM. Applications use *task queues* to issue tasks to the scheduler. This scheduler runs in a different process context from the applications and manages a GPU using a scheduler thread. In case that there are multiple GPUs in a node, the scheduler spawns multiple scheduler threads, as explained in Section IV. A scheduler thread is responsible for dequeuing tasks from a selected queue and issues task kernel to its GPU, based on a policy. Moreover, the scheduler thread checks the status of tasks (killed or finished) and reissues them if needed. A GPU is time-shared among applications, serving tasks from multiple task queues. Next, we describe TReM, our revocation mechanism, in detail.

ALGORITHM 1
Poll mechanism in the kernel *wrapper*.

```
function WRAPPER_KERNEL(,)
    cudaLaunchKernel(actual_kernel);
    while (1) do
        if (revoke_kernel == true) then
            asm(trap);
        end if
    end while
end function
```

As shown in Figure 2(a), TReM encapsulates each CUDA kernel in a CUDA wrapper kernel. The wrapper uses one thread and one thread block i.e., wrapper <1,1>. The wrapper then issues the actual kernel, using CUDA dynamic parallelism [13]. With CUDA dynamic parallelism, a parent grid (in our case, the wrapper) launches kernels called child grids (in our case, the actual kernel). Thus, TReM does not require kernel source code. Subsequently, the wrapper polls a revoke flag, which is set when the host runtime scheduler decides to kill a kernel running on the GPU. This revoke flag is allocated in *unified memory* since it has to be accessible from both the host and GPU. Using *cudaMemcpy* or *cudaMemcpyAsync* cannot fulfill our purpose because all CUDA calls in the same context are executed in issue order. Algorithm 1 presents the code of the wrapper kernel.

Figure 2(b) shows the procedure of stopping a kernel. When the revoke flag is set, the wrapper kernel executes `asm(trap)` to stop the running (i.e., actual) kernel. The NVIDIA driver detects that the `asm(trap)` command is called and marks the context of the issuing host process as unusable. As a result, this process can not execute any other CUDA calls. To launch a new kernel, we use a new process with a new context (p2 in Fig. 2(b)). The process with the unusable context will be removed later from the process pool. To detect that kernel execution has stopped, we check the return value of a CUDA call. If this value is false, the wrapper kernel has called the `asm(trap)` and has stopped its execution.

Figure 3 shows the timing of TReM (see Section VI-A). To revoke a running kernel, the scheduler sets the revoke flag, and the wrapper executes `asm(trap)`), which requires 5ms. However, the next task will start its execution with an additional delay of 17ms (in total 22ms) because of the first CUDA call. Due to `asm(trap)`, the CUDA context of the issuing host process becomes unusable, clearing the CUDA context introduces a delay of 60ms. In the case of user-facing tasks, it is important to avoid this extra delay. For this purpose, we defer clearing the unusable GPU context, as discussed below.

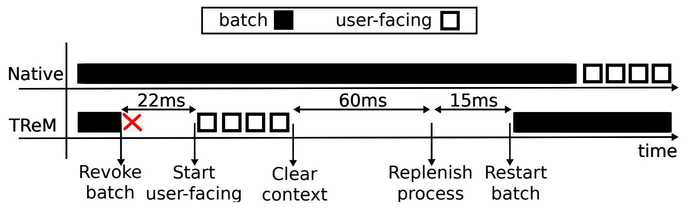
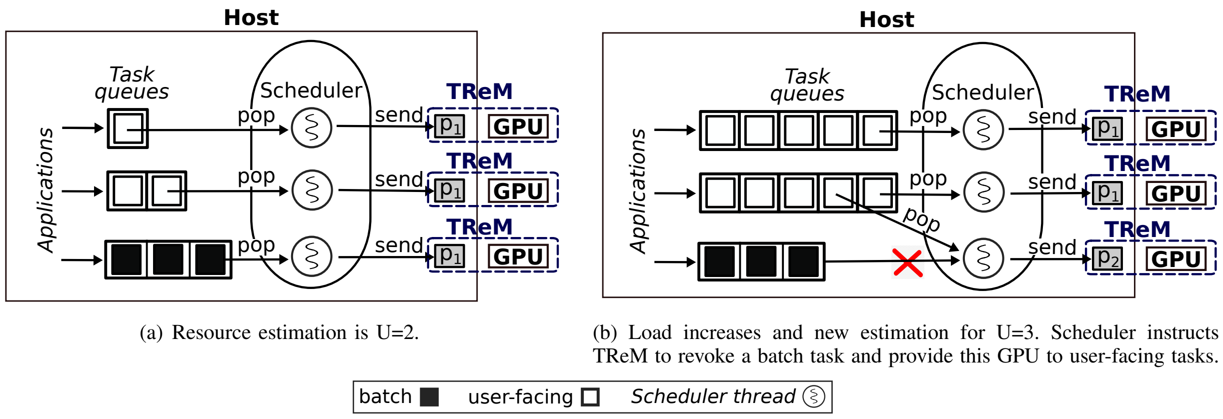


Fig. 3: The timing of TReM compared to native execution. Batch execution time is in the range of seconds.

Initiating the CUDA runtime from a process takes approximately 15ms. To avoid this latency, we use a *pool of processes* with pre-initialized CUDA environments. We use only the pool's head process to issue task kernel (p1 in Figure 2(a)). When the active process becomes unusable after an `asm(trap)`, it is removed from the pool, and the next process becomes the pool head (p2 in Figure 2(b)), which can



(a) Resource estimation is U=2.

(b) Load increases and new estimation for U=3. Scheduler instructs TReM to revoke a batch task and provide this GPU to user-facing tasks.

Fig. 4: TReM + Elastic in multi-GPU setups.

be used to execute kernels immediately.

After an `asm(trap)`, we have an additional unusable GPU context and one process less in the pool. We need to always have at least two processes in the pool to fulfill the case that a batch is executed and we decide to kill it and immediately serve user-facing tasks. For this purpose, we clear the unusable GPU contexts and replenish the process pool when we spawn a batch task. In this manner, the latency of these operations, 60ms + 15ms in our experiments, affects only a subsequent batch task.

TReM is capable of handling tasks that utilize all the GPU DRAM since it does not store any state of the task that is going to be killed, either in the Host or GPU memory. The GPU context of the killed kernel contains useless data and will be removed when we clear the GPU context. If the available GPU DRAM is not sufficient to hold the data of a new task, we immediately pay the penalty of 60ms to free all GPU DRAM.

IV. Reducing SLA violations of user-facing tasks

In multi-GPU setups, our scheduler spawns multiple threads, one for each GPU in the system, as Figure 4 shows. Multiple GPUs can serve independent tasks from the same queue (application) in parallel. The mapping of task queues to GPUs is controlled by the scheduler that prioritizes user-facing tasks to reduce SLA violations. TReM runs in each GPU, and the scheduler thread mapped to that particular GPU is responsible for sending tasks to the active process and set the revoke flag when a task has to be revoked.

Our scheduler prioritizes queues with user-facing tasks over queues with batch tasks. We implement two policies, Priority and Elastic, which differ in multi-GPU setups. In particular, Priority maps all GPUs to all task queues. For instance, when a burst of new user-facing tasks arrives, Priority will spread them to all available GPUs. On the other hand, Elastic exploits the capability to meet the SLA for user-facing tasks using a subset of the GPUs, as described in Section IV-A.

A. Elastic policy

Elastic prioritizes user-facing tasks over batch tasks, similar to Priority. However, it dynamically adjusts the number of GPUs allocated for user-facing tasks such that all outstanding user-facing tasks meet their SLA target. The remaining GPUs are used to serve batch tasks. We recalculate the number of GPUs for user-facing tasks every 100ms or when a task finishes.

We use the following procedure to estimate the number of GPUs needed to satisfy the current user-facing load. At time t, we first compute the maximum latency among all outstanding tasks in the system using Equation 1.

$$L^{max}(t) = l^{e}(t) \cdot q(t) \tag{1}$$

where $l^{e}(t)$ is our current estimate of the average execution time for user-facing tasks and $q(t)$ denote the number of outstanding user-facing tasks. To compute $l^{e}(t)$, we monitor previously executed user-facing tasks' execution time. Our algorithm then estimates the number of GPUs (U) needed to serve the currently outstanding user-facing tasks without violating their SLA according to the Equation 2.

$$U(t) = \left\lceil \frac{L^{max}(t)}{SLA} \right\rceil \tag{2}$$

In Figure 4(a), Elastic assigns two GPUs to user-facing tasks. According to its resource estimation, two GPUs (U=*2*) are sufficient to execute all outstanding user-facing tasks under the SLA. On the other hand, Priority will use all the GPUs to serve the three outstanding tasks and postpone the execution of the batch tasks.

B. Using TReM with Priority and Elastic

In Figure 4(b) the load increases, hence Elastic estimates that user-facing tasks require more resources (U=*3*) to meet their SLA. However, the third GPU executes a batch task. Without TReM, newly arriving user-facing tasks will wait for this batch task to finish its execution and miss their SLA. To overcome this priority inversion problem, we integrate Priority and Elastic with TReM to revoke the batch task executing in the third GPU and assign this GPU to user-facing tasks.

After the arrival of a burst of user-facing tasks, Priority will try to spread the new tasks on multiple GPUs. Being aware of the SLAs, Elastic time-multiplexes user-facing tasks on a reduced number of GPUs, as described in Section IV-A. Consequently, Elastic minimizes the number of revocations and, thus, the loss of useful work.

When we need to include more GPUs to serve user-facing tasks, both Priority and Elastic choose to revoke the batch task that *has started most recently*. Additionally, Elastic checks if the remaining time (predicted by the task execution time minus the task elapsed time) of the task that is going to be revoked is less than the task revocation latency. As a result, Elastic minimizes further the loss of useful work, compared to Priority. To avoid starvation, we use a revocation counter/threshold, as explained in the Discussion section.

V. Experimental Methodology

In this section, we describe the platform and the workloads that we use to evaluate TReM.

A. Multi-GPU server configuration and memory affinity

The server in our testbed consists of an Intel(R) Xeon(R) CPU E5-2630 v3 running at 2.40GHz (CentOS 7). The server is equipped with four (4) NVIDIA Quadro P1000 GPU cards (Pascal architecture), with 4GB of GDDR5 and 640 CUDA cores, each. We use CUDA 9.0 to implement TReM and NVSMI to measure GPU utilization. NVSMI utilization represents the time the GPU is busy and not the amount of GPU resources actually used.

Every P1000 GPU requires a PCIe gen3 x16 port. Our four GPU setup needs a total of 64 PCIe lanes. Our dual-socket motherboard provides 32 lanes for each socket, therefore we attach two GPUs in each socket.

In a multi-GPU configuration, there are significant memory *affinity issues*: the throughput of memory transfers depends on whether the path connecting the memory with the target GPU pass through the QPI bus between the two sockets. Using microbenchmarks, we find that the throughput of transfers to different GPUs can interfere with each other, degrading performance from 2x up to 4x. To eliminate this issue, we enforce each host thread (Fig. 4) to run in the same socket with its corresponding GPU.

B. Workloads

We evaluate our system using batch and user-facing tasks with execution times reported in Table III. The tasks used in our evaluation originate from the Rodinia3.2 benchmark suite [20] and from NVIDIA SDK of CUDA toolkit 9. The execution time is the interval between the time a task is issued until the issuer receives the result back. Thus it includes the transfers to and from the accelerator and the execution time of its kernel(s) on the GPU. The execution time of batch tasks ranges from tens of seconds up to two minutes, whereas the execution time of user-facing tasks ranges from 1 up to 170ms. We consider the SLA for user-facing tasks to be 200ms, as in Baymax [2]. The response time contains queuing delay implied from other outstanding tasks that exist in the system.

TABLE III: Average task execution time (ms).

Tasks	**Average task execution time (ms)**	**Memory footprint (MB)**	**Task type**
Particle Filter	1	12	user-facing
Euclid	8	24	user-facing
NW	38	44	user-facing
BFS	50	48	user-facing
Black&Scholes	60	112	user-facing
Pathfinder	68	74	user-facing
Hot Spot 3D	81	32	user-facing
Monte Carlo	150	68	user-facing
Darkgray	170	200	user-facing
Lava MD	46000	1069	batch
Hot Spot	130696	423	batch
Gaussian	311000	1120	batch

Each user-facing task can have more than one CUDA kernels, as in machine learning inference stages, with the corresponding input and output data. On the other hand, every batch task consists of a single large kernel because this stresses the scheduler more.

In our evaluation we use the following workloads:

- Micro-benchmarks, with a few tasks, to measure the responsiveness and the overheads of our mechanisms.
- A datacenter-inspired synthetic workload, with thousands of tasks, mixing user-facing with batch jobs.

To generate the datacenter-inspired workload mix, we implement a workload generator that mimics traces from Google [21] and Alibaba [22]. Our workload generator takes three parameters; (a) the job duration, (b) the job inter-arrival time, and (c) the user-facing to batch job ratio.

After analyzing Google and Alibaba traces, we found that job duration follows a Pareto distribution. Consequently, to generate a job, we choose the job duration first using a Pareto distribution, with different mean values for batch and user-facing jobs, as presented in Table IV. The mean values for user-facing and batch job duration is again extracted from the traces above. Each application/job consists of identical tasks, taken randomly from Table III.

We use two different values for the ratio between user-facing and batch jobs, 50:50 (according to Alibaba) and 80:20 (according to Google). Job inter-arrival time follows exponential distribution, with a base mean value selected to utilize all four GPUs fully. To emulate different loads, we use a scaling factor on the base inter-arrival mean value ranging from 0.25 to 2.0. Effectively, the scaling factor modifies the density of job arrivals, and as a result, the load and the burstiness of the workload. As a result, we can generate from *low* load, with job inter-arrival 2 (named load 0.25), to *over-subscription*, with job inter-arrival 0.25 (named load 2).

We limit the number of outstanding tasks coming from the same job to eight tasks [2]. We have selected this value empirically to ensure that the majority of tasks in a user-facing job will meet their SLA *if run alone*. There will be multiple user-facing jobs present concurrently at runtime, which increases the system load, as described above. The response time that we report and compare with the SLA counts only for outstanding tasks. Table IV summarizes the workloads

TABLE IV: Workload configurations.

Workload specification	**W1**	**W2**
User-facing to batch ratio	50:50	80:20
Experiment duration	1.5h	1.1h
User-facing job duration (mean)	5s	5s
Batch job duration (mean)	600s	600s
Total number of jobs	30	30
Total number of tasks	1560	1520
Number of user-facing tasks	925	1191
Number of batch tasks	635	329

used for our evaluation. We repeat each experiment five times using different random seeds for each distribution.

VI. Experimental evaluation

In this section, we first present the overheads of TReM and then we evaluate the effectiveness of TReM to improve the QoS of user-facing tasks in the presence of long-running batch tasks.

A. Overhead of TReM revocation

In this section, we first present the overheads of TReM and then evaluate the effectiveness of TReM to improve the QoS of user-facing tasks in the presence of long-running batch tasks.

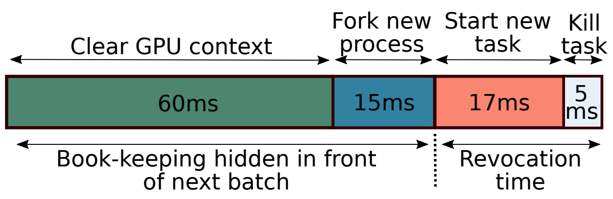


Fig. 5: TReM overhead breakdown.

1) Duration breakdown of TReM: As shown in Figure 3, there are multiple operations involved when deciding to kill a task. Two of them are in the critical path, affecting the latency of waiting for user-facing tasks: kill a task and start a new task using a process from the process pool. Figure 5 shows that these two operations require 5ms (kill) and 17ms (new task), for a total of 22ms. By breaking down the "start new task" operation, we find that 17ms are spent in the first CUDA call, even though we have pre-initialized the CUDA runtime. Note that during normal conditions, i.e., not after a kill operation, this CUDA call typically takes less than 2ms.

Killing the currently executing kernel requires 5ms. In particular, the revoke flag is allocated in the unified memory and is set by the host process and read by the GPU. When either side accesses a page that is not resident to its memory, a page fault occurs. The memory system holding the requested page will unmap it from its page table, and the page will be migrated to the faulting process. In our measurements, the process above has almost 1.2 millisecond latency. The remaining time (i.e., 3.8ms) is due to the CUDA call that we use to detect that the wrapper kernel has stopped (i.e., called `asm(trap)`) by checking its return value. As a result, the 22ms revocation overhead is dominated by the CUDA runtime.

TReM needs to clear the GPU context, which costs another 60ms, and replenish the process pool, which requires 15ms. These 15ms are spent creating a new process and executing a CUDA call to warm up the CUDA runtime, creating a context in the GPU for the new process. However, as discussed already, TReM hides the latency of these last two operations by invoking them before the start of a new batch task.

2) TReM does not incur overhead for non revoked tasks: TReM does not add any new code to executing kernels, thus it does not introduce overhead during task execution. To verify this, we run each task in Table III one thousand times with and without TReM. The results are virtually the same with less than 100μs discrepancies, which is at most 0.01% of the execution time.

3) TReM consumes less than 1% of cores on state-of-the-art GPUs: The wrapper kernel used by TReM to execute the actual kernel is spawned with one thread block using one thread. The NVIDIA runtime starts the wrapper kernel in a warp (32 CUDA cores). In our evaluation, Pascal P1000 has 640 CUDA cores in total, hence TReM requires 5% of NVIDIA P1000's CUDA cores. However, the percentage of resources consumed by TReM decreases to 0.625% with more recent GPUs, such as Volta that provide thousands of CUDA cores i.e., V100 has 5120 CUDA cores.

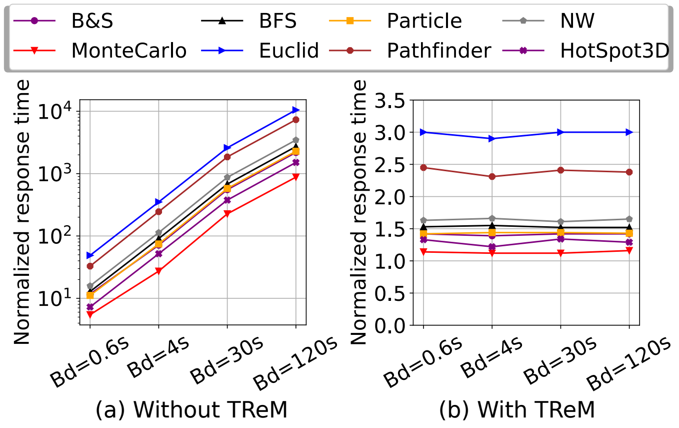


Fig. 6: Normalized response time of user-facing tasks over their stand-alone execution in the presence of batch tasks with different duration (Bd).

4) Resolving priority inversion: Figure 6 depicts the normalized response time of different user-facing tasks over their stand-alone execution when they time-share a GPU with long-running batch tasks. In this experiment, we first start a batch task, whose duration varies along the x-axis, and we record the latency of a subsequent user-facing task.

Without TReM, the response time of user-facing tasks increases linearly with batch task execution time (Figure 6(a)). In particular, it increases to orders of magnitude higher than task execution time itself. On the contrary, TReM results in constant response time for all user-facing tasks, independent of the batch task duration (Figure 6(b)). Their execution time is at most 3x the execution time of the standalone user-facing task in our experiments.

B. Effectiveness of TReM with long-running batch tasks

1) SLA violations: We use workloads W1 and W2 to examine how TReM reduces SLA violations. Figure 7 shows the percentage of user-facing tasks that meet their SLA (200ms) with increasing load. As mentioned, W1 and W2 differ in the ratio of user-facing to batch jobs.

The x-axis is incoming load, from low (0.25) to high (1.0) and oversubscribed (2.0). A load of 0.25 suffices to fully utilize one GPU (i.e., job inter-arrival 2). A load of 1.0 can fully utilize all four GPUs (i.e., job inter-arrival 1). A load of 2.0 over-subscribes our system by 2x (i.e. job inter-arrival 0.25). As we see, at low load, more than 99% of the tasks meet their SLA, irrespectively of the policy (Priority, Elastic), and the workload (W1, W2).

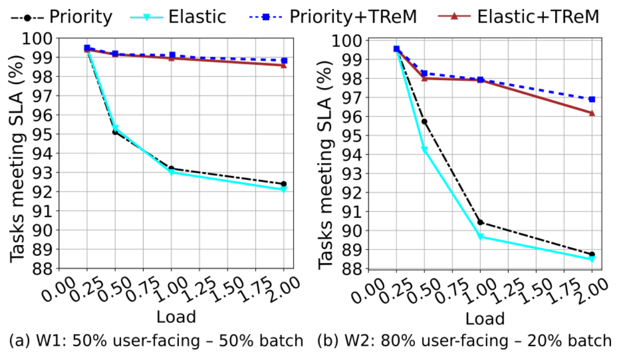


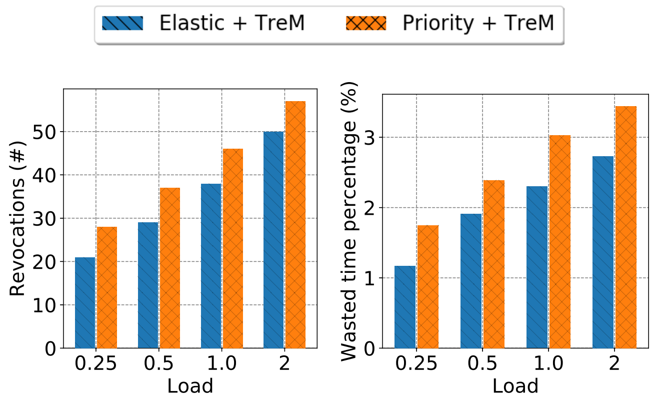
Fig. 7: Percentage of tasks that meet their SLA (y-axis) at increasing GPU load (x-axis), for workloads W1 (left) and W2 (right).

At a higher load, we see that W1 (Figure7(a)), with 50:50 ratio of user-facing to batch jobs, incurs more violations and the efficiency of Priority and Elastic drops to 93% at load 1.0 and 92% at 2.0. On the other hand, using TReM, both policies can tolerate the load increase with a much lower impact on efficiency, meeting the SLA for 99% of tasks at load 1.0 and 98% at load 2.0.

If we increase the number of user-facing tasks, using a ratio of 80:20 in W2 instead of 50:50 in W1, the advantages of TReM are more pronounced (Figure 7(b)). Without TReM, Priority and Elastic meet the SLA for 90% of the tasks at load 1.0, whereas, using TReM, we achieve 98% at load 1.0 and 96% at load 2.0. Therefore, *fast revocations is an effective ingredient to maintain the SLA in the presence of long-running batch tasks*.

At load 1.0, both Priority+TReM and Elastic+TReM achieve 89% GPU utilization. Priority and Elastic achieve 91% GPU utilization but with much more SLA violations.

Comparing Priority with Elastic, we can observe that Elastic leads to less than 0.6% more SLA violations on average compared to Priority. This is expected because Priority utilizes all GPUs to handle user-facing load, which results in significantly more revocations, as we discuss next. On the other hand, Elastic aims to decrease the number of revocations without increasing SLA violations.



(a) Number of revocations (b) Wasted compute time

Fig. 8: Revocations overhead: (a) Number of task revocations; (b) Wasted compute time due to revocations.

2) Lost work due to revocations: Figure 8(a) depicts the overhead of task revocations using TReM. We see that Elastic+TReM performs fewer revocations compared to Priority+TReM. Elastic uses a minimum number of GPUs to satisfy the SLA, packing when possible multiple user-facing task on the same GPU. Effectively, it triggers considerably fewer revocations. Under low load i.e., 0.25, Priority performs 33% more revocations than Elastic, 27% at load 0.5, and 14% at load 2.0. As the load increases, Elastic requires the same number of GPUs as Priority, thus the difference in revocations diminishes.

Figure 8(b) shows the percentage of lost work due to revocations. We measure total lost work *as the elapsed time between the start and kill for each task*. The percentage of lost work is computed as the ratio of the total work discarded over the total useful computation time in the workload. At low load (0.25), the wasted time percentage for both Elastic and Priority is below 2%, while at load 2.0 it reaches 3%. Both policies minimize wasted work by preferring to revoke the most recently started batch tasks. As discussed previously, Elastic outperforms Priority because it packs user-facing tasks in the same GPU and also because it will revoke a batch task only if its remaining time is less than the revocation overhead. Priority does not have the notion of SLA; it just prioritizes user-facing over batch tasks, hence it cannot measure the task remaining time.

3) Batch job duration percentiles: To examine in more detail the effect of our policies on batch jobs, Figure 9 depicts time to completion for batch jobs. As expected, time to completion of batch jobs increases with TReM because tasks are revoked and replayed. We should note, however, that without TReM, these batch jobs would typically wait for all user-facing tasks to complete or would need to execute on additional GPUs. Elastic reduces the impact on completion time from 4% up to 50% compared to Priority. In particular, for 50% of the jobs, Priority+TReM has 1.6x higher time to completion than Priority, whereas Elastic+TReM exhibits only 1.3x increase compared to Elastic. The effect of TReM becomes more pronounced on higher percentiles of batch jobs, and reaches 3.2x for Priority and 2.1x for Elastic.

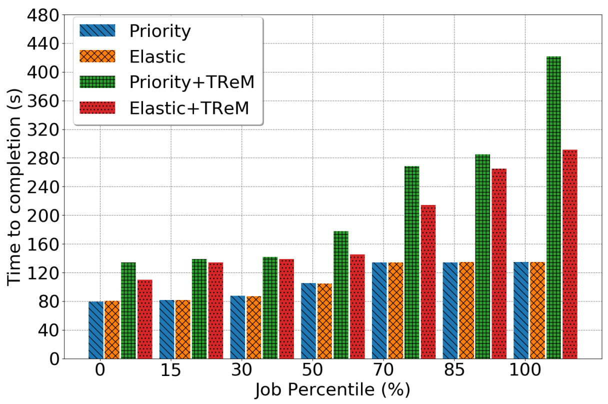


Fig. 9: Time to completion for batch jobs under different scheduling policies, for load 2.0.

4) Dynamic GPU partitioning with Elastic: Figure 10 shows how Elastic partitions the number of GPUs between user-facing and batch tasks over time. The upper part of the figure shows the number of GPUs allocated to user-facing tasks and the number of GPUs used to run batch tasks. The lower part of the figure depicts the actual latency of user-facing tasks over time. When user-facing tasks arrive (red crosses in the lower part), Elastic allocates more GPUs to user-facing tasks to avoid SLA violations. We see that Elastic estimates accurately the GPU requirements of user-facing tasks.

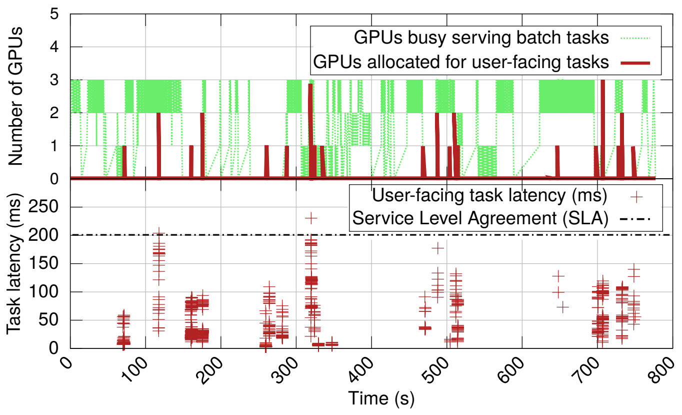


Fig. 10: Dynamic GPU allocation in Elastic and impact on SLA violations.

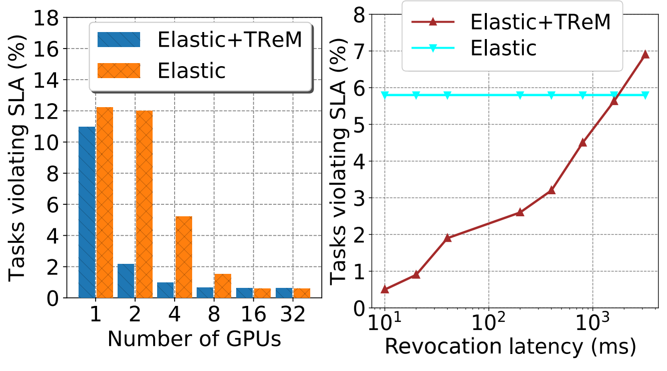
C. Scalability of TReM

To evaluate our system with more than four GPUs and different revocation latencies, we implement a simulator. Our simulator models our policies (Priority, Elastic) and TReM, without modelling the GPU internals. It takes as parameters (1) the task execution time, reported in Table III, (2) the workloads, described in Table IV, and (3) the revocation latency. The simulator runs W1 and W2 with the timings provided from Table III.

Simulation results are a superset of our testbed results in terms of; (a) number of GPUs and (b) revocation latency. Moreover, the simulator and the testbed results are very close. In particular, the violations of our simulator results for four GPUs and load 1.0 are 1% (Figure 11), while for the testbed the violations are 1.3% (Figure 7). Consequently, the difference between the simulator and the testbed results is 0.3%, when trends between policies (in Figure 7 with and without TReM) are in the order of 10%.

In Figure 11, the simulator runs the datacenter workloads, W1 and W2, for load 1.0. and reports their average results. Figure 11(a) shows task violations for Elastic and Elastic+TReM with load 1.0 and *a varying number of GPUs*. The positive effects of TReM are more pronounced with 2-8 GPUs. We see that using TReM we achieve less than 1% violations with 4 GPUs, whereas *without TReM we need 16 (4x) GPUs to achieve the same target.*

Figure 11(b) evaluates the percentage of violations with varying revocation latency, between 10 and 1000ms. The percentage of violations increases proportionally with revocation latency. Consequently, other mechanisms such as `asm(exit)` and process kill (Section III) that introduce more latency can meet SLA for only 94% of user-facing tasks. To ensure SLA for more than 99% of user-facing tasks, we require a revocation mechanism with 10ms latency.



(a) Varying number of GPUs (b) Varying revocation latency

Fig. 11: SLA violations for W1 and W2 under load 1.0; (a) varying the number of GPUs (revocation latency 22ms); (b) varying the revocation latency (4 GPUs).

VII. Discussion

Bounding the amount of wasted work: Figure 8(b) shows that TReM wastes only 3% of the total work due to revocations. However, it is possible that TReM kills a batch job (i.e., training in Machine Learning) after minutes or even hours of computation. In this case, the percentage of lost work can increase significantly. By incorporating a checkpoint mechanism, as GPU Snapshot [14] or Gandiva [15], to TReM, we can bound the wasted work at a single checkpoint interval. Both checkpoint mechanisms [14], [15] do not require changes in the kernel code, hence they can be integrated with TReM. Additionally, the checkpointing overhead for both approaches is less than 100ms, hence the increase in SLA violations will be minimal.

Repeated revocations & starvation: TReM revokes the batch task that has performed the least amount of work so far. If, however, user-facing tasks arrive periodically (e.g., a user-facing arrives every time a batch task has just started), TReM may starve a batch task by killing it repeatedly. To avoid this, TReM can maintain a revocation counter and inform the cluster scheduler to reassign the load across servers when the revocation counter exceeds a certain value.

CUDA streams: TReM uses `asm(trap)` to revoke a kernel. Effectively, TReM destroys the CUDA context of the issuing process. With CUDA streams, all concurrently running kernels belong to a single CUDA context. In case that user-facing kernels run concurrently with batch, user-facing kernels will be revoked as well. TReM+Elastic addresses this by allowing only kernels of the same type, i.e., only batch or user-facing, to run concurrently in a GPU.

VIII. Conclusions

In this paper, we design and implement TReM, a mechanism that revokes batch tasks running in a GPU and starts the next task within 22ms. TReM, in contrast to previous approaches, can revoke a task at any point of its execution using CUDA dynamic parallelism. TReM does not require kernel source code and is supported by almost all NVIDIA GPUs, except the outdated Fermi architecture that does not support dynamic parallelism. We implement two scheduling policies, Priority and Elastic, that aim to meet SLA for user-facing tasks when sharing a GPU with long running batch tasks. We then use TReM to enhance both policies and reduce SLA violations.

We evaluate TReM with two workloads derived from real datacenter traces. We show that Priority+TReM and Elastic+TReM ensure SLAs for 98% of user-facing tasks when collocated with long-running batch tasks, at high load with 89% GPU utilization. On the contrary, scheduling policies Priority and Elastic (without TReM) ensure SLA for 88.8% of tasks under high load. Additionally, Elastic+TReM reduces the number of revocations and the corresponding loss of useful work compared to Priority+TReM to only 3% of the total workload.

IX. Acknowledgments

We thankfully acknowledge the support of the European Commission under the Horizon 2020 Framework Program for Research and Innovation through the EVOLVE (H2020-ICT-825061) project. Furthermore, we thank Yannis Sfakianakis and Christi Symeonidou who provided insight and expertise to generate datacenter-inspired workloads.

References

[1] S. Pai, M. J. Thazhuthaveetil, and R. Govindarajan, "Improving gpgpu concurrency with elastic kernels," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '13, 2013.

[2] Q. Chen, H. Yang, J. Mars, and L. Tang, "Baymax: Qos awareness and increased utilization for non-preemptive accelerators in warehouse scale computers," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '16, 2016.

[3] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa, "Timegraph: Gpu scheduling for real-time multi-tasking environments," in *Proceedings of the 2011 USENIX Annual Technical Conference*, ser. USENIX ATC'11, 2011.

[4] M. Yu, C. Zhang, Z. Qi, J. Yao, Y. Wang, and H. Guan, "Vgris: virtualized gpu resource isolation and scheduling in cloud gaming," in *Proceedings of 2013 High-Performance Parallel and Distributed Computing*, ser. HPDC '13, 2013.

[5] J. Luitjens, "Cuda streams: Best practices and common pitfalls," in *GPU Techonology Conference*, 2015.

[6] P. Garefalakis, K. Karanasos, and P. Pietzuch, "Neptune: Scheduling suspendable tasks for unified stream/batch applications," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '19, 2019.

[7] L. A. Barroso, J. Clidaras, and U. Hölzle, "The datacenter as a computer: An introduction to the design of warehouse-scale machines, second edition," *Synthesis Lectures on Computer Architecture*, 2013.

[8] NVIDIA, "Whitepaper pascal compute preemption," https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf, 2016, online; accessed 25 July 2016.

[9] S. J. Krieder, J. M. Wozniak, T. Armstrong, M. Wilde, D. S. Katz, B. Grimmer, I. T. Foster, and I. Raicu, "Design and evaluation of the gemtc framework for gpu-enabled many-task computing," in *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, ser. HPDC '14, 2014.

[10] B. Wu, X. Liu, X. Zhou, and C. Jiang, "Flep: Enabling flexible and efficient preemption on gpus," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '17, 2017.

[11] H. Zhou, G. Tong, and C. Liu, "Gpes: A preemptive execution system for gpgpu computing," in *Proceedings of 21st IEEE Real-Time & Embedded Technology and Applications Symposium*, ser. RTAS '15, 2015.

[12] K. Sajjapongse, X. Wang, and M. Becchi, "A preemption-based runtime to efficiently schedule multi-process applications on heterogeneous clusters with gpus," in *Proceedings of the 22nd International Symposium on High-Performance Parallel & Distributed Computing*, ser. HPDC '13, 2013.

[13] S. Jones, "Introduction to dynamic parallelism," in *GPU Technology Conference Presentation S*, vol. 338, 2012, p. 2012.

[14] K. Lee, M. B. Sullivan, S. K. S. Hari, T. Tsai, S. W. Keckler, and M. Erez, "Gpu snapshot: Checkpoint offloading for gpu-dense systems," in *Proceedings of the ACM International Conference on Supercomputing*, ser. ICS '19, 2019.

[15] S. Chaudhary, R. Ramjee, M. Sivathanu, N. Kwatra, and S. Viswanatha, "Balancing efficiency and fairness in heterogeneous gpu clusters for deep learning," in *Proceedings of the Fifteenth European Conference on Computer Systems*, ser. EuroSys '20, 2020.

[16] K. Tian, Y. Dong, and D. Cowperthwaite, "A full gpu virtualization solution with mediated pass-through," in *Proceedings of the USENIX Annual Technical Conference*, ser. USENIX ATC'14, 2014.

[17] J. J. K. Park, Y. Park, and S. Mahlke, "Chimera: Collaborative preemption for multitasking on a shared gpu," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '15, 2015.

[18] C. Li, A. Zigerelli, J. Yang, and Y. Guo, "Pep: proactive checkpointing for efficient preemption on gpus," in *Proceedings of the 55th Annual Design Automation Conference*, ser. DAC '18, 2018.

[19] K. Gupta, J. A. Stuart, and J. D. Owens, "A study of persistent threads style gpu programming for gpgpu workloads," in *2012 Innovative Parallel Computing*, ser. InPar '12, 2012.

[20] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Proceedings of the 2009 IEEE International Symposium on Workload Characterization*, ser. IISWC '09, 2009.

[21] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch, "Heterogeneity and dynamicity of clouds at scale: Google trace analysis," in *Proceedings of the Third ACM Symposium on Cloud Computing*, ser. SoCC '12, 2012.

[22] C. Lu, K. Ye, G. Xu, C. Z. Xu, and T. Bai, "Imbalance in the cloud: An analysis on alibaba cluster trace," in *Proceedings of 2017 IEEE International Conference on Big Data*, ser. IEEE Big Data '17, 2017.