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ことが見られる。これだけの性能を持ち、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

- *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.*

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 $LITMUS^{RT}$[] 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 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

What is the best scheduling mechanism (policy and implementation) on GPUs?: We are very difficult to get the answer, it because verification is not sufficient. The GPU has several factors different from the tradisional architecture CPU, these is to have characteristics that are non-preemptive, co-processor, latency and relative data transfer latency. リアルタイム環境下で GPU を利用する場合、上記特性によって self-suspending[15], [16] などの問題も発生させるため、今後さらに研究を進めていく必要がある。

In addition, スケジュール単位によっても状況は大きく変わる。

PTask は

Gdev は CPU 側のタスクには一切関与せずに、一つのデータ転送、一つの GPU カーネル実行それぞれをスケジューリング対象とし、リザベーションベースのスケジューリングを行った上で、runqueue を優先度順に並び替えて実行順序の制御を行っている。Gdev は汎目的 GPU をマルチタスキングな汎用システムで用いることを目指した研究である。そのため、リクエストされたラウンチの管理は可能であるが、ラウンチがリクエストされるまでのタスクの管理は行えない。リアルタイムを目指すとかんがえる場合、如何に GPU 実行の QoS を担保できたとしても機能として不十分であるといえる。

GPUSync では GPU タスク全体をまず、I/O を使うだけの CPU タスクとして解釈しリアルタイムスケジューリングしており、GPU へのデータ転送、GPU カーネル実行、GPU からのデータ転送までを一つのクリティカルセクションと置いてリザベーションベースのスケジューリングとアクセスの調停を併用している。

今後 GPU 研究を進めるにあたり、特に重要と思われる点は、GPU タスクのスケジューリングポリシーと、各カーネルの GPU への割り当て方、メモリマネージメント

-
-

Scheduling policies: Gdev や RGEM は GPU コンテキストに優先度が付いている。GPU スケジューリング

では GPU をスケジューリングする場合、CPU ではいずれのポリシーを選択すれば良いのか、GPU はいずれのポリシにーよって割り当てれば良いのかといった疑問が生じる。GPU のようなヘテロジニアスな環境の場合同期やデータ転送によって、これまでの伝統的なマルチコアプロセッサにおけるコンテキストスイッチなどの遅延と比べて、比較にならないほど多くの遅延が発生することと、コプロセッサという特性から、これまでの知見をそのまま適応することは難しい。

Elliot らが提案する GPUSync などではマルチコア CPU とマルチ GPU、マルチノード環境下において、リアルタイムな実アプリケーションの加速に向けたアプローチを進めており、GPU におけるコンフィギュアラブルなスケジューリング基盤の構築によって、実験環境を整えている。

GPU Synchronization: GPU のようなヘテロジニアス且つメモリを共有しないプロセッサを用いるにあたり、常に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], [17] 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 経由で書き込む用にコマンドを発行する。すると、カーネル実行とメモリ転送終了後にエンジンが値を書き込むため、CPU 側でその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 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、加えて Gdev ではスケジューリングにおいてある GPU カーネル終了を検知し、次のタスクを立ち上げる部分に割込みを用いている。

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。

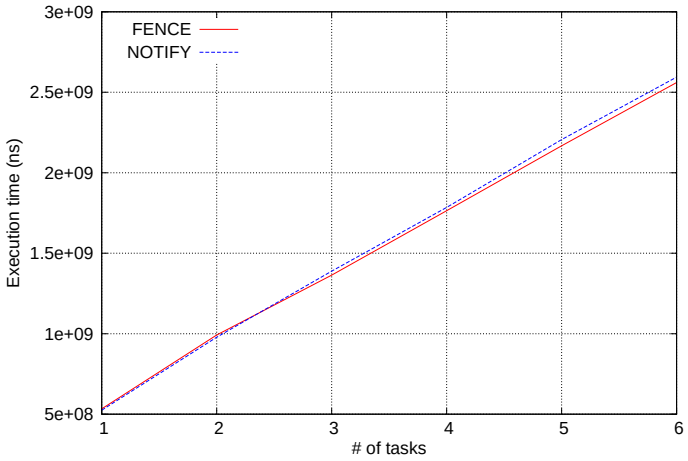


Fig. 1. FENCE vs NOTIFY

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、複数のカーネルが同一タスクに存在しても良いことを想定している。加えて、複数の GPU タスクから GPU カーネルのラウンチを発行できる自由度も兼ね備えたい。そのためスケジューリングには、同期を検知し、どの GPU、どの GPU カーネルが終了したことでの同期なのかを識別する必要がある。

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 による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 を如何に効率よく扱うかとい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、いかにして、この同期を獲得し、識別するかといった点である。

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。しかしながら、同期の取得はランタイムに依存している。同期の取得を行うためにはカーネルスペースからのアプローチが必須である。加えて、前述のように、GPU タスクのリアルタイム性確保のためには、CPU に関するリソースマネージメントについても考慮する必要性がある。

我々はこれまで、RESCH は CPU タスクのリアルタイム拡張を Linux の Loadable kernel module を用いることで No patched で行っている。しかしながら、GPU などのリソースを管理する機能は搭載していなかった。

本稿では、CPU スケジューリングと GPU リソースマネージメントが可能なフレームワークをノーパッチで実現するために、Gdev で培ってきた GPU リソースリザベーションのアプローチを拡張し、ノーパッチで実現した上で、RESCH で行ってきたリアルタイム拡張のアプローチと統合して、CPUGPU リアルタイムエクステンションフレームワークとして Linux-RTXG を提示する。

3 Design and Implementation

In this section, we present linux-rtxg design and implementation. Linux-RTXG は Linux Real-Time eXtension with 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 **??** 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. it is independent Nvidia interrupt raised method (iNVRM). iNVRM is used only on the

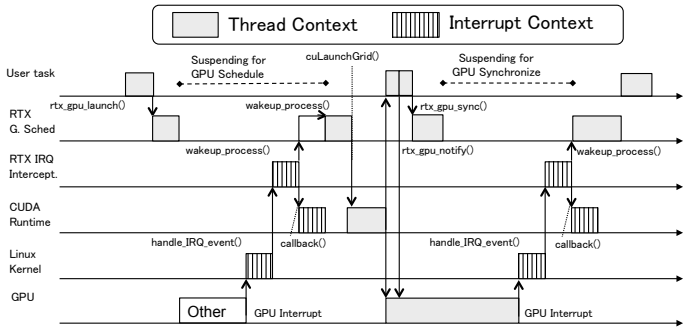


Fig. 2. GPU Scheduling control flow

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

- scheduling mechanism
- interface

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. Figure **??** shows a sample program that is using CUDA API. 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 task 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 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

スケジューラは GPU タスクスケジューリングのために 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

CUDA では cuCtxSynchronize() という API が提供されている。これを用いることで、あるコンテキストに該当するラウンチされたカーネル全てが終了を検知することは可能である。しかしながらスケジューリングにはさらに細かい粒度を要求することも考えられるため、スケジューラ自身が同期を可能とする必要がある。

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

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_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.

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。

上記 SCHED_DEADLINE 時の wait queueue を用いた場合に置いても、たすくの　実行が終わり同期されるまで実行停止状態で待機する。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。

この手法を我々の目的のために適用しようと考えた際、bottom-half では我々の割込み傍受は top-half をオーバーライドする。

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。この構造体には割り込みハンドラ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、我々の傍受用割込みハンドラを設定、コールバック関数を保持している関数ポインタから設定することで、GPU に関する割込みの発生後、先んじて取得が可能である。ただしこの手法は当然のことながら、割込みを遅延させることにほかならないため、オーバヘッドについて綿密な評価を Sec**??**にて示す。

Independent generate sign for Synchronization: NVIDIA のプロプライエタリ・ソフトウェアでは FENCE と NOTIFY をコンテキスト生成時のフラグ生成によって使い分けている。

実際に interrupt interception によって盗聴した結果が**??**である。NVIDIA のドキュメントによると、CUDA はワーカースレッドを立ち上げて、割込みを受け取り、同期を行っている。

しかしながらこの割込みは、我々のアプローチではどのカーネル実行に関連付けされているかが区別できなかった。したがって、ひとつの GPU への複数のアクセスを許可した場合、割込みを利用したスケジューリングが不可能になる。

そのためここでは、ランタイムから独立した割込み機構として、独自に割込みを発生させる仕組みを実装する。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。

本手法は大きく 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の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. We support all scheduling policies that was implemented by linux. However, synchronization does not work well

in a specific scheduling policy when using GPU tasks.

FENCE による同期では積極的に SPIN する方法と、sched_yield() を使って、一度 CPU を開放し、他のタスクの利用率を向上させる方針がある。加えて NOTIFY では self-suspend な状況に入る。

sched_yield() を用いるときに問題が発生する。それは implementation of sched_yield() cede cpu to other tasks, by to set the next deadline to current deadline and to set 0 to budget.

つまり、sched_yield() を用いたポーリングではその周期内での実行を諦めるため、必ずデッドラインミスとなる。しかしながら、sched_yeild() を利用しない場合、同期待ちの間 CPU を専有してしまい好ましくない状態に陥る。

TODO:重要でない部分なのでシンプルに

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。

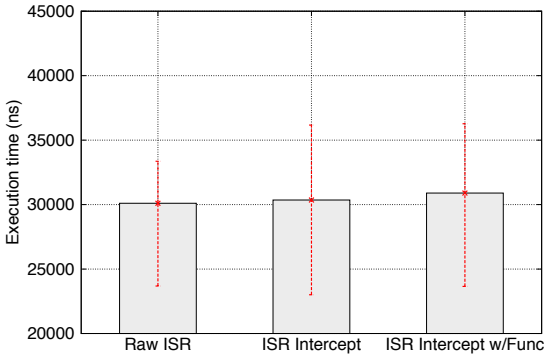


Fig. 3. Interrupt intercept overhead

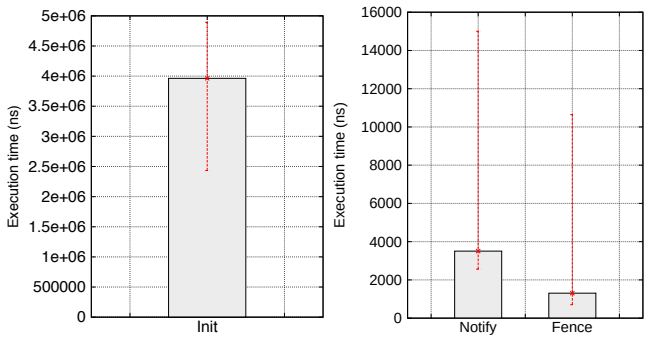
Figure **??**は上記設定で測定した結果である。Raw ISR は通常のルーチンで実行される ISR、ISR Intercept は割り込みを盗聴するのみ、ISR intercept w/Func は盗聴した上でその割り込みが、いずれのカーネルに関連した割込みか識別しスケジューラを立ち上げる機能を実行した場合である。それぞれ 1000 回の測定で平均値を取り、最小値と最大値についてエラーバーで示している。この図から見て取れるように、オーバヘッドは確実に存在する。ISR Intercept だと 247ns のオーバヘッドであり、ISR Intercept w/Func でも 790ns 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

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。



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

Fig. 4. Interrupt raised method overhead

結果を Figure 4 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

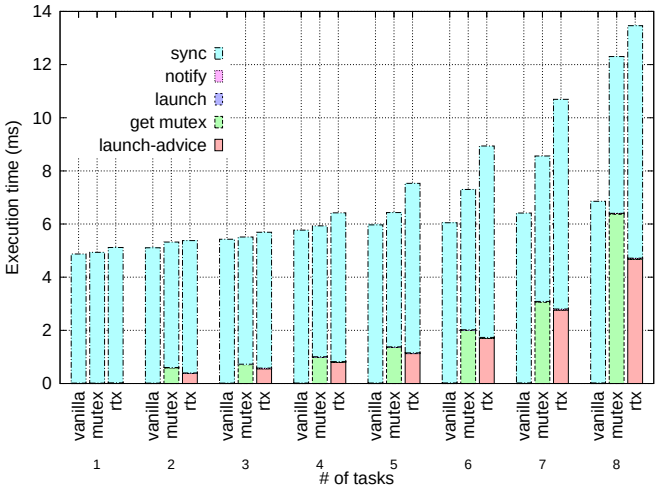


Fig. 5. Scheduling overhead(CPU task scheduling policy is Fixed-Priority scheduling)

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（$number of tasks$）

5 Related Work

我々はこれまで、Timegraph[],RGEM[],Gdev[] として 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

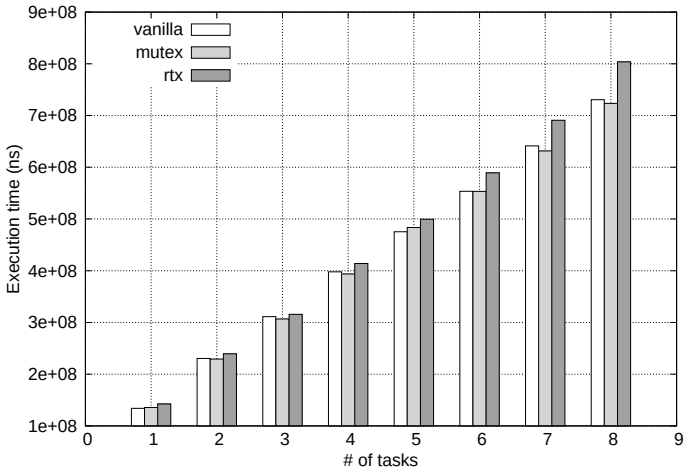


Fig. 6. Scheduling overhead

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。

Elliot 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 を利用している。加えて各GPUごとに Resource Reservation による QoS 担保を行っている。

Han et al. show GPU-SPARC. GPU-Sparc support to automatically split and run the GPU kernel concurrently over multi-GPU,

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

6 Conclusion

In this section.

Futurework.

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.

[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.

TABLE 2
Linux-RTXG vs prior work

	CPU		GPU		Budget Enforcement	Data/Comp. Ovlp.	Multi GPU Aware	Closed Src. Compatible	Kernel Free	Configurable
	FP	EDF	FP	EDF						
Gdev										
GPUSparc										
PTask										
GPUSync										
Linux-RTXG										

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

[15] 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.

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

[17] 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.

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.