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/

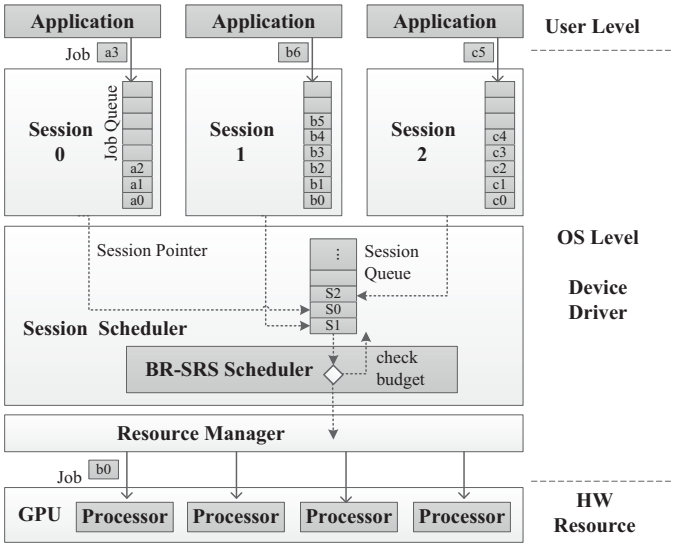

Fig. 1. The architecture of a GPU device driver.

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,