GPGPUアプリケーションのマイグレーションを可能にするフレームワーク

湯原 昌[1]　鈴木 勇介[1]　河野 健二[1]

概要：Graphics Processing Unit (GPU) は汎用計算 (GPGPU) に計算資源として用いられるようになっている．GPU の利用はサーバ向けアプリケーションにも広がり，クラウド環境における計算資源の一つとなっている．しかし，クラウド環境における負荷分散やメンテナンスといった運用・管理は，マイグレーションを用いることが一般的であるにも関わらず，GPU を利用したアプリケーションのマイグレーションは容易ではない．これは，GPU にコンテキストの取得や実行制御を容易に行うためのハードウェア機構が存在せず，軽量なマイグレーションの実現を妨げているからである．本研究では，クラウド環境におけるマイグレーションを可能とする GPGPU アプリケーションフレームワークを提案する．本フレームワークでは，イベント駆動型プログラミングモデルを採用することで GPU のコンテキストの取得と実行制御を行うためのソフトウェア機構を実現する．本フレームワークを用いたアプリケーションは別のマシンにマイグレーションできることを実験により示す．

キーワード：GPGPU，クラウド・コンピューティング，マイグレーション

1. はじめに

Graphics Processing Unit (GPU) はデータ並列性の高い処理を高速に実行できる手段として幅広いアプリケーションに利用されるようになった．グラフィックスや科学技術計算のアクセラレータに限らない汎用的な用途で GPU を利用するための技術は GPGPU として知られ，広く普及している．

特に GPGPU はクラウド環境における主要な計算資源一つとなっていて，サーバアプリケーションにも用途が広がっている．例としてはウェブサーバ [1]，ソフトウェアルータ [2]，SSL リバースプロキシ [3]，ファイルシステム [4] などが該当する．

サーバアプリケーションのような外部のリクエストに依存するようなワークロードで GPU の性能を活かしきるには，GPU をマルチテナント環境で利用ことが必要である．現状 GPU 上でのマルチタスクと資源のアイソレーションは実現する手法は存在している．しかしながら，GPU を用いたサーバアプリケーションをマイグレーションすることはいまだに困難である．マイグレーションによってクラウド事業者は，1) 全アプリケーションを別のマシンに移すことによって利用者に不便を強いることなくメンテナンスを行ったり，2) アプリケーションに対するリクエストが増えたときに負荷を分散することが可能になる．マイグレーションができない場合はアプリケーションの数を増やすのに慎重にならざるを得ず，GPU の性能を活かしきるのが難しくなってしまう．

GPGPU アプリケーションのマイグレーションを阻む最大の要因は GPU のハードウェア機構の不在である．実行中のプロセスをマイグレーションする場合，1) 対象のプロセスの実行制御を行う方法と，2) レジスタやメモリなどのコンテキストにアクセスする方法が必要である．しかし，こうした仕組みは GPU のハードウェアによって提供されていないため，GPU 上で動作しているコードを含めてマイグレーションを行うことはアプリケーションのサポート抜きには実現できない．

本論文ではサーバアプリケーション向けにマイグレーションを可能にするアプリケーションフレームワークを提案する．提案フレームワークではイベント駆動型プログラミングモデルをアプリケーションに提供する．これによって，GPU の実行制御とコンテキスト取得を行うためのソフトウェア機構を作り，GPU アプリケーションをマイグレーションが実現できることを示す．

提案フレームワークは GPGPU アプリケーション向けに協調的マルチタスクを実現するフレームワークである

1　慶應義塾大学
Keio University

ⓒ 2018 Information Processing Society of Japan

GLoop[5] をベースに実装した．予備実験の結果，0.8%のオーバヘッドでマイグレーションを実現した．

本論文の構成は，次のとおりである．第 2 章では，研究背景について説明する第 3 章では，提案フレームワークの設計について述べる．第 4 章では，提案フレームワークの実装について説明する．第 5 章では，提案フレームワークを用いて予備実験を行う．第 7 章では，本論文のまとめを述べる．

2. 背景

2.1 GPGPU アーキテクチャ

GPGPU アプリケーションの実行の流れは主に 3 段階に分けることができる．まず，CPU が GPU で処理したいデータを GPU のメモリに転送する．次に，CPU が GPU カーネル関数を起動し，GPU 側での処理が開始される．最後に，CPU が GPU のメモリから計算結果をコピーする．

GPU の構成を図 1 によって示す．カーネル関数は GPU に搭載されている多数の streaming multiprocessor (SM) によって実行される．SM はワープと呼ばれるスレッドの集合を SIMD 方式で実行することによって高い並列性を実現する．それぞれのスレッドには専用のレジスタが存在し，それぞれの SM には共有メモリと呼ばれるスクラッチパッドメモリが存在する．

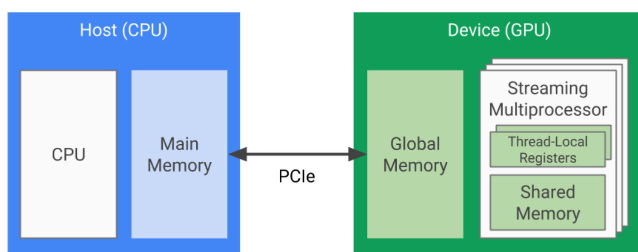

図 1 GPU の構成

マイグレーションを実現するためには，GPU 上で動いているコードを一旦停止する必要がある．しかしながら，GPU にはソフトウェアによって制御可能なプリエンプション機構が存在しない．GPU に送ったコマンドは完了まで動き続けるため，GPU でカーネル関数が実行されている場合は実行が終了するまでマイグレーションを始めることはできない．さらに，サーバアプリケーションのようにファイル I/O やネットワーク I/O を行う GPGPU アプリケーションでは GPU カーネルを起動するオーバヘッドを避けるために GPU からポーリングを行うケースがあり [6] [7]，このような場合にはアプリケーションが終了するまでマイグレーションを開始できる機会が存在しない．

また，GPU 側の実行コンテキストを CPU 側からアクセスするための機構もマイグレーションの実現には不可欠である．しかし，CPU 側からアクセスできる GPU の実行コンテキストはグローバルメモリに限り，SM のスクラッチパッドメモリやレジスタにはアクセスできない．したがって，例えカーネル関数の実行を一時停止することができたとしてもその実行状態を GPU の外に移すことはできないため，マイグレーションは不可能である．また， GPU の SM が利用可能なレジスタやスクラッチパッドメモリは CPU に比べて非常に大きいため，ハードウェアの支援があったとしてもマイグレーションのコストは大きくなると予想される．

2.2 GLoop

GLoop とは協調的マルチタスクによって GPGPU アプリケーションをマルチテナント環境で利用するためのアプリケーションフレームワークである．GLoop はイベント駆動型プログラミングモデルにもとづいたランタイムを提供する．これによって，GLoop アプリケーションはファイル I/O やネットワーク I/O の待ち時間をスケジューリングポイントとして活用し，ソフトウェア的に実行制御を行う．

GLoop アプリケーションは多数のコールバックから成り立ち，I/O 処理のようなイベントに紐ついている．GLoop アプリケーションはファイル I/O やネットワーク I/O を CPU にリクエストする際にコールバックを登録する．GLoop ランタイムはリクエストされた処理が完了したら，登録されたコールバックを実行する．コールバックのスケジューリングは GLoop ランタイムによって管理されるため，マルチタスクが可能になる．

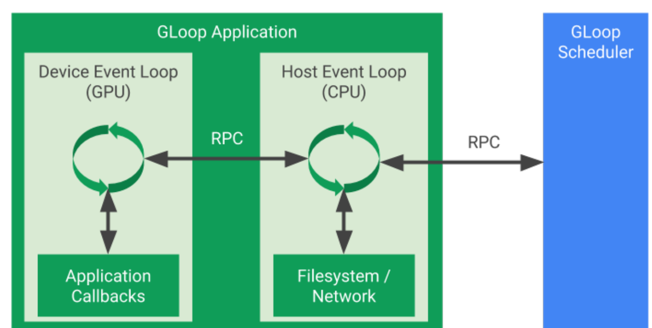

図 2 GLoop のアーキテクチャ

図 2 で GLoop のアーキテクチャを示す．GLoop アプリケーションは GPU 側で動くデバイスイベントループ (デバイスループ) と CPU 側で動くホストイベントループ (ホストループ) から成り立つ．ホストループはデバイスループを起動する際に RPC を介して GLoop スケジューラからスケジューリングトークンを獲得する．デバイスループは I/O 処理完了のようなイベントの発生を監視し，対応するコールバックを実行する．デバイスループはホストループと RPC を介して I/O 処理などのリクエストを行う．デバイスループはコールバックをディスパッチする段階でタイムスライスを使い果していた場合，スケジューリングトークンがまだ有効であるかをホストループを介して確認

する．もしスケージュリングトークンが無効になっていた場合はデバイスループは自ら実行を停止し，GPU の実行権を明け渡す．

3. 設計

3.1 目標

- **GPGPU サーバアプリケーションのマイグレーションを実現する：** カーネルを起動するコストを避けるために GPU 側でポーリングを行うようなアプリケーションは GPU を占有し続けるため，マイグレーションを行うことができなかった．このようなアプリケーションも提案フレームワークによってマイグレーション可能にする．
- **提案フレームワークのランタイムオーバヘッドを最小限に抑える：** 予備実験の結果，イベントループで生じたオーバヘッドは GLoop と比較して 0.8%であった．これは許容範囲の内であると考える．
- **プロプライエタリな GPGPU スタックを最小限の改変で活用可能にする：** 提案フレームワークは CUDA SDK 7.5 で動作する．マイグレーション後に CUDA ランタイムの再初期化を可能にするため，CUDA のランタイムライブラリのバイナリを数バイト改変する必要があった．GPU のドライバおよびハードウェアは無改変で利用することができる．

3.2 概要

提案フレームワークではイベント駆動型プログラミングモデルにもとづいたランタイムをアプリケーションに提供することで GPGPU アプリケーションをマイグレーションする際の問題点を解決する．通常の CUDA アプリケーションとイベント駆動型のランタイムを導入した場合の比較を表 1 に示す．

表 1 イベント駆動型ランタイムの有無による違い

マイグレーションの前提条件	イベント駆動なし	イベント駆動あり
GPU カーネルの停止	✗	✓
GPU 実行コンテキスト取り出し	✗	✓

通常の CUDA アプリケーションが GPU 上でポーリングを行う場合，GPU は占有され続けるため，マイグレーションのために GPU 上での実行を停止することはできない．GPU 上でイベント駆動型のランタイムを導入することで，ポーリングを行う時間はスケジューリングポイントとして利用可能になり，マイグレーションに向けて GPU での実行を停止することが可能になる．

また，CUDA アプリケーションは CPU からアクセスできないレジスタやメモリを操作するため，GPU の実行コンテキストを保存することはできない．これを解決するため，継続渡しスタイルでアプリケーションを記述することでアプリケーションの実行コンテキストをコールバックのクロージャとして収める．GPU カーネルの実行を停止する前に，ランタイムがコールバックのクロージャをグローバルメモリに保存することで GPU の実行コンテキストは全て CPU 側に移すことが可能になる．

3.3 手順

提案フレームワークでは以下の手順にもとづいてマイグレーションを実現する:

(1) **GPU カーネルを停止する：** これは GLoop のスケジューリングポイントを活用することで実現する．
(2) **実行中のホスト I/O が完了するのを待つ：** ホストループで実行される I/O リクエストの処理は GPU-CPU 間のメモリ転送を含む．GPU で利用しているリソースはマイグレーション前に全て解放する必要があるため，I/O 処理を行っている状態でマイグレーションを行うことはできない．したがって，マイグレーションを始める前に実行中の I/O 処理は全て完了させる必要がある．提案フレームワークではホストループとデバイスループ間の RPC に利用されるバッファサイズはコンパイル時に決まっているため，I/O リクエストの処理は必ず終了する．
(3) **GPU の実行コンテキストを CPU のメインメモリに転送：** この段階ではすでに GPU カーネルは実行されていないため，SM のスクラッチパッドメモリやレジスタの内容を保存する必要はなくなる．したがって，GPU の実行コンテキストは全て CPU からアクセス可能なグローバルメモリに存在し，GPU の実行コンテキストを取り出すことが可能になる．なお，GPU の実行コンテキスト取り出しが終わったら GPU で確保したリソースは全て解放する．
(4) **CPU 側のプロセスのマイグレーションを行う：** GPU の実行コンテキストは CPU 側に転送されているため，あとは既存のプロセスマイグレーション技術を利用することでマイグレーションは可能になる．
(5) **GPU の状態復帰：** CPU 側プロセスのマイグレーションが完了したら，GPU の状態を復帰させる．まず GPU のリソースを再確保した後，アプリケーションのコールバックを含んだデバイスループの管理情報とその他アプリケーションが利用していたグローバルメモリの内容を GPU に転送する．
(6) **GPU カーネルの起動：** あとは GPU カーネルを起動するだけでマイグレーションから復帰ができる．

4. 実装

提案フレームワークは CUDA SDK 7.5 を対象に実装し

ⓒ 2018 Information Processing Society of Japan

た．今回マイグレーションを実現する上での要となるイベント駆動型のランタイムは GLoop によって提供されている．したがって，提案フレームワークは GLoop ランタイムに変更を加えることでマイグレーションを実現する．

4.1 概要

提案フレームワークの概要を図 2 に示す．アプリケーションの CPU 側プロセスは Docker コンテナ上で動作させ，マイグレーションには CRIU[8] を用いた．GLoop ランタイムや CUDA ランタイムに依存するコードはアプリケーションの実行ファイルとは別のモジュールとしてビルドされ，dlopen() 関数によってアプリケーションプロセスのアドレス空間に動的に読み込まれる．GLoop スケジューラはコンテナの外で動作し，GLoop のランタイムと RPC を介して通信する．マイグレーションは GLoop スケジューラに対してリクエストを送ることで開始される．

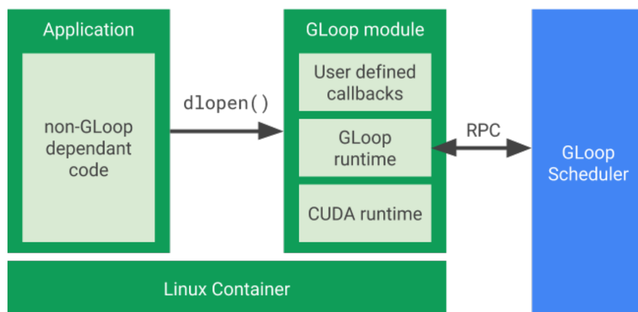

図 3 提案フレームワークのアーキテクチャ

4.2 CUDA ランタイムの再初期化

GPGPU アプリケーションをマイグレーションするためには，GPU のリソースをマイグレーション先で再確保する必要がある．提案フレームワークはプロプライエタリな CUDA の GPGPU スタック上で動作し，GPU のドライバを無改変で利用する．CUDA ランタイムは GPU のドライバと連携して GPU のリソース管理を行うが，CUDA の GPGPU スタックはマイグレーションを行う前提で設計されていないため，マイグレーション先であ CUDA ランタイムと GPU ドライバとの連携は失なわれてしまい，GPU リソースの獲得を行えなくなってしまう．

この問題に対処するため，マイグレーション元で CUDA ランタイムをアプリケーションプロセスからアンロードし，マイグレーション先でロードし直す．これを実現するため，CUDA ランタイム API に直接依存するコードは別モジュールに動的ライブラリとして実装した．GLoop ランタイムと GLoop のイベントループで動くアプリケーションコールバックも CUDA ランタイム API に直接依存するため，一緒のモジュールに含む．モジュールのロードとアンロードには dlopen() と dlclose () を利用する．

モジュールをアンロードする段階では GLoop ランタイムのイベントループは停止しているため，モジュールのアンロードが可能になる．マイグレーション先でモジュールを再ロードした際 CUDA ランタイムは新規プロセス同様に初期化されるため，GPU ドライバと再び連携することが可能になる．CUDA ランタイムのアンロードを可能にするため，CUDA ランタイムライブラリのバイナリを 9 バイト変更する必要があった．

4.3 GPU メモリの扱い

マイグレーション先では GPU のリソースを再確保する際，GPU メモリのポインタのようなハンドルはマイグレーション前と異なる値になってしまう．したがって，マイグレーション前に確保した GPU メモリをマイグレーション後にアクセスするようなアプリケーションコードは正しく動作しなくなってしまう．

マイグレーション後にも正しく GPU メモリにアクセスできるようにするため，GPU メモリの管理は GLoop ランタイムを介して行うようにした．GPU メモリにアクセスする際は GLoop ランタイムが提供するハンドルを介して行う．コールバック実行中にはマイグレーションは起きないため，ハンドルから実際のポインタを取り出す操作は 1 回のコールバック実行つきに 1 回で済む．したがって，メモリアクセスをするたびに間接参照をする必要はなく，オーバヘッドを抑えることができる．

5. 予備実験

今回の実験では 1) CUDA ランタイムの再初期化が可能であるか，および 2) マイグレーション対応によるオーバヘッドを測定した．実験環境を表 2 に示す．

表 2 実験環境

項目	詳細
GLoop	commit 8472d17
CUDA SDK	7.5
NVIDIA GPU ドライバ	384.130
Linux	4.4.0-128
GPU	NVIDIA Tesla K40C w/ 12GB GDDR5 メモリ
CPU	Intel Xeon CPU E5-2620 v3 @ 2.4 GHz
RAM	8GB

5.1 CUDA ランタイム再初期化

提案フレームワークを用いてアプリケーションがマイグレーション後にも GPU を利用できることを確認するため，実際に提案フレームワークを用いたアプリケーションで実験を行った．その結果，マイグレーション後もデバイ

ⓒ 2018 Information Processing Society of Japan

スループが動作することが確認できた．

5.2 ランタイムオーバヘッド

マイグーレション対応によって生じるアプリケーションのランタイムオーバヘッドを確認するため throttle ベンチマークを利用して実験を行った．throttle ベンチマークは空のコールバックを登録することでイベントループのオーバヘッドを測るベンチマークである．今回の実験では無改変の GLoop と提案フレームワークによる throttle の実行時間を比較することでマイグレーション対応によって生じたオーバヘッド計測した．実験結果は表 3 のようになった．提案フレームワークの GLoop と比べたランタイムオーバッドは 0.8%に留まっていることが分かる．

表 3 throttle の実行時間の比較

ランタイム	実行時間
GLoop	534.0 ± 0.2 秒
提案フレームワーク	538.2 ± 0.3 秒

6. 関連研究

6.1 仮想 GPU のマイグレーション

GPU を仮想化することによって，マイグレーションを実現する研究には LoGV[9] と gMig[10] がある．LoGV は NVIDIA の GPU を準仮想化する仕組みである．LoGV はマイグレーションを始める際，新たな GPU コマンドの受け付けを停止させる．これにより，新たなカーネルの実行は起きなくなる．GPU の実行が停止したら，GPU の状態とメモリがコピーされ，マイグレーションが実現する．gMig は Intel の内蔵 GPU を仮想化する技術をベースにマイグレーションを実現するシステムである．Intel の内蔵 GPU のグローバルメモリには CPU のメインメモリが利用されるため，プリコピー・ライブマイグレーションが可能になる．ただし，両方とも実行中のカーネルに関しては終了まで待たないといけないため，ポーリングを行うアプリケーションは対象外である．

6.2 プロセスチェックポイント

gHA は [11] は高可用性を目的として Intel の内蔵 GPU のチェックポイントを作成し，別のマシンで再現するシステムである．これは GPU のメモリの内容と実行したコマンドのコピーを定期的に複製することで実現する．CheCUDA[12] は NVIDIA の GPU を用いたアプリケーションのプロセスチェックポイントを作成するためのシステムである．CheCUDA はアプリケーション開発者がチェックポイントを挿入することすることで，GPU で取得したリソースのバックアップが取得されてプロセスチェックポイントが作成可能になる．我々が第 5 章と同様の環境で試したところ， CUDA ランタイムと GPU ドライバの連携が失われたことによって GPU の状態復帰を行うことができなかった．

7. まとめ

本論文では，サーバ向け GPGPU アプリケーションをマイグレーションするためにアプリケーションフレームワークを提案した．マイグレーションはクラウド環境の運用・管理に欠かせない技術であるが，GPU のハードウェアによる制約によって GPGPU アプリケーションのマイグレーションは実現困難であった．提案フレームワークではイベント駆動型プログラミングモデルにもとづいたランタイムを導入することで従来マイグレーションが困難であったアプリケーションもマイグレーションが可能になる．

提案フレームワークを GPGPU で協調的マルチタスクを行うためのイベント駆動型アプリケーションである GLoop をベースに実装したところ，ランタイムのオーバヘッドは 0.8%になった．

今後の課題として，ファイル I/O やネットワーク I/O を行うアプリケーションを対象にランタイムやマイグレーション時のオーバヘッドを評価する．

謝辞

本研究は，JST, CREST, JPMJCR1683 の支援を受けたものである．

参考文献

[1] Agrawal, S. R., Pistol, V., Pang, J., Tran, J., Tarjan, D. and Lebeck, A. R.: Rhythm: Harnessing Data Parallel Hardware for Server Workloads, *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, New York, NY, USA, ACM, pp. 19–34 (online), DOI: 10.1145/2541940.2541956 (2014).

[2] Han, S., Jang, K., Park, K. and Moon, S.: PacketShader: A GPU-accelerated Software Router, *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, New York, NY, USA, ACM, pp. 195–206 (online), DOI: 10.1145/1851182.1851207 (2010).

[3] Jang, K., Han, S., Han, S., Moon, S. and Park, K.: SSLShader: Cheap SSL Acceleration with Commodity Processors, *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, Berkeley, CA, USA, USENIX Association, pp. 1–14 (online), available from ⟨http://dl.acm.org/citation.cfm?id=1972457.1972459⟩ (2011).

[4] Sun, W., Ricci, R. and Curry, M. L.: GPUstore: Harnessing GPU Computing for Storage Systems in the OS Kernel, *Proceedings of the 5th Annual International Systems and Storage Conference*, SYSTOR '12, New York, NY, USA, ACM, pp. 9:1–9:12 (online), DOI: 10.1145/2367589.2367595 (2012).

[5] Suzuki, Y., Yamada, H., Kato, S. and Kono, K.: GLoop: An Event-driven Runtime for Consolidat-

ing GPGPU Applications, *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, New York, NY, USA, ACM, pp. 80–93 (online), DOI: 10.1145/3127479.3132023 (2017).

[6] Silberstein, M., Ford, B., Keidar, I. and Witchel, E.: GPUfs: Integrating a File System with GPUs, *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, New York, NY, USA, ACM, pp. 485–498 (online), DOI: 10.1145/2451116.2451169 (2013).

[7] Kim, S., Huh, S., Zhang, X., Hu, Y., Wated, A., Witchel, E. and Silberstein, M.: GPUnet: Networking Abstractions for GPU Programs, *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, Broomfield, CO, USENIX Association, pp. 201–216 (online), available from ⟨https://www.usenix.org/conference/osdi14/technical-sessions/presentation/kim⟩ (2014).

[8] OpenVZ team: CRIU, `https://criu.org/`.

[9] Gottschlag, M., Hillenbrand, M., Kehne, J., Stoess, J. and Bellosa, F.: LoGV: Low-Overhead GPGPU Virtualization, *2013 IEEE 10th International Conference on High Performance Computing and Communications 2013 IEEE International Conference on Embedded and Ubiquitous Computing*, pp. 1721–1726 (online), DOI: 10.1109/HPCC.and.EUC.2013.245 (2013).

[10] Ma, J., Zheng, X., Dong, Y., Li, W., Qi, Z., He, B. and Guan, H.: gMig: Efficient GPU Live Migration Optimized by Software Dirty Page for Full Virtualization, *Proceedings of the 14th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '18, New York, NY, USA, ACM, pp. 31–44 (online), DOI: 10.1145/3186411.3186414 (2018).

[11] Zhang, Z., Xu, X., Xue, M., Wang, J., Qi, Z. and Dong, Y.: gHA: An Efficient and Iterative Checkpointing Mechanism for Virtualized GPUs, *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems*, APSys '16, New York, NY, USA, ACM, pp. 1:1–1:8 (online), DOI: 10.1145/2967360.2967362 (2016).

[12] Takizawa, H., Sato, K., Komatsu, K. and Kobayashi, H.: CheCUDA: A Checkpoint/Restart Tool for CUDA Applications, *Proceedings of the 2009 International Conference on Parallel and Distributed Computing, Applications and Technologies*, PDCAT '09, Washington, DC, USA, IEEE Computer Society, pp. 408–413 (online), DOI: 10.1109/PDCAT.2009.78 (2009).

© 2018 Information Processing Society of Japan