Coordinating CPU an GPU Resources with Loadable Real-Time Schedulers

Yusuke Fujii, *Member, IEEE,* Takuya Azumi, *Member, IEEE,*
Nobuhiko Nishio, *Member, IEEE,* Tsuyoshi Hamada, *Member, IEEE,*
Shinpei Kato, *Member, IEEE,*

Abstract—Graphics processing units (GPUs) easily provide the benefits of high-performance computing. However, GPUs runtime environment is target to best-effort oriented application, these does not consider real-time. Previous work have contributed for real-time GPU, these have depends kernel. Dependence on the kernel and the device driver gives the burden developers and users. We present Linux-RTXG which is realized cpu-gpu realtime extending framework while kernel free. Linux-RTXG have a CPU real-time scheduler, a GPU real-time scheduler and a GPU reservation mechanisms. To achieve kernel free framework, we presents interrupt intercept mechanisms and independent interrupt mechanisms and to solved currenly linux kernel's real-time scheduling problem. Our experimental result, we indicate overhead of GPU scheduling using Linux-RTXG framework, and then, kernel free's approach performances of qos management had kept equivalent performance as compared with the existing kernel-dependent approach.

Keywords—Computer Society, IEEEtran, journal, LATEX, paper, template.

✦

1 INTRODUCTION

GPU is become common as a device to accelerate the general purpose application. Its range of application that are navigation[1] for autonomous drive, object detection[2], tokamak control for fusion reactor[3], user-interactive application[4] and databases[5]. And benchmarks[6] that contains a lot of applications has been provided. GPUs performance has been demonstrated by these research.

The past GPU applications shoud have been only "real-fast" since these were best-effort oriented. The recent GPU applications are required "real-time" and "real-fast" by the increasing real-time oriented application of targetting real-world. GPUs runtime environments such as CUDA[7] and OpenCL[8] are

provided by the vendor, however, these runtime environments made to target a best-effort applications, it is not support real-time. Therefore GPU runtime environments are required to have to support real-time.

We have already worked several research of GPU resource management. TimeGraph[9] provides GPU scheeduling and reservation mechanisms at the device driver level to queue and dispatch GPU commands based on task priorities. RGEM[10] provides GPU kernel scheduling and data transfering scheduling, Gdev[11] is applied resource reservation. These work have weak point that is can not to provide fast supporting update architecture and full-funciton because these works is based on reverse engineering.

Some GPU functions are provided API, processing is issued to GPU via library and device driver from the user application. Thus if GPUs truly wants the real-time requirements, there is a need to manage host side task as well as GPU resource management. We have confirmed the fact that a large amount of latency occur, when other tasks appropriative rsources in the host side at studies[12] is evaluating the data trans-

- *Y. Fujii is with the Graduate School of Information Science and Engineering, Ritsumeikan University.*
- *T. Azumi is with the Graduate School of Information Science and Engineering, Osaka University.*
- *N. Nishio is with the College of Information Science and Engineering, Ritsumeikan University.*
- *T. Hamada is with the Advanced Computing Center, University of Nagasaki.*
- *S. Kato is with the School of Information Science, Nagoya University.*

fer time between the host and the device.

GPUSync[13], [14] by Elliot et al. providing CPU task scheduling and budget enforcement on the proprietary runtime, and it is realized configurable framework in order to verify the combination of the policies of tasks allocation to multi-core cpu and the policies of GPU kernel allocation to multiple GPUs.

However, GPUSync is implemented on the $LITMUS^{RT}$[15], it contains a large amount of changes to the kernel. Gdev also request a modification to the device driver. Many of these modifications are to required installation using patch to user. A big burden is given patch to both developer and user. Specifically, developers oblicgation that is maintenance of patch in order to catch up to the latest kernel release. However, software which is using basis in open source such as Linux are updated fast, in most cases, before developers to complete the porting work towards the latest kernel.

Linux have mechanism which is loadable kernel module (LKM). LKM is module that is able to load/unload between running for providing function foreign kernel. We work the real-time extension by LKM, is called RESCH. RESCH is providing real-time scheduling framework while it not modify the kernel and device drivers. RESCH is not support GPU resource.

Contribution: In this paper, we present linux real-time extension for cpu-gpu resouce coordination called linux-rtxg for cpu and gpu coordinated resource management, this extension is able to more easily re-configure the resource management policy and the installation. Linux-RTXG's most contribution is to achieve real-time task scheduling on the using GPU environment while not kernel modification.

To achieve real-time task scheduling, Linux-RTXG provide the CPU task scheduler, GPU kernel scheduler, GPU kernel reservation mechanisms. TODO:moutyotto

Organization This rest of the paper is organized as follows. Section 2 discuss the GPU real-time scheduling by kernel free aproach. Section 3 shows Linux-RTXG design and implementations, especially focus on GPU scheduling. Section **??** indicate Linux-RTXG's adovantages and dis-advantages, furthermore, demon-strate experimental results that are quantitative overheads and reservation performances. Section 5 discusses related work. We provide our concluding remarks in Section 6.

2 SYSTEM MODEL

In this section, we explain GPU task model on this paper while we discuss the gpu scheduling question and prior work. Next, we make available limitation for clearing the implementation of no patched gpu scheduling. This paper focus on a system composed of multiple GPU and multi-core CPU.

2.1 GPU Task Model

When using GPU in general-purpose way, CUDA and OpenCL can use much. In this paper, we only use CUDA, but it is possible the same approach even OpenCL.

GPU applications use a set of the API supported by the system, typically talking the following steps. (i) create GPU context (*cuCtx-Create*), (ii) allocate space to device memory (*cuMemAlloc*), (iii) The data and the GPU kernel are copied to the allocated device memory space from host memory (*cuMemcpyHtoD*), (iv) Launch the GPU kernele (*cuLaunchGrid*), (v) The GPU task is synchronized to the GPU kernel (*cuCtxSynchronize*), (vi) The resultant data transfer to host memory from device memory (*cuMemcpyDtoH*), (vii) release allocated memory space and context (*cuMemFree, cuModuleUnload, cuCtxDestroy*).

We define a GPU task which is to execute the GPU using above API flow, and define a GPU kernel that is unit of processing to executed on the GPU side.

2.2 GPU Scheduling

Real-time OS (RTOS) is a lot of research[16], [17], [18], [19] has been conducted for a long time. In among them have also been many studies[20], [21], [22], [15], [**?**] real-time OS, which is based on the Linux. OS available the GPU is limited to Windows, Mac OS and Linux, we selected Linux in order to achieve real-time on GPU environments.

Requirement that the scheduler the minimum required in carrying to realize the real-time multi-core environment is the following two:

- To use resources according to order.
- To limit the use of the shared resource

The basic approach to satisfy the first is priority-based scheduling (e.g. Rate-Monotonic[23] and Earliest Deadline First[24]) with technique to prevent priority inversion, and the second is resource reservation based scheduling (e.g. Constant Bandwidth Server[25], Total Bandwidth Server[26]). The GPU to handle the data transfer bandwidth and the processing core as a shared resource, it is necessary to satisfy the two requirements similar to above multicore environment. Our previous working only to scheduling GPU accesses. However, GPU kernel is driven by API is issued GPU task, to truly support the real-time scheduling, it is need to schedule GPU task; therefore, for realized real-time GPU, framework need to have CPU's priority-based scheduler, GPU's priority-based schedulera and restrict GPU resource by resource reservation mechanisms. Recently, GPUSync is to work the above interdependence of CPU scheduler and GPU scheduler.

GPUs have some problem on using realtime, except that scheduling mechanisms. GPUs run-time environments are blackbox mechanisms results from GPU environments are provided only GPU vendors and these environments are closed-source. TimeGraph, Gdev and RGEM are solved this problem by ensuring transparency using reverse engineering and open-source driver. GPUSync is solved this problem too by GPU runtime resource management is disabled through the runtime access to single.

The other problem occurs by non-preemptive GPU executions and non-preemptive data transfers. several researchs[27], [28] realize improvement the responce time by preventing divides the kernel overrun.

As these core problems are headling to solve, however, real-time GPU is experimental-phase since have problem for practical realized. The most difficult problem is self-supension due to GPU is treated as a I/O device. GPU task to suspended themselfs until it receives the results from issuing the processing to GPU, referred to as self-suspension. The self-suspension has been proven as a cause the NP-HARD problem in previous work[29], [30], several researchs[31], [32] are working on the scheduling analysis for self-suspension task, but it has not been solved yet to complete.

Hence scheduling framework for the expansion and installation to aim in this study and easy is useful, it is argued that there is a demand.

GPU Synchronization: The synchronization matters for GPU system such as heterogeneous platoform. GPU have two different synchronization techniques. The first techniques is memory map based synchronization which is called FENCE, it sends GPU commands after the command to take action, then GPUs microcontrollers will write the any value to memory-mapped space after action is completed. GPU task monitors it mapped space value using such as polling, therefore task has an exclusive CPU resource, but response time will be the fastest. The other one techniques is interrupts based synchronization which is called NOTIFY, it sends GPU commands similar to FENCE, then GPUs microcontrollers will rise the interrupt and write any value to GPU I/O registers. GPU task is suspending until interrupt, therefore task is able to share the CPU resources with other tasks, but response time will be the slow. Detailed architecture is omitted in this paper, it has been described in the previous documents[**?**], [11], [33].

Gdev use both techniques that are NOTIFY and FENCE for wakeup the waiting task, NOTIFY is used by scheduler, FENCE is used by kernel synchronization. Gdev ' s synchronization implementation is the addtional commands sends to GPU and the modification of device driver ' s interrupt handler. GPUSync use NOTIFY techniques by using tasklet intercept[34] on the proprietary software. Tasklet is linux' s soft-irq implementation. GPUSync identify the interrupt that is issued which kernel by callback pointer with a tasklet.

kernel free scheduler: To achieve " kernel free " , we must not to modify the kernel code and the device driver code. GPU scheduler is required to receive notify the completion

of GPU kernel, for selecton of next executing GPU kernel. It is realized by two methods that are API-driven method and Interrupt-driven method. The API-driven method is explicitly wakeup the scheduler after the synchronized by API such as $cuCtxSynchronize()$ is provided runtime in the RGEM. The Interrupt-driven method is wakeup by trigger of the interrupt which is issued by using NOTIFY in the Gdev and TimeGraph. General utilized $cuCtxSynchronize()$ synchronize completion of all GPU kernels, therefore, API driven is able to use when a GPU context have issued only single kernel. Thus if a GPU context issue multiple kernels, we must use the Interrupt-driven method for not to guarantee overrun due to reduction of the response time.

The Interrupt-driven method can be synchronized for each kernel, it is required to modificate the kernel or the device driver's ISR. Gdev has been achieved independs synchronization mechanisms on the proprietary software, need to modify the kernel modification. The challenge is realize independs synchronization mechanisms by kernel free aproach.

3 DESIGN AND IMPLEMENTATION

In this section, we present Linux-RTXG design and implementation. Linux-RTXG is an abbreviation of Linux Real-Time eXtension including GPU resouce management, which is no patched Linux real-time gpu scheduling framework.

We describe main contribution of the GPU scheduler and the integration to CPU scheduler in this paper, while CPU scheduling description is to minimize by Linux-RTXG is base RESCH.

3.1 Linux-RTXG

Figure 3.1 shows over-view of Linux-RTXG. Linux-RTXG is divided into two components that are loadable kernel module and library. Linux-RTXG library is interface of communicate between application and Linux-RTXG core component(kernel module). it is using system call that is ioctl.

The part of library included speciall method that is independent synchronization method. it method is used only on the nvidia driver.
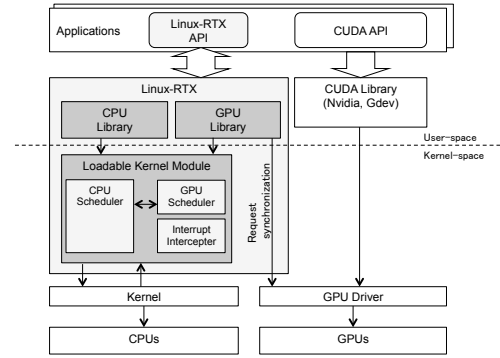

Fig. 1. Over view of the Linux-RTXG

if system use nouveau driver, runtime must use part of gdev. Gdev can happen arbitary interrupt of gpu kernel in the user-space mode, and it have no need to be independent interrupt raised method.

Linux-RTXG loadable kernel module is positioned kernel-space. Thus, module can use kernel exported function.

3.2 GPU Scheduling

Linux-RTXG is API-driven where the scheduler invoked only when computation requests are submitted. The basic APIs supported by Linux-RTXG are listed in Table 1. Some APIs have arguments and others do not. Linux-RTXG APIs are not modified existing CUDA API to cope with proprietary software to be independent from the runtime. However, user have to add Linux-RTXG api to existing CUDA application for using Linux-RTXG scheduler.

The sample code of the using Linux-RTXG scheduler is shown in Figure 2, and to some extent omitted except GPU scheduling. The $gpu_task()$ is

Figure 3.2 shows control flow of run the Figure2's sample code. The configure is the Kernel issue that is restricted to single kernel. User task (GPU Task) can be control the timing of GPU kernel execution by called $rtx_gpu_launch()$. Task goes to sleep until the wakeup by interrupt why task is not permitted issuance of GPU kernel due to already execute other task in the GPU.

Once issued GPU kernel is finished, interrupt is awaken while interrupt intercepter wakeup the GPU scheduler, GPU scheduler wakeup the

TABLE 1
Basic Linux-RTXG APIs

rtx_gpu_open()	To register itself to Linux-RTXG, and create scheduling entity. It will must call first.
rtx_gpu_device_advice()	To get the recommendation of GPU devices to be used
rtx_gpu_launch()	To control the GPU kernel launch timing, in other words it is scheduling entry point. It will must call before the CUDA launch API.
rtx_gpu_sync()	To wait for finishing GPU kernel execution by sleeping with TASK UNINTERRUPTIBLE status.
rtx_gpu_notify()	To send the notify/fence command to GPU microcontroller. The fence or the notify is selected flag is set by argument.
rtx_gpu_close()	To release scheduling entity.

```
void gpu_task(){

  /* variable initialization  */

  /* calling RESCH API */

   dev_id = rtx_device_advice(dev_id);

   cuDeviceGet(&dev, dev\_id);

   cuCtxCreate(&ctx, SYNC_FLAG, dev);

   rtx_gpu_open(&handle, vdev_id);

  /* Module load and set kernel function */

  /* Device memory allocation        */

  /* Memory copy to device from host */

   rtx_gpu_launch(&handle);

   cuLaunchGrid(function, grid_x, grid_y);

   rtx_gpu_notify(&handle);

   rtx_gpu_sync(&handle);

  /* Memory copy to host from device */

  /* Release allicated memory */

 }
```

Fig. 2. sample code of using rtxg scheduler
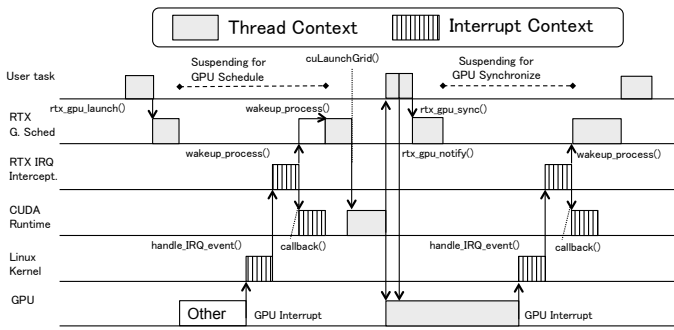
Fig. 3. GPU Scheduling control flow

sleeping task. The wake-up task issue the GPU kernel via CUDA API such as $cuLaunchGrid()$. After the GPU kernel issued, task register the NOTIFY for occurring the interrupt, and task to sleep until it occurs interrupt. To pick up the next task is performed by the GPU scheduler

caused by interruption of GPU kernel finish. Linux-RTXG is doing the execution order control tasks in the above flow.

We present hierarchal scheduling which are group scheduling, GPU kernel scheduling. The group scheduling is a using resource reservation mechanism. The Context scheduling is a priority scheduling. Specifically, GPU kernel execution is associated to each scheduling entity while Linux-RTXG grouped the scheduling entity to VGPUs, these VGPUs belong to any of physical GPUs. In Linux-RTXG, resources are distributed in this group.

Figure 4 shows pseudo-code of scheduling mechanism. $on_arrival$ is called when the GPU task is requested GPU kernel launch issue. In $on_arrival$, GPU task to check whether the given execute permission to group of task itself, and then, check the se permit. If it has not executing permittion, GPU task is enqueued to wait_queue and go to sleep, contrary to this, if it has executing permittion, GPU task go to launch issue.

$on_completion$ is called by the scheduler therad, when the GPU kernel is completion. In $on_completion$, scheduler thread pick up the next group,

3.3 GPU synchronization

Section 2   runtime environment
                                    interrupt-driven
                                          kernel free
                      interrupt   interrupt intercept

Linux-RTXG is use the independent synchronization mechanisms as much as possible, it because we do not want using black-box resource management for to realized truly real-time resource environments.

```
se: The scheduling entity
se->vgpu: The group that is belonged se
se->task: The task that is associated with se
vgpu->parent: The physical GPU identification
```
```
void on_arrival(se) {
check_permit_vgpu(se->vgpu)
while(!check_permit_se(se)){
    enqueue(se->vgpu,se);
    sleep_task(se->task);
}
}
void on_completion(se) {
reset_the_permit(se->vgpu, se)
n_vgpu = pick_up_the_next_vgpu(se->vgpu->parent)
se = pick_up_the_next_se(n_vgpu)
if(se) {
    dequeue(se->vgpu,se);
    wakeup_task(se->task);
}
set_the_permit(se->vgpu, se)
}
```

Fig. 4. High Level Pseudo-code of scheduling mechanisms

Independent synchronization mechanism from runtime We present independent synchronization of NOTIFY and FENCE, these are the to occur interrupt for NOTIFY and the write the fence value by microcontroller. NVIDIA's proprietary software use ioctl interface to communication between kernel-space and user-space. These ioctl interfaces are provided drivers function such as device memory allocation, get the gpu information and memory mapping. Gdev build infrastructure that is able to execute on the NVIDIA's driver using these ioctl interfaces.

We also use this ioctl interface similar to Gdev's command sending method for our method. Specifically, our methods are two divided, the one is Initialize, the other one is Notify. Initialize processes for generating a context dedicated this method. These processes are including the create virtual address space and the allocate indirect buffer object for command sending and the create context object. The indirect buffer is an area for storing GPU commands

GenSign processes send commands to the compute engine or the copy engine that are This independent generate sign for synchronization method is using reverse engineering. However, It is the limit of implementation using the closed-source runtime environment.

Interrupt interception: Interrupts are handled by the ISR (Interrupt Service Routine) that is registered kernel by the device driver. In addition, scheduler require to identify the interrupt by using readling GPU status register, it must be done before original ISR is reset the GPU status register.

Linux kernel have structure that holds the interrupt parameters called irq_desc for each interrupt number. These structures have structures called irq_action including the ISR callback pointer. irq_desc is allocated to global memory space of the kernel, anyone is accessible from kernel space. Linux loadable kernel modules can get an irq_desc for running in kernel, while also can get an callback pointer of ISR. We retain getting callback pointer of GPU device driver's ISR, and then we register interrupt intercept ISRs to kernel. So, we get the to intercept interrupt by it ISR and then call retaining callback pointer, In addition, I/O registers are mapped to kernel memory space by device driver from the PCIe base address registers (BARs)[12], [35]. Therefore Linux-RTXG remap the BAR0 to our allocated space by using $ioremap()$ when the ISR initizalize. The interrupt intercepter identify interrupt by read the mapped-space.

3.4 Scheduler Integration

Linux scheduler have various real-time scheduling policies that were SCHED_DEADLINE, SCHED_FIFO and SCHED_RR. , SCHED_DEADLINE is implementation the Constant Bandwidth Server and Global Earliest Deadline First, while it is including mainline of Linux 3.14.0 kernel. However, synchronization does not work well in a SCHED_DEADLINE scheduling policy when using GPU tasks.

This problems are twofold. The first is implementation of sched_yield—in kernel space used yield()—. The second is implementation of return from sleeping state.

The first problem occurs by releasing the CPU using sched_yield() when waiting for I/O in polling. Polling (Spin) is the exclusive CPU, therefore task may once better to release the CPU can obtain good results. However, sched_yield will set 0 to polling task's runtime of remaining execution time treated as a parameter of SCHED_DEADLINE. Thereby, it task lose execute authority until runtime is replenished in the next period, therefore task is unable to call sched_yield between polling. sched_yield is used much by device drivers and library as well as GPU runtime. These software is affected by this problem. Even NVIDIA CUDA is affected depending on the setting. We support this problem by limit the GPU synchronization method to NOTIFY in the SCHED_DEADLINE policies.

The second problem is subjected to a check equation (1) when restore task from sleeping state. If equation (1) is true, runtime is replenished and absolute deadline is setted next cycle deadline.

$$\frac{Absolute\_Deadline - Current\_Time}{Remaining\_Runtime} > \frac{Relative\_Deadline}{Period} \quad (1)$$

We corresponding to this check by subtracting the GPU execution time from $Remaining\_Runtime$ when task is restored by GPU kernel execution with the exception of the task is restored by period.

4 EVALUATION

We evaluate scheduling overhead and qos management performances.

GPUSync

Linux-RTXG

discuss

4.1 Experimental Environment

Our experiments are conducted with the Linux kernel 3.16.0 on NVIDIA Geforce GTX680
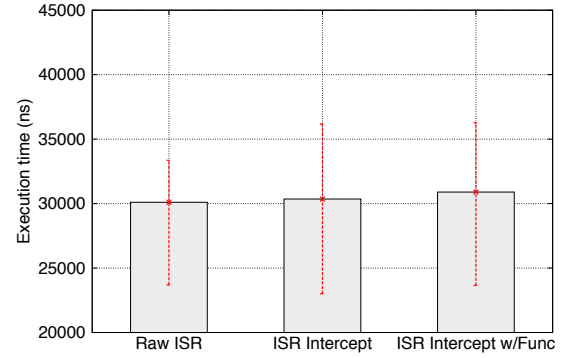
Fig. 5. Interrupt intercept overhead

graphics card and 3.40GHz Intel Core i7 2600, which contains 8 cores (including the two hyper-threading cores) and 8GB main memory. GPU programs are written in CUDA and compiled by NVCC v6.0.1. GPU drivers are used NVIDIA driver 331.62 and Nouveau driver linux-3.16.0. CUDA libraries are used NVIDIA CUDA-6.0 and gdev.

4.2 Interrupt intercept overhead

Interrupt intercept

GPU   nouveau

Nouveau

do_IRQ

Figure **??**
Raw ISR   ISR
ISR Intercept   ISR
intercept w/Func

1000

ISR Intercept   247ns
ISR Intercept w/Func   790ns

(a) Part of Initialize　　(b) Part of notify
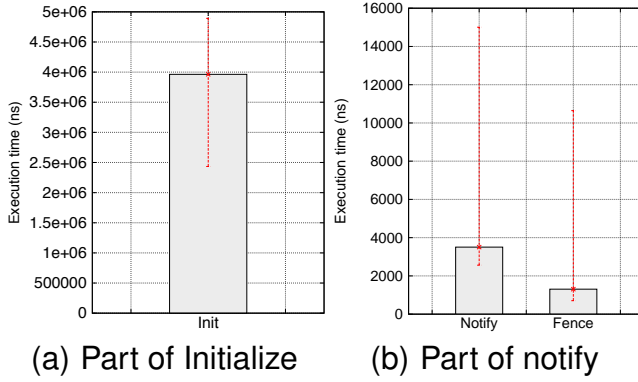
Fig. 6. Interrupt raised method overhead

Fig. 7. Scheduling overhead(between GPU kernel launch request and synchronization)

4.3 Independent generate sign for Synchronization overhead

(e.g.　　　　　　　　)
rtx_nvrm_notify()
　　　　　　　　　　　Vanilla
　API
　API

　　API
　　　Figure 6　　　　Initialize　Indirect
Buffer
　　　Indirect Buffer
　　　　　　　　　　　　　Notify

　　　　　　　　ioctl

　Initialize　　　　　　　　　　1

　　　　　　　　　　　　　Notify

　　Fence　　　　Notify
2000ns

4.4 Scheduling Overhead

rtx
　　　　　　　"vanilla", "mutex", "rtx"

　　　　　1

　　　　　　　1　　　　　GPU
　　　　GPU　　　　　GPU
　　　　　　GPU

　　　　　　　　　mutex
launch　　　　　1
mutex
　rtx　linxu-rtxg
　　vanilla

　CPU　　　　　　linux-rtx
　　　Fixed-priority　　　(Linux
　SCHED_FIFO
　　　　)　　GPU
　　Gdev　　　BAND
Linux-RTX　　　　NOTIFY

　　　　Figure 4.4

　TODO:
　launch_advice　rtx_gpu_launch
GPU

Fig. 8. Scheduling overhead (Time of entire task)

get_mutex   mutex
launch

notify

sync

100

$number of tasks$

4.5 QoS Management

GPU

QoS   Gdev

Linux-RTXG

Gdev

BAND

4.4

Task

25%   GPU
GPU

Gdev

Linux   RTXG
QoS

Gdev

Nouveau
.

Figure 4.5,4.5 show gpu usage on the qos management by gdev and linux-rtxg. TODO:
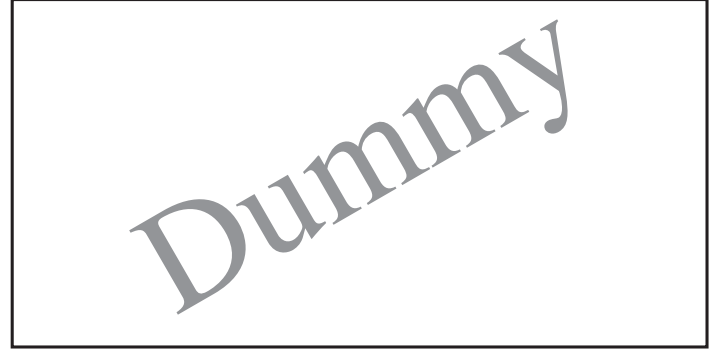
Figure ?? shows

Fig. 9. Utilization of four tasks on the Linux-RTXG only priority scheduling. Each tasks are given different priorities.
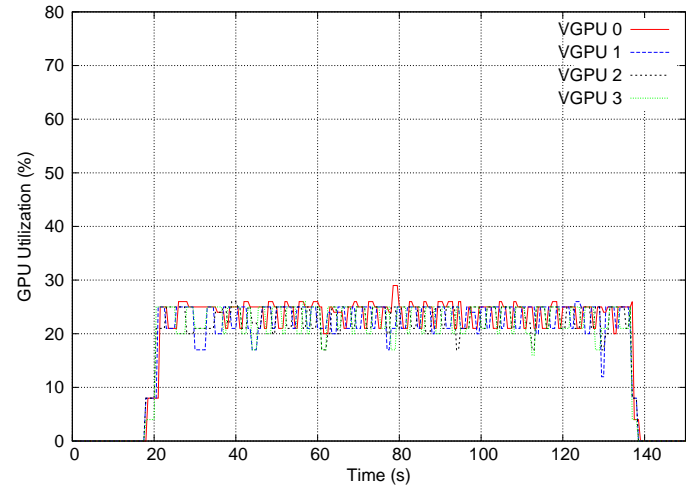

Fig. 10. Utilization of four tasks on the Linux-RTXG resource reservation. Each tasks are given different priorities and all tasks are fairly given resources which is 25%

5 RELATED WORK

Comparison of prior work

GPU   2014

172   [1]

GPU

[9], [4], [10], [11]   Time-
Graph   GPU

CUDA   OpenGL
GPU

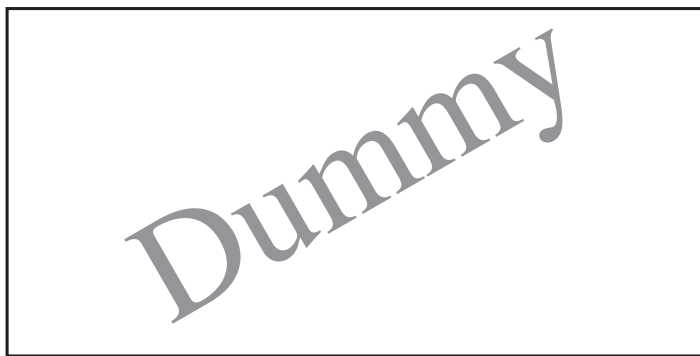1. kernel.org   2014

Fig. 11. Utilization of four tasks on the Gdev resource resevation. All tasks are fairy given resources which is 25%

GPU

RGEM   GPGPU   GPU

Gdev   RGEM   GPU
Resource Reservation   QoS   OS
CUDA

PTask[36] is an OS abstraction for GPU applications that optimizes data transfers and GPU scheduler.

Elliott et al. present GPUSync[37], [14]. GPUSync   GPU
GPU   GPU

runtime

GPU
GPUSync
k-exclusion lock
GPU   Resource Reservation
QoS   GPU   P2P
migration   MultipleGPU

Han et al. show GPU-SPARC[28]. GPU-Sparc support to automatically split and run the GPU kernel concurrently over multi-GPU, and then they supported priority queue based scheduling.

We show the table2 that is result of comparing the Linux-RTXg and prior work.

6 CONCLUSION

In this paper, we present linux real-time extension for cpu-gpu resouce coordination called linux-rtxg for cpu and gpu coordinated resource management. We focus on that are specifically not modify the kernel, worked for GPU resource management. GPU

GPU
running on the CPU
GPU
GPU
top-half

x%(sleep   )
task   x%

Gdev
QoS Management
x%

GPUSparc

GPU

GPU

REFERENCES

[1] M. McNaughton, C. Urmson, J. Dolan, and J.-W. Lee, "Motion planning for autonomous driving with a conformal spatiotemporal lattice," in *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, May 2011, pp. 4889–4895.

TABLE 2
Linux-RTXG vs prior work

	CPU		GPU		Budget Enforcement	Data/Comp. Ovlp.	Auto GPU Allocate	Closed Src. Compatible	Kernel Free	Configurable
	FP	EDF	FP	EDF						
RESCH	x	x							x	x
RGEM			x						x	
Gdev			x		x	x				
PTask					x	x	x	x		
GPUSync	x	x	x		x	x	x	x	x	
GPUSparc			x			x				
Linux-RTXG	x	x	x		x	x		x	x	x

[2] M. Hirabayashi, S. Kato, M. Edahiro, K. Takeda, T. Kawano, and S. Mita, "Gpu implementations of object detection using hog features and deformable models," in *Cyber-Physical Systems, Networks, and Applications (CP-SNA), 2013 IEEE 1st International Conference on*. IEEE, 2013, pp. 106–111.

[3] N. Rath, S. Kato, J. Levesque, M. Mauel, G. Navratil, and Q. Peng, "Fast, multi-channel real-time processing of signals with microsecond latency using graphics processing units," *Review of Scientific Instruments*, vol. 85, no. 4, p. 045114, 2014.

[4] S. Kato, K. Lakshmanan, Y. Ishikawa, and R. Rajkumar, "Resource sharing in gpu-accelerated windowing systems," in *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2011 17th IEEE*. IEEE, 2011, pp. 191–200.

[5] P. Bakkum and K. Skadron, "Accelerating sql database operations on a gpu with cuda," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*. ACM, 2010, pp. 94–103.

[6] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*. IEEE, 2009, pp. 44–54.

[7] "Cuda zone," https://developer.nvidia.com/category/zone/cuda-zone, accessed January 12, 2015.

[8] J. E. Stone, D. Gohara, and G. Shi, "Opencl: A parallel programming standard for heterogeneous computing systems," *IEEE Des. Test*, vol. 12, no. 3, pp. 66–73, May 2010.

[9] S. Kato, K. Lakshmanan, R. R. Rajkumar, and Y. Ishikawa, "Timegraph: Gpu scheduling for real-time multi-tasking environments," in *2011 USENIX Annual Technical Conference (USENIX ATC ' 11)*, 2011, p. 17.

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

[11] S. Kato, M. McThrow, C. Maltzahn, and S. A. Brandt, "Gdev: First-class gpu resource management in the operating system." in *USENIX Annual Technical Conference*, 2012, pp. 401–412.

[12] Y. Fujii, T. Azumi, N. Nishio, S. Kato, and M. Edahiro, "Data transfer matters for gpu computing," in *Parallel and Distributed Systems (ICPADS), 2013 International Conference on*. IEEE, 2013, pp. 275–282.

[13] G. A. Elliott, B. C. Ward, and J. H. Anderson, "Gpusync: A framework for real-time gpu management," in *Real-Time Systems Symposium (RTSS), 2013 IEEE 34th*. IEEE, 2013, pp. 33–44.

[14] G. A. Elliott and J. H. Anderson, "Exploring the multitude of real-time multi-gpu configurations," 2014.

[15] J. Calandrino, H. Leontyev, A. Block, U. Devi, and J. Anderson, "Litmus rt : A testbed for empirically comparing real-time multiprocessor schedulers," in *Real-Time Systems Symposium, 2006. RTSS '06. 27th IEEE International*, Dec 2006, pp. 111–126.

[16] J. A. Stankovic and K. Ramamritham, "The spring kernel: A new paradigm for real-time systems," *Software, IEEE*, vol. 8, no. 3, pp. 62–72, 1991.

[17] T. Yang, T. Liu, E. D. Berger, S. F. Kaplan, and J. E. B. Moss, "Redline: First class support for interactivity in commodity operating systems." in *OSDI*, vol. 8, 2008, pp. 73–86.

[18] H. Monden, "Introduction to itron the industry-oriented operating system," *Micro, IEEE*, vol. 7, no. 2, pp. 45–52, 1987.

[19] R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa, "Resource kernels: A resource-centric approach to real-time and multimedia systems," in *Photonics West'98 Electronic Imaging*. International Society for Optics and Photonics, 1997, pp. 150–164.

[20] S. Oikawa and R. Rajkumar, "Portable rk: A portable resource kernel for guaranteed and enforced timing behavior," in *Real-Time Technology and Applications Symposium, 1999. Proceedings of the Fifth IEEE*. IEEE, 1999, pp. 111–120.

[21] P. Mantegazza, E. Dozio, and S. Papacharalambous, "Rtai: Real time application interface," *Linux Journal*, vol. 2000, no. 72es, p. 10, 2000.

[22] V. Yodaiken *et al.*, "The rtlinux manifesto," in *Proc. of the 5th Linux Expo*, 1999.

[23] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *J. ACM*, vol. 20, no. 1, pp. 46–61, Jan. 1973. [Online]. Available: http://doi.acm.org/10.1145/321738.321743

[24] ——, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM (JACM)*, vol. 20, no. 1, pp. 46–61, 1973.

[25] L. Abeni and G. Buttazzo, "Integrating multimedia applications in hard real-time systems," in *Real-Time Systems Symposium, 1998. Proceedings., The 19th IEEE*. IEEE, 1998, pp. 4–13.

[26] M. Spuri and G. Buttazzo, "Scheduling aperiodic tasks

in dynamic priority systems," *Real-Time Systems*, vol. 10, no. 2, pp. 179–210, 1996.

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

[28] W. Han, H. Bae, H. Kim, J. Lee, and I. Shin, "Gpu-sparc: Accelerating parallelism in multi-gpu real-time systems," 2014.

[29] F. Ridouard, P. Richard, and F. Cottet, "Negative results for scheduling independent hard real-time tasks with self-suspensions," in *Real-Time Systems Symposium, 2004. Proceedings. 25th IEEE International*. IEEE, 2004, pp. 47–56.

[30] K. Lakshmanan, S. Kato, and R. R. Rajkumar, "Open problems in scheduling self-suspending tasks," in *Proc. Real-Time Scheduling Open Problems Seminar*, 2010.

[31] B. Chattopadhyay and S. Baruah, "Limited-preemption scheduling on multiprocessors," in *Proceedings of the 22nd International Conference on Real-Time Networks and Systems*. ACM, 2014, p. 225.

[32] J. Kim, B. Andersson, D. d. Niz, and R. R. Rajkumar, "Segment-fixed priority scheduling for self-suspending real-time tasks," in *Real-Time Systems Symposium (RTSS), 2013 IEEE 34th*. IEEE, 2013, pp. 246–257.

[33] Y. Fujii, T. Azumi, N. Nishio, and S. Kato, "Exploring microcontrollers in gpus," in *Proceedings of the 4th Asia-Pacific Workshop on Systems*. ACM, 2013, p. 2.

[34] G. A. Elliott and J. H. Anderson, "Robust real-time multiprocessor interrupt handling motivated by gpus," in *Real-Time Systems (ECRTS), 2012 24th Euromicro Conference on*. IEEE, 2012, pp. 267–276.

[35] S. Kato, J. Aumiller, and S. Brandt, "Zero-copy i/o processing for low-latency gpu computing," in *Proceedings of the ACM/IEEE 4th International Conference on Cyber-Physical Systems*. ACM, 2013, pp. 170–178.

[36] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel, "Ptask: Operating system abstractions to manage gpus as compute devices," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM, 2011, pp. 233–248.

[37] G. A. Elliott, B. C. Ward, and J. H. Anderson, "Gpusync: A framework for real-time gpu management," in *Real-Time Systems Symposium (RTSS), 2013 IEEE 34th*. IEEE, 2013, pp. 33–44.

Takuya Azumi Biography text here.

Nobuhiko Nishio Biography text here.

Tuyoshi Hamada Biography text here.

Shinpei Kato Biography text here.

Yusuke Fujii Biography text here.