Idempotence-Based Preemptive GPU Kernel Scheduling for Embedded Systems

Hyeonsu Lee, Hyunjun Kim, Cheolgi Kim, Hwansoo Han, and Euiseong Seo

Abstract—Mission-critical embedded systems simultaneously run multiple graphics-processing-unit (GPU) computing tasks with different criticality and timeliness requirements. Considerable research effort has been dedicated to supporting the preemptive priority scheduling of GPU kernels. However, hardware-supported preemption leads to lengthy scheduling delays and complicated designs, and most software approaches depend on the voluntary yielding of GPU resources from restructured kernels. We propose a preemptive GPU kernel scheduling scheme that harnesses the idempotence property of kernels. The proposed scheme distinguishes idempotent kernels through static source code analysis. If a kernel is not idempotent, then GPU kernels are transactionized at the operating system (OS) level. Both idempotent and transactionized kernels can be aborted at any point during their execution and rolled back to their initial state for reexecution. Therefore, low-priority kernel instances can be preempted for high-priority kernel instances and reexecuted after the GPU becomes available again. Our evaluation using the Rodinia benchmark suite showed that the proposed approach limits the preemption delay to 18 μs in the 99.9th percentile, with an average delay in execution time of less than 10 percent for high-priority tasks under a heavy load in most cases.

Index Terms—GPGPU, preemptive scheduling, real-time systems, embedded systems, transaction, idempotence

1 Introduction

GPU computing has been successfully adopted in high-performance computing (HPC) systems for scientific computing and enterprise server systems for stream-data processing. The effectiveness of GPU computing has also been verified in mobile and embedded systems. Especially, owing to its superiority in vision and speech recognition, GPU computing is becoming a core component in emerging advanced control systems, including autonomous driving systems and unmanned aerial vehicle systems [1], [2].

Because embedded systems that control such machinery utilize multiple input sources and provide diverse functions, they concurrently execute multiple independent tasks with different criticality and timeliness requirements, which translate to scheduling priorities. Therefore, to guarantee bounded scheduling delays and to prevent low-priority kernels from blocking high-priority kernels, an embedded OS must provide a preemptive priority scheduler for scheduling its tasks.

In GPU computing, both context saving and restoration of GPU kernels require significant time and memory, due to the massive parallelism of GPU internals. Because of this constraint, hardware-supported kernel preemption complicates GPU design. Consequently, only high-end GPUs currently provide a limited level of preemption support, with a large context-switching overhead [3], [4], [5]. For the same reasons, GPUs targeting embedded systems, which have strict constraints in terms of form factors, manufacturing costs, and energy efficiency, do not support preemptive scheduling.

Several software-based approaches have been proposed to enable preemptive GPU kernel scheduling [6], [7]. Existing approaches commonly insert scheduling points into the GPU kernels to limit scheduling delays. These approaches increase the number of scheduling activities and kernel launches, which in turn cause significant performance degradations. They also require restructuring and recompiling of the existing GPU kernel sources. Despite these disadvantages, their scheduling latency can be prolonged to the lengths of the sub-kernels, which usually span a few hundred microseconds.

The kernel abort command, which is available in most GPUs, does not save the kernel context and thus finishes within a time interval of a few μs. An idempotent kernel is one that does not corrupt its input data, producing the same results every time it is executed. Therefore, when every kernel is guaranteed to be idempotent, it is possible to schedule a high-priority kernel with a scheduling delay of a few μs by aborting a currently running low-priority kernel. This is because the preempted low-priority kernel can be simply reexecuted when the GPU becomes available again.

To realize this approach, we propose a GPU kernel idempotence determination scheme and a kernel transactionization technique. Based on these two schemes, we finally present a preemptive priority GPU kernel scheduling scheme that provides an extremely short scheduling latency.

The kernel idempotence classifier statically analyzes the kernel source code to detect any operations that break

- *Hyeonsu Lee, Hyunjun Kim, Hwansoo Han, and Euiseong Seo are with the Department of Computer Science and Engineering, Sungkyunkwan University, Gyeonggi-do 10540, Republic of Korea. E-mail: {hyunsu, hjunkim, hhan, euiseong}@skku.edu.*
- *Cheolgi Kim is with the Department of Software and Computer Engineering, Korea Aerospace University, Goyang-si, Gyeonggi-do 16419, Republic of Korea. E-mail: cheolgi@kau.ac.kr.*

Manuscript received 5 Sept. 2019; revised 26 Mar. 2020; accepted 11 Apr. 2020. Date of publication 20 Apr. 2020; date of current version 10 Feb. 2021. (Corresponding author: Euiseong Seo.)
Recommended for acceptance by S. Chakraborty.
Digital Object Identifier no. 10.1109/TC.2020.2988251

idempotence. If a kernel is identified as idempotent, then the classifier provides a scheduling hint to the scheduler so that it will be scheduled without any further treatment. However, when a kernel is not idempotent or its source code is unavailable, the scheduler applies the transactionization scheme to the kernel.

The GPU kernel transactionization technique, which was proposed in our previous paper [8], exploits the property that both CPUs and GPUs observe the same physical memory in the heterogeneous system architecture (HSA), which is widely employed for embedded systems. In this scheme, the OS creates a snapshot of the GPU memory before executing a GPU kernel instance. If the kernel execution is aborted, then this snapshot will be used to roll back the GPU memory to its initial state.

Together, these two schemes enable GPU kernels to be forcibly evicted at any time during their execution and reexecuted after the GPU becomes available. Based on these techniques, we propose a preemptive priority scheduling mechanism that immediately evicts a low-priority kernel and schedules a high-priority kernel when a high-priority kernel is ready to be launched. The preempted low-priority kernel will then be relaunched after rollback from its snapshot.

The proposed scheme is not suitable for GPU time-sharing, because a preempted kernel must be reexecuted from the start. However, it enables immediate scheduling of important and urgent kernels, which are spontaneously invoked to react to exceptional conditions. A representative example is the scheduling of an evasion-path-finding task in an autonomous driving system, which is spontaneously invoked when the periodically-running obstacle detection task detects an incoming object.

Under the proposed schemes, preemption occurs instantaneously. The proposed scheduling mechanism is applicable to any hardware that supports the abortion of GPU kernel execution. It can be applied for third-party applications that do not open their source codes, and it even guarantees the scheduling of priority kernels in the presence of malicious applications.

To evaluate the proposed schemes, we implemented them in the Linux kernel for the Samsung Exynos 5422 system on chip (SoC) with a Mali-T628 GPU. We evaluated the proposed schemes using a well-known GPU benchmark suite.

The remainder of this paper is organized as follows. Section 2 introduces the background and related work. Section 3 proposes our approach. Subsequently, Section 4 evaluates the proposed schemes, and finally, Section 5 concludes the paper.

2 Background and Related Work

2.1 HSA and OpenCL Framework

The OpenCL framework enables applications to run on various kinds of processing units, such as CPUs and GPUs [9]. It is one of the most popular general-purpose portable programming frameworks and is supported by most GPU vendors.

HSA unifies various computing devices with different characteristics into a single system. The integrated GPU architecture, which consolidates the CPU and GPU into a single chip, is a representative example of HSA. This is advantageous for embedded systems that have strict limitations in terms of manufacturing costs, form-factors, thermal dissipation, and energy efficiency [10], [11].

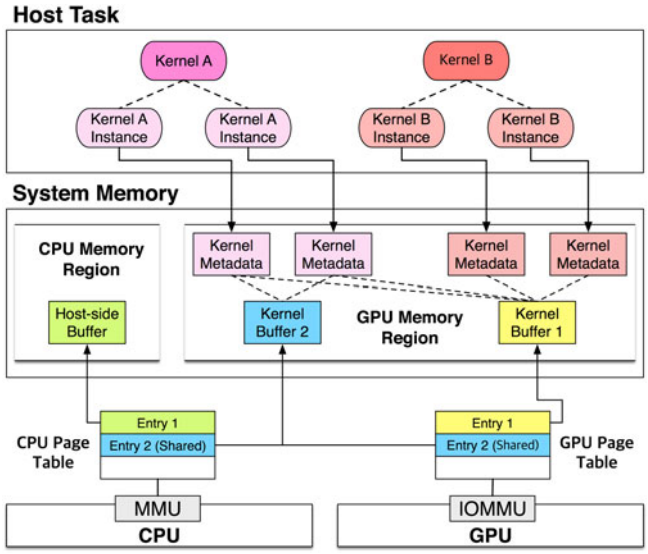


Fig. 1. OpenCL runtime model and memory management in HSA systems.

Fig. 1 illustrates the runtime environment of a GPU kernel that is written in OpenCL in an HSA system.

In the heterogeneous uniform memory access (hUMA) model of HSA, both the CPU and GPU have their own address spaces and MMUs, but share the same physical memory. Consequently, the data transfer overhead between the CPU and GPU can be significantly reduced [12], [13], and the CPU can directly access GPU memory.

In the OpenCL programming model, the host task that issues GPU kernels is executed from the CPU-side. The host task requests a GPU to load its kernels, which are written in the form of functions, into the GPU memory. A kernel loaded into the GPU memory can be executed multiple times with different input buffers, and each execution of a kernel is called a kernel instance.

The host task prepares buffers for storing the input and output of a kernel instance before its execution. A kernel buffer is a memory segment that consists of contiguous virtual pages in the GPU address space. A kernel may utilize multiple kernel buffers. The addresses of kernel buffers to be used by a kernel instance are delivered to the GPU driver as the parameters of the kernel instance. A kernel buffer can be reused multiple times by multiple kernel instances, and the output of a kernel instance can be fed to another kernel instance as input via the shared kernel buffers. Therefore, in most cases kernel buffers are allocated in the initial stage of an application.

In general, a host task runs a series of kernel instances. To request the execution of a specific kernel, the host task creates a kernel instance for that kernel. Every kernel instance has its own metadata to store its state information. This metadata will be referenced by the GPU for execution. Multiple kernel instances can be constructed from the same kernel with different parameters, and they can be simultaneously issued to the GPU driver if they are asynchronous to each other.

When a kernel performs an in-place update over its input data buffer, which is stored in the GPU global memory, the kernel breaks the idempotence property. A non-idempotent

kernel cannot be reexecuted after its execution has been aborted because the aborted run might corrupt the input buffer. Therefore, when a non-idempotent kernel instance is aborted during its execution, the application that initiated the kernel instance must be reexecuted from the beginning.

2.2 GPU Kernel Idempotence

In HPC systems, where a single failure may result in a huge loss of time and resources, or intermittent computing systems in which frequent failures are expected, the checkpointing technique, where a snapshot of a task's current state is stored, is widely employed [14], [15], [16]. This snapshot is used to roll back or restore the task state to the checkpointing time when a crash or failure incident occurs.

However, the checkpointing scheme introduces a significant overhead because creating a snapshot generally requires a huge number of memory copy operations [17]. In fact, storing a snapshot is only required because the target task may modify the input data. If a code block is guaranteed to not modify its input data after reading them, i.e., it does not have operations with anti-dependency, then taking a snapshot beforehand is not necessary because the code block can simply rerun when it fails. Therefore, several approaches that exploit the idempotence property of target code blocks have been proposed to remove the snapshot creation overhead for checkpointing [18], [19], [20], [21], [22].

Kruijf *et al.* [18] proposed a processor architecture that rolls back from a speculative execution failure without checkpointing or state saving when the speculative code block is idempotent. In addition, they proposed a recovery functionality approach that splits a program into a series of idempotent regions and resumes the execution from the last complete idempotent region when a transient fault occurs [19]. Bolt [21] is a compiler-directed soft error recovery scheme for HPC systems that determines the idempotent regions of an application and records the execution progress at the idempotent region granularity level during its execution. In this manner, Bolt can resume the execution from the last marked idempotent region when a soft error occurs, without expensive snapshotting or hardware support. Both Alshboul *et al.* [22] and Lie *et al.* [23] proposed fault recovery schemes that recompute a failed idempotent code region. Both approaches assume that data are stored in non-volatile (NV) memory, and based on this assumption, they do not require expensive snapshot-based checkpointing or logging for fault recovery.

A code block that is idempotent at the semantic level can act as a non-idempotent instruction sequence, depending on the register allocation and spilling policies of the compiler. Kruijf *et al.* [20] proposed a code generation scheme that maintains the idempotent property of a code block at the instruction level with little overhead. They also analyzed the runtime overhead depending on the idempotent region size, instruction set architecture (ISA), and control-flow side-effects based on the proposed scheme.

The aforementioned studies target the CPU program code. A few research results concerning the determination and utilization of idempotence in GPU kernel code have recently been presented. iGPU [24] identifies the semantic idempotent region in GPU kernel source code and makes the kernel act as a flow of idempotent regions using a dynamic idempotent-region code-generation scheme. Based on this, it enables exception support and speculative execution of a GPU kernel with an extremely small overhead. Lin *et al.* [25] proposed a persistency model that allows the GPU kernels to recover from crashes just by using data in memory instead of a checkpoint. In their persistency model, they utilized an idempotence analysis of target kernels to reduce the logging frequency and the size of the logs. Chimera [4], a hardware-assisted preemptive GPU kernel scheduling scheme, identifies the idempotent code region of a GPU kernel based on the approach developed for iGPU. The details of Chimera will be covered in the next section.

Most parallel programs, including GPU kernels, are intended to process a large amount of data and reduce the processed results to a small output. In many cases, this endows GPU kernels with the idempotent property. According to previous research, 12 out of 27 kernels from the well-known Rodinia, Parboil, and Nvidia Computing SDK benchmark suites were idempotent [4]. Therefore, if we can determine whether a kernel is idempotent, then we can resume the execution of an aborted idempotent kernel without expensive snapshotting.

In this research, we employed the parser generator ANTLR 4 [26] to identify idempotent kernels through static code analysis. ANTLR generates a parser based on the syntax rules given by users. When a program source code is fed to the parser generated by ANTLR, it parses the source code and produces an abstract syntax tree (AST), which represents the syntactic structure of the source code in a tree form. Then, ANTLR traverses the nodes in the produced AST and invokes the listener function when it reaches each node. By implementing the *listener* function, it is possible to process each AST node in a customized manner. We programmed this listener function to determine the idempotence of a kernel as described in Section 3.2.

2.3 Preemptive GPU Scheduling

GPU vendors have recently introduced kernel preemption support in their high-end GPUs. The NVIDIA Pascal architecture provides kernel preemption through context saving and restoration. However, a preemption operation takes approximately 100 μs to complete [27]. This long latency, which is not acceptable in real-time embedded systems, results from the enormous overhead involved in saving and restoring the states of massive GPU threads [3], [4]. Moreover, so far, no GPU vendors have publicly released a preemption control interface [7].

Although these have not yet been realized in commercial products, a few approaches that can preempt running kernels without context saving and restoration have been proposed.

A streaming multiprocessor (SM) drain scheme has been proposed, which simply waits for the currently running thread block to finish [3]. Furthermore, a scheduling algorithm has been proposed that dynamically determines and performs the most beneficial context-switching method among hardware-supported preemption, SM drain, and SM flush, preempting an idempotent thread block and recomputing it later [4].

The preemptive scheduling support on the hardware side requires a complicated design, which in turn leads to expensive manufacturing costs and poor energy efficiency.

Therefore, software-based approaches are viable alternatives in embedded systems.

Preemptive Kernel Mode (PKM) partitions a long-running kernel into short-running sub-kernels, so that the execution time of each sub-kernel cannot exceed the allowable scheduling delay [28]. This approach requires the recompilation of existing kernels, and significantly increases the scheduling overhead owing to the increased number of kernel launches. In addition, the shorter the allowable scheduling delay, the smaller the sub-kernel must be.

Gloop [29], EffiSha [6], and FLEP [7] all place persistent GPU threads that micro-schedule the block-tasks of actual kernels. (A block-task is the unit of work done by a thread block.) Compilers are modified to insert a scheduling point at the end of every block. This approach enables preemption without context saving and restoration, although it is performed at the unit block granularity level. However, this can only guarantee that the worst-case scheduling delay is shorter than a certain time-bound when all the computation blocks for execution are guaranteed to be sufficiently short.

As mentioned, these software-based approaches require heavy restructuring of the existing kernel source, and depend on the assumption that kernels voluntarily yield GPU resources on time.

3 Our Approach

3.1 Overview

The abort command is invoked by writing an abort code to the GPU command registers. Once this is issued, the GPU immediately stops the execution at the instruction granularity level and prepares for the next request. Consequently, the abort command can be regarded as a practical method to enforce rapid context-switching to high-priority tasks.

If the execution of a kernel is abruptly aborted, then it is impossible to resume its execution, because the GPU discards its thread states during abortion. It is also technically challenging to reexecute the aborted kernel because the kernel might corrupt the input data by updating them in place. To reexecute an aborted kernel, the series of all previously retired kernel instances of the application must be reexecuted from the beginning, because kernel instances of a task usually form producerconsumer relationships with each other and naturally share the same GPU buffer to deliver the produced data to the next kernel [30]. Therefore, the initial state of a kernel instance can only be obtained by reexecuting the series of all previous kernel instances. As a matter of course, this is not a feasible option for systems with timeliness requirements.

If the initial state of the GPU buffers can be preserved or restored after abortion, a kernel instance can be safely executed again, and the task can continue its operation as if nothing happened. We propose two techniques together that can guarantee the GPU buffers used by a kernel to roll back or to preserve their initial states when the execution of the kernel is aborted.

First, the idempotent kernel classification scheme statically analyzes the kernel source code to determine whether it is idempotent. This idempotency information on a kernel will be delivered to the GPU device driver when the kernel is loaded into memory to be launched.

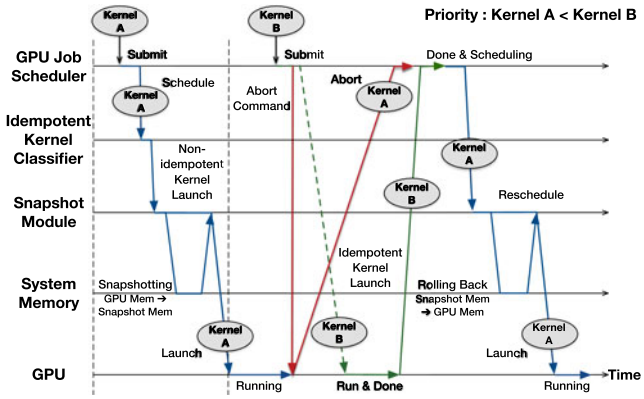


Fig. 2. An example of the proposed preemptive scheduling method.

When the GPU driver launches a kernel instance, it will simply carry out the launching process without performing any measures if the kernel is idempotent. However, if it is not idempotent, or if its source code is unreachable for the classification analysis, then the driver will transactionize the kernel instance before launching it. The transactionization technique creates and stores a snapshot of the kernel buffers, which will be utilized by the launching kernel instance.

When a kernel instance is abruptly aborted, the aborted kernel that was transactionized will be reexecuted following rollback to its initial state. If the kernel instance was not transactionized, it can be simply reexecuted without any operations following abortion.

Based on these two techniques, we propose a preemptive priority scheduling scheme that aborts a low-priority kernel and schedules a high-priority one when the latter becomes runnable.

Fig. 2 depicts an example of preemptive scheduling under the proposed schemes. The GPU scheduler chooses Kernel A, which was issued by Task A and is non-idempotent, to be next in line. The OS creates a snapshot of the GPU memory, including the kernel buffers, to be used by Kernel A. While Kernel A is running, the idempotent Kernel B is issued by Task B, which has a higher priority. Then, the execution of Kernel A is immediately aborted, and Kernel B is scheduled. The GPU scheduler launches Kernel B to the GPU without generating a snapshot. After finishing Kernel B, the GPU scheduler reschedules Kernel A. In turn, the transactionization module initiates the rollback process using the snapshot of Kernel A, as it was previously interrupted and marked as aborted. After rollback, the GPU scheduler launches Kernel A to the GPU, and the execution of Task A continues normally.

The remainder of this section explains the details of these two schemes. Our design requires the modification of the GPU device driver. In this study, we employed the OpenCL programming model on the Linux device driver implementation for Samsung Exynos 5422 SoC with Mali-T628 as the reference design to explain implementation-related issues. This reference driver [31] is fundamentally similar to other HSA GPU device drivers, and thus our approach can be easily adopted to these with minimal modifications. Our implementation also includes the transaction management thread working inside the OS kernel and the idempotent classifier, which is a user-level

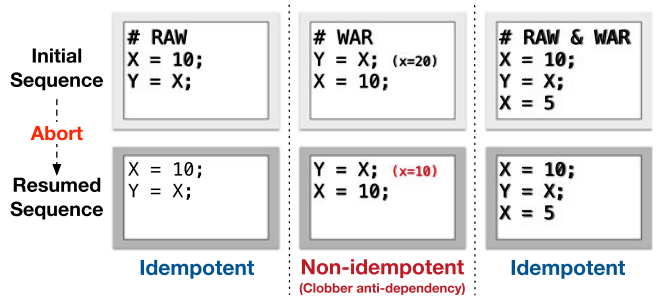


Fig. 3. A clobber anti-dependency pattern is a necessary and sufficient condition for a non-idempotent code block.

application. These are independent to the underlying hardware architecture.

3.2 Idempotent Kernel Classification

Clobber anti-dependency occurs when a variable is accessed in the write-after-read (WAR) pattern without any prior read-after-write (RAW) accesses. If a certain code block has a clobber anti-dependency operation as shown in Fig. 3, then it is not idempotent [19], and its reexecution after abortion may produce incorrect results. Even when a kernel has a WAR pattern, it is idempotent as long as there is a RAW pattern that precedes the WAR pattern.

To determine the idempotency of a kernel, the proposed scheme must detect all statements in the given source code that have clobber anti-dependency. For this, the proposed classifier first caches the names of the kernel buffer parameters to the kernel, and in turn, identifies whether each operation accessing these variables is for reading or writing. This information will be analyzed to determine whether a variable is accessed in the clobber anti-dependency pattern. Once the existence of an anti-dependency pattern is detected, the kernel is labeled as non-idempotent.

Although the fundamental mechanism is simple and straightforward, it is technically challenging to detect every clobber anti-dependency existing in the source code through static analysis, because the actual memory addresses pointed to by some variables, such as pointer-type variables, are often dynamically determined on the fly. As previously stated, the transactionization scheme, which will be explained later, can guarantee consistency, even when a kernel is non-idempotent. Therefore, it is safe to miscategorize an idempotent kernel as non-idempotent, although this may lead to an unnecessary transactionizing overhead. However, if a non-idempotent kernel is misclassified as idempotent, then reexecuting the kernel instance will lead to incorrect results.

To prevent such catastrophic consequences of incorrect classifications, the classifier pessimistically assumes that when the actual address of a read or write operation is not statically determined, every possible address will be accessed by that operation. This policy enables the classifier to determine non-idempotent kernels in every case correctly. However, some of the idempotent kernels that access dynamically determined addresses may be incorrectly categorized as non-idempotent. In other words, the classifier is designed to have a false-positive-only misclassification property, to guarantee the kernel execution soundness.

The pseudocodes presented in Fig. 4 represent the 4 representative cases that create difficulties in analysis, namely conditional branches, dynamically determined addresses, alias pointers, and cross-function accesses. The classifier employs the following rules to handle such cases.

When there is a conditional branch statement, as shown in Fig. 4a, it is impossible to predict whether both branches, first for reading and second for writing, are taken. Therefore, according to the false-positive-only policy, the classifier assumes that both paths are taken. This recurs for nested conditional branches. As the depth of a nested branch statement becomes deep, the number of branch paths to explore will increase exponentially because, at every depth, the number of paths to explore multiplies. However, this will not occur in the real world, because a branch in the GPU code leads to execution path divergence, which critically harms the performance. Thus, developers avoid using branches as much as possible when programming GPU code.

```
1  Kernel (__global int *X, __global int Condition) {
2      if (Condition1)
3          temp = X[idx];
4              ...
5      if (Condition2)
6          X[idx] = value;
7  }
```

(a) Conditional branch.

```
1  Kernel (__global int *X) {
2      int *Alias = X+idx;
3          ...
4      temp = X[idx];
5          ...
6      *Alias = value;
7  }
```

(c) Alias pointers.

```
1  Kernel (__global int *X, __global int *Y) {
2          ...
3      tempX = X[idx+i]; ---------- (case 1)
4          ...
5      X[idx+i] = valueX;
6          ...
7      tempY = Y[idx+i]; ---------- (case 2)
8          ...
9
10     j = ...;  // j happens to be the same as i
11         ...
12     Y[idx+j] = valueY;
13 }
```

(b) Dynamically-determined array index.

```
1  Device (int *Y) {
2          ...
3      Y[idx] = value;
4          ...
5  }
6
7  Kernel (__global int *X) {
8          ...
9      temp = X[idx];
10         ...
11     Device (X);
12         ...
13 }
```

(d) Anti-dependency across function boundaries.

Fig. 4. Four representative cases in which complex anti-dependency patterns appear.

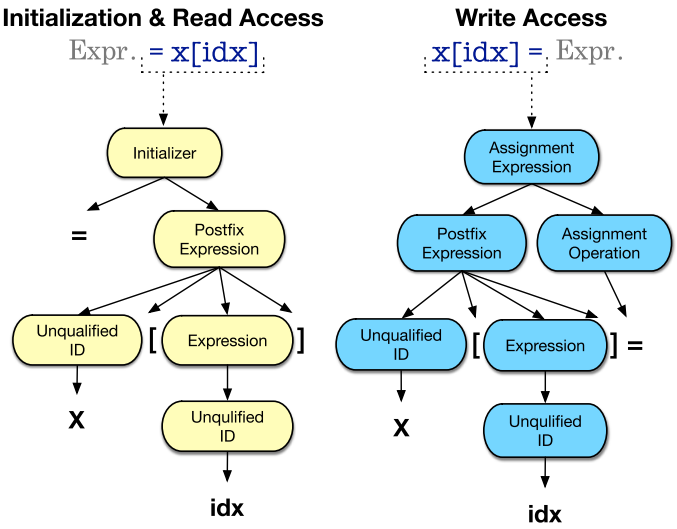


Fig. 5. Example read/write access patterns represented in ASTs produced by ANTLR 4.

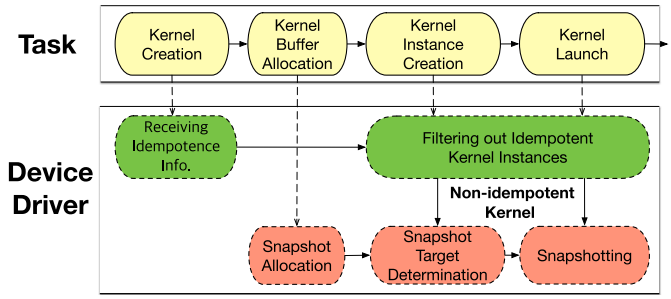


Fig. 6. Idempotent kernel classification and snapshotting steps for each kernel execution stage.

In many cases, the input data are passed to the kernel through the pointer-typed array variables. The actual address to be accessed by an array variable is determined by its array index. As shown in the first case in Fig. 4b, when two accesses to the same array have the same array index and the index value does not change between them, it is possible to determine the presence of anti-dependency in these two accesses. However, as in the second case, the index value may change between a read operation and a following write operation. In such cases, it is difficult to detect whether the two index values are the same. Therefore, following the false-positive-only policy, when an array parameter is written after being read, this will be considered as an anti-dependency case regardless of its indexes.

As illustrated in Fig. 4c, two or more pointer variables can point to the same address in the input buffer. To handle these alias pointers, when an assignment operation to a pointer variable occurs, future occurrences of the left-hand-side (LHS) variable will be replaced with the right-hand-side (RHS) expression of the assignment operation. The variable values in the RHS expression may change after the assignment operation. Therefore, the numerical variables will be replicated to fix their values, and the replicated variables will be utilized for the replacement.

Finally, the GPU kernel code can invoke a device function, which is executed in the GPU. The input buffers, which are the kernel parameters, may be modified in a device function, as shown in Fig. 4d. Therefore, the idempotent kernel classifier should be able to detect anti-dependency across function boundaries. To achieve this, the classifier replaces the device function call with its function body, as if the device function was declared as an inline function. If the called function calls another function or itself, then the function body embedding will be recursively repeated. The parameter variables used inside the function body will be treated as alias variables of the function call arguments, and the scope of these parameter variables will range over the embedded function body.

In addition to conventional code-block or function-level anti-dependency detection, the proposed classifier uses the four aforementioned rules to determine the idempotence of a GPU kernel. Among these rules, function embedding is applied first, and alias processing follows. Finally, the classifier searches for anti-dependency in all possible execution paths.

As explained in Section 2.2, the actual GPU kernel source code analysis is performed using ANTLR 4. We added the syntax rules to process the OpenCL code to the existing C++ ruleset. Fig. 5 presents two example ASTs produced by ANTLR. Each node in these trees is an expression in the C++ language. The classification rules are implemented in the aforementioned listener function. When a variable appears on the LHS of an assignment operator, it is considered to be written, whereas one on the RHS is considered to be read. During the traversal of the tree, the classifier records the read and write the history of each variable. When anti-dependency occurs for a variable, the classifier stops the traversal and labels the kernel as non-idempotent. If the traversal ends without any clobber anti-dependency occurrences, then the kernel is classified as idempotent.

If a kernel is identified as non-idempotent, then the classifier will insert a small piece of code into the initialization part of the kernel source such that the kernel passes its idempotence information to the GPU device driver via the `proc` file system when it is loaded into memory.

3.3 Kernel Snapshotting for Transactionization

If the kernel to be launched has been identified as non-idempotent or its source code is unreachable for classification, then to secure the integrity of the input buffer in case of abortion, kernel transactionization will be performed through snapshotting as follows.

To roll back to the initial state of an aborted kernel instance, all memory areas that the kernel instance can modify during its execution must be included in the snapshot. The snapshot creation process consists of three stages, as shown in Fig. 6.

When a task requests a new kernel buffer, the GPU driver allocates a kernel buffer and maps it onto the GPU address space. At the time of allocation, it is unknown which kernel instances will use the newly allocated buffer. Therefore, the GPU driver also allocates a snapshot space for the newly allocated kernel buffer unless all kernels of the current task are idempotent. Subsequently, when a task issues a kernel instance, the GPU driver determines the snapshot targets of the instance, which include the kernel buffers to be used by the kernel instance.

In general, kernel buffer allocations are performed infrequently, because applications tend to reuse kernel buffers

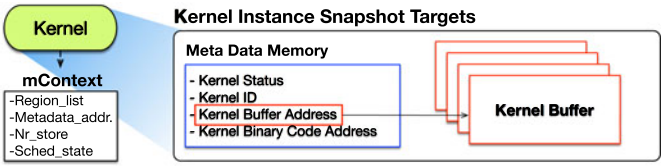


Fig. 7. Snapshot target constitution and a data structure for recording snapshot targets.

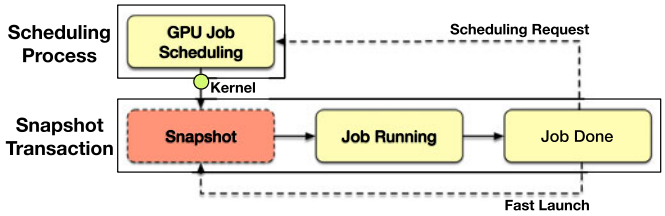


Fig. 8. An independent transaction thread handles the snapshotting and launching of a kernel instance.

once they have been allocated. In addition, there is usually an interval between buffer allocation and kernel instance creation. Thus, the snapshot space allocation time is overlapped by the kernel instance creation interval in most cases.

When a host task requests the creation of a new kernel instance, the OpenCL framework creates a metadata structure for that instance. The instance ID, scheduling state, input parameters, kernel binary address, and so on are included in the metadata, as shown in Fig. 7. To issue a kernel instance, the host task delivers the address of its metadata to the GPU driver via an *ioctl* system call. Therefore, the GPU driver is able to identify the location of the metadata of the kernel instance. By investigating the corresponding fields in the metadata, the GPU driver is able to easily detect the locations of the kernel buffers used by the kernel.

We define the *mContext* data structure to record the snapshot targets and snapshot status of a kernel instance. This is created together with the kernel instance metadata, as illustrated in Fig. 6, and holds the addresses of the kernel buffers and metadata, as shown in Fig. 7. The GPU driver refers to mContext when it creates a snapshot to minimize the snapshot target searching time. Finally, when the kernel instance is scheduled for launch, the kernel buffers included in the snapshot targets will be copied to their pre-allocated snapshot spaces.

The snapshot space of a kernel buffer is released when the corresponding kernel buffer is freed, and the mContext of a kernel instance will be removed when the corresponding kernel instance is eliminated.

3.4 Transactionization Process

The actual snapshot creation and kernel launch processes are managed by an independent kernel thread, called the *transaction thread*, as shown in Fig. 8. By shifting the snapshot and kernel-launch operations to the transaction thread, the GPU scheduler and kernel transaction can run simultaneously. This enables the GPU scheduler to react immediately when a high-priority instance becomes runnable.

After deciding the next kernel instance to execute, the GPU driver delivers the chosen instance to a transaction thread, which is waiting in a sleep state. Then, the transaction thread wakes up and checks the snapshot state of the given kernel instance.

The snapshot of a particular kernel instance can exist in one of the following three conditions. First, if the kernel instance does not have a snapshot because it is first scheduled, the transaction thread will create a snapshot and launch the kernel. Second, if the kernel instance has been aborted during execution, the transaction thread will roll the snapshot target back from its snapshot and relaunch the kernel. Third, if the kernel instance has been aborted during snapshotting, the transaction thread will resume snapshotting and then launch the kernel. Once the kernel instance begins its execution, the transaction thread enters a sleep state and waits for the next request. The kernel instance will be completed by the interrupt handler, which actually invokes the GPU-interrupt handling routine provided by the GPU device driver.

Two or more kernel instances may access the same kernel buffer, allowing them to be snapshotted and executed concurrently will result in consistency issues. Therefore, when two or more kernel instances are waiting to be launched, only one instance must be allowed to exist in either the snapshot stage or the execution stage. This serialization of transactions resolves the consistency problems, and also prevents performance degradation resulting from memory-bus contention, which can occur when both the CPU and GPU access memory intensively at the same time.

However, this approach may degrade the execution time, because it postpones the snapshot creation for the next kernel to the point at which it is scheduled. To relieve this delay, we added a fast launch track to the GPU interrupt handler, as shown in Fig. 8. When the kernel execution is finished, the GPU interrupt handler is invoked to handle the kernel completion. The fast launch track in the GPU interrupt handler checks whether a kernel instance is scheduled, and forwards it to a transaction thread before initiating the completion routine. This enables the snapshotting of the next kernel to overlap with the completion process of the current kernel, consequently covering the snapshot overhead.

Snapshotting requires a significant amount of time to finish, because of a large amount of memory copying is required. However, the snapshot process of a low-priority kernel must be able to stop as soon as a high-priority kernel is ready. For the transaction thread to be canceled in the middle of snapshotting, the transaction thread must provide a communication channel to the GPU driver via an inter-process communication mechanism (IPC). The GPU driver shoots a cancel request to the transaction thread when it needs to terminate an ongoing snapshot operation, and the transaction thread checks whether a cancel request has been received before and after snapshotting. Upon receiving a cancel request, the transaction thread stops its operation immediately and returns to sleep.

During a memory copy operation, the transaction thread cannot check cancellation commands. To resolve this, the transaction thread splits a large-sized memory copy into unit-sized copy operations, and checks the command buffer after each copy operation to suppress the cancellation delay. The larger the size of the memory copy unit, the faster the memory copy will be performed. However, an increase in the memory copy unit size will lead to an increase in the cancellation delay. Consequently, the size of the memory copy unit should be determined by considering

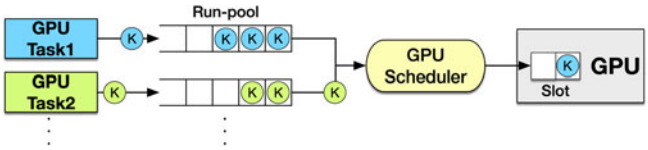


Fig. 9. GPU kernel scheduling structure for target device.

the trade-off between the performance and scheduling delay. We empirically selected 5 MB as the unit size of memory copy for our evaluation.

mContext records *Nr_store*, which is the number of pages copied for snapshotting. The transaction thread checks Nr_store when it receives a kernel instance from the GPU driver. If Nr_store is 0, then the transaction thread will initiate the normal snapshot procedure, because the instance has not yet been scheduled. If Nr_store is equal to the number of snapshot target pages, then the transaction thread will begin the rollback procedure by copying the snapshot data to the snapshot target, because the kernel instance has been aborted during execution. Finally, if Nr_store is neither 0 nor equal to the number of snapshot target pages, then the transaction thread will resume snapshotting from the point at which it was cancelled. By resuming from this point, the transaction thread can minimize the performance loss caused by the snapshotting cancellation.

3.5 Preemptive Priority Scheduling

Fig. 9 illustrates the scheduling procedure for the Mali-T628 GPU driver. When a host task requests the execution of a kernel instance, the instance will be inserted into a *run-pool*. Each task has its own run-pool. The GPU scheduler chooses a task to launch the next kernel. Then, the chosen task picks a kernel queued in its run-pool on a first-come-first-served basis, and places the kernel in an available slot. A slot is mapped one-to-one to the job buffer slot inside the GPU. The kernels in the slots will be launched sequentially and automatically by the GPU.

The GPU driver originally utilized the round-robin scheduler to schedule the next task. However, we modified this to a priority scheduler. The modified scheduler chooses the task with the highest priority among those that have kernels waiting in their run-pools. The priority of a task is determined by its UNIX nice value. A task with a nice value of -20, which represents the most important task, cannot be preempted by other tasks. Such a task is classified as an emergency class task, and its kernel instances are emergency class kernels. The transaction thread will not create snapshots for idempotent-classified or emergency-class kernels. Instead, these are launched immediately.

The GPU scheduler must keep track of the up-to-date state of the current kernel, so that it can be preempted properly when necessary. In addition, the transaction thread must be able to recognize the point at which a given kernel instance was previously preempted, to perform post-preemption correctly. Therefore, mContext also includes the *sched_state* field, which keeps track of the current scheduling state of each kernel instance.

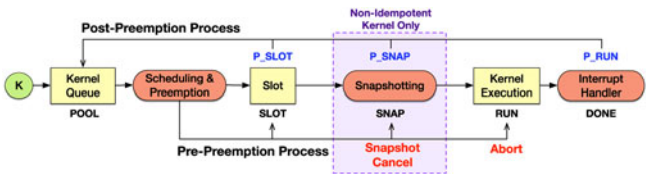


Fig. 10. Kernel scheduling states and pre-/post-preemption procedures.

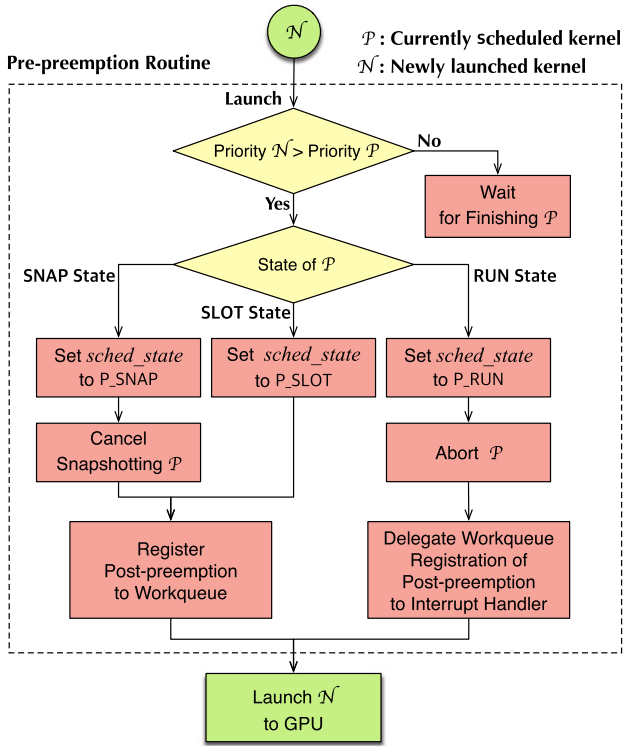


Fig. 11. Flowchart of pre-preemption routine.

Fig. 10 shows the sched_state changes following the scheduling flow. The sched_state is marked as POOL or SLOT when a kernel instance is waiting in the run-pool or in a slot, respectively. When the transaction thread begins snapshotting, it changes the sched_state to SNAP. The sched_state remains as RUN during execution, and the interrupt handler converts it to the DONE state when it begins the kernel completion routine. As previously stated, only one kernel instance can be in either the SNAP or RUN state at a certain time owing to the transaction serialization.

When a preemption occurs, the state of the kernel instance at the preemption point is recorded in the sched_state field. Subsequently, this record will be used to determine the appropriate post-preemption processes for the preempted kernel instance. In the POOL state, a kernel will never be preempted because it is not yet scheduled. When a preemption occurs for a kernel instance in the SLOT, SNAP, or RUN state, the sched_state will be changed to P_SLOT, P_SNAP, or P_RUN, respectively.

The pre-preemption routine, which is illustrated in Fig. 11, is triggered when a new kernel is launched. When the priority of a newly launched kernel is lower than that of the currently scheduled kernel, the newly launched kernel should wait for its turn to be scheduled, and thus no more operations are required. When the new kernel has a higher priority, it evicts the current kernel in the following sequences.

The scheduler first changes the sched_state of the current kernel accordingly. In turn, the scheduler cancels snapshotting the current kernel if it is on-going, or sends an abort

TABLE 1
System Configuration for Evaluation

Board	Odroid-XU3 Rev. 2	SoC	Exynos 5422
CPU	4×Cortex A15@2.0GHz 4×Cortex A7@1.4Ghz 2MB Shared L2 Cache	GPU	Mali-T628 MP6 @600Mhz 256KB L2 Cache
Memory	2GB LPDDR3 RAM @ 933MHz		
Storage	32GB eMMC 5.0 HS400 Flash Storage		
OS	Linux 3.10.72 with Mali r5p0-06rel0 driver		

command to the GPU if the current kernel is being executed. After that, the scheduler registers the post-preemption routine for the kernel to the workqueue. Finally, it continues launching the high-priority kernel.

The post-preemption routine is invoked through the workqueue mechanism of the Linux OS. Therefore, the post-preemption of the preempted low-priority kernel can be executed in parallel with the launching operation of the high-priority kernel.

The post-preemption routine reinitializes the metadata of the preempted kernel if it has been preempted in the P_RUN state. In turn, it returns the kernel instance to the corresponding run-pool, and changes its state to POOL. It is possible that new kernel instances may have been inserted into the run-pool. Because a kernel instance may have dependencies on previously issued kernel instances, kernel instances must be executed in order of their issue sequence. Therefore, in such cases, the post-preemption routine must place the preempted kernel instance back in its correct location in the run-pool to maintain the correct order of instances.

A newly scheduled instance that is about to preempt a low-priority instance can also be preempted in the future unless it is not in the emergency class. Therefore, its snapshot must be taken before its execution. If the instance to be preempted is in the SNAP state, then snapshotting of the high-priority kernel instance can be initiated immediately, because no consistency issues can occur in snapshotting two asynchronously issued kernel instances. Therefore, the proposed scheme prepares and utilizes multiple transaction threads to parallelize snapshotting. When the GPU driver sends a snapshot cancellation command to the working transaction thread, the GPU driver first invokes a waiting transaction thread, and then immediately starts a transaction for the new high-priority kernel instance. Therefore, the snapshot cancellation and snapshot creation can be carried out in parallel to further reduce the scheduling delay.

4 Evaluation

4.1 Evaluation Environment

For evaluation, we implemented the proposed schemes in the system described in Table 1. Although the memory performance is poorer than the cutting-edge accelerated processing units (APUs) available on the market, we chose this system because its device driver source code is publicly available, unlike most embedded APUs. We believe that the proposed scheme will perform significantly better than our implementation when applied to up-to-date APUs.

We used 11 benchmarks from 20 programs of Rodinia, a GPU-computing benchmark suite [32]. Tables 2 and 3 present the properties of these programs. The O and X symbols in the Idemp. column of Table 2 denote that the corresponding workload is idempotent or non-idempotent, respectively. The 9 excluded benchmarks could not be executed, even on the unmodified Linux kernel and device driver, because the evaluation system does not satisfy their resource requirements, such as the number of computing units in the GPU, although the amount of system memory was sufficient to accommodate them.

We performed the idempotent classification on the 11 benchmarks, and the results are listed in Table 2. Seven out of the 11 workloads have idempotent kernels. All the kernels are idempotent in 6 of these 7 applications, whereas the Gaussian has one idempotent and one non-idempotent kernel. We performed a manual analysis of the source code to

TABLE 2
Workload Description and Idempotent Kernel Information

Program	Abbr.	Description	Kernel Name	Idemp.
k-Nearest Neighbors	NN	Dense Linear Algebra	NearestNeighbor	O
Back Propagation	BP	Unstructured Grid	bpnn_layerforward_ocl bpnn_adjust_weights_ocl	X X
PathFinder	PF	Dynamic Programming	dynproc_kernel	O
Breadth-First Search	BFS	Graph Traversal	BFS_1 BFS_2(G)	X X
Kmeans	KM	Dense Linear Algebra	kmeans_kernel_c kmeans_swap	O O
Hotspot3D	3D	Structured Grid	hotspotOpt1	O
LU Decomposition	LUD	Dense Linear Algebra	lud_diagonal lud_perimeter lud_internal	X X X
Needleman-Wunsch	NW	Dynamic Programming	nw_kernel1 nw_kernel2	X X
Gaussian Elimination	GE	Dense Linear Algebra	Fan1 Fan2	O X
Myocyte	MC	Structured Grid	kernel_gpu_opencl	O
CFD Solver	CFD	Unstructured Grid	memset_kernel initialize_variables compute_step_factor compute_flux time_step	O O O O O

TABLE 3
Workload Performance Characteristics

Abbr.	Num. of Kernel Inst.	Snapshot Size per Kernel (Avg.)	Snapshot Memory Size (at Peak)	Avg. Kernel Exec. Time	Worst Kernel Exec. Time	Turnaround Time
NN	1	60.00 MB	60.35 MB	10.38 ms	10.38 ms	0.68 s
BP	2	6.79 MB	9.42 MB	15.74 ms	48.62 ms	0.39 s
PF	5	38.84 MB	39.25 MB	609.00 ms	614.98 ms	3.50 s
BFS	24	21.94 MB	37.55 MB	4.57 ms	37.12 ms	0.91 s
KM	38	67.60 MB	130.38 MB	115.01 ms	116.66 ms	8.40 s
3D	100	24.00 MB	24.30 MB	10.78 ms	10.80 ms	11.82 s
LUD	190	4.15 MB	26.92 MB	2.35 ms	18.73 ms	10.18 s
NW	255	32.47 MB	104.45 MB	0.29 ms	0.53 ms	0.67 s
GE	510	0.50 MB	1.18 MB	0.26 ms	1.28 ms	0.40 s
MC	3,913	1.76 MB	2.12 MB	0.13 ms	0.15 ms	8.63 s
CFD	14,004	7.24 MB	19.30 MB	5.53 ms	11.86 ms	99.99 s

determine the idempotency, and also checked whether there were changes in the input buffer after execution. These verifications showed that there were no false-negative misclassifications. In addition, the source codes of the benchmark suites we used do not contain any false positives. This means that the kernel actually updates its input whenever an input array variable is used in the LHS of an assignment statement, and there were no cases in which a conditional branch decided idempotency.

In addition to using these existing benchmarks, we implemented a hypothetical GPU benchmark program, *M-Bench*, to be used as disturbing tasks. M-Bench repetitively invokes a kernel that executes matrix multiplication operations. M-Bench adjusts the number and execution times of kernel instances, along with the sizes of kernel buffers, according to the given parameters. The numbers of threads and thread blocks are also determined accordingly. We set the default kernel buffer size for each M-Bench kernel instance to 23 MB and the default kernel execution time to 70 ms, reflecting the average values of the benchmark kernels.

We first measured the turnaround time from the initiation of a benchmark program to its completion. This experiment can predict how long the execution times of priority tasks will be when they are executed with the disturbing tasks (M-Bench). We also analyzed the scheduling delay under the proposed schemes. Finally, we investigated the overhead of our approach.

We set the nice value of M-Bench to 19, which represents the lowest priority. The target benchmark programs were configured to have a common priority of 0 or emergency class priority of -20, depending on the experiment.

4.2 Turnaround Time

To analyze the execution time increase for prioritized tasks arising from the competition for GPU resources, we measured the turnaround times of the benchmark programs while simultaneously running M-Bench.

While conducting the experiment, we varied the number of concurrently running M-Bench instances from one to two. Fig. 12 shows the average, maximum, and minimum turnaround times for each benchmark program. Each result was normalized according to the turnaround time, which was measured when the corresponding benchmark program was independently (without competing M-Bench instances) executed in the unmodified Linux kernel. Because each benchmark program generally has a large number of kernel instances, the turnaround time of each run exhibits little variance.

As expected, the turnaround time increased as the number of concurrently running M-Bench tasks increased. This tendency was exhibited strongly for the benchmark programs with a short kernel execution times and a large number of kernel instances. This is because they triggered more frequent GPU scheduling and thus the M-Bench tasks have more chances to steal the GPU from such workloads. For example, NN, BP, and PF have fewer than five kernel instances, and the impacts to their execution times from increased scheduling delays were relatively small, at up to 55 percent. In contrast, the execution time of GE, where the kernels had a short execution time, underwent a 187-fold increase. MC, which has a larger number of kernel instances with a shorter average execution time than GE, exhibited a smaller impact, because the proportion of CPU-computing phases of MC was significantly larger than for GE.

The proposed scheduling scheme successfully suppressed the turnaround times of common priority tasks so that they did not increase by over 1.28 times, except for NW. And, they did not exhibit any meaningful changes in the turnaround time when the number of disturbing tasks increased to two. NW, which has a large number of extremely short non-idempotent kernel instances, underwent a turnaround time increase of approximately 413 percent. However, applications with small numbers of kernel instances and snapshot targets of small sizes, such as BP,

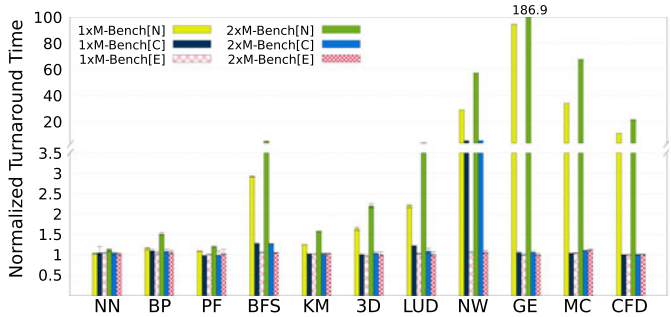


Fig. 12. Normalized turnaround times of benchmark programs while varying the number of disturbing background tasks. [N] denotes the results obtained using the unmodified driver. [C] and [E] represent the results obtained using common and emergency priority under the proposed schemes, respectively.

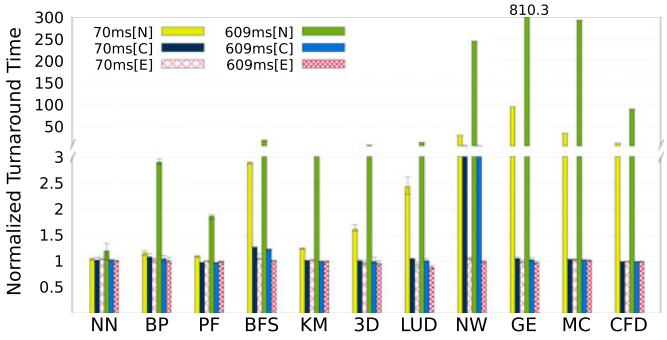


Fig. 13. Normalized turnaround times of benchmark programs while varying the execution time of a background disturbing kernel instance.

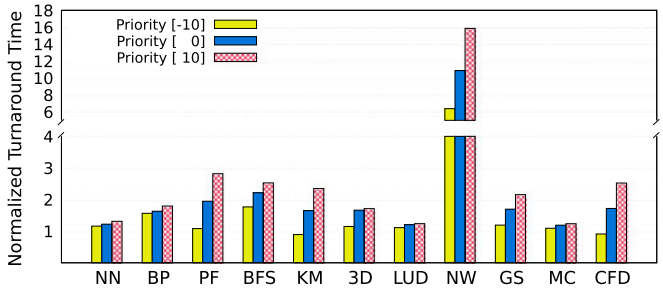


Fig. 14. Normalized turnaround times of benchmarks when three instances of the same benchmark with different priorities were run at the same time. Each instance runs to its finish once.

exhibited minimal increases in execution times of less than 8.8 percent, despite their non-idempotent kernels.

The performances of the workloads with idempotent kernels commonly improved by skipping the transacionization process. The observed maximum turnaround time delays of all-idempotent-kernel workloads were 10.2 and 11 percent for PF and MC, respectively. This slowdown was due to the scheduling delays of host processes, not GPU kernel instances, which occur when new kernel instances are launched. In fact, the average turnaround time for PF was expedited by 3 percent because of the fast launch path introduced in Fig. 8, whereas that of MC decreased by approximately 10 percent owing to its huge number of kernel instances.

The execution times of emergency class tasks also increased, but this increase was below 10 percent in most of the applications. The recorded maximum delay was 14.1 percent, which occurred when PF was run at the same time as two M-Bench tasks. Both the transaction threads and the complicated scheduling structure contributed to this delayed execution time to a certain extent. However, the main reason for this delay stemmed from the jitters in CPU scheduling caused by intermittently running interrupt handlers and other internal OS operations. This is because the turnaround time for NW was very short, and thus, a coincidental scheduling jitter would critically affect its turnaround time.

The scheduling delay of a kernel increases in proportion to the lengths of the competing kernels. This is why existing software-based priority GPU schedulers partition long kernels into multiple small sub-kernels. To investigate the impact on the performance of the extended kernel execution times of competing tasks, we measured the turnaround time of each benchmark program. The execution times of M-Bench kernel instances were set to 70 and 609 ms, which are the observed average and maximum kernel instance durations of Rodinia, respectively.

As shown in Fig. 13, the turnaround times of all cases significantly increased as the execution time of the M-Bench kernel increased, except for NN. The impact of the increased kernel length was clearly greater than that of the increased number of M-Bench instances. For example, GE had a turnaround time that was 800 times longer.

The proposed scheduling scheme yielded indistinguishable results from those in Fig. 12. The turnaround time for common priority tasks increased by up to 418 percent, which was recorded by running NW with 70-ms-long M-Bench kernels. However, other than for NW these exhibited increment below 29 percent. The turnaround times for emergency class tasks were delayed by less than 15 percent. The maximum observed turnaround time delays of emergency class tasks were even shorter than those shown in Fig. 12. This is because the number of competing tasks in this experiment was smaller than in the previous one, and thus fewer CPU scheduling jitters were produced.

To evaluate the proposed scheme when running tasks with multiple priorities, we simultaneously executed 3 instances of the same benchmark with nice values of -10, 0, and 10, respectively. As illustrated in Fig. 14, the higher the task priority was, the faster the task finished. The execution times of the highest priority tasks doesn't increased by more than 100 percent in any benchmark, except for NW. The execution time of the highest priority NW was delayed by over six times its original execution time. This was also affected by the aforementioned factors. Owing to the snapshot overhead and abortion-and-reexecution penalty, the total execution time of each task set increased more than on the unmodified Linux kernel and device driver. The snapshot overhead of our approach will be discussed in further detail in Section 4.4.

Because the proposed scheduler is a priority scheduler rather than a fair-share scheduler, if a high-priority task dominates the GPU resource, then low-priority tasks may be suspended owing to starvation. Nevertheless, in all benchmarks, the prolonged execution times of the lowest priority tasks were significantly shorter than the sum of the execution times of the two high-priority tasks. This is because the benchmarks interleave CPU- and GPU-computing phases, and therefore low-priority tasks could be opportunistically scheduled to finish during the CPU-computing phases of a high-priority task. However, complete suspension of low-priority kernel execution may occur when high-priority tasks are highly GPU-intensive.

To assess the degree of the starvation, we measured the turnaround time of an M-Bench instance, which is the target of measurement, simultaneously running with a disturbing M-Bench instance. The target M-Bench instance was configured to have a low priority while the disturbing one ran at a high priority. The kernel length of the high-priority instance was fixed at 70 ms.

The pink surface in Fig. 15 shows the turnaround time change while varying both the activation interval of the disturbing task and the kernel length of the target task. Naturally, the target task could not progress when the disturbance interval was shorter than the kernel length. However, when the interval was larger than the kernel duration, the turnaround time was not significantly prolonged. Specifically, it stayed

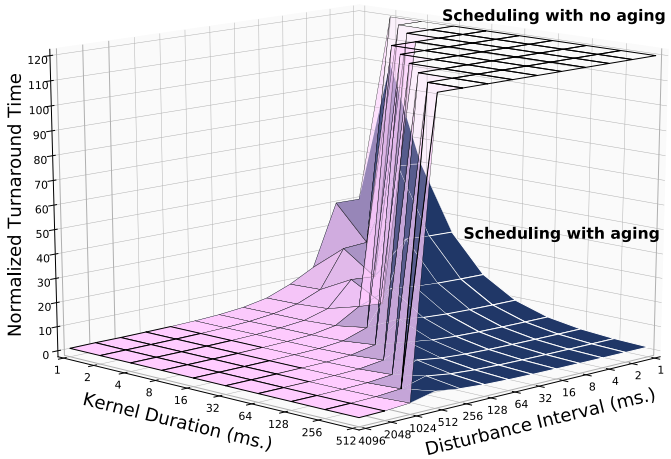


Fig. 15. Normalized turnaround time of a low-priority kernel while varying its kernel length and the disturbance interval of the high-priority kernel.

below twice the original number when the kernel length was longer than 32 ms and the interval was longer than quadruple this.

The starvation caused by priority scheduling can be remedied with the aging scheme [33]. The blue surface in Fig. 15 shows the turnaround time when the scheduler was modified to gradually boost up the priority of a kernel every time it was preempted by a high-priority one. The target kernels were guaranteed to progress at the cost of temporal priority inversion when applying the aging scheme. The execution time delay at the extreme condition where the activation interval of the high-priority task was shorter than the execution time of the low-priority kernel ranged from 1.5 times to 100 times depending on the kernel length.

In summary, these experiments showed that the proposed schemes successfully secured the GPU resource for high-priority tasks under a heavy load or a run with competing tasks with long kernels. In addition, using the emergency class priority guaranteed a bounded increase in the turnaround time in all cases.

4.3 Scheduling Delays

We analyzed the scheduling delays of the benchmark applications observed during the experiments described in Section 4.2.

Fig. 16 shows the distribution of the preemption delay, which is the interval from the issuance of a scheduling

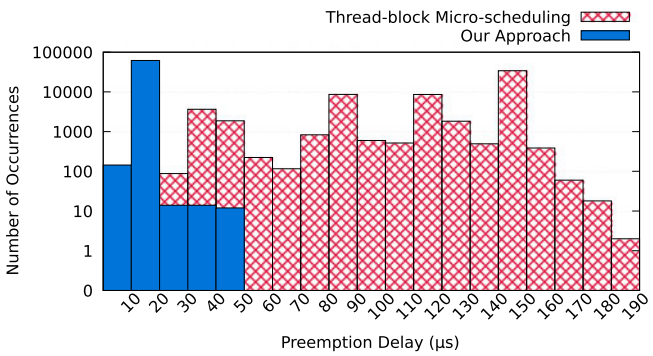


Fig. 16. Distribution of preemption delay depending on preemption scheme.

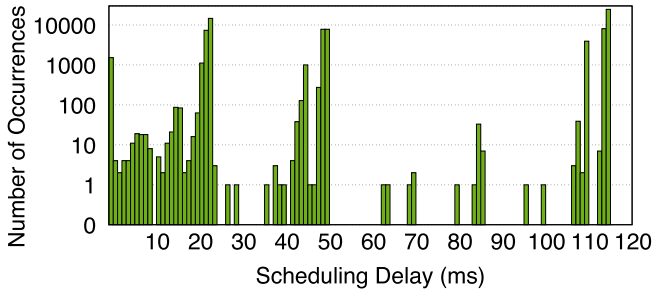


Fig. 17. Distribution of the scheduling delay of benchmark programs executed on the unmodified Linux kernel.

request to the time at which the GPU becomes ready for the next kernel. We compared the preemption delay of our approach with that of the thread-block micro-scheduling scheme, which was introduced in Section 2.3. We could not use any of the existing compilers that produce micro-scheduling code for our evaluation because only a few of them were available to the public. Furthermore, of those that were available, some operated with only the Nvidia CUDA framework but not OpenCL. Therefore, we handcrafted the micro-scheduling versions of the benchmark workloads following EffiSha [6].

As shown in Fig. 16, the preemption delays under the micro-scheduling scheme were distributed over a large span, which ranged from 20 μs to 190 μs. This is because a preemption request must wait until the end of the currently-running thread-block, and each thread-block requires different a execution time to finish. On the contrary, with our approach, the preemption delays were mostly shorter than 16 μs, and their distribution was very narrow because aborting the kernel execution is immediate and the status or characteristics of the currently-running kernel barely impact the abortion process. Note that the Y-axis is log-scaled. The 99.9th percentile of the eviction delay was 18 μs.

For comparison, we measured the scheduling delay while conducting the same set of experiments in the unmodified Linux kernel. As shown in Fig. 17, the scheduling delay exhibited an extremely wide distribution, which resulted in severe fluctuations in the turnaround time.

Fig. 18 shows the times taken for the pre-preemption and launch routines of emergency class kernels. Because an emergency class kernel does not take a snapshot, the launch process in the proposed schemes is the same as that of the unmodified Linux kernel. The pre-preemption delay was

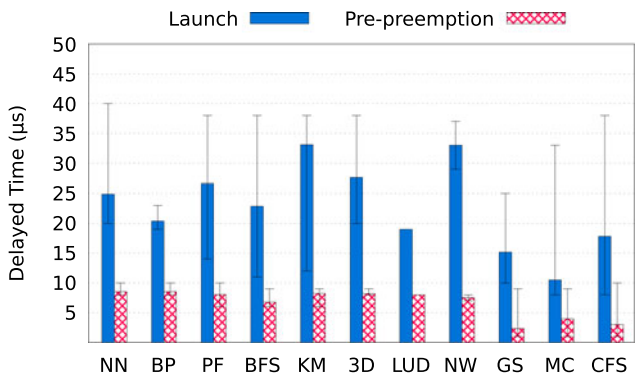


Fig. 18. Minimum, maximum, and average pre-preemption and kernel launch delays observed during the execution of benchmark programs.

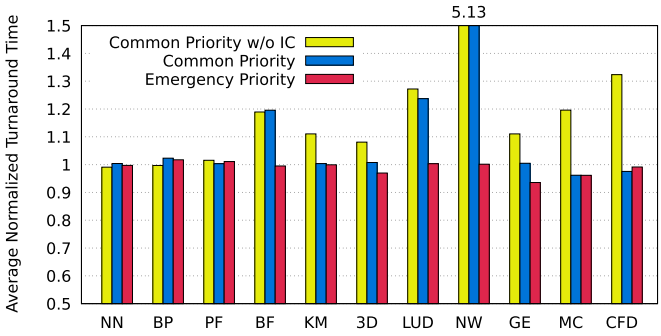


Fig. 19. Performance degradation of benchmark applications when independently run under the proposed schemes.

mostly less than 10 μs, whereas the launch delay was notably longer than 10 μs.

Most of the launch routine can be conducted simultaneously with the eviction process. Therefore, the eviction time has a trivial impact on the overall scheduling delay because the eviction delay is generally shorter than the launch delay. Consequently, the overall scheduling delay is mostly determined by the sum of the pre-preemption and launch delays. The launch procedure is too straightforward to be further reduced. Thus, this can be considered as the minimum scheduling delay possible. The proposed schemes only added approximately 10 μs to the minimal scheduling delay in the case of the emergency class priority.

4.4 Overhead

The proposed transactionization scheme generates massive memory copy operations, which may lead to a significant performance degradation. A few other components introduced in the proposed scheme, including snapshot space allocation, snapshot target identification, and transaction thread operations, also contribute to the performance degradation to some degree. In addition, the preemptive scheduling mechanism adds pre-preemption and post-preemption routines to the execution flow.

Launching idempotent kernel instances does not produce snapshotting overhead. However, both delivering idempotency information to the device driver and matching a kernel instance with its idempotency information may marginally impact performance.

To quantitatively analyze the performance overhead, we measured the turnaround times of the benchmark applications while executing each application independently under the proposed schemes. The results were normalized to those obtained using an unmodified Linux kernel. To analyze the snapshotting overhead in-depth, we additionally experimented without the help of the classifier, so that every kernel instance was considered to be non-idempotent and transactionized.

As shown in Fig. 19, the execution times of the applications were delayed by 5 to 413 percent when they were configured to have a common priority and did not use the classification information. The degree of performance degradation in this experiment was smaller than those of Figs. 12 and 13, because snapshotting was conducted by the CPU, and thus the performance impact increased when there were multiple tasks competing for the CPU resource.

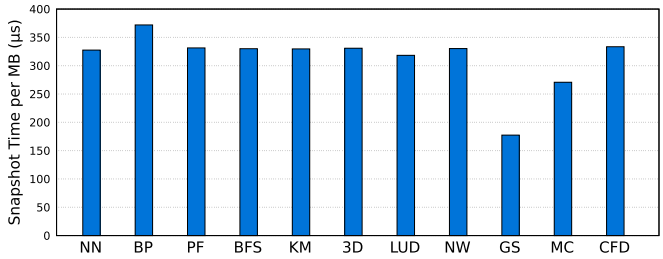


Fig. 20. Snapshot creation time per MB of GPU buffer data for different workloads.

The maximum observed delay from the applications with idempotent kernels was only 1.5 percent, which was from PF. This was a result of from delivering kernel idempotence information and matching with kernel instances.

Only one of the two kernels of GE is idempotent, and the snapshot time was reduced from 46 μs to 23 μs. This reduction improved the turnaround time by 10.6 percent in comparison with that without the idempotent classifier. MC and CFD earned the best performance gains from applying the idempotent classifier. Although the sizes of their kernel buffers to snapshot were small, the numbers of kernel instances were large. This led to high snapshotting overheads, and the idempotent classifier successfully removed it.

No applications yielded delayed execution times when they had the emergency class priority. On the contrary, in this case, the turnaround times of MC and CFD, which performed frequent kernel instance scheduling, notably decreased. This was because of the optimization of the scheduling path, such as the fast launch track.

These results reveal that most of the overhead of the proposed schemes results from the memory copy operations for snapshotting, and the other components do not contribute significantly to the overhead.

Fig. 20 shows the processing times for one MB of snapshot data during the execution of each workload. The smaller the average GPU buffer size was, the longer the time that the snapshot operation required. Depending on the GPU buffer size and the number of GPU buffers being used at the same time, the snapshotting overhead varied significantly. In PF, snapshotting added only 2.2 percent of the GPU computing time. On the other hand, during the execution of NW snapshotting increased the kernel execution time by 36 times, for to the aforementioned reasons. However, we believe that NW represents a rare case, because its kernels commonly receive a large set of GPU buffers as their input, although each of them only accesses a small range of GPU buffers during its unusually short execution time.

At a given time point, the total size of the snapshot memory is approximately the same as the sum of the GPU buffers being used by the currently running kernel. The peak memory usage for managing snapshots is described in Table 3. Considering that the GPU buffers are generally significantly smaller than the capacity of the main memory, the space overhead of the proposed scheme is considered to be within an acceptable range, although not negligible.

Most of the snapshot overhead was due to memory copying, which is expected to improve with hardware performance, including processors, memory, and system buses. For example, an embedded board equipped with a 64-bit

dual-channel LPDDR4X provides a memory bandwidth of 33.4 GB/s [34], which is twice that of the one used in our evaluation, and cutting-edge memory technologies, such as through-silicon via (TSV) memory technology [35] and 3D-stacked memory, will achieve 51 GB/s of memory bandwidth in mobile devices in the near future [36].

5 Conclusion

Both GPU computing and preemptive priority scheduling are quintessential for modern embedded systems. However, considering that embedded systems are rigorously limited in terms of energy efficiency, form-factors, and production costs, it is technically challenging to realize hardware-supported preemptive kernel scheduling.

To achieve immediate scheduling of high-priority kernels, we enabled the immediate abortion of a running kernel instance and its reexecution. This was achieved using two techniques proposed in this paper: idempotent GPU kernel classification and GPU kernel transactionization. Based on these approaches, we also proposed a preemptive priority GPU kernel scheduling scheme. This yields extremely short scheduling delays without any hardware support.

Our evaluation showed that the proposed scheduling scheme successfully suppressed the scheduling delay and provided preemptive priority scheduling under diverse heavy load conditions. However, in some cases, the proposed scheme incurred a significant overhead. We will further refine our approach to resolve such edge cases by combining our current work with existing kernel-slicing and persistent-thread approaches.

Acknowledgments

This work was supported in part by the National Research Foundation of Korea under PF Class Heterogeneous High Performance Computer Development (No. 2016M3C4A7952587) and in part by the Next-Generation Information Computing Development Program through the National Research Foundation of Korea funded by the Ministry of Science, ICT (MSIT) (No. 2017M3C4A7080245).

References

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

[2] J. Kim, R. R. Rajkumar, and S. Kato,"Towards adaptive GPU resource management for embedded real-time systems," *ACM SIGBED Rev.*, vol. 10, no. 1, pp. 14–17, 2013.

[3] I. Tanasic, I. Gelado, J. Cabezas, A. Ramirez, N. Navarro, and M. Valero, "Enabling preemptive multiprogramming on GPUs," in *Proc. 41st Annu. ACM/IEEE Int. Symp. Comput. Archit.*, 2014, pp. 193–204.

[4] J. Park, J. Kyu, Y. Park, and S. Mahlke, "Chimera: Collaborative preemption for multitasking on a shared GPU," in *Proc. 20th Int. Conf. Archit. Support Program. Lang. Operating Syst.*, 2015, pp. 593–606.

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

[6] G. Chen, Y. Zhao, X. Shen, and H. Zhou, "EffiSha: A software framework for enabling effficient preemptive scheduling of GPU," in *Proc. 22nd ACM SIGPLAN Symp. Princ. Practice Parallel Program.*, 2017, pp. 3–16.

[7] B. Wu, X. Liu, X. Zhou, and C. Jiang, "FLEP: Enabling flexible and efficient preemption on GPUs," in *Proc. 22nd Int. Conf. Archit. Support Program. Lang. Operating Syst.*, 2017, pp. 483–496.

[8] H. Lee, J. Roh, and E. Seo, "A GPU kernel transactionization scheme for preemptive priority scheduling," in *Proc. IEEE Real-Time Embedded Technol. Appl. Symp.*, 2018, pp. 202–213.

[9] Khronos Group, "OpenCL," 2009. [Online]. Available: https://www.khronos.org/opencl

[10] P. Muyan-Özçelik and J. D. Owens, "Multitasking real-time embedded GPU computing tasks," in *Proc. 7th Int. Workshop Program. Models Appl. Multicores Manycores*, 2016, pp. 78–87.

[11] S. Azmat, L. Wills, and S. Wills, "Accelerating adaptive background modeling on low-power integrated GPUs," in *Proc. 41st Int. Conf. Parallel Process. Workshops*, 2012, pp. 568–573.

[12] A. Maghazeh, U. D. Bordoloi, P. Eles, and Z. Peng, "General purpose computing on low-power embedded GPUs: Has it come of age?" in *Proc. Int. Conf. Embedded Comput. Syst., Archite. Model. Simul.*, 2013, pp. 1–10.

[13] Heterogeneous System Architecture Foundation, "Heterogeneous system architecture," 2016. [Online]. Available: http://hsafoundation.com

[14] R. Koo and S. Toueg, "Checkpointing and rollback-recovery for distributed systems," *IEEE Trans. Softw. Eng.*, vol. SE-13, no. 1, pp. 23–31, Jan. 1987.

[15] J. Duell, P. Hargrove, and E. Roman, "The design and implementation of berkeley lab's Linux checkpoint/restart," 2002.

[16] K. Maeng and B. Lucia, "Adaptive dynamic checkpointing for safe efficient intermittent computing," in *Proc. 13th USENIX Symp. Operating Syst. Des. Implementation*, 2018, pp. 129–144.

[17] G. Luan, Y. Bai, C. Wang, J. Zeng, and Q. Chen, "An efficient checkpoint and recovery mechanism for real-time embedded systems," in *Proc. IEEE Int. Conf. Parallel Distrib. Process. Appl. Ubiquitous Comput. Commun. Big Data Cloud Comput. Social Comput. Netw. Sustain. Comput. Commun.*, 2018, pp. 824–831.

[18] M. De Kruijf and K. Sankaralingam, "Idempotent processor architecture," in *Proc. 44th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2011, pp. 140–151.

[19] M. De Kruijf, K. Sankaralingam, and S. Jha, "Static analysis and compiler design for idempotent processing," in *Proc. 33rd ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2012, pp. 475–486.

[20] M. De Kruijf and K. Sankaralingam, "Idempotent code generation: Implementation, analysis, and evaluation," in *Proc. IEEE/ACM Int. Symp. Code Gener. Optim.*, 2013, pp. 1–12.

[21] Q. Liu, C. Jung, D. Lee, and D. Tiwari, "Compiler-directed lightweight checkpointing for fine-grained guaranteed soft error recovery," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2016, pp. 228–239.

[22] M. Alshboul, H. Elnawawy, R. Elkhouly, K. Kimura, J. Tuck, and Y. Solihin, "Efficient checkpointing with recompute scheme for non-volatile main memory," *ACM Trans. Archit. Code Optim.*, vol. 16, no. 2, 2019, Art. no. 18.

[23] Q. Liu, J. Izraelevitz, S. K. Lee, M. L. Scott, S. H. Noh, and C. Jung, "iDO: Compiler-directed failure atomicity for nonvolatile memory," in *Proc. 51st Annu. IEEE/ACM Int. Symp. Microarchit.*, 2018, pp. 258–270.

[24] J. Menon, M. De Kruijf, and K. Sankaralingam, "iGPU: Exception support and speculative execution on GPUs," in *Proc. 39th Annu. Int. Symp. Comput. Archit.*, 2012, pp. 72–83.

[25] Z. Lin, M. Alshboul, Y. Solihin, and H. Zhou, "Exploring memory persistency models for GPUs," in *Proc. IEEE 28th Int. Conf. Parallel Archit. Compilation Techn.*, 2019, pp. 311–323.

[26] T. Parr, *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2013.

[27] NVIDIA, "NVIDIA GeForce GTX 1080," White Paper, 2016.

[28] C. Basaran and K. Kang, "Supporting preemptive task executions and memory copies in GPGPUs," in *Proc. 24th Euromicro Conf. Real-Time Syst.*, 2012, pp. 287–296.

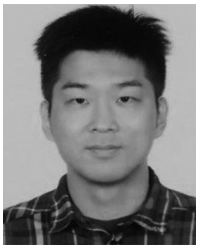
[29] Y. Suzuki, H. Yamada, S. Kato, and K. Kono, "Towards multi-tenant GPGPU: Event-driven programming model for system-wide scheduling on shared GPUs," in *Proc. Workshop Multicore Rack-Scale Syst.*, 2016.

[30] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel, "PTask: Operating system abstractions to manage GPUs as compute devices," in *Proc. 23rd ACM Symp. Operating Syst. Princ.*, 2011, pp. 233–248.

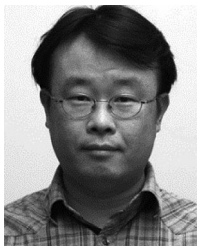
[31] ARM Co., Ltd., "Open source mali midgard GPU kernel drivers (rel. r5p0-06rel0)," 2014. [Online]. Available: https://developer.arm.com/tools-and-software/graphics-and-gaming/mali-drivers/midgard-kernel

[32] S. Che *et al.*, "Rodinia: A benchmark suite for heterogeneous computing," in *Proc. IEEE Int. Symp. Workload Characterization*, 2009, pp. 44–54.
[33] S. Abraham, G. Peter, Baer, and G. Greg, *Operating System Concepts*, 9th ed. Hoboken, NJ, USA: Wiley, 2013, Art. no. 271.
[34] Samsung Electronics Co., Ltd., "Exynos specification," 2019. [Online]. Available: https://www.samsung.com/semiconductor/minisite/exynos/products/all-processors/
[35] M. Motoyoshi, "Through-Silicon Via (TSV)," *Proc. IEEE*, vol. 97, no. 1, pp. 43–48, Jan. 2009.
[36] N. Agarwal, D. Nellans, M. Stephenson, M. O'Connor, and S. W. Keckler, "Page placement strategies for GPUs within heterogeneous memory systems," in *Proc. 20th Int. Conf. Archit. Support Program. Lang. Operating Syst.*, 2015, pp. 607–618.

Hyeonsu Lee received the BS degree in computer science and engineering from Gongju National University, Gongju, Republic of Korea, in 2015. He is currently working toward the PhD degree in the Department of Computer Science and Engineering, Sungkyunkwan University, Seoul, Republic of Korea. His research interests include GPU computing systems, embedded systems, real-time systems, and cloud computing.

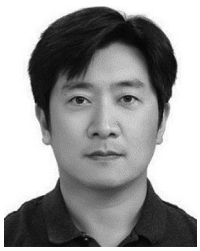


Hyunjun Kim received the BS degree in computer engineering from Sungkyunkwan University, Seoul, Republic of Korea, in 2013. He is currently working toward the PhD degree in the Department of Electrical and Computer Engineering, Sungkyunkwan University, Seoul, Republic of Korea. His current research interests include compiler for high-performance computing.



Cheolgi Kim received the BS degree in computer science from the Korea Advance Institute of Science and Technology (KAIST), Daejeon, South Korea, in 1996, and the PhD degree from the Korea Advance Institute of Science and Technology (KAIST), Daejeon, South Korea, in 2004. He is currently an associate professor with the Software and Computer Engineering Department, Korea Aerospace University, Republic of Korea. He had founded Ratio LLC, Korea, an IoT device company, in 2017. His research area includes safety critical system software architecture, low-energy embedded systems, and real-time systems.

Hwansoo Han received the BS and MS degrees in computer engineering from Seoul National University, Seoul, Republic of Korea, in 1993 and 1995, respectively, and the PhD degree in computer science from the University of Maryland at College Park, College Park, Maryland, in 2001. From 2001 to 2002, he was a senior engineer with Intel, Santa Clara, California. From 2003 to 2008, he was an associate professor with KAIST. Since 2008, he has been a professor with the Department of Computer Science and Engineering, Sungkyunkwan University, Republic of Korea. His current research interests include compiler and operating system technology for high-performance computing, nonvolatile memories, and secure computing.



Euiseong Seo received the BS, MS, and PhD degrees in computer science from KAIST, Daejeon, South Korea, in 2000, 2002, and 2007, respectively. He is currently a professor with the Department of Computer Science and Engineering, Sungkyunkwan University, Republic of Korea. Before joining Sungkyunkwan University in 2012, he had been an assistant professor with the Ulsan National Institute of Science and Technology (UNIST), Republic of Korea from 2009 to 2012, and a research associate with the Pennsylvania State University from 2007 to 2009. His research interests include system software, embedded systems, and cloud computing.