

マイグレーション機能を備えた GPU 仮想化ツール DS-CUDA

成見 哲^{1,a)} 老川 稔^{2,†1,b)} Edgar Josafat Martinez-Noriega^{1,c)} 泰岡 顕治^{2,d)}

概要: DS-CUDA は CUDA のソースコードはそのままコンパイルするだけでネットワーク上の GPU をローカルマシンに装着しているかのように見せるミドルウェアである。コンシューマ向け GPU は信頼性が低いが、DS-CUDA の冗長計算機能により GPU が計算を間違えても自動的に再計算することが出来る。しかし、GPU クラスタを長時間使った場合にはネットワークやノードのマシンそのものの不安定さも耐故障性機能を実現するためには解決しなければならない問題である。本研究では DS-CUDA にマイグレーション機能を追加しオーバーヘッドを評価した。GPU サーバーがフリーズしても他の GPU に処理を移動することで計算を継続することが出来る。また、Dynamics Parallelism や GPU Direct RDMA に対応するための改造に関しても説明する。

キーワード: CUDA, 耐故障性, チェックポインティング, マイグレーション

DS-CUDA: GPU Virtualization Middleware to Support Migration Functionality

NARUMI TETSU^{1,a)} OIKAWA MINORU^{2,†1,b)} EDGAR JOSAFAT MARTINEZ-NORIEGA^{1,c)}
YASUOKA KENJI^{2,d)}

Abstract: DS-CUDA is a middleware to virtualize remote GPUs as if they look like local GPUs, by re-compiling CUDA source code with our compiler. Redundant calculation mechanism of DS-CUDA enables a fault-tolerant calculation with consumer GPUs, which are not usually so reliable. However, when large number of GPU nodes are used, the server nodes or network also should have fault-tolerance since large number of nodes cause more failure. In this research, we added the migration functionality to DS-CUDA, and evaluated the overhead of it. This function makes the user code continue to run even if server nodes of GPUs stop. Other implementations using Dynamics Parallelism and GPU Direct RDMA are also explained.

Keywords: CUDA, Fault tolerance, Checkpointing, Migration

1. はじめに

GPGPU (General-Purpose computing on Graphics Processing Units) が普及するにつれて、より高い性能を求めるために複数 GPU を用いることも多くなった。CUDA (Compute Unified Device Architecture)[1] を用いる場合は、ノード内に複数 GPU が存在していても、`cudaSetDevice` API を呼ぶだけで GPU を選択出来るため、ノード内 GPU に関してはそれ程プログラミングの敷居は高くない。しか

¹ 電気通信大学
The University of Electro-Communications, 1-5-1, Chofugaoka, Chofu, Tokyo 182-8585, Japan
² 慶應義塾大学
Keio University
^{†1} 現在、千葉大学
Presently with Chiba University
^{a)} narumi@cs.uec.ac.jp
^{b)} moikawa@chiba-u.jp
^{c)} m1541011@uec.ac.jp
^{d)} yasuka@mech.keio.ac.jp

し、同一ノード内に搭載できる GPU 数は 2~4 台程度であるため、数十台の GPU を使用したい場合そのためだけに MPI が用いられることもある。

DS-CUDA (Distributed-Shared CUDA) [2], [3] は、CUDA のソースコードをリコンパイルするだけでネットワーク経由の離れた GPU をあたかもノード内の GPU かのように見せることが出来る GPU 仮想化ミドルウェアである。MPI を使わなければ本来通信出来ないようなノードをまたがった GPU 同士の通信も、簡単に行うことが出来る。CUDA を仮想化する試みは rCUDA [4] など他にもあるが、耐故障性機能があることが DS-CUDA の特徴である。

計算機内で起こるエラーは、宇宙線の影響でメモリの特定のビットのみが反転するような一過性の“ソフトエラー”と、ハードウェアが故障してしまい復旧しない“ハードエラー”に分けることが出来る。ECC メモリを搭載する Tesla シリーズの GPU や、搭載しないコンシューマー向けの GeForce シリーズの GPU など多くの GPU においてどの程度のソフトエラーが発生しているか調べた研究 [5] によると、GPU のタイプに寄らず生じており、計算内容によってエラー率は大きく変わることが報告されている。GeForce シリーズでソフトウェア的に ECC 機能を実現する試み [6] もあるが、Tesla シリーズでもソフトエラーが起きることから十分ではないことが分かる。また、テストプログラムなどで事前にエラーが起りやすい GPU を除いたとしても、実際のユーザーアプリケーションでエラーが起きにくいとは限らず、また宇宙線などでたまたま起きることもあり得る。数千台以上の GPU を搭載するシステムが出てきているなど、大規模化する GPU クラスタにおいて、システム全体での MTBF (Mean Time Between Failure) の減少が問題となっている [7]。

DS-CUDA にはこれまで複数台の GPU で冗長に計算をさせることでソフトエラーを回避する「自動冗長計算機能」があった。今回はそれに加えて、GPU がハードエラーを起こした時や、GPU サーバーからネットワーク的に応答が無くなった時に、自動的に他の GPU に計算をマイグレーションする機能を追加した。定期的にチェックポイントングすることでオーバーヘッドは増えるが、クロックアップした GPU によるテストでは 10%程度のオーバーヘッドで正しい計算を続けられることが分かった。また、Dynamic Parallelism や GPU Direct RDMA と呼ばれる比較的新しい CUDA の機能を用いた改良についても述べる。

2. これまでの DS-CUDA

2.1 DS-CUDA の概要

DS-CUDA は、GPU を搭載しないクライアントノードと、GPU を搭載したサーバーノードで構成され、それらがネットワークで接続されているシステムを想定してい

る。DS-CUDA コンパイラにより、クライアントノード上で CUDA によるユーザーアプリケーションをコンパイルすると、サーバーノード上の GPU があたかもクライアントノード上の GPU であるかのようにアクセスすることが出来る。

例えば、`cudaDeviceCount(&ndev)` によりシステム上の GPU 数を得た際には、サーバーノード上の全 GPU 数が `ndev` に代入される。より正確には、ユーザーが環境変数 `DSCUDA_SERVER` によって指定したノード上の GPU 数の合計が返る。ユーザーアプリケーションでは、`0~ndev-1` の値を `GPU_id` にセットして `cudaSetDevice(GPU_id)` を呼ぶことによって、システム内の任意の GPU への操作を行うことが出来る。これはノード内にある GPU に対して通常の CUDA で書くのと同じであり、ユーザーは比較的簡単に多くの GPU を使うことが出来る。過去の例では、一つのプロセスから 1,024GPU を使用したこともある [3]。OpenMP による複数スレッドでの GPU の使用もサポートしている。

DS-CUDA の最大のデメリットはクライアントノードとサーバーノード間の通信が多いことである。多くの GPU を使う場合はレイテンシも問題となる。このため DS-CUDA では InfiniBand のローレベルの API を用い、一方向通信を利用することでレイテンシを重視して最適化している。表 1 は、Intel Core i7 920 のマザーボードと 40 Gbps の InfiniBand を用いた DS-CUDA の場合と、PCI Express 拡張 Box の Elsa Bridge を使った Native の CUDA の場合の `cudaMemcpy` API を用いたデータ転送の最大転送速度とレイテンシである。最大転送速度はかなり違いがあるもののレイテンシの差はそれ程大きくないことが分かる。実際他の同様の GPU 仮想化ソフトウェアの中でも低レイテンシである [8]。

2.2 自動冗長計算機能

DS-CUDA の特徴である耐故障性機能は、これまで冗長計算のみであった。まず、複数の GPU で同じ計算を行わせ、`cudaMemcpy(D2H)` によって GPU から CPU に計算結果を回収した際に複数の GPU からの結果を比較し、もし違う場合はそれまで呼び出した CUDA API を再度実行する。別の GPU で同時に同じソフトエラーを起こす確率は非常に低い事、同じ GPU で同一の計算を二度行って二回ともソフトエラーが発生する確率は非常に低い事から、この冗長計算機能により信頼性の高い計算を行うことが出来る。複数の GPU を使っていたとしてもソースコード上は仮想的に一つの GPU に見せており、信頼性の高い GPU を使っているように振る舞う。CUDA API の履歴には全ての引数の他に、配列の中身も覚えておく。履歴は GPU 同士の結果比較で合致した場合は忘れるので、履歴が溜まり過ぎるということはない。環境変数 `DSCUDA_AUTOVERB` を

表 1 cudaMemcpy のレイテンシと最大転送速度, およびカーネル呼び出しのレイテンシ
Table 1 Latency and maximum bandwidth of cudaMemcpy and latency of kernel call.

		DS-CUDA (InfiniBand)	CUDA (PCI Express 拡張 Box)
cudaMemcpy	最大転送速度	1.8 Gbyte/sec	5.0 Gbyte/sec
Host to Device	レイテンシ	9.4 μ sec	6.1 μ sec
cudaMemcpy	最大転送速度	1.3 Gbyte/sec	3.6 Gbyte/sec
Device to Host	レイテンシ	17 μ sec	10 μ sec
カーネル呼び出し	レイテンシ	100 μ sec	16 μ sec

定義すると使用出来る。

ただし, どんなアプリケーションでもこの方法でうまくいく訳では無い。例えば, GPU 上のメモリでずっと計算を行い, 最後に結果を回収するよう書いてあると, 計算エラーが起こった時に最初から再計算するので実用的ではない。計算途中で何らかの統計量を回収することで計算エラーを検出することが出来れば, この問題は解決する。また, 乱数のようにもう一度計算しようとするとう違う乱数が発生してしまい結果が変わることがあり得る。この場合は再計算で全く同じ乱数が使われるように設計する必要がある。

冗長計算でうまくいかない状況は他にもある。GPU にハードエラーが起きて何度再計算しても計算が一致しなくなったら, それ以上先には進めない。また, サーバーノードとの通信が途中で途切れてしまったら, 同様に先に進めない。

2.3 GPU 間 Peer to Peer 通信

CUDA では, 同じノード内であれば GPU 同士の通信を行うことが出来る。cudaMemcpy(GPU1_mem, GPU2_mem) のように GPU1 のメモリのポインタと GPU2 のメモリのポインタを書くだけで, 異なる GPU 間のメモリコピーを行うことが出来る。実際に GPU 同士で直接通信出来る場合は同じ I/O Chipset の下の GPU だけなど制約があるが, 見かけ上は GPU 同士で通信しているように書ける。

DS-CUDA でも同様に GPU 間通信を書くことが出来る。クライアントから見える全ての GPU のどのメモリであろうと, cudaMemcpy で GPU 同士で通信しているように見せている。仮に GPU1 がサーバーノード 1 に存在し, GPU2 がサーバーノード 2 に存在する場合, 下記のような流れになる。

- (1) クライアントノードがサーバーノード 1 に受信側であることを伝える。
- (2) クライアントノードがサーバーノード 2 に送信側であることを伝える。
- (3) サーバーノード 2 がサーバーノード 1 へ GPU2_mem からのデータを送信する
- (4) サーバーノード 1 がサーバーノード 2 からのデータを

GPU1_mem へ受信する

この時, (3), (4) のステップはクライアントを経由しないので, DS-CUDA の最大のデメリットであるクライアントノード・サーバーノード間の通信が激減する。ただし, 自動冗長計算機能は使用できなくなる。

3. 改良した DS-CUDA

現在の DS-CUDA は CUDA 7.0 に対応し GPL で公開している [9]。ここでは耐故障性機能や Dynamic Parallelism への対応など最近追加された機能について説明する。

3.1 チェックポインティング

計算途中のデータをどこかに保存しておいてエラー発生時にそこから再開する手法は, チェックポインティングと呼ばれアプリケーションの耐故障性を実現する上で基本的なものである [10]。今回 DS-CUDA にこの機能を付加した。

まずある時間 T_c が経過するごとに, CUDA API の履歴を保存するのは自動冗長計算機能と同様だが, それ以外に cudaMalloc により GPU 側で malloc したメモリ空間内の全情報をクライアントノードに保存する。より詳細には, cudaMemcpy(D2H) が呼ばれた時に前回のチェックポインティング時刻から T_c が経過していた場合に, 今回のチェックポインティングを行う。新しいチェックポインティングを行ったら前回保存した物は破棄する。 T_c は環境変数 DSCUDA_CP_PERIOD に設定する。

3.2 マイグレーション機能

マイグレーションを実現するためには以下のようなことを考える必要がある。

- ハードエラーにより故障した GPU 上にあったデータを後から回収出来るとは限らない。
- 新しい GPU を使って途中から同じ計算をやらせるには, 故障直前の GPU の状態が分かっている必要はない。

GPU の内部状態には様々な情報があるが, ここでは cudaMalloc したメモリ領域の情報だけを考える。ただし, cudaMemcpy(D2H) が終了した直後のタイミングに絞る。

何故なら, `cudaMemcpy(D2H)` の終了は通常カーネルが終了した後にしか生じ得ないので, GPU の内部状態として `cudaMalloc` した領域の情報だけを考えれば基本的によくなるからである.

以上のことを考えて, 冗長計算機能付きのマイグレーションは以下のように設計されている.

- (1) 割り込みとして, サーバーノードからのネットワークの反応がなくなったら (8) に飛ぶように設定しておく.
- (2) CUDA API の履歴を記録しながら `cudaMemcpy(D2H)` が呼ばれるまで待つ.
- (3) 前回のチェックポインティングの時刻から T_c が経過していなかったら (2) に戻る.
- (4) 冗長計算機能が有効になっている場合は, 複数の GPU でチェックポインティングするメモリ空間に対して内容を比較し, 異なっていれば (6) に飛ぶ.
- (5) チェックポインティングで保存するデータを更新し, (2) に戻る.
- (6) 最新のチェックポインティングのデータをそれぞれの GPU の `cudaMalloc` した領域にコピーする.
- (7) 前回チェックポインティングした時からの CUDA API の履歴に従って再度 API を実行し, (4) に戻る.
- (8) 環境変数 `DSCUDA_SERVER_SPARE` に定義されているサーバーノードの GPU を新しい GPU とする.
- (9) チェックポインティングした時点で `cudaMalloc` していた領域を新しい GPU で確保し, (7) に戻る.

3.3 Dynamic Parallelism のサポート

GPU カーネル関数の中から別のカーネル関数を呼び出せる機能は Dynamic Parallelism と呼ばれており再帰など不可欠な場面も多い. しかし親カーネルから子カーネルを呼び出す際は CPU から子カーネルを直接呼び出すよりも時間がかかることが分かっており, デメリットもあってそれ程使われていない. しかし DS-CUDA では大きなメリットがある. それはカーネル呼び出しのオーバーヘッドが減ることである. 表 1 にある通り, DS-CUDA ではネットワーク経由でサーバーノードの GPU に対してカーネルを呼び出すため, Native の CUDA に比べてレイテンシのオーバーヘッドがかなり大きい (約 6 倍). ネットワークを経由するのは CPU から呼び出されるカーネルだけで親から子カーネルが呼び出される部分は Native のままであるため, Dynamic Parallelism のサポートは DS-CUDA では重要な意味を持つ.

これまでの DS-CUDA では, クライアントノード側でソースのカーネル部分だけを取り出して `nvcc` コンパイラにより PTX 形式までコンパイルし, そのバイナリをサーバーノードに送って実行していた. サーバーノード側の DS-CUDA サーバー (サーバーノードで GPU 一つにつき一つ動いて

クライアントノードと通信するプログラム) はどんなユーザーのソースコードであれ共通で, CUDA のドライバ API を使用することで PTX ファイルを読み込んで実行すればよかった. しかしドライバ API では Dynamic Parallelism を使用できないことが分かった. `Nvcc` コンパイラで PTX ではなく直接 OS の実行ファイルを作れば問題ない.

そこで新しい DS-CUDA では設計を大きく変えて Dynamic Parallelism をサポートした. まず, ユーザーのソースコードから main 部分を入れ替えた上で `nvcc` でコンパイルする. 入れ替えた main 部分には DS-CUDA のサーバー機能が記述されており, ユーザーのカーネルと組み合わせられて DS-CUDA サーバーの実行ファイルが出来上がる. サーバーノードに DS-CUDA サーバーの実行ファイルを転送して実行することで DS-CUDA サーバーが起動する.

3.4 GPU Direct RDMA の使用

これまでの DS-CUDA の Peer to peer の GPU 間通信は, 下記のように 3 段階に分かれていた.

- (1) `cudaMemcpy(server2_mem, GPU2_mem)` により GPU2 の領域がサーバーノード 2 のメモリにコピーされる
- (2) InfiniBand の low level API を使ってサーバーノード 2 のメモリをサーバーノード 1 に送信する
- (3) InfiniBand の low level API を使ってサーバーノード 1 がメモリに受信する
- (4) `cudaMemcpy(GPU1_mem, server1_mem)` によりサーバーノード 1 のメモリから GPU1 のメモリにコピーされる

新しい DS-CUDA では GPU Direct RDMA 機能を使うことにより, (1) と (2), (3) と (4) を一つの API で実行出来る. ただし, GPU Direct RDMA を使うための詳細な情報が少なく, 動作はしたもののこれまでの方法より高速化はされなかった.

4. マイグレーションのオーバーヘッドの評価

DS-CUDA の耐故障性機能を実際に評価するために, オーバークロックした GPU によって起こしたエラーからリカバリ出来ることを確認し, その際に余分にかかった時間を測定した. 表 2 は検証に用いたハードウェアを示している.

4.1 GPU のオーバークロック

実際の GPU でのソフトエラーの頻度には大きなばらつきがあり, ソフトエラーからのリカバリーの検証は難しい. このため, GPU をオーバークロックすることでソフトエラーの頻度を人工的に上げ, 耐故障性機能の検証をやり易くした. ただしクロックを上げ過ぎると OS がフリーズしたりしてソフトエラーではなくなるので, たまにエラーが発生する程度のオーバークロックにする必要がある. X-Window の設定に追加して書くことで, `nvidia-settings` コマン

表 2 テスト環境
Table 2 Testing system

Client/Server node	CPU	Intel Core i7 920 2.7 GHz, 4 cores
	Memory	6 GiB
	GPU	NVIDIA GeForce GTX580 512 cores (8 multiprocessor) Core clock 1,544 MHz Memory clock 2,004 MHz
	OS	Linux 2.6.35.6
Interconnect	InfiniBand	32 Gbit/sec x4 QDR

ドでオーバークロック出来るようになる。

今回の GPU の場合、コアのクロックが 1,544 MHz、メモリのクロックが 2,004 MHz であった (表 2)。コアに関しては 12%、メモリに関しては 10%オーバークロックするのが限界であったため、それ以下の範囲で変化させることでエラーの頻度を調整した。

エラーの頻度は平均故障率 λ (平均故障間隔の逆数) で定義する。例えば $\lambda = 1/900$ の場合、900 秒に一回ソフトウェアエラーが発生するという頻度である。

4.2 アプリケーション

試した CUDA のソースコードは、レプリカ交換分子動力学シミュレーションを行うものである。分子動力学シミュレーションでは、原子のフェムト秒単位の動きを追う事で様々な条件下の現象を解析する。特にレプリカ交換分子動力学シミュレーションは、温度が異なる複数の分子動力学シミュレーション (一つ一つはレプリカと呼ぶ) を同時に動かし、レプリカ同士で安定な状態に移ったかどうかを判断しながら適宜温度を交換することで、最安定な状態を求めることが出来る手法である。特にレプリカ同士の通信量が少ないことから、DS-CUDA を用いた場合でもそれ程性能が落ちないというメリットがある。

今回は各レプリカが 256 個のアルゴン原子で構成され、16 個のレプリカを用いた。 $\epsilon = 0.9661(\text{kJ/mol})$, $\sigma = 0.3405\text{nm}$ の Lennard-Jones ポテンシャルを用い、 3σ で相互作用をカットオフした。それぞれのレプリカは 1.5×10^7 ステップ計算する。そのうち 1,000 ステップ毎に温度交換を行う。

4.3 オーバーヘッドの測定

今回の CUDA プログラムの実行時間 (sec) を以下の四つの場合について計測した。正常な GPU とオーバークロックした GPU を用意した。

- $T_0 = 5,189$: 正常な GPU を用いて Native な CUDA を用いて実行する
- $T_1 = 5,267$: 正常な GPU を一つだけ用いて DS-CUDA を使って実行するが、チェックポイントや冗長

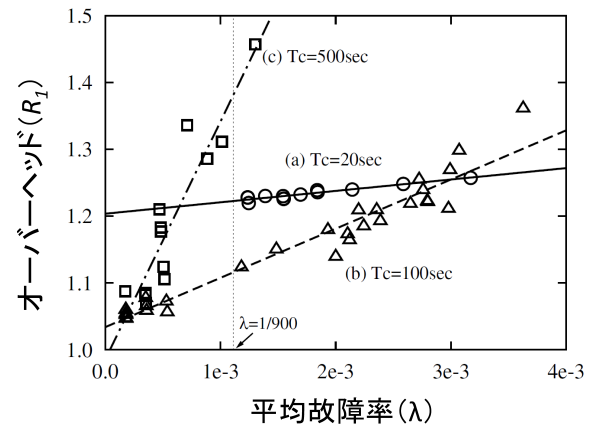


図 1 故障率を変えた時のオーバーヘッド

Fig. 1 Overhead against failure rate.

計算機能は無効にする

- T_2 : 正常な二つの GPU を用いて DS-CUDA を使って実行し、チェックポイントと冗長計算機能は有効にするが、ソフトウェアエラーは起きない
- T_3 : 正常な GPU とオーバークロックした GPU の二つを用いて DS-CUDA を使って実行し、チェックポイントと冗長計算機能を有効にしたうえで実際にソフトウェアエラーが起こる

四つのどの場合も計算結果は全く同じであったことから、DS-CUDA の耐故障性機能が有効に働いていることが分かる。

T_0 から T_1 の間では、DS-CUDA によって GPU を仮想化することでネットワーク上の GPU とやり取りを行う必要が発生し、その転送にかかるオーバーヘッドがある。 T_1 から T_2 の間では、二つの GPU を使うために両方の GPU と通信をすることで、チェックポイントのために GPU メモリの情報を保存すること、によってオーバーヘッドが発生する。 T_2 から T_3 の間では、ソフトウェアエラーが発生して前回のチェックポイント地点まで戻って再度計算を行うことによってオーバーヘッドが発生する。

オーバーヘッドの指標として $R_0 = T_3/T_0$ と $R_1 = T_3/T_1$ を定義する。 R_0 は Native な CUDA と比べてどのくらい実行時間が長くなるかを、 R_1 はソフトウェアエラーが起きることによりどのくらい実行時間が長くなるかを意味している。

4.4 故障率を変えた時のオーバーヘッド

図 1 は、平均故障率に対しどの程度オーバーヘッド (R_1) があつたかを示している。オーバークロックした GPU の平均故障率は $1 \times 10^{-4} \sim 3.7 \times 10^{-3}$ の範囲になっており、チェックポイント間隔 T_c は、20 秒、100 秒、500 秒で変化させた。それぞれの T_c に応じて最小二乗法で求めた直線も描画してある。

このグラフから、いずれの場合も故障率が高くなればオーバーヘッドが大きくなることが分かる。故障すればするほど再計算の可能性が高まるため、計算時間がより多く

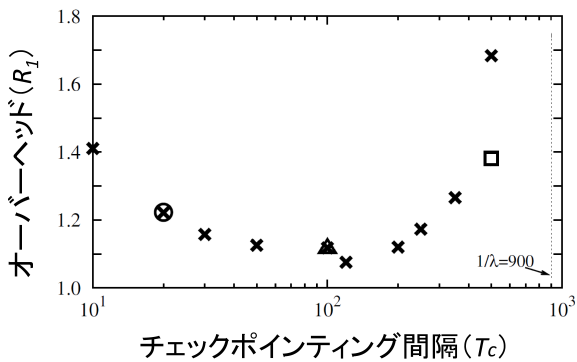


図 2 チェックポインティング間隔を変えた時のオーバーヘッド

Fig. 2 Overhead against checkpointing period.

かかる。更に、チェックポインティングの間隔が長いほど傾きが急になっている。これは、チェックポインティング間隔が長い場合、故障率が少し上がるだけでもその間にソフトウェアが起きる可能性が急激に高くなるからである。チェックポインティング間隔でソフトウェアが一度は起きる程度に故障率が高くなってしまうと、何度も再計算をするだけで永遠に先に進まなくなってしまう。

4.5 チェックポインティング間隔を変えた時のオーバーヘッド

図 2 は、チェックポインティング間隔 T_c を変えた時のオーバーヘッド R_1 を表示したものである。故障率が $1/900$ に近いものを図 1 から三点選んで $\circ$ 、 $\square$ で示している。 $\times$ は人工的に $\lambda = 1/900$ になるようにソフトウェアを発生させる DS-CUDA サーバーを使った時のものである。

故障率が一定の時、チェックポインティングの間隔を短くし過ぎるとチェックポインティングの保存作業そのものに時間がかかることでオーバーヘッドが大きくなる。一方チェックポインティングの間隔を長くし過ぎると、その間にソフトウェアが起きる可能性が高まり再計算をするロスによってオーバーヘッドが大きくなる。このため T_c は 10 倍程度の範囲はあるものの、ある程度の最適値にすることが望ましい。最適に近ければオーバーヘッドは 10% 程度であることから、実用的な性能を有していると判断出来る。

5. まとめ

本研究では GPU 仮想化ソフトウェア DS-CUDA にマイグレーション機能を追加し、そのオーバーヘッドを評価した。オーバークロックした GPU を用いて実際にソフトウェアからの回復が出来ることを示し、その時のオーバーヘッドが 10% 程度であった。

実際の GPU の故障率は今回試した数字よりもはるかに小さいものではあるが、大きな故障率でのオーバーヘッドが 10% 程度であるということは実際にはもっと少ないオーバーヘッドで済むはずである。ただし、チェックポインティング間隔が長くなると覚えなければならない CUDA API (及

びその引数) が膨大になるため、今回のような最適値には設定出来ない可能性がある。

また、今回の CUDA プログラムでは Dynamic Parallelism を使っていないが、別の実験で有効性を確認していることから、Dynamic Parallelism を使った場合のオーバーヘッドについても今後評価する必要がある。

謝辞 本研究の一部は、科学技術振興事業団 JST の戦略的基礎研究推進事業 CREST における研究領域「ポストペタスケール高性能計算に資するシステムソフトウェアの創出」の支援により行った。DS-CUDA の全般に関して主開発者である株式会社 K&F Computing Research の川井敦氏に、cudaMemcpy の測定データに関し瀬戸口幸寿氏に感謝します。

参考文献

- [1] CUDA C Programming Guide, 入手先 <http://docs.nvidia.com/cuda/cuda-c-programming-guide/> (2016.01.31).
- [2] Atsushi Kawai, Kenji Yasuoka, Kazuyuki Yoshikawa, and Tetsu Narumi: *Distributed-Shared CUDA: Virtualization of Large-Scale GPU Systems for Programmability and Reliability*, The Fourth International Conference on Future Computational Technologies and Applications, Nice, France (2012).
- [3] 老川 稔, 野村 昂太郎, 泰岡 顕治, 成見 哲: 1,024GPU を使用したレプリカ交換分子動力学シミュレーションの並列化, 情報処理学会 ACS 論文誌, 7/4, pp. 1-14 (2014).
- [4] rCUDA: 入手先 <http://www.rcuda.net/> (2016.01.31)
- [5] Imran S. Haque, Vijay S. Pande: *Hard Data on Soft Errors: A Large-Scale Assessment of Real-World Error Rate in GPGPU*, 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing, pp.691-696 (2010).
- [6] Naoya Maruyama, Akira Nukada, Satoshi Matsuoka: *A High-Performance Fault-Tolerant Software Framework for Memory on Commodity GPUs*, Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS), (2010).
- [7] P.Kogge, K.Bergman, S.Borkar, D.Campbell, W.Carlson, W.Dally, M.Denneau, P.Franzon, W.Harrod, K.Hill, J.Hiller, S.Karp, S.Keckler, D.Klein, R.Lucas, M.Richards, A.Scarpelli, S.Scott, A.Snively, T.Sterling, R.S.Williams and K.Yelick: *Exascale Computing Study: Technology challenges in achieving Exascale Systems, 2008*, 入手先 <http://www.cse.nd.edu>
- [8] Antonio J. Pena, Carlos Reano, Federico Silla, Rafael Mayo, Enrique S. Quintana-Orti, Jose Duato: *A Complete and Efficient CUDASharing Solution for HPC Clusters*, Parallel Computing, 40/10, pp. 574-588 (2014).
- [9] DS-CUDA: 入手先 <http://narumi.cs.uec.ac.jp/dscuda/> (2016.01.31)
- [10] John W.Young: *A First Order Approximation to the Optimum Checkpoint Interval*, Communications of the ACM, 17/9, pp.530-531 (1974).