PEP: Proactive Checkpointing for Efficient Preemption on GPUs[*]

Chen Li
College of Computer,
National University of Defense Technology
lichen@nudt.edu.cn

Andrew Zigerelli
Electrical and Computer Engineering Department,
University of Pittsburgh
anz37@pitt.edu

Jun Yang
Electrical and Computer Engineering Department,
University of Pittsburgh
juy9@pitt.edu

Yang Guo
College of Computer,
National University of Defense Technology
guoyang@nudt.edu.cn

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 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. 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.

1 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 [3][22][23]. These computations are often done on cloud providers, who may offer GPUs as a service to be shared between users. Hence, GPU support for multitasking is now necessary. Some preliminary hardware features for multitasking are already in-place, such as Hyper-Q provided by Nvidia's Kepler architecture [15], and the command processor supported by AMD [2][9]. While this was a step in the right direction, much still needs to be done for true multitasking support [1][6][9].

Context switching, a technique used in CPUs to support multitasking [11], has also been proposed for GPUs [1][20]. CPU processes are relatively lightweight, allowing for fast context switching and efficient time-multiplexing. However, a GPU context is massive compared to a CPU context [14]. For example, in the GTX980 GPU [16], the context size may be as large as 256KB for registers and 96KB for shared memory per streaming multiprocessor (SM); the total context size can be 5664KB for the entire GPU (with 16 SMs). Saving such large context takes significant memory bandwidth, severely degrading performance [8][12]. This is especially harmful for latency sensitive applications. Furthermore, context switching increases power consumption [5].

There have been several attempts to reduce the context switching overhead for GPUs. The earliest technique limits context switching to a subset of SMs, so that the remaining SMs can continue execution [26]. 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 TBs in an SM to continue execution while swapping a particular TB [27], which maximally overlaps memory accesses due to context switching with 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 how soon the deadline of the preemption is [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 [12]. 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. In addition, Nvidia Pascal GP100 supports compute preemption feature, which can solve the problem that long-running applications may monopolize the whole system. However, no details are released.

In this paper, we propose a proactive and efficient preemption mechanism, PEP, to reduce preemption latency and overhead. By observing the kernel launch process, we find that the actual execution of a kernel on a GPU is always preceded by the kernel launch

[*]This work is supported in part by the National Natural Science Foundation of China (No. 61672526, No. 61572508), National Science Foundation, CCF-1422331,CCF-1617071,CCF-1718080,and CCF-1725657.

DAC '18, June 24–29, 2018, San Francisco, CA, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5700-5/18/06…$15.00
https://doi.org/10.1145/3195970.3196091

action, and the time from when a kernel is launched from CPU to the time that the kernel starts to execute on the GPU is on the order of tens of microseconds. Leveraging this 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 lessened. Hence, the effective preemption time is shortened. The preparation we perform for context switching utilizes the concept of checkpointing. First, a base checkpoint is performed if 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 context of a preempted kernel.

Moreover, PEP is capable of handling misprediction. If the base checkpoint has saved the incorrect context, we can still save all context at the incremental checkpoint. On average, the total amount of state saved is no more than the full context with simple context size reduction techniques.

Our contributions can be summarized as follows:

(1) We study the kernel launch process, and observe that preemption can be predicted.
(2) We introduce a proactive preemption mechanism to reduce the stall time for the preempting kernel
(3) We use a simple dirty data-saving technique to reduce context size.
(4) We develop a more precise estimation on TB draining time and context switch time, and design a runtime selection algorithm for preemption decisions.

Our experimental results show that we can reduce the average preemption latency from 8.9μs to 3.6μs, 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, using only simple context size reduction techniques. Our total overhead, average switch time per TB, is 6.3% lower than Chimera.

2 BACKGROUND AND MOTIVATION

2.1 GPU Architecture

Figure 1 is the baseline GPU architecture. We use CUDA terminology in this paper, though the description applies to GPUs from other vendors as well. 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[10]. 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 global memory on the GPU. Then, a cudaMemcpy (H2D) call moves 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). Then, 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

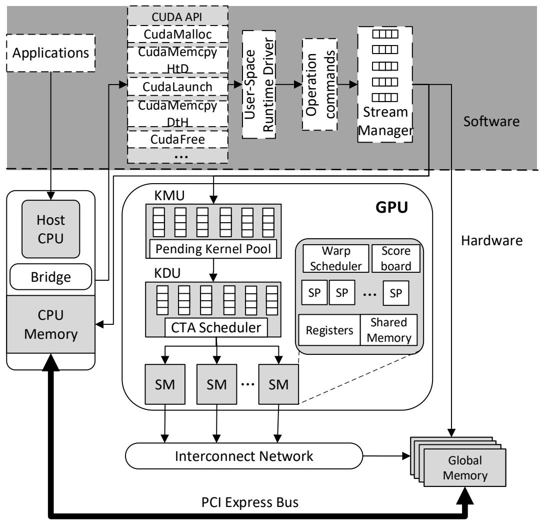

Figure 1: Baseline GPU Architecture

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 (KDU). TBs are then dispatched to SMs by the CTA Scheduler. The maximum number of TBs that can be executed on an SM depends on the resource constraints, including the number of resident TBs, registers and shared memory space [16].

2.2 Motivation

A recent work on optimizing preemption overhead, Chimera [21], chooses a preemption method per TB. Between draining, flushing, and context switching, Chimera chooses the one with least overhead, as long as the resulting preemption runtime allows the preemption deadline to be met. This requires estimating the preemption latency and overhead of each method. Consequently, different TBs in a single SM may be preempted with different methods.

However, draining TBs compete for memory bandwidth with switching TBs, especially for memory intensive kernels. As an example, Figure 2 shows the average switching time per TB as the number of switching TBs increases. The remaining TBs are drained; the total number of TBs per SM is 9 in this example. As we see, when most of the TBs are drained, switching TBs take longer to complete, due to memory interference between draining and switching TBs. The switching time keeps decreasing as more TBs switch.

This leads to our next observation regarding the uniformity of preemption decisions. In most cases, TBs of an SM all either switch or drain. Table 1 shows that in general, for large kernels, TB execution time is much larger than the switching time; there is a wide gap. For short kernels, those timings might be close. Thus, the best way to preempt short kernels is to drain their executing TBs, which best helps meeting the preemption deadline. However for long kernels, it is more beneficial to context switch, as draining all TBs would be too slow.

However, the main challenge to make context switching feasible is its long latency. The total context size of current GPU [17] is 352KB per SM (256KB for registers and 96KB for shared memory). In bad cases, transferring this context to global memory takes at

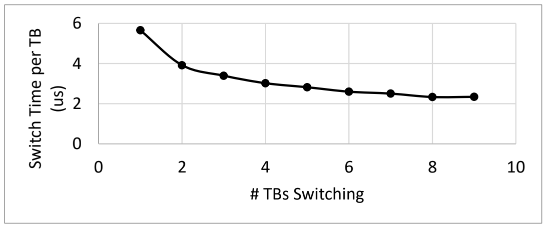

Figure 2: Switch Time for LBM (9 TBs per SM).

Benchmarks	**Source**	**Kernel**	**Avg Launch Time**	**Avg TB Execution Time**	**Avg Switch Time**
CUTCP(CP)	Parboil[25]	cuda_cutoff _potential	5.8μs	516.2μs	10.1μs
LBM	Parboil[25]	performStream Collide_kernel	21.8μs	31.7μs	20.9μs
MRI-Q(MRI)	Parboil[25]	ComputeQ_GPU	10.4μs	865.2μs	11.6μs
STENCIL(ST)	Parboil[25]	block2D_hybrid _coarsen	4.5μs	41.3μs	4.2μs
STREAM CLUSTER(SC)	Parboil[25]	kernel_ compute_cost	6.7μs	605.6μs	8.3μs
GEMM(GM)	Darknet[24]	matrixMulCUDA	23.4μs	193.6μs	17.9μs
BLACK SCHOLES(BS)	Nvidia SDK[19]	BlackScholesGPU	3.4μs	387.5μs	16.7μs
KMEANS(KS)	Rodinia[7]	invert_mapping	29.7μs	984.7μs	9μs
PATHFINDER(PF)	Rodinia[7]	dynproc_kernel	11.3μs	24.2μs	11.6μs
SRAD_V1(SRAD1)	Rodinia[7]	extract	5.2μs	1.8μs	4μs
SRAD_V2(SRAD2)	Rodinia[7]	srad_cuda	15μs	11.5μs	16.4μs
SRAD_V1(SRAD3)	Rodinia[7]	srad	5.2μs	7.9μs	7.8μs
HOTSPOT(HS)	Rodinia[7]	calculate_temp	33.3μs	4.5μs	7.7μs
LUD	Rodinia[7]	lud_internal	4.4μs	5.3μs	10.5μs
BACKPROP(BP)	Rodinia[7]	bpnn_ layerforward	16.7μs	4.7μs	2μs
BACKPROP(BP2)	Rodinia[7]	bpnn_adjust _weights	16.7μs	1.5μs	1.2μs

Table 1: Benchmarks Time Comparison

least 15μs, assuming the bandwidth is fully utilized. Also, existing preemption techniques are reactive, meaning that a wait time of $\sim 15\mu$s or higher (with memory interference) is needed. We propose a proactive technique to shorten the effective wait time so that the incoming kernel execute sooner.

3 DESIGN

We introduce PEP, a proactive and efficient preemption mechanism via checkpointing. Checkpointing is widely used in fault tolerance to save the state of a running process periodically. We use a lightweight mechanism to save a base checkpoint of the context before preemption. When the actual preemption is invoked, we perform an incremental update relative to the base checkpoint. The kernel only needs to wait for the completion of the latter update before execution, effectively reducing the latency overhead of the preempting kernel. We develop prediction and estimation techniques to limit the number of checkpoints to 1 or 2 in order to control the potential performance degradation due to too frequent context savings. Further, we also reduce the context size by saving only dirty data, and perform in-place savings whenever possible [12]. Finally, for short kernels, we still preserve the capability of draining, which has no overhead. However, we do not leverage flushing in this paper. On one hand, it requires re-execution, which often incurs a larger overhead than context switching; on the other hand, most of TBs are able to be flushed only at the very beginning of its execution.

3.1 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, enabling us to save less. We track the active context using dirty bits. GPU registers are allocated for each thread, and threads are executed in a warp group. Thus, a register's lifetime is lifetime of its associated warp. When a warp is done, 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 write-back stage, and we unset it if we checkpoint and whenever the warp finishes. Therefore, every time we switch context, only registers with dirty bits set are saved. We do not apply a liveness register saving technique [12], since it requires setting fixed preemption points to avoid the high overhead.

We also leverage in-place context saving [12] at the incremental checkpoint. It can further reduces the actual preemption time, as less context need to be saved to global memory.

3.2 Prediction and Estimation

The prediction of kernel launch time and estimation of draining and switching time are key components of PEP. TB runtime selection in Section 3.3 is based on them.

Prediction In order not to wastefully checkpoint, we must be able to predict when preemption occurs. Since the cudaLaunch call triggers the kernel launch action, this is the time from when cudaLaunch is called to the time that kernel information arrives to the KMU without queuing in the stream manager. We observed from Table 1, for the set of applications we examined, the length of context switch time can be similar to the kernel launch time, approximately. Thus, if we start checkpointing at the moment cudaLaunch is called, then preemption probably has occurred by the time the first base checkpoint finishes. In this case, we release resources immediately, making room for new kernels. Our design does not require precise predictions of kernel launch time. This is because if the checkpoint finishes before the actual preemption request arrives, the SM can just continue executing the checkpointed TB. Once preemption actually starts, we then save a much smaller dirty context.

We also find that the average TB execution time varies a lot depending on the length of the kernel. Short kernels' TBs will drain relatively fast. Only long TBs will be more likely to context switch. As these TBs' execution times may be hundreds of microseconds, it is easy 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 20μs, for prediction purposes. 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 the predicted launch time is smaller than the remaining TB execution time, then we start checkpointing right away. Otherwise, we drain the TB. Note that the variation of the kernel launch time is relatively small compared with TB execution time. As shown in Figure 3, PEP can handle all 5 possibilities, which we will clarify in Section 3.3. As shown, even if the true kernel launch time is away from 20μs, it is unlikely to cause a different preemption decision. In the worst case, the wrong estimation causes a base checkpoint to occur too early, causing a misprediction of TB choice. In this case, the incremental checkpoint can save all of the context of the correct TB.

Estimation Once a preemption is predicted to happen, we must make a decision regarding currently executing TBs. If the remaining execution time of the TBs is shorter than the time to perform a base checkpoint (e.g. for short TBs), then we drain. Otherwise, we perform a checkpointing (context saving). Hence, we need to estimate the draining time of current TBs and the latency of checkpointing. Chimera[21] uses time estimation to compare the throughput overheads between different preemption methods. They estimate the drain time 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, this TB-based estimation is inaccurate. For context switching, the context of TBs in SMs are transferred serially; their estimation only considers a single TB. Thus, their switch time estimation is much shorter than actual switch times. Furthermore, some applications like LBM and KS have multiple phases; their CPI is time-varying. In these cases, their drain time estimation may be far off. Also, they do not consider the bandwidth competition between switching TBs and draining TBs.

From Table 1, we observe a wide gap between TB execution time and TB switch time. In most applications, we choose to drain all TBs or switch all TBs. Hence, the drain time and context switch time can be estimated without interference. We also profile previously executed TBs to obtain average TB execution time, and use this to estimate a TBs remaining execution time. However, if there is no previous profile available, we use Chimera's estimation. To estimate context switching time, we assume the worst case, where all the TBs in an SM switch. As the context switch time has a small range of variation compared to the execution time, we are safe in using the worst case estimate.

3.3 Proactive Preemption Design

Checkpoint Saving Checkpointing is only used for long running preempted kernels, since 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 [13]. We estimate that this latency is less than 3μs, which is the time for launching an empty kernel. 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.

When the checkpointing is done, all the dirty bits are cleared. Then, the GPU checks whether the new kernel is transferred to the KMU or not. If it is in the pending kernel pool, it starts 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 continues executing until the actual preemption request arrives. When the actual preemption request arrives, we only need to save a much smaller incremental update to the base checkpoint, which takes much less time. Restoration of the preempted kernel is similar to conventional checkpointing. If we have two checkpoints' states to restore, we must restore one by one.

Runtime Selection As we know from the Table 1, execution time, context size, and launch time can vary among kernels. Hence, when cudaLaunch triggers our proactive preemption mechanism, there are several possibilities, as shown in Figure 3.

(a) Two checkpoints: The kernel launch time is longer than the base checkpoint saving time. When the actual preemption starts, we save an incremental checkpoint.

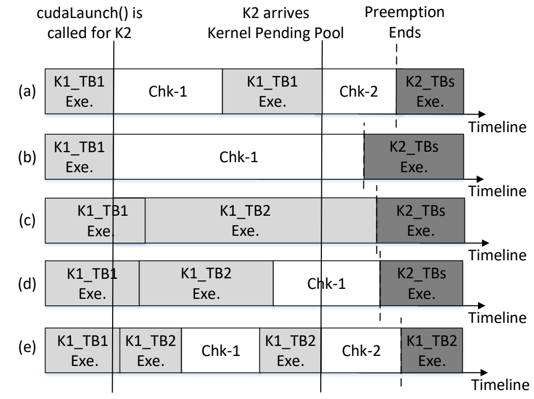

Figure 3: PEP Possibilities (K1: the preempted kernel, K2: the preempting kernel, Chk-1: the base checkpoint, Chk-2: the incremental checkpoint)

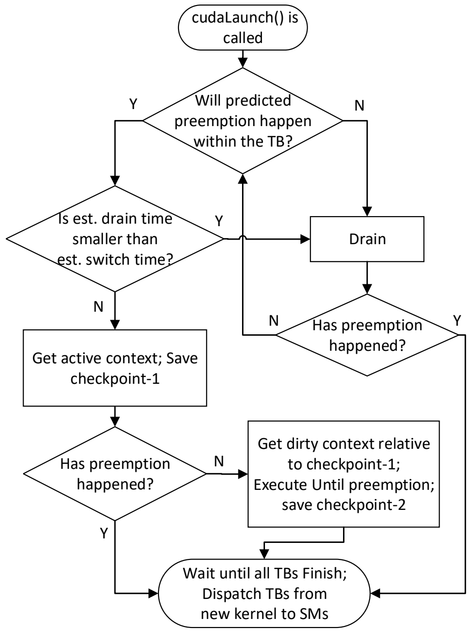
cudaLaunch() is called
Will predicted preemption happen within the TB?
Y
N
Is est. drain time smaller than est. switch time?
Y
Drain
N
Has preemption happened?
N
Y
Get active context; Save checkpoint-1
Has preemption happened?
N
Get dirty context relative to checkpoint-1; Execute Until preemption; save checkpoint-2
Y
Wait until all TBs Finish; Dispatch TBs from new kernel to SMs

Figure 4: Design Flow

(b) Single checkpoint: This is the same as a traditional context switch, but it starts earlier.
(c) Drain: The TB execution time is shorter than preempting kernel launch time,
(d) Drain then single checkpoint: The preempted kernel is the same as in case (b), but the current TB is almost finished. The new TB will start execution for a fixed number of instructions before saving the checkpoint.
(e) Drain then two checkpoints: The preempted kernel is the same as in case (a). The current TB is almost finished when cudaLaunch is called.

We design a runtime selection mechanism shown in Figure 4to handle all possibilities. When cudaLaunch is called for the preempting kernel, we compare the predicted kernel launch time with the estimated remaining execution time of current TB. If the predicted

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

Table 2: GPGPU-Sim Configuration Parameters

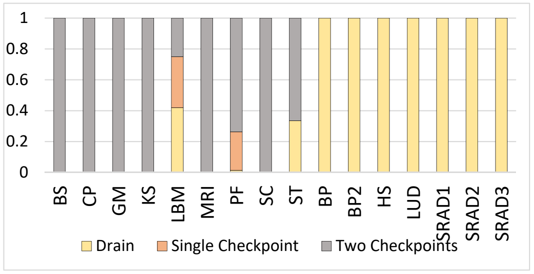

Figure 5: Preemption Technique Distribution

preemption will occur within the TB, proactive preemption starts. If the TB's drain time is longer than the switch time, we define this kernel as long. Case (a) and (b) operate for 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 current TB's lifetime, so the choice is to drain them. After that, we will do another round of prediction and estimation. If the new TB is estimated to drain sooner than being checkpointed, we have case (c). Otherwise, we are in case (d) or case (e), which is similar to case (b) or case (a) again.

4 EXPERIMENTS

We implement PEP on the latest version of GPGPU-Sim [4]. The system configuration is summarized in Table 2. For comparison, we implement Chimera and PEP with different variations: vanilla Chimera, Chimera with saving dirty context only, vanilla PEP and PEP with the in-place context saving. We test using a wide range of kernels from Nvidia Computing SDK [19], Parboil [25], Rodinia [7] and Darknet [24]. For Chimera, we observed that the average context switch time is always smaller than 20.9μs, so we test a sequence of deadlines of 5μs, 10μs and 15μs. For PEP, the actual kernel launch time may vary from kernel to kernel, so we vary this parameter from 5μs, 15μs, 25μs to 35μs. However, our prediction algorithm only uses a fixed time to predict for simplicity, so we perform sensitivity study from 20μs to 30μs for this time, as seen from most applications. Further, we also assume different progress, 25%, 50% and 75%, of TBs when the preempting kernel is launched to cover more general cases. The results shown below are averages over a particular parameter. We then compare the preemption latency, context size and preemption overhead of these different designs. In Table 1, the average launch times are collected from the Nvidia Visual Profiler [18], while the other times are collected from the GPGPU-Sim and then calibrated to the real time.

4.1 Selection Distribution

In Figure 5, we show the breakdown of runtime selection for all TBs of each application. TBs incurring two checkpoints are from

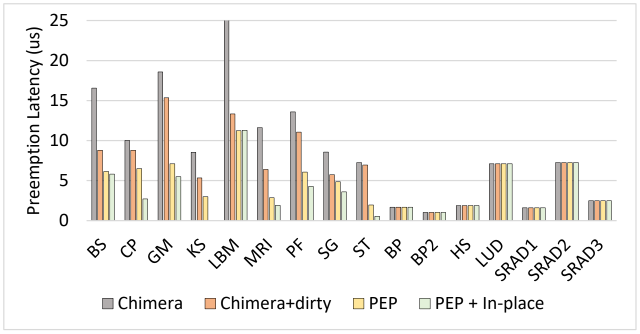

Figure 6: Average Preemption Time

the six applications with average TB execution time longer than 100μs (See Table 1). TBs incurring only draining are from short kernels with short average TB execution time. For LBM, the average TB execution time is 30.1μs, comparable to its average switch time which is 20.9μs. This closeness is what allows for the variation of choices between draining and checkpointing, depending on the progress parameter. Similar observation is made for PF as well.

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.

4.2 Preemption Latency

Figure 6 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. For the first nine kernels, since the dominant TB actions is checkpointing, the context size is the key to the preemption latency. Compared with Chimera, Chimera+Dirty reduces the preemption latency by 31.8%, as it only saves dirty data. Both versions of PEP are more effective than Chimera+Dirty. Compared with Chimera, PEP, and PEP+In-place reduces the average preemption latency by 58.5% and 70.3% respectively. In particular, for KS, PEP+in-place has zero preemption latency because the dirty context for incremental checkpoint is small enough to be saved on-chip completely.

For the last seven kernels which drain all TBs per SM achieve low latencies because they have short TB execution times. With our proactive approach, on average, PEP and PEP+In-place reduce the total preemption latency from 8.9μs in Chimera to 4.5μs and 3.6μs respectively. A shorter preemption latency allows kernels to meet a stricter deadline, increasing its effectiveness for multitasking.

4.3 Context Size Reduction

Figure 7 compares the context size among different designs. Saving only dirty context, Chimera+Dirty 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, its average total context size might be larger than Chimera+dirty. However, the amount is almost the same as original Chimera. With in-place saving, PEP can further reduce the context size by 16.2%.

Figure 8 shows the breakdown of context for PEP+in-place. For applications that choose to context switch all the TBs, the total

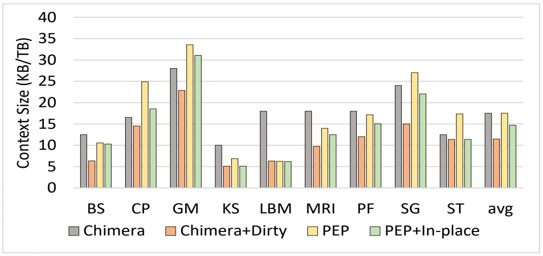

Figure 7: Context Size Comparison

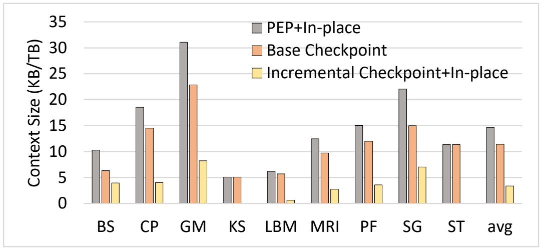

Figure 8: Context Size with In-place per TB

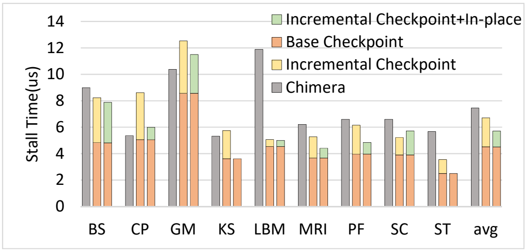

Figure 9: Context Saving Overhead(Chimera+dirty is similar to the Base Checkpoint)

context size is the sum of the base checkpoint and the incremental checkpoint. The result 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. This is a critical size as it largely determines the wait time of a preempting kernel.

4.4 Impact of Preemption and HW Overhead

The overhead for preemption is the idle time of execution units caused by preemption. When the SM is switching context, switching TBs are stopped from fetching instructions and executing. SMs are idle for both context swap out and context restore. The only difference between swapping 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 for the preempted kernel.

Figure 9 shows that the base checkpoint reduces the average overhead by 37.9% from Chimera. This overhead is similar to Chimera+dirty. With checkpointing, the overhead of our PEP is still 6.3% lower than Chimera. The overhead can be further reduced by 16.4% with the in-place saving. Some applications with high register reuse rate do have higher overhead compared with Chimera. However, the context size and the context switch overhead are positively correlated. Some applications such as LBM and ST save more than 50% of overhead, due to a small incremental context sizes.

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]; (3) profiler counters, which are used for collecting TB execution times. Overall, the majority of overhead is largely in the dirty bit storage.

5 CONCLUSION

In this paper, we proposed PEP, a 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. We also support SM draining for short kernels. 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 reduce to 3.6μs, which allows for stricter deadlines, thus increasing multitasking support.

REFERENCES

[1] Jacob T Adriaens and et al. 2012. The case for GPGPU spatial multitasking. In *HPCA'12*. IEEE, 1–12.
[2] AMD. 2013. *AMD GRAPHIC CORE NEXT*. http://developer.amd.com/wordpress/media/2013/06/2620_final.pdf
[3] Joshua A Anderson and et al. 2008. General purpose molecular dynamics simulations fully implemented on graphics processing units. *J. Comput. Phys.* 227, 10 (2008), 5342–5359.
[4] Ali Bakhoda and et al. 2009. Analyzing CUDA workloads using a detailed GPU simulator. In *ISPASS'09*. IEEE, 163–174.
[5] Can Basaran and et al. 2012. Supporting preemptive task executions and memory copies in gpgpus. In *Real-Time Systems, 2012 24th Euromicro Conference on*. IEEE, 287–296.
[6] Jon Calhoun and et al. 2012. Preemption of a cuda kernel function. In *SNPD'12*. IEEE, 247–252.
[7] Shuai Che and et al. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *IISWC'09*. Ieee.
[8] Guoyang Chen and et al. 2017. EffiSha: A software framework for enabling effficient preemptive scheduling of GPU. In *PPoPP'17*. ACM, 3–16.
[9] HSA foundation. 2016. HSA Platform System Architecture Specification 1.1. *Specification* (Jan 2016).
[10] Shinpei Kato and et al. 2011. Operating systems challenges for GPU resource management. In *OSPERT'11*. 23–32.
[11] Chuanpeng Li and et al. 2007. Quantifying the cost of context switch. In *Proceedings of the 2007 workshop on Experimental computer science*. ACM, 2.
[12] Zhen Lin and et al. 2016. Enabling efficient preemption for simt architectures with lightweight context switching. In *SC'16*. IEEE, 898–908.
[13] Konstantinos Menychtas, Kai Shen, and Michael L Scott. 2014. Disengaged scheduling for fair, protected access to fast computational accelerators. *ASPLOS'14* 42, 1 (2014), 301–316.
[14] Nvidia. 2011. Nvidia cuda c programming guide. 120, 18 (2011), 8.
[15] Nvidia. 2012. Nvidias next generation cuda compute architecture: Kepler gk110. *Whitepaper* (2012).
[16] Nvidia. 2014. NVIDIA Geforce GTX980 Whitepaper. *Whitepaper* (2014).
[17] Nvidia. 2016. Nvidia Tesla P100 Whitepaper. *Whitepaper* (April 2016).
[18] NVIDIA. 2017. *Profiler User's Guide*. http://docs.nvidia.com/cuda/profiler-users-guide/
[19] NVIDIA SDK. [n. d.]. https://developer.nvidia.com/cuda-toolkit. ([n. d.]).
[20] Sreepathi Pai and et al. 2013. Improving GPGPU concurrency with elastic kernels. In *ASPLOS'13*, Vol. 48. ACM, 407–418.
[21] Jason Jong Kyu Park, Yongjun Park, and Scott Mahlke. 2015. Chimera: Collaborative Preemption for Multitasking on a Shared GPU. In *ASPLOS'15*. ACM, New York, NY, USA, 593–606.
[22] Steven G Parker and et al. 2010. Optix: a general purpose ray tracing engine. In *ACM Transactions on Graphics*, Vol. 29. ACM, 66.
[23] Victor Podlozhnyuk. 2007. Black-Scholes option pricing. (2007).
[24] Joseph Redmon and et al. 2016. YOLO9000: Better, Faster, Stronger. (2016).
[25] John A Stratton and et al. 2012. Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing.
[26] Ivan Tanasic and et al. 2014. Enabling preemptive multiprogramming on GPUs. In *ISCA'14*, Vol. 42. IEEE Press, 193–204.
[27] Zhenning Wang and et al. 2016. Simultaneous multikernel GPU: Multi-tasking throughput processors via fine-grained sharing. In *HPCA'16*. IEEE, 358–369.