Linux-RTXG: Coordinating CPU and GPU Resources with Loadable Real-Time Schedulers

Yusuke Fujii, *Student Member, IEEE,* Takuya Azumi, Nobuhiko Nishio, Tsuyoshi Hamada, 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, and do not consider real-time. Previous work have contributed for real-time GPU, and depends kernel. Dependence on the kernel and the device driver gives the burden developers and users. We present Linux-RTXG which is cordinating CPU and GPU resources for real-time extending framework without kernel modification by loadable kernel modules approach. Linux-RTXG has a CPU real-time scheduler, a GPU real-time scheduler, and a GPU reservation mechanism. To achieve framework without kernel modification, we presents interrupt intercept mechanisms and independent interrupt mechanisms and to solved currently Linux kernel real-time scheduling problem. Experimental results indicate that overhead of GPU scheduling using Linux-RTXG framework, and kernel free's approach performances of QoS management had kept equivalent performance as compared with the existing kernel-dependent approach.

Keywords—GPU, resource managmenet, scheduling, real-time system, operating system

1 Introduction

Graphic Processing Units (GPUs) are 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 an fusion reactor [3], user-interactive application [4], databases [5], and benchmarks [6] that contains a lot of applications. GPUs performance has been demonstrated by these research.

The past GPU applications should have been only "real-fast" since they 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] mainly target a best-effort applications, they do not support real-time requirements. Therefore, GPU runtime environments are required to support real-time scheduling.

In our previous work, we showed how to support GPU resource management. TimeGraph [9] provides GPU scheduling and reservation mechanisms at the device driver level to queue and dispatch GPU commands based on task priorities. Gdev [10] is applied resource reservation. These work have weak point that can not to provide fast supporting update architecture and full-funciton because these work is based on reverse engineering.

In the GPU environments, GPU functions are provided by the API of library and the Application binary interface (ABI) of device drivers. GPU functions consume CPU time to issue the GPU kernel by the API and the ABI. 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 resources in the host side at studies [11] is evaluating the data transfer time between the host and the device.

GPUSync [12], [13] 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 policies of tasks allocation to multi-core and multi-processor CPUs and policies of GPU kernel allocation to multiple GPUs.

However, GPUSync is implemented on the $LITMUS^{RT}$ [14], 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 users. A big burden is given patch to both developers and users. Specifically, developers obligation that is maintenance of patch in order to catch up to the latest kernel release. However, if software 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 supports a loadable kernel module (LKM) which 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 does not support GPU resource.

Contribution: In this paper, we present linux loadable real-time extension for CPU and GPU resource coordination called Linux-RTXG. 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 without kernel modification. To achieve real-time task scheduling, Linux-RTXG provide a CPU task scheduler, a GPU kernel scheduler, and a GPU kernel reservation mechanisms. The CPU task scheduler is based on the RESCH. The GPU kernel scheduler and the GPU kernel reservation mechanisms are based on the Gdev. They are integrated to Linux-RTXG by kernel free's approach. We define the kernel free that is do not modify kernel source codes and device drivers source codes.

Organization This rest of the paper is organized as follows. Section 2 discusses the GPU real-time scheduling by kernel free approach. Section 3 shows Linux-RTXG design and implementations, especially focus on GPU scheduling. Section 4 indicates Linux-RTXG's advantages and dis-advantages, furthermore, demonstrates 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 in this paper, and also we discuss the GPU scheduling and prior works. Next, we make available limitation for clearing the implementation of GPU scheduling without kernel modification. This paper focuses on a system composed of multiple GPUs and multi-core CPUs.

2.1 GPU Task Model

In case of General Purpose GPU computing (GPGPU), CUDA and OpenCL are used to implement GPU applications. This paper focuses on CUDA, but this paper approach is possible to adapt the same approach to OpenCL.

We define a GPU task as a process wich executes GPU, which cyclic executes unit is called GPU job, and, a GPU kernel is a processing to be executed on the GPU side. We support that a GPU application has multiple tasks.

GPUs use a set of the API supported by the runtime environment such as CUDA, typically GPU application takes the following steps: (i) *cuCtxCreate* creates a GPU context, (ii) *cuMemAlloc* allocates memory spaces to device memory, (iii) *cuModuleLoad* and *cuMemcpyHtoD* copy the data and the GPU kernel to the allocated device memory spaces from host memory spaces, (iv) *cuLaunchGrid* invokes the GPU kernel, (v) *cuCtxSynchronize* synchronizes waiting GPU task to that is completion of kernel, (vi) *cuMemcpyDtoH* transfers resultant data to host memory from device memory, and (vii) *cuMemFree*, *cuModuleUnload*, and *cuCtxDestroy* release allocated memory spaces and the GPU context.

2.2 GPU Scheduling

Real-time OS (RTOS) researches [15], [16], [17], [18] have been conducted for a long time. In among them, there are many existing studies concerning RTOS based on Linux [14], [19], [20], [21], [22]. The available OS for GPUs are limited to Windows, Mac OS and Linux. We selected Linux in order to achieve real-time processing on GPU environments.

In order to meet real-time constraints on shared resource environment such as a multi-core environment, there are two requirements for the scheduler as follows:

- To use resources according to a specified order
- To limit the use of the shared resources

A basic approach of the previous works to satisfy the first requirement uses priority-based scheduling (e.g. Rate-Monotonic [23] and Earliest Deadline First [24]) with technique to prevent priority inversion, and the second one is resource reservation-based scheduling (e.g. Constant Bandwidth Server [25], Total Bandwidth Server [26]). GPUs need to handle a data transfer bandwidth and a processing core as a shared resource; thus, we satisfy the two requirements similar to above multicore environment. Our previous works schedule only GPU accesses. However, GPU kernel consumes CPU time because GPU task driven by API. In order to truly support the real-time scheduling, scheduler is need to schedule GPU tasks of CPU cide. Therefore, the framework needs to have CPU's priority-based scheduler, GPU's priority-based scheduler, and GPU's resource reserbation mechanism to realize real-time GPU Recently, GPUSync target interdependence of CPU scheduler and GPU scheduler.

GPUs have some problems on real-time environment, except scheduling mechanisms. GPUs runtime environments are black-box mechanisms results from GPU environments are provided only GPU vendors, and these environments are closed-source. TimeGraph and Gdev address this problem by ensure transparency using reverse engineering and an open-source driver. GPUSync achieve closed-source compatible by GPU resource management that approaches are the interrupt handling and the arbitrate runtime access.

The other problem occurs by non-preemptive GPU executions and non-preemptive data transfers. Several researches [27], [28] have improvemed the response time by preventing overrun which occurs while dividing the kernel. However, these existing methods concerning real-time GPU are typically experimental, not practical enough. The most difficult problem is self-suspending because GPU is treated as an I/O device. GPU tasks suspend until it receives the results from invoking 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 researches [31], [32] are working on the scheduling analysis for self-suspension task, they have not been solved yet completely.

Hence, the proposed scheduling framework aims at easier expansion and installation.

GPU Synchronization: The synchronization must be considered for GPU system such as heterogeneous platform. GPU have two different synchronization technique. The first techniques is memory map based synchronization which is called FENCE, FENCE sends GPU commands after the command to take action, then GPU microcontrollers will write the any value to a memory-mapped space after action is completed. A GPU task monitors the mapped space value using such as polling. Therefore task has an exclusive CPU resource, but response time will be the fastest. The other techniques is interruption based synchronization which is called NOTIFY. NOTIFY sends GPU commands similar to FENCE, then GPU microcontrollers will rise the interrupt and write any value to GPU I/O registers. A GPU task is suspending until interrupt. Therefore a task is able to share the CPU resources with other tasks, but a response time will be the slow. A Detailed architecture is omitted in this paper, and the detailed architecture has been described in the previous documents [9], [10], [33].

Gdev uses both techniques, NOTIFY and FENCE for wakeup the waiting task, and NOTIFY is used by scheduler, FENCE is used by kernel synchronization. In Gdev, synchronization implementation is the additional commands sends to GPU and the modification of device driver's interrupt handler. GPUSync uses NOTIFY technique by using tasklet intercept [34] on the proprietary software. Tasklet is Linux's soft-irq implementation. GPUSync identifies the interrupt that is invoked kernel by callback pointer with a tasklet.

kernel free scheduler: We must not modify the kernel code and the device driver code to achieve "kernel free". GPU scheduler is required to receive a notice of GPU kernel completion, for selection 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 $cuCtxSynchronize()$ provided runtime in the RGEM. The Interrupt-driven method is woken up by a 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 task invokes 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, and its method is required to modify the kernel or the device driver's ISR. Gdev has been achieved independent synchronization mechanisms on the proprietary soft-ware, the independent synchronization mechanisms are needed to modify the kernel modification. The challenge is realizing independent synchronization mechanisms without kernel modification

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 Linux real-time GPU scheduling framework without kernel modification.

We describe the main contribution of the GPU scheduler and the integration to CPU scheduler in this paper. CPU scheduling description is minimized because Linux-RTXG is based on RESCH.

3.1 Linux-RTXG

Figure 1 shows an overview of Linux-RTXG. Linux-RTXG is divided into two parts: a Linux-RTXG core and a Linux-RTXG library. The Linux-RTXG core has a CPU task scheduler, a GPU task scheduler, and a GPU resource reservation mechanism. Because the Linux-RTXG core is loaded into the kernel-space, it can use the kernel exported functions such as $schedule()$, $mod\_timer()$, $wake\_up\_process()$ and $set\_cpus\_allowed\_ptr()$.

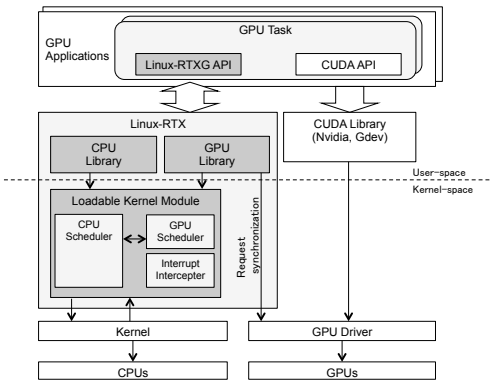


Fig. 1. Overview of the Linux-RTXG

The Linux-RTXG library is interface to communicate between application and Linux-RTXG core. These interface are implemented using an ioctl system call, which is a standard way of communicating to a driver.

The part of the library included a method that is an independent synchronization method. The independent synchronization method is used only on the Nvidia driver. In case of using an open-source GPU driver such as Nouveau [35], GPU runtime must use part of Gdev. Because Gdev can manage arbitrary interrupt of the GPU kernel in the user-space mode, the GPU scheduler have not need the independent synchronization method because

3.2 GPU Scheduling

Linux-RTXG is API-driven scheduler 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 the others do not. Linux-RTXG does not modify the existing CUDA API to cope with proprietary software to be independent from the runtime. However, user has to add Linux-RTXG API to existing CUDA application for using the Linux-RTXG scheduler.

The sample code of using the Linux-RTXG scheduler is shown in Figure 2, and its code except GPU scheduling is omitted. GPU tasks may be provided with a function by calling Linux-RTXG's API at strategic points.

Figure 3 shows the control flow of the sample code. The configuration is the Kernel issue that

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_gpu_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 Linux-RTXG scheduler

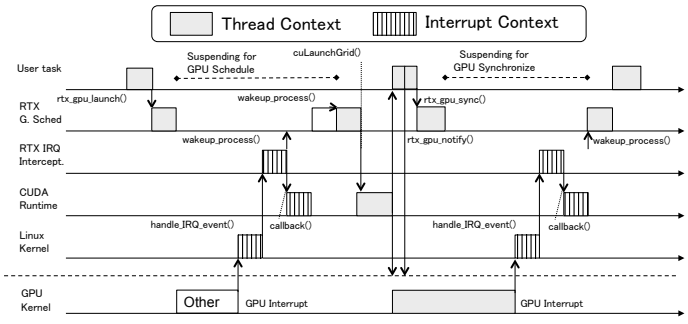


Fig. 3. GPU Scheduling control flow

is restricted to single kernel. A GPU Task can control the timing of GPU kernel execution by calling $rtx\_gpu\_launch()$. The task goes to sleep until the wakeup by interrupt why the task is not permitted issuance of GPU kernel due to already execute other task in the GPU.

Other task's GPU kernel is finishied, then, linux kernel handle the interrupt. The interrupt intercepter wakeup the GPU scheduler (RTX G. Sched. in Figure 3) by intercept interrupt. The GPU scheduler wakeup the sleeping task. The waken up the 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 included group (virtual GPU) scheduling and GPU kernel scheduling. The virtual GPU scheduling uses a resource reservation mechanism. The GPU kernel scheduling uses a priority scheduling. Specifically, GPU kernel execution is associated to each scheduling entity while Linux-RTXG grouped the scheduling entity to virtual GPUs. The virtual GPUs belong to any of physical GPUs. In Linux-RTXG, resources are distributed in its virtual GPUs.

Figure 4 shows pseudo-code of scheduling mechanism. $on\_arrival$ is called when a GPU task is requested GPU kernel launch issue. In $on\_arrival$, a GPU task to check whether the given execute permission to virtual GPU of task itself, and then, check the se permit. If a virtual GPU to which the GPU task belongs has not executing permission, the GPU task is enqueued to wait_queue and goes to sleep. On the other hand, if the virtual GPU has executing permission, the GPU task goes to launch issue.

$on\_completion$ is called by a scheduler thread, when the GPU kernel is completion. $on\_completion$ picks up the next virtual GPU and the next a GPU task. Next then,

```
 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

on_completion wakes up the pick uped the GPU task.

3.3 GPU synchronization

We describe the independent synchronization mechanism and the interrupt intercept. The independent synchronization mechanism is occuring NOTIFY and FENCE without using GPU runtime's API. The interrupt intercept is to realize interrupt-driven wakeup the scheduler without kernel modification. Linux-RTXG uses the independent synchronization mechanisms as much as possible, because we do not want using black-box resource management to realize truly real-time resource environments.

Independent synchronization mechanism from runtime We present an independent synchronization mechanism of the NOTIFY and the FENCE. The mechanism is to occur interrupt for NOTIFY, and to write the fence value by microcontrollers for FENCE. NVIDIA's proprietary software uses ioctl interface to communicate 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 method is divided into two parts: Initialize and Notify. Initialize processes for generating a GPU context dedicated this method. These processes are included a creating virtual address space, a allocating indirect buffer object for command sending, and a creating context object. The virtual address space is used for managing the GPU device memory space and kernel memory space (Host-side) such as indirect buffer. The indirect buffer is an area of storing GPU commands for sending GPU commands. The creating context object is preparation for using the FIFO engine, such as allocating kernel memory object and mapping FIFO engine register to host memory space by memory-mapped I/O. The FIFO engine is GPU microcontroller for receiving commands. The Notify processes send commands to the compute engine or the copy engine by iowrite commands to mapped FIFO engine's register. This independent generating synchronization sign for synchronization mechanisms is using reverse engineering. However, the method has limitation because the method depends on the proprietary 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.

The Linux kernel has structures 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 a callback pointer of ISR. We retain getting the callback pointer of a

GPU device driver's ISR, and then we register a interrupt interception ISR to kernel. So, we get the interrupt interception by the interrupt interception ISR and calling retaining callback pointer. In addition, I/O registers are mapped to kernel memory space by device driver from the PCIe base address registers (BARs) [11], [36]. Therefore, Linux-RTXG remaps the BAR0 to our allocated space by using $ioremap()$ when the ISR is initializes. The interrupt interception identifies interrupt by reading the mapped-space.

3.4 Scheduler Integration

Linux scheduler has various real-time scheduling policies that were $SCHED\_DEADLINE$, $SCHED\_FIFO$ and $SCHED\_RR$. $SCHED\_DEADLINE$ is implementation of Constant Bandwidth Server and Global Earliest Deadline First, while $SCHED\_DEADLINE$ is included in 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 a 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, the task can not execute until runtime is replenished in the next period. Therefore, the task is unable to call $sched\_yield()$ between polling. The $sched\_yield()$ is used much by device drivers and library as well as GPU runtime. These software is affected by this problems. 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 support 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 scheduling performance. Scheduling experiments are limited on the GPU scheduling since CPU scheduling performance is already experiments [22]. In this evaluation, we focus two point. The one is to indicate the Linux-RTXG disadvantages. The other is demonstrating QoS performance.

Real-world oriented applications using GPU [2], [3] are executed periodically. These evaluation applications are equivalent to the real-world oriented applications because GPU applications characteristics are these application on the single GPU kernel. We will discuss comparing to hold the functions and features between the this work and previous works with a qualitative evaluation in the next chapter.

4.1 Experimental Environment

Our experiments are conducted with the Linux kernel 3.16.0 on NVIDIA Geforce GTX680 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.

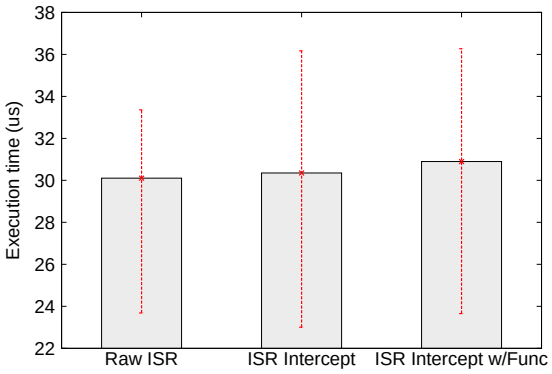


Fig. 5. Interrupt intercept overhead

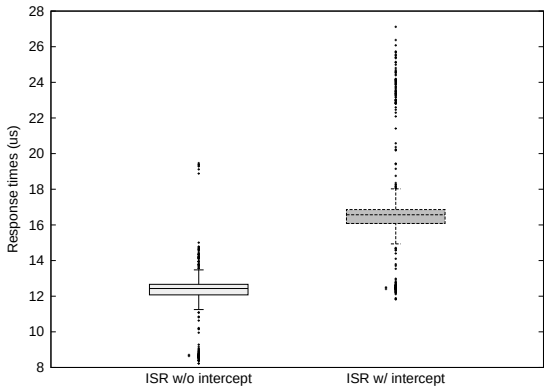


Fig. 6. Comparison of the response time of the interrupt processing with intercept and the interrupt processing without intercept

4.2 Interrupt intercept overhead

We measured overhead of interrupt interception. We use GPU driver which is used the Nouveau in order to compare and identify the type of interrupt. We compared consumption time from begin of ISR to the end of ISR.

Figure 5 shows results of measurements in the above setting. Raw ISR is execute ISR in the original routine. ISR Intercept is only intercept our approach. ISR Intecept w/Func is interception with processing functions which are identify the ISR and wake-up the scheduler thread. They are showed the average times of 1000 times with error bar is indicate minimum and maximum.

As a result, the overhead exist certainly. ISR Intercept has overhead which is 247 nano seconds, and its overhead is 0.8% of the entire Raw ISR process. ISR Intercept w/Func also has overhead that is 790 nano seconds, and its overhead is 2.6% of the entire Raw ISR process. Intuitively, these overhead does not affect system since very small values, however, interrupt is occur frequently including such as timer interrupt, should be aware as disadvantages.

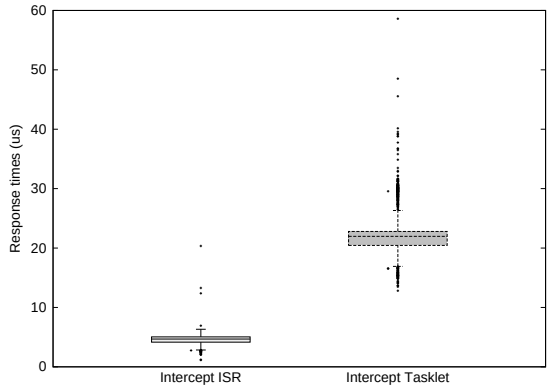
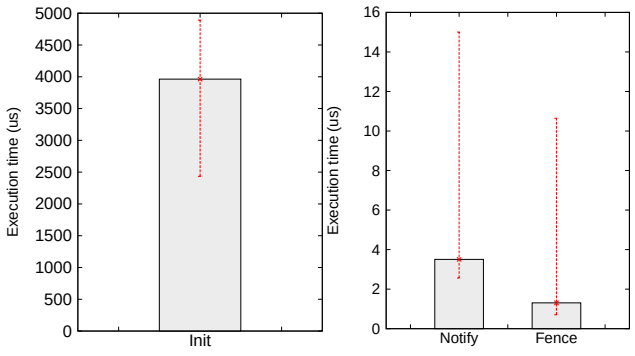


Fig. 7. Comparison of the response time of the interrupt top-half and bottom-half

Next, we compare response times as the interrupt impact of the interrupt itnercept. The response times are consumption times from start interrupt processing to processing for the type. We show the response times that are a ISR with intercept and a ISR without intercept, as a result show in Figure 6. If we use the interrupt interception, response time become about 1.4 times as compared to without intterupt interception. However, we targetting systems do not use the GPU runtime's resource management features. Therefore, We should aim to fast response time of intercepted ISR, and original ISR response time is slow there is not affected much.

In addition, we evaluate comparing response time of the ISR (top-half) and the tasklet (bottom-half) in an environment with non CPU load for comparing prior works approach. GPUSync intercept the $tasklet\_schedule()$ if $tasklet\_schedule()$ is called from the Nvidia driver. We confirm whether the fast response time of a tasklet intercept using GPUSync and the ISR intercept. This evaluation does measurement time until the responsible timing from the start of interrupt process which is do_IRQ function is called). Figure 7 shows the result of this evaluation. Tasklet interception



(a) Part of Initialize (b) Part of notify

Fig. 8. Interrupt raised method overhead

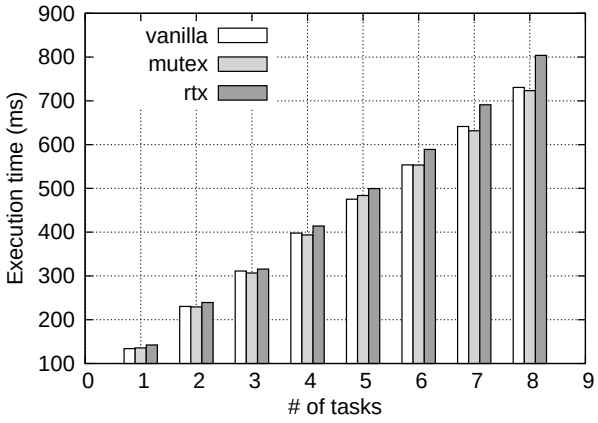


Fig. 9. Scheduling overhead (Time of entire task)

response time is worse than ISR, as shown in Figure 7. This result because the tasklet is generally called after the significant procesing in ISR.

4.3 Independent Synchronization mechanism overhead

We evaluated the overhead according to using the independent synchronization mechanism. The mechanism is needed to call the $rtx\_nvrm\_notify()$ at the timing of requested synchronization (e.g. after the kernel launch issue) and $rtx\_nvrm\_init()$. In vanilla environment, the APIs are not necessary. Therefore, the overhead includes the API execution time. We measured the overhead by measuring API execution time between call and return of APIs.

Figure 8 shows measurement result. Initialize is need to called the at awaken a Linux process for allocating an indirect buffer and registers several engines such as compute and copy to the device driver. Notify and Fence are GPU commands sending to GPU devices which called at timing of the need to synchronization such as after the kernel launch issues. Execution time of the methods have scatter that is affected by ioctl system call.

Initialize average time is about 4msec. However, application is not affected too much because above characteristics is only called once. Notify is not taking much time that is about 3.5sec. Fence likewise is not taking much time that is about 2sec. Although may be not have to worry about for most applications. However, overhead is necessary to considered in a cycle of short application.

4.4 Scheduling Overhead

We evaluated scheduling overhead by using the Linux-RTXG scheduler. We prepared three applications that are "vanilla", "mutex", and "rtx" for measurement overheads. These applications are based common application based on Gdev's microbenchmark which has GPU looping function. Changes are arranged to generate multiple GPU tasks by the fork. Each tasks have 10 jobs, and a job is included GPU data transferring and a GPU kernel execution.

The "rtx" application is scheduled by rtx. The "mutex" application is limited to single kernel issue similar to scheduling environment. The "vanilla" application is not to change the base application.

CPU scheduling policy is using the simple fixed-priority scheduling by Linux-RTXG similar to Linux's $SCHED\_FIFO$ that difference is the presence or absence of the job management. GPU scheduling policy is fixed-priority scheduling with resrouce reservation which is BAND scheduling policy. The synchronization is using NOTIFY of the independent synchronization mechanism.

We measured the average time in 100 times GPU task execution (1000 jobs). The result show in Figure 9. The "launch_advice" is time of until GPU kernel launch request is accept on the Linux-RTXG. The "mutex" is time to get

the mutex lock. The “sync” is time of untile synchronizationed from issued the GPU kernel launch issue.

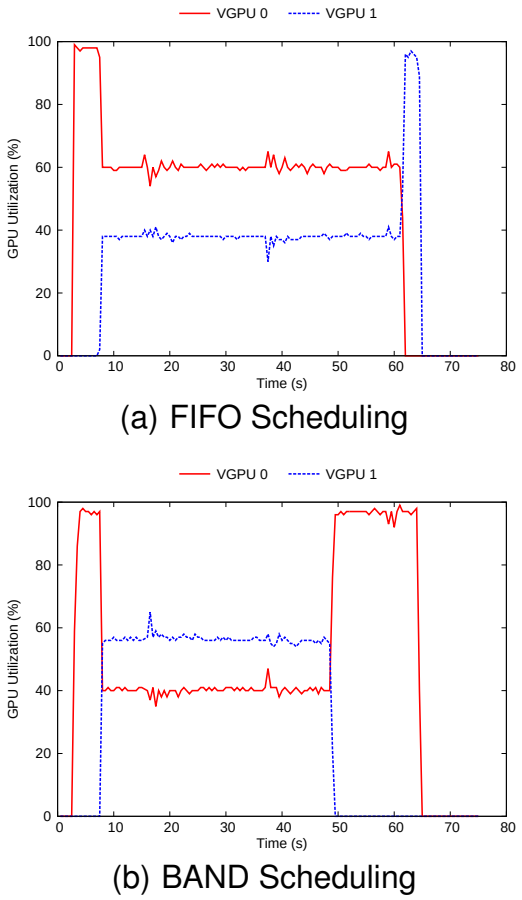
Overhead increases in proportion as the number of tasks increases because task scheduling processing times is increased. Max overhead is 10% at the eight tasks based on ”vanilla” time. The most factors contained in the overhead is the API’s overheads and scheduling algorithms (e.g. Band scheduling include yield times). If application periods is short, users are conscious of the overheads.

4.5 Performance of QoS management

Next experiments are evaluating QoS management performance. In this evaluate, we measure the utilization of each tasks on the several environments. Thereby, we indicates performance that is not falling by not modificate kernel.

We experiment task isolation performances on the Linux-RTXG’s GPU scheduler with GPU driver which is used the Nvidia GPU driver and the Nouveau GPU driver. The first, we measure the utilization of the running two GPU tasks. The each GPU tasks have a different workload and a different resource. A GPU task is allocated VGPU0, and given the 40% resource, and has about 1.2 times the workload of other tasks. The other GPU task is allocated VGPU1, and given the 60% resource. VGPU1’s task is to start after about 5 seconds of VGPU0’s task.

A Nvidia GPU driver’s result is showed in Figures 10. The FIFO scheduling policies shown in Figure 10(a), and the BAND scheduling policies [10] shown in Figure 10(b). A Nouveau GPU driver’s result is showed in Figures 11. The FIFO scheduling policies shown in Figure 11(a), and the BAND scheduling policies shown in Figure 11(b). Figure 10(a), 11(a) indicate that the GPU tasks performed in accordance with the workload by fair scheduling. On the other hand, Figure 10(b), 11(b) indicate that the GPU tasks performed in accordance with the utilization by resource reservation mechanisms. A VGPU1’s maximum error is about 3% in the starting BAND scheduling policies resource management using by Nvidia GPU



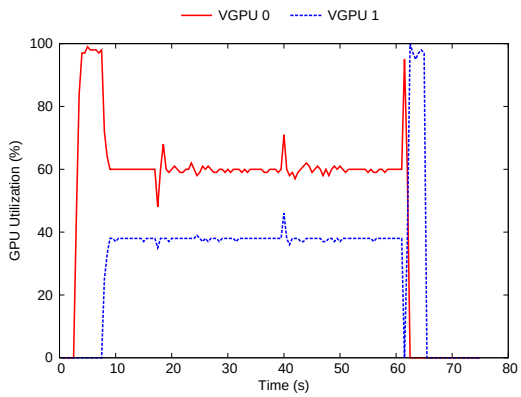
(a) FIFO Scheduling

(b) BAND Scheduling

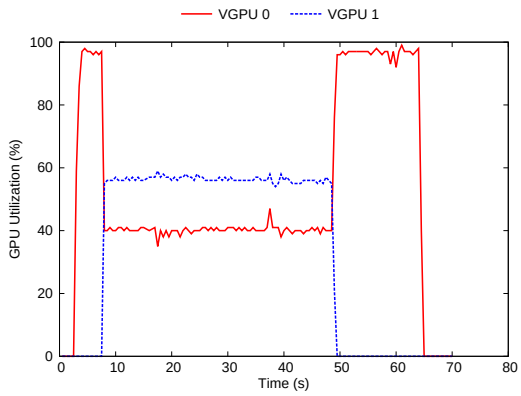
Fig. 10. Utilization of two tasks on the Linux RTXG scheudler and using Nvidia driver. Each tasks have different workloads and different resources (VGPU0 = 40%, VGPU1 = 60%).

driver, and a VGPU0’s maximum error is about 5%. A VGPU1’s maximum error is about 2% in the starting BAND scheduling policies resource management using by Nouveau GPU driver, and a VGPU0’s maximum error is about 2%. Large spikes are occured by GPU kernel’s overrun. If the GPU scheduler need to large spike is reduced, the GPU scheduler is needed to runtime approachs such as making preemptive GPU kernel.

The second, we measure the utilization of the running four GPU tasks. The each GPU tasks have a same workload and same resources, and each GPU tasks allocate to each VGPUs. The BAND scheduling policies result shown in Figure 12. A maximum error of these VGPUs is about 9%, it values are happend by a timing of budget replenish and a synchronization latency.



(a) FIFO Scheduling



(b) BAND Scheduling

Fig. 11. Utilization of two tasks on the Linux RTXG scheudler and using Nouveau driver. Each tasks have different workloads and different resources (VGPU0 = 40%, VGPU1 = 60%).

Therefore, the synchronization mechanism of Linux-RTXG excluding kernel modification is show that possible scheduling without sacrificing performance.

5 Related Work

RGEM and GPU-Sparc [28] have GPU resource management without modification of the kernel and device drivers. However, synchronization mechanisms of these work depends on proprietary-software. TimeGraph, Gdev, Ptask, and GPUSync are realized independent synchronization mechanisms for modifying the kernel and devie drivers. To our knowledge, Linux-RTXG is the only real-time GPU framework using a synchronization mechanisms that is independent of the runtime while it was not modificated the kernel and device drivers.

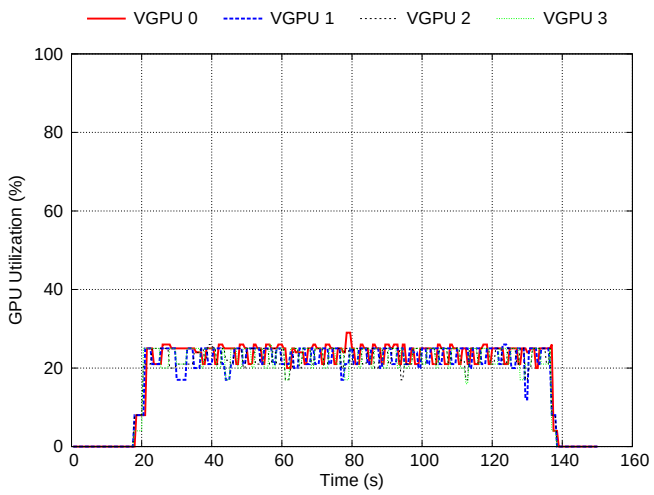


Fig. 12. Utilization of four tasks on the Linux-RTXG's BAND VGPU scheduling. Each tasks have a fair workload and a fair resources.

Table 2 shows that result of comparing the Linux-RTXG and prior work. GPUSync mentioned that GPU task dependent to CPUs scheduling, and supports CPU scheduling policies that are the Fixed-Priority scheduling and the EDF scheduling. GPUSparc have setting of scheduling priorities that is Linux scheduling policies of $SCHED\_FIFO$. Linux-RTXG is meet the all of things except the GPU. Specially, Linux-RTXG's contribution is meet to both GPU runtime independent and OS independent.

The more-depth resource management would require the detail executing mechanisms in the black-box GPU stack. Menychtas et al. present enabling GPU using OS research by inferring interaction in the black-box GPU stack [37]. We present information of GPU microcontrollers [33] and an open-source GPGPU runtime [10]. GPUSync presents details verification information on the proprietary runtime mechanisms. Nouveau project provides an open-source GPU driver [35]. These works are very important, and we will also views of the further development of resource management their researches.

6 Conclusion

This paper has presented Linux real-time extension for CPU-GPU resouce coordination is

TABLE 2
Linux-RTXG vs Prior Work

	CPU FP	CPU EDF	GPU Prio. Sched.	Budget Enforcement	Data/Comp. Ovlp.	Closed Src. Compatible	Kernel Free	OS independent	GPU Runtime independent
RGEM			x			x	x	x	
Gdev			x	x	x				
PTask			x	x	x	x			x
GPUSync	x	x	x	x	x	x			x
GPUSparc	▲		x		x	x		x	
Linux-RTXG	x	x	x	x	x	x	x	x	x

called Linux-RTXG for CPUs and GPUs coordinated resource management. We focus on that are specifically not modify the kernel, worked for GPU resource management. Linux-RTXG presented the CPU task scheduling, the GPU task sheduling and the GPU resource reservation mechanisms. The CPU task scheduling is based on RESCH. The GPU task scheduling provides prioritized scheduling by our synchronization mechanisms. Our synchronization mechanisms are not need to modify the kernel and device drivers, presented by intercept interrupt top-half ISRs.

We indicated a unit of job overhead met within about x%, and unit of task met overhead within x%. In addition, we the evaluations demonstrated that the QoS management performance whitout kernel modification by comparing Linux-RTXG and Gdev. The basic scheduling framework has already completed to realized real-time GPU computing. Future work will address GPU execution such as preemption, and p2p migration to deal with more complex real-time problems.

REFERENCES

[1] M. McNaughton, C. Urmson, J. Dolan, and J.-W. Lee, "Motion planning for autonomous driving with a conformal spatiotemporal lattice," in *Proc. ICRA*. IEEE, 2011, pp. 4889–4895.

[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 *Proc. Int. Conf. CPSNA*. 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 *Proc. RTAS*. IEEE, 2011, pp. 191–200.

[5] P. Bakkum and K. Skadron, "Accelerating SQL database operations on a GPU with CUDA," in *Proc. Workshop GPGPU*. 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 *Proc. IISWC*. 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. Comput.*, vol. 12, no. 3, pp. 66–73, 2010.

[9] S. Kato, K. Lakshmanan, R. R. Rajkumar, and Y. Ishikawa, "TimeGraph: GPU scheduling for real-time multi-tasking environments," in *Proc. USENIC ATC*, 2011, p. 17.

[10] S. Kato, M. McThrow, C. Maltzahn, and S. A. Brandt, "Gdev: First-Class GPU Resource Management in the Operating System." in *Proc. USENIX ATC*, 2012, pp. 401–412.

[11] Y. Fujii, T. Azumi, N. Nishio, S. Kato, and M. Edahiro, "Data transfer matters for GPU computing," in *Proc. ICPADS*. IEEE, 2013, pp. 275–282.

[12] G. A. Elliott, B. C. Ward, and J. H. Anderson, "GPUSync: A framework for real-time GPU management," in *Proc. RTSS*. IEEE, 2013, pp. 33–44.

[13] G. A. Elliott and J. H. Anderson, "Exploring the Multitude of Real-Time Multi-GPU Configurations," in *Proc. RTSS*. IEEE, 2014.

[14] J. Calandrino, H. Leontyev, A. Block, U. Devi, and J. Anderson, "$LITMUS^{RT}$: A Testbed for Empirically Comparing Real-Time Multiprocessor Schedulers," in *Proc. RTSS*, 2006, pp. 111–126.

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

[16] 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 *Proc. OSDI*, vol. 8, 2008, pp. 73–86.

[17] H. Takada and K. Sakamura, "μITRON for small-scale embedded systems," *IEEE Micro*, vol. 15, no. 6, pp. 46–54, 1995.

[18] 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.

[19] S. Oikawa and R. Rajkumar, "Portable RK: A portable resource kernel for guaranteed and enforced timing behavior," in *Proc. RTAS*. IEEE, 1999, pp. 111–120.

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

[21] V. Yodaiken, "The RTLinux Manifesto," in *Proc. Fifth Linux Expo*, 1999.

[22] S. Kato, R. Rajkumar, and Y. Ishikawa, "A loadable real-time scheduler suite for multicore platforms," Tech. Rep. CMU-ECE-TR09-12, Tech. Rep., 2009.

[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, 1973.

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

[25] L. Abeni and G. Buttazzo, "Integrating multimedia applications in hard real-time systems," in *Proc. RTSS*. 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 *Proc. ECRTS*, 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," Tech. Rep., 2014.

[29] F. Ridouard, P. Richard, and F. Cottet, "Negative results for scheduling independent hard real-time tasks with self-suspensions," in *Proc. RTSS*. 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 *Proc. Int. Conf. RTNS*. 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 *Proc. RTSS*. IEEE, 2013, pp. 246–257.

[33] Y. Fujii, T. Azumi, N. Nishio, and S. Kato, "Exploring microcontrollers in GPUs," in *Proc. APSys*. ACM, 2013, p. 2.

[34] G. A. Elliott and J. H. Anderson, "Robust real-time multiprocessor interrupt handling motivated by GPUs," in *Proc. ECRTS*, 2012, pp. 267–276.

[35] "Nouveau," http://nouveau.freedesktop.org/wiki/, accessed January 12, 2015.

[36] S. Kato, J. Aumiller, and S. Brandt, "Zero-copy I/O processing for low-latency GPU computing," in *Proc. Int. Conf. ICCPS*. ACM, 2013, pp. 170–178.

[37] K. Menychtas, K. Shen, and M. L. Scott, "Enabling OS Research by Inferring Interactions in the Black-Box GPU Stack." in *Proc. USENIX ATC*, 2013, pp. 291–296.

PLACE PHOTO HERE

Yusuke Fujii Biography text here.

PLACE PHOTO HERE

Takuya Azumi Biography text here.

PLACE PHOTO HERE

Nobuhiko Nishio Biography text here.

PLACE PHOTO HERE

Tuyoshi Hamada Biography text here.

PLACE PHOTO HERE

Shinpei Kato Biography text here.