A Dynamic and Proactive GPU Preemption Mechanism Using Checkpointing

Chen Li, Andrew Zigerelli, Jun Yang, *Member, IEEE*, Youtao Zhang, *Member, IEEE*, Sheng Ma, and Yang Guo

***Abstract*—The demand for multitasking GPUs increases whenever the GPU may be shared by multiple applications, either spatially or temporally. This requires that GPUs can be preempted and switch context to a new application while already executing one. Unlike CPUs, context switching in GPUs is prohibitively expensive due to the large context states to swap out. There have been a number of efforts on reducing the overhead of preemption, through reducing the context sizes or overlapping context switching with execution. All those techniques are reactive approaches, meaning that context switching occurs when the preemption request arrives. In this paper, we propose a dynamic and proactive mechanism to reduce the latency of preemption. We observe that kernel execution is almost always preceded by known commands in both CUDA and OpenCL implementations. Hence, a preemption can be anticipated before the actual request arrives. We study such lead time and develop a prediction scheme to perform an early state saving. When the actual preemption is invoked, an incremental update relative to the previous saved state is performed, much like the conventional checkpointing mechanism. Our design can also choose to drain or checkpointing dynamically and accurately according to the feature of kernels in the runtime. This design effectively reduces the stall time of the preempting kernel due to context switching by 58.6%. Moreover, through careful handling of the saved state, we can also reduce the overall size of saved state by an average of 23.3%, compared with a full context switching.**

***Index Terms*—Checkpointing, context switch, GPU, preemption, runtime selection.**

I. Introduction

DUE TO their massive parallel processing capability, GPUs are now seen in various domains, such as high-performance computing, machine learning, and scientific computing [1]–[6]. These types of computing are now often provided as services in data centers or clouds, so that GPUs can be provided as shared infrastructure to users. Multitasking has become essential for GPUs to support concurrent services and requests. Some preliminary hardware features for multitasking are already in-place, such as the Hyper-Q provided by Nvidia's Kepler architecture [7], and the command processor supported by AMD [8]–[10]. While this was a step in the right direction, much is still needed to be done for true multitasking support [11], [12].

Context switching, a technique used in CPUs to support concurrency [13], has been proposed for GPUs for multitasking as well [11], [14]. CPU processes are relatively lightweight, allowing for a fast context switch and efficient time-multiplexing of tasks. However, a CUDA context is massive compared to the CPU [15]. For example, on the GTX980 GPU [16], the context size can be as large as 256KB for registers and 96KB shared memory per streaming multiprocessor (SM); the total context size can be 5664KB for the whole GPU (with 16 SMs). Saving such large context takes significant memory bandwidth and severely degrades performance [17], [18].

There have been several attempts to reduce the overhead of context switching for GPUs. The earliest technique lets context switching occur on a subset of SMs so the remaining SMs can continue execution [19]. The switching SMs are completely stalled to perform just context swapping, and the burden on memory bandwidth remains high. Later, a partial context switching technique allows thread blocks (TBs) in an SM to continue execution while swapping a particular TB [20], which maximally overlaps memory accesses, due to context switching, and kernel execution. This technique was further enhanced to allow a mix of draining (execute to completion), flushing (drop execution if idempotent), and switching TBs within each SM (depending on the deadline of the preemption) [21]. In parallel with those efforts, a lightweight context switch scheme was designed for reducing the amount of context that has to be saved off-chip [18]. All those prior works perform preemption via a reactive approach, meaning that all operations are activated upon the arrival of the preemption request. As a result, the preemption latency remains a threat to performance if the preempted kernel is not flushed.

Manuscript received June 10, 2018; revised September 8, 2018; accepted November 6, 2018. Date of publication November 29, 2018; date of current version December 23, 2019. This work was supported in part by the National Science Foundation under Grant CCF-1422331, Grant CCF-1617071, Grant CCF-1718080, and Grant CCF-1725657, in part by the National Natural Science Foundation of China under Grant 61832018, and Grant 61762526, in part by the Research Project of NUDT under Grant ZK17-03-06, and in part by the Science and Technology Innovation Project of Hunan Province under Grant 2018RS3083. The work of C. Li was supported by the Chinese Scholarship Council. A preliminary version of this paper was presented at the 55th Annual Design Automation Conference (DAC'18) [48]. This paper was recommended by Associate Editor S. Parameswaran. *(Corresponding author: Chen Li.)*

C. Li, S. Ma, and Y. Guo are with the College of Computer, National University of Defense Technology, Changsha 410073, China (e-mail: lichen@nudt.edu.cn; masheng@nudt.edu.cn; guoyang@nudt.edu.cn).

A. Zigerelli and J. Yang are with the Department of Electrical and Computer Engineering, University of Pittsburgh, Pittsburgh, PA 15261 USA (e-mail: anz37@pitt.edu; juy9@pitt.edu).

Y. Zhang is with the Computer Science Department, University of Pittsburgh, Pittsburgh, PA 15261 USA (e-mail: zhangyt@cs.pitt.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCAD.2018.2883906

In this paper, we propose a dynamic and proactive preemption mechanism, PEP, to reduce preemption latency and overhead. Through observing the kernel launch process, we find that the actual execution of a kernel on a GPU is always preceded by the kernel launch action, and the time from when a kernel is launched from CPU to the time the kernel starts to execute on the GPU is in the order of tens of microseconds. Leveraging such lead time and known operation pattern, we can anticipate the arrival of a preemption request and proactively prepare for context switching. When the preempting kernel arrives, the remaining work for completing the context switching is minimized. Hence, the effective preemption time is short. The preparation we perform for context switching utilizes the concept of checkpointing [22], [23]. The first base checkpoint is performed when a preemption is predicted to occur. Then an incremental checkpoint is performed when the preempting kernel arrives at the GPU. Saving the incremental checkpoint takes much less time than saving the full-size context of a preempted kernel, reducing the effective wait time of the preempting kernel. On average, the total amount of state saved is no more than the full context. We also observe that the context allocated is not completely active during the TB's lifetime. Therefore, we set dirty bits for registers to indicate whether the register is active or not. Only active registers must be saved, thus reducing the overall size of saved context significantly. Moreover, we design a dynamic runtime selection algorithm for preemption decisions. Short kernels can be preempted by draining, while long kernels can be preempted by checkpointing (context switching). This algorithm can achieve both low latency and small overhead.

Our contributions can be summarized as follows.

1) We study the kernel launch process, and observe that the event of preemption can be predicted.
2) We introduce a proactive preemption mechanism to reduce the stall time for the preempting kernel due to context switching. With proactive checkpointing, when preemption finally occurs, only a small subset of dirty context must be saved.
3) We use a simple dirty data-saving technique to reduce context size, which reduces the unnecessary context saving.
4) We develop a more precise estimation on TB draining time and context switch time, and design a dynamic runtime selection algorithm for preemption decisions. We can preempt both short and long kernels with low latency and small overhead.

We evaluate PEP and compare with previous best-effort preemption work Chimera [21] on several types of benchmarks [24]–[27]. Our experimental results show that we can reduce the average preemption latency from 8.9 to 3.6 μs, compared with previous best-effort preemption work, Chimera [21]. We also reduce the total state that needs to be saved by 16.1% compared to saving the full context size, using only simple context size reduction techniques. The total overhead, average switch time per TB, of PEP is 6.3% lower than Chimera.

```
1  __global__ void axa(double a, double *x){
2    int i = blockIdx.x*blockDim.x+threadIdx.x;
3    x[i] = a*x[i] + a;
4  }
5
6  void main(){
7    int N = 1048576;
8    double *x, *d_x;
9    x = (double*)malloc(N*sizeof(double));
10   for (int i = 0; i < N; i++) {
11     x[i] = 3.0;
12   }
13   cudaMalloc(&d_x, N*sizeof(double));
14   cudaMemcpy(d_x, x, N*sizeof(double),
         cudaMemcpyHostToDevice);
15   axa<<<N/256, 256>>>(3.0, d_x);
16   cudaMemcpy(x, d_x, N*sizeof(double),
         cudaMemcpyDeviceToHost);
17   std::cout<<"Output:"<<x<<std::endl;
18   cudaFree(d_x);
19   free(x);
20 }
```

Fig. 1. Simple CUDA program.

II. Background and Motivation

In this section, we provide a brief description of the baseline GPU architecture, including the execution model. The baseline models a Nvidia discrete GPU architecture. Hence, we will use Nvidia/CUDA terminology throughout this paper. However, the ideas also apply to GPUs from other vendors [28]–[31]. We also discuss checkpointing, which plays a key role in our method.

A. Baseline Architecture

1) GPU Program Execution: Typical GPU programs contain two parts of code: 1) host code that runs on the CPU and 2) device code (*kernels*) that runs on the GPU. Kernels are executed in a single instruction, multiple threads (SIMT) fashion. A kernel is executed by running multiple threads in parallel on the GPU. Threads are grouped into TBs by the programmer.

Nvidia's CUDA programming model for GPUs is exposed to the programmer through CUDA C, an extension to the C language, and runtime libraries. Fig. 1 is a sample CUDA code. The typical sequence of operations for a CUDA C program includes the following.

1) Declare and allocate host and device memory (lines 8–13).
2) Migrate data from the host to the device (line 14).
3) Launch the kernel(s). Here, the programmer launched N/256 TBs, each containing 256 threads (line 15).
4) Migrate results from the device to the host (line 16).
5) Release the memory space (lines 18 and 19).

TBs are considered independent from each other and are dispatched to SMs separately. The number of TBs that may execute concurrently is limited by the device's resources (registers, shared memory, and thread number), which is known at compile time. Most prior proposed preemption schemes work

at the TB granularity level, using the resource information to make preemption decisions.

2) GPU Architecture: Fig. 2 is the baseline GPU architecture, which we will refer to throughout this paper. A GPU program receives operation commands from the host CPU during execution. The user-space runtime engine transforms API calls to control data operations and kernel launches [32]. The GPU device driver sends these operation commands to the queues in the stream manager. The stream manager manages multiple streams using software queues; all commands in the same stream execute serially. Typically, the CPU first declares and allocates its memory and then invokes cudaMalloc to allocate the global memory on the GPU. Then, a cudaMemcpy (HtD) call moves the data from the host to the device. Once all data is transferred, the stream manager can launch the kernel by passing kernel information (such as dimension configurations and entry PC address) to the kernel management unit (KMU). When all the information is ready, the kernel requests SM resources. If there are not enough resources, the kernel waits in the kernel pending pool. If the waiting kernel has higher priority than executing kernels, it may preempt executing TBs in an SM to obtain resources. Otherwise, it waits for previous kernels to finish.

Once the kernel is ready for execution, it is transferred to the kernel distributor unit. Its TBs are then dispatched to SMs by the CTA Scheduler. The maximum number of TBs that can be executed on an SM depends on resource constraints, including the number of resident TBs, threads, registers and shared memory space. During the execution of a kernel in the SM, TBs are split into warps which are groups of 32 threads. An SM has one or more warp schedulers that choose which warp to issue. Each warp scheduler controls 32 stream processors (SPs) in GTX980 GPU architecture [16]; each SP computes a single thread. The scheduler switches among warps if a current warp is stalled by a memory access or other long operations. There is no overhead for switching among warps in an SM, as all warps' contexts are already in the registers and shared memory. As a result, the GPU can hide the delay of stalled warps, improving overall performance.

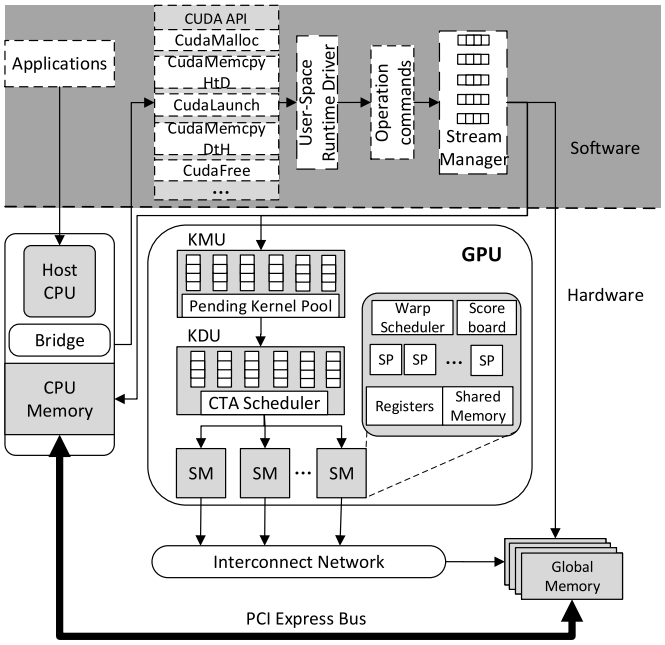


Fig. 2. Baseline GPU architecture.

B. Prior Preemption Methods

When preemption occurs, each SM can operate independently, meaning that some may be preempted while others may continue to execute. Preempted SMs need to save their context to the global memory. An SM's context is its execution state, which includes the SIMT stack, registers and shared memory. The SIMT stack stores the thread execution information, such as the program counters and active masks (used for branch divergence). Compared with the size of the registers and shared memory, the SIMT stack size is negligible, so we ignore it for the remainder of this paper. A TB owns its portion of SM's resources while it is active; it remains active until all its threads complete. However, during its own execution, there may arrive a new kernel that has a strict deadline to meet. The new kernel may not be able to wait for the current kernel to finish, as that may violate the deadline. Therefore, we need to preempt some active TBs to make room for the new kernel's TBs. However, as the context of TBs are typically large, saving them to the global memory incurs high overhead, so the preemption latency is prohibitively long. As shown in Table I, the preemption latency (Avg Switch Time) can be over 20 μs. This latency could pose threat to meeting the deadline of the incoming kernel.

To achieve a lower preemption latency, Park *et al.* [21] proposed one technique that simply flushes a TB. In this method, the SM drops the TB's context without saving it and directly executes new TBs from the higher priority kernel. After that kernel is finished, the SM re-executes the dropped TB from the beginning. Flushing has almost no preemption latency. However, not all kernels can be flushed at any point in execution. Flushing requires the kernel to be idempotent, meaning the kernel will generate the same result independent of how many times it is executed, i.e., there are no atomic operations nor global memory writes before the point of flushing. Most applications are not idempotent (only 30% in Rodinia) [24]. Idempotence may be relaxed, but requires much bookkeeping overhead. In either case, flushing may have high overhead, which is proportional to the number of instructions that are re-executed.

To achieve low preemption overhead, SM draining was proposed, where the executing TB will run till completion before new TB from the incoming kernel starts [19], [21]. This method does not require any context saving so the preemption overhead is minimized. However, the preemption latency can be very high because the executing kernel may be of long latency. This could lead to deadline violation for the incoming kernel. Table I includes the execution time of a TB for various kernels we measured. As we can see, some TBs (e.g., Kmeans) have an execution time of nearly 1 ms. Therefore, SM draining is best suited for short-latency TBs.

Lin *et al.* [18] proposed a lightweight context switching to reduce context size that need to go off-chip. Those techniques

TABLE I
BENCHMARKS TIME COMPARISON

Benchmarks	Source	Kernel	Avg Launch Time	Avg TB Execution Time	Avg Switch Time	Context Size Per TB	TB num Per SM
CUTCP(CP)	Parboil[25]	cuda_cutoff _potential	5.8μs	516.2μs	10.1μs	16.5KB	8
LBM	Parboil[25]	performStream Collide_kernel	21.8μs	31.7μs	20.9μs	18KB	14
MRI-Q(MRI)	Parboil[25]	ComputeQ_GPU	10.4μs	865.2μs	11.6μs	18KB	8
STENCIL(ST)	Parboil[25]	block2D_hybrid _coarsen	4.5μs	41.3μs	4.2μs	12.5KB	4
STREAM CLUSTER(SC)	Parboil[25]	kernel_ compute_cost	6.7μs	605.6μs	8.3μs	24KB	4
GEMM(GM)	Darknet[27]	matrixMulCUDA	23.4μs	193.6μs	17.9μs	28KB	8
BLACK SCHOLES(BS)	Nvidia SDK[26]	BlackScholarGPU	3.4μs	387.5μs	16.7μs	12.5KB	16
KMEANS(KS)	Rodinia[24]	invert_mapping	29.7μs	984.7μs	9μs	10KB	8
PATHFINDER(PF)	Rodinia[24]	dynproc_kernel	11.3μs	24.2μs	11.6μs	18KB	8
SRAD_V1(SRAD1)	Rodinia[24]	extract	5.2μs	1.8μs	4μs	12KB	4
SRAD_V2(SRAD2)	Rodinia	srad_cuda	15μs	11.5μs	16.4μs	25KB	8
SRAD_V1(SRAD3)	Rodinia[24]	srad	5.2μs	7.9μs	7.8μs	24KB	4
HOTSPOT(HS)	Rodinia	calculate_temp	33.3μs	4.5μs	7.7μs	38KB	3
LUD	Rodinia[24]	lud_internal	4.4μs	5.3μs	10.5μs	16KB	8
BACKPROP(BP)	Rodinia[24]	bpnn_ layerforward	16.7μs	4.7μs	2μs	12KB	1
BACKPROP(BP2)	Rodinia[24]	bpnn_adjust _weights	16.7μs	1.5μs	1.2μs	22KB	1

include in-place context switching, which saves context in unused registers and shared memory, dead register removal, which reduces the context size, and register value compression. We also leverage the in-place context switching in PEP. However, incorporating the liveness information requires a liveness bit for each register per instructions, which implies a large liveness table stored in hardware. To reduce this large overhead, preemption is allowed only at certain points of a kernel to reduce the storage requirement for liveness information. Further, the register value compression algorithm also introduces additional hardware overhead [33].

C. Checkpointing in GPUs

Checkpointing is to save the state of a running process so that it may be resumed later in the event of faults. Checkpointing in GPUs has been implemented in software [34], [35]. Even though checkpoints allow a process to resume, they are not directly suitable for preemption as their purpose for checkpointing is for fault tolerance. The device running a process may fail, so it is necessary to save its state on another device. This is a long latency operation, but the overhead is acceptable compared to the work lost in case of a fault. For preemption, our goal is a reasonable response time for the preempting kernel as it may need to meet a close deadline. Thus, we save our context to the device's global memory. Checkpointing is used to shorten the latency of a future context switch. To introduce checkpointing into preemption, it is important to limit the number of checkpoints, as another goal of ours is to reduce the overhead in preemption.

D. Motivation

Chimera [21] uses a selection algorithm to choose the preemption methods for different TBs during a preemption request; the selection is based on the tradeoffs of the three methods. Chimera estimates each technique's preemption latency and overhead to choose the most effective preemption method. Consequently, different TBs in the SM may be preempted with different preemption techniques.

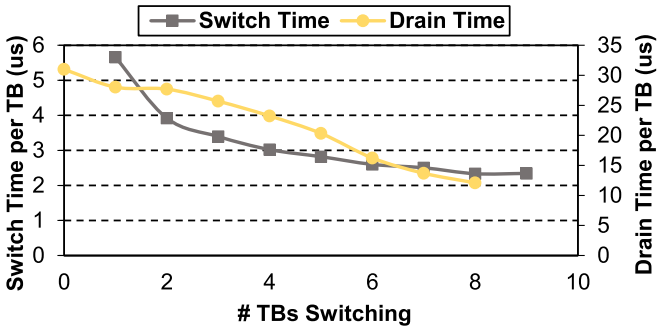


Fig. 3. Switch time and drain time for LBM (9 TBs per SM).

However, we observe that draining and switching compete for the global memory bandwidth. For example, in Fig. 3, the memory intensive application LBM has such a conflict. LBM has nine TBs per SM; we show all ten possible combinations of switching and draining. If all TBs are switched one by one, there is no competition. In the case, where one TB is switched and all others are drained, both the drain time and the switch time are longer than the situation in which eight TBs are switched and one TB is drained. This is because the one switching TB competes with the other eight draining TBs. Therefore, we find that bandwidth competition causes Chimera's estimation to be inaccurate. In the case, where one TB is drained, there is not bandwidth competition because TBs switch one at a time. Also, the draining TB has no competition

for execution units. Thus, the IPC may be affected by the number of TBs draining.

We also observe that the decision making between switching and draining can usually be the same for all TBs. Table I shows that the range of TB execution times is much larger than the range of switch times. There may be a wide gap between the drain time and the switch time for long kernels. Hence, the best way for preempting short kernels is to drain all executing TBs, which can meet the deadline with almost zero overhead. However, we must context switch for long kernels if they are not idempotent.

The total context size of current GPU [36] is 352KB per SM (256KB for registers and 96KB for shared memory). To transfer this context to global memory, it takes at least 15 μs, assuming the bandwidth is fully utilized. As all previous techniques are reactive, they must take at least this amount of time to switch. To further reduce the preemption time for context switching, we need not only to reduce the context size but also preempt with a proactive technique.

Checkpointing is a proactive mechanism widely used in fault tolerance; it saves the state of the running process periodically. Similarly, we can also save the context of the running TBs for preemption. To implement proactive preemption, we introduce PEP, our checkpoint method. We can save a checkpoint context before preemption, and when the actual preemption is invoked, we perform an incremental update relative to the checkpoint, which shortens the actual preemption latency.

III. Design

In this section, we first give an overview of our proactive preemption design. Then we demonstrate the feasibility of predicting kernel launch time and estimating preemption time. Finally, we propose the designs of our checkpoint method and runtime selection algorithm.

A. Overview

Our method is based on the observation that context switching may be used at any time during a TB's execution phase, as long as the latency and overhead are acceptable. To reduce the latency and overhead, we will reduce the context size. To reduce the preemption latency, we can context switch earlier. We also use draining in appropriate cases, as there is almost no overhead.

To reduce the context size, we use a dirty bit to indicate whether a register is active or not. Thus, we never save context that is unused or has been released. We also leverage in-place context saving proposed by Lin *et al.* [18], which allows context to be saved in idle local memory. In this method, no data transfer to global memory through the interconnect network is required.

For proactive context switching, we use checkpointing. Our algorithm can decide to save the context at a checkpoint to the global memory, prior to preemption. Then, we continue execution until the preemption occurs. At this moment, we only need to save an incremental update to the base checkpoint. If a TB finishes execution after the base checkpoint but before a preemption request, we just release this base context. This method achieves much lower overhead than a full context save.

To do checkpointing for preemption, we must limit the number of checkpoints; if we checkpoint too often, the overhead may be unacceptable. Moreover, if a TB finishes before preemption but after some checkpoints, the previous checkpoints are wasteful and contribute to overhead. For these reasons, it is necessary to predict which TBs will still be executing at the preemption time point. Then, we can only save checkpoints for those TBs. We observe that the CUDA API call cudaLaunch is always the last software operation before a new kernel is launched. After this call, a kernel launch command is sent to the stream manager. If the command is at the head of the stream queue, the kernel information will be passed to the KMU and start requesting SM resources. Hence, we find that the kernel launch time can be predicted.

Our checkpointing method is suitable for long kernels. For those short kernels, we still perform draining instead of context switching. To utilize both preemption techniques, we estimate both draining and switching times to select the preemption method during runtime.

B. Prediction and Estimation

The prediction of kernel launch time and estimation of draining and switching time are key components of PEP. Through our studies on a collection of various applications, we find that there are three timings that are critical to the success of our prediction scheme: 1) kernel launch time; 2) context switch time; and 3) TB execution time. Kernel launch time is what we use to predict when the preemption will actually need to occur. Context switch time and TB execution time are used to determine if checkpointing or draining need to be performed. Table I shows our measurements for the three timings.

1) Prediction: From Table I we have two important observations. The first important observation we make is that the kernel launch time and the context switch time (close to the latency of checkpointing) are on the same order of magnitude. This implies that if we start checkpointing at the time of prediction, then when preemption request actually occurred, we would have just finished saving necessary context. The second observation is that both those timings have much smaller variation than the TB execution time. For long-running TBs, the inaccuracy in prediction would not make a difference nor impact the decision on checkpointing or draining. For short-running TBs, the decision is most likely draining, so misprediction would not have much impact on the final overhead or latency either. We will elaborate those timings in this section.

As already mentioned, we must be able to predict when preemption occurs in order not to wastefully checkpoint. A CUDA application typically has five steps, which can be marked by five CUDA API calls: 1) cudaMalloc; 2) cudaMemcpy (H2D); 3) cudaLaunch; 4) cudaMemcpy (D2H); and 5) cudaFree. The cudaLaunch call triggers the kernel launch action. It passes kernel information to the GPU, including TB organization information (grid and block dimensions) and the

amount of shared memory allocated. We tested on a large number of applications and observed that the kernel launch time is typically in the order of tens of microseconds. Table I shows that, for the set of applications we examined, the kernel launch time ranges from 3.3 to 33.3 μs. This is the time from when cudaLaunch is called to the time that kernel information arrives to the KMU, assuming no queuing in the stream manager. The high kernel launching time includes the software API call and copying the kernel code itself as well as copying the arguments to the pending kernel pool in the GPU [37]. Hence, the overhead can vary quite a bit.

In addition, the average switch time per TB is from 1 to 20 μs, which depends on the context size per TB. From Table I we know that the kernel launch time and the switch time are in the same order of magnitude. Approximately, the length of context switch time can be similar to the kernel launch time. This means that when the first base checkpoint is finished, the preemption probably has occurred. In this case, we can release resources immediately and make room for new kernels. Our design does not require a very precise prediction for kernel launch time. This is because if checkpoint finishes before the actual preemption request arrives, the SM can continue executing the TB until the preemption starts and save the dirty context only.

We also find that the average TB execution time ranges from 1.5 μs to more than 900 μs from Table I. The execution time varies a lot depending on the length of the kernel. Short kernels' TBs will be drained, which costs almost no overhead and does not affect meeting the deadline, as the draining time is short. Only long kernels' TBs will be more likely to perform context switch. As those long TB execution time can be as long as hundreds of microseconds, it is easy for us to roughly predict whether the TB will be preempted or not at the time cudaLaunch is called. We can set a certain kernel launch time, such as 20 μs for prediction purpose. When a cudaLaunch is called by the GPU driver, we compare the predicted kernel launch time with the remaining TB execution time for each TB. If predicted kernel launch time is smaller than the remaining TB execution time, then we can start checkpointing right away. Otherwise, we will drain the TB. Note that the variation of the kernel launch time is relatively small, compared with TB execution time. Hence, even if the true kernel launch time is away from 20 μs, it is unlikely to cause a different preemption decision.

In reality, the kernel launch may be delayed due to queuing time in the stream manager; for example, a preceding long memory copy operation may not yet be finished. However, this delay is not problematic for our algorithm. In the case, where the average TB execution time is much larger than launch time, such as for CUTCP, the delay is not likely to be large enough for the TB to finish execution, so our checkpoint scheme is not wasteful. In the case, where the average TB execution time is similar to the launch time, we will choose to drain, so the delay definitely does not affect checkpointing overhead.

Our prediction scheme will ensure that the number of checkpointing performed is no more than two. The base checkpointing is triggered by the cudaLaunch call and the incremental checkpointing is triggered by the preemption request. Therefore, the checkpoint overhead can be limited.

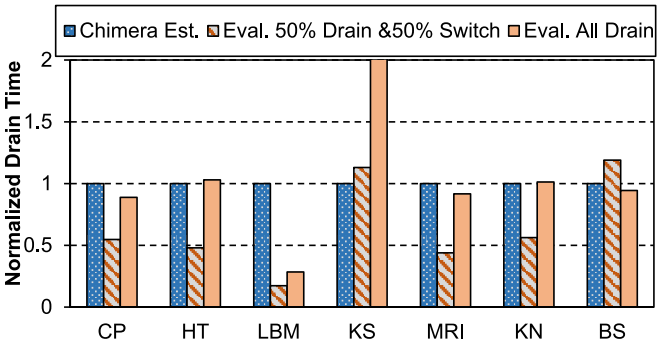


Fig. 4. Drain time estimated by Chimera.

2) Estimation: We must estimate the context switch time and the drain time in order to select the preemption method, either checkpointing or draining. We also need the estimation to predict if the preemption occurs within the current TB or not. Chimera [21] uses time estimation to compare the throughput overheads between different preemption methods. Chimera estimates the drain time for a TB as the product of the remaining instructions and the previous CPI of the TB; the context switch time is the context size of the TB divided by the global memory bandwidth shared by the SM. However, the TB-based estimation is inaccurate in some cases; further, it is inestimable when context switching overlaps draining. As shown in Fig. 4, when half of TBs are draining and others are switching at the same time, the estimations are quite off. This is because fewer draining TBs means fewer conflicts on SPs and more conflicts on the bandwidth of the global memory with switching TBs.

Although Chimera's estimation for draining all TBs is much better, there is still significant inaccuracy in certain cases. For example, applications like LBM and Kmeans (KS) have multiple phases; their CPI is time-varying. In Fig. 4, KS has very low CPI at the beginning, but its CPI increases during execution. Thus, the estimation time is far off from the evaluated time. In addition, the switch time estimation only considers a single TB by itself when making the preemption decision. For example, if three similar TBs are switched, the context size is three times as high; thus the actual total switch time can be three times larger than Chimera's per TB estimation.

From Table I, there is a wide gap between TB execution time and TB switch time. In most applications, we will choose to drain all TBs or switch all TBs. Hence, the drain time and context switch time are estimable; we do not need to worry about switching/draining interference. To avoid the impact from the CPI time-variance, we profile previous TBs in the runtime. As the instructions are the same between different TBs in the same kernel, the TB execution time is stable. Thus, we can use average TB execution time (profiled) minus the already executed time to obtain the remaining TB execution time. However, if there is no profiled TB execution time available when we need to estimate, we can use Chimera's estimation. To estimate context switching time, we always use the worst case estimation, which estimates the time for switching

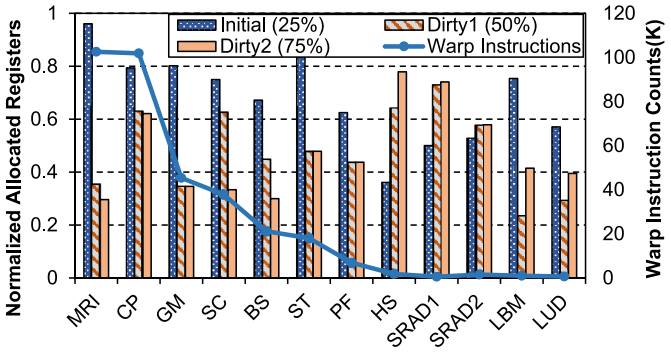


Fig. 5. Normalized dirty registers. Initial: 25% of the TB progress. Dirty1(Initial): 50% of the TB progress. Dirty2(Initial): 75% of the TB progress.

all the TBs in an SM. As the context switch time has a small range compared to execution time, we are safe but not too conservative in using the worst case estimate.

C. Context Reduction

Traditional context switching saves all allocated context to global memory. However, the active context at a particular point in time is always smaller than its allocated size, allowing us to save less. We track the active context using dirty bits. However, TBs mainly have two sources of context: 1) registers and 2) shared memory, which have different lifetimes. Shared memory is private per TB. As it is managed by the programmer, we consider its lifetime to be the whole lifetime of the TB. On the other hand, registers are allocated per thread, and threads are executed in a warp group. Thus, a register's lifetime is per warp. When a warp is finished, all registers allocated for these threads are released. To track register use, we set a dirty bit once a register is written to in the writeback stage, and we unset if we checkpoint, or whenever the warp finishes. We can similarly track shared memory writes.

Fig. 5 shows dirty register size for applications, normalized by allocated size. We collect the dirty register percentage for different execution progress points. Our initial collection is at 25% execution. Dirty1 and Dirty2 are the percentage of dirty registers at 50% and 75% of the TB execution progress, relative to the initial collection. For the kernels with a large number of warp instructions (MRI through PF), Dirty1 and Dirty2 are reduced by 38.2% and 48%, on average, from the initial state. In general, Dirty2 has less dirty registers than the Dirty1 because many warps are finished at the 75% point; these warps' registers have been released. We find that this dirty analysis is enough; we do not use a compiler liveness analysis nor register value compression in this paper.

We also leverage in-place context saving [18] at the incremental checkpoint. In-place context switching can be used because the new kernel can use free space in the SM that the old kernel did not use. This further reduces the actual preemption time.

D. Proactive Preemption Design

1) Checkpoint Saving: Checkpointing will only be used for long running preempted kernels, as their drain times are too long. When a kernel is running on the SMs, if cudaLaunch is called by the GPU driver, we know that a new kernel will be transferred to the GPU within several to tens of microseconds. At this moment, the GPU driver sends a signal to activate the microprogrammed trap routine [38]. It is implemented by command queues and the memory-mapped register [39]. Current GPUs may expose registers that can be poked by developers to force preemption, but not by end users. When a preempting kernel launch is detected, a base checkpointing command will be written to the command queue which will further modify the memory-mapped register to start the base checkpointing in each SM [40]. We measure the signal transferring time in the NVIDIA GTX 1060 GPU. This latency is around 1.3 μs and fairly constant across different applications. The signal triggers a base checkpoint saving. We pause fetching new instructions, and drain the pipeline. Otherwise, the state of checkpoint context will be inconsistent. If the current kernel is compute-intensive, this process may only take tens of cycles. If it is memory-intensive, we must wait for the memory request to return. Thus, pipeline draining time can be hundreds of cycles per SM. The context of the base checkpoint is dirty registers and shared memory corresponding to the initial state.

When the checkpointing is done, all the dirty bits are cleaned. Then, the GPU checks whether the new kernel is transferred to the KMU or not. If it is in the pending kernel pool, it can start execution once it obtains SM resources. Then, the current kernel can be preempted immediately, as the current execution state has been saved. Otherwise, the current kernel will continue executing until the actual preemption request arrives. When the actual preemption request arrives, we only need to save the incremental update. The incremental checkpoint is the dirty context corresponding to the base checkpoint, which is much smaller and takes much less time. Since the dirty context of the incremental checkpoint is those context which is modified after the base checkpoint, no redundant data is saved. As above, the base checkpoint saving occurs when the cudaLaunch for new high priority kernel is called. Thus, the new kernel will be launched soon; therefore, the incremental update will almost surely be small. Further, with in-place context saving, the amount needed to be saved can be reduced even more.

Restoration of the preempted kernel is similar to conventional checkpointing. If we have two checkpoints' states to restore, we must restore one by one. However, at this time, the SM will be idle; thus, the full bandwidth can be used for the context restore.

2) Runtime Selection: As we know from the Table I, execution time, context size, and launch time can vary among kernels. Hence, when cudaLaunch triggers our proactive preemption mechanism, there are several possibilities. Fig. 6 shows the possibilities.

1) *Two Checkpoints:* Most often case. The kernel launch time is longer than the base checkpoint saving time. When the actual preemption starts, we save an incremental checkpoint relative to the base.
2) *Single Checkpoint:* This is the same as a traditional context switch, but it starts earlier. It occurs when the kernel launch time is shorter than checkpoint saving time.

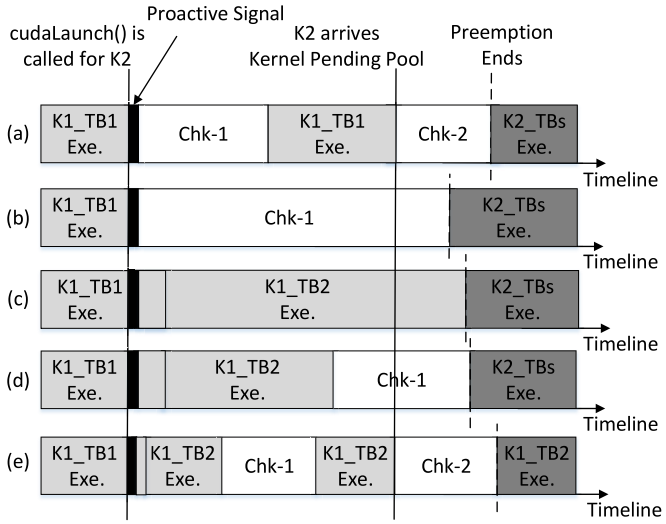


Fig. 6. PEP Possibilities (K1: the preempted kernel, K2: the preempting kernel, Chk-1: the base checkpoint, and Chk-2: the incremental checkpoint).

3) *Drain:* The preempted kernel is short. Its TB execution time is shorter than preempting kernel launch time, possibly finishing before the deadline. In this case, we drain all TBs, achieving very little overhead.
4) *Drain Then Single Checkpoint:* The preempted kernel is the same as in case 2). If TB is nearly finished, the preemption will not occur within the TB. Hence, we will first drain the TB, then a new TB will be dispatched to the SM. The new TB will start execution for a fixed number of instructions before saving the checkpoint. In this paper, we set the number of instructions to be 1000.
5) *Drain Then Two Checkpoints:* The preempted kernel is the same as in case 1). The TB is nearly finished when cudaLaunch is called, which is similar to case 4).

We design a dynamic runtime selection mechanism to handle all possibilities. Fig. 7 illustrates our runtime selection. When cudaLaunch is called for the preempting kernel, we compare the predicted kernel launch time with the current TB's remaining execution time, which is estimated. If the predicted preemption will occur within the TB, proactive preemption starts. If the TB's estimated drain time is longer than the switch time, we define this kernel as long. Cases 1) and 2) operate on long kernels, collecting active context and saving a base checkpoint to global memory. For the other cases, the kernel's predicted preemption will not occur within the TB's lifetime, so we drain, and a new TB from the current kernel is dispatched. Then, we have to do the prediction and estimation again. If the new TB can drain in time, and then preemption is ready, we have case 3). Otherwise, we are in case 4), which is just case 2) again, or case 5), which is just case 1) again.

E. Hardware Overhead

In order to implement PEP, the GPU needs to be extended with new control logic to mainly implement the following: 1) prediction and estimation units, which involve counters for profiling and comparators for making decision; 2) dirty bits, one bit for each register, totaling 8 KB per SM for the GTX980 GPU [16]; and 3) profiler counters, which are used for collecting TB execution times. Overall, the majority of overhead is largely in the dirty bit storage.

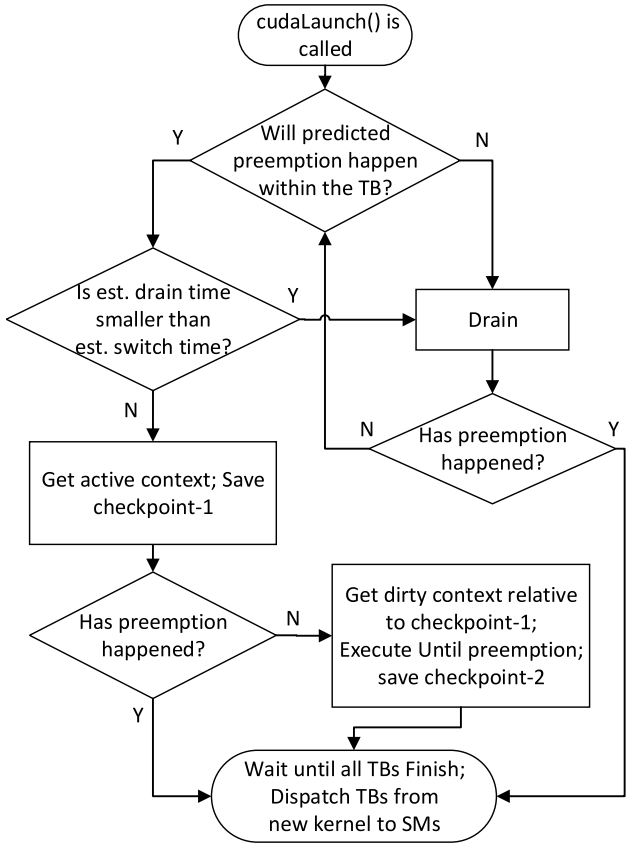


Fig. 7. Runtime selection.

TABLE II
GPGPU-SIM CONFIGURATION PARAMETERS

Configurations	Nvidia Geforce GTX980
Num. of SMs	16
SIMD Width	32
SIMT Core Clock	1216MHz
Memory Clock	7GHz
Memory Controller	4
Schedule Scheme	4 warp schedulers with LRR
Registers	256KB
Shared memory	96KB
TB Limit	32

IV. EXPERIMENTS

A. Methodology

We implement PEP, and for comparison, Chimera, on the latest version of GPGPU-Sim [41]. The system configuration is summarized in Table II. The configuration of 256KB registers and 96KB shared memory reflects the large context in recent GPU architectures. By default, GPGPU-Sim simulates PTX instruction, which is a pseudo-assembly instruction set with unlimited registers. It does not execute directly on the hardware; SASS is the native instruction set run by the hardware. Therefore, we simulate PTXPlus, which is a one-to-one

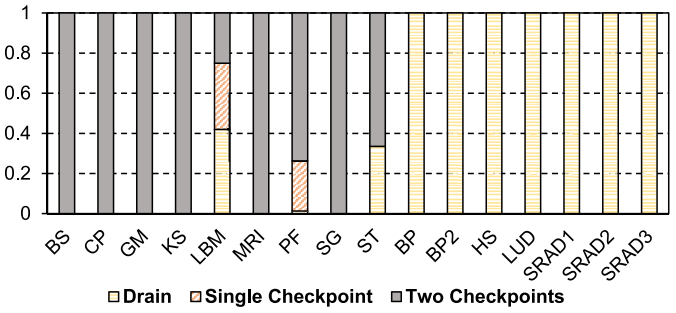


Fig. 8. Preemption technique distribution.

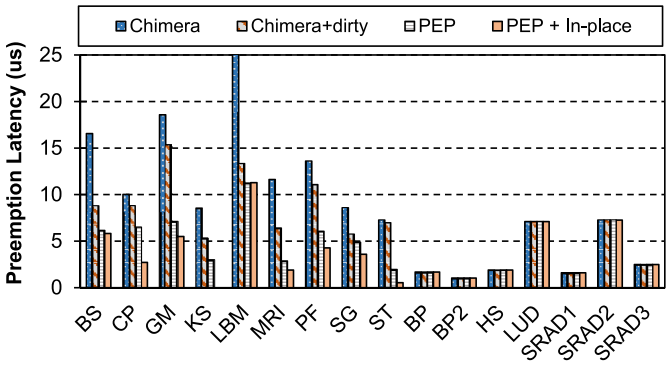


Fig. 9. Average preemption time.

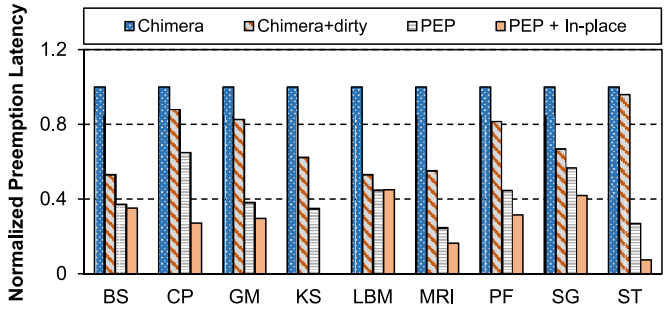


Fig. 10. Normalized preemption time.

mapping from SASS. This is necessary to accurately model dirty registers.

For comparison, we implement Chimera and PEP with different variations: vanilla Chimera, Chimera with dirty context saving, vanilla PEP and PEP with the in-place context saving. We test using a wide range of kernels from GPGPU applications from Nvidia Computing SDK [26], Parboil [25], Rodinia [24], and Darknet [27]. For Chimera, we vary the deadline. We observed that the average context switch time is always smaller than 20.9 μs, so we set deadlines of 5, 10, and 15 μs. For PEP, we vary the preempting kernel launch times, the predicted kernel launch time, and current progress percentage at preemption. The PEP parameters are explained later. We then compare the preemption latency, context size, and preemption overhead of these different designs.

Because GPGPU-Sim does not model the timing from the cudaLaunch API call to the kernel's actual launch time, we design our own experimental method. We profile kernel launch times with Nvidia's profiler [42]; launch times range from 3 to 33 μs in Table I. We then set the preempting kernel launch time as 5, 15, 25, or 35 μs. We also set the predicted kernel launch time as 20 or 30 μs. Moreover, as preemption can happen at any time during the execution of preempted kernels, in order to make our evaluation more comprehensive, we vary when the cudaLaunch occurs for the preempting kernel. For experimental purposes, we invoke it at 25%, 50%, and 75% of the average TB execution progress of the preempted kernel. Hence, each application runs 24 times, exhausting all possibilities. The results shown in the following sections are averages over a particular parameter.

B. Selection Distribution

In Fig. 8, we collect the runtime selections of all TBs for each application. TBs which do two checkpoints are only from the six applications with average TB execution time longer than 100 μs (see Table I). On the contrary, applications whose TBs all choose to drain are from short kernels with short average TB execution time. For LBM, the average TB execution time is 30.1 μs, while the average switch time is 20.9 μs; the drain time and switch time are comparable. This closeness is what allows for the variation of choice between draining or checkpoint, varying over the percent progress parameter. Similarly for pathfinder (PF), the base checkpoint saving time is similar to the preempting kernel launch time. This closeness allows for the variation of choice between a single checkpoint or two checkpoints, varying over the percent progress parameter. In the case of single checkpoint, we save overhead and shorten latency by avoiding the second checkpoint.

Since there is a large gap between the average drain time and average switch for most kernels, we usually choose a single preemption method for all their TBs. Choosing a single method means there is no bandwidth competition between draining and switching TBs. Hence, our estimates on latencies do not suffer from interference of memory contention.

C. Preemption Latency

Fig. 9 shows the preemption latency, which is measured from time of the arrival of kernel at KMU to the last TB's context is saved. This is also the actual waiting time for the preempting kernel in the kernel pending pool. We observe that the last seven kernels which drain all TBs per SM achieve low latency. These applications also have short TB execution time. Although Chimera suffers from inaccurate time estimation, it still chooses the same preemption techniques as PEP; this is due to the wide gap between the drain time and the switch time, which does not require high accuracy. Accordingly, all four designs have the same latency for draining; the average drain time is 3.4 μs. However, PEP and PEP+In-place reduce the total average preemption latency from 8.9 μs in Chimera to 4.5 and 3.6 μs, respectively. A shorter preemption latency allows kernels to meet a stricter deadline, increasing its usefulness for multitasking.

Fig. 10 shows normalized worst case preemption latency. The first nine applications do not choose to drain all TBs. Two main factors affect their preemption latency: 1) the pipeline

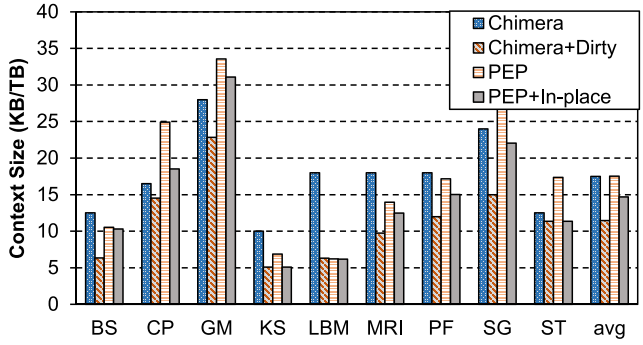


Fig. 11. Context size comparison.

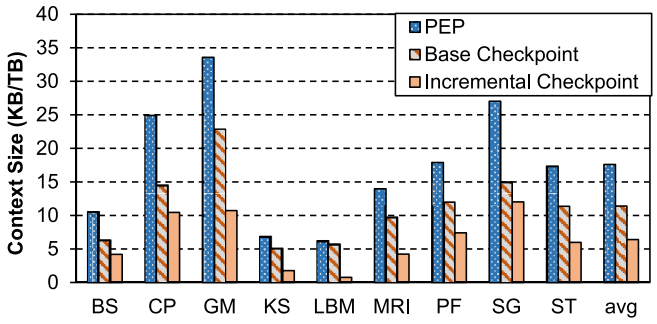


Fig. 12. Context size per TB.

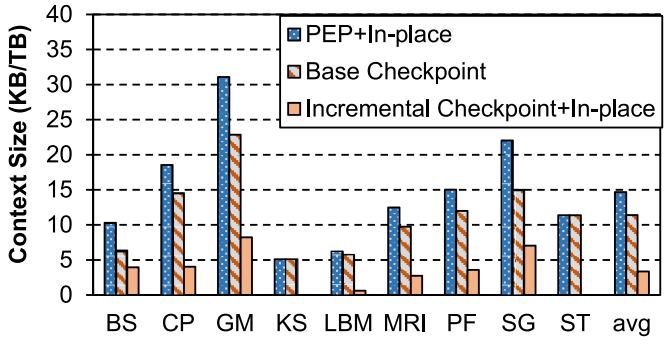


Fig. 13. Context size with in-place per TB.

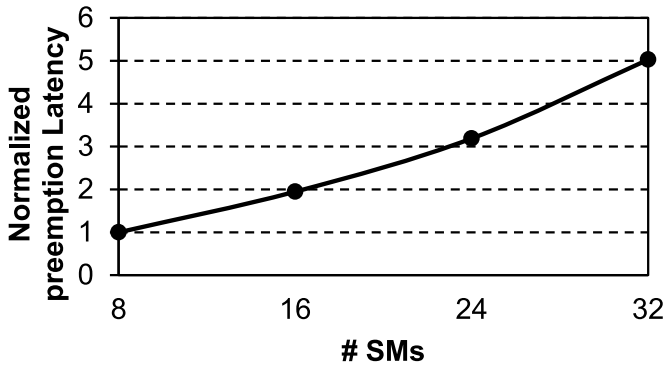


Fig. 14. Sensitivity to the num. of SMs.

draining time and 2) the total context size; however, the context size is the key factor. Compared with Chimera, Chimera with dirty context saving reduces the preemption latency by 31.8%, as it reduces the saved context size. Compared with Chimera, PEP reduces the average preemption latency by 58.5%; it reduces by 70.3% if we leverage in-place saving. For KS, PEP with in-place saving has zero preemption latency, because the dirty context size for incremental checkpoint is very small; it can completely be saved in place.

D. Context Size Reduction

For our experimental context, we only consider registers and shared memory. Properly used shared memory should be accessed frequently, because programmers use it in order not to pay the global memory latency cost. Therefore, the shared memory may become too dirty, causing unacceptable overhead. Thus, we only use dirty bits for registers; we always save the whole allocated shared memory. In our results, "context size" refers to only the context which must be saved to global memory.

Fig. 11 compares the context size among designs. Saving only dirty context, Chimera can reduce the average context size by 6KB per TB, which is 34.4% of its average total context size. Since PEP may save checkpoint states twice, the average total context size for PEP will be larger than Chimera+dirty; however, it is almost the same as original Chimera. However, PEP can further reduce the context size by 16.2% with in-place saving.

Figs. 12 and 13 show the context size details for both PEP and PEP with in-place saving, respectively. For applications that choose to context switch all the TBs, the total context size is the sum of the base checkpoint and the incremental checkpoint. The result shows that the context size of the incremental checkpoint is only 56.1% of the base checkpoint, on average. The in-place saving can further reduce the context size for the incremental checkpoint. The result in Fig. 13 shows that the context size in the incremental checkpoint averages 3.34KB per TB, which is only 29.4% of the average context size in the base checkpoint. We can see that two checkpoints reduce the context size significantly.

E. Sensitivity Analysis

In this section, we measured the sensitivity of scalability and bandwidth. We vary the number of SMs but maintain the same number of memory partitions and the same bandwidth to analyze the impact on the preemption latency of the checkpointing scheme. Fig. 14 shows that the preemption latency increases almost linearly with the increasing number of SMs due to increase memory traffic. Since the context inside each SMs are the same, more SMs lead to high contention in memory bandwidth. As shown in Fig. 14, the average preemption latency of 32 SMs is 2.58 times longer than the average preemption latency of 16 SMs, which means the checkpointing scheme is sensitive to the memory bandwidth. Hence, this result further proves the decision in that instead of overlapping the execution and the context switching, all bandwidth should be provided for the context switching to accelerate the preemption.

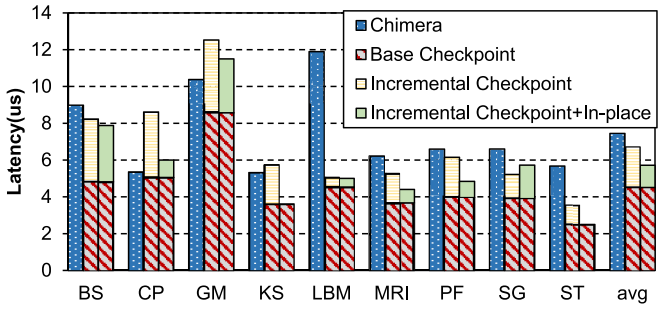


Fig. 15. Context saving overhead.

F. Impact of Preemption Overhead

The overhead for preemption is the idle time of execution units caused by preemption. This is shown in Fig. 15. When the SM is switching context, switching TBs must stop fetching instructions and stop execution. SPs idle for both context swap out and context restore. The only difference between swap out and restoration times is that we drain the pipeline before swapping out. Hence, we only compare the average context saving latencies per TB as overhead.

Fig. 15 shows that the base checkpoint reduces the average overhead by 37.9% from Chimera. The overhead of the base checkpoint is similar to Chimera with dirty context saving. With two checkpoints, the overhead of our PEP is still 6.3% lower than Chimera. When using the in-place context saving to reduce the context size of the incremental checkpoint, the overhead of PEP can be further reduced by 16.4% on average. Some applications do have higher overhead when compared with Chimera. In these cases, the register reuse rate is high, so the dirty context size is larger. Also, with two checkpoints, more time is needed to drain the SM pipeline. However, a the context size and the context switch overhead are positively correlated, some applications, like LBM and ST save more than 50% of overhead, due to a small dirty context size.

V. Related Work

The main focus of GPU preemption research is reducing the preemption latency and overhead; it is prohibitive to use CPU methods naively as the context size for GPUs is much larger. In addition to traditional context switching, Tanasic *et al.* [19] proposed SM draining, which works for preempted kernels which are relatively short. Park *et al.* [21] proposed the SM flushing. It can achieve zero preemption latency for idempotent preempted kernels. Furthermore, their work combines context switching, SM-draining and SM-flushing to work collaboratively based on the progress of TBs. Wang *et al.* [20] designed a fine-grain dynamic sharing mechanism, SMK. Their design enables a fine-grain context switch mechanism on per TB basis to achieve low turnaround time.

To focus on context size reduction, iGPU [43] has the insight that context can be saved and restored at the boundaries between idempotent code regions. They leverage liveness analysis to identify recovery points; these points have a small set of live registers. Lin *et al.* [18] proposed three techniques to implement lightweight context switching, including in-place context saving, liveness analysis, and context compression.

Checkpointing is typically used for fault tolerance [44], [45]. Traditional checkpointing software, such as BLCR [46], supports checkpointing the CPU state by using a custom Linux kernel. This does not work for off chip GPUs because a driver manages GPU memory; thus, BCLR cannot restore its state. CheCUDA [34] was the first attempt to solve this issue for Nvidia GPUs. It is implemented as an add-on for BCLR; it works by sidestepping BCLR. It requires recompilation of applications. NVCR [35] improves upon this, supporting the larger class of applications which use the runtime API. Furthermore, it replaces `libcuda.so`; thus, it can be used without recompilation. Additionally, virtualization is another technique used to checkpoint applications; vCUDA [47] is the first to do such work.

Our checkpointing is not real checkpointing. Actually, it is context saving which behaves similarly to the checkpointing. We utilize the procedure of checkpointing for reducing context size saved during preemption. Unlike any checkpointing which is performed periodically, PEP only checkpoints at most two times (base checkpointing and incremental checkpointing). The base checkpointing is triggered by the system call of preempting kernel launching which belongs to software, while the incremental checkpointing is triggered by the actual preemption signal, which is a hardware signal. State-of-the-art checkpointing needs both context saving and fault recovery units to guarantee reliability, while our PEP checkpointing is more lightweight design as there is not faulty recovery unit involved.

All previous preemption mechanisms are reactive, meaning the mechanism will not start until the preempting kernel is launched and requests resources. Thus, the algorithm must wait for the SM to context switch or drain TBs. By leveraging the kernel launch process, PEP is a proactive technique. By utilizing checkpoint, PEP can obtain a lower latency then other methods, still with acceptable overhead.

VI. Conclusion

In this paper, we proposed PEP, a dynamic and proactive preemption mechanism on GPUs. With only a rough prediction of preempting kernel launch time, we can successfully anticipate preemption before the actual request arrives. We borrow checkpointing from fault tolerance, which allows us to shorten preemption latency. Further, checkpointing can tolerate imperfect predictions. To predict preemption, we leverage the kernel launch process done by the GPU driver. The driver triggers a base checkpointing when it receives a kernel launch command from the CPU. This allows us to later only save an incremental checkpoint as soon as the actual preemption request arrives. Further, SMs can execute between two checkpoints as usual. We also support SM draining for short kernels, and we further borrow in-place context saving to achieve low preemption overhead. For our proactive checkpoint mechanism, we achieve 58.6% average preemption latency reduction and 23.3% average context switch overhead reduction. The average preemption latency is also reduced to 3.6 μs, which allows for stricter deadlines, thus increasing multitasking support.

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for their valuable feedback and improvements to this paper.

REFERENCES

[1] J. A. Anderson, C. D. Lorenz, and A. Travesset, "General purpose molecular dynamics simulations fully implemented on graphics processing units," *J. Comput. Phys.*, vol. 227, no. 10, pp. 5342–5359, 2008.

[2] J. Mosegaard and T. S. Sørensen, "Real-time deformation of detailed geometry based on mappings to a less detailed physical simulation on the GPU," in *Proc. IPT/EGVE*, 2005, pp. 105–111.

[3] S. G. Parker *et al.*, "OptiX: A general purpose ray tracing engine," *ACM Trans. Graph.*, vol. 29, no. 4, p. 66, 2010.

[4] V. Podlozhnyuk. (2007). *Black-Scholes Option Pricing*. [Online]. Available: https://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/BlackScholes/doc/BlackScholes.pdf

[5] M. Abadi *et al.*. (2015). *TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems*. [Online]. Available: http://download.tensorflow.org/paper/whitepaper2015.pdf

[6] *An Introduction to High Performance Computing on AWS*, Amazon, Seattle, WA, USA, 2015. [Online]. Available: https://d0.awsstatic.com/whitepapers/Intro_to_HPC_on_AWS.pdf

[7] *Nvidias Next Generation CUDA Compute Architecture: Kepler GK110*, Santa Clara, CA, USA, Nvidia, White Paper, 2012.

[8] M. Mantor, "AMD Radeon^{TM} HD 7970 with graphics core next (GCN) architecture," in *Proc. IEEE Hot Chips 24 Symp. (HCS)*, 2012, pp. 1–35.

[9] P. Rogers and A. Fellow, "Heterogeneous system architecture overview," in *Proc. Hot Chips*, vol. 25, 2013, pp. 1–41.

[10] *AMD Radeon R9 290X*, Adv. Micro Devices, Inc., Santa Clara, CA, USA, 2018. [Online]. Available: http://www.amd.com/us/press-releases/Pages/amd-radeon-r9-290x-2013oct24.aspx

[11] J. T. Adriaens, K. Compton, N. S. Kim, and M. J. Schulte, "The case for GPGPU spatial multitasking," in *Proc. IEEE 18th Int. Symp. High Perform. Comput. Archit. (HPCA)*, 2012, pp. 1–12.

[12] J. Calhoun and H. Jiang, "Preemption of a CUDA kernel function," in *Proc. 13th ACIS Int. Conf. Softw. Eng. Artif. Intell. Netw. Parallel Distrib. Comput. (SNPD)*, 2012, pp. 247–252.

[13] C. Li, C. Ding, and K. Shen, "Quantifying the cost of context switch," in *Proc. Workshop Exp. Comput. Sci.*, 2007, p. 2.

[14] S. Pai, M. J. Thazhuthaveetil, and R. Govindarajan, "Improving GPGPU concurrency with elastic kernels," *ACM SIGPLAN Notices*, vol. 48, no. 4, pp. 407–418, 2013.

[15] *Nvidia CUDA C Programming Guide*, Nvidia Corporat., Santa Clara, CA, USA, p. 8, 2011.

[16] *Nvidia Geforce GTX980 Whitepaper*, Santa Clara, CA, USA, Nvidia, White Paper, 2014.

[17] G. Chen, Y. Zhao, X. Shen, and H. Zhou, "Effisha: A software framework for enabling efficient preemptive scheduling of GPU," in *Proc. 22nd ACM SIGPLAN Symp. Principles Pract. Parallel Program.*, 2017, pp. 3–16.

[18] Z. Lin, L. Nyland, and H. Zhou, "Enabling efficient preemption for SIMT architectures with lightweight context switching," in *Proc. Int. Conf. High Perform. Comput. Netw. Stor. Anal. (SC)*, 2016, pp. 898–908.

[19] I. Tanasic *et al.*, "Enabling preemptive multiprogramming on GPUs," in *Proc. ACM SIGARCH Comput. Archit. News*, vol. 42, 2014, pp. 193–204.

[20] Z. Wang *et al.*, "Simultaneous multikernel GPU: Multi-tasking throughput processors via fine-grained sharing," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, 2016, pp. 358–369.

[21] J. J. K. Park, Y. Park, and S. Mahlke, "Chimera: Collaborative preemption for multitasking on a shared GPU," *ACM SIGARCH Comput. Archit. News*, vol. 43, no. 1, pp. 593–606, 2015.

[22] L. Shi, H. Chen, and T. Li, "Hybrid CPU/GPU checkpoint for GPU-based heterogeneous systems," in *Proc. Int. Conf. Parallel Comput. Fluid Dyn.*, 2013, pp. 470–481.

[23] S. Kannan, N. Farooqui, A. Gavrilovska, and K. Schwan, "Heterocheckpoint: Efficient checkpointing for accelerator-based systems," in *Proc. 44th Annu. IEEE/IFIP Int. Conf. Depend. Syst. Netw. (DSN)*, 2014, pp. 738–743.

[24] S. Che *et al.*, "Rodinia: A benchmark suite for heterogeneous computing," in *Proc. IEEE Int. Symp. Workload Characterization (IISWC)*, 2009, pp. 44–54.

[25] J. A. Stratton *et al.*, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," Center Rel. High Perform. Comput., Univ. Illinois at Urbana–Champaign, IMPACT Rep. IMPACT-12-01, Mar. 2012.

[26] Nvidia. (2018). *NVIDIA Computing SDK*. [Online]. Available: https://developer.nvidia.com/cuda-toolkit

[27] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 779–788.

[28] *OpenCL: The Future of Accelerated Application Performance Is Now*, Adv. Micro Devices, Inc., Santa Clara, CA, USA, 2018. [Online]. Available: https://www.amd.com/Documents/FirePro_OpenCL_Whitepaper.pdf

[29] *ATI Radeon GPGPUs*, Adv. Micro Devices, Inc., Santa Clara, CA, USA, 2018. [Online]. Available: http://www.amd.com/us/products/desktop/graphics/amd-radeon-hd-6000/Pages/amd-radeon-hd-6000.aspx

[30] *Intel Microprocessors*, Intel, Santa Clara, CA, USA, 2018. [Online]. Available: http://www.intel.com/content/www/us/en/homepage.html

[31] *What Is Heterogeneous System Architecture (HSA)?*, Adv. Micro Devices, Inc., Santa Clara, CA, USA, 2013.

[32] S. Kato, S. Brandt, Y. Ishikawa, and R. R. Rajkumar, "Operating systems challenges for GPU resource management," in *Proc. Int. Workshop Oper. Syst. Platforms Embedded Real Time Appl.*, 2011, pp. 23–32.

[33] S. Lee *et al.*, "Warped-compression: Enabling power efficient GPUs through register compression," *ACM SIGARCH Comput. Archit. News*, vol. 43, no. 3, pp. 502–514, 2015.

[34] H. Takizawa, K. Sato, K. Komatsu, and H. Kobayashi, "CheCUDA: A checkpoint/restart tool for CUDA applications," in *Proc. Int. Conf. Parallel Distrib. Comput. Appl. Technol.*, 2009, pp. 408–413.

[35] A. Nukada, H. Takizawa, and S. Matsuoka, "NVCR: A transparent checkpoint-restart library for NVIDIA CUDA," in *Proc. IEEE Int. Symp. Parallel Distrib. Process. Workshops Ph.D. Forum (IPDPSW)*, 2011, pp. 104–113.

[36] *Nvidia Tesla P100 Whitepaper*, Santa Clara, CA, USA, Nvidia, White Paper, Apr. 2016.

[37] S. Puthoor, X. Tang, J. Gross, and B. M. Beckmann, "Oversubscribed command queues in GPUs," in *Proc. 11th Workshop Gen. Purpose GPUs*, 2018, pp. 50–60.

[38] K. Menychtas, K. Shen, and M. L. Scott, "Disengaged scheduling for fair, protected access to fast computational accelerators," in *Proc. Symp. Archit. Support Program. Lang. Oper. Syst. (ASPLOS)*, vol. 42, 2014, pp. 301–316.

[39] Y. Suzuki, S. Kato, H. Yamada, and K. Kono, "GPUvm: Why not virtualizing GPUs at the hypervisor?" in *Proc. USENIX ATC*, 2014, pp. 109–120.

[40] Y. Suzuki, S. Kato, H. Yamada, and K. Kono, "GPUvm: GPU virtualization at the hypervisor," *IEEE Trans. Comput.*, vol. 65, no. 9, pp. 2752–2766, Sep. 2016.

[41] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw. (ISPASS)*, 2009, pp. 163–174.

[42] *Profiler User's Guide*, Nvidia, Santa Clara, CA, USA, Jun. 2017.

[43] J. Menon, M. De Kruijf, and K. Sankaralingam, "iGPU: Exception support and speculative execution on GPUs," *ACM SIGARCH Comput. Archit. News*, vol. 40, no. 3, pp. 72–83, 2012.

[44] R. Teodorescu, J. Nakano, and J. Torrellas, "SWICH: A prototype for efficient cache-level checkpointing and rollback," *IEEE Micro*, vol. 26, no. 5, pp. 28–40, Sep./Oct. 2006.

[45] T. Li, M. Shafique, J. A. Ambrose, J. Henkel, and S. Parameswaran, "Fine-grained checkpoint recovery for application-specific instruction-set processors," *IEEE Trans. Comput.*, vol. 66, no. 4, pp. 647–660, Apr. 2017.

[46] P. Hargrove and J. Duell, "Berkeley lab checkpoint/restart (BLCR) for Linux clusters," *J. Phys. Conf. Series*, vol. 46, pp. 494–499, Sep. 2006.

[47] L. Shi, H. Chen, J. Sun, and K. Li, "vCUDA: GPU-accelerated high-performance computing in virtual machines," *IEEE Trans. Comput.*, vol. 61, no. 6, pp. 804–816, Jun. 2012.

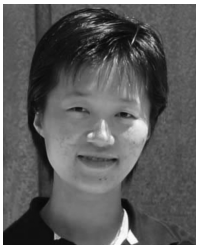
[48] C. Li, A. Zigerelli, J. Yang, and Y. Guo, "PEP: Poactive checkpointing for efficient peemption on GPUs," in *Proc. 55th Annu. Design Autom. Conf. (DAC)*, San Francisco, CA, USA, pp. 1–114, 2018.

Chen Li received the B.E. and M.E. degrees from the National University of Defense Technology, Changsha, China, in 2012 and 2014, respectively, where he is currently pursuing the Ph.D. degree, under the supervision of Prof. Y. Guo.

He was a visiting student with the University of Pittsburgh, Pittsburgh, PA, USA, from 2016 to 2018, co-advised by Prof. J. Yang and Prof. Y. Zhang. His current research interests include GPU architecture, virtual memory, and network on chip.

Andrew Zigerelli received the B.S. degree in mathematics from the University of Pittsburgh, Pittsburgh, PA, USA, in 2015, where he is currently pursuing the Ph.D. degree with Electrical and Computer Engineering Department, under the supervision of Dr. J. Yang.

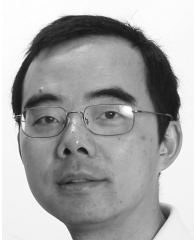
His current research interests include security and GPUs.



Jun Yang (M'14) received the B.S. degree in computer science from Nanjing University, Nanjing, China, in 1995 and the Ph.D. degree in computer science from the University of Arizona, Tucson, AZ, USA, in 2002.

She is a Professor with the Electrical and Computer Engineering Department, University Pittsburgh, Pittsburgh, PA, USA. Her current research interests include GPU architecture, secure processor architecture, emerging nonvolatile memory technologies, and performance and reliability of memories.

Dr. Yang was a recipient of the U.S. NSF Career Award in 2008 and the Best Paper Awards from ICCD 2007 and ISLPED 2013.



Youtao Zhang (M'08) received the Ph.D. degree in computer science from the University of Arizona, Tucson, AZ, USA, in 2002.

He is currently an Associate Professor of computer science with the University of Pittsburgh, Pittsburgh, PA, USA. His current research interests include memory systems, GPUs, and secure hardware designs.

Prof. Zhang was a recipient of the U.S. National Science Foundation Career Award in 2005 and several best paper awards for his co-authored papers. He is a member of ACM.

Sheng Ma received the B.S. and Ph.D. degrees in computer science and technology from the National University of Defense Technology (NUDT), Changsha, China, in 2007 and 2012, respectively.

He visited the University of Toronto, Toronto, ON, Canada from 2010 to 2012. He is currently an Assistant Professor with the College of Computer, NUDT. His current research interests include on-chip networks, SIMD architectures, and arithmetic unit designs.

Yang Guo received the Ph.D. degree from the National University of Defense Technology, Changsha, China, in 1999.

He is currently a Professor with the National University of Defense Technology. He leads the Digital Signal Processor Group and the Director of the Integrated Circuits. His current research interests include low power very-large-scale integration (VLSI) circuits, microprocessor design and verification, and electronic design automation techniques for VLSI circuits