Vol.2012-HPC-134 No.6
2012/6/1

GPU 仮想化による自動冗長計算システム

吉川 和幸[1,a]　川井 敦[2,b]　泰岡 顕治[2,c]　成見 哲[1,d]

概要：画像処理装置である GPU の高い計算能力を活用する技術である GPGPU が近年普及しつつあり，高性能な画像編集ソフトや動画エンコーディングソフトにも用いられるようになるなど，用途は一般化してきている．しかし，一般の人が使うコンシューマ向け GPU は演算性能的にはサーバー向け GPU に劣らないものの，ECC メモリを搭載しないなど信頼性の面で問題がある．そこで本研究では，GPU 仮想化技術である DS-CUDA を改良して，コンシューマ向け GPU による GPGPU の信頼性を向上させるシステムを開発する．具体的には，複数の GPU で同一の計算をさせる冗長計算を行い，その結果を比較することでいずれかの GPU で計算ミスが発生したことを検出する．計算ミスが発生したら，自動で再計算を行う機能も搭載する．既存のプログラムを変更することなく使用出来ることが最大のメリットである．

キーワード：GPGPU, CUDA, 仮想化, 冗長計算

Automatic Redundant Calculation with GPU Virtualization

Yoshikawa Kazuyuki[1,a)]　Atsushi Kawai[2,b)]　Kenji Yasuoka[2,c)]　Tetsu Narumi[1,d)]

Abstract: GPGPU is becoming more and more popular, and recently software for image manipulation or movie encoding also supports GPGPU. However, consumer GPUs for such applications are not so reliable, for example no ECC, even though the performance of them is superior to that of server GPUs. We modified DS-CUDA, which is a framework for GPU virtualization, to enhance the reliability of consumer GPUs. Our system performs redundant calculation with multiple GPUs and checks the difference of the results with each other. When error occurred, the system automatically repeats the former operations on the GPUs. The key is that we do not need any modification of the application software to get the benefit of the reliability.

Keywords: GPGPU, CUDA, Virtualization, Redundant calculation

1. はじめに

近年，画像処理装置である GPU を様々な用途に活用する技術である GPGPU (General-Purpose computing on Graphics Processing Units) が広がりを見せている．最初は科学技術計算に用いられていたが，近年ではフォトレタッチソフト [1] や動画エンコードソフト [2] 等の一般向けソフトウェアにも GPGPU が用いられるようになってきている．

しかし，現在のコンシューマ向け GPU（NVIDIA 社では GeForce シリーズ）はまれに計算ミスが発生することが知られている．例えば GeForce GTX295 421 枚のうち 1 割程度は計算ミスが発生したという報告がある [3]．画像出力用途ならば計算ミスが発生してもあるピクセルの色が意図したものと違うといった程度で済み，またその結果はすぐに次の計算結果によって置き換えられるため，計算ミスが表面化することは稀である．コンシューマ向け GPU は主にゲーミング用途であるため，こういった多少のミスは無視してでも演算性能を向上させることが重視されてきた．「シミュレーションをさせると計算ミスが発生する」と文

[1] 電気通信大学情報理工学研究科 Faculty of Informatics and Engineering, University of Electro-Communications
[2] 慶應義塾大学理工学部 Faculty of Science and Technology, Keio University
a) y1231102@sun.edu.cc.uec.ac.jp
b) kawai@kfcr.jp
c) yasuoka@mech.keio.ac.jp
d) narumi@cs.uec.ac.jp

© 2012 Information Processing Society of Japan

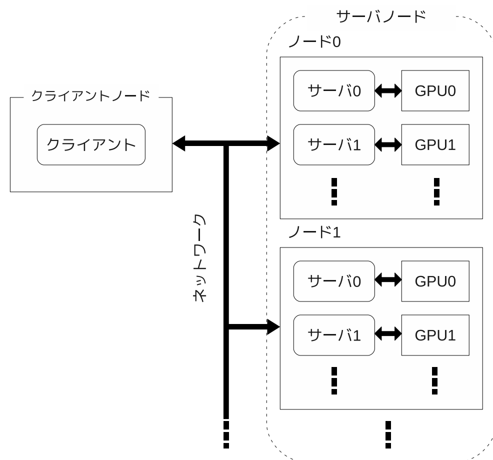

図 1 DS-CUDA の動作するシステムの全体像
Fig. 1 A system for DS-CUDA

句を言っても販売店が交換してくれることはあまり期待できない．

一方サーバー向け GPU（NVIDIA 社では Tesla シリーズ）は計算ミスに対する対応がなされている．特にメモリに対する ECC（Error Check and Correction）機能は，サーバー向け GPU にしかない機能である．シミュレーションでは直前の計算結果を次の計算に用いることが多く，一度計算ミスが発生してしまうとそこから先の計算全てが異常な結果になってしまう．しかしサーバー向け GPU は，コンシューマー向けの 4〜10 倍も高価で一枚あたりの演算性能も劣るため，価格性能比を追求するために GPGPU を使用したい場合などにはあまりメリットがなくなってしまう．

丸山らはソフトウェアで ECC を実装する手法を提案した [4]．GPGPU アプリケーション中に ECC を計算・検査するコードを追加することにより誤りを検出・訂正する．CUDA 向けのライブラリとして実装されているため，GeForce シリーズでも高い信頼性を保て，GPGPU におけるコストを抑えることが可能となる．しかし，メモリ以外のエラーに関しては信頼性が改善しないため，この手法だけでは完全とはいえない．

本論文では，コンシューマ向け GPU を使いつつ自動的に信頼性を向上するシステムを提案する．複数の GPU に同じ計算をさせることで計算エラーを検出し，自動的に再計算を行う．ユーザーからそれらの機能を隠ぺいさせるため，GPU 仮想化ソフトウェアである DS-CUDA [5] を利用する．ユーザーにはあたかも一枚の GPU のように見せておきながら，実際には複数の GPU を使うことになる．

DS-CUDA は，ユーザーアプリケーションのソースコードレベルで GPU を仮想化するフレームワークである．ネットワーク的に離れた別の PC の GPU をあたかもローカルに装着された GPU かのように見せることが出来る．このような仮想化を行うソフトウェアとして rCUDA [6], [7] が

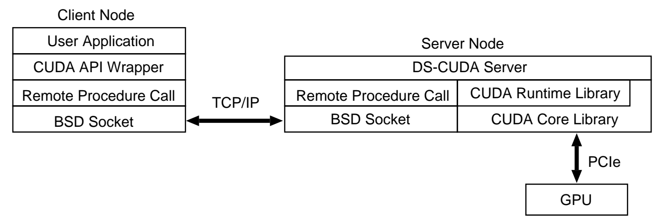

図 2 ソフトウェアのレイヤー
Fig. 2 Software layer stack

あるが，rCUDA は自動冗長計算機能は有していない．

以下では，第 2 章で DS-CUDA の概要を述べ，第 3 章で冗長計算機能の概要と性能を述べる．最後に第 4 章でまとめと今後の課題について述べる．

2. DS-CUDA

DS-CUDA は，GPU が複数の PC に分散されて搭載されているような一般的な GPU クラスタを，あたかも大きな一台の PC かのように見せることが出来るフレームワークである．ユーザーアプリケーションにはその大きな PC に全ての GPU が接続されているように見える．DS-CUDA は GPU 仮想化の一種ではあるが，OS レベルで仮想化している訳ではなく，ソースコードレベルの仮想化である．具体的には，NVIDIA の GPGPU 向けフレームワークである CUDA [8] の Runtime API の仮想的な実装をしているに過ぎない．このためリコンパイルが必要となる．

2.1 全体構成

図 1 は DS-CUDA を動かすハードウェアを示している．右側は複数のノードからなる GPU クラスタである．一つのノードには GPU がいくつか搭載されている．GPU それぞれに対して DS-CUDA のサーバーを起動させておく．左側はユーザーアプリケーションが動作するクライアント PC であり，ネットワークによって GPU クラスタと接続されている．クライアント PC には GPU は搭載されない．DS-CUDA を使うとクライアント PC 上に見かけ上全ての GPU が搭載されているように見える．

2.2 ソフトウェアレイヤー

図 2 は DS-CUDA のソフトウェアのレイヤーを示している．クライアント側で走るユーザーアプリケーションは，CUDA API のラッパー関数を呼び出す．すると DS-CUDA のライブラリが RPC (Remote Procedure Call) を通じて DS-CUDA のサーバーと通信する．サーバー側では対応する実際の CUDA API を呼び出す．つまり，ユーザーが CUDA API を呼び出すと，一旦ネットワークを経由して全ての引数がサーバー側に渡され，その引数を使って実 GPU に対して API が実行されることになる．GPU からの結果を回収する場合はその逆の経路を辿ってデータがクライア

© 2012 Information Processing Society of Japan

```
cudaGetDeviceCount(&n);
for(i=0; i<n; i++){
    cudaSetDevice(i);
    cudaMalloc(&d_data[i], size_d);
    cudaMalloc(&d_result[i], size_r);
    cudaMemcpy(d_data[i], h_data[i], size_d,
               cudaMemcpyHostToDevice);
    myKernel<<<g, b>>>(d_data[i], d_result[i], ...);
}
for(i=0; i<n; i++){
    cudaSetDevice(i);
    cudaMemcpy(h_result[i], d_result[i], size_r,
               cudaMemcpyDeviceToHost);
}
```

図 3 複数ボードを使う CUDA のコード例
Fig. 3 A sample code with CUDA to use multiple GPUs

ント側に戻されることになる．

2.3 コンパイラ

ユーザーアプリケーションの CUDA のソースに対しては, GPU 向けコンパイラである nvcc の代わりに, dscudacc と呼ぶ特別なコンパイラ（実体は Ruby スクリプト）を使用してコンパイルする．cudaMemcpy のような CUDA API に対しては，DS-CUDA ライブラリに同じ名前のラッパー関数が用意され，実際には NVIDIA のライブラリは呼び出されない．libcudart.so のような NVIDIA のライブラリの代わりに，別のライブラリをリンクすることになる．その後，DS-CUDA ライブラリが RPC を使ってサーバーとやりとりを行う．

しかし，単純な API の置き換えだけでは対応できないものがいくつかある．例えばカーネル呼び出しやコンスタントメモリの使用などである．GPU で実行されるカーネル関数は，CUDA 独自の記述方法で myKernel<<<g, b>>>(val, ...) のように書く．すると, dscudacc によって dscudamyKernel というラッパー関数が生成され，その関数を呼び出すことで引数などがサーバー側に通知される．また，別途 nvcc によってコンパイルされて生成された.ptx ファイルがサーバー側に送られる．サーバー側では，送られてきた.ptx ファイルと引数を使ってカーネルが呼び出される．

2.4 使用方法

図 3 は CUDA を使って複数の GPU を使用しているソースコードの例を示している．表 1 にはここで使っている CUDA API の概要を示す．GPU の台数分だけデータ (d_data) を GPU に送り，それぞれの GPU で myKernel のルーチンを実行し，計算結果（d_result）を GPU から回収する．myKernel を呼び出した時にはこのルーチンの終了を待たずに次のループに入るため，複数 GPU で同時に処理を行うことが出来る．

表 1 CUDA API の例
Table 1 Example of CUDA API

CUDA API	機能
cudaGetDeviceCount	PC に搭載している CUDA 対応 GPU の枚数を返す
cudaSetDevice	その後の CUDA API の対象となる GPU 番号を指定する
cudaMalloc	GPU 用メモリを確保する
cudaMemcpy	CPU ⇔ GPU メモリ間で転送する
myKernel<<<g, b>>>	GPU 用カーネルを実行する

一般に一台の PC に搭載できる GPU の数は，PCI Express のスロット数や電源容量などの制限で 2〜4 枚程度に限られる．このため，より多くの GPU を使いたい場合は MPI（Message Passing Interface）などで並列化する必要がある．しかし DS-CUDA を使った場合はクライアント PC に全ての GPU が搭載されているように見えるため，cudaGetDeviceCount で返される値はサーバーノードの全ての GPU の数になる．図 3 のソースコードはそのままで数十枚の GPU を使うことが可能になる．

より詳細に使い方を説明すると，クライアント側の環境変数 DSCUDA_SERVER に DS-CUDA サーバーの IP アドレスと GPU 番号を指定することで使用する GPU を切り替えることが出来る．例えば,

```
sh> export DSCUDA_SERVER= \
    "node0:0 node0:1,node1:0 node1:1"
```

のように書くと，node0 の GPU の 0 番が仮想的に GPU 0 番に，node0 の GPU の 1 番と node1 の GPU の 0 番は仮想的に GPU 1 番に，node1 の GPU の 1 番は仮想的に GPU 2 番に見える．ここで node0 の GPU の 1 番と node1 の GPU の 0 番が同じ仮想 GPU に見えているが，これは 2 枚の GPU を使って冗長計算を行うことを意味する．

3. 冗長計算

冗長計算機能は以下のように実装されている．まず複数の GPU で同じ計算を行わせ，その計算結果をチェックする必要がある (3.1 節)．次に，計算結果が異常であった場合に，以前正しい結果が得られた箇所まで計算を戻る処理が必要となる (3.2 節)．3.3 節では実際のアプリケーションを動作させた際の性能を示す．

3.1 複数 GPU での計算結果のチェック

DSCUDA_SERVER で指定される複数の GPU に対して全く同じ計算を行わせる．これは API ラッパーから複数の DS-CUDA サーバーに対し同一のデータを送ることで可能になる．cudaMemcpy が cudaMemcpyDeviceToHost で呼び出された場合，GPU から計算結果が返ってくる．その際

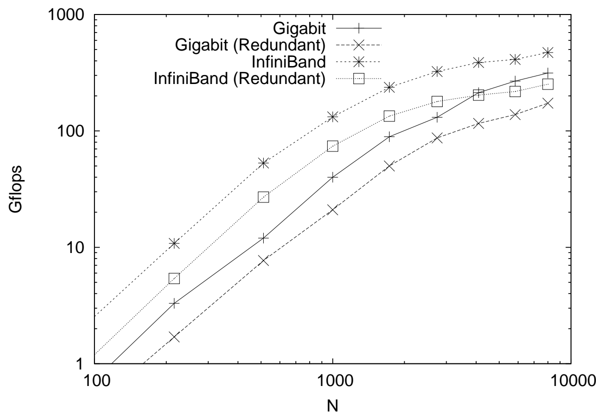

図 4 分子動力学シミュレーションの速度
Fig. 4 Calculation speed of the molecular dynamics simulation

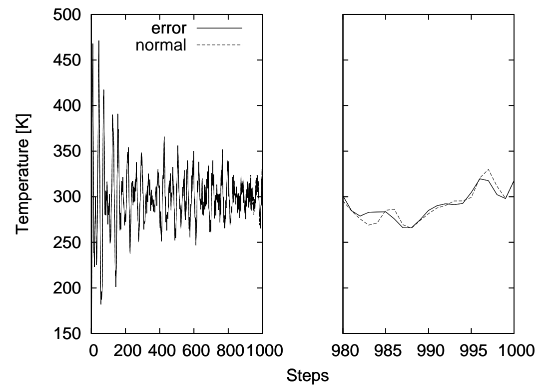

図 5 シミュレーション中の系の温度
Fig. 5 Temperature of a system during the simulation

は，複数の GPU での計算結果を比べ，もし 1 bit でも違っていたら cudaMemcpy がエラーを返す．もし自動冗長計算機能を有効にしている場合は，次の再計算処理に入る．

3.2 再計算処理

GPU での計算が間違える要因はいくつかありえる．GPU へのデータ転送が間違える場合，GPU のメモリ⇔プロセッサ間の読み書きで間違える場合，プロセッサが間違える場合などである．いずれにせよ同じ条件で再計算すれば，もう一度間違える可能性は低い．いくら信頼性の低いコンシューマ向け GPU であったとしても，毎回計算間違いする程のものはほとんどないからである．

そこで，再計算を可能にするために全ての CUDA API 呼び出しやカーネル呼び出しを覚えておく．このためにはラッパー関数内で呼び出しがあったことを記録すればよい．そして，cudaMemcpyDeviceToHost で呼び出された cudaMemcpy がエラーを返した場合，最初に戻ってもう一度全ての API 呼び出しやカーネル呼び出しを実行する．こうすれば途中のどこかで間違いがあったとしても正しく上書きされる可能性が高い．

計算間違いを見つける度に最初に戻ったのでは計算が進まないので，実際には最後に計算チェックが通ったところまでしか計算を戻さない．このため，たまに計算間違いがあっても再計算によって計算速度はほとんど低下することはない．ただし，例え途中の計算チェックを通っても GPU が間違った内部状態になっている可能性はあり，それがずっと後の計算チェックで発覚することもありえる．そのようなコードの場合は計算チェックが通ったところまで戻るだけでは正しい計算を行うことは出来ない．DS-CUDA ではそのような場合は想定しておらず，毎回必要なデータを GPU に送るようにアプリ側が書かれていると仮定している．以下で試す分子動力学シミュレーションなどでは通常毎回データを送るため，上記のような問題は発生しない．

3.3 性能

図 4 は NaCl の分子動力学シミュレーションを行った時の性能を示している．一つのペアの原子の相互作用計算には Tosi-Fumi ポテンシャル [9] を使用し，78 演算相当と換算して計算速度を算出している．横軸は系の原子の数である．クライアントとサーバー間のネットワークとしては，Gigabit ネットワークか InfiniBand ネットワークを使用した．使用した GPU は GeForce GTX 285 である．冗長計算を行う場合は 2 枚の GPU での冗長計算を行っている．

全ての場合において粒子数が多くなると計算速度は上昇する．これは粒子数が少ない場合，GPU との通信がボトルネックになるからである．このアプリケーションの GPU への実装では，原子間の力の計算だけが GPU で実行される．つまり，毎回原子の座標を GPU に送り，原子に働く力を GPU から返す必要がある．このため，粒子数が少ない場合には力の計算よりも座標や力の転送に時間がかかり，性能が出ない．DS-CUDA を使用した場合は GPU との通信は全てネットワーク上を経由する．このためネットワーク速度は非常に重要であり，InfiniBand を使用した方が性能が高いことが分かる．

冗長計算を使用した場合は性能が半分近くに低下している．GPU が 2 枚になるため通信が倍増するので性能が低下するのは仕方がないが，片方の GPU が計算中はもう片方の GPU が計算を行えるはずである．このため，より粒子数が多い場合などを試せば，性能低下がそれ程ない条件があると思われる．現在は詳細を調査中である．

図 5 はシミュレーション中の系の温度を示している．右側は後半部分の拡大図である．破線は GPU が計算間違いをしない場合，実線は計算間違いを起こすように作った DS-CUDA サーバーで計算した場合である．シミュレーション後半で温度の推移が異なることが分かる．計算間違いを起こす DS-CUDA サーバーでは，確率 0.1%で，cudaMemcpyDeviceToHost の cudaMemcpy の転送の先頭 1 バイトを 0 に上書きして返している．今回の例では 1000

回この cudaMemcpy が呼び出されているため，数回程度しかエラーは発生していない．しかし一旦エラーが発生すると，その後の原子の軌道が変わってくるため，全体の温度も変わってしまう．エラーが発生してからのシミュレーションが正しくなくなってしまうことが分かる．

DS-CUDA の自動冗長計算機能を使用した場合は，破線の間違えない温度の推移と完全に一致した．これは故意に間違える DS-CUDA サーバーからの間違った結果を検出して自動的に再計算を行い，正しく計算が続けられたことを示している．仮に長いシミュレーション期間中に本当に GPU が計算間違いを起こした場合でも，同様に正しく計算が続けられることが期待される．

4. まとめと今後の課題

コストパフォーマンスに優れるコンシューマ向け GPU を GPGPU に使用した場合に信頼性の低さが問題となりえるが，我々は GPU 仮想化フレームワークである DS-CUDA を改良することで信頼性を向上出来ることを示した．ユーザーアプリケーションを変更することなく自動冗長計算機能を実装することにより，GPU がたまに計算間違いを起こしても正しく計算を続けられることが分かった．ただし，現状では冗長計算機能を使用した際の性能低下が大きいため，ボトルネックの箇所の調査が必要である．今後は通信が途切れた時に別のサーバーに自動的に接続する機能など，より信頼性に優れる機能を実装していく予定である．

謝辞 本研究の一部は，電気通信大学事業化支援プログラム及び JST CREST 事業の一環として行われました．

参考文献

[1] Adobe Photoshop CS5 [Online]. Available: http://www.adobe.com/jp/products/photoshopfamily.html
[2] TMPGEnc [Online]. Available: http://www.tmpgenc.net/index.html
[3] T. Hamada, R. Yokota, K. Nitadori, T. Narumi, K. Yasuoka, M. Taiji, and K. Oguri, "42 TFlops Hierarchical N-body Simulations on GPUs with Applications in both Astrophysics and Turbulence" in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC) 2009*, USB memory (2009).
[4] 丸山直也, 額田彰, 松岡聡, "GPU 向けソフトウェア ECC の性能評価", 情報処理学会研究報告. 2009-HPC-119, pp. 25–30 (2009).
[5] Atsushi Kawai, Kenji Yasuoka, Kazuyuki Yoshikawa, and Tetsu Narumi, "Distributed-Shared CUDA : Virtualization of Large-Scale GPU Systems for Programmability and Reliability" in *The Fourth International Conference on Future Computational Technologies and Applications*, accepted.
[6] J. Duato, A. J. Pẽna, F. Silla, R. Mayo, and E. S. Quintana-Ortí, "Performance of CUDA Virtualized Remote GPUs in High Performance Clusters" in *2011 IEEE International Conference on Parallel Processing*, pp. 365–374 (2011).
[7] R. M. Gual, "rCUDA:an approach to provide remote access to GPU computational power" in *HPC Advisory Council European Workshop 2011*, [Online]. Available: http://www.hpcadvisorycouncil.com/events/2011/european_workshop/pdf/3_jaume.pdf
[8] The NVIDIA website. [Online]. Available: http://www.nvidia.com/object/cuda_home_new.html
[9] M. P. Tosi, and F. G. Fumi, "Ionic sizes and born repulsive parameters in the NaCl-type alkali halides–II: The generalized Huggins-Mayer form" *J. Phys. Chem. Solids*, vol. 25, pp 45–52 (1964).

© 2012 Information Processing Society of Japan