Loadable Real-Time Extension for CPU/GPU Resource Coordination

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 は汎目的アプリケーションを加速させるデバイスとして既に一般的になりつつある。その応用範囲は自動運転に用いるナビゲーション [1] や障害物検知 [2]、核融合炉に用いる Tokamak の制御 [3]、ユーザインタラクティブなアプリケーション [4]、データベース [5] まで多岐にわたっており、ベンチマーク suit[6] に提供される。これらの研究によって GPU の性能は既に実証されており、広いアプリケーションドメインに受け入れられている。

より普及した要因としては GPU が大量のプロセッシングコアを用いてデータ並列性のあるアプリケーションを実行することで高速な処理が可能であること、それらを支える CUDA[7] や OpenCL[8] などの言語やランタイムなどが統合されたプラットフォームがベンダーによって提供され始めたことが背景にある

しかしながら公式ベンダーから提供されるランタイムシステムでは、綿密な資源管理機能が保証されておらず、汎目的利用は可能かもしれないが、Multitasking system では利用が困難であることから、GPU 資源管理ソフトウェアが必要となっている。加えて、近年ではサイバーフィジカルシステムなどリアルタイム性を要求するシステムにおいても GPU の利用が望まれており、汎用システム向けだけでなく、リアルタイムシステム向けの資源管理が求められる。

我々はこれまでに、いくつかの GPU resource management に関する研究を進めてきた。TIMEGRAPH[9] では GPU に送信されるコマンドを each atomic set にグルーピングした GPU command group を対象として、スケジューリングやリザベーションメカニズムを提供している。RGEM[10]、Gdev[11] では GPGPU にフォーカスし、GPU で実行される単位であるカーネルとデータ転送をスケジューリングしている。これらの研究はリバースエンジニアリングによって提供されており、アーキテクチャの更新や、全ての機能の提供が困難であるといったデメリットが存在している。

GPU の機能の多くは API によって提供されており、ユーザアプリケーションからライブラリを通じて、デバイスドライバ、GPU へと処理が発行される。そのためリアルタイムを前提としたシステムにおいて、真にその要件を満たすためには、GPU のリソースマネージメントだけでなく、ホスト側のタスクについても管理してやる必要がある。我々がホストとデバイス間のデータ転送時間について [12] 評価した研究では、CPU 側で他のタスクがリソースを専有していた場合、多量のレイテンシが発生することが実際に確認できている。

Elliott らの提案する GPUSync[13][14] では資源管理システムを既存ランタイムから分離し、アクセスの調停を行うことで、プロプライエタリ・ソフトウェアが行う資源管理をスルーした上で自身の資源管理を行っている。GPUSync では上記のホスト側のタスクに関する資源管理についても言及しており、GPU 側の資源管理との組み合わせによる検証を行うためにコンフィギュアラブルなものを目指している。

しかしながら GPUSync は $LITMUS^{RT}$[15] 上に実装されており、カーネルへの変更を多分に含んでいる。Gdev についても同様にデバイスドライバ自体の変更を必要としている。これらの変更の多くはパッチを利用して、ユーザへインストレーションを要求する形が用いられるが、このパッチには開発者とユーザ両者に大きな負担を与える。具体的には開発者は、常に最新のカーネルのリリースに追い付くために、パッチを維

- *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.*

持していく義務がある。しかしLinuxはその更新頻度が早く、開発者が最新のカーネルにむけてポーティング作業を完了させる前に、新しいバージョンのリリースが起きることが多い。そのためカーネルの選択について制限される傾向がある。

カーネルの修正を含む問題に対して我々はこれまでCPU側のタスクをスケジューリングするためのリアルタイム拡張としてRESCH[?]を提供してきた. RESCHはローダブルカーネルモジュールを利用しており、カーネルの内部関数を利用することで、カーネル自体にパッチを当てること無くリアルタイム拡張可能としている。しかしながら、RESCHはGPUを保持するシステムのようなヘテロジニアスな環境には対応していない。

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の最大きな貢献は、OS機能を持ちながら、OSから独立することで、既存のGPU資源管理に関する研究において考慮されていないカーネル編集に伴う開発者、ユーザへの負担をなくしたことである。

加えて、既存研究において既に取り組まれている技術を含有し、これまでのフレームワークからより簡易にコンフィギュア可能な構成を保持することで、今後GPUが含まれるシステムにおける資源管理に関する研究を加速させることを最終目標とする。本論文においては本手法利用時に発生するオーバヘッドを提示し、利用可能範囲内であることを証明する。

TODO:細かい部分

Organization This paper rest of 本論文は7章で構成される。

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

GPUを汎用的に利用する場合多くはCUDA、OpenCLが用いられる。本稿ではCUDAに限定して記述するが、OpenCL等でも同様のアプローチが利用可能である。

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*).

我々は本稿においては、GPU実行が少しでも含まれるプロセスであり、ある事柄を成し遂げる一つの単位をアプリケーションとする。 (e.g. pedestrian detection application). またこの一連の流れによってGPUを実行する1プロセスをGPUタスクとし、GPU側で実行されるカーネルをGPUカーネルと定義する。

2.2 Scheduling Discussion

リアルタイムOSは古来より多くの研究[16], [17], [18], [19]が行われてきている。Linuxをベースとしたリアルタイム OSとしても数多く研究[20], [21], [22], [15], [?]されている。GPUを利用可能な環境はWindows, Mac OS, Linuxと限られており、リアルタイムという性質を追求するためにはLinuxの利用が最適であるため、今回我々はlinuxを利用する。

現在のシステムではGPUはI/Oデバイスとして利用される。これまでに行われているRTOSに関する研究においても、リソースカーネルなどでI/Oデバイスに関する研究は行われているが、GPUをリアルタイムに扱うためには、これまでの知見だけでは不十分である。これまでの一般的なディスクシステムなどのI/Oとの違いを以下にまとめる。

- データの送信、実行、データの受信と順序が固定されている且つそれぞれ独立した処理であること
- ノンプリエンプティブであること
- データ内容に依存しないため、複数のGPUが搭載された時に、どのGPUで実行してもよいこと

以上の特性はGPUをリアルタイム且つ効率的に利用するにあたり問題を複雑化させる。具体的には、データの送受信、実行がオーバラップ可能なことで、スケジューリングが複雑化すること、ノンプリエンプティブによってオーバランを防ぐことが実質不可能であること、GPUへの自動割り当てや静的割り当てなど資源管理の複雑化が発生する。

加えてGPUタスクはデバイスな特性からSelf-suspending[23], [24]を発生させる。Self-suspensionなタスクにおけるスケジューリング解析については多くの研究がなされているが[25], [26]、一般的な利用を行うためのハードルはあまりにも高い。

したがって、本誌ではまずはGPUをよりリアルタイムに扱うための環境を整えるために、GPUタスクのスケジューリング且つ(マルチコア)、GPUスケジュー

リングが可能な基盤を、より簡易にインストールができ、より簡易にコンフィグ可能な形で提供する。

Scheduling policies: ここでは上記スケジューリング基盤として備えるべきスケジューリングポリシーについて議論する。CPU 側で動作するタスクに関してはリアルタイムなシステムにおいては、優先度スケジューリング且ができていればよい。固定優先度方式として Deadline-Monotonic (DM)[**?**] や Rate-Monotonic (RM)[27] などがあり、動的優先度方式として Earist deadline first (EDF) などがある。

GPU カーネルの実行に関しては優先度ベースなスケジューリングでは不都合が生じる。GPU カーネルは GPU タスクに含まれるものである。そのため GPU カーネルの優先度と GPU タスクの優先度が異なるケースは、その時点で GPU タスクの優先度逆転が発生する。したがって、GPU カーネルスケジューリングは基本的に FIFO でスケジューリングすることが適正であり、それに加えて必要なのが各 GPU タスクごとの GPU リクエストに対する QoS を担保することである。QoS を担保する方法としてこれまでリソースリザベーションベースのスケジューリングポリシーが提案 [**?**], [**?**], [**?**] されてきており、本論文では Gdev で GPU の QoS 担保に一定の成果を挙げた BAND スケジューリングを利用する。

No patched scheduler: To achieve "No patched", we must not to modify the kernel code and the device driver code. RGEM はユーザスペースのみで GPU カーネルラウンチのスケジューリングを行っている。GPU タスク間で POSIX によって提供される IPC を用いてタスク間の協調を行い、優先度ベースでのスケジューリングを実現している。GPU のスケジューリングは同期ベースなスケジューリングを行うが、RGEM では同期の取得をランタイムに依存している。

同期の取得を行うように変更するためにカーネルスペースからのアプローチが必須である。Gdev ではカーネルスペースからのアプローチを行っているが、同期の取得のためにカーネルへの変更を含んでいるため、同期の取得をノーパッチで行うことが大きな課題である。

GPU Synchronization: GPU を利用するシステムのようなヘテロジニアス且つメモリを共有しない場合に大きな課題となるのが同期である GPU の同期は 2 つの手法がある。一つは Polling を用いた方法で本誌では FENCE と呼ぶ。もう一つは Interrupt によって同期する方法で本誌では NOTIFY と呼ぶ。これらは両者とも GPU に搭載されたエンジンを用いて行われる。GPU には多くのエンジン（マイクロコントローラ）が搭載されている。本論文では詳しいアーキテクチャはメインではないので省略するが、詳細は過去の文献 [9], [28] に記載しています。このエンジンにはコンテキスト管理用、コマンド受け取り用、データ転送用などが存在している。通常、コマンド受け取り用のエンジンが受け取ったコマンドのヘッダから、そのコマンドを使用するエンジンへと送信し、処理が行われる。この順序はすべて FIFO で行われる。FENCE を用いた方式では、まず同期用に仮想アドレス空間にマップされたバッファを GPU メモリに用意する。そして、このメモリに値を Engine 経由で書き込むためのコマンドを発行する。Engine はカーネル実行とメモリ転送終了後に値を書き込むため、CPU 側でマップされたメモリアドレスをポーリングしながらチェックすれば同期が可能である。NOTIFY についても Engine の機能を利用する。タスクは TASK_INTERRUPTIBLE or TASK_UNINTERRUPTIBLE にした上で schedule() を呼ぶか、waitqueue などを用いて上記に相当する処理を行い suspend する。そして Engine から割込みを発生させるコマンドを発行し、割り込みコントローラがそれを獲得、GPU ドライバが登録した $InterruptServiceRoutine$ (ISR) を立ち上げる。ISR 内では、各割込みに関するステータスをマッピングされたレジスタから読み込み、各割込みごとに処理を行う。処理後は割込み完了フラグを書き込み、初期化する。Gdev では synchronization ベースのスケジューリングを行っており、GPU カーネル終了の検知し次のタスクを立ち上げる部分に割込みを用いている。

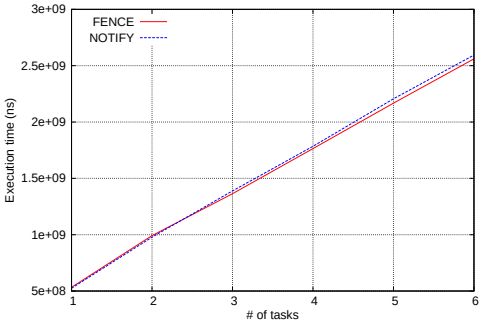


Fig. 1. FENCE vs NOTIFY

二種類の手法が存在する理由としては、FENCE と NOTIFY にそれぞれアドバンテージがあるためである。Figure 2.2 shows FENCE を 1 とした時の NOTIFY の relative speed である。タスクが 1 つの場合は FENCE が NOTIFY よりも 8ms 速い結果が出ており、タスクが 3 個に増えた時点で NOTIFY の方が早くなり、タスク 6 個の時点では 33ms 速い結果がでている。NOTIFY によってタスクがスリープしている間に他のタスクの CPU 利用部分が動作することで、効率的に GPU タスクが実行できているためである。GPU をより効率的に利用するためには FENCE、NOTIFY をうまく使い分ける必要がある。

本稿で対象とするプラットフォームでは、複数の GPU タスクが存在し、複数の GPU カーネルが同一タスクに存在することを想定している。そのため複数の GPU カーネルと一般的に同期に利用される CUDA

API の $cuCtxSynchronize()$ の回数は一対一で対応付けされずに、複数の GPU カーネルの発行を一度の $cuCtxSynchronize()$ によって同期する可能性がある。そのため、$cuCtxSynchronize()$ を用いた同期ベースなスケジューリングを行うにあたり、各 GPU カーネルの終了と同期のために、アプリケーション自体の変更を必要とするため好ましくない。

本問題に対して GPUSync では NOTIFY に限って対処しており、$LITMUS^{RT}$ の拡張という形で Linux の割込みの bottom-half である tasklet をカーネル内部で傍受し、コールバック関数の引数のポインタが指すメモリスペースによって、どの GPU からの割込みかを判断している。その手法では、soft-irq を利用しているためレイテンシが大きく、どのカーネルであるかの判断もできない。Gdev では GPU タスクの同期は FENCE、スケジューラの立ち上げは NOTIFY を用いている。NOTIFY の獲得は、デバイスドライバがカーネルに登録する ISR に Gdev のコールバック関数を追記することで割込みタイミングを獲得している。GPU タスクの同期に FENCE を用いている理由としては、あくまで GPU を効率よく扱う点にフォーカスしており、CPU 側の処理も含めた効率について考慮していないためである。カーネル、デバイスドライバの編集無しにスケジューリングするための最も大きな課題は、この同期を獲得し、識別するかといった点である。

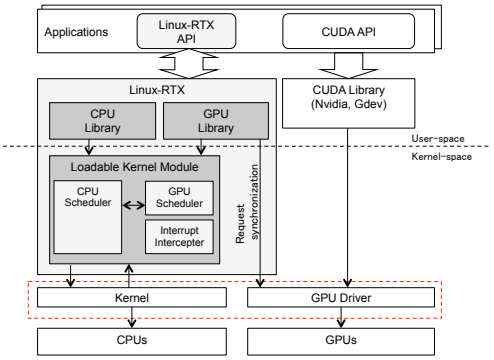


Fig. 2. Over view of the Linux-RTXG

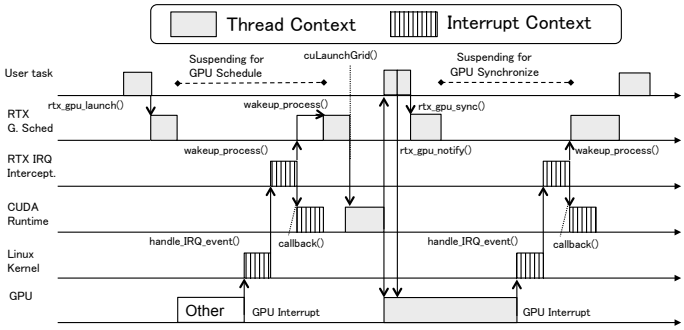


Fig. 3. GPU Scheduling control flow

3 Design and Implementation

In this section, we present linux-rtxg design and implementation. Linux-RTXG は Linux Real-Time eXtension included GPU resource management の略称であり、Linux のリアルタイム拡張に加えて GPU の Resource マネージメントを行うためのフレームワークである。

Linux-RTXG はベースとして RESCH を用いているため、CPU スケジューラに関する記述は最小限に抑え、本論文の大筋である、GPU スケジューラをメインに、CPU スケジューラとの統合といった部分を記載していく。

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.

3.2 GPU Scheduler

Linux-RTXg's scheduler function is provided RTXG API. The basic APIs supported by Linux-RTXg are listed in Table 1. Some APIs have arguments and others do not. Our API is 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.

Figure 3.2 shows control flow of the wakeup GPU job on the linux-rtxg's gpu scheduler. The configure is GPU カーネルの発行はひとつに制限されており、すでに GPU でタスクがうごいている状態とし、同期は割込みを用いた NOTIFY によって行うものとする。ユーザタスクは $rtx_gpu_launch()$ を呼び出すことで、GPU カーネル実行のタイミングをコントロールすることができる。設定通り、既に GPU でタスクが動いており、現在カーネル発行を許可されているタスクが自身でないため、割込みによって起床されるまで task

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 register 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.

uninterruptible 状態で自らスリープに入る。動いていたタスクが終了した時点で割込みが発行され、そのコンテキストの Interrupt intercepter によってスリープしていた次のタスクが起床される。

起床したタスクは $cuLaunchGrid()$ などの CUDA API を通じて GPU カーネルの発行を行う。カーネル発行後、割込みを発生させるための Notify の登録を行い、その割込みが発生するまでスリープに入る。

割込みが発生すると次のタスクが動作するといったフローを持ってスケジューリングが行われる。次のタスクの選択は、割込みによって起こされる GPU scheduler によって行われる。

3.3 GPU synchronization

本スケジューラ synchronization based なスケジューラであり GPU カーネルのラウンチを要求されたタイミングと、そのカーネルが終了したタイミングを知る必要がある。Linux-RTXG では前述のように rtx_gpu_launch() によって GPU カーネルのラウンチ要求を受け取る. アプリケーションは rtx_gpu_launch() を呼び出すことで ioctl システムコールによって、ユーザコンテキストからカーネルコンテキストへと処理が移り、スケジューラが保持する GPU タスク情報 (e.g. waiting task status, running task status, GPU device status) を基に自身が動作可能かを確認し動作する。スケジューラを通じて、資源を獲得できた場合には GPU カーネルのラウンチの発行が可能となる。

ラウンチされたカーネルが終了したタイミングをスケジューラは NOTIFY か FENCE によって取得する。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: 割込みはデバイスドライバ（カーネルと共にパッケージされている）によって登録された ISR がハンドルする。

Linux-RTXG ではタスクの選択のためにワーカースレッドを保持する。本ワーカースレッドは実行中のカーネルが終了した時点で次のタスクの選択を行う。ワーカースレッドはタスクの選択後に CPU 資源を他のタスクに明け渡すためにサスペンドに入る。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 では、割込み番号ごとに irq_desc という割込みのパラメータを保持する構造体を持っている。この構造体には ISR の関数ポインタを含む irq_action という構造体がリストで接続されている。irq_desc はグローバルな領域に確保されており、カーネル空間からであれば誰でも参照可能である。Linux のローダブルカーネルモジュールはカーネル空間で動作しているため、この irq_desc を取得でき、Interrupt handler の関数ポインタも取得可能である。

我々はこの取得した関数ポインタを保持し、我々の傍受用 ISR をカーネルに登録する。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 [12], [29] 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 BAR０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: ここでは、ランタイムから独立した割込み機構として、独自に NOTIFY、FENCE に用いる sign を発生させる仕組みを提供する。ここでの sign は、NOTIFY は割込み、FENCE は mapped メモリへの値の書き込みである。NVIDIA のクローズドソースドライバは Nouveau プロジェクトのリバースエンジニアリングによる解析によって、ioctl を使ったインタフェースになっていることがわかっている。Gdev ではこの解析された情報を用いて、NVIDIA のクローズドソースドライバとオープ

ンソースライブラリという掛け合わせでCUDAを実行できる基盤が構築されている。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。

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は、いわゆるコンテキストの生成に値する。Virtual Address Spaceやコマンド送信に用いるIndirect Bufferの確保、コンテキストオブジェクトの生成などを行う。NotifyはComputeエンジンやCopyエンジンに向けて割込み発生、もしくはFENCE用に値の書き込みを行うコマンドを送信する。

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. 特にSCHED_DEADLINEはLinux 3.14よりメインラインに含まれたConstant Bandwidth ServerとGlobal-EDFの実装であり、Linuxをリアルタイム拡張するにおいて有効に利用できるクラスである。However, synchronization does not work well in a SCHED_DEADLINE scheduling policy when using GPU tasks.

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。

一つ目のsched_yield(カーネル内ではyield関数)は、FENCEのようにPollingされる場合に生じる。pollingはCPUを専有してしまう方式であり、他のタスクへの影響を考えた場合、一度CPUを放棄したほうが良い結果が得られる場合がある。しかしながらsched_yieldでは、SCHED_DEADLINEの内部パラメータとして扱う、runtime（残りの実行しても良い時間）を0にしてしまう。これによって、次の周期が訪れるまではruntimeが補充されることがなく、そのタスクは実行権限を失う。sched_yieldはGPUランタイムに限らず、デバイスドライバやライブラリなどで多く利用されており、それら全てで、SCHED_DEADLINEポリシー上では正常に動作しない結果が生じる可能性が高い。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。

2つ目の問題はタスクが一度sleeping状態に入り、復帰時に式(1)を用いて実行可能性についてチェックを行う。式(1)が真の時、runtimeが補充され、absolute deadlineが次の周期に設定される。

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

linux-rtxgでは本チェックについて、sleeping状態から起床するという状態を、GPUカーネル実行による復帰と、周期による復帰とで種類分けし、GPUカーネル実行による復帰時にのみ、$Remaining_Runtime$からGPU execution timeを引くことで対応した。

4 EVALUATION

我々の知る限り、カーネルやドライバを修正しないという観点を意識してGPU資源管理に取り組んだ例は無い。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。

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

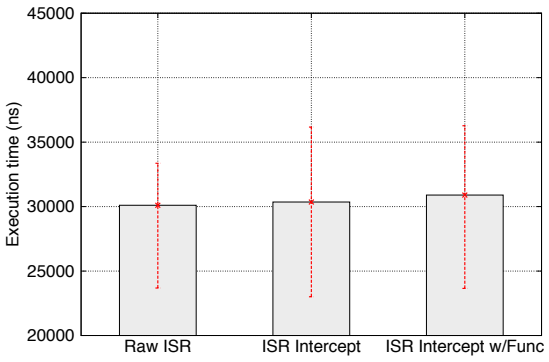
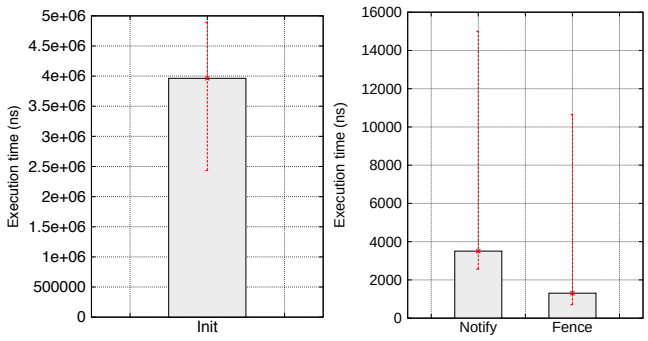


Fig. 4. Interrupt intercept overhead



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

Fig. 5. Interrupt raised method overhead

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

TODO:fence についても

本稿では割込み立ち上げ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めるタイミング (e.g. カーネルラウンチ後) に rtx_nvrm_notify() が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 5 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。

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 4.4 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（$numberoftasks$）

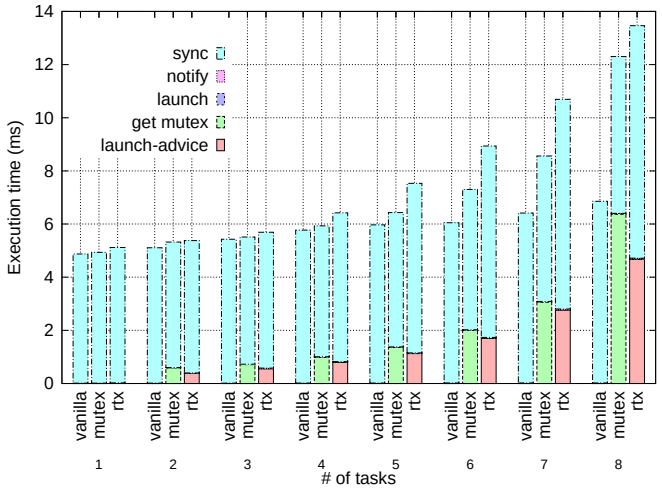


Fig. 6. Scheduling overhead(between GPU kernel launch request and synchronization)

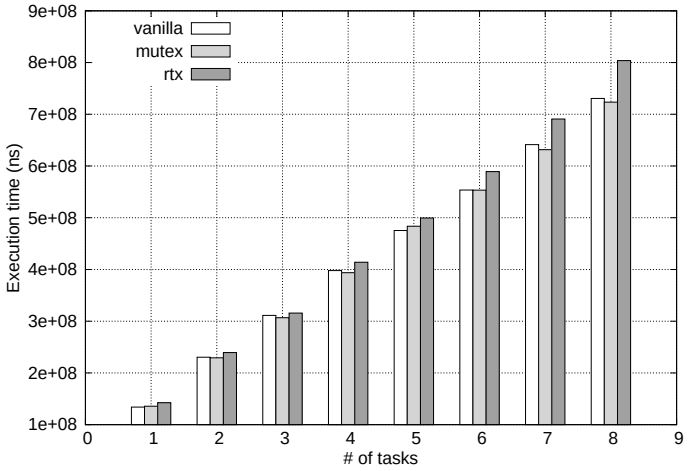


Fig. 7. Scheduling overhead (Time of entire task)

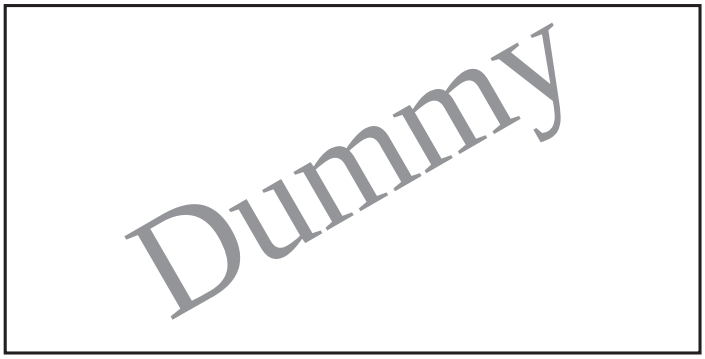


Fig. 8. Task Isolation Performance on the Gdev management

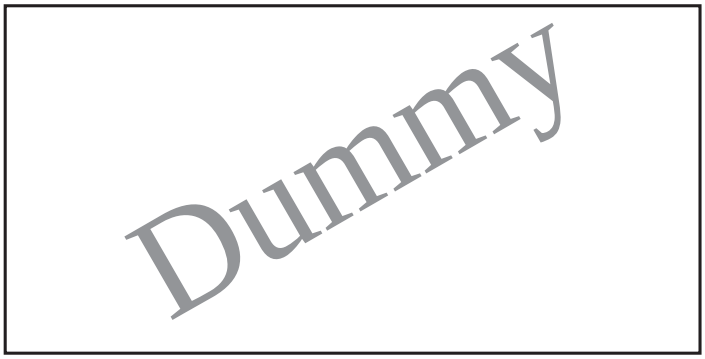


Fig. 9. Task Isolation Performance on the linux-RTXG Management

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、4.4 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 4.5,4.5 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 [9], [4], [10], [11] 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。

PTask は [30]

Elliott et al. present GPUSync[31], [14], 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[32]. 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 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 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

[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] 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

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

Takuya Azumi Biography text here.

Nobuhiko Nishio Biography text here.

Tuyoshi Hamada Biography text here.

Shinpei Kato Biography text here.