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—
We present non pached cpu/gpu scheduler is called Linux-RTXG.

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[?] for autonomous drive, object detection[?], tokamak control for fusion reactor[?], user-interactive application[?] and databases[?]. And benchmarks[?] 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[?] and OpenCL[?] 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[?] provides GPU scheeduling and reservation machanisms at the device driver level to queue and dispatch GPU commands based on task priorities. RGEM[?] provides GPU kernel scheduling and data transfering scheduling, Gdev[?] 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[?] is evaluating the data transfer time between the host and the device.

GPUSync[?], [?] 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 $LITMUS^{RT}$[?], 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

- 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.

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, demonstrate experimental results that are quantitative overheads and reservation performances. Section **??** discusses related work. We provide our concluding remarks in Section **??**.

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 (*cuCtxCreate*), (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[**?**], [**?**], [**?**], [**?**] has been conducted for a long time. In among them have also been many studies[**?**], [**?**], [**?**], [**?**], [**?**] 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[**?**] and Earliest Deadline First[**?**]) with technique to prevent priority inversion, and the second is resource reservation based scheduling (e.g. Constant Bandwidth Server[**?**], Total Bandwidth Server[**?**]). 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 runtime 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[] realize implovement 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[**?**], [**?**], several researchs[**?**], [**?**] 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.

kernel free scheduler: To achieve "kernel free", we must not to modify the kernel code and the device driver code. RGEM does scheduling on the only user-space, this is realized priority-based sheduling by using is IPC (Inter Process Communication) privided by POSIX.

GPU scheduling does synchronization based scheduling, however, RGEM is to depends on the runtime software synchronization.

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.

GPU Synchronization: The synchronization matters for GPU system such as heterogeneous platoform. Furthermore, as mentioned we need to achieve the independs synchronization mechanisms by the kernel free aproach.

GPU have two different synchronization techniques. The first techniques memory map based synchronization 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 is interrupts based synchronization 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[**?**], [**?**], [**?**].

NOTIFY による同期を実現するためには，割込みを取得しそれがどのコンテキストから発行されたものかを識別しなければならない．FENCE による同期を実現するためには，コマンドを送信できるようにしなければならない．

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. tasklet is linux's soft-irq implementation. GPUSync identify the interrupt that is issued

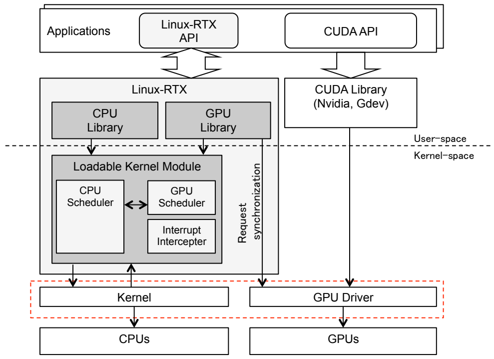


Fig. 1. Over view of the Linux-RTXG

which kernel by callback pointer with a tasklet.

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. 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.

```
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

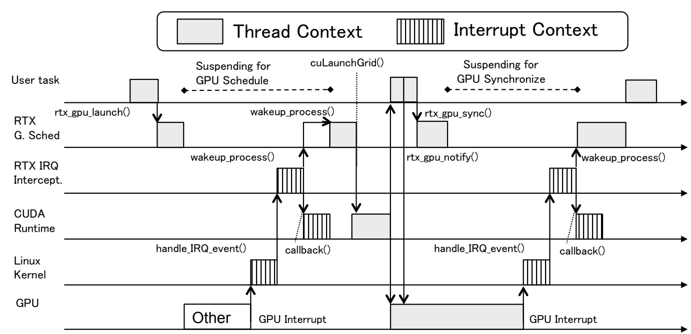


Fig. 3. GPU Scheduling control flow

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 **??**. Some APIs have arguments and others do not. Linux-RTXG APIs are not modificated 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 $gpu_t ask()$ is

Figure 3.2 shows control flow of run the Fig-

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.

ure2'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 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.

In Linux-RTXG, GPU scheduling purpose is the QoS management by task isolation based on Gdev's reservation mechanisms. GPU kernel execution is associated to each scheduling entity. Linux-RTXG grouped the scheduling entity to VGPUs, these VGPUs belong to any of physical GPUs.

GPU task to check whether the given execute permission to group of task itself. We present hierarchal scheduling which are group scheduling, context scheduling. group scheduling is resource reservation, context scheduling is priority scheduling.

3.3 GPU synchronization

Linux-RTXG is synchronization based scheduler that need to know the timing of GPU kernel launch request and the timing of it kernel finish. Linux-RTXG is given requesting of GPU kernel launch by rtx_gpu_launch().

```
void on_arrival(vgpu, ctx) {
 check_permit_vgpu(vgpu)
 while(!check_permit_ctx(ctx))
   sleep_task();
 return;
 }
```

Fig. 4. High Level Pseudo-code of rtxg sample code.

In order to notify the completion of the current GPU kernel execution, finish by using NOTIFY or FENCE. NOTIFY and FENCE are able to generated by

NVIDIA proprietary software can awaken NOTIFY か FENCE は NVIDIA のプロプライエタリ・ソフトウェアではコンテキスト生成時にフラグをセットすることで発生させることが可能であるが、どのカーネルが終了したかの識別と、

rtx_gpu_notify()、もしくはランタイムによって発生される割込みを獲得するか、cuCtxSynchronize 後に rtx_gpu_sync() に専用フラグをセットすることで呼び出す

Interrupt interception: Interrupts are handled by the ISR (Interrupt Service Routine) that is registered kernel by the device driver.

Linux-RTXG is hold the kernel thread for to select task. It kernel thread is select to next task when the kernel execution is completed, and then thread is going to sleep state in order to yield the CPU resources for other tasks.

for sleeping thread.

need register

これらを適切に立ち上げるためには任意の割込みを獲得し、外部 ISR がその割込みがどのカーネルに関連しているかを識別できる仕組みが必要である。

加えて、割込みの識別は GPU のステータス・レジスタを読み込んで行う必要があり、GPU ドライ

バが割込みレジスタをリセットする前に、実行される必要がある。

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 intercept ISRs to kernel. Therefore, we get the interrupt intercept by intercept ISR and then call retaining callback pointer, そして傍受用 ISR で、事前に保持しておいた GPU ドライバの割込みハンドラの関数コールバック関数として呼び出すことで、通常の割込みハンドリングを実行する。加えて我々のこれまでの研究 [?], [?] で、GPU の io register は PCIe の BAR0 によって指定されたアドレスから存在しておりカーネル空間にデバイスドライバによってマッピングされていることがわかっている。そのため Linux-RTXG が傍受用 ISR の初期化の際に $ioremap()$ によって BAR 0 空間をマッピングしておき、傍受用 ISR が呼び出された際にマッピングされたレジスタを読み込むことで、割込みの識別を行う。

Independent generate sign for Synchronization: We present independent generate sign for synchronization of NOTIFY and FENCE. It sign is 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. Gdev build infrastructure that is able to execute on the NVIDIA's driver using ioctl interface (Apprication Binary Interface). We utilized Gdev knowledge 本論文では、この基盤から割込みを発生させる部位のみ抽出し、スケジューリングに用いる。

This method 本メソッドは大きく 2 つに分かれ、それぞれ Initialize と Notify と呼ぶ。

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

Notify processes send commands to the compute engine or the copy engine that are

本アプローチに用いるインタフェースは公式にサポートされていないために、ベンダーによる急な仕様変更には対応できない。しかしながら、これ以外に割込みを発生させるアプローチがなく、クローズドソースを用いた場合の限界であるといえる。

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 problem is twofold. The first is implementation of sched_yield—in kernel space, yield()— The second is 本問題は 2 種類存在しており、sched_yield() による CPU 放棄の実装によるものと、suspending した後の復帰の実装によるものである。

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. NVIDIA の CUDA においても同期フラグの設定次第で本問題に影響を受ける。Linux-RTXG では SCHED_DEADLINE 時は NOTIFY を使うことを推奨し、sched_yield の利用を制限することで対応した。

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

我々の知る限り、カーネルやドライバを修正せずに GPU 資源管理に取り組んだ例は無い。

2014 年では旧バージョンのマイナーアップデートも含め 172 個 [1] のアップデートがリリースされている。汎用的な a

・優先度スケジューリングの評価はしなくてよいのか

・GPUSync と比較しなくて良いのか

・実アプリで動かすとどうなるのか

Linux-RTXG の優位性は

加えて、カーネルやドライバを修正しないことによる優位性は既に先行研究 [?] によって示されているため、

本稿における評価では、本 Linux-RTXG を利用した際のオーバヘッドを定量的に計測し、利用に伴ってどれだけのデメリットを含んでいるかを明記する。定性的な評価としては関連する研究と保持する機能や特徴の差について次章で discuss する。

4.1 Experimental Environment

本論文では次のマシンを用いて評価する。

CPU は Intel Core i7 2600 3.40GHz、4GB*2 のメモリ、GPU は GeForce GTX680 を用いる。Kernel は Linux kernel 3.16.0 を用い、ディストリビューションは Ubuntu 14.04 である。CUDA コンパイラは NVCC v6.0.1、CUDA ランタイムは cuda-6.0 or Gdev、GPU ドライバは NVIDIA (331.62)、Nouveau (linux-3.16.0) を用いる。各ランタイム、ドライバは評価項目ごとに使い分ける。

1. kernel.org で提供されるカーネルのうち 2014 年に更新されたファイルの数

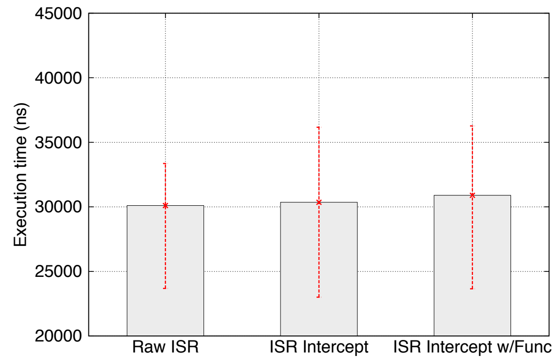


Fig. 5. Interrupt intercept overhead

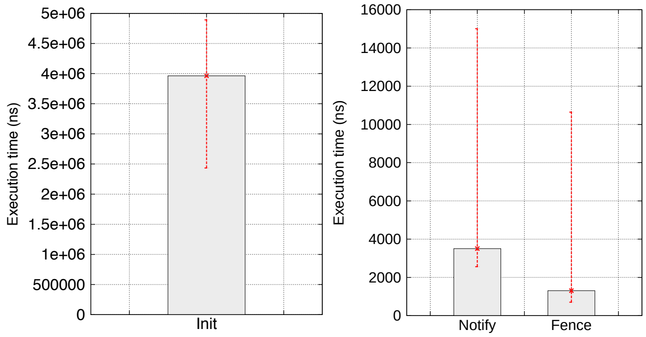
4.2 Interrupt intercept overhead

まず Interrupt intercept のオーバヘッドの測定を行う。本評価では、GPU ドライバは nouveau を用いる。割り込み処理は、各割り込みの種類によって、処理時間が異なり、その分布は一様ではないため、単に測定して平均をとっても比較ができない。そのため各割り込みの種類の判別のために Nouveau を用いて、割り込みの種類が同一のもので、カーネル内の do_IRQ 関数内でハンドラが呼ばれてから終了までの時間を測定しどの程度のオーバヘッドで割り込みの盗聴及び、盗聴した割り込みがいずれのカーネルに関連したものであるかの識別ができるかどうかを測定する。

Figure **??**は上記設定で測定した結果である。Raw ISR は通常のルーチンで実行される ISR、ISR Intercept は割り込みを盗聴するのみ、ISR intercept w/Func は盗聴した上でその割り込みが、いずれのカーネルに関連した割込みか識別しスケジューラを立ち上げる機能を実行した場合である。それぞれ 1000 回の測定で平均値を取り、最小値と最大値についてエラーバーで示している。この図から見て取れるように、オーバヘッドは確実に存在する。ISR Intercept だと 247ns のオーバヘッドであり、ISR Intercept w/Func でも 790ns のオーバヘッドである。この数値は直感的に考えると小さくシステム自体に影響を及ぼすほどではないと考えられ、しかしその割込みが乱発することによる積み重ねによっては影響を与えることは、本手法のデメリットとして意識しなければならない。

4.3 Independent generate sign for Synchronization overhead

本稿では同期用サイン生成のためのオーバヘッドを測定する。割込みの立ち上げは同期を求



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

Fig. 6. Interrupt raised method overhead

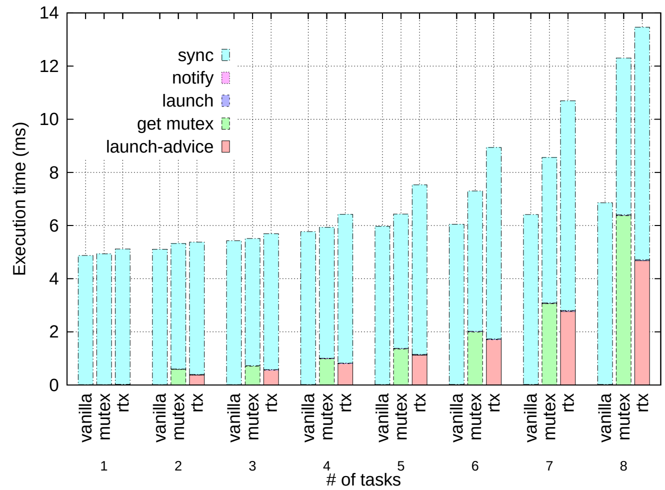


Fig. 7. Scheduling overhead(between GPU kernel launch request and synchronization)

めるタイミング (e.g. カーネルラウンチ後) に rtx_nvrm_notify() を呼び出す必要がある。スケジューリングを行わない Vanilla な状態ではこれらの API は必要ではないものであるため、これらの API にかかった時間はすべてオーバヘッドとなる。

そのためこれらのオーバヘッドの計測を行う。計測は API の呼び出しから戻るまでを測定する。

結果を Figure **??**に示す。Initialize は Indirect Buffer はプロセスが立ち上がるたびに、コマンド送信用の Indirect Buffer の確保や各エンジンの登録のために呼び出される必要がある。Notify はカーネル実行後や非同期メモリコピー実行後のような実際に割込みを発生させたいタイミングで呼び出される。これらは ioctl システムコールによってユーザ空間とカーネル空間をまたいでる影響か、実行時間のバラ付きが大きく出ている。

Initialize は比較的時間がかかっているが、1 プロセスにつき一度しか呼ばれないため、アプリケーション全体への影響は少ないと考えられる。Notify に関してはそれほど時間がかかっておらず、同期待ちの間に実行されるべき処理なため、こちらもアプリケーション全体への影響は少ないと考えられる。Fence についても Notify と同様に平均で 2000ns 以下とほぼ誤差といってもよい程度の時間である。

4.4 Scheduling Overhead

rtx でスケジューリングした場合のオーバヘッドを測定するために、”vanilla”, ”mutex”, ”rtx”の３種類のアプリケーションを用意した。全てに共通するのが、1 個のアプリケーションに複数のタスクが存在しており、各タスクには１０個のジョブが含まれることである。1 個のジョブは GPU へのデータ転送、GPU カーネル実行、GPU からのデータ転送を含んでいる。GPU カーネルは単純な行列の計算を行う。

３種で異なる点として、まず mutex は同時に launch が発行されるのが 1 つに調停されるように mutex を用いてロックしたバージョンである。そして rtx は linxu-rtxg を用いて実行したケースであり、vanilla はそれらの追加が無くスケジューリングや調停を一切行わないケースである。

CPU のスケジューリングは linux-rtx を用いたシンプルな Fixed-priority スケジューリング (Linux の SCHED_FIFO と同様のポリシー、ジョブ管理のみを行う) を用いる。GPU 側のスケジューリングは、Gdev で提案された BAND スケジューラ、Linux-RTX での同期は全て NOTIFY を用いて行う。

計測結果を Figure **??**に示す。アプリケーションに含まれるタスク数ごとにプロットしており、各ジョブ内のラウンチ要求から実行完了までにかかった時間の平均値を各処理毎に積み上げ式で示している。

TODO:結果について説明と、考察

launch_advice は rtx_gpu_launch によって GPU 利用のためのリクエストを出してから、許可がでるまでを示しており、get_mutex は mutex によってロックを獲得するまでの時間、launch、notify はそれぞれコマンドを発行するまでにかかった時間で、sync は発行されてから同期完了するまでの時間である。全て、100 回のアプリケーション実行（$numberoftasks$）

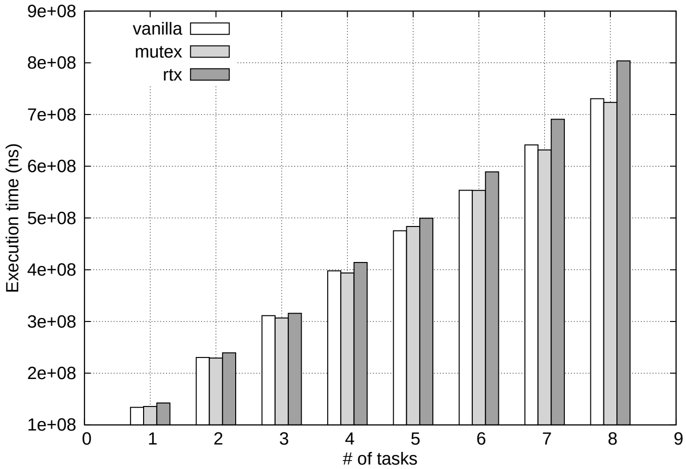


Fig. 8. Scheduling overhead (Time of entire task)

4.5 QoS Management

次にGPUのバジェットエンフォースメントの性能を評価する。ここでは、今回同一アルゴリズムでQoSマネージメントを行っているGdevとの比較を行い、パッチを利用しない実装においても、性能をほぼ落とすこと無くできていることを示す。

比較対象は、本Linux-RTXGと同様のスケジューリングアルゴリズムが提供可能なGdevのモジュール版とで比較する。評価に用いるスケジューリングポリシーはBANDスケジューラを用いる。実験に利用するアプリケーションとして、**??**節で利用したものと同様のものでTaskを4つ生成し、各タスク毎に25%のGPU利用権限を与える。これらのタスクの実行中のGPU利用率を計測し、Gdevと同様のアルゴリズムを用いることで、今回提供するLinux－RTXGによるアプローチによってどれだけQoSマネージメントについてのパフォーマンスに影響するかを示す。Gdevを用いることから、両者ともデバイスドライバはNouveauドライバを用いる.

Figure **??**,**??** show gpu usage on the qos management by gdev and linux-rtxg. TODO:結果に合わせて記述

Figure **??** shows

5 Related Work

Comparison of prior work 我々はこれまでいくつかのGPUの資源管理に関する研究 [**?**], [**?**], [**?**], [**?**] を行ってきた。TimeGraphはGPUに送信されるコマンドをスケジューリングすることでCUDAにかぎらず、OpenGLなど、全てのGPUを利用に関する資源管理を行っている。しかしながらGPUのコマンドは処理の実行だけでなく、データ転送、割込み処理登録などの処理時にも送信されており、本当にスケジューリングするべき単位でのスケジューリングには向いていないことがわかっている。そのため、RGEMはGPGPUに特化し、GPUカーネル実行単位でのスケジューリングを目指し、固定優先度でのスケジューリングを実現してる。加えて、データ転送のセグメント分けによってノンプリエンプティブな特性にもたらされるデメリットを最小限にし、レスポンスタイムの向上を目指している。GdevはRGEMの発展形であり、仮想GPUとResource ReservationによるQoS制御や、OS空間でのCUDA実行などを実現している。加えてデータ転送とカーネル実行をオーバラップさせることで実行時間自体の縮小を実現している。

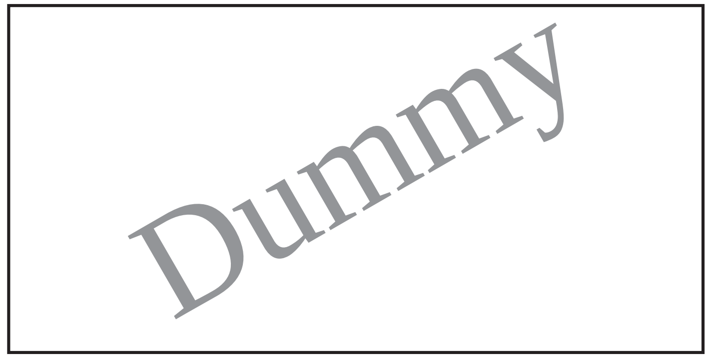


Fig. 9. Task Isolation Performance on the Gdev management

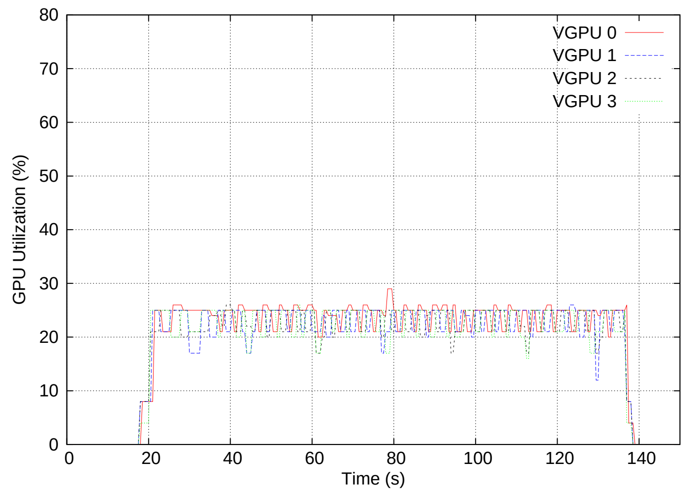


Fig. 10. Task Isolation Performance on the linux-RTXG Management

PTask は [?]

Elliott et al. present GPUSync[?], [?], robust tasklet handling, . GPUSync ではホストから GPU へのデータ転送開始から、GPU での処理、GPU からホストへのデータ転送までをクリティカルセクションと設定し、runtime へのアクセスはクリティカルセクションを区切りとして単一のアクセスとなるように調停を行っている。これによってクローズドソースなランタイムを利用しつつ、自身の GPU 資源管理を実現可能としている。GPUSync はアクセス調停の手法として k-exclusion lock の拡張を利用している。加えて各 GPU ごとに Resource Reservation による QoS 担保を行っており、各 GPU 間の P2P migration も実現しており、MultipleGPU への自動割り当ても行っている。

Han et al. show GPU-SPARC[?]. 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 table**??** 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 をよりリアルタイムにするために必要と考えた、GPU タスク（running on the CPU）のスケジューリング拡張、GPU カーネルのスケジューリング拡張について取り組んでおり、GPU の同期に用いられる割込みの top-half を傍聴することで実現した。今回実現したフレームワークは、ジョブ一個あたりのオーバヘッドは約 x%(sleep している時間も含む) になり、task あたりのオーバヘッドは約 x%に収まることを示した。加えて既存フレームワークである Gdev と同一アルゴリズムを用いて、割込み傍聴を用いた QoS Management の性能を検証したところ、約 x%のオーバヘッドに収まることを示した。

フレームワークとしては、GPUSparc などでも用いられるカーネル分割による擬似プリエンプティブスケジューリングへの対応などのアプローチを統合することでより完成度を高めることができる。本フレームワークはリアルタイム GPU を実現するための基盤としてメリットを持っているが、リアルタイム GPU を完全に実現しているわけではない。今後本フレームワークを利用してどれだけの成果を得られるかが重要である。

ACKNOWLEDGMENTS

The authors would like to thank...

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.

[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 (CPSNA), 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.

TABLE 2
Linux-RTXG vs prior work

	CPU		GPU		Budget	Data/Comp.	Auto GPU	Closed Src.	Kernel	Configurable
	FP	EDF	FP	EDF	Enforcement	Ovlp.	Allocate	Compatible	Free	
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

[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] 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.

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

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

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

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.