Journal of Systems Architecture 76 (2017) 17–27

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

Priority-driven spatial resource sharing scheduling for embedded graphics processing units

Yunji Kang, Woohyun Joo, Sungkil Lee, Dongkun Shin*

Sungkyunkwan University, 2066 Seobu-ro, Jangan-gu, Suwon, 16419, Republic of Korea

ARTICLE INFO

Article history:
Received 19 May 2016
Revised 6 February 2017
Accepted 19 April 2017
Available online 26 April 2017

Keywords:
Embedded GPU
GPU job scheduling
Spatial resource sharing
Resource reservation

ABSTRACT

Many visual tasks in modern personal devices such smartphones resort heavily to graphics processing units (GPUs) for their fluent user experiences. Because most GPUs for embedded systems are non-preemptive by nature, it is important to schedule GPU resources efficiently across multiple GPU tasks. We present a novel spatial resource sharing (SRS) technique for GPU tasks, called a budget-reservation spatial resource sharing (BR-SRS) scheduling, which limits the number of GPU processing cores for a job based on the priority of the job. Such a priority-driven resource assignment can prevent a high-priority foreground GPU task from being delayed by background GPU tasks. The BR-SRS scheduler is invoked only twice at the arrival and completion of jobs, and thus, the scheduling overhead is minimized as well. We evaluated the performance of our scheduling scheme in an Android-based smartphone, and found that the proposed technique significantly improved the performance of high-priority tasks in comparison to the previous temporal budget-based multi-task scheduling.

1. Introduction

Recent smart devices such as smartphones, smart TVs, and tablet PCs run many visual applications in parallel, which include graphical games, video players, web browsers, and rich graphical user interfaces (GUIs). For instance, a user can launch multiple GPU applications on the home screen, such as live wallpaper, widgets, and popup browsers. Another example is when a user video-chats with colleagues, while playing a graphical game.

GPUs embedded within recent system-on-chips strongly facilitate the execution of such visual tasks by exploiting multiple cores in parallel [1,2]. For example, ARM Mail-400 GPU has one geometry processor (GP) and four pixel processors (PPs) [3]. The realm of GPUs further expanded beyond the traditional area of visual computing owing to unified shaders, which encompasses even computation-intensive workloads such as augmented reality, real-time object recognition, and deep learning [4–7]. For instance, Mali-T880 GPU is composed of 16 shader cores [8].

As the number of applications relying on GPUs grows rapidly, an efficient multi-task scheduling of GPUs is becoming increasingly important. Fluent user experiences across multiple visual tasks require the supports of priority-driven service, quality-of-service (QoS), and performance isolation. In particular, time-critical interactive foreground processes should be prioritized over background processes (e.g., live wallpapers). Nonetheless, the majority of current GPUs schedule them still on the basis of the first-come-first-service (FCFS) without considering priorities of GPU tasks.

A priority-driven GPU scheduling algorithm was recently proposed to mitigate the problem for desktop GPUs, which allocates different time budgets to GPU tasks based on their priorities [9]. While effective in general, such an approach does not perfectly fit with embedded GPUs for several reasons. First, the non-preemptive nature of GPU tasks does not allow complete individual control of the time utilization for each task. Second, its timer interrupt handling additionally incurs non-negligible overhead in embedded systems. Third, their scheduling algorithm assumes that a single task entirely uses a GPU at a time, but this is not true in recent GPUs; the currently available GPUs support a spatial multi-tasking to allow for multiple GPU tasks to be executed in parallel at different GPU cores [3,8], and there were many studies on the spatial multi-tasking of GPU [10–14]. These limitations motivated us to explore a better way to schedule multiple GPU tasks.

This paper presents a spatial resource sharing (SRS) technique for non-preemptive sporadic GPU tasks, which schedules multiple GPU tasks at different GPU cores simultaneously. Our *budget reservation-based* spatial resource sharing (BR-SRS) scheduler reserves a different number of processing cores for each task based on its priority. Unlike the previous time-based multi-tasking, BR-SRS can effectively deal with the non-preemptive nature of GPU jobs. In particular, the BR-SRS scheduler is invoked only twice

* Corresponding author.
E-mail address: dongkun@skku.edu (D. Shin).

http://dx.doi.org/10.1016/j.sysarc.2017.04.002
1383-7621/

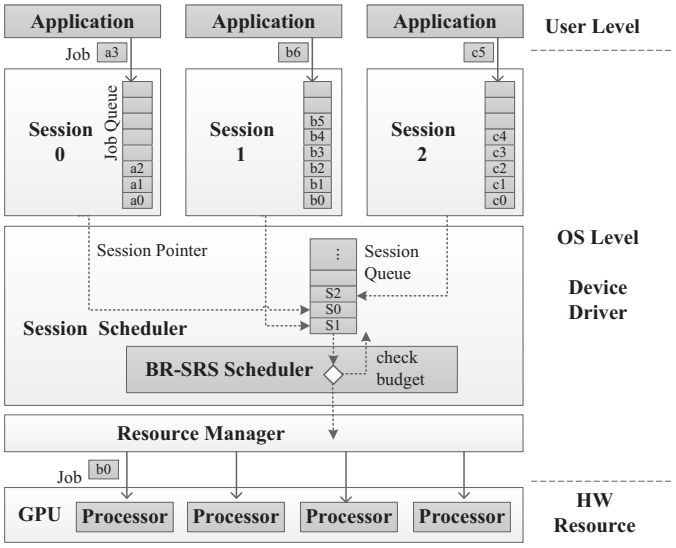


Fig. 1. The architecture of a GPU device driver.

at the job arrival and completion, and thus, the significant overhead of timer interrupt handling present in the time-based multi-tasking is avoided as well. In comparison to the previous spatial multi-tasking techniques, the BR-SRS scheduling can provide instant responsiveness to a user-interactive foreground graphics application in personal mobile devices. We implemented the BR-SRS scheduler with an Android-based smartphone device, and carried out experiments to assess its performance against the previous temporal budget-based multi-tasking algorithm.

2. GPU processing model

When a user implements graphical applications, a GPU library is generally used to efficiently communicate with GPU devices. One of the most common examples is OpenGL for Embedded Systems (OpenGL-ES). Such a library interacts with GPUs and provides an abstract interface, which allows a user to write a graphical application without deeply figuring out the underlying architecture of the GPU. Functions invoked by the GPU library generate a series of commands, and enqueue them into the GPU job queue. Then, GPU device driver sends the jobs to the GPU device to perform them in a row.

Since multiple applications can simultaneously use the GPU devices, the device driver should schedule GPU jobs based on fairness or priorities among applications. When a GPU job starts in the GPU, it cannot be preempted, in general, until the job is completed; recently, there have been several studies to tackle the preemptive scheduling of GPU jobs, which will be reviewed in Section 3. Since a GPU is composed of multiple processing cores, one GPU job can use multiple processing cores, or multiple jobs can be simultaneously scheduled at different cores in the GPU.

Fig. 1 shows the overall architecture of a GPU device driver and its job scheduling. The device driver contains sessions, a session scheduler, and a resource manager. A session is a data structure to manage the job queue of a user application. Each session is allocated for each application, inheriting the priority of its application. The session scheduler selects the session to be scheduled and sends a GPU job from the session to the GPU device, when there are available GPU resources. The session scheduler considers the priorities of different sessions, and thus, a low-priority session can be scheduled only when there are no jobs in higher-priority sessions. However, within a single application, the jobs need to be sent in the order they were enqueued, and therefore, the scheduler dispatches the oldest job from the job queue without reordering. The resource manager controls the state of each processing core. If a job is completed by the GPU or a new job is inserted into a job queue, the session scheduler dispatches a new job from a session queue. The GPU has multiple homogeneous processing cores, and several GPU jobs can be processed by multiple processing cores in parallel.

In this paper, our BR-SRS scheduler focuses on how to improve GPU job scheduling in the session scheduler. It manages the resource budgets of GPU tasks. The initial budget values are assigned based on the priorities of GPU tasks. A GPU task is not allowed to use more resources than its resource budget. The BR-SRS scheduler will then examine the next GPU task in the session queue.

Generally, only one foreground GPU application interacts with the user at a time in a smarthpone whereas there can be many background GPU applications. The performance of a foreground application should not be delayed by background applications. Therefore, we can divide GPU applications into two groups, i.e., high-priority group and low-priority group. If a GPU application runs in foreground, it is categorized into the high-priority group. However, if another foreground application is launched and the previous application is changed to a background application, the background application is moved into the low-priority group. The Android's graphics architecture uses the SurfaceFlinger to draw graphic windows at display unit, which accepts buffers of graphical data from multiple applications, makes a composite of them, and sends it to the display [15]. The SurfaceFlinger is a separate process isolated from user applications, and it also uses the GPU. Because the SurfaceFlinger is responsible for making the final frame buffer image to be displayed, it also should be categorized as a high-priority task in addition to the foreground application.

3. Related work

3.1. Temporal budget reservation

TimeGraph [9] is a priority-driven GPU job scheduler that uses the temporal budget reservation (TBR) technique. The TBR scheduler assigns different time budgets to GPU tasks with different priorities. Each task can utilize GPU resources only when its time budget and resources are available. The time budget is replenished periodically. The TBR scheduler uses two resource reservation policies: posterior enforcement (PE) and a priori enforcement (AE). While the PE policy enforces GPU resource usage once a GPU task is completed, the AE policy predictively enforces GPU resource usage before a GPU task is submitted based on the predicted task execution time.

In order to manage the temporal budget, the TBR scheduler demands the timer interrupt service, which results in interrupt handling and context switching overheads. Moreover, the TBR with PE policy cannot prevent the overrun of low-priority tasks due to the non-preemptive nature of GPU processing. Therefore, the processing of a higher-priority task may be delayed by a lower-priority task if the lower-priority task arrives before the higher-priority task and holds all the GPU resources.

The TBR with AE policy can alleviate such a problem by predicting the execution times of GPU tasks based on profiling. It prevents a GPU task from being scheduled if the remaining temporal budget is smaller than the expected execution time. Since the GPU task generates dynamically variable workloads of GPU jobs, it is not sufficient to predict the execution time based only on the task identification. Therefore, the AE technique predicts the execution time using the command sequences of GPU jobs. Such a prediction technique requires a considerable amount of CPU and memory overhead, and thus, is not applicable to mobile devices. In addition,

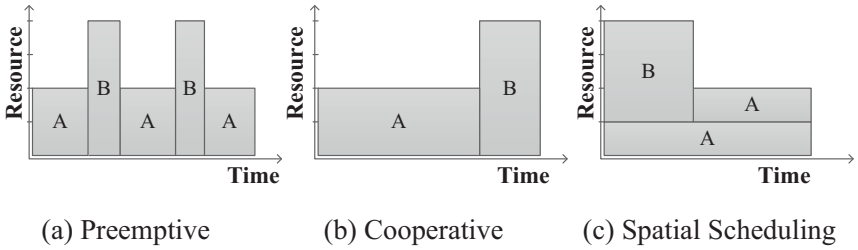


Fig. 2. Different multi-tasking algorithms.

because it is difficult to precisely predict the execution time, TBR cannot completely avoid overrun situations.

Our budget reservation scheduler also uses a prediction technique on the resource demands of GPU tasks. Whereas the TBR with AE policy must predict the execution times of all tasks to prevent the overruns of the tasks, our approach requires to predict the GPU resources of only primary tasks. If there are overruns of low-priority tasks due to the miss-prediction on resource demands, the high-priority tasks can be delayed in the TBR scheme. Therefore, the TBR scheme must consider a large margin when predicting the resource demand of each task, and must allocate more time budget to each task than required. However, our technique needs to consider only the margins for primary tasks. In addition, as will be explained in Section 4.2, our BR-SRS scheme provides an option of over-allocation for high-priority tasks to allocate more resources to them if there are available resources in GPU. By reserving sufficient resources of primary tasks, they can utilize the reserved GPU cores irrespective of the dynamic workloads of low-priority background tasks.

3.2. Spatial multi-tasking in GPU

There are two types of multi-task scheduling techniques for single processor, preemptive scheduling (Fig. 2(a)) and cooperative scheduling (Fig. 2(b)). The cooperative scheduling allows a task to use a resource only when the currently running task voluntarily releases the resource, whereas the preemptive scheduling allows a higher-priority task to preempt lower-priority tasks. The task scheduler in the preemptive scheduling examines the priorities of all the runnable tasks at each time tick. However, since there is no timer interrupt handling mechanism in GPUs, the GPU scheduler usually uses the cooperative scheduling.

In multi-processor or multi-core systems, multiple independent tasks can be simultaneously scheduled for different processors or cores (Fig. 2(c)); this technique is called the *spatial resource sharing* (SRS), which controls computing resources among competing tasks, rather than their processing times. Therefore, multiple tasks can be executed simultaneously at different processing cores. Adriaens et al. [10] showed that SRS can make a speed-up in the GPU up to 1.19 times against the cooperative multi-tasking.

While SRS is adopted by most CPU scheduling techniques, it is optional in GPU scheduling. To enable SRS, the GPU device driver must be able to manage the state of each processing core in a GPU device. For example, TimeGraph assumes that SRS is not supported by the target GPU; therefore, only one GPU job can be executed at a time in the GPU. However, as recent GPUs start to adopt SRS, our work assumes that SRS is allowed for the target GPU.

There have been many researches on the spatial multi-tasking on GPU [10–14]. They relied on a simple heuristics on resource allocation, such as the even-split policy, or a priority and QoS-driven resource sharing algorithm. However, the previous researches are focused on general-purpose GPUs (GPGPUs). The GPGPU workloads are generally bandwidth-sensitive rather than latency-sensitive. The goals of the previous techniques are to maximize the GPU utilization, minimize the power consumption of GPU, or satisfy the average bandwidth requirement of each application. They partition GPU cores for multiple pending GPU jobs considering the bandwidth requirement and priority of each job. No GPU resources are reserved for high-priority tasks. Therefore, a latency-sensitive GPU request cannot be serviced immediately if all the GPU cores are allocated for other tasks. In user-interactive personal devices, it is more important to improve the responsiveness of latency-sensitive applications rather than the fair-sharing of GPU resources. Our BR-SRS scheduler uses a resource reservation policy to provide instant responsiveness to a foreground graphics application. Our previous work [16] first proposed the resource reservation-based SRS algorithm for embedded GPUs. In the present work, several enhanced SRS scheduling techniques are proposed, and they are compared with previous techniques.

3.3. Preemptive GPU scheduling

GPUs run asynchronously against the host, and their jobs are not under strict control from the host. GPU jobs typically trigger very heavy kernels and memory transactions, and their scheduling often relies on the hardware scheduler for load balancing. Hence, the scheduling of GPU jobs has been non-preemptive in general. Recently, there have been several approaches to tackle the preemptive scheduling of GPU jobs, particularly for GPGPUs.

The key idea of the preemptive GPU scheduling is to decompose kernels and memory transactions so that the custom scheduler could find preemption points within short intervals. Long-running kernels are decomposed to sub-kernels [12,17–20]. Also, large data for memory transactions are decomposed to small chunks [17,19,21]. The majority of the works on preemptive GPGPU scheduling required user-level as well as drive-level supports.

Recently, the preemption becomes to be supported in the levels of hardware and proprietary drivers. For example, NVIDIA's newest Pascal architecture supports pixel-level preemption, which saves off context information to resume when preemption is requested [22]. Such a hardware-level preemption support greatly facilitates responsive computing/rendering of time-critical GPU workloads, but it is mostly designed for desktop computing platforms, not available for mobile platforms yet.

Despite the great work for the preemptive GPU scheduling, the approaches cannot be easily applied to mobile GPUs. While most previous studies focused on GPGPU rather than visual rendering, most of the foreground jobs are visual tasks in mobile devices. Unlike the computing-style kernels, visual rendering typically follows the pipeline architecture, which is not easily decomposable for both shaders and memory transactions. At present, the preemption for the pipelined rendering is supported only for the latest desktop GPUs (e.g., NVIDIA Pascal Architecture). Also, even when large kernels are decomposed to smaller sub-kernels, immediate preemption is not guaranteed, because preemption points are still too sparse in comparison to typical CPU workloads. In contrast, our work reserves GPU cores for high priority jobs, and thus, is better for instant running of high priority tasks.

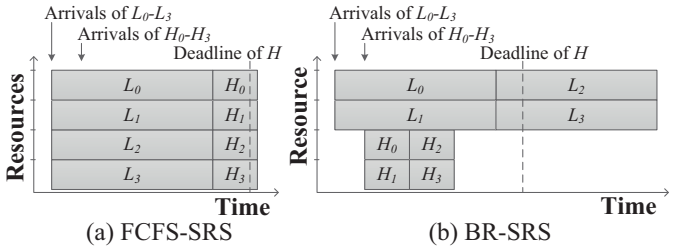


Fig. 3. Comparison between FCFS scheduler and BR-SRS scheduler.

4. Budget reservation-based spatial resource sharing

4.1. Overview

The normal SRS technique uses the FCFS policy, which can allocate all idle GPU cores to any arriving task irrespective of its priority. We call such a scheduling technique FCFS-SRS. The non-preemptive scheduling of GPU jobs may delay a high-priority task under the FCFS-SRS scheme if all the GPU resources are allocated for low-priority tasks. Our BR-SRS scheduling uses a resource reservation technique, where a pre-specified number of processing cores are *reserved* for the high-priority task group and the others are used for the low-priority task group. The resource budget for high-priority task group is determined statically by profiling the workload of high-priority tasks. As commented in Section 3.1, the determined resource budget will be larger than the profiled data to cope with miss-predictions. The number of GPU resources for background tasks is determined at runtime by excluding the resource budget of the higher-priority tasks from the available resources. Since a lower-priority GPU job cannot use the reserved resources of higher-priority jobs, a high-priority task is never delayed by lower-priority tasks.

Fig. 3 compares the FCFS-SRS and BR-SRS techniques. When the jobs of low-priority task, L, arrives before the jobs of high-priority task, H, the jobs of $L_0 - L_3$ occupy all the processing cores in the FCFS-SRS. (Generally, multiple pixel processor (PP) jobs are generated almost simultaneously during a graphic processing. This is because one GP processing job generates multiple related PP jobs.) Therefore, when the jobs of $H_0 - H_3$ arrive, there are no available resources and the high-priority task jobs must wait the completions of $L_0 - L_3$ without meeting their deadlines. In BR-SRS, because two of four processing cores are reserved for the jobs of high-priority task, the low-priority jobs can use only two processing cores. The jobs of high-priority task can therefore be scheduled immediately or with only small delays, and meet their deadlines.

The BR-SRS scheduler is invoked only twice at the events of job arrival and completion, while the TBR scheduler requires timer interrupts. Although the BR-SRS also depends on the interrupts from GPU to be informed of job completion, the TBR requires periodic timer interrupts for budget replenishment in addition to the GPU interrupts for the notification of job completion. Therefore, the TBR requires additional interrupts, and thus incurs more overheads. The BR-SRS scheduler examines the first GPU job in the session queue. If there are available resources other than those reserved for higher-priority tasks, the job can be scheduled immediately. Otherwise, the low-priority jobs must wait in the session queue. The budget for each task is reduced at the start of its GPU job and is replenished at the completion of its GPU job.

Both the BR-SRS and TBR schedulers assign different budgets to tasks with different-priorities. While TBR uses temporal budgets, BR-SRS uses spatial budgets. Even when there are available resources, TBR does not assign the resources to a task if the task has no remaining time budget. If the remaining time budget is less than zero (or less than the predicted execution time in the AE scheme), TBR registers a timer interrupt that will be fired at the next replenishment time of the task. However, BR-SRS can immediately schedule a waiting GPU job when any resource is released by the completion of another job. Therefore, BR-SRS can execute GPU jobs with lower latencies and increase GPU utilization.

Fig. 4 shows the job schedules and budget changes of the TBR-AE and BR-SRS schedulers. A high-priority task, H, and a low-priority task, L, are scheduled. In Fig. 4(a), each of two tasks is assigned 50% of the time budget and thus can use 2.5 s within 5 s of time interval. It is assumed that the AE version of TBR will predict the execution time of L to be 2.5 s. Therefore, the low-priority task in the TBR scheduler can use all four processing cores if the remaining time budget is not less than 2.5 s. As a result, the high-priority jobs of $H_0 - H_3$ are significantly delayed by the jobs of $L_0 - L_3$. In addition, when $L_0 - L_3$ are completed in the TBR scheduling, the time budget is less than zero, since the task execution time exceeds the assigned budget. Therefore, the next GPU jobs, $L_4 - L_7$, cannot be immediately scheduled at the next period and must wait until the budget is fully replenished; the next jobs, $L_8 - L_{11}$, are also delayed. The period of timer interrupts in the TBR scheme affects the timer interrupt overhead and the processor utilization. Although a long period can reduce the total timer interrupt overhead, there can be many idle intervals before the periodic budget replenishments. On the other hand, a short period can improve GPU utilization but will increase the total interrupt overhead.

In Fig. 4(b), however, two processing cores in the BR-SRS scheduler are reserved for the high-priority task, H. Therefore, low-priority task, L, can use only at maximum two processing cores. The BR-SRS scheduler provides shorter latencies for the high-priority task, H, than the TBR scheduler does. In addition, the low-priority task, L, also has shorter latencies in the BR-SRS scheduler compared to the TBR scheduler, because the low-priority jobs can be executed immediately in the presence of available resources.

The BR-SRS scheduler can be integrated with power management techniques. When there are only low-priority tasks, several GPU processing cores will be idle as a result of the reservation policy of BR-SRS. For low power consumption, idle devices can enter a low-power mode. However, the wake-up delay from the low-power mode should be considered when the resource budget is assigned.

4.2. Resource over-allocation and under-allocation for high-priority tasks

In the BR-SRS scheme, even the high-priority tasks cannot use more resources than their reserved budgets. Considering that the high-priority tasks defined in our work are user-interactive foreground applications, if we permit the BR-SRS scheduler to assign more resources to higher-priority tasks than their static budgets at runtime, the responsiveness of high-priority GPU jobs will be improved significantly. We call this technique *over-allocation*. Under the over-allocation technique, a high-priority task is allowed to use more resources than its initial budget if there are available resources. Therefore, if all the GPU cores have already been allocated for higher-priority tasks, low-priority tasks may not be scheduled. Consequently, the over-allocation technique can improve the performance of only the highest-priority tasks with the finite processing cores of the GPU, and the performance of low-priority tasks may be degraded. However, the scheduling goal is to improve the performance of high-priority tasks even though the performances of low-priority tasks are sacrificed. This is a reasonable approach, because the performance of a user-interactive foreground job is more important than other background jobs. To prevent starvation of background tasks, the scheduler monitors the performance of background GPU tasks by measuring the frame rate of them. If the

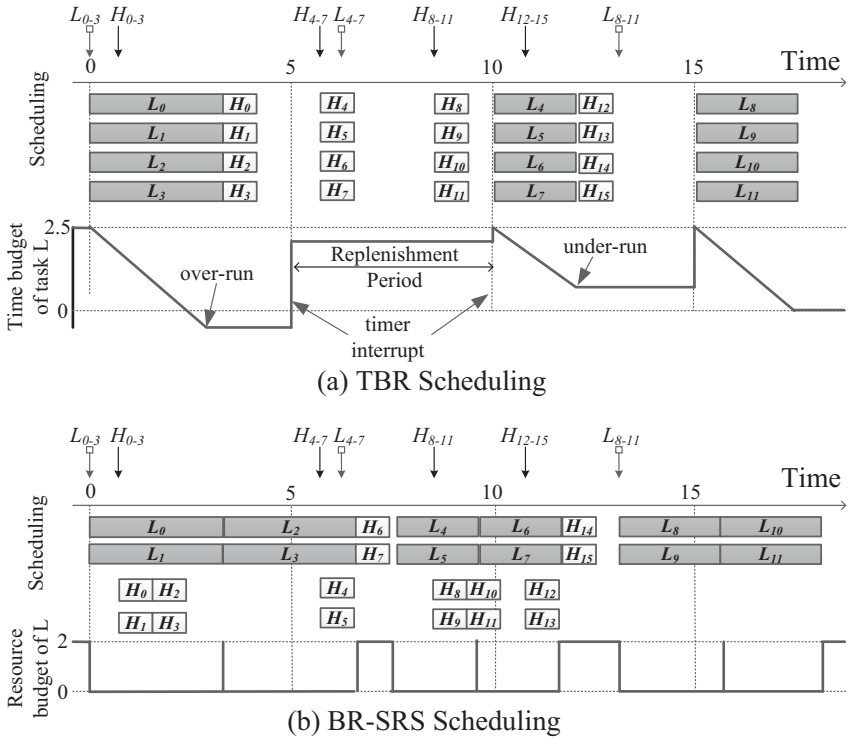


Fig. 4. Task scheduling and budget changes of the TBR and BR-SRS schedulers.

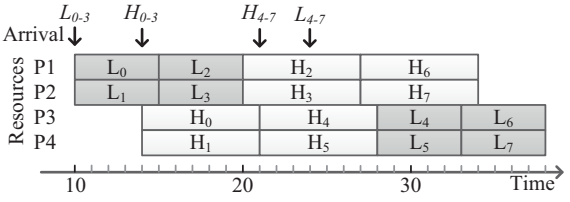


Fig. 5. An example of resource over-allocation at BR-SRS.

performance is less than the threshold θ_{bg}, the over-allocation is disabled. The value of θ_{bg} can be configured considering the responsiveness requirements of foreground and background tasks.

Fig. 5 shows an example of resource over-allocation for two different priorities of GPU tasks, *H* and *L*. The higher-priority task, *H*, is allocated with the minimum budgets of two processing cores. When four low-priority jobs, $L_0 - L_3$, arrive at a time of 10, task *L* can use only two processing cores even though there are four available cores. When four jobs of *H* arrive at a time of 14, task *H* can use only two reserved processing cores. When all the jobs of *L* are completed at a time of 20, task *H* can occupy two processing cores additionally if the over-allocation is allowed. As a result, all the processing cores are occupied by the high-priority task during the time interval of 20–28. When the low-priority jobs arrive at a time of 24, they cannot be scheduled immediately because there are no available resources.

On the other hand, if the GPU requests of high-priority tasks are generated infrequently in the original BR-SRS scheme, the GPU utilization can be wasted due to the reserved GPU cores for high-priority tasks. To improve GPU utilization without degrading the performance of foreground GPU tasks, an adaptive *under-allocation* technique can be used, which decreases the number of GPU cores reserved for the high-priority task group if the GPU workload of high-priority tasks is less than the threshold θ_{fg}. If the workload of high-priority tasks becomes larger than the threshold value, the original budget of the high-priority tasks is restored. In a real usage scenario of smartphone, user interaction with a foreground GPU application generally switches between burst mode and idle mode. The under-allocation technique will reduce the reserved budget of high-priority tasks only at the idle mode. Therefore, the value of θ_{fg} should be configured to detect the idle mode. The under-allocation scheme will be effective to increase GPU utilization without a significant performance degradation of foreground applications.

5. Experiment

5.1. Setup and benchmarks

We evaluated the performance of BR-SRS on a real Android-based smartphone equipped with an embedded GPU, ARM Mali-400MP (267 MHz), which is composed of one geometry processor (GP) and four pixel processors (PPs). The system is also equipped with Samsung Exynos 4210 (ARM Cortex-A9 Dual Core) CPU, and 1GB LPDDR2 DRAM. The Android version is 4.0.4. The BR-SRS scheduler was implemented within the GPU device driver in Linux 3.0.15. The FCFS-SRS and TBR schedulers were also implemented for comparison. The evaluated schedulers were used only for PPs, because there is only one GP in the target GPU. To profile of the processing of GPU requests, a few profiling codes were inserted in the GPU device driver.

As evaluation scenarios, the popup browser [23] was chosen as a foreground task, while several GPU benchmark programs are executed as background tasks. Five different background applications were used, each of which has a different GPU workload. The Nenamark1 [24] is a benchmark of OpenGL ES 2.0, using programmable shaders for graphical effects such as reflections, dynamic shadows, parametric surfaces, particles and different light models to push the GPU to its limits. The KFS [25] is a simple OpenGL benchmark, which renders three different scenes (KFS-B: Bamboo, KFS-W: Wavescape, KFS-G: Galactic Core) with differ-

Table 1
Workload characteristics.

Benchmark	FPS	Average latency (usec)			Workload
		GP	PP	GP+PP	Type
Nenamark1	59.8	2930	2955	5885	Low
KFS-B	31.2	15,462	4900	20,362	GP-intensive
KFS-W	48.2	339	4900	5239	PP-intensive
Basemark	34.9	6132	17,397	23,529	PP-intensive
Monjori	30.0	131	24,186	24,317	PP-intensive
Popup Browser	59.7	259	1096	1355	Low

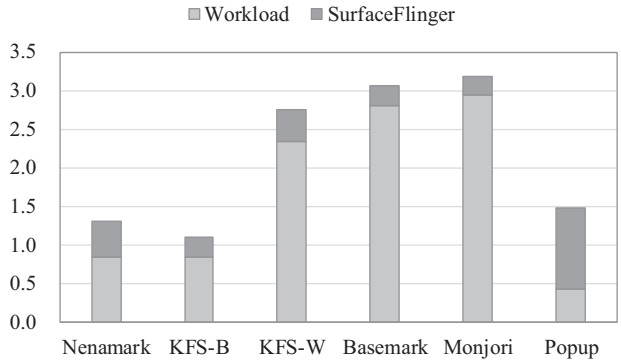


Fig. 6. Average PP utilizations of benchmark programs.

ent properties specifically designed to stress vertex throughput, fill rate, and draw calls. The basemark (Basemark ES 2.0 Taiji Free) [26] is a gaming and graphics performance measurement utility, which is a consumer-version of the mobile industry's standard graphics benchmarking product. The monjori [27] is a GLES 2.0 shader demo, which measures the raw fragment shader performance for a full screen resolution. It makes heavy use of the fragment shader. The popup browser generates GPU requests when there are user inputs for web page rendering. We used the Android's MonkeyRunner [28] in order to make user inputs automatically based on a predefined input scenario.

Table 1 shows the characteristics of each GPU application, including the frames per seconds (FPS), the average latency of one frame processing, and the workload type. The FPS value is measured while running only the target application. A lower FPS represents a higher GPU workload. The nenamark1 and popup browser are low-complexity workloads. Whereas KFS-W, basemark, and monjori are PP-intensive workloads, KFS-B is GP-intensive. Fig. 6 shows the PP utilization of each benchmark application. Since the target GPU has four PPs, the maximum PP utilization is 4. For all benchmarks, SurfaceFlinger occupies 0.5–1 of PP utilization.

The implemented BR-SRS scheduler manages the numbers of reserved and allocated PPs for two groups, high-priority group and low-priority group, and maintains the number of allocated PPs for each group so that it is not larger than the reserved budget. The high-priority group includes the popup browser and SurfaceFlinger. When the scheduler dispatches a job from the session queue, if the corresponding group has already been allocated the maximum budget, the scheduler skips the session without dispatching any jobs.

While the target GPU supports SRS (each PP can be allocated for a different GPU job), the original TBR scheduler assumes that all the PPs should be allocated for a single process at a time [9]. We implemented an SRS version of the TBR scheduler (i.e., TBR-SRS) in this work for comparison, which enables multiple GPU jobs to be executed at different GPU cores simultaneously. The original AE version of TBR predicts the execution time of a GPU job with the history table that records the GPU job execution times indexed by the sequence of GPU commands. However, it is impossible to access the GPU commands in the device driver of the target embedded GPU. Therefore, the execution time of a GPU job is predicted based only on the past execution times of GPU jobs generated by the same task in the modified TBR-SRS scheduler. The prediction accuracy of TBR-SRS is similar to that of the original TBR in the experiments. The replenishment period of TBR-SRS was set to be 20 ms.

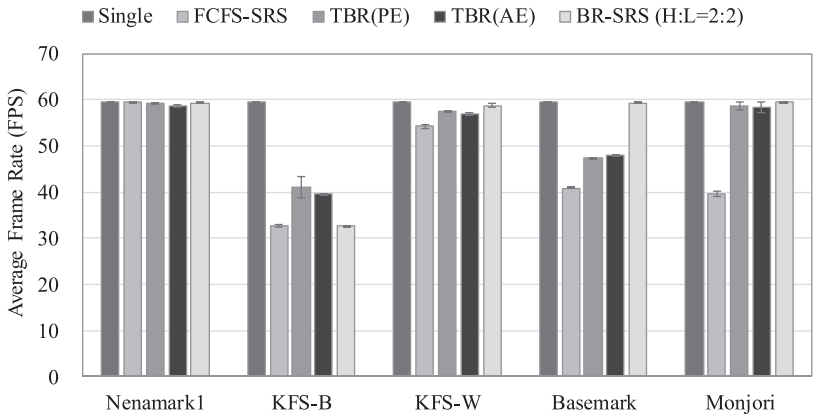
5.2. Performance comparison under different GPU schedulers

First, the performances of different GPU schedulers were compared. A foreground task (popup browser) and a background task were executed simultaneously, where the two tasks are competing for the GPU. Based on the results in Fig. 6, the PP budget for the high-priority group (popup browser and SurfaceFlinger) was set to two, and the remaining two PPs were allocated for the background tasks. Four scheduling algorithms were compared: FCFS-SRS, TBR with PE policy, TBR with AE policy, and BR-SRS. The `single` mode represents the performance of each application when it is executed alone without any other GPU tasks. The TBR schedulers are the modified SRS versions. Since the TBR allocates time budgets, both the high-priority and low-priority groups were assigned 50% of the total time budget, i.e., 10 ms for each group. Five experiments were performed for each configuration, and the average and standard deviation are shown in the following graphs.

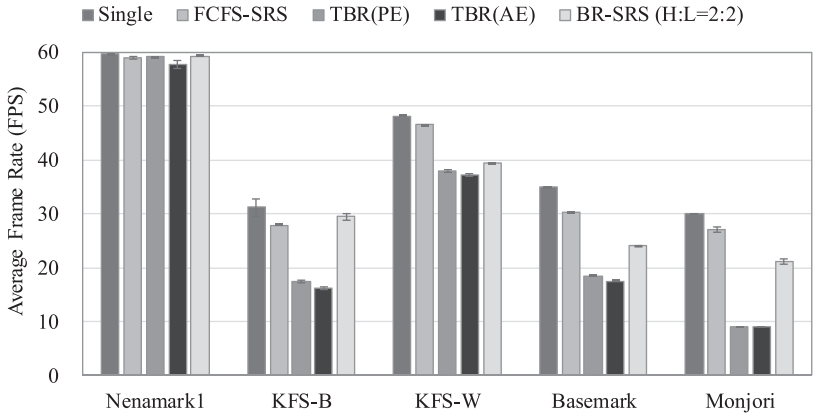
As shown in Fig. 7(a), in most of the cases, the performance of the popup browser under the BR-SRS scheduler is similar to the performance in the `single` mode. One exception is when the background task is KFS-B. Because the background task generates high-cost GP jobs, the GP becomes a bottleneck. However, the proposed BR-SRS scheduler is applied only for the PPs. Therefore, the foreground task does not achieve the maximum performance. For the background task of nenamark1, all the schedulers showed the maximum performance because the background task has a low workload. However, for other background applications, the FCFS and TBR schedulers show significant performance degradation of the foreground task. From these experiments, it can be concluded that the priority-based budget reservation of the BR-SRS scheduler is effective in isolating the performance of foreground tasks from the workload of background tasks.

The BR-SRS scheduler improves the performance of the background task also in comparison with the TBR, as shown in Fig. 7(b). This is because the background tasks can be serviced if there are available GPUs in the BR-SRS while they must wait until the budget is replenished in the TBR scheduler. From the results in Fig. 8, we can see the different PP utilizations of different schedulers while running the popup browser and the basemark application. The FCFS scheduler shows the highest PP utilization because it does not make PPs to be idle if there are pending jobs. However, the FCFS can degrade the performance of foreground task. The TBR scheduler degraded significantly the PP utilization due to its budget management technique. In particular, the PP utilization of background application was reduced compared with other schedulers. The BR-SRS shows a high total PP utilization without degrading the PP utilization of a foreground application. Therefore, the BR-SRS can provide better performances for background tasks.

Fig. 9 compares the average waiting times of the high-priority GPU jobs in its session queue when using different schedulers, because the performance improvement of foreground tasks at BR-SRS (as shown in Fig. 7(a)) results from the reduction on waiting time of the GPU jobs. The basemark were executed as a background application. The waiting time of the high-priority task is longer when two tasks are competing for a GPU resource, compared with the case when only the high-priority task is executed (single mode). In particular, the FCFS scheduler increases significantly the waiting



(a) performance of foreground task (popup browser) while running with a background application



(b) performance of background task while running with the popup browser

Fig. 7. Comparison of GPU scheduling algorithms.

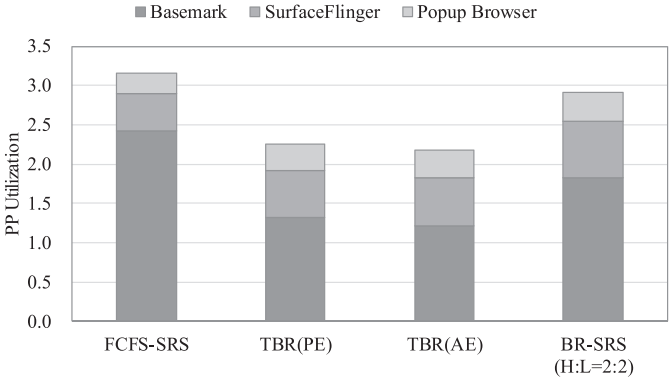


Fig. 8. PP utilizations of GPU scheduling algorithms.

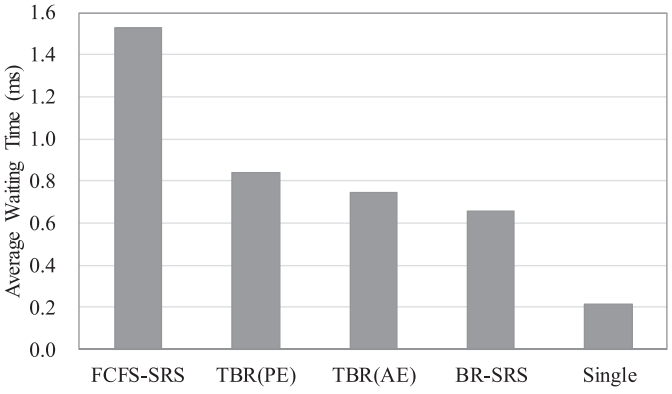


Fig. 9. The average waiting times at different schedulers.

time of the high-priority task because the background application can be scheduled to use all the PPs. Although the TBR schedulers exhibit shorter waiting times than the FCFS scheduler, the waiting times of these schedulers are longer than that of the BR-SRS scheduler. In the TBR scheduler, due to the non-preemptive GPU scheduling, a low-priority task can result in the delay of a higher-priority task if the low-priority task arrives earlier than the high-priority task. However, a higher-priority task can be scheduled immediately with the reserved GPU resources in the BR-SRS scheme.

5.3. Scheduling overhead

Fig. 10 compares the context switching and system time overheads of the BR-SRS and TBR schedulers. The system time overhead means the ratio of the time consumed by kernel over the total time. A large number of context switches will increase the system time overhead because the kernel handles the context switch operations. A high-priority task (popup browser) and a low-priority task (Basemark or KFS-W) were executed simultaneously in the two schedulers. The values were measured by the *vmstat* utility of

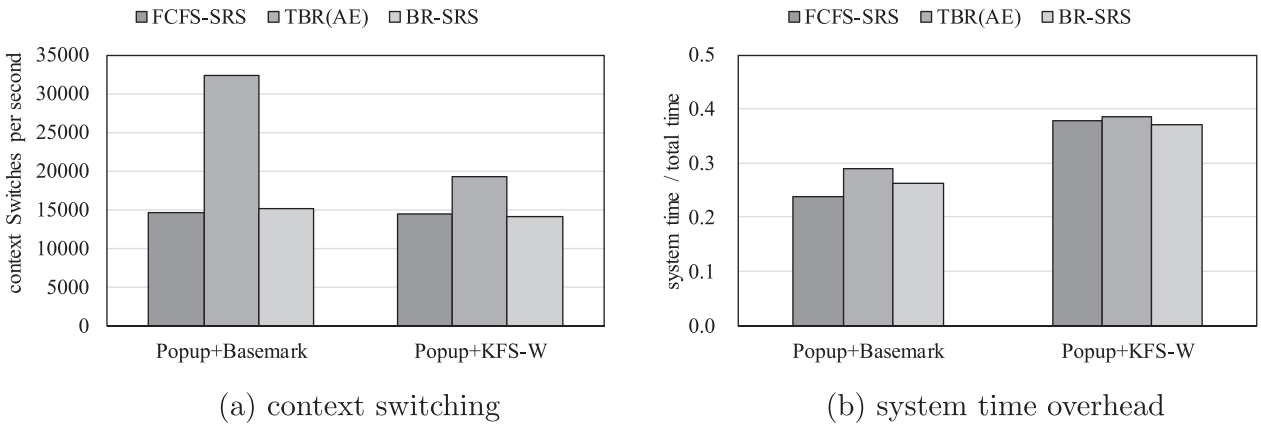


Fig. 10. Overhead comparison betwen the BR-SRS and TBR schedulers.

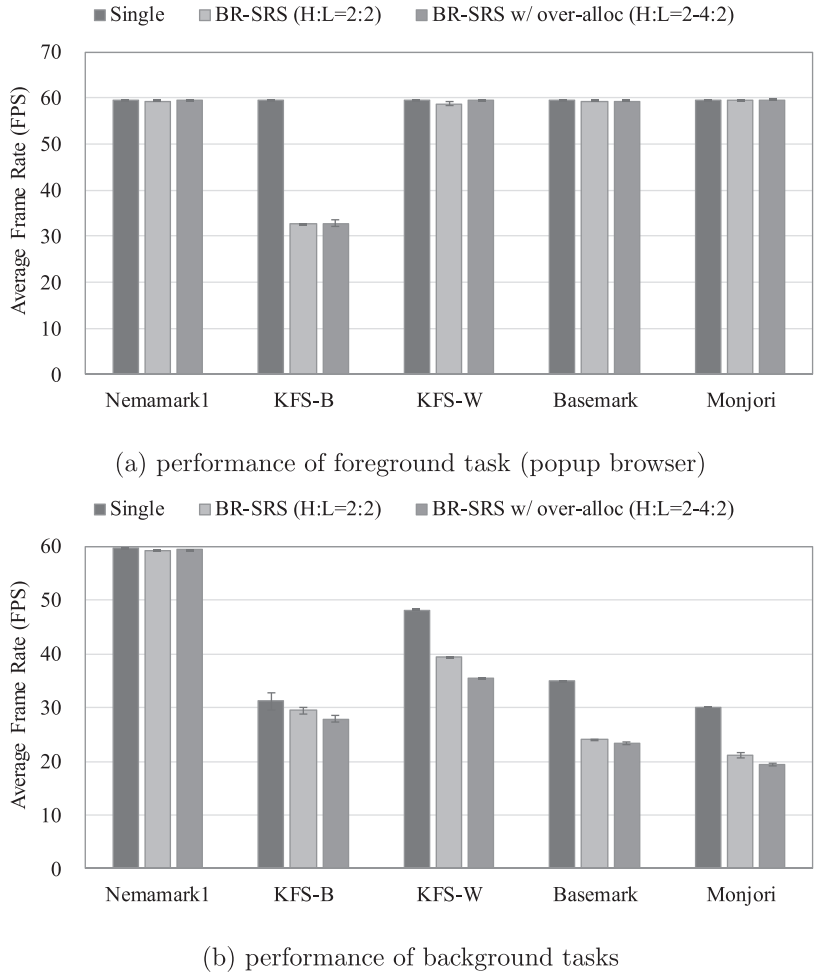


Fig. 11. Effect of over-allocation in BR-SRS.

Linux. The BR-SRS exhibits fewer context switches than the TBR scheduler as shown in Fig. 10(a). The higher context switching overhead of TBR scheduler results from the timer interrupt handling which is required to implement the time-based scheduling. The higher context switching overhead of TBR scheduler makes a larger system time overhead as shown in Fig. 10(b).

5.4. Effect of over-allocation and under-allocation in BR-SRS scheduler

As the last experiment, the effects of the over-allocation and under-allocation schemes were evaluated. The same task sets were used in the experiment. For the over-allocation, θ_{bg} is configure as 10 FPS considering the FPS values of benchmark workloads in Table 1. Therefore, if there are pending GPU jobs of background tasks and the total frame rates of them is less than 10, the over-allocation is disabled. For the under-allocation, θ_{fg} is configured as 30 FPS because the SurfaceFlinger consumes about 30 FPS without any foreground GPU applications. If the frame rate of foreground tasks including SurfaceFlinger is less than 30 FPS, the current state is an idle mode. Therefore, the under-allocation is enabled.

Fig. 11(a) shows the performance change of the foreground application by the over-allocation scheme. Even when more than two PPs are allocated for high-priority tasks by the over-allocation

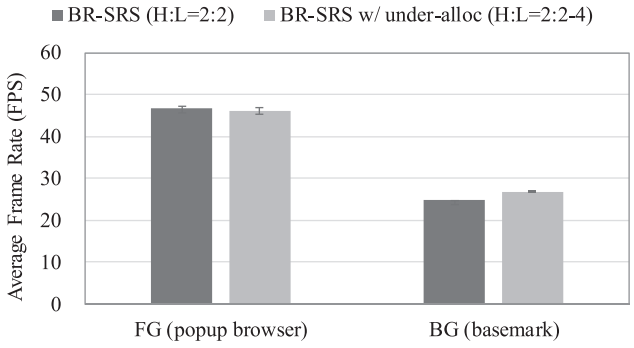


Fig. 12. Effect of under-allocation in BR-SRS.

technique, there is no significant changes on the performance of foreground application. This is because the target foreground application (i.e., popup browser) does not require more than two PPs for full performance as shown in Fig. 6.

However, the over-allocation scheme decreases the performance of background tasks slightly as shown in Fig. 11(b). Therefore, the over-allocation should be applied carefully. A proper case for over-allocation is when the workload of foreground application fluctuates significantly. For the case, even though BR-SRS reserves the GPU resource of high-priority tasks based on the average workload, the over-allocation technique can allocate more PPs for foreground application, and thus, the intermittent performance delay can be avoided.

Fig. 12 shows the performance change by the under-allocation technique. In this experiment, we used a different usage scenario of the foreground application. The burst mode and idle mode executions are repeated alternatively by giving burst touch inputs intermittently. With the under-allocation technique, the background application can use up to four PPs when the foreground application is in the idle mode. The under-allocation technique improved the performance of background application by 8.7%. However, there is no significant performance degradation on the foreground application.

Fig. 13 shows the changes of PP utilization during the experiments. In Fig. 13(a), the PP utilizations of SurfaceFlinger and popup browser decrease when there is no user input (e.g., at the time of 19 s). Irrespective of the dynamic changes on the foreground workloads, the PP utilization of the background task (i.e., basemark) has little changes in the original BR-SRS scheme because the background task cannot use more than its budget. In contrast, the PP utilization of the background application changes depending on the workload of foreground tasks in the under-allocation technique, as shown in Fig. 13(b). At the time of 19 s, the PP utilizations of SurfaceFlinger and popup browser decrease because there is no user input. Then, the under-allocation technique reduces the number of PPs reserved for the foreground tasks. As a result, the basemark task can use more than two PPs and the PP utilization increases. The increased PP utilization results in the performance improvement of background task as shown in Fig. 12.

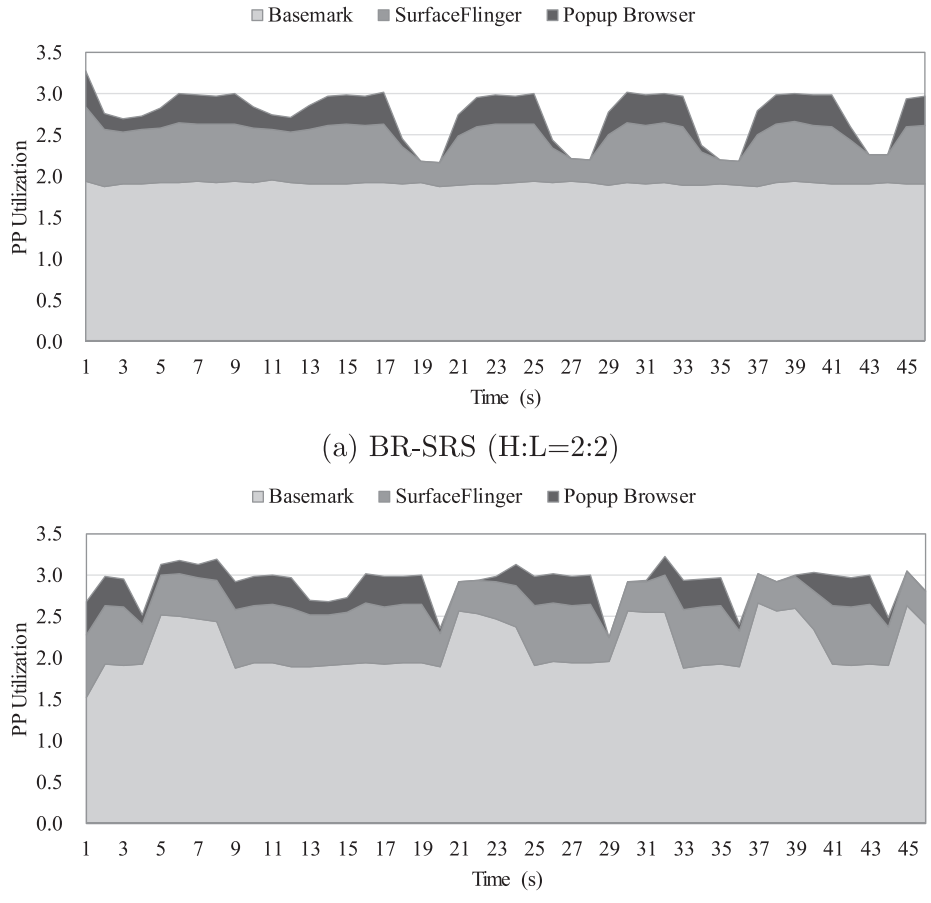


Fig. 13. Change of PP utilization by under-allocation in BR-SRS.

6. Conclusion

GPUs are indispensable to many visual applications in modern mobile devices. We presented a novel spatial multi-tasking technique for embedded GPUs considering the non-preemptive feature of GPU tasks. Unlike the previous temporal budget reservation techniques, our spatial budget reservation policy improved the performance of high-priority GPU tasks by reserving the GPU processing cores, while reducing the scheduling overhead as well. As a future work, we plan to devise a spatial and temporal scheduling technique, which can manage both the spatial and temporal budgets.

Acknowledgments

This research was supported by the MSIP, Korea, under the G-ITRC support program (IITP-2016-R6812-16-0001) supervised by the IITP.

References

[1] B.-G. Nam, M.-w. Lee, H.-J. Yoo, Development of a 3-D graphics rendering engine with lighting acceleration for handheld multimedia systems, IEEE Trans. Consumer Electronics 51 (3) (2005) 1020–1027, doi:10.1109/TCE.2005.1510517.

[2] L.-B. Chen, R.-T. Gu, W.-S. Huang, C.-C. Wang, W.-C. Shiue, T.-Y. Ho, Y.-N. Chang, S.-F. Hsiao, C.-N. Lee, I.-J. Huang, An 8.69 Mvertices/s 278 Mpixels/s tile-based 3D graphics SoC HW/SW development for consumer electronics, in: Proc. Asia and South Pacific Design Automation Conference, IEEE Press, 2009, pp. 131–132, doi:10.1109/ASPDAC.2009.4796467.

[3] T. Olson, Mali-400 MP: a scalable GPU for mobile devices, in: Proc. International Conference on High Performance Graphics, 2010.

[4] K.-T. Cheng, Y.-C. Wang, Using mobile GPU for general-purpose computing–a case study of face recognition on smartphones, in: Proc. VLSI Design, Automation and Test, IEEE, 2011, pp. 1–4, doi:10.1109/VDAT.2011.5783575.

[5] M.B. López, H. Nykänen, J. Hannuksela, O. Silvén, M. Vehviläinen, Accelerating image recognition on mobile devices using GPGPU, in: Proc. IS&T/SPIE Electronic Imaging, International Society for Optics and Photonics, 2011, 78720R–78720R, doi:10.1117/12.872860.

[6] C. Cuevas, D. Berjón, F. Morán, N. García, Moving object detection for real-time augmented reality applications in a GPGPU, IEEE Trans. Consumer Electronics 58 (1) (2012) 117–125, doi:10.1109/TCE.2012.6170063.

[7] G. Wang, Y. Xiong, J. Yun, J.R. Cavallaro, Accelerating computer vision algorithms using OpenCL framework on the mobile GPU-a case study, in: Proc. Acoustics, Speech and Signal Processing, IEEE, 2013, pp. 2629–2633, doi:10.1109/ICASSP.2013.6638132.

[8] I. Bratt, The ARM Mali-T880 mobile GPU, in: Hot Chips 27 Symposium (HCS), 2015 IEEE, 2015, pp. 1–27.

[9] S. Kato, K. Lakshmanan, R. Rajkumar, Y. Ishikawa, TimeGraph: GPU scheduling for real-time multi-tasking environments, in: Proc. USENIX Annual Technical Conference, 2011, p. 17.

[10] J.T. Adriaens, K. Compton, N.S. Kim, M.J. Schulte, The case for GPGPU spatial multitasking, in: Proc. High Performance Computer Architecture, IEEE, 2012, pp. 1–12, doi:10.1109/HPCA.2012.6168946.

[11] P. Aguilera, K. Morrow, N.S. Kim, QoS-aware dynamic resource allocation for spatial-multitasking GPUs, in: 2014 19th Asia and South Pacific Design Automation Conference (ASP-DAC), IEEE, 2014, pp. 726–731, doi:10.1109/ASPDAC.2014.6742976.

[12] I. Tanasic, I. Gelado, J. Cabezas, A. Ramirez, N. Navarro, M. Valero, Enabling preemptive multiprogramming on GPUs, in: Proc. ACM/IEEE Int. Symp. Computer Architecture (ISCA), IEEE Press, 2014, pp. 193–204.

[13] Y. Liang, H.P. Huynh, K. Rupnow, R.S.M. Goh, D. Chen, Efficient GPU spatial-temporal multitasking, IEEE Trans. Parallel Distrib. Syst. 26 (3) (2015) 748–760, doi:10.1109/TPDS.2014.2313342.

[14] Z. Wang, J. Yang, R. Melhem, B. Childers, Y. Zhang, M. Guo, Simultaneous multikernel GPU: multi-tasking throughput processors via fine-grained sharing, in: 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA), IEEE, 2016, pp. 358–369, doi:10.1109/HPCA.2016.7446078.

[15] Android, SurfaceFlinger and hardware composer, https://source.android.com/devices/graphics/arch-sf-hwc.html.

[16] W. Joo, D. Shin, Resource-constrained spatial multi-tasking for embedded GPU, in: Proc. Consumer Electronics, IEEE, 2014, pp. 339–340, doi:10.1109/ICCE.2014.6776031.

[17] C. Basaran, K.-D. Kang, Supporting preemptive task executions and memory copies in GPGPUs, in: Proc. Euromicro Conference on Real-Time Systems, IEEE, 2012, pp. 287–296.

[18] J.J.K. Park, Y. Park, S. Mahlke, Chimera: collaborative preemption for multitasking on a shared GPU, in: ACM Int. Conf. Architecture Support for Programming Languages and Operating Systems (ASPLOS), ACM, 2015, pp. 593–606.

[19] H. Zhou, G. Tong, C. Liu, GPES: a preemptive execution system for GPGPU computing, in: Proc. IEEE Real-Time and Embedded Technology and Applications Symposium, IEEE, 2015, pp. 87–97.

[20] Y. Suzuki, H. Yamada, S. Kato, K. Kono, Towards multi-tenant GPGPU: event-driven programming model for system-wide scheduling on shared GPUs (2016) 1–7.

[21] S. Kato, K. Lakshmanan, A. Kumar, M. Kelkar, Y. Ishikawa, R. Rajkumar, Rgem: a responsive GPGPU execution model for runtime engines, in: Proc. IEEE Real–Time Systems Symposium (RTSS), IEEE, 2011, pp. 57–66.

[22] NVIDIA, White paper: NVIDIA Tesla P100, Technical Report WP-08019-001_v0.1, NVIDIA, 2016.

[23] GPCSOFT, Popup browser, https://play.google.com/store/apps/details?id=gpc.myweb.hinet.net.PopupWeb.

[24] Nena Innovation AB, Nenamark1, https://play.google.com/store/apps/details?id=se.nena.nenamark1.

[25] Kittehface Software, KFS OpenGL benchmark, https://play.google.com/store/apps/details?id=fishnoodle.benchmark.

[26] Rightware, Basemark ES 2.0 Taiji free, https://play.google.com/store/apps/details?id=com.rightware.tdmm2v10jnifree.

[27] S. Wagner, Monjori shader benchmark, https://play.google.com/store/apps/details?id=de.swagner.monjori.

[28] Android, MonkeyRunner, https://developer.android.com/studio/test/monkeyrunner/MonkeyRunner.html.

Yunji Kang received the BS and MS degrees in computer engineering from Sungkyunkwan University, Korea, in 2013 and 2015, respectively. She is currently a PhD student in the IT Convergence Department at Sungkyunkwan University. Her research interests included embedded software, file systems, flash memory, and mobile platform.

Woohyun Joo received the BS degree in computer science and engineering from Korea University, Korea, in 2005, and the MS degree in digital media and communications engineering from Sungkyunkwan University, Korea, in 2013. In 2005, he joined Samsung Electronics Co., Korea, and is a senior engineer. His research interests include embedded software, graphic systems, and digital signal processing.

Sungkil Lee received the BS and PhD degrees in materials science and engineering and computer science and engineering at POSTECH, Korea, in 2002 and 2009, respectively. He is currently an associate professor in the Department of Computer Science and Engineering at Sungkyunkwan University, Korea. He was a postdoctoral researcher at the Max-Planck-Institut Informatik (2009–2011). His research interests include real-time GPU rendering, perception-based rendering, information visualization, GPU algorithms, and human-computer interaction.

Dongkun Shin received the BS degree in computer science and statistics, the MS degree in computer science, and the PhD degree in computer science and engineering from Seoul National University, Korea, in 1994, 2000, and 2004, respectively. He is currently an associate professor in the Department of Computer Science and Engineering at Sungkyunkwan University, Korea. He was a senior engineer of Samsung Electronics Co., Korea (2004–2007). His research interests include embedded software, low-power systems, computer architecture, and real-time systems.