2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS) | 978-1-6654-4066-0/20/$31.00 ©2021 IEEE | DOI: 10.1109/IPDPS49936.2021.00021

CTXBack: Enabling Low Latency GPU Context Switching via Context Flashback

Zhuoran Ji
Department of Computer Science
The University of Hong Kong
Hong Kong, China
zrji2@cs.hku.hk

Cho-Li Wang
Department of Computer Science
The University of Hong Kong
Hong Kong, China
clwang@cs.hku.hk

***Abstract*—Efficient GPU preemption mechanisms are critical for task prioritization in multitasking environments, especially for latency-sensitive GPU applications. However, due to the large context of GPU kernels, simply borrowing context switching mechanisms from CPU space incurs substantial latency and overhead. To address this problem, we propose *CTXBack*, which allows a thread block to execute context switching at a preceding instruction with a smaller context. It enables low latency GPU context switching for latency-sensitive applications on shared GPUs. Three complementary ways are proposed to make more preceding instructions into valid points to execute context switching, giving *CTXBack* a higher chance to find instructions with smaller contexts. Evaluations show that *CTXBack* reduces the context by 61.0%, which is only $1.09\times$ of the minimum possible context size. With only 0.41% runtime overhead, the preemption latency and resuming time are reduced by 63.1% and 50.0% on average compared to the traditional approach.**

***Index Terms*—GPU, GPU sharing, GPU context switching**

I. Introduction

Graphics processing units (GPUs) and other single-instruction multiple-thread (SIMT) processors have been widely used for massively data-parallel applications. As throughput-oriented accelerators, GPUs feature large register files to accommodate massively concurrent threads [1]. The large register files result in a long preemption latency for preemptive GPU context switching. However, many emerging GPU applications need Quality of Service (QoS) guarantee. Edge computing platforms, such as mobile phones and autopilot cars, become to integrate GPUs and other SIMT processors [2]. They deal with latency-sensitive tasks and need low-latency preemption for prioritization. Moreover, cloud service providers have been providing GPU services for massively parallel applications [3], which consist of both latency-sensitive jobs and batch jobs. The multi-user and multitasking environment requires efficient low latency preemption mechanisms for GPU sharing.

There have been several attempts to reduce the GPU context switching cost. Prior studies propose to defer context switching and continue execution until reaching an instruction with a small register context [4]. It significantly reduces the preemption overhead, but the instructions executed before context switching increase the preemption latency. These instructions can contain costly device memory access, which incurs a long waiting time for the incoming latency-sensitive jobs. Other studies propose many novel checkpoint-based GPU fault tolerance mechanisms [5], [6], which can be adapted for GPU context switching. However, saving contexts as checkpoints during normal execution incurs constant runtime overhead. Some techniques, such as checkpoint pruning [6], can reduce the overhead, but it forces the preempted thread blocks to restart from a very early checkpoint. The long resuming time is not a problem for rarely occurred errors but can incur considerable overhead for context switching.

In this paper, we propose context-flashback (*CTXBack*) to enable low latency GPU context switching for latency-sensitive applications on shared GPUs. *CTXBack* allows a thread block to execute context switching at a preceding instruction (named flashback-point) of the instruction where the preemption occurs. It reduces the preemption latency, as the context of the flashback-point should be much smaller, and the context switching is performed without any deferral. Even if *CTXBack* needs to re-execute the in-between instructions during resuming, the overall preemption overhead is still reduced. The context size reduction achieves less device memory access, which outweighs the overhead of re-executing several instructions on GPUs. *CTXBack* does not need to save checkpoints, which avoids complex trade-off decisions on the resuming cost and the runtime overhead incurred by checkpoints. No matter where the preemption occurs, the context switching is executed at a nearby preceding instruction with a smaller context. Only a few instructions need to be re-executed during resuming so that the preemption overhead is reasonable for GPU context switching.

Executing context switching at an instruction needs its context to remain available when preemption occurs. However, some registers of the preceding instructions' register contexts may have been overwritten along with the execution. *CTXBack* identifies the flashback-points of an instruction with the use-define chains analyzed from the assembly code. The key to *CTXBack*'s performance is the number of flashback-points of each instruction, as more flashback-points give a higher chance to find instructions with smaller contexts. *CTXBack* adopts three complementary methods to make more preceding instructions into flashback-points. First, *CTXBack* combines saving/reloading with re-execution when restoring an instruc-

978-1-6654-4066-0/21/$31.00 ©2021 IEEE
DOI 10.1109/IPDPS49936.2021.00021

tion's context from its flashback-point. It relaxes the necessity of the whole register context of the flashback-point, as not all in-between instructions need to be re-executed. Second, based on the observation that many instructions are reversible, we propose instruction reverting, which can recover the previous values of the registers that have been overwritten. Third, we notice that many instructions are not re-executable due to unavailable scalar operands, which forces *CTXBack* to choose a worse flashback-point and swap more vector registers. The size of a scalar register is negligible compared with vector registers. Thus, we propose to back up several scalar registers into unused on-chip resources proactively, which reduces the preemption latency with negligible runtime overhead.

This work makes the following major contributions:

1) We present *CTXBack*, which allows a thread block to execute context switching at a preceding instruction. *CTXBack* enables low latency GPU context switching with reasonable preemption overhead.
2) We proposed three novel methods to increase the number of flashback-points, including relaxed flashback-point condition, instruction reverting, and on-chip scalar register backup. These methods help *CTXBack* find more flashback-points with smaller contexts.
3) We illustrate how *CTXBack* is used for GPU preemption, which consists of online and offline parts. We show that *CTXBack* incurs negligible extra work to the preemption handler.

We evaluate *CTXBack* on an AMD Radeon VII GPU [1] with various benchmarks. With our three novel techniques, the average context size of flashback-points is only 9.2% more than the minimum possible size, which is 61.0% less than the baseline. With 0.41% runtime overhead, *CTXBack* reduces the preemption latency and resuming time by 63.1% and 50.0% on average compared with the traditional approach.

II. Background and Motivation

This section motives *CTXBack* by discussing the cost of the traditional preemptive GPU context switching approach and also the prior GPU preemption techniques. NVIDIA's terminology is used throughout the paper.

A. Preemptive GPU Context Switching

GPUs provide high throughput for massively data-parallel applications via high degrees of thread-level parallelism. To accommodate massively concurrent threads, GPUs feature large amounts of on-chip resources. The vector registers, which store variables belonging to each thread, are the most critical resources for the SIMT programming model. For example, in the AMD Vega architecture [1], each streaming multiprocessor (SM) has 256 KB vector registers, 12.5 KB scalar registers, and 64 KB shared memory. GPUs from other vendors, such as NVIDIA, feature similar amounts of on-chip resources.

For preemptive GPU context switching, the on-chip states (context) of the preempted thread blocks are saved into device memory. After that, their on-chip resources can be released and assigned to other kernels. As the context is preserved, the preempted thread blocks can restore it from the device memory and resume execution at a later point. Due to the large context and the slow device memory, saving and restoring the context cause long preemption latency and high preemption overhead. The preemption latency refers to the elapsed time before releasing the GPU, which determines the waiting time. The preemption overhead is the additional work induced by preemption and resuming, which impacts the system throughput. The GPU context switching mechanism in Linux, which swaps all occupied on-chip resources of the preempted thread blocks regardless of the register liveness, shows $165\mu s$ preemption latency on average and up to $327\mu s$, while the resuming time is $135\mu s$ on average and up to $283\mu s$ (Table I in section V). Many studies report that the execution time of typical latency-sensitive GPU applications is about $1 \sim 2ms$ [7], [8]. MLPerf [9] also shows that the inference time is in the order of hundred microseconds for small neural networks and milliseconds for large neural networks. The long preemption latency of the traditional GPU context switching mechanism causes a considerable waiting time for these latency-sensitive jobs, and the high overhead also degrades the throughput.

B. Prior GPU Preemption Techniques

To avoid the preemption overhead, SM-draining [10] has been proposed, which stops to dispatch new thread blocks when receiving the preemption signal. When the currently running thread blocks have finished, the resources can be assigned to another kernel. On the other hand, SM-flushing [11] is proposed to address the long preemption latency, which drops the running thread blocks immediately if the idempotent condition holds. The preempted thread blocks are restarted from the beginning during resuming. These two methods either defer the preemption to the end or restart from the beginning of the kernel, which is too coarse-grained, especially for batch jobs. The kernels of batch jobs usually adopt the persistent thread programming model [12] and loop unrolling to improve resource utilization and hide memory latency. As a side effect, a single thread block's execution time is prolonged. When latency-sensitive jobs preempt thread blocks of batch jobs, SM-draining and SM-flushing may either cause long preemption latency for the latency-sensitive jobs or high overhead for the batch jobs. For most of the execution time, context switching is still needed, as it provides reasonable preemption latency and overhead no matter when preemption occurs.

Some studies [4] notice the variety of different instructions' context size and propose to temporarily ignore the context switching signal and continue the execution until reaching an instruction with a small context. This method (denoted as *CS-Defer*) reduces the preemption overhead significantly by reducing the context size. However, the preemption latency counts the execution time of the instructions executed before context switching, which may contain costly device memory access. *CS-Defer* incurs relatively long and undetermined

preemption latency, which is not friendly for systems with QoS requirements.

Many checkpoint-based GPU fault tolerance mechanisms are proposed [5], [6], which can be adapted for GPU context switching. Saving contexts as checkpoints involves device memory access, which can significantly hurt the kernel performance. Many techniques, such as checkpoint pruning [6] and sparse idempotent regions [5], are proposed to reduce the runtime overhead without compromising the recoverability guarantee. However, it results in a long checkpoint interval, making the preempted thread blocks restart from a very early checkpoint. These fault tolerance mechanisms rarely add checkpoints in the inner loops, while the long execution time of a thread block mainly results from loops. The resuming time can be as long as SM-flushing if a thread block restarts from a checkpoint out of the loop. For the rarely occurred errors (1/day in 16nm [6]), trading the recovery cost for low runtime overhead benefits the performance. However, the long resuming time can incur considerable overhead for GPU context switching, which is much more frequent. There is a natural trade-off between the resuming time and the runtime overhead when using checkpoint-based GPU fault tolerance mechanisms for context switching. Excessive checkpoints incur high runtime overhead, while deficient checkpoints lead to high resuming cost.

It motives *CTXBack*, a mechanism specially designed for GPU context switching, which provides low preemption latency with reasonable resuming costs.

III. Context Flashback

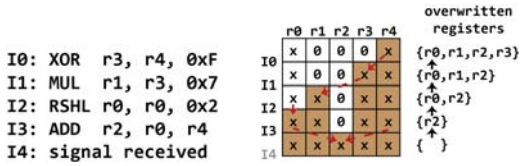
This section introduces the basic idea of *CTXBack* and then describes the three techniques, which make more preceding instructions into flashback-points.

To facilitate the following discussion, we denote a preceding instruction as I_{prec}, the current instruction where the preemption signal is received as I_{cur}, and the instructions between I_{prec} and I_{cur} as $[I_{prec}, I_{cur})$. We assume the preemption signal is processed before executing each instruction. The register contexts of I_{prec} and I_{cur} are denoted as $\mathcal{R}_{prec}$ and $\mathcal{R}_{cur}$, respectively.

A. Basic Idea

Executing context switching at I_{prec} needs $\mathcal{R}_{prec}$ to be available at the time the preemption signal is received. When the execution progress reaches I_{cur}, $[I_{prec}, I_{cur})$ may have overwritten some registers of $\mathcal{R}_{prec}$. A preceding instruction's register context is available at I_{cur} if no registers of $\mathcal{R}_{prec}$ have been overwritten by $[I_{prec}, I_{cur})$. On the other hand, if some registers of $\mathcal{R}_{prec}$ have been overwritten, *CTXBack* cannot select this preceding instruction as a flashback-point without further mechanisms.

An instruction's register context is just its live-in registers. *CTXBack* first uses liveness analysis to determine each preceding instruction's live-in registers. For the example code in Figure 1 (a), each instruction's live-in registers are marked with "x" in Figure 1 (b), if we assume $\{r0-r4\}$ are live-in registers of I4. *CTXBack* then analyzes the use-define chains to figure out each register is overwritten by which instruction. The dashed arrows in Figure 1 (b) show the use-define chains of the example assembly code. To find out whether their register contexts are available at I4, *CTXBack* iterates the instructions from I3 to I0 backwardly. For each iteration, it invalidates the registers that have been overwritten by the previously iterated instructions. The registers, whose values are still available at I4 (I_{cur}), are highlighted in Figure 1 (b). In this example, all live-in registers of I3 and I4 are available at I4. In contrast, r0 is a live-in register of I0, I1, and I2, which has been overwritten by I2. I0, I1, and I2 have unavailable live-in registers when execution progress reaching I4. Without applying further mechanisms, they are not flashback-points of I4.



(a) Example code (b) Register context

Fig. 1: Determine register context availability of preceding instructions

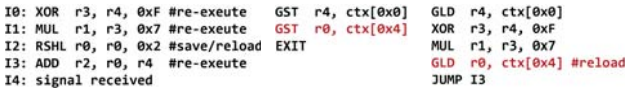
B. Relaxing Flashback-Point Condition

To properly resume the execution, $\mathcal{R}_{cur}$ should be restored. Traditional approaches restore $\mathcal{R}_{cur}$ from I_{prec} by re-executing $[I_{prec}, I_{cur})$, which needs the whole live-in registers of I_{prec} (i.e., $\mathcal{R}_{prec}$). This is the reason why $\mathcal{R}_{prec}$ needs to be available for I_{prec} to be a flashback-point. However, the goal is to restore $\mathcal{R}_{cur}$ rather than $\mathcal{R}_{prec}$. A preceding instruction should be a flashback-point as long as $\mathcal{R}_{cur}$ can be restored from it.

When preemption signal is received at I_{cur}, $[I_{prec}, I_{cur})$ have already been executed. For each instruction of $[I_{prec}, I_{cur})$, its results are still stored in the physical registers, unless the registers have already been overwritten. During context saving, an instructions results can be saved into device memory together with the context. Then, this instruction does not need to be re-executed when restoring $\mathcal{R}_{cur}$ from I_{prec}. When reaching this instruction, the resuming routine loads its results from the device memory rather than re-executing it.

CTXBack relaxes the flashback-point condition based on the above two observations. It first proves that it is sufficient to restore $\mathcal{R}_{cur}$, provided that the results of all instructions of $[I_{prec}, I_{cur})$ are restorable, and then combines re-execution and saving/reloading when restoring an instruction's result registers.

Figure 2 (a) shows the same example as Figure 1 (a). I2 is not re-executable due to the missing operand r0, which has been overwritten by itself. However, the result of I2 can also be restored by saving/reloading. As the result



(a) Example code (b) Preemption (c) Resuming

Fig. 2: Combining re-executing and saving/reloading

registers of {I0−I3} can all be restored, either by re-execution or saving/reloading, {I0−I3} are all flashback-points. Restoring a result register by saving/reloading costs two device memory accesses, which is usually far more expensive than re-executing an instruction. *CTXBack* prefers re-execution to saving/reloading if both are feasible. When preemption signal is received at I4, *CTXBack* only needs to save r0 and r4 (Figure 2 (b)). During resuming, I0, I1, and I3 are re-executed, while the result of I2 is restored via saving/reloading (Figure 2 (c)).

Algorithm 1 illustrates how *CTXBack* finds flashback-points with the relaxed condition. It is a two-pass algorithm that starts by iterating the instructions from the current instruction I_{cur} to I_{head} and then the second pass iterates them forwardly. The backward pass (*lines* $3-6$) finds out which instructions can be restored by saving/reloading, which is similar to identifying the availability of the preceding instructions' register contexts. For each iterated instruction, it checks whether its result registers are in the *invalid_reg*. *CTXBack* only considers the result registers belonging to this instruction's live-out registers, which do need to be restored. Then, the result registers are inserted into *invalid_reg*, as this instruction has overwritten these registers, which becomes unavailable to the following iterated instructions.

Algorithm 1 Check Relaxed Flashback-Points Condition

```
1:  I_head = first possible preceding instruction
2:  invalid_reg = ∅
3:  for inst from I_cur to I_head do
4:      if inst.res ∩ invalid_reg == ∅ then
5:          mark inst.res as available
6:      invalid_reg = invalid_reg ∪ inst.res
7:  for inst from I_head to I_cur do
8:      if inst.operand ∩ invalid_reg == ∅ then
9:          mark inst as re-executable
10:     if inst is re-executable or inst.res is available then
11:         invalid_reg = invalid_reg \ inst.res
12:     else
13:         invalid_reg = invalid_reg ∪ inst.res
```

After that, the second pass figures out whether each instruction is re-executable. An instruction is re-executable if none of its operands are in the *invalid_reg*. A register is not in *invalid_reg* if it has not been overwritten or can be restored by re-execution. No matter the result registers can be restored by re-execution or saving/reloading, their values becomes valid for the following iterated instructions. Thus, the register results are erased from *invalid_reg* (*lines* $10-13$) if so.

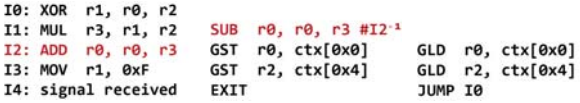
The remaining of this subsection proves that it is sufficient to restore $\mathcal{R}_{cur}$, provided that the results of all instructions of $[I_{prec}, I_{cur})$ (denoted as $\mathcal{R}_{[I_{prec},I_{cur})}$) are restorable. When execution progress reaches I_{cur}, the registers of $\mathcal{R}_{prec}$ that have not been overwritten are $\mathcal{R}_{prec} \setminus \mathcal{R}_{[I_{prec},I_{cur})}$. The union of $\mathcal{R}_{prec} \setminus \mathcal{R}_{[I_{prec},I_{cur})}$ and $\mathcal{R}_{[I_{prec},I_{cur})}$ is a superset of $\mathcal{R}_{cur}$.

$$
\begin{aligned}
&(\mathcal{R}_{prec} \setminus \mathcal{R}_{[I_{prec},I_{cur})}) \cup \mathcal{R}_{[I_{prec},I_{cur})} \\
=&(\mathcal{R}_{prec} \cup \mathcal{R}_{[I_{prec},I_{cur})}) \setminus (\mathcal{R}_{[I_{prec},I_{cur})} \setminus \mathcal{R}_{[I_{prec},I_{cur})}) \\
=&(\mathcal{R}_{prec} \cup \mathcal{R}_{[I_{prec},I_{cur})}) \setminus \emptyset \supseteq \mathcal{R}_{cur}
\end{aligned}
$$

The relaxed flashback-point condition has no direct requirement for $\mathcal{R}_{prec}$. The availability of $\mathcal{R}_{prec}$ determines whether an instruction between I_{prec} and I_{cur} is re-executable. It then affects whether $\mathcal{R}_{cur}$ can be restored from I_{prec}.

C. Instruction Reverting

Many instructions use the same physical register as both the operand register and result register. For the example code in Figure 3 (a), I1 uses r0 as both the operand register and result register. These instructions have the form $r_{share} = op(r_{share}, \{R\})$, where r_{share} is the shared physical register, $\{R\}$ are the other operands, and *op* is the operator. The r_{share} on the left side is named as r'_{share} for a clear presentation, and it becomes $r'_{share} = op(r_{share}, \{R\})$. Executing this instruction overwrites r_{share}. Its preceding instructions, which use r_{share} as an operand, become not re-executable.



(a) Example code (b) Preemption (c) Resuming

Fig. 3: Instruction reverting

It is noticed that many of these kinds of instructions can be reverted so that the overwritten operands can be recovered. An instruction is reversible if there exists an operator op^{-1} such that $r_{share} = op^{-1}(r'_{share}, \{R\})$. The preceding instructions that use r_{share} as an operand, are then re-executable, if r_{share} is the only missing operand.

For the example code in Figure 3 (a), I0 can be restored neither by re-execution nor saving/reloading when preemption occurs at I4. However, the previous value of r0 can be recovered with the reverting instruction of I2 (i.e., SUB r0, r0, r3). It makes I0 re-executable, which further makes I1 re-executable. Thus, *CTXBack* only needs to save r2 and the recovered r0 into device memory (Figure 3 (b)). With the extra cost of executing the reverting instructions, more preceding instructions become re-executable. It not only makes more preceding instructions into flashback-points but also reduces the preemption cost for the existing flashback-points, as restoring the results of an instruction by saving/reloading is usually much more expensive than re-execution.

Algorithm 2 illustrates how to determine the reversibility of an instruction. To revert an instruction, the operator itself

Algorithm 2 Find Reverting Strategy

```
1: if inst.op is not reversible then
2:     return irreversible
3: cand = register to be recovered
4: operands_for_revert = inst.res ∪ (inst.operand − {cand})
5: for reg in operands_for_revert do
6:     last_def = last definition of reg
7:     if reg has not been overwritten then
8:         continue
9:     else if last_def is reversible then
10:        if last_def must be reverted during resuming then
11:            must_revert_at_resume = true
12:    else if last_def is re-executable then
13:        must_revert_at_resume = true
14:    else
15:        return irreversible                ▷ missing operand
16: if must_revert_at_resume then
17:    revert_pos = at_resume
18: else
19:    revert_pos = MIN_COST(at_resume, at_preempt)
20: construct the reverting instruction for inst
21: return reversible
```

needs to be reversible. Many heavily used operators are reversible, such as addition and left shifting used in memory address calculation. Reverting an instruction also needs its result registers and operand registers (except the operand to be recovered) to be available. A register is available if it

1) has not been overwritten (*lines* $7-8$),
2) can be recovered by instruction reverting (*lines* $9-11$),
3) can be restored by re-execution (*lines* $12-13$).

If one of the registers can only be recovered during resuming (*line* 10) or restored via re-execution (*line* 12), this instruction must also be reverted during resuming. As shown in Figure 4 (a), reverting `I2` needs `r2`, which can only be restored by re-execution. *CTXBack* executes the reverting instruction after the last operand has been restored during resuming (Figure 4 (c)). On the other hand, the algorithm compares the cost of executing the reverting instruction at different stages. The reverting instruction is executed at the preemption stage if it reduces the context size. Otherwise, *CTXBack* executes the reverting instruction during resuming to prevent it from contributing to the preemption latency.

```
I0: MUL  r2, r1, 0xE                                  GLD  r0, ctx[0x0]
I1: XOR  r3, r0, r2                                   GLD  r1, ctx[0x4]
I2: ADD  r0, r0, r2      GST  r0, ctx[0x0]            MUL  r2, r1, 0xE #I0
I3: MOV  r2, 0xFF        GST  r1, ctx[0x4]            SUB  r0, r0, r2  #I2-1
I4: signal received      EXIT                         JUMP I1
```

(a) Example code (b) Preemption (c) Resuming

Fig. 4: Instruction reverted during resuming

D. On-Chip Scalar Register Backup

For most GPUs, there are two kinds of registers: scalar registers and vector registers. While each thread has a private copy for each vector register, the threads within a warp share the same scalar register. A scalar register only occupies 4 bytes for each warp. However, an overwritten scalar operand can also make many instructions not re-executable. *CTXBack* then needs to restore their result registers, which can be vector registers, by saving/reloading. For the example code in Figure 5 (a), `I0` is not re-executable due to `S4` (scalar register). It forces *CTXBack* to swaps `V7` (vector register), which increases the context size. Even worse, it may prevent some preceding instructions, which have smaller register contexts, from being flashback-points.

```
I0: ADD  V7, V2, S4
I1:  OR  S4, S4, 0x2
I2: MUL  V11, V11, V2
I3: SUB  V12, V11, V7
I4: signal received
```

(a) Original code

```
I0: ADD  V7, V2, S4
    MOV  S11, S4 #OSRB
I1:  OR  S4, S4, 0x2
I2: MUL  V11, V11, V2
I3: SUB  V12, V11, V7
I4: other insts ...
```

(b) Code with *OSRB*

Fig. 5: On-chip scalar register backup

To address this problem, we propose a mechanism named on-chip scalar register backup (*OSRB*). It proactively copies the scalar registers, which lead to worse flashback strategies for some instructions, into unused on-chip resources. As shown in Figure 5 (b), *OSRB* copies `S4` into `S11` during normal execution, as `S4` leads to worse flashback strategies for `I4`. `S11` is an unused register, which usually comes from two sources. Firstly, many modern GPUs need aligned register allocation. The extra scalar registers and vector registers caused by alignment are always unused. Moreover, the occupied registers are not always live. The registers, which are dead within the backup period, can also be used to store the backup.

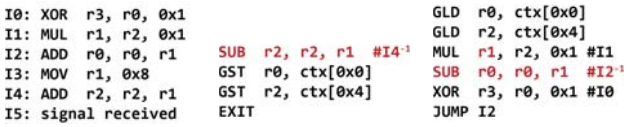
The register copying instructions are executed regardless of whether the preemption occurs, much like the conventional checkpoint mechanisms. However, *OSRB* incurs negligible runtime overhead, even if performed in inner loops. Firstly, *OSRB* only backs up the scalar registers that do provide better flashback-points for some instructions. In reality, *OSRB* mainly backs up the iteration induction variable and the execution mask. Only these scalar registers are frequently updated and critical for better flashback-points. Moreover, the scalar-scalar and scalar-vector register copying instructions only cost one or several cycles and do not occupy device memory bandwidth.

When finding places to store the backup, the scalar registers are used first, and then the vector registers. A warp has *warp-size* copies of a vector register, and each copy can store a scalar register. A single unused vector register is enough for most cases. *CTXBack* can also store the backup into unused shared memory if there is no unused register. However, shared memory is the last choice as it is slower than registers and has unpredictable latency.

E. Finding Flashback-Points

The three proposed methods are complementary to each other. A register recovered by instruction reverting may make some instructions reversible or re-executable. The result registers of these newly re-executable instructions may further

make more instructions re-executable or reversible. As shown in Figure 6, reverting `I4` recovers the previous value of `r2`, which makes `I1` re-executable. Re-executing `I1` restores the value of `r1`, which then makes `I2` reversible. The recovered `r0` further makes `I0` re-executable.



(a) Example code (b) Preemption (c) Resuming

Fig. 6: A more complex example of instruction reverting

To handle the above cases and avoid iterating the in-between instructions multiple times, we augment the forward pass of Algorithm 1 (*lines* $7-13$). For each iterated instruction, it checks whether this instruction is re-executable and/or reversible. If not, this instruction is inserted into a hash map, whose key is the unavailable register. In the following iterations, a register may be restored by re-execution or recovered by instruction reverting. The algorithm searches the instructions with the newly available register from the hash map and checks whether these instructions have become re-executable or reversible. If so, these instructions are eliminated from the hash map, and the procedure is repeated recursively until no more registers become available. After the forward pass, it tries to use *OSRB* to further eliminate the remaining instructions recursively.

Each instruction is inserted into and eliminated from the hash map at most twice. Each instruction is searched at most C times before being eliminated, where C is the maximum number of operands of an instruction. In general, C is a small number, which can be regarded as a constant. Thus, the augmented algorithm's time complexity is still $\mathcal{O}(N)$, where N is the number of instructions between I_{head} and I_{cur}.

The flashback-points are limited to be selected within the basic block, as the control flow between I_{cur} and the flashback-points needs to be statically determinable. A basic block is a sequence of straight-line instructions that have no internal branches. The control flow within a basic block is always determined. This limitation has a minor impact on *CTXBack*. The basic blocks of GPU kernels are usually large because of their simple control logic. The number of instructions within a basic block is sufficient for finding a good enough flashback-point. Meanwhile, *CTXBack* can only select flashback-points within the idempotent region. An idempotent region is a sequence of instructions that have the same effect when executed once or many times [13]. Preemption mechanisms, which need to re-execute some instructions, usually need the idempotent condition to be satisfied [5], [11].

IV. *CTXBack* for Preemption

A. Compilation Time

CTXBack is implemented as an additional compiler pass, which can be invoked after the assembly code emission pass or on a standalone basis. It identifies the flashback-points and selects the one with the least preemption latency for each instruction. After that, it generates a dedicated preemption routine and a dedicated resuming routine for each instruction, which swap the context of its selected flashback-point. The dedicated routines also perform the instruction reverting and the required re-execution.

All dedicated preemption routines are transferred to the device memory with the kernel code, while only the necessary dedicated resuming routines are transferred on-demand during resuming. When preemption occurs, the host side does not know the program counters of the preempted warps unless it queries the device. It involves costly CPU-GPU communication, and the preempted warps need to be suspended during the communication. Thus, *CTXBack* transfers the dedicated preemption routines of all instructions regardless of whether the preemption occurs.

CTXBack shares the preemption routines among different instructions to reduce the transferring and storing cost. In reality, the selected flashback-points of many instructions are the same preceding instruction, whose context size is local minima. For the instructions with the same flashback-point, their dedicated preemption routines are the same, except the instruction reverting part. The reverting instructions are executed before saving the context, and the instruction reverting part of an instruction's preemption routine is always a subset of that of its succeeding instructions. These instructions can share one dedicated preemption routine, and *CTXBack* only needs to store an offset for each of them. In general, only several preemption routines need to be transferred and stored, whose cost is negligible.

B. Runtime

If no preemption occurs, *CTXBack* incurs no extra work, except on-chip scalar register backup. It has almost no impact on the performance of the kernels.

When preemption occurs, the preempted warps enter a general preemption routine. It does the same setup as the conventional GPU context switching routine, such as saving the program counter (PC) into temporary registers. After that, the general preemption routine uses the saved PC to query the corresponding dedicated preemption routine. As different warps usually have different PCs, the returned dedicated routines may be different for different warps. Each warp then jumps to its dedicated preemption routine to perform the actual context saving.

C. Combining CTXBack and CS-Defer

Even if *CS-Defer* [4] focuses on preemption overhead, it sometimes may have shorter preemption latency than *CTXBack*. It happens when the preemption occurs at an instruction, whose next few succeeding instructions have small contexts. Moreover, *CTXBack* and *CS-Defer* have different trade-offs between preemption latency and preemption overhead. Combining them can meet the different requirements of latency and throughput. Thus, we integrate *CS-Defer* into

our work, which is named as *CTXBack+CS-Defer*. If *CS-Defer* provides shorter preemption latency for an instruction, its dedicated routines are generated by *CS-Defer*.

V. Evaluation

In this section, we provide a detailed evaluation of *CTXBack* on an AMD Radeon VII graphics card. While the experiments are conducted on this GPU, *CTXBack* does not rely on its unique features and can also be applied to other GPUs. As a software solution, *CTXBack* only needs an assembler to compile the dedicated context switching routines, which is available on both AMD GPUs [1] and NVIDIA GPUs [17]. *CTXBack* is implemented as a code generation pass of LLVM [18], a compiler framework that supports *AMDGPU* backend.

We select a wide range of OpenCL kernels from different areas as benchmarks, including BLAS (Basic Linear Algebra Subprograms) libraries [15], deep learning libraries [14], and the traditional Rodinia [16] benchmarks. While OpenCL kernels are selected, *CTXBack* can also be applied to assembly codes compiled from other high-level programming languages. The kernels are compiled with `-O3` optimization option. We list the resource usage per warp, the context saving time (preemption time), and the context restoring time (resuming time) in Table I. The resource usage usually determines the number of active warps of an SM. It counts the extra resource usage caused by alignment. For AMD Radeon VII, the vector registers and scalar registers are allocated in groups of 4 registers and 16 registers, respectively [1]. The context saving time and restoring time are measured with the context switching routine in the AMD GPU driver, which is available in Linux source code. It swaps all occupied on-chip resources of the preempted thread blocks regardless of the register liveness. It is noticed that the context switching time is not strictly proportional to the occupied resources. The context switching routines are highly memory-intensive, whose execution time is affected by the bandwidth usage of other thread blocks (i.e., thread blocks which are not preempted). Moreover, the resuming time is usually shorter than the preemption time because of better memory latency hiding.

We evaluate six preemption techniques. *BASELINE* refers to the GPU context switching mechanism in Linux. *LIVE* [4] is to use liveness information to exclude dead registers from the context, while *CKPT* [5] adapts the checkpoint-based GPU fault tolerance techniques for GPU context switching. In our experiments, *CTXBack*, *CS-Defer*, and *CTXBack+CS-Defer* rank the candidates with the preemption latency. The performance is evaluated in terms of context size, preemption time, and resuming time. The context size reduction is analyzed statically, while the others are measured with running kernels. Results are normalized with the baseline, whose absolute context switching time is reported in Table I.

A. Context Reduction

Figure 7 shows the size of the context that needs to be swapped during context switching, which is normalized with the baseline. *CKPT* has saved the context into device memory during normal execution. The checkpoint size is reported, which is indicated by dash lines. For *CTXBack*, the context includes the result registers restored by saving/reloading.

By excluding dead registers, *LIVE* reduces the context size by 37.8% on average. The context is further reduced by utilizing the variety of the register context size. On average, *CTXBack* and *CS-Defer* reduce the context by 61.03% and 62.07% compared with the baseline, while the combination of them (i.e., *CTXBack+CS-Defer*) reduces it by 62.09%. If only considering the kernels from BLAS and deep learning libraries, *CTXBack* reduces the context by 68.8%. *CKPT* can always save the context of the instructions with the least live registers (minimum possible size). With the three proposed complementary methods, the context size of the flashback-points found by *CTXBack* is only 9.2% more than *CKPT*'s.

The context reduction depends on the context composition and the variety of the register liveness. For HS, the shared memory accounts for more than 65% of the occupied resources. None of the evaluated approaches achieve satisfactory reduction as shared memory's liveness cannot always be determined by compilers. However, Yang et al. [19] observe the short lifetimes of shared memory and propose to explicitly de-allocate the shared memory to improve the performance. It can be used to avoid swapping shared memory if shared memory is not live. Some studies [20] also propose to use warp-level primitives to replace the use of shared memory, which further reduces its usage. On the other hand, the register pressure of RELU and VA is low. Compilers are more flexible in scheduling the instructions to improve the instruction-level parallelism (ILP). It results in a rapid and drastic variety of the number of live registers, and thus a significant context reduction. For VA, *CTXBack* reduces the context size by 78.2%, and *CTXBack+CS-Defer* reduces the context size by 80.9% compared with the baseline.

B. Preemption Time

Figure 8 shows the normalized execution time of the preemption routines. The preemption routines spend most of their execution time saving the context (except *CS-Defer* and *CKPT*). The ratios of the preemption time of different approaches are about the same as the ratios of their context size. Compared with the baseline, *CTXBack* reduces the preemption time by 63.1% on average. As *CS-Defer* needs to execute the in-between instructions in the preemption routine, its preemption latency is 34.8% longer than *CTXBack* on average. If only considering the kernels from BLAS and deep learning libraries, *CS-Defer* suffers 44.2% longer preemption latency than *CTXBack*. *CTXBack* and *CS-Defer* can collaborate to further reduce the preemption latency. On average, *CTXBack+CS-Defer* reduces the preemption latency by 65.2% compared with the baseline.

CTXBack+CS-Defer selects between *CTXBack* and *CS-Defer* based on their estimated preemption latency, while its preemption latency is longer than the smaller one of *CTXBack* and *CS-Defer* for some benchmarks. It is caused by the underestimation of *CS-Defer*'s preemption latency. GPUs are

TABLE I: Benchmark Specification

Benchmark Name	Vector Regs	Scalar Regs	Shared Mem	Preempt Time	Resume Time	Abbreviation
Average Pooling	7.0 KB	0.188 KB	0.0 KB	103.4 μs	87.1 μs	AP [14]
Direct Convolution	8.0 KB	0.141 KB	0.0 KB	153.0 μs	114.2 μs	DC [14]
Dot Product	6.0 KB	0.141 KB	1.0 KB	138.6 μs	101.0 μs	DOT [14], [15]
Gaussian Elimination	8.0 KB	0.141 KB	0.0 KB	92.3 μs	74.0 μs	GE [16]
Hybrid Sort	7.0 KB	0.141 KB	12.0 KB	304.0 μs	280.7 μs	HS [16]
K-Means	13.0 KB	0.141 KB	0.0 KB	327.4 μs	283.1 μs	KM [16]
Local Response Norm	4.0 KB	0.141 KB	0.0 KB	74.9 μs	57.8 μs	LRN [14]
Matrix-Matrix Multiple	13.0 KB	0.141 KB	0.5 KB	214.6 μs	152.7 μs	MM [14], [15]
Merge Sort	10.5 KB	0.141 KB	0.0 KB	119.0 μs	93.8 μs	MS [16]
Matrix-Vector Multiply	13.0 KB	0.141 KB	0.25 KB	254.7 μs	217.5 μs	MV [14], [15]
ReLU Activation	4.0 KB	0.141 KB	0.0 KB	93.8 μs	75.5 μs	RELU [14]
Vector Addition	3.0 KB	0.141 KB	0.0 KB	102.2 μs	81.1 μs	VA [14], [15]

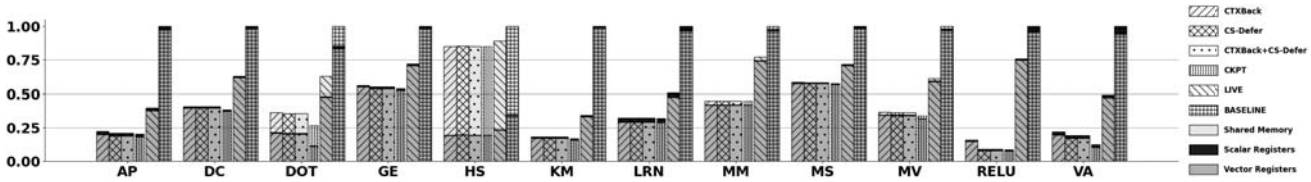


Fig. 7: Normalized context size

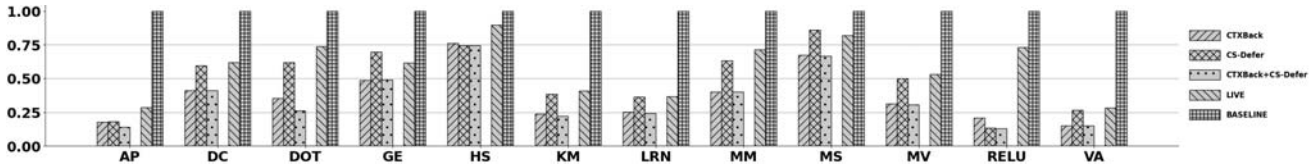


Fig. 8: Normalized execution time of the preemption routines

highly pipelined processors. After issuing an instruction, a warp can continue to execute the next instruction until reaching an unsolved register dependency. It is the instructions with unsolved register dependencies, rather than the long-latency instructions themselves, that suffer long latency. Most GPUs do not guarantee the execution order of the warps. Estimating the potential latency induced by the preceding instructions is hard without timestamps. Thus, when estimating the preemption latency of *CS-Defer*, the potential latency induced by the preceding instructions is not considered. *CS-Defer*'s preemption latency may be underestimated, which may lead *CTXBack+CS-Defer* to choose the sub-optimal preemption mechanism for some instructions. However, compared with the element-wise minimum of *CTXBack*'s and *CS-Defer*'s preemption latency, the preemption latency increased due to underestimation is negligible.

C. Resuming Time

In addition to restoring the context, the resuming time also counts the re-execution time, if any. The re-execution time may occupy a large portion of the resuming time, which can even dominate the resuming time if too much useful work is wasted. As shown in Figure 9, the average resuming time reduction of *CTXBack* is 50.0% compared with the baseline. Even though *CTXBack* use preemption latency as the criterion to rank the flashback-points in our experiments, the average resuming time of *CTXBack* is still shorter than *LIVE*'s for all benchmarks, except KM. On average, *CS-Defer* reduces the resuming time by 65.6% compared with the baseline, which is 31.2% less than that of *CTXBack*. The average resuming time reduction of *CTXBack+CS-Defer* is 54.8%. In addition to preemption latency reduction, combining *CTXBack* and *CS-Defer* also significantly reduces the resuming time.

As opposite approaches, the difference of *CTXBack*'s and *CS-Defer*'s preemption time is about the same as the difference of their resuming time. The unrolling factor and instruction scheduling policy influence this difference. Large unrolling factors and ILP-oriented instruction scheduling policy usually result in a more significant live register variety. Both of them increase this difference, with which the relative performance of *CTXBack+CS-Defer* will be even higher.

On the other hand, *CKPT* wastes considerable execution progress, leading to a long resuming time for some benchmarks. The resuming time of *CKPT* is influenced by the checkpoint interval, which also influences its runtime overhead (Figure 10). Thus, the resuming time and runtime overhead need to be analyzed together. In our experiment, the check-

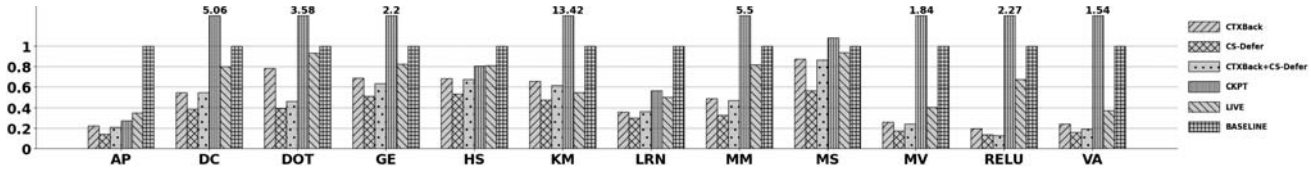


Fig. 9: Normalized execution time of the resuming routines

point interval is every 16 times executing the same basic block. The experiments show that the selected checkpoint frequency is deficient for some kernels but is excessive for others.

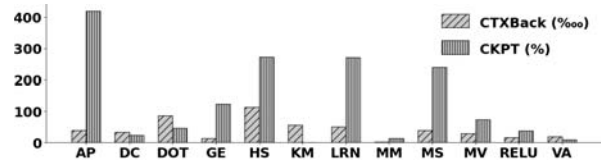


Fig. 10: Runtime overhead

The resuming time of *CKPT* is 318% of the baseline, and the corresponding runtime overhead is 130% on average. Increasing the checkpoint interval can decrease the runtime overhead, but the resuming time will increase correspondingly. Figure 10 also shows the runtime overhead induced by *CTXBack*, or more specifically, induced by on-chip scalar register backup. The runtime overhead only contains data movement among registers. On average, the runtime overhead of *CTXBack* is 0.41% for all benchmarks and 0.35% for kernels from BLAS and deep learning libraries.

In terms of resuming time and runtime overhead, *CTXBack* performs much better than *CKPT*. With the cost of 9.2% less context reduction, the resuming time of *CTXBack* is only 15.7% of that of *CKPT*. *CKPT* does not perform well for kernels without a significant variety of live registers. For these kernels, the context size reduction is not significant. Even worse, the checkpoint size is relatively large compared with the occupied resources, which incurs more runtime overhead. *CTXBack* also favors kernels with a large variety of live registers. Unlike *CKPT*, *CTXBack* decays to *LIVE* when dealing with kernels without a significant variety of live registers. It rarely results in a longer resuming time than *LIVE*'s nor incurs nontrivial runtime overhead. However, *CKPT* also supports fault tolerance and speculative execution, while *CTXBack* is specially designed for GPU context switching.

VI. Related Work

For CPUs, reducing the context switching latency and overhead has also been researched. Some studies [21] propose to exclude dead registers from the context for fast context switching. Both the throughput and response time are improved, especially for programs with explicitly yield control. Another approach [22] is to classify the context into different levels and partitions. The hot contexts are allowed to reside on processors. GPUs differ from CPUs in both software and hardware. Each vector register has *number-of-concurrent-threads* copies. It is worth utilizing more sophisticated approaches to reduce the registers from GPU contexts.

Prior studies have proposed many novel mechanisms to reduce the GPU preemption overhead. RGEM [23], PKM [24], and GPES [25] attempt to slice the memory-copy transactions and thread grids into many smaller ones. Preemption can happen at their boundaries. SM-draining [10] stops to dispatch new thread blocks when receiving the preemption signal. EffiSha [26] and FLEP [27] enable software draining by proactively checking the preemption with a while loop. As these mechanisms focus on preemption overhead, they may suffer a long preemption latency. Lin et al. [4] have proposed several novel ways to enable lightweight GPU context switching, and one of them is *CS-Defer*. It allows a thread block to defer to any instruction and can usually have a shorter preemption latency than SM-draining. However, the deferring time is undeterminable, which can be relatively long for some latency-sensitive applications, especially when the in-between instructions involve device memory access.

Many studies have paid attention to preemption latency. Park et al. [11] propose SM-flushing, which instantly preempts an SM by flushing the running thread blocks, provided the relaxed idempotent condition holds. It has almost no preemption latency nor runtime overhead, but the flushed thread blocks' work is wasted. CLPKM [28] provides a software solution for dropping running kernels. It logs whether a thread has finished or not and ignores the finished threads during resuming. They have almost zero preemption latency but may incur significant preemption overhead. *CTXBack* is much more fine-grained, which allows each instruction to have a customized flashback-point. It only wastes the work of the in-between instructions and thus incurs less preemption overhead in general.

To reduce the preemption overhead caused by dropping thread blocks, PEP [29] proactively saves the context into device memory during normal execution. It predicts whether preemption may happen for early context saving. Penny [6] and iGPU [5] are proposed to enable fault tolerance and speculative execution on GPUs by checkpoints, which can be adapted for GPU context switching. These approaches have a natural trade-off between the resuming time and runtime overhead, which is not a problem for fault tolerance but critical to GPU context switching. *CTXBack* is specially designed for GPU context switching, which enables low latency preemption with reasonable preemption overhead.

Park et al. [11] recognize the different trade-offs of SM-flushing, traditional context switching, and SM-draining. To balance the preemption latency and overhead, they propose

Chimera, which selects the preemption technique based on the thread block's execution progress. Many studies [7], [30] also propose to allow multiple kernels to share the GPU via spatially partitioning the resources. In addition to higher resource utilization, they can also be used for prioritization. *CTXBack* is orthogonal to these studies. It can be integrated into Chimera to replace the traditional context switching mechanism. Spatial sharing also needs temporal sharing mechanisms for better prioritization in each partition, and *CTXBack* is a candidate.

VII. Conclusion

In this paper, we presented *CTXBack*, a mechanism that enables a thread block to execute context switching at a preceding instruction. *CTXBack* is specifically designed for GPU context switching, which enables low latency GPU context switching with reasonable overhead. It adopts three novel approaches: (1) relaxing flashback-point condition, (2) instruction reverting, and (3) on-chip scalar register backup, to find better flashback-points. *CTXBack* reduces the context size by 61.0%, which is only $1.09\times$ of the minimum possible size. Correspondingly, the preemption time and resuming time is reduced by 63.1% and 50.0% on average compared with the baseline. As a checkpoint-free mechanism, *CTXBack* breaks the trade-off between the resuming time and runtime overhead. The resuming time and runtime overhead of *CTXBack* are only 15.7% and 0.33% of *CKPT*'s.

Acknowledgment

This research is supported by Hong Kong RGC Research Impact Fund R5060-19. We appreciate IPDPS reviewers for their constructive comments and suggestions.

References

[1] AMD, "Vega instruction set architecture reference guide," July 2017. [Online]. Available: https://developer.amd.com/wp-content/resources/Vega_Shader_ISA_28July2017.pdf

[2] D. Franklin, "Nvidia jetson tx2 delivers twice the intelligence to the edge," *NVIDIA Accelerated ComputingParallel Forall*, 2017.

[3] S. Iserte, F. J. Clemente-Castelló, A. Castelló, R. Mayo, and E. S. Quintana-Ortí, "Enabling gpu virtualization in cloud environments." in *CLOSER (2)*, 2016, pp. 249–256.

[4] Z. Lin, L. Nyland, and H. Zhou, "Enabling efficient preemption for simt architectures with lightweight context switching," in *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2016, pp. 898–908.

[5] J. Menon, M. De Kruijf, and K. Sankaralingam, "igpu: exception support and speculative execution on gpus," *ACM SIGARCH Computer Architecture News*, vol. 40, no. 3, pp. 72–83, 2012.

[6] H. Kim, J. Zeng, Q. Liu, M. Abdel-Majeed, J. Lee, and C. Jung, "Compiler-directed soft error resilience for lightweight gpu register file protection," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 989–1004.

[7] C. Yu, Y. Bai, H. Yang, K. Cheng, Y. Gu, Z. Luan, and D. Qian, "Smguard: A flexible and fine-grained resource management framework for gpus," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 12, pp. 2849–2862, 2018.

[8] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa, "Timegraph: Gpu scheduling for real-time multi-tasking environments," in *Proc. USENIX ATC*, 2011, pp. 17–30.

[9] V. J. Reddi, C. Cheng, D. Kanter, P. Mattson, G. Schmuelling, C.-J. Wu, B. Anderson, M. Breughe, M. Charlebois, W. Chou *et al.*, "Mlperf inference benchmark," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 446–459.

[10] I. Tanasic, I. Gelado, J. Cabezas, A. Ramirez, N. Navarro, and M. Valero, "Enabling preemptive multiprogramming on gpus," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3, pp. 193–204, 2014.

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

[12] K. Gupta, J. A. Stuart, and J. D. Owens, *A study of persistent threads style GPU programming for GPGPU workloads*. IEEE, 2012.

[13] M. A. De Kruijf, K. Sankaralingam, and S. Jha, "Static analysis and compiler design for idempotent processing," in *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, 2012, pp. 475–486.

[14] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *Proceedings of the 22nd ACM international conference on Multimedia*, 2014, pp. 675–678.

[15] C. Nugteren, "Clblast: A tuned opencl blas library," in *Proceedings of the International Workshop on OpenCL*, 2018, pp. 1–10.

[16] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE international symposium on workload characterization (IISWC)*. Ieee, 2009, pp. 44–54.

[17] D. Yan, W. Wang, and X. Chu, "Optimizing batched winograd convolution on gpus," in *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2020, pp. 32–44.

[18] C. A. Lattner, "Llvm: An infrastructure for multi-stage optimization," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 2002.

[19] Y. Yang, P. Xiang, M. Mantor, N. Rubin, and H. Zhou, "Shared memory multiplexing: a novel way to improve gpgpu throughput," in *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 2012, pp. 283–292.

[20] S. G. De Gonzalo, S. Huang, J. Gómez-Luna, S. Hammond, O. Mutlu, and W.-m. Hwu, "Automatic generation of warp-level primitives and atomic instructions for fast and portable parallel reduction on gpus," in *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2019, pp. 73–84.

[21] S. Dolan, S. Muralidharan, and D. Gregg, "Compiler support for lightweight context switching," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 9, no. 4, pp. 1–25, 2013.

[22] X. Zhou and P. Petrov, "Rapid and low-cost context-switch through embedded processor customization for real-time and control applications," in *2006 43rd ACM/IEEE Design Automation Conference*. IEEE, 2006, pp. 352–357.

[23] S. Kato, K. Lakshmanan, A. Kumar, M. Kelkar, Y. Ishikawa, and R. Rajkumar, "Rgem: A responsive gpgpu execution model for runtime engines," in *2011 IEEE 32nd Real-Time Systems Symposium*. IEEE, 2011, pp. 57–66.

[24] C. Basaran and K.-D. Kang, "Supporting preemptive task executions and memory copies in gpgpus," in *2012 24th Euromicro Conference on Real-Time Systems*. IEEE, 2012, pp. 287–296.

[25] H. Zhou, G. Tong, and C. Liu, "Gpes: A preemptive execution system for gpgpu computing," in *21st IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE, 2015, pp. 87–97.

[26] G. Chen, Y. Zhao, X. Shen, and H. Zhou, "Effisha: A software framework for enabling efficient preemptive scheduling of gpu," in *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2017, pp. 3–16.

[27] B. Wu, X. Liu, X. Zhou, and C. Jiang, "Flep: Enabling flexible and efficient preemption on gpus," *ACM SIGPLAN Notices*, vol. 52, no. 4, pp. 483–496, 2017.

[28] M.-T. Chiu and Y.-P. You, "Clpkm: A checkpoint-based preemptive multitasking framework for opencl kernels," *Journal of Systems Architecture*, vol. 98, pp. 53–62, 2019.

[29] C. Li, A. Zigerelli, J. Yang, Y. Zhang, S. Ma, and Y. Guo, "A dynamic and proactive gpu preemption mechanism using checkpointing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 1, pp. 75–87, 2018.

[30] W. Zhang, W. Cui, K. Fu, Q. Chen, D. E. Mawhirter, B. Wu, C. Li, and M. Guo, "Laius: Towards latency awareness and improved utilization of spatial multitasking accelerators in datacenters," in *Proceedings of the ACM International Conference on Supercomputing*, 2019, pp. 58–68.