Vol.2014-HPC-145 No.42
2014/7/30

GPU間マイグレーションによる効率的な並列実行

鈴木 太一郎[1,a)]　額田 彰[1]　松岡 聡[1]

概要：Hyper-Q によって複数の CUDA アプリケーションが並列実行可能となり，アプリケーションの組み合わせによっては合計実行時間の短縮に期待ができる．GPU のデバイスメモリ容量を複数のアプリケーションで分け合う必要があるため，デバイスメモリ不足に陥る可能性があり，事前にアプリケーションの情報を得られない場合には検出が容易ではない．そこで各アプリケーションが使用するデバイスメモリ量を監視し，適宜アプリケーションの実行中断や GPU 間のマイグレーションを行う機能を NVIDIA のランタイムライブラリを置き換える形態で実装することによって，アプリケーションの変更なく透過的に実現する Mobile CUDA を提案する．4GPU で 100 アプリケーションを実行する場合，Mobile CUDA によって実行時間を 13.1%，消費エネルギーを 2.2%削減できることを確認した．

1. はじめに

GPU は高い演算性能とメモリバンド幅が要求されるグラフィック処理の高速化用として開発された．GPU はシェーダと呼ばれる単純な処理を多数のデータに繰り返し行うことに特化している．かつてはグラフィック処理で行う演算はハードウェアに実装されていたが，その複雑化に伴いシェーダプログラムをプログラミング可能となった．これにより GPU プログラミングが誕生し，GPU の柔軟性と表現力が向上した．近年では GPGPU[1](General-Purpose GPU) と呼ばれる GPU を汎用計算にも用いる技術が注目されている．GPGPU により汎用的なコードを実行することで，CPU よりはるかに高速な計算資源として用いることが可能となった．

大規模システムでは，演算性能は元より消費電力も重要な要素となってきているため，高い電力効率を誇る GPU はそのような分野で大きな役割を担っている．実際 NVIDIA 社の最新の Kepler 世代 GPU を計算資源の主体とする東京工業大学のスパコン TSUBAME-KFC と TSUBAME2.5 は，スパコンの電力効率を競うランキング Green500 List の 2013 年 11 月版でそれぞれ 1 位と 6 位にランクされている．GPU 上でアプリケーションを実行する GPU コンピューティングを実現する実行環境は様々存在している．CUDA[2] もその中の 1 つで，NVIDIA によって提供されている GPU コンピューティングの統合開発環境である．

GPU はデバイスメモリと呼ばれる主記憶とは別のメモリを搭載している．これと区別するために，主記憶はホストメモリと呼ばれる．CUDA アプリケーションが GPU 上で計算を行うためには，デバイスメモリにデータを送信する必要があり，また計算結果をホストメモリに送信する必要もある．また，デバイス側で実行されるコードはカーネルと呼ばれる．

通常の GPU アプリケーションは 1GPU を占有するが，CUDA アプリケーションは GPU を共有して実行することができる．すなわち，複数の CUDA アプリケーションを 1GPU 上で並列実行可能である．NVIDIA の GT200 世代の GPU ではアプリケーション内での通信と計算の同時実行のみが可能であり，複数のカーネルが同時に実行されることはできなかった．また Fermi 世代の GPU では複数のカーネルが GPU の資源を共有できるようになったが，1 つのアプリケーション内に限られた．これに対し Kepler 世代の GPU は搭載されたハードウェア機能である Hyper-Q によって，複数のアプリケーションのカーネルに対し GPU のリソースを割り当てることが可能になった．CUDA アプリケーションは演算量が多い場合や，メモリアクセス量が多い場合などボトルネックになる GPU のリソースがそれぞれ異なるため，これらの同時実行によって GPU のリソースの利用率が向上し合計の実行時間を削減することが可能となる．

Hyper-Q を活用するには，単に複数のアプリケーションを同一 GPU 上で実行すればよい．しかし，一般的にデバイスメモリはホストメモリよりも少量であるため，同時実行されることを想定し作成されていない CUDA アプリケーションはデバイスメモリ不足に陥る可能性がある．そこで

[1] 東京工業大学
Tokyo Institute of Technology
[a)] suzuki.t.cf@m.titech.ac.jp

本論文では，Hyper-Q を活用するための実行環境 Mobile CUDA を提案する．Mobile CUDA はアプリケーションに変更を加える事なくダイナミックリンクライブラリの置き換えによって，透過的にアプリケーションの実行中断・再開や GPU 間移動を行う．これによりデバイスメモリ不足を回避し，より多くの CUDA アプリケーションを同時実行することで Hyper-Q を活用し，アプリケーションの実行時間の合計を短縮する．

2. Mobile CUDA の提案

複数のアプリケーションを実行した際のデバイスメモリ不足を回避する方法は主に 2 つある．1 つは各アプリケーションが使用するデバイスメモリ量に合わせて実行するアプリケーションを制限する方法である．しかし，この方法ではアプリケーションが使用するデバイスメモリ量を予め把握する必要があり，現実には既知である場合は極めて少ない．もう 1 つはアプリケーションがデバイスメモリ不足に陥った際に GPU 上のいずれかのアプリケーションを中断させ，退避させる方法である．この方法はアプリケーションのデバイスメモリ不足を検知できれば実現可能である．

デバイスメモリ不足に陥ったアプリケーションは，他のアプリケーションがデバイスメモリを解放するまで実行を再開できない．しかしより多くのアプリケーションを並列実行するには，実行を中断したアプリケーションが GPU に留まっている状況は望ましくない．そこでそのようなアプリケーションはホストへ退避させ，デバイスメモリに十分な空き容量を確認したら実行を再開させる．複数 GPU を搭載する計算環境では，アプリケーションは別の GPU で実行を再開するケースも考えられる．このような動作を透過的に実現する実行環境 Mobile CUDA を提案する．

2.1 GPU リソースの解放と復元

チェックポイント・リスタートの実装の 1 つである BLCR[3] は，CUDA アプリケーションに対応しておらず，NVCR[4] ではチェックポイントをとる前に CUDA リソースを破壊することで対応している．Mobile CUDA でもホストに退避，及び再開を行う必要があるために類似の手法を用いる．Mobile CUDA は MOCU Library というダイナミックリンクライブラリを導入しており，これによりアプリケーションが透過的に利用することができる．MOCU Library は CUDA の Runtime API の全て関数を実装しており，アプリケーションがそれらの関数を呼び出すと，NVIDIA Library(libcudart.so など) 内の対応する関数へ中継する．

多くの CUDA アプリケーションが Runtime API を利用しているが，Runtime Library から間接的に Driver API を呼び出している．NVCR 自体は Driver API を実装することで，Runtime API を利用するアプリケーションと Driver API を利用するアプリケーションの両方に対応できていた．しかし CUDA のバージョン 4.0 以降，Runtime Library の実装が変更され，この方法ではコンテキストが残留する問題が生じていた．そこで MOCU Library は Runime API レベルで実装を行っており，cudaDeviceReset() 関数によって GPU リソースの解放を行っている．

同じアドレスにデバイスメモリを確保するために，NVCR と同様に Replay 手法を用いる．Replay 手法とは確保したデバイスメモリのアドレスの決定に影響を及ぼす API 呼び出しを記録し，データ復元の際に同じ順序で再呼び出しすることで，同じアドレスにデバイスメモリを確保する手法である．デバイスメモリ以外の各種 CUDA リソースはカプセル化することが可能である．Replay 手法を適応する関数一覧を表 1 に示す．

表 1 Replay 手法で再呼び出しするランタイム API

関数名	説明
cudaMalloc	デバイスメモリを確保する
cudaFree	デバイスメモリを開放する
cudaHostRegister	ホストメモリをデバイスに pin down する
cudaHostUnregister	pin down したホストメモリを解除する

CUDA には GPU からの高速アクセスを可能とするホスト側のメモリ領域として CUDA pinned メモリというものがある．このメモリ領域はデバイスメモリ空間にマップすることも可能である．デバイスメモリ空間にマップされる場合には，Replay 手法の対象にする必要がある．cudaHostAlloc() 関数はホストメモリの確保を行い，その領域を pin down するのに対して，cudaHostRegister() 関数は確保済みのメモリ領域を pin down する．これらの関数に cudaHostAllocMapped フラグまたは cudaHostRegisterMapped フラグが指定された場合に，デバイスメモリ空間へのマップが行われる．ホストメモリの確保を含む cudaHostAlloc() は Replay することができないため，MOCU Library ではホストメモリを確保する valloc() 関数と cudaHostRegister() に分割して処理している．また Mobile CUDA ではアプリケーションが GPU 間を移動する可能性があるため，CUDA pinned メモリが複数 GPU からアクセス可能になるように，cudaHostRegister() 呼び出し時に cudaHostRegisterPortable フラグを付加する．

2.2 デバイスメモリ使用量の集中管理

Mobile CUDA の実行環境は図 1 のようになる．Mobile CUDA はデバイスメモリ使用量を集中管理する方式を採用している．これによりアプリケーションの実行中断，退避，再開などを一意に決定し，オーバーヘッドを減らすことができる．各アプリケーションの MOCU Library が socket

表 2　MOCU Library から Manager へのコマンド一覧

コマンド	説明
CONNECT	最初にアサインされるデバイス番号を要求する.
CUDAMALLOC	cudaMalloc() の要求量以上の空き容量があるかを問う.
CUDAFREE	cudaFree() によってデバイスメモリを解放したことを知らせる.
MALLOC_DONE	cudaMalloc() が正常に終了したことを知らせる.
MIG_DONE	マイグレーションが正常に終了したことを知らせる.
SUSPEND_DONE	バックアップ及び CUDA リソースの解放が完了したことを知らせる.
CONTEXT_CHECK	コンテキストが作成されているかどうかを確認する.
QUERY	Manager からの要求に対して応える
FIN	アプリケーションの終了を通知する

表 3　Manager から MOCU Library へのコマンド一覧

コマンド	説明
CONNECT	アプリケーションを適切な GPU へアサインする.
GOAHEAD	アプリケーションに処理の続行を命じる.
MIGRATE	待機集合に登録されたアプリケーションにマイグレーションを命じる.
SUSPEND	アプリケーションにバックアップ及び CUDA リソースの解放を命じる.
CCHECK_OK	コンテキストの存在が確認されたことを通知する.
CCHECK_FAILED	コンテキストの存在が確認できなかったことを通知する.

を用いて Manager と通信を行い，デバイスメモリ使用量を報告する．Mobile CUDA では，デバイスメモリの空き容量を NVML(NVIDIA Management Library) で取得するため，現在のデバイスメモリ使用量 (used) と追加で要求しているデバイスメモリ量 (req) を分けて考える必要がある．またアプリケーションは Manager からの指示にしたがって動作する．

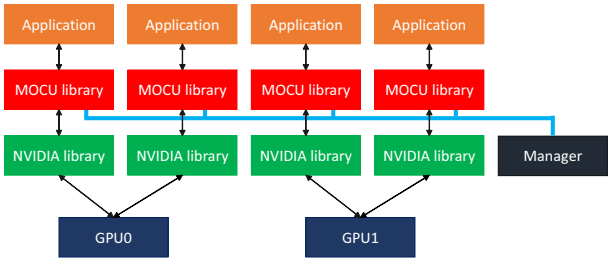

図 1　Mobile CUDA の実行環境

2.3　Manager と MOCU Library 間通信

MOCU Library は通信に伴うオーバーヘッドを抑えるために，デバイスメモリの使用状況に変化があった時のみ通信を行う．デバイスメモリの使用状況に変化があるのはコンテキスト作成時，デバイスメモリの確保 (解放) 時である．コンテキスト作成時には CUDA リソース分 (約 64[MB]) がデバイスメモリ上に確保されると同時に，静的に宣言されたデバイス変数分も確保される．MOCU Library と Manager とのやりとりはコマンドによって行われる．MOCU Library から Manager へのコマンド一覧を表 2，Manager から MOCU Library へのコマンド一覧を表 3 で示す．

2.3.1　CUDA アプリケーション実行の流れ

MOCU Library からのコマンドの送信について記述する都合上，CUDA アプリケーションの実行の流れについて述べる必要がある．CUDA Runtime API を用いるアプリケーションは起動時にコンストラクタからいくつかの関数を呼び出す．まず最初に__cudaRegisterFatBinary() が呼ばれ，CUDA Fat Binary という形式のデータを処理する．CUDA Fat Binary は ELF フォーマットで CUDA の実行可能コードやシンボルなどの情報を保持している．その後，__cudaRegisterFunction() によって CUDA カーネルの呼び出しの準備する．また，__cudaRegisterVar() によってデバイス変数を参照可能な状態にする．これらが終わった後，main 関数へと進む．

2.3.2　予約されたデバイスメモリ容量の管理

Manager は各 GPU に割り当てられているアプリケーションの req 値の合計を予約メモリ量 (reserved) として管理する．アプリケーションが CUDAMALLOC コマンドで追加のデバイスメモリを要求してきた場合には要求量を req に追加し，デバイスメモリの空き容量 (free) から reserved 値を引いた値とこの req の値と比較する．空き容量が十分であれば GOAHEAD コマンドでデバイスメモリの確保を許可する．その後デバイスメモリ確保の完了を示す MALLOC_DONE コマンドを Manager が受け取ると、req を used に加算し req を 0 にする．

実行中のアプリケーションが cudaFree() を呼んだ時，ホストへ退避した時，終了した時にデバイスメモリの空き容量が増える．この時に，待機中のアプリケーションの中から実行可能なものを探す．

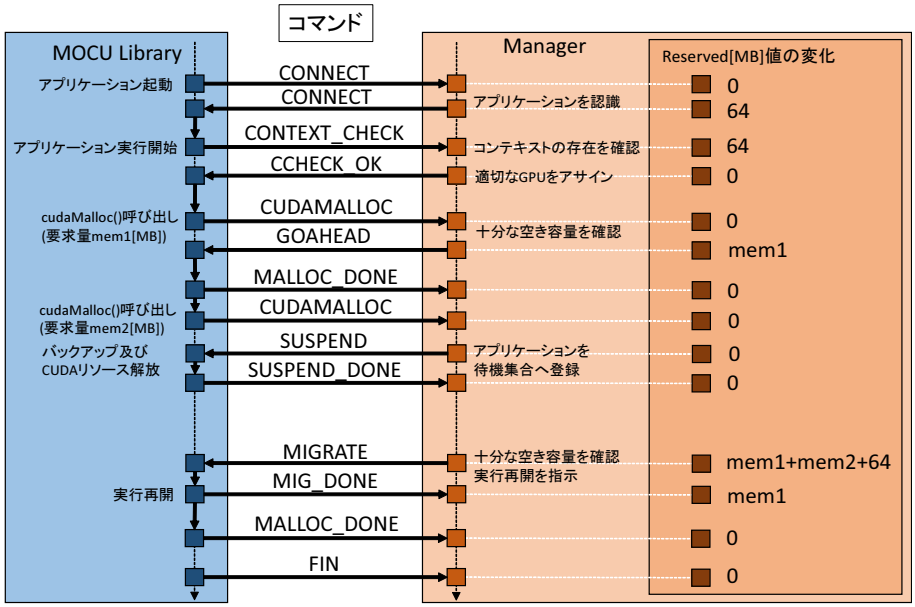

図 2　Mobile CUDA における MOCU Library と Manager 間の通信

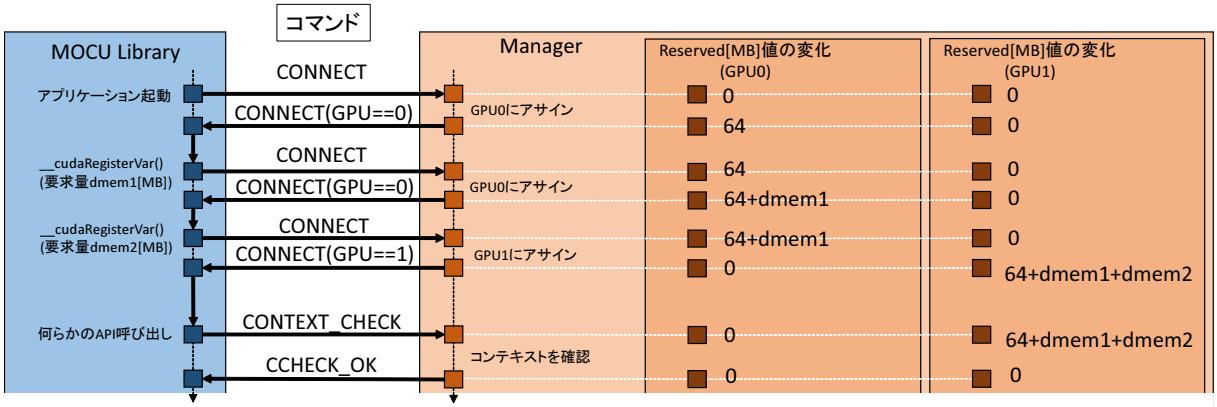

図 3　デバイス変数を使用する際の通信

2.3.3　Mobile CUDA の処理

Mobile CUDA を使用したアプリケーションの処理の流れと reserved 値の変化の例を図 2 で示す．このアプリケーションはデバイスメモリを 2 領域 (mem1[MB] と mem2[MB]) 確保する．__cudaRegisterFatBinary() 直前に MOCU Library から CONNECT コマンドが送信され，Manager はアプリケーションを適切な GPU へアサインするとともに，コンテキスト分を reserved 値へ加算する．この reserved 値は CONTEXT_CHECK コマンドでコンテキストの存在が確認されたら解放される．なお，CONTEXT_CHECK コマンドは全ての Runtime API 呼び出し後に行われるが，CCHECK_OK コマンドによりコンテキストの存在が確認されたら二度と送信されない．

図 2 の例では 1 度目の cudaMalloc() に成功し，2 度目の cudaMalloc() では十分な空き容量が確認できず，一度ホストへ退避する．十分な空き容量が確認された後，2 度目の cudaMalloc() を行い実行を再開する．

2.3.4　デバイス変数を利用する場合の処理

静的に宣言されたデバイス変数はデバイスメモリ要求量として加算する必要がある．その要求量は__cudaRegisterVar() が呼び出される際に取得することができる．__cudaRegisterVar() が呼ばれる時点ではコンテキストは作成されない．MOCU Library は__cudaRegisterVar() の呼び出し時に再び CONNECT コマンドを送ることで，コンテキスト作成時に確保されるデバイスメモリ要求量以上の空き容量を持つ GPU へのアサインを要求する．

図 3 はデバイス変数を 2 領域 (dmem1[MB] と dmem2[MB]) 使用したアプリケーションの起動時の例である．Manager は起動時の CONNECT，及び 1 つめのデバイス変数宣言時にはアプリケーションを GPU0 へアサインしている．しかし，GPU0 には 2 つ目の領域を確保するのに十分な空き容量がないことが確認され，アサインが GPU1 へと変更している．

表 5　実験に使用したアプリケーション

アプリケーション	実行時間 [sec]	サイズ [GB]	Mobile CUDA のオーバーヘッド [%]	説明
matrixMul	73.3	2.0	0.37	CUDA SDK に内包されている．計算律速．
matrixMulSmall	59.4	1.5	0.41	CUDA SDK に内包されている．計算律速．
matrixMulLarge	56.8	2.5	0.06	CUDA SDK に内包されている．計算律速．
pcie	54.3	2.0	0.49	2 本のベクトル加算．PCI-e 通信律速．
pcieSmall	72.0	1.5	0.11	2 本のベクトル加算．PCI-e 通信律速．
pcieLarge	50.7	2.4	0.18	2 本のベクトル加算．PCI-e 通信律速．
bandWidth	45.7	2.0	0.00	2 本のベクトル加算．GPU メモリバンド幅律速．
bandWidthSmall	47.0	1.5	0.42	2 本のベクトル加算．GPU メモリバンド幅律速．
bandWidthLarge	52.8	2.4	0.23	2 本のベクトル加算．GPU メモリバンド幅律速．
malloc_in_kernel	30.7	2.4	0.06	2 本のベクトル加算．カーネル内で malloc() を呼び出す．Exclusive Mode で実行
devmem	61.0	2.0	0.07	2 本のベクトル加算．デバイス変数にて 2 本のベクトルを確保．
map	46.4	2.0	0.11	2 本のベクトル加算．pinned メモリを GPU へマップさせる．

2.3.5　特殊な動作モード

CUDA ではカーネル内で malloc() 関数や new キーワードを用いて動的にデバイスメモリを確保することができる．これらのデバイスメモリ確保は追跡することができず，また実行中断や再開を行うことができない．

そこで Mobile CUDA ではこのようなアプリケーションを排他モード (Exclusive Mode) で実行し，1 個の GPU を占有させる．MOCU Library は__cudaRegisterFatBinary() が呼び出された際に CUDA Fat Binary を解析し，malloc シンボルが発見された場合にはそのアプリケーションを Exclusive Mode で実行するように Manager に通知する．Manager は，同じ GPU が割り当てられている他のアプリケーションを全て退避させる．退避対象のアプリケーションは Manager からシグナルを受信する．シグナル受信後の CUDA API 呼び出し時に QUERY コマンドを送信し，SUSPEND コマンドを受信するためホストに退避する．MOCU Library は全ての API 呼び出し直前に Manager からシグナルで要求が来ていないかを確認している．Exclusive Mode のアプリケーションは 1GPU に 2 つ以上アサインすることはできない．そこで，GPU 数以上の Exclusive Mode のアプリケーションが投入されたら，コンテキストが作成される前に実行を一時停止させる．いずれかの GPU で Exclusive Mode のアプリケーションが終了したら，その GPU で実行を再開する．

2.3.6　スケジューリングの制限

Manager は CONNECT コマンドによって初期 GPU をアサインするアプリケーション数に制限を設けている．許容アプリケーション数を超えたアプリケーションが投入されても，Manager は GPU をアサインしない．許容アプリケーション数は GPU × 4 に設定している．これによって，実行アプリケーション数過多な状況を避ける事ができる．GPU をアサインされなかったアプリケーションはコンテキストを作成していないため，バックアップやリソースの解放は必要ない．そこで Manager は CONNECT コマンドによって GPU アサインとアプリケーションの実行を再開させる．

コンテキストを作成していないアプリケーションが多数 GPU 上に存在すると，GPU を圧迫し処理を遅らせる要因になる．また，その後多くのアプリケーションがデバイスメモリ確保に失敗し，退避することが予想できる．そこで Manager はコンテキストを作成していないアプリケーションを 1GPU に 4 つまでしかアサインしない．

3.　性能評価

1 ノード上で Mobile CUDA の性能評価を行った．実験環境を表 4 に示す．なお，CUDA のバージョンは 5.5 を使用している．さらに実験に使用したアプリケーションを表 5 に示す．アプリケーションは異なる性質を持つものを使用した．malloc_in_kernel は malloc() をカーネル内で呼び出すアプリケーションで，Exclusive Mode で実行することになる．devmem はデバイス変数を使用するアプリケーションであり，map は pinned メモリをデバイスメモリにマップするアプリケーションである．

表 4　実験環境

	CPU	GPU
搭載数	2 ソケット	4
種類	Xeon E5-2687W	Tesla K20c
周波数	3.10 GHz	0.71 GHz
コア数	8	2496
メモリ	128 GB	4800 MB

アプリケーションを起動するためのスケジューラを 2 種類用意した．1 つ目は Mobile CUDA の使用を想定した MOCU スケジューラで，全てのアプリケーションを同時に起動する．2 つ目はシンプルスケジューラで，常に 1 デバイスで 1 アプリケーションが実行されているようになるようにアプリケーションを実行する．シンプルスケジューラの実行には NVIDIA の Library を使用しており，Mobile CUDA のオーバーヘッドはかからないようになっている．

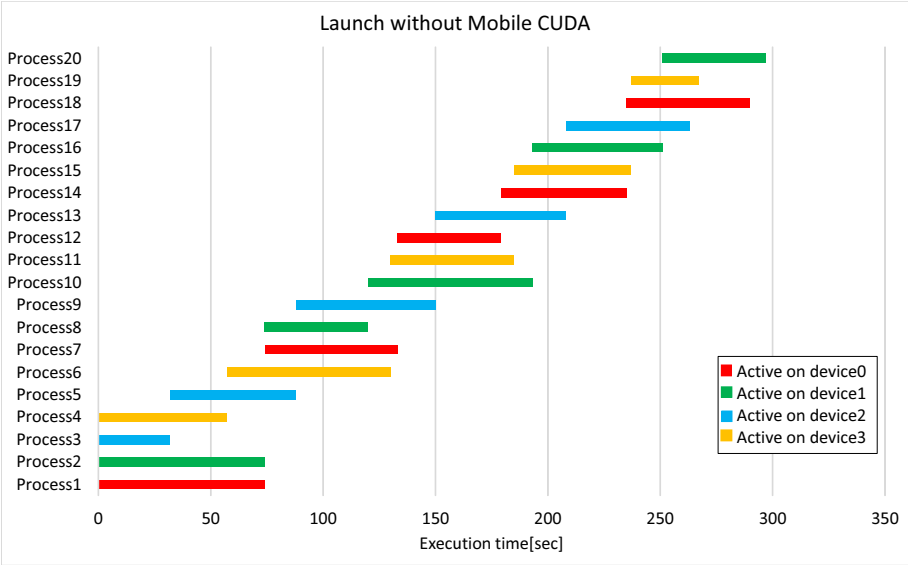

図 4　シンプルスケジューラで 20 アプリケーション実行した状況

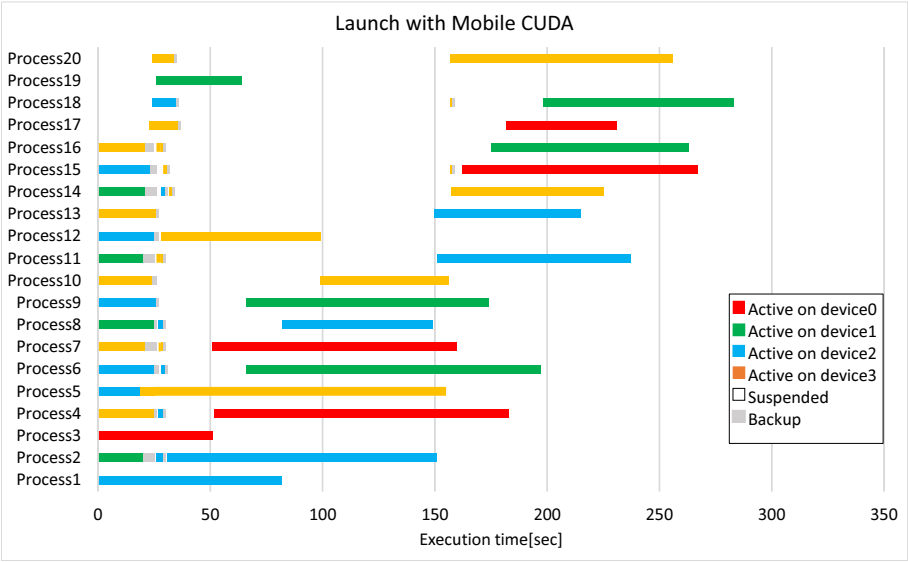

図 5　MOCU スケジューラで 20 アプリケーション実行した状況

実行するアプリケーションはランダムに選択されるが，MOCU スケジューラとシンプルスケジューラで実行するアプリケーションの順番は同じようになっている．

まず，2 つのスケジューラを用いて 20 アプリケーション実行した．シンプルスケジューラの挙動を図 4 で示す．1 つのアプリケーションが GPU を占有し，終了すると他のアプリケーションがその GPU で実行開始しているのが分かる．次に MOCU スケジューラの挙動を図 5 で示す．複数のアプリケーションが 1 デバイス上で実行されていることが分かる．Process3 が Exclusive Mode のアプリケーションで，それ以降に投入されたアプリケーションは GPU0 にアサインされていない．また，Process19 も Exclusive Mode のアプリケーションである．こちらは他のアプリケーションが退避するまで待機し，退避が完了したら実行を開始している．Process17 以降は許容アプリケーション数の制限で，初期 GPU をアサインされていない．図 4 と図 5 を比較すると，並列実行している分プロセスごと実行時間は MOCU スケジューラの方が増大しているが，合計実行時間は短縮していることが分かる．

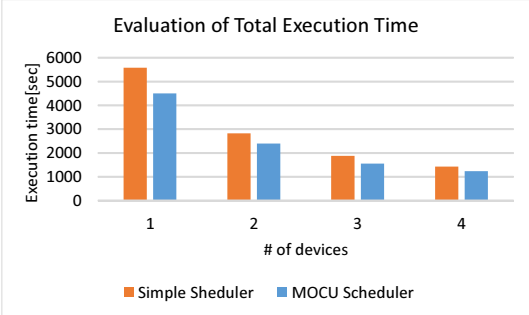

図 6　100 アプリケーションの合計実行時間

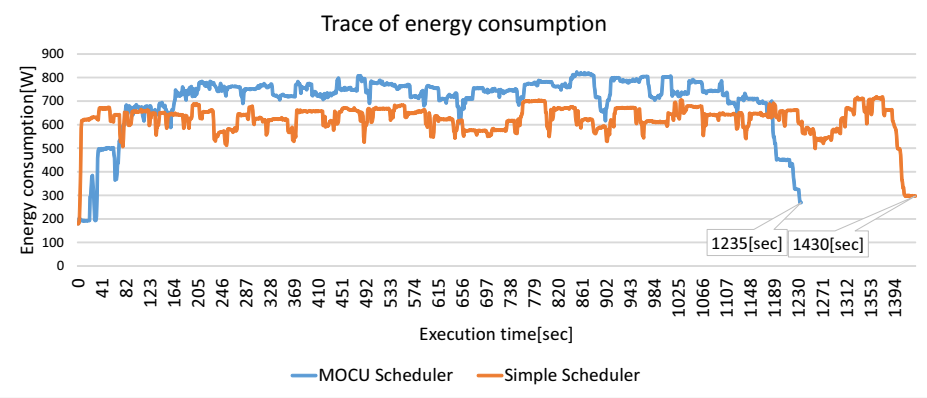

図 7　100 アプリケーション実行時の消費電力遷移図

次に 2 つのスケジューラで 100 アプリケーション実行した際の実行時間を測定した．測定の際には使用するデバイス数を 1 から 4 へ変化させた．実行時間を最も短縮したのはデバイス数 1 の時で 19.3%であった．全ての場合で，MOCU スケジューラの方が実行時間を短縮させた．

また，同時に消費電力及び消費エネルギーの測定も行った．デバイス数は 4 である．消費電力の遷移図を図 7 で示す．GPU のリソースを効率よく使用している分，平均消費電力は MOCU スケジューラの方が高くなっている．消費エネルギーは MOCU スケジューラが 871[MJ]，シンプルスケジューラが 891[MJ] となった．実行時間を短縮しているため，消費エネルギーは MOCU スケジューラの方がシンプルスケジューラに対して 2.2%削減している．

表 6　同時実行による実行時間の増加率

	matrixMul	pcie	bandwidth	devmem	map
matrixMul	1.94	1.17	1.24	1.22	5.01
pcie	1.63	1.88	1.04	1.03	2.57
bandwidth	3.97	1.62	2.03	1.98	3.00
devmem	3.78	1.54	1.96	2.07	10.72
map	1.10	1.35	1.33	1.02	1.60

表 6 は実験に使用した各アプリケーションを同時実行し，それぞれの実行時間にどのように影響しているかを示している．測定の際には，検証する 2 アプリケーションを同時実行し続け，実行時間の平均を単独での実行時間と比較した．例えば matrixMul と pcie を同時実行すると，matrixMul は単独で実行した時に比べ実行時間を 1．17 倍に増加させる．2 つのアプリケーション A と B を同時実行し，それらの実行時間を a と b とした時，(1/a)+(1/b)¿1 であれば実行時間を削減していると言える．bandwidth 同士と devmem 同士では競合により実行効率が低下している．

4. 関連研究

浜野ら [5] は複数の CPU アプリケーションと GPU アプリケーションを実行した場合のアプリケーション間の競合による性能低下予測モデルの構築，及びモデルに基づいたタスクスケジューリング手法を提案している．浜野らの手法では，まず各アプリケーションの PCI-e 通信量，及びメモリアクセス量から性能低下を予測する．競合を起こしやすいアプリケーションの同時実行を避け，CPU リソースと GPU リソースの両者を効率的に使用することで，システム全体のエネルギー効率の向上と実行時間の短縮を達成している．なお，アプリケーションの情報は既知とされている．しかしながら，実環境ではアプリケーションの情報が既知である場合は極めて少ない．また，浜野らは GPU アプリケーション同士の干渉ではなく，GPU アプリケーションと CPU アプリケーションの干渉を対象としている．我々はアプリケーションの情報は既知とせず，複数のアプリケーションを GPU 上で同時実行し，Hyper-Q によって実行時間の短縮，消費エネルギーの削減を目指している．

また vCUDA[6] や rCUDA[7] といった GPU の仮想化に取り組んでいるものもある．GPU の仮想化によって複数 CUDA アプリケーションが GPU を共有する場合も考えられるが，いずれもデバイスメモリ不足の問題は想定されていない．

MOCU Library は NVCR の手法に従いリソースの保存，再構築を実現していたが，他にも CheCUDA[8] は NVCR とは別の手法で CUDA アプリケーションに対してチェックポイント・リスタートを提供している．しかしながら，CheCUDA の実装は class ライブラリによって CUDA の Driver API が使用するリソースを override しており，使用の際にはアプリケーションをリコンパイルする必要がある．これは透過的な手法ではないため，Mobile CUDA には適さない．

5. まとめと今後の課題

Hyper-Q を利用して複数の CUDA アプリケーションの効率的な並列実行を行う Mobile CUDA を提案した．Mobile CUDA では各アプリケーションのデバイスメモリ使用量を集中管理することで，デバイスメモリ不足を回避する．デバイスメモリの空き容量が不足する場合にホストに

退避して，十分な空き容量が確認されたら実行を再開する．Mobile CUDA を使用した MOCU スケジューラと，使用しないシンプルスケジューラで 100 アプリケーション実行し，性能を評価した．MOCU スケジューラは 1GPU 環境の時に最も実行時間を削減し，19.3%であった．4GPU 環境では実行時間を 13.3%，消費エネルギーを 2.2%削減した．

今後の課題として，MOCU Library の Texture や Array などの構造体への対応が挙げられる．これにより，より多くのアプリケーションが Mobile CUDA 環境で実行可能になる．

謝辞

本研究の一部は科学研究費補助金基盤研究 (S)23220003「10 億並列・エクサスケールスーパーコンピュータの耐故障性基盤」及び科学技術振興機構戦略的創造研究推進事業「EBD:次世代の年ヨッタバイト処理に向けたエクストリームビッグデータの基盤技術」による．

参考文献

[1] Luebke, D., Harris, M., Govindaraju, N., Lefohn, A., Houston, M., Owens, J., Segal, M., Papakipos, M. and Buck, I.: GPGPU: general-purpose computation on graphics hardware, *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, ACM, p. 208 (2006).

[2] Lindholm, E., Nickolls, J., Oberman, S. and Montrym, J.: NVIDIA Tesla: A unified graphics and computing architecture, *Ieee Micro*, Vol. 28, No. 2, pp. 39–55 (2008).

[3] Hargrove, P. H. and Duell, J. C.: Berkeley lab checkpoint/restart (blcr) for linux clusters, *Journal of Physics: Conference Series*, Vol. 46, No. 1, IOP Publishing, p. 494 (2006).

[4] Nukada, A., Takizawa, H. and Matsuoka, S.: NVCR: A Transparent Checkpoint-Restart Library for NVIDIA CUDA, *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW)*, pp. 104–113 (online), DOI: 10.1109/IPDPS.2011.131 (2011).

[5] 浜野智明， 額田彰，遠藤敏夫， 松岡聡：GPU クラスタにおける省電力タスクスケジューリング，情報処理学会研究報告.[ハイパフォーマンスコンピューティング]，Vol. 2010, No. 17, pp. 1–9 (2010).

[6] Shi, L., Chen, H., Sun, J. and Li, K.: vCUDA: GPU-accelerated high-performance computing in virtual machines, *IEEE Transactions on Computers*, Vol. 61, No. 6, pp. 804–816 (2012).

[7] Duato, J., Pena, A. J., Silla, F., Mayo, R. and Quintana-Ortí, E. S.: rCUDA: Reducing the number of GPU-based accelerators in high performance clusters, *2010 International Conference on High Performance Computing and Simulation (HPCS)*, IEEE, pp. 224–231 (2010).

[8] Takizawa, H., Sato, K., Komatsu, K. and Kobayashi, H.: CheCUDA: A checkpoint/restart tool for CUDA applications, *2009 International Conference on Parallel and Distributed Computing, Applications and Technologies*, IEEE, pp. 408–413 (2009).